Java SE 8 Programming

Activity Guide

D84838GC10

Edition 1.0

August 2014

D87759

ORACLE®

Authors

Anjana Shenoy, Michael Williams, Tom McGinn, Peter Fernandez

Technical Contributors and Reviewers

Pete Daly, Sravanti Tatiraju, Nick Ristuccia, Stuart Marks, Hiroshi Hiraga, Peter Hall, Matthew Slingsby, Marcus Hirt, Irene Rusman, Joanne Sun, Marilyn Beck, Joe A Boulenouar

This book was published using: **Oracle** **Tutor**

Table of Contents

Practices for Lesson 1: Introduction

Chapter 1

Practices for Lesson 1: Introduction

Practices for Lesson 1: Overview

Practice Overview

In these practices, you explore the systems and tools that are used throughout the remaining practices.

Practice 1-1: Log In to Oracle Linux

Overview

In this practice, you log in to the Oracle Linux operating system.

Assumptions

Oracle Linux 6 is installed on your system, and it is on and functioning.

Tasks

1. At the login screen, enter the following information:

    User name: *oracle*

    Password: *oracle*

2. Click OK.

Root Access

Some of the utilities used in the practices require root system access. To obtain root access, enter the following in a terminal window:

su

When prompted for the password, enter:

oracle

Practice 1-2: Open Terminal Windows in Oracle Linux

Overview

In this practice, you open two terminal windows in Oracle Linux.

Assumptions

You are logged in to Oracle Linux, and you are running a Gnome Desktop.

Tasks

1. From the menu, select **Applications > System Tools > Terminal**.
    A terminal session should start.
2. Repeat step 1 to open another terminal window.
3. Alternatively, press **Ctrl + Shift + T** to open additional tabs in the same terminal window.

For Windows users: UNIX commands to use in your terminal window

```
DOS       UNIX           Description
dir       ll             list long (name, date, size, owner, etc)
          ll -latr       same as ll but sorted by date
dir/w     ls             list wide (no details)
dir/s     locate         find a file anywhere
del       rm             delete or remove files
copy      cp             copy file1 to file2
move      mv             move file1 to file2
ren       mv             rename file1 to file2
cd        pwd            print working directory
cd ..     cd ..          change directory UP one level
cd \      cd /           change directory to TOP level (root)
C-A-D     ps -ef         process statistics (often used with grep)
          top            dynamic list of top processes by percent
md        mkdir          make directory
rd        rmdir          remove directory
edit      vi             full-screen character-based editor (see below)
more      more           list a file and pause (space/enter to continue)
          tail -20 file1 list the last 20 lines of a file
type      cat            list a file and don't pause
          strings        same as cat but for files with binary chars
set       set            display all environment variables such as $HOME
help      man            manual (help) pages
find      grep           find a word in a line in a larger list of lines
prompt    PS1='$PWD >'   change the prompt to include current dir
logoff    su -           switch user (usually to Super User)
chkdsk    df -k          how much free space is left on disk
(n/a)     which file1    finds executables along paths
ver       uname -a       version of operating system software
```

- **Remember:** Everything in UNIX is case-sensitive.
- To change to a `ReallyLongDirectoryName`, just type `cd Rea*`.

Practices for Lesson 1: Introduction

Practice 1-3: Add the Java `bin` Directory to the Path

Overview

In this practice, you open a terminal window and add the Java directory to the $PATH in Oracle Linux.

Assumptions

You are logged in to Oracle Linux and you are running a Gnome Desktop.

Tasks

1. From the menu, select **Applications** > **System Tools** > **Terminal**.
2. A terminal session should start.
3. At the command prompt, type:

   ```
   gedit .bashrc
   ```

   Note: This loads the bash configuration file.
4. Add the following line to the end of the file:

   ```
   export PATH=/usr/java/jdk1.8.0/bin:/home/oracle/netbeans-
   8.0/bin:$PATH:.
   ```
5. Save the file.
6. Close gedit.
7. Close the terminal.

Practice 1-4: Start NetBeans and Open a Project

Overview

In this practice, you launch NetBeans and open a NetBeans project.

Assumptions

NetBeans is installed and functioning correctly. You are logged in to Oracle Linux and you are running Gnome Desktop.

Note

A new feature in NetBeans 8 is to store user name and password information in the Linux keyring. The first time you exit NetBeans, the following dialog will be displayed:

Enter "oracle" as the password for the keyring. Click **Create**.

The keyring for Linux should now be setup.

Tasks

1. Open a terminal window.
2. At the command prompt, enter:

    ```
    netbeans &
    ```

3. The first time you run NetBeans, you may be prompted to register the product:

4. Just click **Never Register** and continue.

   Note: The first time NetBeans runs, it caches and indexes a lot of information. Therefore, the initial load time might be a little slow. Subsequent launches of the application will be much faster.

   After it launches, NetBeans should look like this:
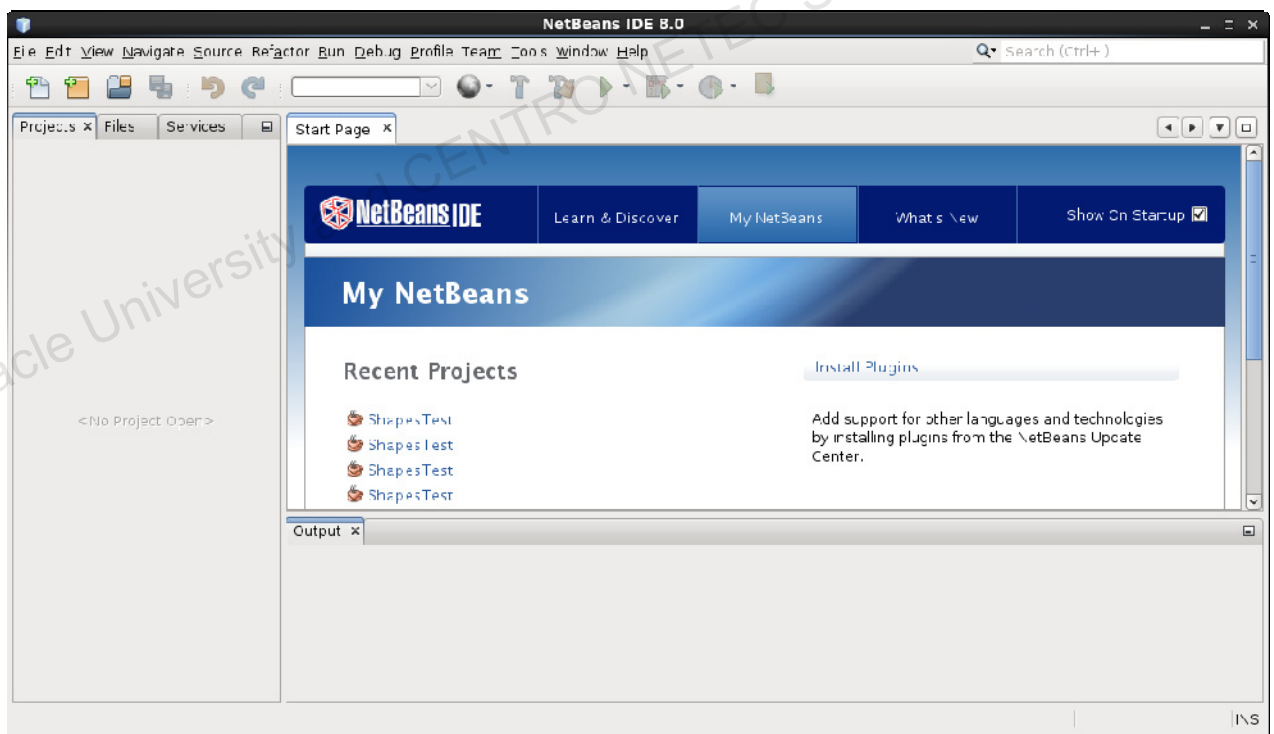
5. Open a sample NetBeans project by selecting **File** > **Open Project**.

6. Navigate to the `/home/oracle/labs/01-Intro/practices` directory.

7. Select the **Java2D** project and then click **Open Project**.

8. To run the project, right-click the project name and select **Run**.

9. Explore the user interface. Open some source files and other elements of the user interface.
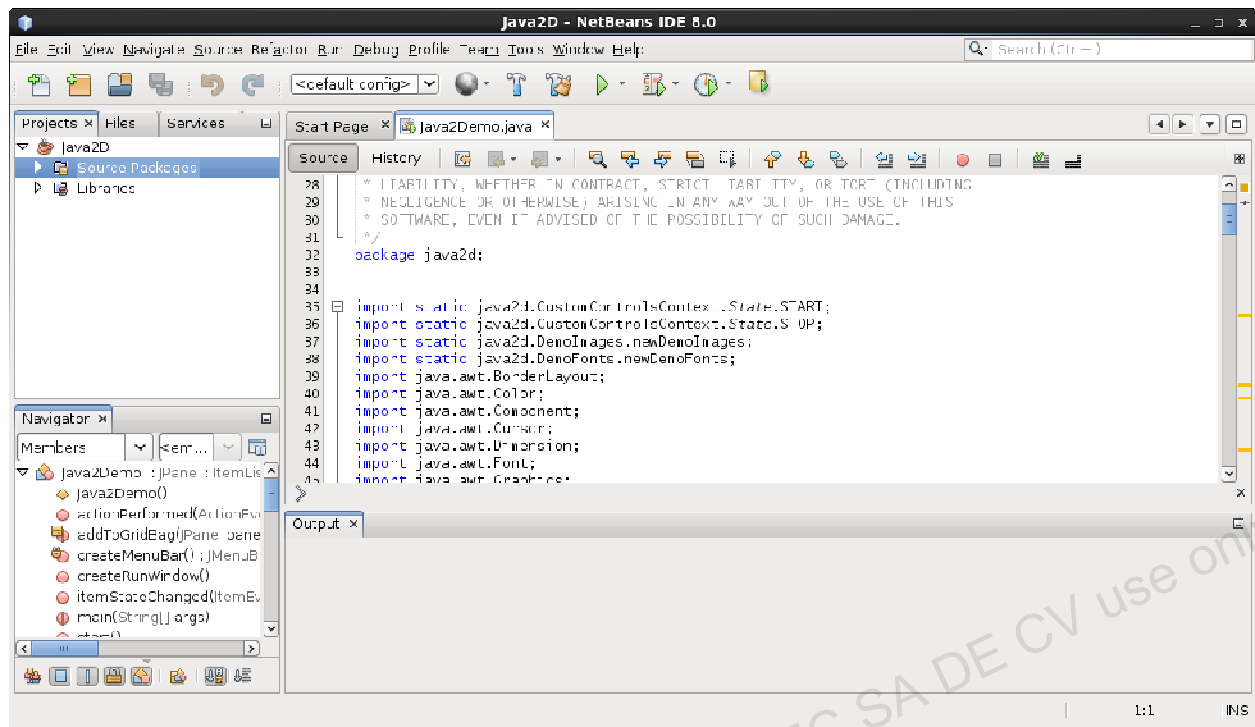
10. When you are done, right-click the project name and select **Close**.
11. Close NetBeans.

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. 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 `Employee` in the `/home/oracle/labs/02-Review/practices /practice1` directory with an `EmployeeTest` main class in the `com.example` package.

3. Set the Source/Binary format to JDK 8.

   a. Right-click the project and select Properties.

   b. Select JDK 8 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	`int`
Employee name	`String`
Employee Social Security Number	`String`
Employee salary	`double`

   b. Create a `no-arg` constructor for the `Employee` class.

   c. Add accessor/mutator methods for each of the fields.

   Note that NetBeans has a feature to create the getter and setter 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 and setters from the Generate menu, and click the boxes next to the fields for which you want getter and setter methods generated.

7. 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	012-34-5678
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.

8. Run the `Employee` project.

9. (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 `Employee` 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 window, perform the following steps:

Window/Page Description	Choices or Values
Project Name:	`Employee`
Project Location	`/home/oracle/labs/02-Review/practices/practice1`
Use Dedicated Folder for Storing Libraries	Ensure this is **not** selected.
Create Main Class	Ensure this is selected. Change the name to `com.example.EmployeeTest` `com.example` is the package name.

e. Click Finish.
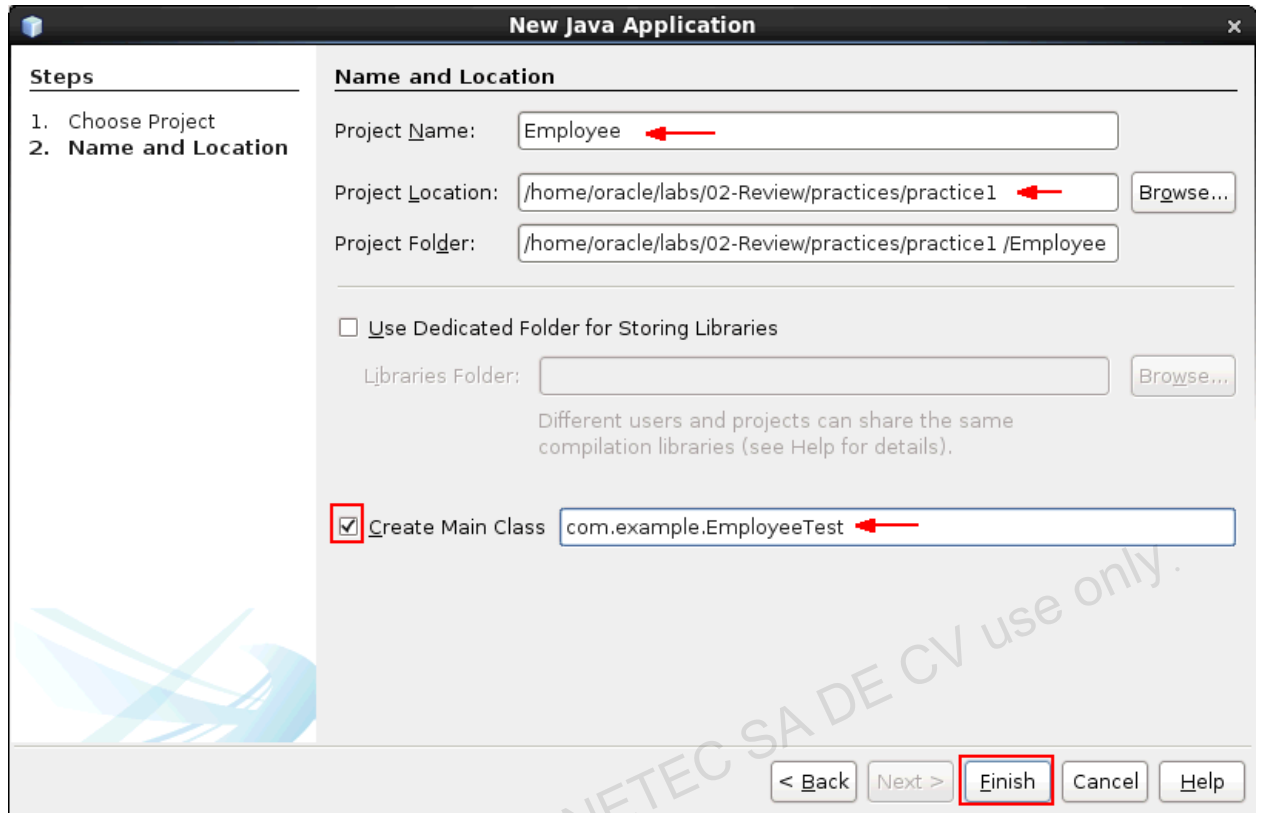
f. In the Projects tab, expand the `Employee` project, you will notice that NetBeans has created a class called `EmployeeTest`, including the package name of `com.example`, and skeleton of the `main` method is generated.
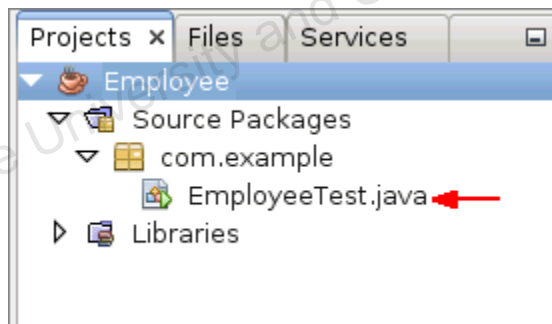
3. Set the Source/Binary format to JDK 8.

    a. Right-click the `Employee` project and select Properties.

    b. In the Project Properties window perform the following steps:

        1) Select JDK 8 from the drop-down list for Source/Binary Format.

2) Click OK.
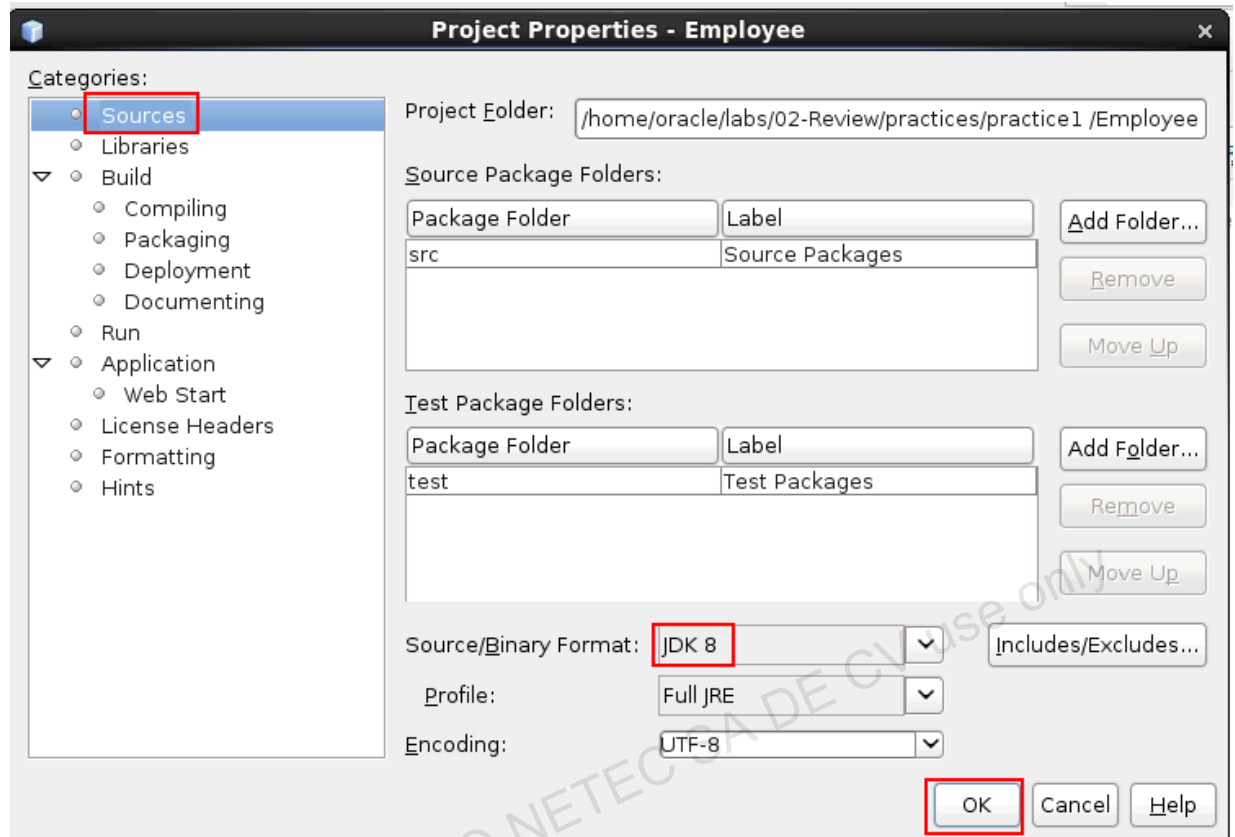
4. Create another package called `com.example.domain`.

    a. Right-click the current package `com.example` under `Source Packages`.

    b. Select **New > Java Package**.

    c. In the New Java Package window, perform the following steps:

       1) Enter `com.example.domain` in the Package Name field.

       2) 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`.

Note: You can format your code in NetNeans: right-click anywhere in the class and select Format, or press the Alt-Shift-F key combination.

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() { }
   ```
   c. Create accesor/mutator (getter/setter) methods for each of the fields.
      Note that NetBeans has a feature to create the getter and setter methods for you.
      1) Click in your class where you want the methods to go, then right-click and choose Insert Code (or press the Alt-Insert keys).
      2) Select "Getter and Setter" from the Generate menu.
      3) 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.
      4) Click Generate to insert the code.
   d. Save your class.
7. Modify 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:
   ```
   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.

   a. Right-click the project and select Properties.

   b. In the Project Properties window, perform the below steps:

      1) 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.

      2) 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.

      3) Click Cancel to close the Project Properties.

9. Run the `Employee` project.

   a. To run your `Employee` project, right-click the project and select Run. If your classes have no errors, your should see the following output in the Output window:
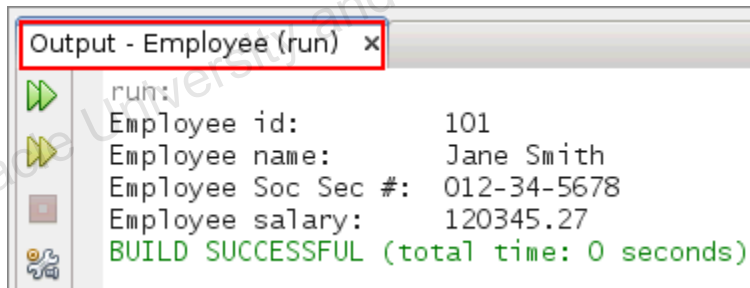
10. (Optional) Add some additional employee instances to your test class.

Practices for Lesson 2: Java Syntax and Class Review

Chapter 2 - Page 10

Practices for Lesson 3: Encapsulation and Subclassing

Chapter 3

Practices for Lesson 3: Encapsulation and Subclassing

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.
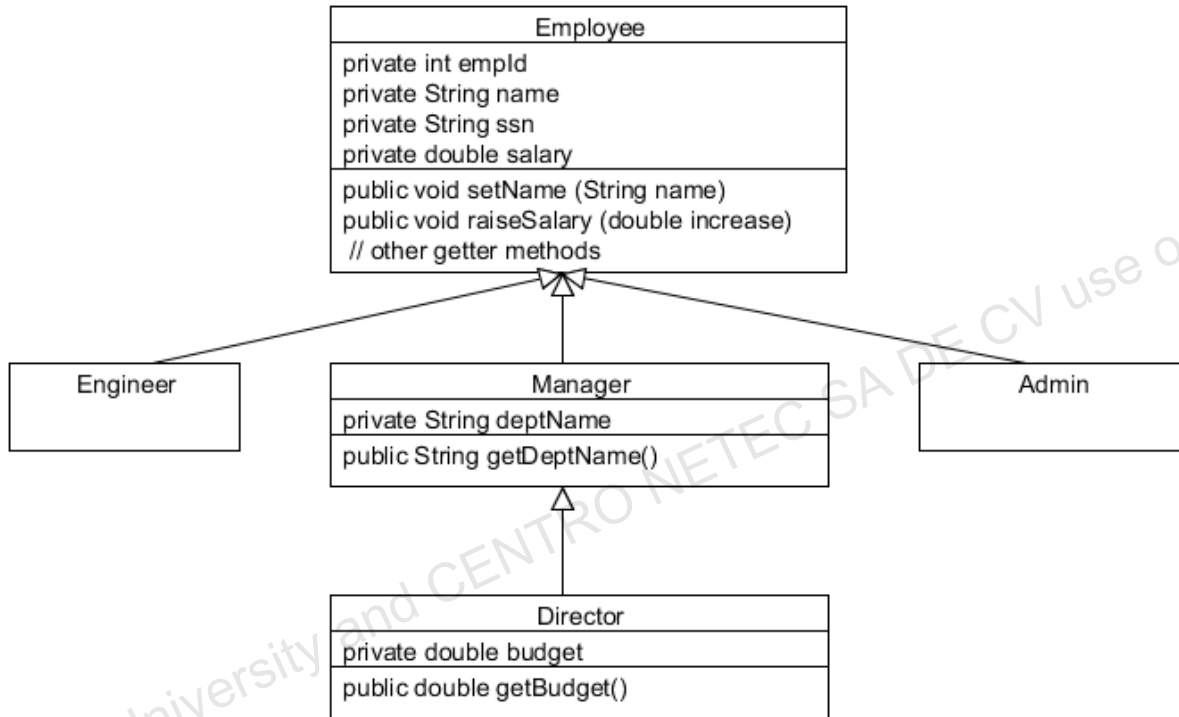
Tasks

1. Open the project `Employee03-01Prac` in the `practices/practice1` 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. Add a method named `printEmployee` to print the `Employee` object details.

   f. 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

   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 Munroe
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.   Use the `printEmployee` method to print out information about each of your Employee objects.

g.   (Optional) Use the `raiseSalary` and `setName` methods on some of your objects to make sure that those methods work.

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

8.   (Optional) Improve the look of the salary print output using the `NumberFormat` class.

a.   In the `printEmployee()` method of `Employee.java`, 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((double)
getSalary()));
```

9.   (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.
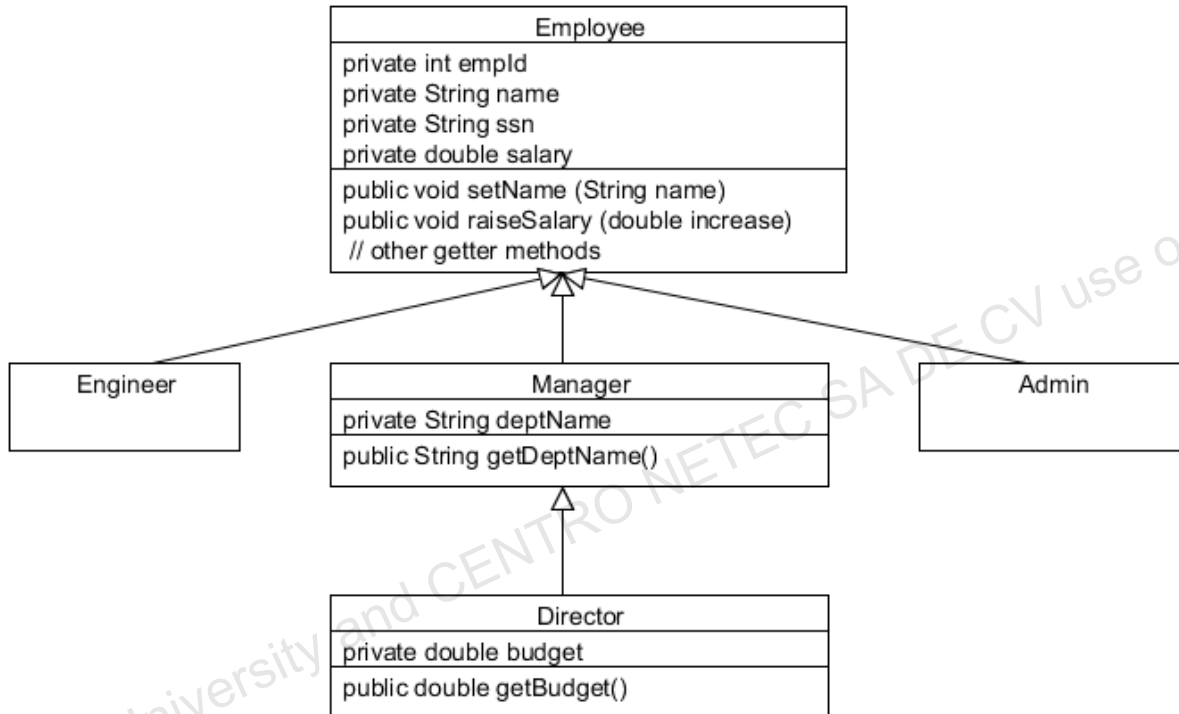
Tasks

1. In NetBeans, open the project `Employee03-01Prac` from the `practices` directory.
   a. Select **File > Open Project**.
   b. Browse to `/home/oracle/labs/03-Encapsulation/practices/practice1`.
   c. Select `Employee03-01Prac`.
   d. Click Open Project.
2. Apply encapsulation to the `Employee` class.
   a. Open `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.   Add a method named `printEmployee`.

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

g.   Resolve any missing import statements.

h.   Save `Employee.java`.

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

a.   Right-click the package `com.example.domain` and select **New > Java Class**.

b.   In the New Java Class window, perform the following steps:

1)   Enter the class name as `Manager`.

2)   Click Finish.

c.   Modify the `Manager` 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.

1)   Add a private String field called `deptName` to store the department name.

2) 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;
}
```

3) Add a getter method for `deptName`.

d. 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.

a. 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.

b. 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

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 Munroe
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. Delete the `System.out.println` statements used to display the details of the `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());
```

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

```
eng.printEmployee();
adm.printEmployee();
mgr.printEmployee();
dir.printEmployee();
```

h. (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);
mgr.printEmployee();
```

8. Save the `EmployeeTest` class.

9. Test your work, run the `EmployeeTest` class.

10. (Optional) Improve the look of the salary print output by using the `NumberFormat` class.

a. In the `printEmployee()` method of `Employee.java`, 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.

b. Replace `emp.getSalary()` by

```
NumberFormat.getCurrencyInstance().format((double)
getSalary()));
```

11. (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.

Practices for Lesson 4: Overriding Methods and Applying Polymorphism

Chapter 4

Practices for Lesson 4

Practices Overview

In these practices, you will

- Use static method
- Override methods, including the `toString` method in the Object class
- Create a method in a class that uses the `instanceof` operator to determine which object was passed to the method
- Overload methods
- Use casting

Practice 4-1: Summary Level: Overriding and Overloading Methods

Overview

In this practice, you will use a static method, 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 `Employee04-01Prac` project in the `practices/practice1` directory.

2. Edit the `Employee` class:

    a. Delete the instance method `printEmployee()`.

    b. Override the `toString()` method from the `Object` class. `Object`'s `toString` method returns a `String`.

    > I.   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"`).
    >
    > II. To format the double salary, use the following:
    >
    > > i.`NumberFormat.getCurrencyInstance().format(getSalary())`
    >
    > III. Fix any missing import statements.
    >
    > IV. 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:

    ```
    14 ⌐         }
    15 
       ⊚    ┃    @Override
    17 ⤢⊟         public String toString() {
    18      ┃         return super.toString() + "\nDepartment:        " + getDeptName();
    19 ⌐         }
    ```
    Overrides method from: com.example.domain.Employee

    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

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

   b. 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.

   c. Resolve any missing import statements.

   d. Save the `EmployeeStockPlan` class.

6. Modify the `EmployeeTest` class:

a. Add a static `printEmployee` method that invokes the toString method of the Employee class.

```
public static void printEmployee(Employee emp) {

        System.out.println(emp);

    }
```

b. Overload the `printEmployee` method to take a second parameter, `EmployeeStockPlan`, and print out the number of stock options that this employee will receive.

    i. 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));
```

c. 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);
```

d. Modify the remaining `printEmployee` invocations.

```
printEmployee(adm, esp);
printEmployee(mgr, esp);
printEmployee(dir, esp);
```

e. Modify the code used to display the Managers stock plan after invoking the `raiseSalary` method to

```
printEmployee(mgr, esp);
```

7. 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 SSN:  012-34-5678
Employee salary:    $120,345.27
Stock Options:      10
```

8. 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`, and so on). The output of the first employee record should now look like this:

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

Practice 4-1: Detailed Level: Overriding and Overloading Methods

Overview

In this practice, you will use a static method, 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.

Tasks

1. Open the `Employee04-01Prac` project in the `practices` directory.

    a. Select File > Open Project.

    b. Browse to `/home/oracle/labs/04-Polymorphism/practices/practice1`.

    c. Select `Employee04-01Prac` 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. Delete the instance method `printEmployee()` from the `Employee` class.

    ```
    public  void printEmployee() {


            System.out.println(); // Print a blank line as a
    separator
            // Print out the data in this Employee object
            System.out.println("Employee id:        " +
    getEmpId());
            System.out.println("Employee name:        " + getName());
            System.out.println("Employee SSN:  " + getSsn());
            System.out.println("Employee salary:        " +
    NumberFormat.getCurrencyInstance().format((double)
    getSalary()));
            }
    ```

    b. Add the `toString` method to the `Employee` class with the following signature:

    `public String toString() {`

    c. 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());
    ```

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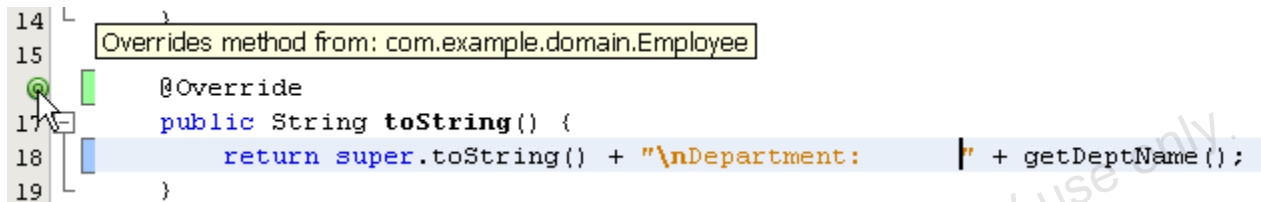
Click the icon, and NetBeans will open the `Employee` class and position the view to the `toString()` method.

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. In the New Java Class window, perform the following steps:

1) Enter the class name as `EmployeeStockPlan`.

2) Enter the package name as `com.example.business`.

3) Click Finish.

c. Add fields to the `EmployeeStockPlan` 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:

a. Add a static `printEmployee` method.

```
public static void printEmployee(Employee emp) {

        System.out.println(emp);
}
```

Note: This code of line invokes the `toString()` method of the Employee class.

The instance method `printEmployee` has been converted to a static method in this practice.

b. Overload the `printEmployee` method to take a second parameter, `EmployeeStockPlan`, and print out the number of stock options that this employee will receive.

I. Create another `printEmployee` method that takes an instance of the `EmployeeStockPlan` class:

   a. `public static void printEmployee(Employee emp, EmployeeStockPlan esp) {`

II. This method first calls the original `printEmployee` method:

   a. `printEmployee(emp);`

III. 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));
```

c. Resolve any missing import statements.

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);
```

e. Modify the remaining printEmployee invocations.

```
printEmployee(adm, esp);
printEmployee(mgr, esp);
printEmployee(dir, esp);
```

f. Modify the code used to display the Managers stock plan after invoking the raiseSalary method to

```
printEmployee(mgr, esp);
```

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 SSN:  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 SSN:  012-34-5678
Employee salary:      $120,345.27
Stock Options:        10
```

Practice 4-2: Summary Level: Using Casting

Overview

In this practice, you will cast object references and invoke appropriate methods.

You are provided with an `Employee04-02Prac` project that has some compilation errors. You will fix the errors and review the desired output. On running the project, you will encounter a runtime exception for which you need to determine the cause and fix it.

Tasks

Open the `Employee04-02Prac` project in the `practices/practice2` directory.

1. Examine the `main` method of `EmployeeTest.java` and identify lines of code that does object casting.
2. Examine the compilation errors related to casting and identify their cause.
3. Fix the compilation errors.
4. Run the project. Verify if you get a run time exception.
5. Identify the specific exception and determine the line number that caused the run time exception.
   a. Fix the cause of the exception.
6. Run the project and verify the output.

Practices for Lesson 4: Overriding Methods and Applying Polymorphism

Practice 4-2: Detailed Level: Using Casting

Overview

In this practice, you will cast object references and invoke appropriate methods.

You are provided with the `Employee04-02Prac` project that has some compilation errors. You will fix the errors and review the desired output. On running the project, you will encounter a runtime exception for which you need to determine the cause and fix it.

Tasks

1. Open the `Employee04-02Prac` project in the `/home/oracle/labs/04-Polymorphism/practices/practice2` directory.

2. Examine the `main` method of `EmployeeTest.java` and identify lines of code that does object casting.

3. Examine the compilation errors at line numbers 17, 20, and 23 related to casting and identify their cause.

```
12          // Create the classes as per the practice
13          Engineer eng = new Engineer(101, "Jane Smith", "012-34-5678", 120_345.27);
14          Employee emp = new Employee(13, "Lionel Power", "099-90-6789", 67_990.90);
15          Employee obj = new Engineer(102, "Robert Stock", "012-54-7812", 220_345.27);
16
17          obj.engineerMethod();
18          printEmployee(obj);
19
20          Engineer engobj = new Employee(1, "Brenda Wills", "013-78-5678", 221_500.00);
21          printEmployee(engobj);
22
23          String s = (String) emp;
24
25      }
```

4. Fix the compilation errors.

   a. Modify line 17 to: `eng.engineerMethod();`

   b. Modify line 20 to downcast:
   ```
   Engineer engobj =  (Engineer)new Employee(1, "Brenda Wills", "013-78-5678", 221_500.00);
   ```

   c. Comment out line 23: `//String s = (String) emp;`

5. On the Projects tab, select `Employee04-02Prac`, right-click and select **Run** from the drop down menu.

   a. Verify if you get a run time exception:

```
Output - EmployeeSolution (run) ×
 run:
 Method specific to Engineer class

 Employee id:         102
 Employee name:       Robert Stock
 Employee Soc Sec #:  012-54-7812
 Employee salary:     $220,345.27
 Exception in thread "main" java.lang.ClassCastException: com.example.domain.Employee cannot be cast to com.example.domain.Engineer
         at com.example.EmployeeTest.main(EmployeeTest.java:33)
 Java Result: 1
 BUILD SUCCESSFUL (total time: 0 seconds)
```

6. Identify the specific exception and determine the cause of the run time exception.

a. Fix the cause of the exception.
   Modify Line 20 to:
   ```
   Engineer engobj =  new Engineer(1, "Brenda Wills", "013-78-
   5678", 221_500.00);
   ```

7. Run the project and verify the output.

```
run:
Method specific to Engineer class

Employee id:         102
Employee name:       Robert Stock
Employee Soc Sec #:  012-54-7812
Employee salary:     $220,345.27

Employee id:         1
Employee name:       Brenda Wills
Employee Soc Sec #:  013-78-5678
Employee salary:     $221,500.00
BUILD SUCCESSFUL (total time: 0 seconds)
```

Practice 4-3: 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.

Summary

You are working on server software that synchronizes with other servers. Your task is to create a Singleton class which stores the hostnames of the servers to connect with. The server list is declared in a static initialization block.

Tasks

1. Open the `Singleton04-03Prac` project.
   a. Select File > Open Project.
   b. Browse to `\home\oracle\labs\04-Polymorphism\practices\practice3`.
   c. Select `Singleton04-03Prac` and click Open Project.
2. Expand the project directories.
3. Modify the `PeerSingleton` class to implement the Singleton design pattern.
   a. Open the `PeerSingleton.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:
      i. `private`
      ii. Marked `static`
      iii. Marked `final`
      iv. Type of `PeerSingleton`
      v. Initialized to a new `PeerSingleton` instance
   d. Create a static method named `getInstance` that returns the value stored in the `instance` field.
4. Modify the `Main` class to use the singleton.
   a. Open the `Main.java` file (under the `com.example` package).
   b. Perform the following steps in the `main` method:
      1) Create a `PeerSingleton` reference named `peerList01` and initialize it using the `getInstance` method.
      2) Create a second `PeerSingleton` reference named `peerList02` and initialize it using the `getInstance` method.
      3) Display the host names by invoking `getHostNames` on `peerList01` in a `for` loop.
      4) Next, display the host names by invoking `getHostNames` on `peerList02` in a `for` loop.
5. Run the project. You should see a list of host names.

Practice 4-3: 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.

Summary

You are working on server software that synchronizes with other servers. Your task is to create a Singleton class, which stores the hostnames of the servers to connect with. The server list is declared in a static initialization block.

Tasks

1. Open the `Singleton04-03Prac` project.

    a. Select File > Open Project.
    b. Browse to `\home\oracle\labs\04-Polymorphism\practices\practice3`.
    c. Select `Singleton04-03Prac` and click Open Project.

2. Expand the project directories.

3. Modify the `PeerSingleton` class to implement the Singleton design pattern.

    a. Open the `PeerSingleton.java` file (under the `com.example` package).
    b. Change the constructor's access level to `private`.

    ```
    private PeerSingleton()
    {
      }
    ```

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

    a. `private`
    b. Marked `static`
    c. Marked `final`
    d. Type of `PeerSingleton`
    e. Initialized to a new `PeerSingleton` instance

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

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

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

5. Modify the `Main` class to use the singleton.

    a. Open the `Main.java` file (under the `com.example` package).
    b. Perform the following steps in the `main` method:
    c. Create a `PeerSingleton` reference named `peerList01` and initialize it using the `getInstance` method.

    ```
    PeerSingleton peerList01 = PeerSingleton.getInstance();
    ```

d. Create a second `PeerSingleton` reference named `peerList02` and initialize it using the `getInstance` method.

```
PeerSingleton peerList02 = PeerSingleton.getInstance();
```
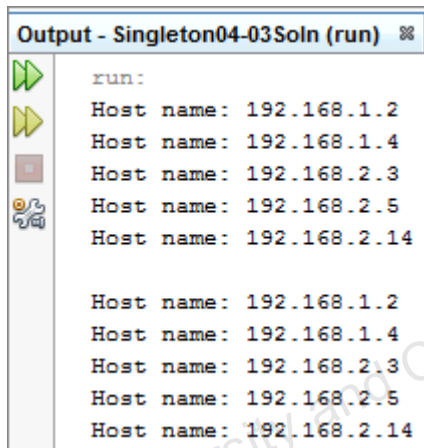
e. Display the host names by invoking `getHostNames` on `peerList01` in a `for` loop.

```
for(String hostName:peerList01.getHostNames()){
      System.out.println("Host name: " + hostName);
   }
```

f. Next, display the host names by invoking `getHostNames` on `peerList02` in a `for` loop.

```
System.out.println();
    for(String hostName:peerList02.getHostNames()){
       System.out.println("Host name: " + hostName);
    }
```

6. Run the project. You should see a list of host names.

```
Output - Singleton04-03Soln (run)    ✖

  run:
  Host name: 192.168.1.2
  Host name: 192.168.1.4
  Host name: 192.168.2.3
  Host name: 192.168.2.5
  Host name: 192.168.2.14

  Host name: 192.168.1.2
  Host name: 192.168.1.4
  Host name: 192.168.2.3
  Host name: 192.168.2.5
  Host name: 192.168.2.14
```

Practices for Lesson 4: Overriding Methods and Applying Polymorphism

Practices for Lesson 4: Overriding Methods and Applying Polymorphism

Chapter 4 - Page 16

Practices for Lesson 5:
Abstract and Nested Classes

Chapter 5

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 use inner class as a helper class to a top level class.

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 an abstract class.

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 saving accounts. You will enhance the software to support checking accounts.

Additional types of accounts might be added in the future.

Tasks

1. Open the `AbstractBanking05-01Prac` project.

    a. Select File > Open Project.

    b. Browse to `/home/oracle/labs/05-Advanced_Class_Design /practices/practice1`.

    c. Select `AbstractBanking05-01Prac` and click the Open Project button.

2. Expand the project directories.

3. Review the `SavingsAccount` class.

    a. Open the `SavingsAccount.java` file (under the `com.example` package).

    b. Examine the fields and method implementations of `SavingsAccount..`

4. Review the `Account.java`, under the `com.example` package, this class is an `abstract` class. This class contains two abstract methods:

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


public abstract String getDescription();
```

5. 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 would 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.

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

```
// Create several customers and their accounts
        bank.addCustomer("Will", "Smith");
        customer = bank.getCustomer(0);
        customer.addAccount(new SavingsAccount(500.00));

        bank.addCustomer("Bradley", "Cooper");
        customer = bank.getCustomer(1);
        SavingsAccount sack = new SavingsAccount(500.00);
        customer.addAccount(sack);
        sack.deposit(500);

        bank.addCustomer("Jane", "Simms");
        customer = bank.getCustomer(2);
        customer.addAccount(new CheckingAccount(200.00,
400.00));

        bank.addCustomer("Owen", "Bryant");
        customer = bank.getCustomer(3);
        customer.addAccount(new CheckingAccount(200.00));

        bank.addCustomer("Tim", "Soley");
        customer = bank.getCustomer(4);
        customer.addAccount(new CheckingAccount(200.00));

        bank.addCustomer("Maria", "Soley");
        customer = bank.getCustomer(5);
        CheckingAccount chkAcct = new CheckingAccount(100.00);
        customer.addAccount(chkAcct);
        if (chkAcct.withdraw(900.00)) {
            customer.addAccount(chkAcct);
```

Practices for Lesson 5: Abstract and Nested Classes

```
                        System.out.print(" withdraw is successful" +
chkAcct.getBalance());
            }
```

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

```
Output - AbstractBankingPractice (run)  ✕
   run:
                            CUSTOMERS REPORT
                            ================

   Customer: Smith, Will
        Checking Account: current balance is 500.0

   Customer: Cooper, Bradley
        Checking Account: current balance is 1060.0

   Customer: Simms, Jane
        Checking Account: current balance is 200.0

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

   Customer: Soley, Tim
        Checking Account: current balance is 200.0

   Customer: Soley, Maria
        Checking Account: current balance is 100.0
   BUILD SUCCESSFUL (total time: 1 second)
```

Practices for Lesson 5: Abstract and Nested Classes

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 saving accounts. You will enhance the software to support checking accounts. Additional types of accounts might be added in the future.

Tasks

1. Open the `AbstractBanking05-01Prac` project as the main project.

    a. Select File > Open Project.

    b. Browse to `/home/oracle/labs/05-Advanced_Class_Design /practices/practice1`.

    c. Select `AbstractBanking05-01Prac`.

    d.   Click Open Project.

2. Expand the project directories.

3. Review the `SavingsAccount` class.

    a. Open the `SavingsAccount.java` file (under the `com.example` package).

    b. Examine the fields and method implementations of `SavingsAccount`.

4. Review the `Account.java`, under the `com.example` package, this class is an `abstract` class. This class contains two abstract methods:

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


    public abstract String getDescription();
    ```

5. 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;
    }
}
```

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

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

```
// Create several customers and their accounts
        bank.addCustomer("Will", "Smith");
        customer = bank.getCustomer(0);
        customer.addAccount(new SavingsAccount(500.00));

        bank.addCustomer("Bradley", "Cooper");
        customer = bank.getCustomer(1);
        SavingsAccount sack = new SavingsAccount(500.00);
        customer.addAccount(sack);
        sack.deposit(500);

        bank.addCustomer("Jane", "Simms");
        customer = bank.getCustomer(2);
        customer.addAccount(new CheckingAccount(200.00,
400.00));
```

```
            bank.addCustomer("Owen", "Bryant");
            customer = bank.getCustomer(3);
            customer.addAccount(new CheckingAccount(200.00));

            bank.addCustomer("Tim", "Soley");
            customer = bank.getCustomer(4);
            customer.addAccount(new CheckingAccount(200.00));



            bank.addCustomer("Maria", "Soley");
            customer = bank.getCustomer(5);
            CheckingAccount chkAcct = new CheckingAccount(100.00);
            customer.addAccount(chkAcct);
            if (chkAcct.withdraw(900.00)) {
                customer.addAccount(chkAcct);
                System.out.print(" withdraw is successful" +
chkAcct.getBalance());
            }
```

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

```
Output - AbstractBankingPractice (run)  ⊠
    run:
                        CUSTOMERS REPORT
                        ================

    Customer: Smith, Will
        Checking Account: current balance is 500.0

    Customer: Cooper, Bradley
        Checking Account: current balance is 1060.0

    Customer: Simms, Jane
        Checking Account: current balance is 200.0

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

    Customer: Soley, Tim
        Checking Account: current balance is 200.0

    Customer: Soley, Maria
        Checking Account: current balance is 100.0
    BUILD SUCCESSFUL (total time: 1 second)
```

Practices for Lesson 5: Abstract and Nested Classes

Practice 5-2: Summary Level: Implementing Inner Class as a Helper Class

Overview

In this practice, you will take an existing application and develop an inner class as a helper class to compute employee benefits.

Assumptions

You have reviewed the nested class section of this lesson.

Summary

You have been given a small project that contains an `Employee.java`, implement an inner class as a helper class to compute employee benefits.

Tasks

1. Open the `EmployeeInner05-02Prac` project as the main project.

   a. Select File > Open Project.

   b. Browse to `/home/oracle/labs/05-Advanced_Class_Design /practices/practice2`

   c. Select `EmployeeInner05-02Prac`

   d. Click Open Project.

2. Edit `Employee.java` and make the following changes:

   a. Develop an innerclass, `BenefitsHelper`.

   b. Declare two class variables: `bonusRate` and `withholdingRate`.

   c. Initialize `bonusRate` and `withholdingRate`.

   ```
   private final double bonusRate = 0.02;
   private final double withholdingRate = 0.07;
   ```

   d. Add 2 methods: `calcBonus` (to compute the bonus) and `calcWithholding` (to compute the withhholding).

   e. Create an instance of `BenefitsHelper` in the `Employee` class.

   f. Add 2 getter methods to the `Employee` class to return the bonus and withholding.

3. Develop `Main.java`:

   a. Create a Java class, `Main.java` in the `com.example` package.

   b. Add a `main` method to the `Main` class.

   c. Perform the following steps in the `main` method:

   - Create two instances of the `Employee` class.

   ```
   Employee jane = new Employee("Jane Doe", "Manager", "HR",
   65000);
   Employee john = new Employee("John Doe", "Staff", "HR", 55000);
   ```

- Invoke the `getWithholding()` and `getBonus()` methods to display employee benefits.

```
 System.out.println("Jane's withholding: " +
jane.getWithholding());
 System.out.println("John's bonus " + john.getBonus());
```

4. Run the project. You should see the output in the output window.
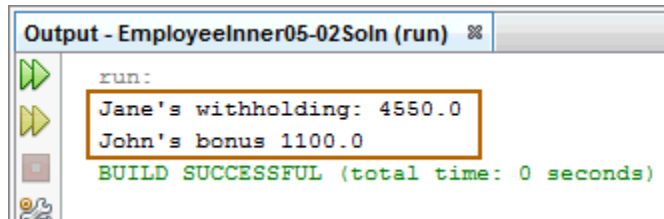
Practice 5-2: Detailed Level: Implementing Inner Class as a Helper Class

Overview

In this practice, you will take an existing application and develop an inner class as a helper class to compute employee benefits.

Assumptions

You have reviewed the nested class section of this lesson.

Summary

You have been given a small project that contains an `Employee.java`, implement an inner class as a helper class to compute employee benefits.

Tasks

1. Open the `EmployeeInner05-02Prac` project as the main project.

   a. Select File > Open Project.

   b. Browse to `/home/oracle/labs/05-Advanced_Class_Design /practices/practice2`

   c. Select `EmployeeInner05-02Prac`

   d. Click Open Project.

2. Expand the project directories.

3. Edit `Employee.java` under the `com.example` package.

   a. Create an inner class, `BenefitsHelper.java` inside the `Employee` class.

   b. Declare two variables: `bonusRate` and `withholdingRate`

   c. Initialize `bonusRate` and `withholdingRate`

   ```
   private final double bonusRate = 0.02;
   private final double withholdingRate = 0.07;
   ```

   d. Add a method `calcBonus` to calculate the bonus of the employee.

   ```
   protected double calcBonus(double salary){
      return salary * bonusRate;
   }
   ```

   e. Add a method `calcWithholding` to calculate the withholding of the employee.

   ```
   protected double calcWithholding(double salary){
       return salary * withholdingRate;
   }
   ```

   f. Create an instance of `BenefitsHelper` in the `Employee` class.

   ```
   private BenefitsHelper helper = new BenefitsHelper();
   ```

g. Add two getter methods to the `Employee` class to return the bonus and withholding.

    i. Add the `getWithholding()` method:

```
public double getWithholding(){
        return helper.calcWithholding(salary);

    }
```

    ii. Add the `getBonus()` method:

```
 public double getBonus(){
        return helper.calcBonus(salary);

    }
```

4. Create `Main.java` class under `com.example` package.

5. Modify `Main.java`:

    a. Add a `main` method to the class.

    b. Create 2 instances of the `Employee` class in the `main` method.

```
Employee jane = new Employee("Jane Doe", "Manager", "HR",
65000);

Employee john = new Employee("John Doe", "Staff", "HR", 55000);
```

    c. Invoke the `getWithholding()` and `getBonus()` methods to output the bonus and withholding of the employee instances.

```
System.out.println("Jane's withholding: " +
jane.getWithholding());

System.out.println("John's bonus " + john.getBonus());
```

6. Run the project. You should see the output in the output window.
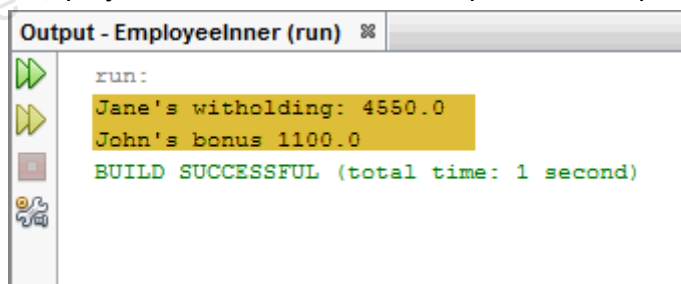
Practice 5-3: Summary Level: Using Java Enumerations

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. By creating a new Java enum you will modify the application to hold various branch locations of the bank. By using enum to store the branch details, in the future it is easy to add more branch locations to the bank, it is easy to validate branch information.

Tasks

1. Open the `EnumBanking05-03Prac` project as the main project.

   a. Select File > Open Project.

   b. Browse to `/home/oracle/labs/05-Advanced_Class_Design /practices/practice3`

   c. Select `EnumBanking05-03Prac` and 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, `Branch` in the `com.example` package.

5. Modify the enum, `Branch.java`. The `Branch` enum stores the location at which the customer banks at. In addition, information about the types of services offered by the bank is also stored.

   a. Create `Branch` instances, `LA`, `BOSTON`, `BANGALORE`, `MUMBAI` that call the `Branch` constructor with values "Basic", "Loan", "Full", and "Full", respectively.

   b. Declare a `serviceLevel` field along with a corresponding constructor and getter method.

```java
public enum Branch {

    LA("Basic"), BOSTON("Loan"), BANGALORE("Full"), MUMBAI("Full");

        String serviceLevel;
        private Branch(String serviceLevel){
            this.serviceLevel = serviceLevel;
        }

        public String getServiceLevel(){
          return serviceLevel;
        }


}
```

6. Modify the `Customer` class to store branch information.

    a. Open the `Customer.java` file (under the `com.example` package).

    b. Declare a variable of type Branch.

```
    private Branch branch;
```

    c. Modify the existing constructor to receive an enum, `Branch` as the third parameter.

    d. Add getter and setter methods for the `branch` field.

7. Modify the `Bank` class to modify `addCustomer` method.

    a. Open the `Bank.java` file (under the `com.example` package).

    b. Within the `addCustomer` method, add `Branch` instance as a parameter.

    c. Within the customer instance creation statement, modify the constructor to include `Branch` instance as a parameter.

```
public void addCustomer(String f, String l, Branch b) {
        int i = numberOfCustomers++;
        customers[i] = new Customer(f, l, b);
    }
```

8. Modify the `CustomerReport.java` to display the branch for each customer.

```
                // Print the customer's name
                System.out.println();
                System.out.println("Customer: "
                        + customer.getLastName() + ", "
                        + customer.getFirstName()
                        + "\nBranch: " + customer.getBranch() + ", "
                        + customer.getBranch().getServiceLevel());
```

9. Modify `AbstractBankingMain.java` to update the customers information with the branch details, for example:

```
bank.addCustomer("Will", "Smith",Branch.LA);
        customer = bank.getCustomer(0);
        customer.addAccount(new SavingsAccount(500.00));
```

Practices for Lesson 5: Abstract and Nested Classes

10. Run the project. You should see a report of all customers and their accounts with the branch locations of the bank.

```
                CUSTOMERS REPORT
                ================

Customer: Smith, Will
Branch: LA, Basic
    Checking Account: current balance is 500.0

Customer: Cooper, Bradley
Branch: BOSTON, Loan
    Checking Account: current balance is 1060.0

Customer: Simms, Jane
Branch: MUMBAI, Full
    Checking Account: current balance is 200.0

Customer: Bryant, Owen
Branch: BANGALORE, Full
    Checking Account: current balance is 200.0

Customer: Soley, Tim
Branch: LA, Basic
    Checking Account: current balance is 200.0

Customer: Soley, Maria
Branch: BANGALORE, Full
    Checking Account: current balance is 100.0
```

Practices for Lesson 5: Abstract and Nested Classes

Practice 5-3: Detailed Level: 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. By creating a new Java enum you will modify the application to hold various branch locations of the bank. By using enum to store the branch details, in the future it is easy to add more branch locations to the bank, it is easy to validate branch information.

Tasks

1. Open the `EnumBanking05-03Prac` project as the main project.

   a. Select File > Open Project.

   b. Browse to `/home/oracle/labs/05-Advanced_Class_Design /practices/practice3`.

   c. Select `EnumBanking05-03Prac`.

   d. Click Open Project.

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, `Branch` in the `com.example` package, by performing the following steps:

   a. In NetBeans, right-click on the project, select New > Other.

   b. Select **Java** from Categories column

   c. Select **Java Enum** from File Types column

   d. Click **Next.**

5. In the Name and Location dialog box, enter the following details:

   a. Class: `Branch`

   b. Package: `com.example`

   c. Click Finish.

6. Modify the enum, `Branch.java`. The `Branch` enum stores the location at which the customer banks at. In addition, information about the types of services offered by the bank are also stored.

   a. Create `Branch` instances, `LA`, `BOSTON`, `BANGALORE`, `MUMBAI` that call the `Branch` constructor with values "Basic", "Loan", "Full", and "Full", respectively.

   b. Declare a `serviceLevel` field along with a corresponding constructor and getter method.

```
public enum Branch {

    LA("Basic"), BOSTON("Loan"), BANGALORE("Full"), MUMBAI("Full");
```

```
            String serviceLevel;
            private Branch(String serviceLevel){
                this.serviceLevel = serviceLevel;
            }

            public String getServiceLevel(){
              return serviceLevel;
            }


}
```

7. Modify the `Customer` class to store branch information.

    a. Open the `Customer.java` file (under the `com.example` package).

    b. Declare a variable of type Branch.

```
    private Branch branch;
```

c. Modify the existing constructor to receive an enum, `Branch` as the third parameter.

```
    public Customer(String f, String l,Branch b) {
        firstName = f;
        lastName = l;
        // initialize accounts array
        accounts = new Account[10];
        numberOfAccounts = 0;
        branch=b;

    }
```

    d. Add getter and setter methods for the `branch` field.

```
public Branch getBranch() {
        return branch;
    }

    public void setBranch(Branch branch) {
        this.branch = branch;
    }
```

8. Modify the `Bank` class to modify `addCustomer` method.

    a. Open the `Bank.java` file (under the `com.example` package).

    b. Within the `addCustomer` method, add `Branch` instance as a parameter.

    c. Within the customer instance creation statement, modify the constructor to include `Branch` instance as a parameter.

```
public void addCustomer(String f, String l, Branch b) {
        int i = numberOfCustomers++;
        customers[i] = new Customer(f, l, b);
    }
```

Practices for Lesson 5: Abstract and Nested Classes

9. Modify the `CustomerReport.java` to display the branch for each customer.

```
                    // Print the customer's name
            System.out.println();
            System.out.println("Customer: "
                        + customer.getLastName() + ", "
                        + customer.getFirstName()
                        + "\nBranch: " + customer.getBranch() + ", "
                        + customer.getBranch().getServiceLevel());
```

10. Modify `AbstractBankingMain.java` to update the customer's information with the branch details.

```
bank.addCustomer("Will", "Smith",Branch.LA);
        customer = bank.getCustomer(0);
        customer.addAccount(new SavingsAccount(500.00));

        bank.addCustomer("Bradley", "Cooper", Branch.BOSTON);
        customer = bank.getCustomer(1);
        SavingsAccount sack = new SavingsAccount(500.00);
        customer.addAccount(sack);
        sack.deposit(500);

        bank.addCustomer("Jane", "Simms", Branch.MUMBAI);
        customer = bank.getCustomer(2);
        customer.addAccount(new CheckingAccount(200.00, 400.00));

        bank.addCustomer("Owen", "Bryant", Branch.BANGALORE);
        customer = bank.getCustomer(3);
        customer.addAccount(new CheckingAccount(200.00));

        bank.addCustomer("Tim", "Soley", Branch.LA);
        customer = bank.getCustomer(4);
        customer.addAccount(new CheckingAccount(200.00));

        bank.addCustomer("Maria", "Soley",Branch.BANGALORE);
        customer = bank.getCustomer(5);
        CheckingAccount chkAcct = new CheckingAccount(100.00);
```

d. Run the project. You should see a report of all customers and their accounts with the branch locations of the bank.

```
            CUSTOMERS REPORT
            ================

Customer: Smith, Will
Branch: LA, Basic
    Checking Account: current balance is 500.0

Customer: Cooper, Bradley
Branch: BOSTON, Loan
    Checking Account: current balance is 1060.0

Customer: Simms, Jane
Branch: MUMBAI, Full
```

Practices for Lesson 5: Abstract and Nested Classes

```
        Checking Account: current balance is 200.0

Customer: Bryant, Owen
Branch: BANGALORE, Full
        Checking Account: current balance is 200.0

Customer: Soley, Tim
Branch: LA, Basic
        Checking Account: current balance is 200.0

Customer: Soley, Maria
Branch: BANGALORE, Full
        Checking Account: current balance is 100.0
```

Practices for Lesson 5: Abstract and Nested Classes

Practices for Lesson 6: Interfaces and Lambda Expressions

Chapter 6

Practices for Lesson 6: Overview

Practices Overview

In these practices, you will use Java interfaces and lambda expressions.

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

1. Open the `Pet06-01Prac` project.

   a. Select File > Open Project.

   b. Browse to `/home/oracle/labs/06-Interfaces/practices/practice1`.
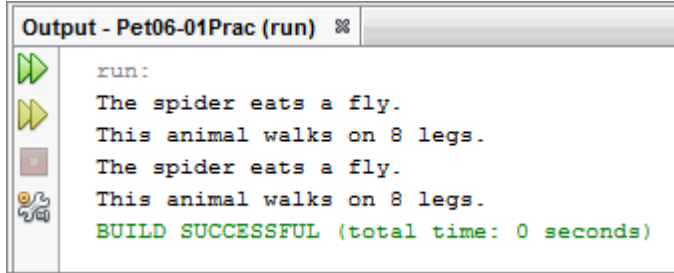
   c. Select `Pet06-01Prac` click Open Project.

2. Expand the project directories.

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

```
Output - Pet06-01Prac (run)  ✖

run:
The spider eats a fly.
This animal walks on 8 legs.
The spider eats a fly.
This animal walks on 8 legs.
BUILD SUCCESSFUL (total time: 0 seconds)
```

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

Practices for Lesson 6: Interfaces and Lambda Expressions

- 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.

```
Output - Pet06-01Soln (run)  ⌘

run:
The spider eats a fly.
This animal walks on 8 legs.
The spider eats a fly.
This animal walks on 8 legs.
Cats like to eat spiders and fish.
This animal walks on 4 legs.
Tom likes to play with string.
Cats like to eat spiders and fish.
This animal walks on 4 legs.
Mr. Whiskers likes to play with string.
Fish eat pond scum.
This animal walks on 0 legs.
Fish, of course, can't walk; they swim.
Just keep swimming.
Fish eat pond scum.
This animal walks on 0 legs.
Fish, of course, can't walk; they swim.
Danger! Wild Animal
Tom likes to play with string.
Just keep swimming.
BUILD SUCCESSFUL (total time: 0 seconds)
```

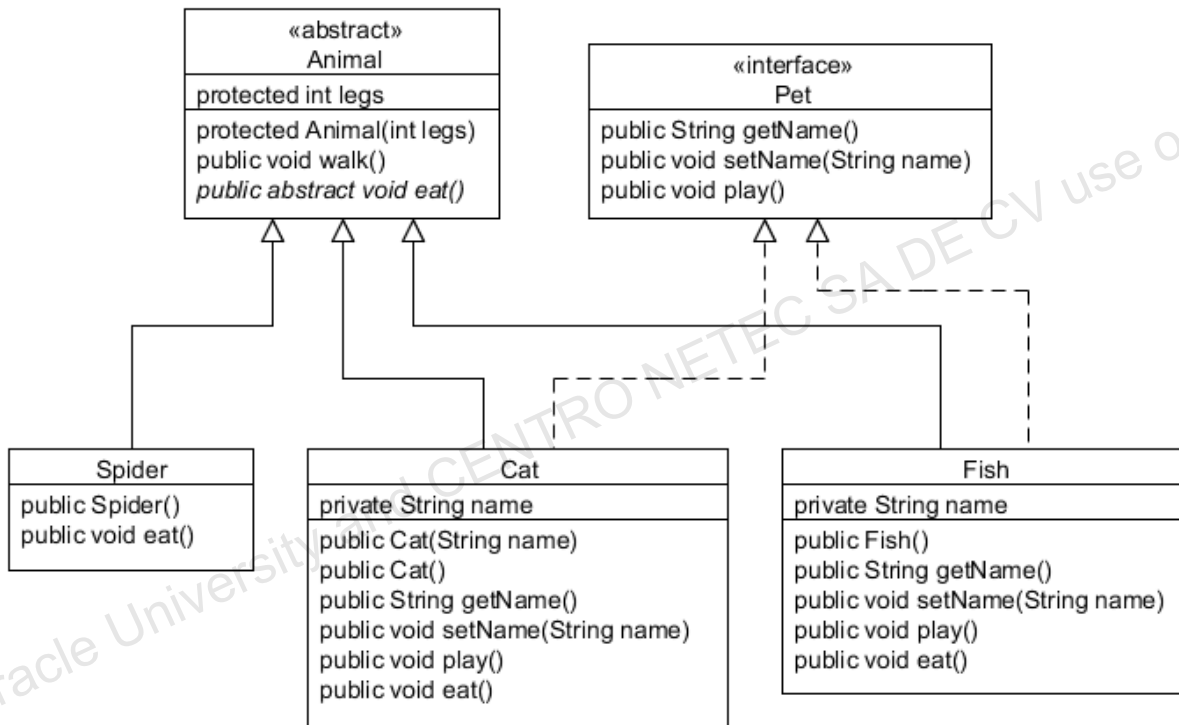# Practice 6-1: Detailed Level: Implementing an Interface

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

1. Open the `Pet06-01Prac` project.

    a. Select File > Open Project.

    b. Browse to `/home/oracle/labs/06-Interfaces/practices/practice1`.

    c. Select `Pet06-01Prac` and click Open Project.

2. Expand the project directories.

3. 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");
}
```

Practices for Lesson 6: Interfaces and Lambda Expressions

- 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();
```

Practices for Lesson 6: Interfaces and Lambda Expressions

```
a = new Fish();
a.eat();
a.walk();
```

d. Implement and test the `playWithAnimal(Animal a)` method.
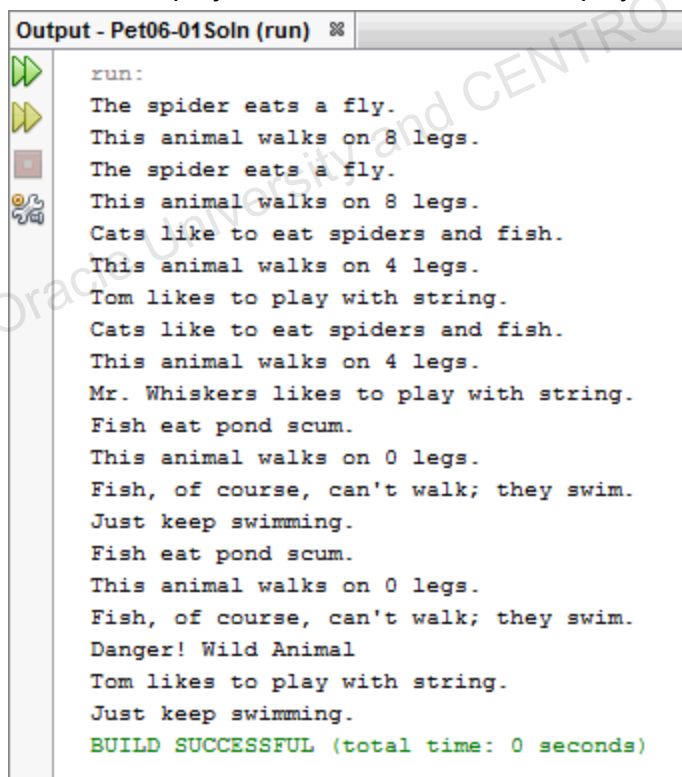
  ▪ Determine whether the argument implements the `Pet` interface. If so, cast the reference to a `Pet` and invoke the `play` method. If not, print a message of `"Danger! Wild Animal"`.

```
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.


```
run:
The spider eats a fly.
This animal walks on 8 legs.
The spider eats a fly.
This animal walks on 8 legs.
Cats like to eat spiders and fish.
This animal walks on 4 legs.
Tom likes to play with string.
Cats like to eat spiders and fish.
This animal walks on 4 legs.
Mr. Whiskers likes to play with string.
Fish eat pond scum.
This animal walks on 0 legs.
Fish, of course, can't walk; they swim.
Just keep swimming.
Fish eat pond scum.
This animal walks on 0 legs.
Fish, of course, can't walk; they swim.
Danger! Wild Animal
Tom likes to play with string.
Just keep swimming.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Practices for Lesson 6: Interfaces and Lambda Expressions

Practice 6-2: Summary Level: Using Java Interfaces

Overview

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

Assumptions

You have reviewed the `interface` section of this lesson.

Summary

You have been given a project that implements the logic for a bank. Update the application to use Java interfaces.

Tasks

1. Open the `InterfaceBanking06-02Prac` project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/06-interfaces/practices/practice2`.
   c. Select `InterfaceBanking06-02Prac` and click Open Project.
2. Expand the project directories.
3. Run the project. You should see a report of all customers and their accounts.
4. Two interface files have been created for you `AccountOperations.java` and `BankOperations.java`. You will update these files.
   Note: Certain steps that follow may generate a number of errors in your source files. Do not panic! The errors will be fixed as you proceed through the changes.
5. Open the `Account.java` file and the `AccountOperations.java` file.
6. Copy the following method signatures from the `Account.java` file to the `AccountOperations.java` file. Here are the method names you should copy: `getBalance()`, `deposit()`, `withdraw()`, and `getDescription()`.
7. Update `CheckingAccount.java` to use implement `AccountOperations`.
8. Update `SavingsAccount.java` to use implement `AccountOperations`.
9. In `Account.java` remove the following methods: `getBalance()`, `deposit()`, `withdraw()`, and `getDescription()`.
10. In `Account.java` update the `toString()` method to print a message without calling `getDescription()`.
11. Save `Account.java`. Close the file.
12. Edit `CheckingAccount.java`.
13. Implement a `getBalance()` method.
14. Implement a `deposit()` method.
15. Override the `toString` method.
16. Save the file. Close the file.
17. Edit `SavingsAccount.java`.
18. Implement a `getBalance()` method.
19. Override the `toString` method.

20. Save the file. Close the file.
21. Edit the `Bank.java` file.
22. Update `Bank.java` so that it implements the `BankOperations` class.
23. Save the file.
24. Edit the `BankOperations.java` file.
25. Copy the following method signatures from the `Bank.java` file to the `BankOperations.java` file. The methods signatures to copy are: `addCustomer()`, `getNumOfCustomers()`, and `getCustomer()`.
26. Save the file.
27. Open the `CustomerReport.java` file.
28. Copy the `generateReport()` method to the `BankOperations.java` file.
29. In the newly copied method, change any reference to `bank` to `this`.
30. Save the `BankOperations.java` file.
31. Delete the `CustomerReport.java` file.
32. Open the `Main.java` file.
33. Change the type definition of `bank` to the new interface `BankOperations`.
34. Change the code to call the `generateReport` method from `bank`.
35. Run the project. Everything should print again.
36. Edit the `Customer.java` file.
37. Change the `Account[]` array to an `AccountOperations[]` array.
38. Fix any resulting errors by changing the references from `Account` to `AccountOperations`.
39. Save the file.
40. Fix the reference error in `BankOperations` caused by this change. Save the file.
41. Edit the `Main.java`.
42. Change any Checking or Savings account references to `AccountOperations` references.
    Hint: Changes should be made to accounts: 1 and 5

43. Run the project. The output should look like the following:

```
            CUSTOMERS REPORT
            ================

Customer: Smith, Will
Branch: LA, Basic
     Savings Account balance is 500.0

Customer: Cooper, Bradley
Branch: Boston, Loan
     Savings Account balance is 1060.0

Customer: Simms, Jane
Branch: Mumbai, Full
     Checking Account balance is 200.0

Customer: Bryant, Owen
Branch: Bangalore, Full
     Checking Account balance is 200.0

Customer: Soley, Tim
Branch: LA, Basic
     Checking Account balance is 200.0

Customer: Soley, Maria
Branch: Bangalore, Full
     Checking Account balance is 100.0
```

Practices for Lesson 6: Interfaces and Lambda Expressions

Practice 6-2: Detailed Level: Using Java Interfaces

Overview

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

Assumptions

You have reviewed the `interface` section of this lesson.

Summary

You have been given a project that implements the logic for a bank. Update the application to use Java interfaces.

Tasks

1. Open the `InterfaceBanking06-02Prac` project.

    a. Select File > Open Project.

    b. Browse to `/home/oracle/labs/06-interfaces/practices/practice2`.

    c. Select `InterfaceBanking06-02Prac` and click Open Project.

2. Expand the project directories.

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

4. Two interface files have been created for you `AccountOperations.java` and `BankOperations.java`. You will update these files.

    Note: Certain steps that follow may generate a number of errors in your source files. Do not panic! The errors will be fixed as you proceed through the changes.

5. Open the `Account.java` file and the `AccountOperations.java` file.

6. Copy the following method signatures from the `Account.java` file to the `AccountOperations.java` file. Here are the method names you should copy: `getBalance()`, `deposit()`, `withdraw()`, and `getDescription()`.

7. Update `CheckingAccount.java` to use implement `AccountOperations`.

    ```
    public class CheckingAccount extends Account implements
    AccountOperations
    ```

8. Update `SavingsAccount.java` to use implement `AccountOperations`.

    ```
    public class SavingsAccount extends Account implements
    AccountOperations
    ```

9. In `Account.java` remove the following methods: `getBalance()`, `deposit()`, `withdraw()`, and `getDescription()`.

10. In `Account.java` update the `toString()` method to print a message without calling `getDescription()`.

    ```
    return "Current balance is "  + balance;
    ```

11. Save `Account.java`. Close the file.

12. Edit `CheckingAccount.java`.

13. Implement a `getBalance()` method.

```
@Override
public double getBalance(){
  return balance;
}
```

14. Implement a `deposit()` method.

```
@Override
public void deposit(double amount) {
    balance += amount;
}
```

15. Override the `toString` method.

```
@Override
public String toString() {
  return this.getDescription() +" balance is " + balance;
}
```

16. Save the file. Close the file.

17. Edit `SavingsAccount.java`.

18. Implement a `getBalance()` method.

```
@Override
public double getBalance(){
  return balance;
}
```

19. Override the `toString` method.

```
@Override
public String toString() {
  return this.getDescription() +" balance is " + balance;
}
```

20. Save the file. Close the file.

21. Edit the `Bank.java` file.

22. Update `Bank.java` so that it implements the `BankOperations` class.

```
public class Bank implements BankOperations
```

23. Save the file.

24. Edit the `BankOperations.java` file.

25. Copy the following method signatures from the `Bank.java` file to the `BankOperations.java` file. The methods signatures to copy are: `addCustomer()`, `getNumOfCustomers()`, and `getCustomer()`.

26. Save the file.

27. Open the `CustomerReport.java` file.

28. Copy the `generateReport()` method to the `BankOperations.java` file.

   ▪ Change the method signature in `BankOperations.java` to:

```
public default void generateReport()
```

29. In the newly copied method, change any `bank` references to `this`.

30. Save the `BankOperations.java` file.

31. Delete the `CustomerReport.java` file.

32. Open the `Main.java` file.

33. Change the definition of `bank` to the following:

```
BankOperations bank = new Bank();
```

34. Change the code to call the `generateReport` method from bank.

   ▪ Replace these lines:

```
CustomerReport report = new CustomerReport();
report.setBank(bank);
report.generateReport();
```

   ▪ with this line:

```
bank.generateReport();
```

35. In the same file, update the `initializeCustomers(BankOperations bank)` method. Make the method `static` and note that a `BankOperations` object is passed in.

36. Save the file.

37. Run the project. Everything should print again.

38. Edit the `Customer.java` file.

39. Change the `Account[]` array to an `AccountOperations[]` array.

40. Fix any resulting errors by changing the references from `Account` to `AccountOperations`.

41. Save the file.

42. Fix the reference error in `BankOperations` caused by this change. Save the file.

43. Edit the `Main.java`.

44. Change any Checking or Savings account references to `AccountOperations` references.
   Hint: Changes should be made to accounts: 1 and 5

45. Run the project. The output should look like the following:

```
                 CUSTOMERS REPORT
                 ================

Customer: Smith, Will
Branch: LA, Basic
     Savings Account balance is 500.0

Customer: Cooper, Bradley
Branch: Boston, Loan
     Savings Account balance is 1060.0

Customer: Simms, Jane
Branch: Mumbai, Full
     Checking Account balance is 200.0

Customer: Bryant, Owen
Branch: Bangalore, Full
     Checking Account balance is 200.0

Customer: Soley, Tim
Branch: LA, Basic
     Checking Account balance is 200.0

Customer: Soley, Maria
Branch: Bangalore, Full
     Checking Account balance is 100.0
```

Practice 6-3: Summary Level: Write Lambda Expressions

Overview

In this practice, write additional lambda expressions for the `StringAnalzyer` application.

Assumptions

You have reviewed the lambda expressions section of this lesson.

Summary

Use the `StringAnalyzer` project from the lecture to create 3 additional lambda expressions.

Tasks

1. Open the `LambdaBasics06-03Prac` project.

    a. Select File > Open Project.

    b. Browse to `/home/oracle/labs/06-interfaces/practices/practice3`.

    c. Select `LambdaBasics06-03Prac` and click Open Project.

2. Expand the project directories.

3. Open the `LambdaTest.java` file.

4. Write a lambda expression that displays strings that end with the search string.

5. Write a lambda expression that displays strings that contain the search string and are 5 characters or less in length.

6. Write a lambda expression that displays strings that contain the search string and are greater than 5 characters in length.

7. Run the project. The output should be as follows:

```
Searching for: to
==Contains==
Match: tomorrow
Match: toto
Match: to
Match: timbukto
==Starts With==
Match: tomorrow
Match: toto
Match: to
==Equals==
Match: to
==Ends With==
Match: toto
Match: to
Match: timbukto
==Less than 5==
Match: toto
Match: to
==Greater than 5==
Match: tomorrow
Match: timbukto
```

Practice 6-3: Detailed Level: Write Lambda Expressions

Overview

In this practice, write additional lambda expressions for the `StringAnalzyer` application.

Assumptions

You have reviewed the lambda expressions section of this lesson.

Summary

Use the `StringAnalyzer` project from the lecture to create three additional lambda expressions.

Tasks

1. Open the `LambdaBasics06-03Prac` project.

    a. Select File > Open Project.

    b. Browse to `/home/oracle/labs/06-interfaces/practices/practice3`.

    c. Select `LambdaBasics06-03Prac` and click Open Project.

2. Expand the project directories.

3. Open the `LambdaTest.java` file

4. Write a lambda expression that displays strings that end with the search string.

```
(t,s) -> t.endsWith(s));
```

5. Write a lambda expression that displays strings that contain the search string and are 5 characters or less in length.

```
(t,s) -> t.contains(s) && t.length() < 5;
```

6. Write a lambda expression that displays strings that contain the search string and are greater than five characters in length.

```
(t,s) -> t.contains(s) && t.length() > 5;
```

7. Run the project. The output should be as follows:

```
Searching for: to
==Contains==
Match: tomorrow
Match: toto
Match: to
Match: timbukto
==Starts With==
Match: tomorrow
Match: toto
Match: to
==Equals==
Match: to
==Ends With==
Match: toto
Match: to
Match: timbukto
==Less than 5==
Match: toto
Match: to
==Greater than 5==
Match: tomorrow
Match: timbukto
```

Practices for Lesson 7:
Generics and Collections

Chapter 7

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

1. In NetBeans, open the `GenericsHashMap07-01Prac` project

    a. Select File > Open Project.

    b. Browse to `/home/oracle/labs/07-Generics_Collections/practices/practice1`

    c. Select `GenericsHashMap07-01Prac` and click Open Project.

2. Expand the project directories.

3. Open `ProductCounter.java` in the editor and make the following changes:

    a. 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.

    b. 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.

c. 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){ }
```

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

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

e. Add code to the `main` method to create the `ProductCounter` object and process the same.

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

Practices for Lesson 7: Generics and Collections

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

1. In NetBeans, open the `GenericsHashMap07-01Prac` project.

   a. Select File > Open Project.

   b. Browse to `/home/oracle/labs/07-Generics_Collections/practices/practice1`

   c. Select `GenericsHashMap07-01Prac` and click Open Project.

2. Expand the project directories.

3. Open `ProductCounter.java` in the editor and make the following changes:

   a. Add two private map fields- `productCountMap` and `productNames`. 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<>();
   ```

Practices for Lesson 7: Generics and Collections

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

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

c. 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));
        }
    }
}
```

d. 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)));
    }
}
```

4. Add the following code to the `main` method to create the `ProductCounter` object and process the same.

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

Practices for Lesson 7: Generics and Collections

5. Run the `ProductCounter.java` and verify the output.

```
run:
=== 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
BUILD SUCCESSFUL (total time: 0 seconds)
```

Practices for Lesson 7: Generics and Collections

Practice 7-2: Summary Level: Implementing Stack using a `Deque`

Overview

In this practice, you use the `Deque` object to implement a Stack.

Assumptions

You have reviewed all the content in this lesson.

Summary

Use the `Deque` data structure to implement a stack to support `push`, `pop` and `peek` operations.

Tasks

1. In NetBeans, open the `Stack07-02Prac` project

   a. Select File > Open Project.

   b. Browse to `/home/oracle/labs/07-Generics_Collections/practices/practice2`

   c. Select `Stack07-02Prac` and click Open Project.

2. Expand the project directories.

3. Open `IntegerStack.java` in the editor and make the following changes:

   a. Implement the `push()` method to add an Integer to the stack:

      Use the method `addFirst(element)` from the `Deque` API.

   b. Implement the `pop()` method that deletes an Integer from the top of the stack:

      Use the `removeFirst()` method from the `Deque` API, also check for stackunderflow condition before deleting the element by using `isEmpty()` method from the `Deque` API.

   c. Implement `peek()` method which returns the element at the top of the stack:

      Use the method `peekFirst()` from the `Deque` API.

   d. Override the `toString()` method.

4. Add a `main` method the class and perform the following steps:

   a. Create an instance of the Stack Class:

   ```
   IntegerStack stack = new IntegerStack();
   ```

   b. Perform various operations on the `stack` by invoking various methods: `push()`,`pop()` and `peek()`.

5. Run the `IntegerStack.java` class to ensure that your program produces the desired output.

Practice 7-2: Detailed Level: Implementing Stack Using a `Deque`

Overview

In this practice, you use the `Deque` object to implement a Stack.

Assumptions

You have reviewed all the content in this lesson.

Summary

Use the `Deque` data structure to implement a stack to support `push`, `pop` and `peek` operations.

Tasks

1. In NetBeans, open the `Stack07-02Prac` project.

    a. Select File > Open Project.

    b. Browse to `/home/oracle/labs/07-Generics_Collections/practices/practice2`

    c. Select `Stack07-02Prac` and click Open Project.

2. Expand the project directories.

3. Open `IntegerStack.java` in the editor and make the following changes:

4. Implement the `push()` method to add an Integer to the stack:

```
public void push(Integer element) {
        data.addFirst(element);
    }
```

5. Implement the `pop()` method that deletes an Integer from the top of the stack:

```
public Integer pop() {
    if(data.isEmpty())
    {
      System.out.print("Stack is empty");

    }
    return data.removeFirst();
    }
```

6. Implement the `peek` method():

```
public Integer peek() {
        return data.peekFirst();
    }
```

7. Override the `toString()` method:

```
public String toString() {
        return data.toString();
    }
```

8. Add a `main` method the class.

    a. Create an instance of the Stack Class:

```
IntegerStack stack = new IntegerStack();
```

    b. Perform various operations on the `stack` by invoking various methods:
        `push()`, `pop()` and `peek()`.

```java
public static void main(String[] args) {
        IntegerStack stack = new IntegerStack();
        for (int i = 0; i < 5; i++) {
            stack.push(i);
        }
        System.out.println("After pushing 5 elements: " +
stack);

        int element = stack.pop();
        System.out.println("Popped element = " + element);

        System.out.println("After popping 1 element : " +
stack);

        int top = stack.peek();
        System.out.println("Peeked element = " + top);
        System.out.println("After peeking 1 element : " +
stack);
    }
```

9. Run the `IntegerStack.java` class to ensure that your program produces the desired output.

```
run:
After pushing 5 elements: [4, 3, 2, 1, 0]
Popped element = 4
After popping 1 element : [3, 2, 1, 0]
Peeked element = 3
After peeking 1 element : [3, 2, 1, 0]
BUILD SUCCESSFUL (total time: 2 seconds)
```

Practices for Lesson 8: Collections Streams, and Filters

Chapter 8

Practice Overview

In these practices, you use lambda expressions to improve an application.

The RoboCall App

The RoboCall app is an application for automating the communication with groups of people. The app can contact individuals by phone, email, or regular mail. In this example, the app will be used to contact three groups of people.

- Drivers: Persons over the age of 16

- Draftees: Male persons between the ages of 18 and 25

- Pilots (specifically commercial pilots): Persons between the ages of 23 and 65

Person

The `Person` class creates the master list of persons you want to contact. The class uses the builder pattern to create new object. The following are some key parts of the class.

First, private fields for each Person are as follows:

Person.java

```
 9 public class Person {
10   private String givenName;
11   private String surName;
12   private int age;
13   private Gender gender;
14   private String eMail;
15   private String phone;
16   private String address;
17   private String city;
18   private String state;
19   private String code;
20
```

So these will be the fields that our application can search.

A static method is used to create a list of sample users. The code looks something like this:

Person.java

```
167    public static List<Person> createShortList(){
168      List<Person> people = new ArrayList<>();
169
170      people.add(
171        new Person.Builder()
172              .givenName("Bob")
173              .surName("Baker")
174              .age(21)
175              .gender(Gender.MALE)
176              .email("bob.baker@example.com")
177              .phoneNumber("201-121-4678")
178              .address("44 4th St")
179              .city("Smallville")
180              .state("KS")
```

```
181          .code("12333")
182          .build()
183      );
184
```

forEach

All collections have a new `forEach` method.

RoboCallTest06.java

```
 9 public class RoboCallTest06 {
10
11   public static void main(String[] args){
12
13      List<Person> pl = Person.createShortList();
14
15      System.out.println("\n=== Print List ===");
16      pl.forEach(p -> System.out.println(p));
17
18   }
19 }
```

Notice that the `forEach` takes a method reference or a lambda expression as a parameter. In the example, the `toString` method is called to print out each `Person` object. Some form of expression is needed to specify the output.

Stream and Filter

The following example shows how `stream()` and `filter()` methods are used with a collection in the RoboCall app.

RoboCallTest07.java

```
10 public class RoboCallTest07 {
11
12   public static void main(String[] args){
13
14      List<Person> pl = Person.createShortList();
15      RoboCall05 robo = new RoboCall05();
16
17      System.out.println("\n=== Calling all Drivers Lambda ===");
18      pl.stream()
19         .filter(p -> p.getAge() >= 23 && p.getAge() <= 65)
20         .forEach(p -> robo.roboCall(p));
21
22   }
23 }
```

The `stream` method creates a pipeline of immutable `Person` elements and access to methods that can perform actions on those elements. The `filter` method takes a lambda expression as a parameter and filters on the logical expression provide. This indicates that a `Predicate` is

the target type of the filter. The elements that meet the filter criteria are passed to the `forEach` method, which does a `roboCall` on matching elements.

The following example is functionally equivalent to the last. But in the case, the lambda expression is assigned to a variable, which is then passed to the stream and filter.

RoboCallTest08.java

```
10 public class RoboCallTest08 {
11
12   public static void main(String[] args){
13
14     List<Person> pl = Person.createShortList();
15     RoboCall05 robo = new RoboCall05();
16
17     // Predicates
18     Predicate<Person> allPilots =
19         p -> p.getAge() >= 23 && p.getAge() <= 65;
20
21     System.out.println("\n=== Calling all Drivers Variable ===");
22     pl.stream().filter(allPilots)
23         .forEach(p -> robo.roboCall(p));
24   }
25 }
```

Method References

In cases where a lambda expression just calls an instance method, a method reference can be used instead.

A03aMethodReference.java

```
 9 public class A03aMethodReference {
10
11     public static void main(String[] args) {
12
13         List<SalesTxn> tList = SalesTxn.createTxnList();
14
15         System.out.println("\n== CA Transations Lambda ==");
16         tList.stream()
17             .filter(t -> t.getState().equals(State.CA))
18             .forEach(t -> t.printSummary());
19
20         tList.stream()
21             .filter(t -> t.getState().equals(State.CA))
22             .forEach(SalesTxn::printSummary);
23     }
24 }
```

So lines 18 and 22 are essentially equivalent. Method reference syntax uses the class name followed by "::" and then the method name.

Chaining and Pipelines

The final example compares a compound lambda statement with a chained version using multiple filter methods.

A04IterationTest.java

```
 9 public class A04IterationTest {
10
11     public static void main(String[] args) {
12
13         List<SalesTxn> tList = SalesTxn.createTxnList();
14
15         System.out.println("\n== CA Transations for ACME ==");
16         tList.stream()
17             .filter(t -> t.getState().equals(State.CA) &&
18                 t.getBuyer().getName().equals("Acme Electronics"))
19             .forEach(SalesTxn::printSummary);
20
21         tList.stream()
22             .filter(t -> t.getState().equals(State.CA))
23             .filter(t -> t.getBuyerName()
24                 .equals("Acme Electronics"))
25             .forEach(SalesTxn::printSummary);
26     }
27 }
```

The two examples are essentially equivalent. The second example demonstrates how methods can be chained to possibly make the code a little easier to read. Both are examples of pipelines created by the stream method.

Practices for Lesson 8: Collections Streams, and Filters

Practice 8-1: Update RoboCall to use Streams

Overview

In this practice, you have been given an old email mailing list program named `RoboMail`. It is used to send emails or text messages to employees at your company. Refactor `RoboMail` so that it uses lambda expressions instead of anonymous inner classes.

Assumptions

You have completed the lecture and reviewed the overview for this practice.

Tasks

1. Open the `EmployeeSearch08-01Prac` project.

   - Select File > Open Project.
   - Browse to `/home/oracle/labs/08-CollectionsStreamsFilters /practices/practice1`.
   - Select `EmployeeSearch08-01Prac` and click Open Project.

2. Open the `RoboMail01.java` file and remove the `mail` and `text` methods. They are no longer needed since a `stream` will be used to filter the employees and a `forEach` will call the required communication task.

3. Open the `RoboMailTest01.java` file and review the code there.

4. Update `RoboMailTest01.java` to use `stream`, `filter`, and `forEach` to perform the mailing and texting tasks of the previous program.

5. Your program should continue to perform the following tasks to the following groups.

   - Email all sales executives using `stream`, `filter`, and `forEach`.
   - Text all sales executives using `stream`, `filter`, and `forEach`.
   - Email all sales employees older than 50 using `stream`, `filter`, and `forEach`.
   - Text all sales employees older than 50 using `stream`, `filter`, and `forEach`.

6. To mail or text a group in the `forEach` method, use a lambda expression for each task.

   a. Mail example: `p -> robo.roboMail(p)`
   b. Text example: `p -> robo.roboText(p)`

Your output should look similar to the following:

```
==== RoboMail 01

=== Sales Execs
Emailing: Betty Jones age 65 at betty.jones@example.com
Texting: Betty Jones age 65 at 211-33-1234

=== All Sales
Emailing: John Adams age 52 at john.adams@example.com
Emailing: Betty Jones age 65 at betty.jones@example.com
Texting: John Adams age 52 at 112-111-1111
Texting: Betty Jones age 65 at 211-33-1234
```

Practice 8-2: Mail Sales Executives using Method Chaining

Overview

In this practice, continue to work with the RoboMail app from the previous lesson.

Assumptions

You have completed the lecture and completed the previous practice.

Tasks

1. Open the `EmployeeSearch08-02Prac` project.

   - Select File > Open Project.
   - Browse to `/home/oracle/labs/08-CollectionsStreamsFilters /practices/practice2`.
   - Select `EmployeeSearch08-02Prac` and click Open Project.

2. Open the `RoboMailTest01.java` file and review the code there.

3. Update the `RoboMailTest01.java` file to mail all sales executives. Use two filter methods to select the recipients of the mail.

The output from the program should look similar to the following:

```
==== RoboMail 01

=== Sales Execs
Emailing: Betty Jones age 65 at betty.jones@example.com
```

Practice 8-3: Mail Sales Employees over 50 Using Method Chaining

Overview

In this practice, continue to work with the RoboMail app from the previous lesson.

Assumptions

You have completed the lecture and completed the previous practice.

Tasks

1. Open the `EmployeeSearch08-03Prac` project.

    - Select File > Open Project.

    - Browse to `/home/oracle/labs/08-CollectionsStreamsFilters /practices/practice3`.

    - Select `EmployeeSearch08-03Prac` and click Open Project.

2. Open the `RoboMailTest01.java` file and review the code there.

3. Update the `RoboMailTest01.java` file to mail all sales employees over 50. Use two filter methods to select the recipients of the mail.

The output from the program should look similar to the following:

```
==== RoboMail 01

=== All Sales 50+
Emailing: John Adams age 52 at john.adams@example.com
Emailing: Betty Jones age 65 at betty.jones@example.com
```

Practice 8-4: Mail Male Engineering Employees Under 65 Using Method Chaining

Overview

In this practice, continue to work with the RoboMail app from the previous lesson.

Assumptions

You have completed the lecture and completed the previous practice.

Tasks

1. Open the `EmployeeSearch08-04Prac` project.

    - Select File > Open Project.

    - Browse to `/home/oracle/labs/08-CollectionsStreamsFilters` `/practices/practice4`.

    - Select `EmployeeSearch08-04Prac` and click Open Project.

2. Open the `RoboMailTest01.java` file and review the code there.

3. Update the `RoboMailTest01.java` file to mail all male engineering employees under 65. Use three filter methods to select the recipients of the mail.

The output from the program should look similar to the following:

```
==== RoboMail 01

=== Male Eng Under 65
Emailing: James Johnson age 45 at james.johnson@example.com
Emailing: Joe Bailey age 62 at joebob.bailey@example.com
```

Practices for Lesson 9: Lambda Built-in Functional Interfaces

Chapter 9

Practice Overview

In these practices, create lambda expressions using the built-in functional interfaces found in the java.util.function package.

The focus of this lesson and examples is to make you familiar with the built-in functional interfaces for use with lambda expressions. They are often used as parameters for method calls with streams. Familiarity with these interfaces makes working with streams much easier.

Predicate

The Predicate interface has already been covered in the last lesson. Essentially, it is a lambda expression that takes a generic type and returns a boolean.

A01Predicate.java

```
10 public class A01Predicate {
11
12   public static void main(String[] args){
13
14     List<SalesTxn> tList = SalesTxn.createTxnList();
15
16     Predicate<SalesTxn> massSales =
17         t -> t.getState().equals(State.MA);
18
19     System.out.println("\n== Sales - Stream");
20     tList.stream()
21         .filter(massSales)
22         .forEach(t -> t.printSummary());
23
24     System.out.println("\n== Sales - Method Call");
25     for(SalesTxn t:tList){
26         if (massSales.test(t)){
27             t.printSummary();
28         }
29     }
30   }
31 }
```

In the preceding code, the lambda expression is used in a filter for a stream. The second example also shows that the test method can be executed on any SalesTxn element using the functional interface that stores the Predicate.

To repeat, a Predicate takes in a generic type and returns a boolean.

Consumer

The `Consumer` interface specifies a generic type but returns nothing. Essentially, it is a `void` return type for lambdas. In the following example, the lambda expression specifies how a transaction should be printed.

A02Consumer.java

```
10 public class A02Consumer {
11
12   public static void main(String[] args){
13
14     List<SalesTxn> tList = SalesTxn.createTxnList();
15     SalesTxn first = tList.get(0);
16
17     Consumer<SalesTxn> buyerConsumer = t ->
18         System.out.println("Id: " + t.getTxnId()
19             + " Buyer: " + t.getBuyerName());
20
21     System.out.println("== Buyers - Lambda");
22     tList.stream().forEach(buyerConsumer);
23
24     System.out.println("== First Buyer - Method");
25     buyerConsumer.accept(first);
26   }
27 }
```

For the `forEach` method, the default argument is a `Consumer`. The lambda expression is basically just a print statement that is used in the two cases shown. In the second example, the `accept` method is called along with a transaction. This prints the first transaction in the list.

The key point here is that the `Consumer` takes a generic type and returns nothing. It is essentially a `void` return type for lambda expressions.

Function

The `Function` interface specifies two generic object types to be used in the expression. The first generic object is used in the lambda expression and the second is the return type from the lambda expression. The example uses a `SalesTxn` to return a `String`.

A03Function.java

```
10 public class A03Function {
11
12   public static void main(String[] args){
13
14     List<SalesTxn> tList = SalesTxn.createTxnList();
15     SalesTxn first = tList.get(0);
16
17     Function<SalesTxn, String> buyerFunction =
18         t -> t.getBuyerName();
19
20     System.out.println("\n== First Buyer");
21     System.out.println(buyerFunction.apply(first));
```

Practices for Lesson 9: Lambda Built-in Functional Interfaces

The `Function` has one method named `apply`. In this example, a `String` is returned to the print statement.

With a `Function` the key concept is that a `Function` takes in one type and returns another.

Supplier

The `Supplier` interface specifies one generic type, which is returned from the lambda expression. Nothing is passed in so this is similar to a `Factory`. The follow expression example creates and returns a `SalesTxn` and adds it to our existing list.

A04Supplier.java

```
13    public static void main(String[] args){
14
15      List<SalesTxn> tList = SalesTxn.createTxnList();
16      Supplier<SalesTxn> txnSupplier =
17         () -> new SalesTxn.Builder()
18            .txnId(101)
19            .salesPerson("John Adams")
20            .buyer(Buyer.getBuyerMap().get("PriceCo"))
21            .product("Widget")
22            .paymentType("Cash")
23            .unitPrice(20)
24            .unitCount(8000)
25            .txnDate(LocalDate.of(2013,11,10))
26            .city("Boston")
27            .state(State.MA)
28            .code("02108")
29            .build();
30
31      tList.add(txnSupplier.get());
32      System.out.println("\n== TList");
33      tList.stream().forEach(SalesTxn::printSummary);
34    }
```

Notice a `Supplier` has no input arguments, there is merely empty parentheses: `() ->`. The example uses a builder to create a new object. Notice Supplier has only one method `get`, which in this case returns a `SalesTxn`.

The key take away with a `Supplier` is that it has no input parameters but returns a generic type.

So that pretty much covers the basic function interfaces. However, there are a lot of variations.

Primitive Types - ToDoubleFunction and AutoBoxing

There are primitive versions of all the built-in lambda functional interfaces. The following code shows an example of the ToDoubleFunction interface.

A05PrimFunction.java

```
11 public class A05PrimFunction {
12
13   public static void main(String[] args){
14
15     List<SalesTxn> tList = SalesTxn.createTxnList();
16     SalesTxn first = tList.get(0);
17
18     ToDoubleFunction<SalesTxn> discountFunction =
19         t -> t.getTransactionTotal()
20             * t.getDiscountRate();
21
22     System.out.println("\n== Discount");
23     System.out.println(
24         discountFunction.applyAsDouble(first));
25
```

Remember a Function takes in one generic and return a different generic. However, the ToDoubleFunction interface has only one generic specified. That is because it takes a generic type as input and returns a double. Notice also that the method name for this functional interface is applyAsDouble. So to repeat, the ToDoubleFunction takes in a generic and returns a double. There are also long and int versions of this interface.

Why create these primitive variations? Consider this piece of code.

A05PrimFunction.java

```
26     // What's wrong here?
27     Function<SalesTxn, Double> taxFunction =
28         t -> t.getTransactionTotal() * t.getTaxRate();
29     double tax = taxFunction.apply(first); // What happerns here?
30   }
31 }
```

With object types, this would require the autoboxing and unboxing of primitive values. Not good for performance. These specialized primitive interfaces address this issue and allow for operations on primitive types.

Primitive Types -– DoubleFunction

What if you need to pass in a primitive to a lambda expression? Well, the DoubleFunction interface is a great example of that.

A06DoubleFunction.java

```
 5 public class A06DoubleFunction {
 6
 7   public static void main(String[] args) {
 8
 9     A06DoubleFunction test = new A06DoubleFunction();
10
11     DoubleFunction<String> calc =
12         t -> String.valueOf(t * 3);
```

Practices for Lesson 9: Lambda Built-in Functional Interfaces

```
13
14     String result = calc.apply(20);
15     System.out.println("New value is: " + result);
16   }
17 }
```

Primitive interfaces like `DoubleFunction`, `IntFunction`, or `LongFunction` take a primitive as input and return a generic type. In this case, a double is passed to the lambda expression and a String is returned. Once again, this avoids any boxing issues.

Binary Intefaces – BiPredicate

A number of examples having the `Predicate` interface have been explored so far in this course. A `Predicate` takes a generic class and returns a `boolean`. But what if you want to compare two things? There is a binary specialization for that.

The `BiPredicate` interface allows two object types to be used in a lambda expression. Binary interfaces for the other main interface types are also available.

A07Binary.java

```
10 public class A07Binary {
11
12   public static void main(String[] args){
13
14     List<SalesTxn> tList = SalesTxn.createTxnList();
15     SalesTxn first = tList.get(0);
16     String testState = "CA";
17
18     BiPredicate<SalesTxn,String> stateBiPred =
19       (t, s) -> t.getState().equals(State.CA);
20
21     System.out.println("\n== First in CA?");
22     System.out.println(
23       stateBiPred.test(first, testState));
24   }
25 }
```

The example specifies a `SalesTxn` and a `String` as the generic types used in the lambda expression. Note that the types are specified with `t` and `s` and a `boolean` is still returned. It is the same result as a `Predicate`, but with two input types.

UnaryOperator

The `Function` interface takes in one generic and returns a different generic. What if you want to return the same thing? Then the `UnaryOperator` interface is what you need.

A08Unary.java

```
10 public class A08Unary {
11
12   public static void main(String[] args){
13
14     List<SalesTxn> tList = SalesTxn.createTxnList();
15     SalesTxn first = tList.get(0);
16
17     UnaryOperator<String> unaryStr =
18         s -> s.toUpperCase();
```

```
19
20       System.out.println("== Upper Buyer");
21       System.out.println(
22           unaryStr.apply(first.getBuyerName()));
23   }
24 }
```

The example takes a `String` and returns an uppercase version of that `String`.

API Docs

As a reminder, it is difficult to remember all the variations of functional interfaces and what they do. Make liberal use of the API docs to remember your options or what is returned for the `java.util.function` package.

Practice 9-1: Create Consumer Lambda Expression

Overview

In this practice, create a `Consumer` lambda expression to print out employee data.

Note that `salary` and `startDate` fields were added to the Employee class. In addition, enumerations are included for `Bonus` and `VacAccrual`. The enums allow calculations for bonuses and vacation time.

Assumptions

You have completed the lecture portion of the course.

Tasks

1. Open the `EmployeeSearch09-01Prac` project.

    - Select File > Open Project.
    - Browse to `/home/oracle/labs/09-LambdaBuiltIns/practices/practice1`.
    - Select `EmployeeSearch09-01Prac` and click Open Project.

2. Open the `Employee.java` file and become familiar with the code included in the file.

3. Open the `ConsumerTest.java` file and make the following updates.

4. Write a `Consumer` lambda expression to print data about the first employee in the list.

    a. The data printed should be the following: `"Name: " + e.getSurName() + " Role: " + e.getRole() + " Salary: " + e.getSalary()`

5. Write a statement to execute the lambda expression on the `first` variable.

6. Your output should look similar to the following:

```
=== First Salary
Name: Baker   Role: STAFF   Salary: 40000.0
```

Practice 9-2: Create a Function Lambda Expression

Overview

In this practice, create a `ToDoubleFunction` lambda expression to calculate an employee bonus.

Assumptions

You have completed the lecture portion of the course and the previous practice.

Tasks

1. Open the `EmployeeSearch09-02Prac` project.

   - Select File > Open Project.
   - Browse to `/home/oracle/labs/09-LambdaBuiltIns/practices/practice2`.
   - Select `EmployeeSearch09-02Prac` and click Open Project.

2. Open the `Bonus.java` file and review the code included in the file.

3. Open the `FunctionTest.java` file and make the following updates.

4. Write a `ToDoubleFunction` lambda expression to calculate the bonus for the first employee in the list.

   a. The bonus can be calculated as follows: `e.getSalary() * Bonus.byRole(e.getRole())`

5. Write a statement to execute the lambda expression on the `first` variable.

6. Your output should look similar to the following:

```
=== First Employee Bonus
Name: Bob Baker Role: STAFF Dept: ENG eMail: bob.baker@example.com
Salary: 40000.0
Bonus: 800.0
```

Practice 9-3: Create a Supplier Lambda Expression

Overview

In this practice, create a `Supplier` lambda expression to add a new employee to the employee list.

Assumptions

You have completed the lecture portion of the course and the previous practice.

Tasks

1. Open the `EmployeeSearch09-03Prac` project.

   - Select File > Open Project.

   - Browse to `/home/oracle/labs/09-LambdaBuiltIns/practices/practice3`.

   - Select `EmployeeSearch09-03Prac` and click Open Project.

2. Open the `SupplierTest.java` file and make the following updates.

3. Write a `Supplier` lambda expression to add a new employee to the list. The employee data is as follows:

   Given name: Jill

   SurName: Doe

   Age: 26

   Gender: Gender.FEMALE

   Role: Role.STAFF

   Dept: Sales

   StartDate: LocalDate.of(2012, 7, 14)

   Salary: 45000

   Email: jill.doe@example.com

   PhoneNumber: 202-123-4678

   Address: 33 3rd St

   City: Smallville

   State: KS

   Code: 12333

   Hint: Her data is almost exactly the same as her sister Jane and can be found in the `Employee.java` file.

4. Write a statement to add the new employee to the employee list.

5. Your output should look similar to the following after adding the new employee to the list:

```
=== Print employee list after
Name: Bob Baker Role: STAFF Dept: ENG eMail: bob.baker@example.com
Salary: 40000.0
Name: Jane Doe Role: STAFF Dept: Sales eMail: jane.doe@example.com
Salary: 45000.0
Name: John Doe Role: MANAGER Dept: Eng eMail: john.doe@example.com
Salary: 65000.0
Name: James Johnson Role: MANAGER Dept: Eng eMail:
james.johnson@example.com Salary: 85000.0
Name: John Adams Role: MANAGER Dept: Sales eMail:
john.adams@example.com Salary: 90000.0
Name: Joe Bailey Role: EXECUTIVE Dept: Eng eMail:
joebob.bailey@example.com Salary: 120000.0
Name: Phil Smith Role: EXECUTIVE Dept: HR eMail:
phil.smith@examp;e.com Salary: 110000.0
Name: Betty Jones Role: EXECUTIVE Dept: Sales eMail:
betty.jones@example.com Salary: 140000.0
Name: Jill Doe Role: STAFF Dept: Sales eMail: jill.doe@example.com
Salary: 45000.0
```

Practices for Lesson 9: Lambda Built-in Functional Interfaces

Practice 9-4: Create a BiPredicate Lambda Expression

Overview

In this practice, create a `BiPredicate` lambda expression to calculate an employee bonus.

Assumptions

You have completed the lecture portion of the course and the previous practice.

Tasks

1. Open the `EmployeeSearch09-04Prac` project.

    - Select File > Open Project.
    - Browse to `/home/oracle/labs/09-LambdaBuiltIns/practices/practice4`.
    - Select `EmployeeSearch09-04Prac` and click Open Project.

2. Open the `BiPredicateTest.java` file and make the following updates.

3. Write a `BiPredicate` lambda expression to compare a field in the employee class to a string.

    a. The `searchState` variable should be compared to the state value in the employee element.

4. Write an expression to perform the logical test in the `for` loop.

5. Your output should look similar to the following:

```
=== Print matching list
Name: Bob Baker Role: STAFF Dept: ENG eMail: bob.baker@example.com
Salary: 40000.0
Name: Jane Doe Role: STAFF Dept: Sales eMail: jane.doe@example.com
Salary: 45000.0
Name: John Doe Role: MANAGER Dept: Eng eMail: john.doe@example.com
Salary: 65000.0
```

Practices for Lesson 10: Lambda Operations

Chapter 10

Practices for Lesson 10: Lambda Operations

Practices for Lesson 10: Overview

Practice Overview

In these practices, create lambda expressions and streams to process data in collections.

Employee List

Here is a short list of Employees and their data that will be used for the examples that follow.

```
Name: Bob Baker Role: STAFF Dept: Eng St: KS Salary: $40,000.00
Name: Jane Doe Role: STAFF Dept: Sales St: KS Salary: $45,000.00
Name: John Doe Role: MANAGER Dept: Eng St: KS Salary: $65,000.00
Name: James Johnson Role: MANAGER Dept: Eng St: MA Salary: $85,000.00
Name: John Adams Role: MANAGER Dept: Sales St: MA Salary: $90,000.00
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
```

Map

The `map` method in the `Stream` class allows you to extract a field from a stream and perform some operation or calculation on that value. The resulting values are then passed to the next stream in the pipeline.

A01MapTest.java

```
 9 public class A01MapTest {
10
11     public static void main(String[] args) {
12
13         List<Employee> eList = Employee.createShortList();
14
15         System.out.println("\n== CO Bonuses ==");
16         eList.stream()
17             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18             .filter(e -> e.getState().equals("CO"))
19             .map(e -> e.getSalary() * Bonus.byRole(e.getRole()))
20             .forEach( s -> System.out.printf("Bonus paid: $%,6.2f %n", s));
21
```

The example prints out the bonuses for two different groups. The `filter` methods select the groups and then `map` is used to compute a result.

Output

```
== CO Bonuses ==
Bonus paid: $7,200.00
Bonus paid: $6,600.00
Bonus paid: $8,400.00
```

Peek

The `peek` method of the `Stream` class allows you to perform an operation on an element in the stream. The elements are returned to the stream and are available to the next stream in the pipeline. The `peek` method can be used to read or change data in the stream. Any changes will be made to the underlying collection.

A02MapPeekTest.java

```
15          System.out.println("\n== CO Bonuses ==");
16          eList.stream()
17              .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18              .filter(e -> e.getState().equals("CO"))
19              .peek(e -> System.out.print("Name: "
20                + e.getGivenName() + " " + e.getSurName()))
21              .map(e -> e.getSalary() * Bonus.byRole(e.getRole()))
22              .forEach( s ->
23                System.out.printf(
24                  "  Bonus paid: $%,6.2f %n", s));
```

In this example, after filtering the data, `peek` is used to print data from the current stream to the console. After the `map` method is called, only the data returned from `map` is available for output.

Output

```
== CO Bonuses ==
Name: Joe Bailey   Bonus paid: $7,200.00
Name: Phil Smith   Bonus paid: $6,600.00
Name: Betty Jones  Bonus paid: $8,400.00
```

Find First

The `findFirst` method of the `Stream` class finds the first element in the stream specified by the filters in the pipeline. The `findFirst` method is a terminal short-circuit operation. This means intermediate operations are performed in a lazy manner resulting in more efficient processing of the data in the stream. A terminal operation ends the processing of a pipeline.

A03FindFirst.java

```
10 public class A03FindFirst {
11
12     public static void main(String[] args) {
13
14         List<Employee> eList = Employee.createShortList();
15
16         System.out.println("\n== First CO Bonus ==");
17         Optional<Employee> result;
18
19         result = eList.stream()
20             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
21             .filter(e -> e.getState().equals("CO"))
22             .findFirst();
23
24         if (result.isPresent()){
25             result.get().print();
26         }
27
28     }
```

The code filters the pipeline for executives in the state of Colorado. The first element in the collection that meets this criterion is returned and printed out. Notice that the type of the result variable is `Optional<Employee>`. This is a new class that allows you to determine if a value is present before trying to retrieve a result. This has advantages for concurrent applications.

Practices for Lesson 10: Lambda Operations

Output

```
== First CO Bonus ==

Name: Joe Bailey
Age: 62
Gender: MALE
Role: EXECUTIVE
Dept: Eng
Start date: 1992-01-05
Salary: 120000.0
eMail: joebob.bailey@example.com
Phone: 112-111-1111
Address: 111 1st St
City: Town
State: CO
Code: 11111
```

Find First Lazy

The following example compares a pipeline, which filters and iterates through an entire collection to a pipeline with a short-circuit terminal operation (findFirst). The peek method is used to print out a message associated with each operation.

A04FindFirstLazy.java

```
10 public class A04FindFirstLazy {
11
12     public static void main(String[] args) {
13
14         List<Employee> eList = Employee.createShortList();
15
16         System.out.println("\n== CO Bonuses ==");
17         eList.stream()
18             .peek(e -> System.out.println("Stream start"))
19             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
20             .peek(e -> System.out.println("Executives"))
21             .filter(e -> e.getState().equals("CO"))
22             .peek(e -> System.out.println("CO Executives"))
23             .map(e -> e.getSalary() * Bonus.byRole(e.getRole()))
24             .forEach( s -> System.out.printf(
25                 "  Bonus paid: $%,6.2f %n", s));
26
27         System.out.println("\n== First CO Bonus ==");
28         Employee tempEmp = new Employee.Builder().build();
29         Optional<Employee> result = eList.stream()
30             .peek(e -> System.out.println("Stream start"))
31             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
32             .peek(e -> System.out.println("Executives"))
33             .filter(e -> e.getState().equals("CO"))
34             .peek(e -> System.out.println("CO Executives"))
35             .findFirst();
36
37         if (result.isPresent()){
```

Practices for Lesson 10: Lambda Operations

```
38              result.get().printSummary();
39          }
40      }
41 }
```

The pipeline prints out 17 different options. The second, with a short-circuit operator, prints 8. This demonstrates how lazy operations can really improve the performance of iteration through a collection.

Output

```
== CO Bonuses ==
Stream start
Stream start
Stream start
Stream start
Stream start
Stream start
Executives
CO Executives
  Bonus paid: $7,200.00
Stream start
Executives
CO Executives
  Bonus paid: $6,600.00
Stream start
Executives
CO Executives
  Bonus paid: $8,400.00

== First CO Bonus ==
Stream start
Stream start
Stream start
Stream start
Stream start
Stream start
Executives
CO Executives
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary:
$120,000.00
```

anyMatch

The anyMatch method returns a boolean based on the specified Predicate. This is a short-circuiting terminal operation.

A05AnyMatch.java

```
10 public class A05AnyMatch {
11
12      public static void main(String[] args) {
13
14          List<Employee> eList = Employee.createShortList();
15
16          System.out.println("\n== First CO Bonus ==");
```

Practices for Lesson 10: Lambda Operations

```
17        Optional<Employee> result;
18
19        if (eList.stream().anyMatch(
20            e -> e.getState().equals("CO"))){
21
22          result = eList.stream()
23             .peek(e -> System.out.println("Stream"))
24             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
25             .filter(e -> e.getState().equals("CO"))
26             .findFirst();
27
28          if (result.isPresent()){result.get().printSummary();}
29        }
```

The example shows how the `anyMatch` method could be used to check for a value before executing a more detailed query.

Count

The `count` method returns the number of elements in the current stream. This is a terminal operation.

A06StreamData.java

```
15     List<Employee> eList = Employee.createShortList();
16
17     System.out.println("\n== Executive Count ==");
18     long execCount =
19        eList.stream()
20           .filter(e -> e.getRole().equals(Role.EXECUTIVE))
21           .count();
22
23     System.out.println("Exec count: " + execCount);
```

The example returns the number of executives in Colorado and prints the result.

Output

```
== Executive Count ==
Exec count: 3
```

Max

The `max` method returns the highest matching value given a `Comparator` to rank elements. The `max` method is a terminal operation.

A06StreamData.java

```
23     System.out.println("Exec count: " + execCount);
24
25     System.out.println("\n== Highest Paid Exec ==");
26     Optional highestExec =
27        eList.stream()
28           .filter(e -> e.getRole().equals(Role.EXECUTIVE))
29           .max(Employee::sortBySalary);
30
31     if (highestExec.isPresent()){
```

Practices for Lesson 10: Lambda Operations

```
32        Employee temp = (Employee) highestExec.get();
33        System.out.printf(
34            "Name: " + temp.getGivenName() + " "
35            + temp.getSurName() + "   Salary: $%,6.2f %n ",
36            temp.getSalary());
37     }
```

The example shows `max` being used with a `Comparator` that has been written for the class. The `sortBySalary` method is called using a method reference. Notice the return type of `Optional`. This is not the generic version used in previous examples. Therefore, a cast is required when the object is retrieved.

Output

```
== Highest Paid Exec ==
Name: Betty Jones    Salary: $140,000.00
```

Min

The `min` method returns the lowest matching value given a `Comparator` to rank elements. The `min` method is a terminal operation.

A06StreamData.java

```
39        System.out.println("\n== Lowest Paid Staff ==");
40        Optional lowestStaff =
41            eList.stream()
42            .filter(e -> e.getRole().equals(Role.STAFF))
43            .min(Comparator.comparingDouble(e -> e.getSalary()));
44
45        if (lowestStaff.isPresent()){
46           Employee temp = (Employee) lowestStaff.get();
47           System.out.printf("Name: " + temp.getGivenName()
48            + " " + temp.getSurName() +
49            "   Salary: $%,6.2f %n ", temp.getSalary());
50        }
```

In this example, a different `Comparator` is used. The `comparingDouble` static method is called to make the comparison. Notice that the example uses a lambda expression to specify the comparison field. If you look at the code closely, a method reference could be substituted instead: `Employee::getSalary`. More discussion on this subject follows in the `Comparator` section.

Output

```
== Lowest Paid Staff ==
Name: Bob Baker    Salary: $40,000.00
```

Sum

The `sum` method calculates a sum based on the stream passed to it. Notice the `mapToDouble` method is called before the stream is passed to `sum`. If you look at the `Stream` class, no `sum` method is included. Instead, a `sum` method is included in the primitive version of the `Stream` class, `IntStream`, `DoubleStream`, and `LongStream`. The `sum` method is a terminal operation.

A07CalcSum.java

```
26      System.out.println("\n== Total CO Bonus Details ==");
27
28      result = eList.stream()
29        .filter(e -> e.getRole().equals(Role.EXECUTIVE))
30        .filter(e -> e.getState().equals("CO"))
31        .peek(e -> System.out.print("Name: "
32          + e.getGivenName() + " " + e.getSurName() + " "))
33        .mapToDouble(e -> e.getSalary() * Bonus.byRole(e.getRole()))
34        .peek(d -> System.out.printf("Bonus paid: $%,6.2f %n", d))
35        .sum();
36
37      System.out.printf("Total Bonuses paid: $%,6.2f %n", result);
```

Looking at the example, can you tell the type of `result`? If the API documentation is examined, the `mapToDouble` method returns a `DoubleStream`. The `sum` method for `DoubleStream` returns a `double`. Therefore, the result variable must be a `double`.

Output

```
== Total CO Bonus Details ==
Name: Joe Bailey Bonus paid: $7,200.00
Name: Phil Smith Bonus paid: $6,600.00
Name: Betty Jones Bonus paid: $8,400.00
Total Bonuses paid: $22,200.00
```

Average

The `average` method returns the average of a list of values passed from a stream. The `avg` method is a terminal operation.

A08CalcAvg.java

```
28      System.out.println("\n== Average CO Bonus Details ==");
29
30      result = eList.stream()
31        .filter(e -> e.getRole().equals(Role.EXECUTIVE))
32        .filter(e -> e.getState().equals("CO"))
33        .peek(e -> System.out.print("Name: " + e.getGivenName()
34          + " " + e.getSurName() + " "))
35        .mapToDouble(e -> e.getSalary() * Bonus.byRole(e.getRole()))
36        .peek(d -> System.out.printf("Bonus paid: $%,6.2f %n", d))
37        .average();
38
39      if (result.isPresent()){
40        System.out.printf("Average Bonuses paid: $%,6.2f %n",
41          result.getAsDouble());
42      }
43    }
```

Once again, the return type for `avg` can be inferred from the code shown in this example. Note the check for `isPresent()` in the if statement and the call to `getAsDouble()`. In this case an `OptionalDouble` is returned.

Output

```
== Average CO Bonus Details ==
Name: Joe Bailey Bonus paid: $7,200.00
Name: Phil Smith Bonus paid: $6,600.00
```

```
Name: Betty Jones Bonus paid: $8,400.00
Average Bonuses paid: $7,400.00
```

Sorted

The sorted method can be used to sort stream elements based on their natural order. This is an intermediate operation.

A09SortBonus.java

```
10 public class A09SortBonus {
11   public static void main(String[] args) {
12     List<Employee> eList = Employee.createShortList();
13
14     System.out.println("\n== CO Bonus Details ==");
15
16     eList.stream()
17       .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18       .filter(e -> e.getState().equals("CO"))
19       .mapToDouble(e -> e.getSalary() * Bonus.byRole(e.getRole()))
20       .sorted()
21       .forEach(d -> System.out.printf("Bonus paid: $%,6.2f %n", d));
```

In this example, the bonus is computed and those values are used to sort the results. So a list for `double` values is sorted and printed out.

Output

```
== CO Bonus Details ==
Bonus paid: $6,600.00
Bonus paid: $7,200.00
Bonus paid: $8,400.00
```

Sorted with Comparator

The `sorted` method can also take a `Comparator` as a parameter. Combined with the `comparing` method, the `Comparator` class provides a great deal of flexibility when sorting a stream.

A10SortComparator.java

```
11 public class A10SortComparator {
12   public static void main(String[] args) {
13     List<Employee> eList = Employee.createShortList();
14
15     System.out.println("\n== CO Bonus Details Comparator ==");
16
17     eList.stream()
18       .filter(e -> e.getRole().equals(Role.EXECUTIVE))
19       .filter(e -> e.getState().equals("CO"))
20       .sorted(Comparator.comparing(Employee::getSurName))
21       .forEach(Employee::printSummary);
```

In this example, notice on line 20 that a method reference is passed to the `comparing` method. In this case, the stream is sorted by surname. However, clearly the implication is any of the `get` methods from the `Employee` class could be passed to this method. So with one simple expression, a stream can be sorted by any available field.

Practices for Lesson 10: Lambda Operations

Output

```
== CO Bonus Details Comparator ==
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
```

Reversed

The reversed method can be appended to the comparing method thus reversing the sort order of the elements in the stream. The example and output demonstrate this using surname.

A10SortComparator.java

```
23      System.out.println("\n== CO Bonus Details Reversed ==");
24
25      eList.stream()
26        .filter(e -> e.getRole().equals(Role.EXECUTIVE))
27        .filter(e -> e.getState().equals("CO"))
28        .sorted(Comparator.comparing(Employee::getSurName).reversed())
29        .forEach(Employee::printSummary);
```

Output

```
== CO Bonus Details Reversed ==
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
```

Two Level Sort

In this example, the thenComparing method has been added to the comparing method. This allows you to do a multilevel sort on the elements in the stream. The thenComparing method takes a Comparator as a parameter just like the comparing method.

A10SortComparator.java

```
31      System.out.println("\n== Two Level Sort, Dept then Surname ==");
32
33      eList.stream()
34        .sorted(
35          Comparator.comparing(Employee::getDept)
36            .thenComparing(Employee::getSurName))
37        .forEach(Employee::printSummary);
```

In the example, the stream is sorted by department and then by surname. The output is as follows.

Output

```
== Two Level Sort, Dept then Surname ==
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Bob Baker Role: STAFF Dept: Eng St: KS Salary: $40,000.00
Name: John Doe Role: MANAGER Dept: Eng St: KS Salary: $65,000.00
Name: James Johnson Role: MANAGER Dept: Eng St: MA Salary: $85,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: John Adams Role: MANAGER Dept: Sales St: MA Salary: $90,000.00
Name: Jane Doe Role: STAFF Dept: Sales St: KS Salary: $45,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
```

Practices for Lesson 10: Lambda Operations

Collect

The `collect` method allows you to save the results of all the filtering, mapping, and sorting that takes place in a pipeline. Notice how the `collect` method is called. It takes a `Collectors` class as a parameter. The `Collectors` class provides a number of ways to return the elements left in a pipeline.

A11Collect.java

```
12 public class A11Collect {
13
14     public static void main(String[] args) {
15
16         List<Employee> eList = Employee.createShortList();
17
18         List<Employee> nList = new ArrayList<>();
19
20         // Collect CO Executives
21         nList = eList.stream()
22             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
23             .filter(e -> e.getState().equals("CO"))
24             .sorted(Comparator.comparing(Employee::getSurName))
25             .collect(Collectors.toList());
26
27         System.out.println("\n== CO Bonus Details ==");
28
29         nList.stream()
30             .forEach(Employee::printSummary);
31
32     }
33
34 }
```

In this example, the `Collectors` class simply returns a new `List`, which consists of the elements selected by the filter methods. In addition to a `List`, a `Set` or a `Map` may be returned as well. Plus there are a number of other options to save the pipeline results. Below are the three `Employee` elements that match the filter criteria in sorted order.

Output

```
== CO Bonus Details ==
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
```

Collectors and Math

The `Collectors` class includes a number of math methods including `averagingDouble` and `summingDouble` along with other primitive versions.

A12CollectMath.java

```
12 public class A12CollectMath {
13
14     public static void main(String[] args) {
15
16         List<Employee> eList = Employee.createShortList();
```

```
17
18          // Collect CO Executives
19          double avgSalary = eList.stream()
20               .filter(e -> e.getRole().equals(Role.EXECUTIVE))
21               .filter(e -> e.getState().equals("CO"))
22               .collect(
23                    Collectors.averagingDouble(Employee::getSalary));
24
25          System.out.println("\n== CO Exec Avg Salary ==");
26          System.out.printf("Average: $%,9.2f %n", avgSalary);
27
28      }
29
30 }
```

In this example, an average salary is computed based on the filters provided. A double `primitive` value is returned.

Output

```
== CO Exec Avg Salary ==
Average: $123,333.33
```

Collectors and Joining

The `joining` method of the `Collectors` class allows you to join together elements returned from a stream.

A13CollectJoin.java

```
12 public class A13CollectJoin {
13
14      public static void main(String[] args) {
15
16          List<Employee> eList = Employee.createShortList();
17
18          // Collect CO Executives
19          String deptList = eList.stream()
20               .map(Employee::getDept)
21               .distinct()
22               .collect(Collectors.joining(", "));
23
24          System.out.println("\n== Dept List ==");
25          System.out.println("Total: " + deptList);
26
27      }
28
29 }
```

In this example, the values for department are extracted from the stream using a `map`. A call is made to the `distinct` method, which removes any duplicate values. The resulting values are joined together using the `joining` method. The output is shown in the following.

Output

```
== Dept List ==
Total: Eng, Sales, HR
```

Practices for Lesson 10: Lambda Operations

Collectors and Grouping

The groupingBy method of the Collectors class allows you to generate a Map based on the elements contained in a stream.

A14CollectGrouping.java

```
12 public class A14CollectGrouping {
13
14     public static void main(String[] args) {
15
16         List<Employee> eList = Employee.createShortList();
17
18         Map<String, List<Employee>> gMap = new HashMap<>();
19
20         // Collect CO Executives
21         gMap = eList.stream()
22             .collect(Collectors.groupingBy(Employee::getDept));
23
24         System.out.println("\n== Employees by Dept ==");
25         gMap.forEach((k,v) -> {
26             System.out.println("\nDept: " + k);
27             v.forEach(Employee::printSummary);
28         });
29
30     }
31
32 }
```

In this example, the groupingBy method is called with a method reference to getDept. This created a Map with the department names used as key and a list of elements that match that key become the value for the Map. Notice how the Map is specified on line 18. In addition, starting on line 25 the code iterates through the resulting Map. The output from the Map is shown in the following.

Output

```
== Employees by Dept ==

Dept: Sales
Name: Jane Doe Role: STAFF Dept: Sales St: KS Salary: $45,000.00
Name: John Adams Role: MANAGER Dept: Sales St: MA Salary: $90,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00

Dept: HR
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00

Dept: Eng
Name: Bob Baker Role: STAFF Dept: Eng St: KS Salary: $40,000.00
Name: John Doe Role: MANAGER Dept: Eng St: KS Salary: $65,000.00
Name: James Johnson Role: MANAGER Dept: Eng St: MA Salary: $85,000.00
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
```

Collectors, Grouping, and Counting

Another version of the `groupingBy` function takes a `Function` and `Collector` as parameters and returns a `Map`. This example builds on the last and instead of returning matching elements, it counts them.

A15CollectCount.java

```
12 public class A15CollectCount {
13
14     public static void main(String[] args) {
15
16         List<Employee> eList = Employee.createShortList();
17
18         Map<String, Long> gMap = new HashMap<>();
19
20         // Collect CO Executives
21         gMap = eList.stream()
22             .collect(
23                 Collectors.groupingBy(
24                     e -> e.getDept(), Collectors.counting()));
25
26         System.out.println("\n== Employees by Dept ==");
27         gMap.forEach((k,v) ->
28             System.out.println("Dept: " + k + " Count: " + v)
29         );
30
31     }
32
33 }
```

Note how the method once again creates the `Map` based on department. But this time, `Collectors.counting` is used to return `long` values to the `Map`. The output from the `Map` is shown in the following.

Output

```
== Employees by Dept ==
Dept: Sales Count: 3
Dept: HR Count: 1
Dept: Eng Count: 4
```

Collectors and Partitioning

The `partitioningBy` method offers an interesting way to create a `Map`. The method takes a `Predicate` as an argument and creates a `Map` with two `Boolean` keys. One key is `true` and includes all the elements that met the true criteria of the Predicate. The other key, `false`, contains all the elements that resulted in false values as determined by the `Predicate`.

A16CollectPartition.java

```
12 public class A16CollectPartition {
13
14     public static void main(String[] args) {
15
16         List<Employee> eList = Employee.createShortList();
17
18         Map<Boolean, List<Employee>> gMap = new HashMap<>();
19
20         // Collect CO Executives
21         gMap = eList.stream()
22             .collect(
23                 Collectors.partitioningBy(
24                     e -> e.getRole().equals(Role.EXECUTIVE)));
25
26         System.out.println("\n== Employees by Dept ==");
27         gMap.forEach((k,v) -> {
28             System.out.println("\nGroup: " + k);
29             v.forEach(Employee::printSummary);
30         });
31
32     }
33
34 }
```

This example creates a `Map` based on role. All executives will be in the `true` group, and all other employees will be in the `false` group. Here is a printout of the map.

Output

```
== Employees by Dept ==

Group: false
Name: Bob Baker Role: STAFF Dept: Eng St: KS Salary: $40,000.00
Name: Jane Doe Role: STAFF Dept: Sales St: KS Salary: $45,000.00
Name: John Doe Role: MANAGER Dept: Eng St: KS Salary: $65,000.00
Name: James Johnson Role: MANAGER Dept: Eng St: MA Salary: $85,000.00
Name: John Adams Role: MANAGER Dept: Sales St: MA Salary: $90,000.00

Group: true
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
```

Practices for Lesson 10: Lambda Operations

Practice 10-1: Using Map and Peek

Overview

In this practice, use lambda expressions and the `stream` method along with the `map` and `peek` methods to print a report on all the Widget Pro sales in the state of California (CA).

Assumptions

You have completed the lecture portion of this course.

Tasks

1. Open the `SalesTxn10-01Prac` project.

   - Select File > Open Project.
   - Browse to `/home/oracle/labs/10-LambdaOperations /practices/practice1`.
   - Select `SalesTxn10-01Prac` and click Open Project.

2. Review the code for the `SalesTxn` class. Note that enumerations exist for `BuyerClass`, `State`, and `TaxRate`.

3. Modify the `MapTest` class to create a sales tax report.

   a. Filter the transactions for the following.
      - Transactions from the state of CA: `t.getState().equals(State.CA)`
      - Transactions for the Widget Pro product:
        `t.getProduct().equals("Widget Pro")`

   b. Use the `map` method to calculate the sales tax. The calculation is as follows:
      `t.getTransactionTotal() * TaxRate.byState(t.getState())`

   c. Print a report similar to the following:

```
=== Widget Pro Sales Tax in CA ===
Txn tax: $36,000.00
Txn tax: $180,000.00
```

   Note: To get the comma-separated currency, use something like this:

```
System.out.printf("Txn tax: $%,9.2f%n", amt)
```

4. Copy the main method from the `MapTest` class to the `PeekTest` class.

5. Update your code to print more detailed information about the matching transaction using the `peek` method. A `Consumer` is provided for you that adds the following:

   - Transaction ID
   - Buyer
   - Total Transaction amount
   - Sales tax amount

6. The output should look similar to the following:

```
=== Widget Pro Sales Tax in CA ===
Id: 12 Buyer: Acme Electronics Txn amt: $400,000.00 Txn tax:
$36,000.00
Id: 13 Buyer: Radio Hut Txn amt: $2,000,000.00 Txn tax: $180,000.00
```

Practice 10-2: FindFirst and Lazy Operations

Overview

In this practice, compare a `forEach` loop to a `findFirst` short-circuit terminal operation and see how the two differ in number of operations.

The following Consumer lambda expressions have been written for you to save you from some typing. The variables are: `quantReport`, `streamStart`, `stateSearch`, and `productSearch`.

Assumptions

You have completed the lecture portion of the lesson and the previous practice.

Tasks

1. Open the `SalesTxn10-02Prac` project.

   - Select File > Open Project.
   - Browse to `/home/oracle/labs/10-LambdaOperations/practices/practice2`.
   - Select `SalesTxn10-02Prac` and click Open Project.

2. Edit the `LazyTest` class to perform the steps in this practice.

3. Using `stream` and lambda expressions print out a list of transactions that meet the following criteria.

   a. Create a filter to select all "Widget Pro" sales.

   b. Create a filter to select transactions in the state of Colorado (CO).

   c. Iterate through the matching transactions and print a report similar to the following using `quantReport` in the `forEach`.

```
=== Widget Pro Quantity in CO ===
Seller: Betty Jones-- Buyer: Radio Hut -- Quantity:    20,000
Seller: Dave Smith-- Buyer: PriceCo -- Quantity:     6,000
Seller: Betty Jones-- Buyer: Best Deals -- Quantity:    20,000
```

4. Perform the same search as in the previous step. This time use the `peek` method to display each step in the process. Put a `peek` method call in the following places.

   a. Add a `peek` method after the `stream()` method that uses the `streamStart` as its parameter.

   b. Add a `peek` method after the `filter` for state that uses `stateSearch` as its parameter.

   c. Add a `peek` method after the `filter` for product that uses `productSearch` as its parameter.

   d. Print the final result using `forEach` as in the previous step.

   e. The output should look similar to the following.

```
=== Widget Pro Quantity in CO ===
Stream start: Jane Doe ID: 11
Stream start: Jane Doe ID: 12
Stream start: Jane Doe ID: 13
Stream start: John Smith ID: 14
Stream start: Betty Jones ID: 15
```

```
State Search: Betty Jones St: CO
Product Search
Seller: Betty Jones-- Buyer: Radio Hut -- Quantity:    20,000
Stream start: Betty Jones ID: 16
State Search: Betty Jones St: CO
Stream start: Dave Smith ID: 17
State Search: Dave Smith St: CO
Product Search
Seller: Dave Smith-- Buyer: PriceCo -- Quantity:     6,000
Stream start: Dave Smith ID: 18
State Search: Dave Smith St: CO
Stream start: Betty Jones ID: 19
State Search: Betty Jones St: CO
Product Search
Seller: Betty Jones-- Buyer: Best Deals -- Quantity:    20,000
Stream start: John Adams ID: 20
Stream start: John Adams ID: 21
Stream start: Samuel Adams ID: 22
Stream start: Samuel Adams ID: 23
```

5. Copy the code from the previous step so you can modify it.
6. Replace the `forEach` with a `findFirst` method.
7. Add the following code:

    a. Use an `Optional<SalesTxn>` named `ft` to store the result.

    b. Write an `if` statement to check to see if `ft.isPresent()`.

    c. If a value is returned, call the `accept` method of `quantReport` to display the result.

    d. Your output should look similar to the following:

```
=== Widget Pro Quantity in CO (FindFirst)===
Stream start: Jane Doe ID: 11
Stream start: Jane Doe ID: 12
Stream start: Jane Doe ID: 13
Stream start: John Smith ID: 14
Stream start: Betty Jones ID: 15
State Search: Betty Jones St: CO
Product Search
Seller: Betty Jones-- Buyer: Radio Hut -- Quantity:    20,000
```

Take a moment to consider the difference between terminal and short-circuit terminal operations.

Practices for Lesson 10: Lambda Operations

Practice 10-3: Analyze Transactions with Stream Methods

Overview

In this practice, count the number of transactions and determine the min and max values in the collection for transactions involving Radio Hut.

Assumptions

You have completed the lecture portion of this lesson and the last practice.

Tasks

1. Open the `SalesTxn10-03Prac` project.

    -   Select File > Open Project.
    -   Browse to `/home/oracle/labs/10-LambdaOperations /practices/practice3`.
    -   Select `SalesTxn10-03Prac` and click Open Project.

2. Edit the `RadioHutTest` class to perform the steps in this practice.

3. Using `stream` and lambda expressions print out all the transactions involving Radio Hut.

    a. Use a `filter` to select all "Radio Hut" transactions.
    b. Use the `radioReport` variable to print the matching transactions.
    c. Your output should look similar to the following:

```
=== Radio Hut Transactions ===
ID: 13  Seller: Jane Doe-- Buyer: Radio Hut -- State: CA -- Amt:
$2,000,000
ID: 15  Seller: Betty Jones-- Buyer: Radio Hut -- State: CO -- Amt:
$  800,000
ID: 23  Seller: Samuel Adams-- Buyer: Radio Hut -- State: MA -- Amt:
$1,040,000
```

4. Use `stream`, `filter`, and lambda expressions to calculate and print out the total number of transactions involving Radio Hut. (**Hint:** Use the `count` method.)

5. Use `stream` and lambda expressions to calculate and print out the largest transaction based on the total transaction amount involving Radio Hut. Use the `max` function with a `Comparator`, for example:

```
.max(Comparator.comparing(SalesTxn::getTransactionTotal))
```

6. Using `stream` and lambda expressions calculate and print out the smallest transaction based on the total transaction amount involving Radio Hut. Use the `min` method in a manner similar to the previous method.

    Hint: Remember to check the API documentation for the return types for the specified methods.

7. When complete, your output should look similar to the following.

```
=== Radio Hut Transactions ===
ID: 13   Seller: Jane Doe-- Buyer: Radio Hut -- State: CA -- Amt: $2,000,000
ID: 15   Seller: Betty Jones-- Buyer: Radio Hut -- State: CO -- Amt: $  800,000
ID: 23   Seller: Samuel Adams-- Buyer: Radio Hut -- State: MA -- Amt: $1,040,000
Total Transactions: 3
=== Radio Hut Largest ===
ID: 13   Seller: Jane Doe-- Buyer: Radio Hut -- State: CA -- Amt: $2,000,000
=== Radio Hut Smallest ===
ID: 15   Seller: Betty Jones-- Buyer: Radio Hut -- State: CO -- Amt: $  800,000
```

Practices for Lesson 10: Lambda Operations

Practice 10-4: Perform Calculations with Primitive Streams

Overview

In this practice, calculate the sales totals and average units sold from the collection of sales transactions.

Assumptions

You have completed the lecture portion of this lesson and the previous practice.

Tasks

1. Open the `SalesTxn10-04Prac` project.

   - Select File > Open Project.
   - Browse to `/home/oracle/labs/10-LambdaOperations /practices/practice4`.
   - Select `SalesTxn10-04Prac` and click Open Project.

2. Edit the `CalcTest` class to perform the steps in this practice.

3. Calculate the total sales for "Radio Hut", "PriceCo", and "Best Deals" and print the results.

   - For example, filter Radio Hut with a lambda like this:
     ```
     t -> t.getBuyerName().equals("Radio Hut")
     ```
   - For example, get the transaction total with:
     ```
     .mapToDouble( t -> t.getTransactionTotal())
     ```

4. Calculate the average number of units sold for the "Widget" and "Widget Pro" products and print the results.

   - For example, the Widget Pro code looks like the following:

     ```
     .filter(t -> t.getProduct().equals("Widget Pro"))
     .mapToDouble( t-> t.getUnitCount())
     ```

     Hint: Be mindful of the method return types. Use to the API doc to ensure you are using the correct methods and classes to create and store results.

5. The output from your test class should be similar to the following:

   ```
   === Transactions Totals ===
   Radio Hut Total: $3,840,000.00
   PriceCo Total: $1,460,000.00
   Best Deals Total: $1,300,000.00
   === Average Unit Count ===
   Widget Pro Avg:    21,143
   Widget Avg:    12,400
   ```

Practice 10-5: Sort Transactions with Comparator

Overview

In this practice, sort transactions using the Comparator class, the comparing method, and the sorted method.

Assumptions

You have completed the lecture portion of this lesson and the previous practice.

Tasks

1. Open the `SalesTxn10-05Prac` project.

   - Select File > Open Project.

   - Browse to `/home/oracle/labs/10-LambdaOperations /practices/practice5`.

   - Select `SalesTxn10-05Prac` and click Open Project.

2. Edit the `SortTest` class to perform the steps in this practice.

3. Use streams and lambda expressions to print out all the PriceCo transactions by transaction total in ascending order.

   - The sorted method should look something like this:

   `.sorted(Comparator.comparing(SalesTxn::getTransactionTotal))`

   - Use the `transReport` variable to print the results.

4. Use the same data from the previous step to print out the PriceCo transactions in descending order.

5. Print out all the transactions sorted using the following sort keys.

   - Buyer name
   - Sales person
   - Transaction total

6. When complete, the output should look similar to the following:

```
=== PriceCo Transactions ===
Id: 17 Seller: Dave Smith Buyer: PriceCo Amt: $240,000.00
Id: 20 Seller: John Adams Buyer: PriceCo Amt: $280,000.00
Id: 18 Seller: Dave Smith Buyer: PriceCo Amt: $300,000.00
Id: 21 Seller: John Adams Buyer: PriceCo Amt: $640,000.00

=== PriceCo Transactions Reversed ===
Id: 21 Seller: John Adams Buyer: PriceCo Amt: $640,000.00
Id: 18 Seller: Dave Smith Buyer: PriceCo Amt: $300,000.00
Id: 20 Seller: John Adams Buyer: PriceCo Amt: $280,000.00
Id: 17 Seller: Dave Smith Buyer: PriceCo Amt: $240,000.00

=== Triple Sort Transactions ===
Id: 11 Seller: Jane Doe Buyer: Acme Electronics Amt: $60,000.00
Id: 12 Seller: Jane Doe Buyer: Acme Electronics Amt: $400,000.00
Id: 16 Seller: Betty Jones Buyer: Best Deals Amt: $500,000.00
Id: 19 Seller: Betty Jones Buyer: Best Deals Amt: $800,000.00
Id: 14 Seller: John Smith Buyer: Great Deals Amt: $100,000.00
```

Practices for Lesson 10: Lambda Operations

```
Id: 22 Seller: Samuel Adams Buyer: Mom and Pops Amt: $60,000.00
Id: 17 Seller: Dave Smith Buyer: PriceCo Amt: $240,000.00
Id: 18 Seller: Dave Smith Buyer: PriceCo Amt: $300,000.00
Id: 20 Seller: John Adams Buyer: PriceCo Amt: $280,000.00
Id: 21 Seller: John Adams Buyer: PriceCo Amt: $640,000.00
Id: 15 Seller: Betty Jones Buyer: Radio Hut Amt: $800,000.00
Id: 13 Seller: Jane Doe Buyer: Radio Hut Amt: $2,000,000.00
Id: 23 Seller: Samuel Adams Buyer: Radio Hut Amt: $1,040,000.00
```

Practice 10-6: Collect Results with Streams

Overview

In this practice, use the `collect` method to store the results from a `stream` in a new list.

Assumptions

You have completed the lecture portion of this lesson and the previous practice.

Tasks

1. Open the `SalesTxn10-06Prac` project.

   - Select File > Open Project.
   - Browse to `/home/oracle/labs/10-LambdaOperations /practices/practice6`.
   - Select `SalesTxn10-06Prac` and click Open Project.

2. Edit the `CollectTest` class to perform the steps in this practice.

3. Filter the transaction list to only include transactions greater than $300,000 sorted in ascending order.

4. Store the results in a new list using the `collect` method. For example:

```
.collect(Collectors.toList())
```

5. Print out the transactions in the new list. The output should look similar to the following:

```
=== Transactions over $300k ===
Id: 12 Seller: Jane Doe Buyer: Acme Electronics Amt: $400,000.00
Id: 16 Seller: Betty Jones Buyer: Best Deals Amt: $500,000.00
Id: 21 Seller: John Adams Buyer: PriceCo Amt: $640,000.00
Id: 15 Seller: Betty Jones Buyer: Radio Hut Amt: $800,000.00
Id: 19 Seller: Betty Jones Buyer: Best Deals Amt: $800,000.00
Id: 23 Seller: Samuel Adams Buyer: Radio Hut Amt: $1,040,000.00
Id: 13 Seller: Jane Doe Buyer: Radio Hut Amt: $2,000,000.00
```

Practice 10-7: Join Data with Streams

Overview

In this practice, use the `joining` method to combine data returned from a stream.

Assumptions

You have completed the lecture portion of this lesson and the previous practice.

Tasks

1. Open the `SalesTxn10-07Prac` project.

    - Select File > Open Project.
    - Browse to `/home/oracle/labs/10-LambdaOperations /practices/practice7`.
    - Select `SalesTxn10-07Prac` and click Open Project.

2. Edit the `JoinTest` class to perform the steps in this practice.

3. Get a list of unique buyer names in a sorted order. Follow these steps to accomplish the task:

    a. Use `map` to get all the buyer names.
    b. Use `distinct` to remove duplicates.
    c. Use `sorted` to sort the names.
    d. Use `joining` to join the names together in the output you see in the following.

4. When complete, your output should look similar to the following:

```
=== Sorted Buyer's List ===
Buyer list: Acme Electronics, Best Deals, Great Deals, Mom and Pops,
PriceCo, Radio Hut
```

Practice 10-8: Group Data with Streams

Overview

In this practice, create a `Map` of transaction data using the `groupingBy` method from the `Collectors` class.

Assumptions

You have completed the lecture portion of this lesson and the previous practice.

Tasks

1. Open the `SalesTxn10-08Prac` project.

    - Select File > Open Project.
    - Browse to `/home/oracle/labs/10-LambdaOperations /practices/practice8`.
    - Select `SalesTxn10-08Prac` and click Open Project.

2. Edit the `GroupTest` class to perform the steps in this practice.

3. Populate the `Map` by using the stream `collect` method to return the list elements grouped by buyer name.

    a. Use `Collectors.groupingBy()` to group the results.

    b. Use `SalesTxn::getBuyerName` to determine what to group by.

4. Print out the result.

5. Use the `printSummary` method of the `SalesTxn` class to print individual transactions.

6. Your output should look similar to the following:

```
=== Transactions Grouped by Buyer ===

Buyer: PriceCo
ID: 17 - Seller: Dave Smith - Buyer: PriceCo - Product: Widget Pro - ST: CO - Amt:
240000.0 - Date: 2013-03-20
ID: 18 - Seller: Dave Smith - Buyer: PriceCo - Product: Widget - ST: CO - Amt:
300000.0 - Date: 2013-03-30
ID: 20 - Seller: John Adams - Buyer: PriceCo - Product: Widget - ST: MA - Amt:
280000.0 - Date: 2013-07-14
ID: 21 - Seller: John Adams - Buyer: PriceCo - Product: Widget Pro - ST: MA - Amt:
640000.0 - Date: 2013-10-06

Buyer: Acme Electronics
ID: 11 - Seller: Jane Doe - Buyer: Acme Electronics - Product: Widgets - ST: CA -
Amt: 60000.0 - Date: 2013-01-25
ID: 12 - Seller: Jane Doe - Buyer: Acme Electronics - Product: Widget Pro - ST: CA
- Amt: 400000.0 - Date: 2013-04-05

Buyer: Radio Hut
ID: 13 - Seller: Jane Doe - Buyer: Radio Hut - Product: Widget Pro - ST: CA - Amt:
2000000.0 - Date: 2013-10-03
ID: 15 - Seller: Betty Jones - Buyer: Radio Hut - Product: Widget Pro - ST: CO -
Amt: 800000.0 - Date: 2013-02-04
ID: 23 - Seller: Samuel Adams - Buyer: Radio Hut - Product: Widget Pro - ST: MA -
Amt: 1040000.0 - Date: 2013-12-08

Buyer: Mom and Pops
ID: 22 - Seller: Samuel Adams - Buyer: Mom and Pops - Product: Widget - ST: MA -
Amt: 60000.0 - Date: 2013-10-02

Buyer: Best Deals
ID: 16 - Seller: Betty Jones - Buyer: Best Deals - Product: Widget - ST: CO - Amt:
500000.0 - Date: 2013-03-21
ID: 19 - Seller: Betty Jones - Buyer: Best Deals - Product: Widget Pro - ST: CO -
Amt: 800000.0 - Date: 2013-07-12

Buyer: Great Deals
ID: 14 - Seller: John Smith - Buyer: Great Deals - Product: Widget - ST: CA - Amt:
100000.0 - Date: 2013-10-10
```

Practices for Lesson 10: Lambda Operations

Practices for Lesson 10: Lambda Operations

Practices for Lesson 11: Exceptions and Assertions

Chapter 11

Practices Overview

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

Practice 11-1: 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. Perform the following tasks to create a new `CatchingExceptions11-01` project.

    a. Select File > New Project.

    b. Select Java under Categories and Java Application under Projects.

    c. Click Next.

    d. Enter the following information in the "Name and Location" dialog box:

        1) Project Name: `CatchingExceptions11-01`

        2) Project Location: `/home/oracle/labs/11-Exceptions /practices/practice1/CatchingExceptions11-01`.

        3) Check Create Main Class: `com.example.ExceptionMain`

    e. Click Finish.

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 `CatchingExceptions11-01` project and select Properties.

   b. In the Project Properties dialog box, select the Run category.

   c. In the Arguments field, enter a value of:
   `/home/oracle/labs/resources/DeclarationOfIndependence.txt`

   d. Click OK.

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

```
Reading from file:
/home/oracle/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 `CatchingExceptions11-01` 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 8.

   d. Click OK.

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 `/home/oracle/labs/resources/DeclarationOfIndependence.txt` file displayed in the output window.

Practice 11-1: 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. Perform the following steps to create a new `CatchingExceptions11-01` project as the main project.

   a. Click File > New Project.

   b. Select Java from Categories, and Java Application from Projects.

   c. Click Next.

   d. Enter the following information in the "Name and Location" dialog box:

      1) Project Name: `CatchingExceptions11-01`

      2) Project Location: `/home/oracle/labs/11-Exceptions
      /practices/practice1/CatchingExceptions11-01`.

      3) Check Create Main Class: `com.example.ExceptionMain`.

   e. Click Finish.

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 `CatchingExceptions11-01` project and click Properties.
   b. In the Project Properties dialog box, select the Run category.
   c. In the Arguments field, enter a value of:
      `/home/oracle/labs/resources/DeclarationOfIndependence.txt`
   d. Click OK.

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

```
Reading from
/home/oracle/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
```

Practices for Lesson 11: Exceptions and Assertions

11. Modify the project properties to support the try-with-resources statement.

    a. Right-click the CatchingExceptions11-01 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 8.

    d. Click OK.

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
    /home/oracle/labs/resources/DeclarationOfIndependence.txt file displayed
    in the output window.

Practice 11-2: Summary Level: Extending `Exception` and Throwing `Exception`

Overview

In this practice, you will take an existing application and refactor the code to make use of a custom exception class and throwing exception using `throw` and `throws`.

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, deleting, and listing of `Employee` objects.

Tasks

1.   Open the `CustomExceptions11-02Prac` project as the main project.

     a.   Select File > Open Project.

     b.   Browse to `/home/oracle/labs/11-Exceptions/practices/practice2`

     c.   Select `CustomExceptions11-02Prac`

     d.   and click Open Project.

2.   Expand the project directories.

3.   Create a `InvalidOperationException` class in the `com.example` package.

5.   Complete the `InvalidOperationException` class. The `InvalidOperationException` class should:

     ▪ Extend the `Exception` class

     ▪ Contain four public 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 `EmployeeImpl` class.

     a.   Modify the methods: `add`, `delete` and `findById`

     b.   Declare that a `InvalidOperationException` may be produced during execution of these method.

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

12. Modify the `EmployeeTest` class to handle the `InvalidOperationException` objects that are thrown by the `EmployeeImpl`.

     a.   Modify the `main` method:

     Add the `throws` statement from the `main` method.

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

13. Run the project. Test all the operations by invoking the methods: `add,delete` and `findById`.

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

```
Exception in thread "main"
com.example.InvalidOperationException: Error deleting employee,
no such employee 7
```

Practices for Lesson 11: Exceptions and Assertions

Practice 11-2: Detailed Level: Extending `Exception` and Throwing `Exception`

Overview

In this practice, you will take an existing application and refactor the code to make use of a custom exception class and throwing exception using `throw` and `throws`.

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, deleting, and listing of `Employee` objects.

Tasks

1. Open the `CustomExceptions11-02Prac` project as the main project.

    a. Select File > Open Project.

    b. Browse to `/home/oracle/labs/11-Exceptions/practices/practice2`

    c. Select `CustomExceptions11-02Prac` and Click Open Project.

2. Expand the project directories.

3. Create a `InvalidOperationException` class in the `com.example` package.

5. Complete the `InvalidOperationException` class. The `InvalidOperationException` class should:

    ▪ Extend the `Exception` class.

    ▪ Create four public 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.

```
package com.example;

public class InvalidOperationException extends Exception {

    public InvalidOperationException() {
        super();
    }

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

    public InvalidOperationException(Throwable cause) {
        super(cause);
    }
```

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

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

- Declare that a `InvalidOperationException` 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 `InvalidOperationException` and deliver it to the caller of the method. The `InvalidOperationException` 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 `InvalidOperationException` and deliver it to the caller of the method. The `InvalidOperationException` should contain a message String indicating what went wrong and why.

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

8. Modify the `delete` method within the `EmployeeImpl` class to:

- Declare that a `InvalidOperationException` 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 `InvalidOperationException` and deliver it to the caller of the method. The `InvalidOperationException` 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.

Practices for Lesson 11: Exceptions and Assertions

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

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

9. Modify the `findById` method within the `EmployeeImpl` class to:

- Declare that a `InvalidOperationException` 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
  `InvalidOperationException` and deliver it to the caller of the method. The
  `InvalidOperationException` should contain a message String indicating what went
  wrong and why.

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

10. Modify the `EmployeeTest` class to handle the `InvalidOperationException` objects
    that are thrown by the `EmployeeImpl`

.

b. Modify the `main` method:

Add the `throws` statement from the `main` method.

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

11. Run the project. Test all the operations by invoking the methods: `add,delete` and `findById`.

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

```
Exception in thread "main"
com.example.InvalidOperationException: Error deleting employee,
no such employee 7
```

Practices for Lesson 11: Exceptions and Assertions

Practices for Lesson 12:
Using the Date/Time API

Chapter 12

Practices for Lesson 12

Practices Overview

In these practices, you will work with the new date and time API.

Practices for Lesson 12: Using the Date/Time API

Chapter 12 - Page 2

Practice 12-1: Summary Level: Working with local dates and times

Overview

In this practice you work with `LocalDate`, `LocalTime`, and `LocalDateTime` objects to provide answers to the questions asked in the practice. Local objects have no concept of a time zone, so you can assume that all of the questions in the practice relate to the local time zone. Also, all of the dates utilize the ISO calendar.

Tasks

1. Open the `LocalDatesAndTimes12-01Prac` project in the `/home/oracle/labs/12-DateTime/practices/practice1` directory.

2. Open the Java class file, `LocalDatesAndTimes`, in the `com.example` package.

3. Read through the comments—these indicate what code you need to write to answer the questions provided.

   • Consult the lecture slides and the documentation if you get stuck.

   • When you have completed, the output from your class should look similar to the following output. (You do not need to format the print statements exactly the same way.)

```
Abe was 46 when he died.
Abe lived for 16863 days.


Bennedict was born in a leap year: true
Days in the year he was born: 366
Bennedict is 3 decades old.
It was a SATURDAY on his 21st birthday.


Planned Travel time:  340 minutes
Delayed arrival time: 20:44


The flight arrives in Miami: 2014-03-25T01:30
The delayed arrival time is: 2014-03-25T05:57


School starts: 2014-09-09
School ends:   2015-06-25
Number of school days: 183


The meeting time is: 2014-08-05T13:30
```

Practice 12-2: Detailed Level: Working with local dates and times

Overview

In this practice you work with `LocalDate`, `LocalTime` and `LocalDateTime` objects to provide answers to the questions asked in the practice. Local objects have no concept of a time zone, so you can assume that all of the questions in the practice relate to the local time zone. Also, all of the dates utilize the ISO calendar.

Tasks

1. Open the `LocalDatesAndTimes12-01Prac` project.

   a. Select File > Open Project.

   b. Browse to `/home/oracle/labs/12-DateTime/practices/practice1`.

   c. Select `LocalDatesAndTimes12-01Prac` and click Open Project.

2. Open the Java class file, `LocalDatesAndTimes`, in the `com.example` package.

3. Given the scenario:

   Abe Lincoln's Birthday: February 12, 1809, died April 15, 1855

   How old was he when he died?

   How many days did he live?

   To implement this scenario, add the following code to `LocalDatesAndTimes.java`

   ```java
   LocalDate abeBorn = LocalDate.of(1809, FEBRUARY, 12);
   LocalDate abeDies = LocalDate.of(1855, APRIL, 15);
   System.out.println("Abe was " + abeBorn.until(abeDies, YEARS) +
   " when he died.");
   System.out.println("Abe lived for " + abeBorn.until(abeDies,
   DAYS) + " days.");
       System.out.println("");
   ```

4. Given the scenario:

   Bennedict Cumberbatch, July 19, 1976

   Born in a leap year?

   How many days in the year he was born?

   How many decades old is he?

   What was the day of the week on his 21st birthday?

   To implement this scenario, add the following code to `LocalDatesAndTimes.java`

   ```java
   LocalDate bennedict = LocalDate.of(1976, JULY, 19);
       System.out.println("Bennedict was born in a leap year: " +
   bennedict.isLeapYear());
       System.out.println("Days in the year he was born: " +
   bennedict.lengthOfYear());
       LocalDate now = LocalDate.now();
       System.out.println("Bennedict is " + bennedict.until(now,
   DECADES) + " decades old.");
   ```

```
        System.out.println("It was a " +
bennedict.plusYears(21).getDayOfWeek() + " on his 21st
birthday.");
        System.out.println("");
```

5. Given the scenario:

   Train departs Boston at 1:45PM and arrives New York 7:25PM

   How many minutes long is the train ride?

   If the train was delayed 1 hour 19 minutes, what is the actual arrival time?

To implement this scenario, add the following code to `LocalDatesAndTimes.java`

```
        LocalTime depart = LocalTime.of(13, 45);
        LocalTime arrive = LocalTime.of(19, 25);
        System.out.println("Planned Travel time:  " +
depart.until(arrive, MINUTES) + " minutes");
        System.out.println("Delayed arrival time: " +
arrive.plusHours(1).plusMinutes(19));
        System.out.println("");
```

6. Given the scenario:

   Flight: Boston to Miami, leaves March 24th 9:15PM. Flight time is 4 hours 15 minutes

   When does it arrive in Miami?

   When does it arrive if the flight is delays 4 hours 27 minutes?

To implement this scenario, add the following code to `LocalDatesAndTimes.java`

```
        LocalDateTime leaveBoston = LocalDateTime.of(2014, MARCH,
24, 21, 15);
        LocalDateTime arriveMiami =
leaveBoston.plusHours(4).plusMinutes(15);
        System.out.println("The flight arrives in Miami: " +
arriveMiami);
        System.out.println("The delayed arrival time is: " +
arriveMiami.plusHours(4).plusMinutes(27));
        System.out.println("");
```

7. Given the scenario:

School semester starts the second Tuesday of September of this year.

   Hint: Look at the TemporalAdjusters class

   What is the date?

   School summer vacation starts June 25th

   Assuming:

   /* Two weeks off in December

    * Two other vacation weeks

   * School is taught Monday - Friday

   How many days of school are there?

   Hint: keep track of the short weeks also

To implement this scenario, add the following code to `LocalDatesAndTimes.java`

```
int excludeWeeks = 5;
    LocalDate schoolStarts = LocalDate.of(2014, SEPTEMBER,
1).with(TemporalAdjusters.firstInMonth(TUESDAY)).with(TemporalAd
justers.next(TUESDAY));
    LocalDate endOfFirstWeek =
schoolStarts.with(TemporalAdjusters.next(FRIDAY));
    long firstWeekDays = schoolStarts.until(endOfFirstWeek,
DAYS) + 1;
    System.out.println("School starts: " + schoolStarts);
    LocalDate schoolEnds = LocalDate.of(2015, JUNE, 25);
    System.out.println("School ends:   " + schoolEnds);
    long lastWeeksDays = 0;
    if (schoolEnds.getDayOfWeek() != MONDAY) {
      LocalDate lastWeekStart =
schoolEnds.with(TemporalAdjusters.previous(MONDAY));
      lastWeeksDays = lastWeekStart.until(schoolEnds, DAYS) + 1;
      excludeWeeks++;
    }
    long days = ((schoolStarts.until(schoolEnds, WEEKS) -
excludeWeeks) * 5); // 7 days per week, weekdays are 5/7 of a
week.
    days = days + firstWeekDays + lastWeeksDays;
    System.out.println("Number of school days: " + days);
    System.out.println("");
```

8. Given the scenario:

   A meeting is scheduled for 1:30 PM next Tuesday. If today is Tuesday, assume it is today.
   What is the time of the week's meetings?

   To implement this scenario, add the following code to `LocalDatesAndTimes.java`

```
    LocalTime meetingTime = LocalTime.of(13, 30);
    LocalDate meetingDate =
LocalDate.now().with(TemporalAdjusters.nextOrSame(TUESDAY));
    LocalDateTime meeting = LocalDateTime.of(meetingDate,
meetingTime);
    System.out.println("The meeting time is: " + meeting);
    System.out.println("");
```

9. Run the project, the output should look similar to the following output.

```
Abe was 46 when he died.
Abe lived for 16863 days.


Bennedict was born in a leap year: true
Days in the year he was born: 366
Bennedict is 3 decades old.
It was a SATURDAY on his 21st birthday.


Planned Travel time:  340 minutes
Delayed arrival time: 20:44


The flight arrives in Miami: 2014-03-25T01:30
The delayed arrival time is: 2014-03-25T05:57


School starts: 2014-09-09
School ends:   2015-06-25
Number of school days: 183


The meeting time is: 2014-08-05T13:30
```

Practice 12-2: Summary Level: Working with dates and times across time zones

Overview

In this practice, you work with time zone classes to calculate dates and times across time zones.

Tasks

1. Open the NetBeans project `DepartArrive12-02Prac` in the `/home/oracle/labs/12-DateTime/practices/practice2` directory.

2. Open the class file, `DepartArrive.java` in the `com.example` package.

3. Read through the comments—these indicate what code you need to write to answer the questions provided.

- Consult the lecture slides and the documentation if you get stuck.

- When you are complete, the output from your class should look similar to this (note that you need not format the print statements exactly the same way).

```
Flight 123 departs SFO at:   2014-06-13T22:30-07:00[America/Los_Angeles]
Local time BOS at departure: 2014-06-14T01:30-04:00[America/New_York]
Flight time: 5 hours 30 minutes
Flight 123 arrives BOS:      2014-06-14T07:00-04:00[America/New_York]
Local time SFO at arrival:   2014-06-14T04:00-07:00[America/Los_Angeles]


Flight 456 leaves SFO at:    2014-06-28T22:30-07:00[America/Los_Angeles]
Local time BLR at departure: 2014-06-29T11:00+05:30[Asia/Calcutta]
Flight time: 22 hours
Flight 456 arrives BLR:      2014-06-30T09:00+05:30[Asia/Calcutta]
Local time SFO at arrival:   2014-06-29T20:30-07:00[America/Los_Angeles]


Flight 123 departs SFO at:   2014-11-01T22:30-07:00[America/Los_Angeles]
Local time BOS at departure: 2014-11-02T01:30-04:00[America/New_York]
Flight time: 5 hours 30 minutes
Flight 123 arrives BOS:      2014-11-02T06:00-05:00[America/New_York]
Local time SFO at arrival:   2014-11-02T03:00-08:00[America/Los_Angeles]
```

Practice 12-2: Detailed Level: Working with dates and times across time zones

Overview

In this practice, you work with time zone classes to calculate dates and times across time zones.

Tasks

1. Open the project `DepartArrive12-02Prac`.

    a. Select File > Open Project.

    b. Browse to `/home/oracle/labs/12-DateTime/practices/practice2`

    c. Select `DepartArrive12-02Prac` and click Open Project.

2. Open the `DepartArrive.java` file in the `com.example` package and make the following changes:

    a. Set the time zone for the three cities.

    ```
    ZoneId SFO = ZoneId.of("America/Los_Angeles");
    ZoneId BOS = ZoneId.of("America/New_York");
    ZoneId BLR = ZoneId.of("Asia/Calcutta");
    ```

    b. Given the scenario:

Flight 123, San Francisco to Boston, leaves SFO at 10:30 PM June 13, 2014

  The flight is 5 hours 30 minutes

  What is the local time in Boston when the flight takes off?

  What is the local time at Boston Logan airport when the flight arrives?

  What is the local time in San Francisco when the flight arrives?

Complete the following steps:

To compute the local time in Boston when the flight takes off, add the following code:

```
LocalDateTime departure = LocalDateTime.of(2014, JUNE, 13, 22,
30);
ZonedDateTime departSFO = ZonedDateTime.of(departure, SFO);
System.out.println("Flight 123 departs SFO at:   " + departSFO);
ZonedDateTime departTimeAtBOS =
departSFO.toOffsetDateTime().atZoneSameInstant(BOS);
System.out.println("Local time BOS at departure: " +
departTimeAtBOS);
System.out.println("Flight time: 5 hours 30 minutes");
```

    c. To compute local time at Boston Logan airport when the flight arrives, add the following code:

```
ZonedDateTime arriveBOS =
departSFO.plusHours(5).plusMinutes(30).toOffsetDateTime().atZone
SameInstant(BOS);
    System.out.println("Flight 123 arrives BOS:      " +
arriveBOS);
```

Practices for Lesson 12: Using the Date/Time API

d. To compute the local time in San Francisco when the flight arrives, add the following code:

```
ZonedDateTime arriveTimeAtSFO =
arriveBOS.toOffsetDateTime().atZoneSameInstant(SFO);
    System.out.println("Local time SFO at arrival:   " +
arriveTimeAtSFO);
    System.out.println("");
```

3. Given the scenario:

- Flight 456, San Francisco to Bangalore, India, leaves SFO at Saturday, 10:30 PM June 28, 2014
- The flight time is 22 hours
- Will the traveler make a meeting in Bangalore Monday at 9 AM local time?
- Can the traveler call her husband at a reasonable time?

Modify `DepartArrive.java`.

a. Compute the local departure time at SFO.

```
departure = LocalDateTime.of(2014, JUNE, 28, 22, 30);
    departSFO = ZonedDateTime.of(departure, SFO);
    System.out.println("Flight 456 leaves SFO at:    " +
departSFO);
```

b. Compute the local departure time at Bangalore.

```
ZonedDateTime departTimeAtBLR =
departSFO.toOffsetDateTime().atZoneSameInstant(BLR);
    System.out.println("Local time BLR at departure: " +
departTimeAtBLR);
    System.out.println("Flight time: 22 hours");
```

c. Compute the local arrival time at Bangalore.

```
ZonedDateTime arriveBLR =
departSFO.plusHours(22).toOffsetDateTime().atZoneSameInstant(BLR
);
System.out.println("Flight 456 arrives BLR:     " + arriveBLR);
```

d. Compute the local arrival time at SFO.

```
arriveTimeAtSFO =
arriveBLR.toOffsetDateTime().atZoneSameInstant(SFO);
     System.out.println("Local time SFO at arrival:    " +
arriveTimeAtSFO);
     System.out.println("");
```

4. Given the scenario:

Flight 123, San Francisco to Boston, leaves SFO at 10:30 PM Saturday, November 1st, 2014
Flight time is 5 hours 30 minutes.
What day and time does the flight arrive in Boston?
What happened?

Modify DepartArrive.java.

a. Compute the local departure time at SFO.

```
departure = LocalDateTime.of(2014, NOVEMBER, 1, 22, 30);
     departSFO = ZonedDateTime.of(departure, SFO);
     System.out.println("Flight 123 departs SFO at:    " +
departSFO);
```

b. Compute the local departure time at Boston.

```
departTimeAtBOS =
departSFO.toOffsetDateTime().atZoneSameInstant(BOS);
     System.out.println("Local time BOS at departure: " +
departTimeAtBOS);
     System.out.println("Flight time: 5 hours 30 minutes");
```

c. Compute the local arrival time at SFO with the delay of 5 hours.

```
   arriveBOS =
departSFO.plusHours(5).plusMinutes(30).toOffsetDateTime().atZone
SameInstant(BOS);
     System.out.println("Flight 123 arrives BOS:      " +
arriveBOS);
     arriveTimeAtSFO =
arriveBOS.toOffsetDateTime().atZoneSameInstant(SFO);
     System.out.println("Local time SFO at arrival:   " +
arriveTimeAtSFO);
     System.out.println("");
```

5. Run the project, the output could be similar to this:

```
Flight 123 departs SFO at:   2014-06-13T22:30-07:00[America/Los_Angeles]
Local time BOS at departure: 2014-06-14T01:30-04:00[America/New_York]
Flight time: 5 hours 30 minutes
Flight 123 arrives BOS:      2014-06-14T07:00-04:00[America/New_York]
Local time SFO at arrival:   2014-06-14T04:00-07:00[America/Los_Angeles]

Flight 456 leaves SFO at:    2014-06-28T22:30-07:00[America/Los_Angeles]
Local time BLR at departure: 2014-06-29T11:00+05:30[Asia/Calcutta]
Flight time: 22 hours
Flight 456 arrives BLR:      2014-06-30T09:00+05:30[Asia/Calcutta]
Local time SFO at arrival:   2014-06-29T20:30-07:00[America/Los_Angeles]

Flight 123 departs SFO at:   2014-11-01T22:30-07:00[America/Los_Angeles]
Local time BOS at departure: 2014-11-02T01:30-04:00[America/New_York]
Flight time: 5 hours 30 minutes
Flight 123 arrives BOS:      2014-11-02T06:00-05:00[America/New_York]
Local time SFO at arrival:   2014-11-02T03:00-08:00[America/Los_Angeles]
```

Practice 12-3: Summary Level: Formatting Dates

Overview

In this practice, you work with the `DateTimeFormatter`.

Tasks

1. Open the project `TimeBetween12-03Prac` in NetBeans from the `/home/oracle/labs/12-DateTime/practices/practice3` directory.

2. Open the class, `TimeBetween.java`.

3. Modify the class to read a string from the console and correctly identify the delta between today and the entered date in years, months, and days.

   - Use the appropriate method to ensure that the values for the year, month and days are always positive.

4. The output should look similar to this:

```
Enter a date: (MMMM d, yyyy): July 9, 2014
Date entered: July 9, 2014
There are 0 years, 4 months, 16 days between now and the date
entered.
```

Practice 12-3: Detailed Level : Formatting Dates

Overview

In this practice, you work with the DateTimeFormatter.

Tasks

1. Open the project TimeBetween12-03Prac in NetBeans.

    a. Select File > Open Project.

    b. Browse to /home/oracle/labs/12-DateTime/practices/practice3

    c. Select TimeBetween12-03Prac and click Open Project.

2. Open the class, TimeBetween.java.

3. Modify the class to read a string from the console and correctly identify the delta between today and the entered date in years, months, and days.

    - Use the appropriate method to ensure that the values for the year, month and days are always positive.

a. Declare two variables.

```
String dateFormat = "MMMM d, yyyy";
    LocalDate aDate = null;
```

b. Create a formatter to accept date entries using the USA common standard(month day, year).

```
    DateTimeFormatter formatter =
DateTimeFormatter.ofPattern(dateFormat);
```

c. Use the parse method with the formatter to create a date. Add the following statement within the try block.

```
            aDate = LocalDate.parse(dateEntered, formatter);
```

```
while (!validStr) {
  System.out.print("Enter a date: (" + dateFormat + "): ");
  try {
    String dateEntered = br.readLine();
    aDate = LocalDate.parse(dateEntered, formatter);
    validStr = true;
  } catch (IOException | DateTimeParseException ex) {
    validStr = false;
  }
}
```

d. To calculate the years, months, and days between now and the date entered, enter the following code:

```
Period between;
if (aDate.isBefore(now)) {
  between = Period.between(aDate, now);
} else {
  between = Period.between(now, aDate);
}
```

e. Obtain the value of day, month and year and assign it to the variables: `days`, `months`, and `years`.

```
int years = between.getYears();
int months = between.getMonths();
int days = between.getDays();
```

f. Print the values of the years, months, and days.

```
System.out.println("There are " + years + " years, "
            + months + " months, "
            + days + " days between now and the date entered.");
```

4. Run the project, the output could be similar to this:

```
run:
Enter a date: (MMMM d, yyyy): July 9, 2014
Date entered: July 9, 2014
There are 0 years, 2 months, 7 days between now and the date entered.
```

Practices for Lesson 12: Using the Date/Time API

Chapter 12 - Page 15

Practices for Lesson 13: Java I/O Fundamentals

Chapter 13

Practices for Lesson 13: 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 13-1: 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 `FileScanner13-01Prac` in the following directory: `/home/oracle/labs/13-IO_Fundamentals/practices/practice1`

2. Open the file `FileScanInteractive.java`.

   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:
      `/home/oracle/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:
/home/oracle/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)
```

Practices for Lesson 13: Java I/O Fundamentals

Practice 13-1: 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 `FileScanner13-01Prac`.

    a. Select File > Open Project.

    b. Browse to `/home/oracle/labs/13-IO_Fundamentals/practices/practice1`.

    c. Select `FileScanner13-01Prac` and select the "Open as Main Project" check box.

    d. Click the Open Project button.

2. Open the file `FileScanInteractive.java`.

    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);
}
```
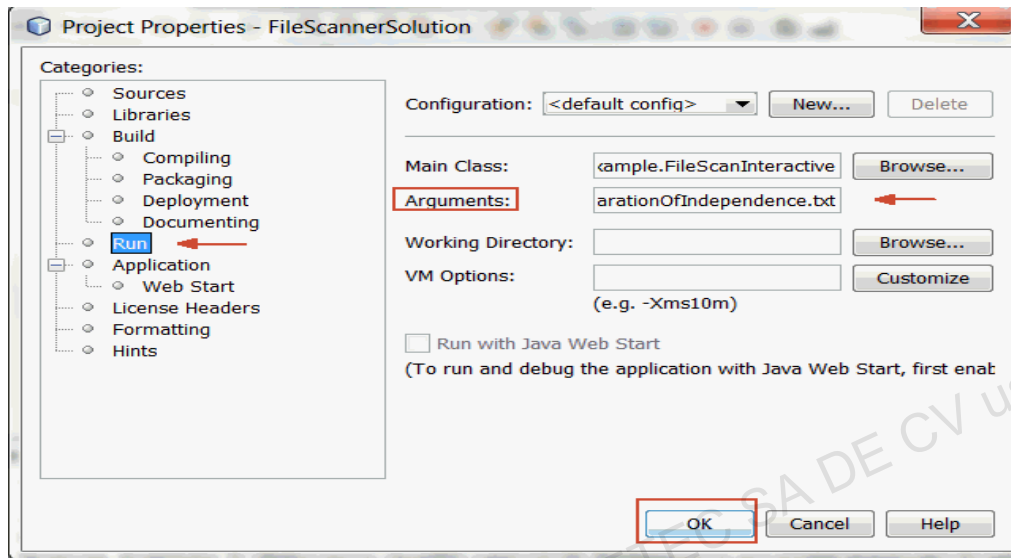
  c. In the try block that you created, add a while loop. The while loop should run until a break statement. Inside the while loop, read from the system console into a string until the string "q" is entered on the console by itself.
  Note: You can use `equalsIgnoreCase` to allow your users to enter an 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.

Practices for Lesson 13: Java I/O Fundamentals

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. Select Run from the Categories column.

   c. Enter the name of a file to open in the Arguments text box, for example:
      `/home/oracle/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:
/home/oracle/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)
```

Practices for Lesson 13: Java I/O Fundamentals

Practice 13-2: 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 `SerializeShoppingCart13-02Prac` project in the directory:

   `/home/oracle/labs/13-IO_Fundamentals/practices/practice2`

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.java`. 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.java`. 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 `ObjectOutputStream` 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.java`. 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.java`, 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.java`, right-click the class name and select Run File.

d. Enter the ID `101` and the output will look like something this:

```
run:
Enter the ID of the cart file to deserialize or q exit.
101
Restored Shopping Cart from: Apr 16, 2014
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
BUILD SUCCESSFUL (total time: 4 seconds)
```

Practice 13-2: 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 `SerializeShoppingCart13-02Prac` project in the `/home/oracle/labs/13-IO_Fundamentals/practices/practice2` directory.

    a. Select File > Open Project.

    b. Browse to the `/home/oracle/labs/13-IO_Fundamentals/practices/practice2` directory.

    c. Select the project `SerializeShoppingCart13-02Prac`.

    d. Click Open Project.

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.java`. 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 `DeserializeTest.java`. 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.   Your code should look like this:

```
try (FileInputStream fis = new FileInputStream (cartFile);
     ObjectInputStream in = new ObjectInputStream (fis)) {
    cart = (ShoppingCart)in.readObject();
} catch (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.java. 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.

  a. To run the `SerializeTest.java`, right-click the class name and select Run File. Enter a cart ID, such as 101.

  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.java`, right-click the class name and select Run File.

  d. 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 13: Java I/O Fundamentals

Practices for Lesson 14: Java File NIO2

Chapter 14

Practices for Lesson 14: Overview

Practice Overview

In these practices, explore various new features in Java 8 that relate to streams.

Practices for Lesson 14: Java File NIO2

Chapter 14 - Page 2

Practice 14-1: Working with Files

Overview

In this practice, read text files using new features in Java 8 and the `lines` method.

Assumptions

You have completed the lecture portion of this lesson and the previous practice. A text excerpt from the play Hamlet has been provided you as a test file in the root directory of the project. The contents of the files are as follows.

```
                 Enter Rosencrantz and Guildenstern.

Pol. Fare you well, my lord.
Ham. These tedious old fools!
Pol. You go to seek the Lord Hamlet. There he is.
Ros. [to Polonius] God save you, sir!

                                          Exit [Polonius].
Guil. My honour'd lord!
Ros. My most dear lord!
Ham. My excellent good friends! How dost thou, Guildenstern? Ah,
  Rosencrantz! Good lads, how do ye both?
Ros. As the indifferent children of the earth.
Guil. Happy in that we are not over-happy.
  On Fortune's cap we are not the very button.
Ham. Nor the soles of her shoe?
Ros. Neither, my lord.
Ham. Then you live about her waist, or in the middle of her
  favours?
Guil. Faith, her privates we.
Ham. In the secret parts of Fortune? O! most true! she is a
  strumpet. What news ?
Ros. None, my lord, but that the world's grown honest.
Ham. Then is doomsday near! But your news is not true. Let me
  question more in particular. What have you, my good friends,
  deserved at the hands of Fortune that she sends you to prison
  hither?
Guil. Prison, my lord?
Ham. Denmark's a prison.
Ros. Then is the world one.
Ham. A goodly one; in which there are many confines, wards, and
  dungeons, Denmark being one o' th' worst.
Ros. We think not so, my lord.
Ham. Why, then 'tis none to you; for there is nothing either good
  or bad but thinking makes it so. To me it is a prison.
Ros. Why, then your ambition makes it one. 'Tis too narrow for
your
  mind.
Ham. O God, I could be bounded in a nutshell and count myself a
  king of infinite space, were it not that I have bad dreams.
Guil. Which dreams indeed are ambition; for the very substance of
  the ambitious is merely the shadow of a dream.
Ham. A dream itself is but a shadow.
```

Tasks

1. Open the `LambdaFiles14-01Prac` project.

    - Select File > Open Project.
    - Browse to `/home/oracle/labs/14-NIO.2/practices/practice1`.
    - Select `LambdaFiles14-01Prac` and click Open Project.

2. Edit the `P01BufferedReader` class to perform the steps that follow.

3. Using a `BufferedReader` and a stream, read in and print out the `hamlet.txt` file.

4. The output should look like the original text provided above.

5. Edit the `P02NioRead` class to perform the steps that follow.

6. Using the `Path`, `File`, and `Files` classes and a stream to read and print the contents of the `hamlet.txt` file.

7. The output should look like the original text provided above.

8. Edit the `P03NioReadAll` class to perform the steps that follow.

9. Using the NIO features and streams, read the contents of the `hamlet.txt` file into an `ArrayList`.

10. Filter and print out the lines for Rosencrantz for example: `String.contains("Ros.")`. The output should look similar to the following:

```
=== Rosencrantz ===
  Ros. [to Polonius] God save you, sir!
  Ros. My most dear lord!
  Ros. As the indifferent children of the earth.
  Ros. Neither, my lord.
  Ros. None, my lord, but that the world's grown honest.
  Ros. Then is the world.
  Ros. We think not so, my lord.
  Ros. Why, then your ambition makes it one. 'Tis too narrow for
your
```

11. Filter and print out the lines for Guildenstern (`"Guil."`). The output should look similar to the following:

```
=== Guildenstern ===
  Guil. My honour'd lord!
  Guil. Happy in that we are not over-happy.
  Guil. Faith, her privates we.
  Guil. Prison, my lord?
  Guil. Which dreams indeed are ambition; for the very substance of
```

12. Edit the `P04NioReadAll` class to perform the steps that follow.

13. Using the NIO features and streams, read the contents of the `hamlet.txt` file into an `ArrayList`.

15. Filter and print out the word `"lord"`. Print a count of the number of times the word occurs. The output should look similar to the following:

```
=== Lord Count ===
lord.
lord!
lord!
lord.
lord,
lord?
lord.
Word count: 7
```

16. Filter and print out the word `"prison"`. Print a count of the number of times the word occurs. The output should look similar to the following:

```
=== Prison Count ===
prison
prison.
prison.
Word count: 3
```

Practice 14-2: Working with Directories

Overview

In this practice, list directories and files using new features found in Java 8.

Assumptions

You have completed the lecture portion of this lesson and the previous practice.

Tasks

1. Open the `LambdaDirectory14-02Prac` project.

   - Select File > Open Project.
   - Browse to `/home/oracle/labs/14-NIO.2/practices/practice2`.
   - Select `LambdaDirectory14-02Prac` and click Open Project.

2. Edit the `DirList` class to perform the steps that follow.

3. Read all the files in the current directory using the `list` method.

4. Print the results. The output should look similar to the following:

```
=== Dir list ===
./build
./hamlet.txt
./nbproject
./src
./manifest.mf
./build.xml
```

5. Edit the `DirWalk` class to perform the steps that follow.

6. Use the `Files.walk` method to read the directory tree for the project.

7. Print the results. The output should look similar to the following:

```
=== Dir walk ===
.
./build
./build/classes
./build/classes/.netbeans_automatic_build
./build/classes/.netbeans_update_resources
./build/classes/com
./build/classes/com/example
./build/classes/com/example/lambda
./build/classes/com/example/lambda/DirFind.class
./build/classes/com/example/lambda/DirList.class
./build/classes/com/example/lambda/DirWalk.class
./build/classes/com/example/lambda/Main.class
./build.xml
./hamlet.txt
./manifest.mf
./nbproject
./nbproject/build-impl.xml
./nbproject/genfiles.properties
./nbproject/private
./nbproject/private/private.properties
```

Practices for Lesson 14: Java File NIO2

```
./nbproject/private/private.xml
./nbproject/project.properties
./nbproject/project.xml
./src
./src/com
./src/com/example
./src/com/example/lambda
./src/com/example/lambda/DirFind.java
./src/com/example/lambda/DirList.java
./src/com/example/lambda/DirWalk.java
./src/com/example/lambda/Main.java
```

8. Next, walk the directory tree and filter the results so that only paths containing "build" are displayed.

9. The output should look similar to the following:

```
=== Dir build ===
./build
./build/classes
./build/classes/.netbeans_automatic_build
./build/classes/.netbeans_update_resources
./build/classes/com
./build/classes/com/example
./build/classes/com/example/lambda
./build/classes/com/example/lambda/DirFind.class
./build/classes/com/example/lambda/DirList.class
./build/classes/com/example/lambda/DirWalk.class
./build/classes/com/example/lambda/Main.class
./build.xml
./nbproject/build-impl.xml
```

10. Edit the DirFind class to perform the steps that follow.

11. Use the Files.find method to search the directory structure for entries that are directories.

12. Print the results. The output should look similar to the following:

```
=== Find all dirs ===
.
./build
./build/classes
./build/classes/com
./build/classes/com/example
./build/classes/com/example/lambda
./nbproject
./nbproject/private
./src
./src/com
./src/com/example
./src/com/example/lambda
```

Practices for Lesson 14: Java File NIO2

Chapter 14 - Page 8

Practices for Lesson 15: Concurrency

Chapter 15

Practices for Lesson 15: Overview

Practices Overview

In these practices, you will use the `java.util.concurrent` package and sub-packages of the Java programming language.

Practice 15-1: Summary Level: 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 simple multithreaded counting application. Instead of manually creating threads, you will leverage an `ExecutorService` from the `java.util.concurrent` package.

Tasks

1. Open the `ConCount15-01Prac` project as the main project.
   a. Select File > Open Project.
   b. Browse to /home/oracle/labs/15-Concurrency/practices/practice1.
   c. Select ConCount15-01Prac and click the Open Project button.
2. Expand the project directories.
3. Open the `CountRunnable` class in the `com.example` package.
4. Create a constructor to initialize the `count` and `threadName` variables.
5. Uncomment the `count` and `threadName` variables.
6. In the run method, setup a `for` loop to print out the thread name and each number counted.
7. Open the `Main` class in the `com.example` package.
8. Setup the `ExecutorService` in the main method using the `Executors` class and the `newCachedThreadPool` method.
9. Setup three `CountRunnable` objects to count to 20, named threads A, B, and C.
10. Shut down the `ExecutorService`.
11. Run the project. You should see each thread count to 20. Because of out of order processing, the counts of the three threads should be all jumbled together.

Practice 15-2: Detailed Level: 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 simple multithreaded counting application . Instead of manually creating threads, you will leverage an `ExecutorService` from the `java.util.concurrent` package.

Tasks

1. Open the `ConCount15-01Prac` project as the main project.

   a. Select File > Open Project.

   b. Browse to /home/oracle/labs/15-Concurrency/practices/practice1.

   c. Select ConCount15-01Prac and click the Open Project button.

2. Expand the project directories.

3. Open the `CountRunnable` class in the `com.example` package.

4. Create a constructor to initialize the `count` and `threadName` variables.

```
public CountRunnable(int count, String name){
    this.count = count;
    this.threadName = name;
}
```

5. Uncomment the `count` and `threadName` variables.

```
final int count;
final String threadName;
```

6. In the run method, set up a `for` loop to print out the thread name and each number counted.

```
for (int i = 1; i <= count; i++){
    System.out.println("Thread " + threadName +
        ": " + i);
}
```

7. Open the `Main` class in the `com.example` package.

8. Setup the `ExecutorService` in the main method using the `Executors` class and the `newCachedThreadPool` method.

```
ExecutorService es = Executors.newCachedThreadPool();
```

9. Setup three `CountRunnable` objects to count to 20, named threads A, B, and C.

```
es.submit(new CountRunnable(20,"A"));
es.submit(new CountRunnable(20,"B"));
es.submit(new CountRunnable(20,"C"));
```

10. Shut down the `ExecutorService`.

```
es.shutdown();
```

11. Run the project. You should see each thread count to 20. Because of out of order processing, the counts of the three threads should be all jumbled together.

Practices for Lesson 15: Concurrency

Practice 15-2: Summary Level: Create a Network Client 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 `ExecutorService15-02Prac` project as the main project.
    a. Select File > Open Project.
    b. Browse to `/home/oracle/labs/15-Concurrency/practices/practice2`.
    c. Select `ExecutorService15-02Prac` and 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.
    a. Add a constructor and a field to receive and store a `RequestResponse` reference.
    b. Implement the `Callable` interface with a generic type of `RequestResponse`.

    ```
    public class NetworkClientCallable implements
    Callable<RequestResponse>
    ```

    c. 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` instance for each network request.

e.   The servers should be running on localhost, ports 10000–10009.

f.   Submit each `NetworkClientCallable` to the ExecutorService. Store each `Future` in the `Map` created in step 7c.

g.   Shut down the `ExecutorService`.

h.   Await the termination of all threads within the `ExecutorService` for 5 seconds.

i.   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.

Practices for Lesson 15: Concurrency

Practice 15-2: Detailed Level: Create a Network Client 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 `ExecutorService15-02Prac` project as the main project.

   a. Select File > Open Project.

   b. Browse to /home/oracle/labs/15-Concurrency/practices/practice2.

   c. Select ExecutorService15-02Prac and 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 that implements the Callable interface. Use the notation for generics to define the Callable as of type RequestResponse.

   ```
   public class NetworkClientCallable implements
   Callable<RequestResponse>
   ```

   NetBeans shortcut: Right-click and select Fix Imports to add the necessary import statement.

   a. Add a constructor and a field named lookup of type `RequestResponse` to receive and store a `RequestResponse` reference during construction.

   NetBeans shortcut: Add the field first, as a private class field, then right-click and select Insert Code. Then Select Constructor. Select the lookup field and click Generate.

   b. Implement the `Callable` interface with a generic type of `RequestResponse`.

   NetBeans shortcut: Select the light bulb beside the class signature and click Implement all abstract methods.

   c. Remove the line of code in the generated `call` method.

d. 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.

```
try (Socket sock = new Socket(lookup.host, lookup.port);
   Scanner scanner = new Scanner(sock.getInputStream())) {
   lookup.response = scanner.next();
   return lookup;
}
```

e. Use the NetBeans hint above to add the necessary import statements.

f. Note: Click the lightbulb with the caution triangle next to the class field to add `final` to the class field instance.

g. Save the file.

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.

```
ExecutorService es = Executors.newCachedThreadPool();
```

c. Create a `Map` that will be used to tie a request to a future response.

```
Map<RequestResponse, Future<RequestResponse>> callables = new
HashMap<>();
```

d. Copy the following lines of the `for` loop and code that creates an instance of a `RequestResponse` from the commented out code:

```
String host = "localhost";
for (int port = 10000; port < 10010; port++) {
   RequestResponse lookup = new RequestResponse(host, port);
```

e. Add a line of code that creates an instance of a `NetworkClientCallable` and passes the instance of the `RequestResponse` object to it for each network request.

f. Submit each `NetworkClientCallable` to the ExecutorService. Store each `Future` in the `Map` created above.

g. Your complete for loop should look like this:

```
for (int port = 10000; port < 10010; port++) {
   RequestResponse lookup = new RequestResponse(host, port);
   NetworkClientCallable callable =
      new NetworkClientCallable(lookup);
   Future<RequestResponse> future = es.submit(callable);
   callables.put(lookup, future);
}
```

h. Shut down the `ExecutorService`.

i. Await the termination of all threads within the `ExecutorService` for 5 seconds. Recall from the lesson that `awaitTermination` method throws an `InterruptedException`, so use a try-catch block.

```
es.shutdown();

try {
  es.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException ex) {
  System.out.println("Stopped waiting early");
}
```

j. Loop through the `Future` objects stored in the `Map` created above. Use the `keyset` method to return and `Iterable` that contains the `RequestResponse` object.

k. Get the `Future<RequestResponse>` object from the `RequestResponse` object retrieved from the `Map`.

l. Print out the servers' response or an error message with the server details if there was a problem communicating with a server.

m. Your code should look similar to this:

```
for (RequestResponse lookup : callables.keySet()) {
  Future<RequestResponse> future = callables.get(lookup);
  try {
    lookup = future.get();
    System.out.println(lookup.host + ":" + lookup.port + " " +
                       lookup.response);
  } catch (ExecutionException | InterruptedException ex) {
    System.out.println("Error talking to " + lookup.host +
                       ":" + lookup.port);
  }
}
```

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.

Practices for Lesson 15: Concurrency

Practices for Lesson 16: The Fork-Join Framework

Chapter 16

Practices for Lesson 16: Overview

Practices Overview

In these practices, you will use the Fork-Join Framework.

Practice 16-1: Detailed Level: 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 `ForkJoinFindMax16-01Prac` project as the main project.

   - Select File > Open Project.
   - Browse to `/home/oracle/labs/16-ForkJoin/practices/practice1`.
   - Select `ForkJoinFindMax16-01Prac` and 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 `FindMaxTask` class is called.

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.
   - Take note of how the `compute` method splits the `data` array if the count of elements to process is too great.

5. Create a `RandomArrayAction` class in the `com.example` package.

   - Add four fields.

   ```
   private final int threshold;
   private final int[] myArray;
   private int start;
   private int end;
   ```

   - 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)
   ```

   - Modify the class signature to extend the `RecursiveAction` class from the `java.util.concurrent` package.

   Note: A `RecursiveAction` is used when a `ForkJoinTask` with no return values is needed.

- Add the `compute` method. Note that unlike the `compute` method from a `RecursiveTask`, the `compute` method in a `RecursiveAction` returns `void`.

```
protected void compute() { }
```

- Begin the `compute` method. If the number of elements to process is below the threshold, you should initialize the array.

```
if (end - start < threshold) {
  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.

- 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`.

```
} else {
  int midway = (end - start) / 2 + start;
  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`.

    - Your code should look like this:

    ```
    //          for (int i = 0; i < data.length; i++) {
    //              data[i] = ThreadLocalRandom.current().nextInt();
    //          }

    ForkJoinPool pool = new ForkJoinPool();

    RandomArrayAction action = new RandomArrayAction(data, 0,
                            data.length - 1, data.length / 16);
    pool.invoke(action);
    ```

7. Run the `ForkJoinFindMax16-01Prac` project by right-clicking the project and choosing *Run*.

Practices for Lesson 16: The Fork-Join Framework

Practices for Lesson 16: The Fork-Join Framework

Practices for Lesson 17: Parallel Streams

Chapter 17

Practices for Lesson 17: Parallel Streams

Practices for Lesson 17: Overview

Practice Overview

In these practices, explore the parallel stream options available in Java.

Old Style Loop

The following example iterates through an `Employee` list. Each member who is from Colorado and is an executive has their information printed out. In addition, the `sum` mutator is used to calculate the total amount of executive pay for the selected group.

A01OldStyleLoop.java

```
 9 public class A01OldStyleLoop {
10
11     public static void main(String[] args) {
12
13         List<Employee> eList = Employee.createShortList();
14
15         double sum = 0;
16
17         for(Employee e:eList){
18             if(e.getState().equals("CO") &&
19                 e.getRole().equals(Role.EXECUTIVE)){
20                 e.printSummary();
21                 sum += e.getSalary();
22             }
23         }
24
25         System.out.printf("Total CO Executive Pay: $%,9.2f %n", sum);
26     }
27
28 }
```

There are a couple of key points that can be made about the above code.

- All elements in the collections must be iterated through every time.
- The code is more about "how" information is obtained and less about "what" the code is trying to accomplish.
- A mutator must be added to the loop to calculate the total.
- There is no easy way to parallelize this code.

The output from the program is as follows.

Output

```
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Total CO Executive Pay: $370,000.00
```

Lambda Style Loop

The following example shows the new approach to obtaining the same data using lambda expressions. A stream is created, filtered, and printed. A `map` method is used to extract the salary data, which is then summed and returned.

A02NewStyleLoop.java

```
 9 public class A02NewStyleLoop {
10
11     public static void main(String[] args) {
12
13         List<Employee> eList = Employee.createShortList();
14
15         double result = eList.stream()
16             .filter(e -> e.getState().equals("CO"))
17             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18             .peek(e -> e.printSummary())
19             .mapToDouble(e -> e.getSalary())
20             .sum();
21
22         System.out.printf("Total CO Executive Pay: $%,9.2f %n",
result);
23     }
24
25 }
```

There are also some key points worth pointing out for this piece of code as well.

- The code reads much more like a problem statement.
- No mutator is needed to get the final result.
- Using this approach provides more opportunity for lazy optimizations.
- This code can easily be parallelized.

The output from the example is as follows.

Output

```
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Total CO Executive Pay: $370,000.00
```

Practices for Lesson 17: Parallel Streams

Streams with Code

So far all the examples have used lambda expressions and stream pipelines to perform the tasks. In this example, the `Stream` class is used with regular Java statements to perform the same steps as those found in a pipeline.

A03CodeStream.java

```
11 public class A03CodeStream {
12
13     public static void main(String[] args) {
14
15         List<Employee> eList = Employee.createShortList();
16
17         Stream<Employee> s1 = eList.stream();
18
19         Stream<Employee> s2 = s1.filter(
20             e -> e.getState().equals("CO"));
21
22         Stream<Employee> s3 = s2.filter(
23             e -> e.getRole().equals(Role.EXECUTIVE));
24         Stream<Employee> s4 = s3.peek(e -> e.printSummary());
25         DoubleStream s5 = s4.mapToDouble(e -> e.getSalary());
26         double result = s5.sum();
27
28         System.out.printf("Total CO Executive Pay: $%,9.2f %n",
result);
29     }
30
31 }
```

Even though the approach is possible, a stream pipeline seems like a much better solution.

The output from the program is as follows.

Output

```
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Total CO Executive Pay: $370,000.00
```

Making a Stream Parallel

Making a stream run in parallel is pretty easy. Just call the `parallelStream` or `parallel` method in the stream. With that call, when the stream executes it uses all the processing cores available to the current JVM to perform the task.

A04Parallel.java

```java
 9 public class A04Parallel {
10
11    public static void main(String[] args) {
12
13        List<Employee> eList = Employee.createShortList();
14
15        double result = eList.parallelStream()
16            .filter(e -> e.getState().equals("CO"))
17            .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18            .peek(e -> e.printSummary())
19            .mapToDouble(e -> e.getSalary())
20            .sum();
21
22        System.out.printf("Total CO Executive Pay: $%,9.2f %n",
result);
23
24        System.out.println("\n");
25
26        // Call parallel from pipeline
27        result = eList.stream()
28            .filter(e -> e.getState().equals("CO"))
29            .filter(e -> e.getRole().equals(Role.EXECUTIVE))
30            .peek(e -> e.printSummary())
31            .mapToDouble(e -> e.getSalary())
32            .parallel()
33            .sum();
34
35        System.out.printf("Total CO Executive Pay: $%,9.2f %n",
result);
36
37        System.out.println("\n");
38
39        // Call sequential from pipeline
40        result = eList.stream()
41            .filter(e -> e.getState().equals("CO"))
42            .filter(e -> e.getRole().equals(Role.EXECUTIVE))
43            .peek(e -> e.printSummary())
44            .mapToDouble(e -> e.getSalary())
45            .sequential()
46            .sum();
47
48        System.out.printf("Total CO Executive Pay: $%,9.2f %n",
result);
49    }
50 }
```

Remember, the last call wins. So if you call the sequential method after the parallel method in your pipeline, the pipeline will execute serially.

The following output is produced for this sample program.

Output

```
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Total CO Executive Pay: $370,000.00


Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Total CO Executive Pay: $370,000.00


Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Total CO Executive Pay: $370,000.00
```

Stateful Versus Stateless Operations

You should avoid using stateful operations on collections when using stream pipelines. The `collect` method and `Collectors` class have been designed to work with both serial and parallel pipelines.

A05AvoidStateful.java

```
11 public class A05AvoidStateful {
12
13     public static void main(String[] args) {
14
15         List<Employee> eList = Employee.createShortList();
16         List<Employee> newList01 = new ArrayList<>();
17         List<Employee> newList02 = new ArrayList<>();
18
19         eList.parallelStream() // Not Parallel. Bad.
20             .filter(e -> e.getDept().equals("Eng"))
21             .forEach(e -> newList01.add(e));
22
23         newList02 = eList.parallelStream() // Good Parallel
24             .filter(e -> e.getDept().equals("Eng"))
25             .collect(Collectors.toList());
26
27     }
28 }
```

Lines 19 to 21 show you how NOT to extract data from a pipeline. Your operations may not be thread safe. Lines 23 to 25 demonstrate the correct method for saving data from a pipeline using the `collect` method and `Collectors` class.

Deterministic and Non-Deterministic Operations

Most stream pipelines are deterministic. That means that whether the pipeline is processed serially or in parallel the result will be the same.

A06Determine.java

```
10 public class A06Determine {
11
12     public static void main(String[] args) {
13
14         List<Employee> eList = Employee.createShortList();
15
16         double r1 = eList.stream()
17             .filter(e -> e.getState().equals("CO"))
18             .mapToDouble(Employee::getSalary)
19             .sequential().sum();
20
21         double r2 = eList.stream()
22             .filter(e -> e.getState().equals("CO"))
23             .mapToDouble(Employee::getSalary)
24             .parallel().sum();
25
26         System.out.println("The same: " + (r1 == r2));
```

Practices for Lesson 17: Parallel Streams

```
27      }
28 }
```

The example shows that the result for a sum is the same that is processed using either highlighted method.

The output from the sample is as follows:

Output

```
The same: true
```

However, some operations are not deterministic. The findAny() method is a short-circuit terminal operation that may produce different results when processed in parallel.

A07DetermineNot.java

```
10 public class A07DetermineNot {
11
12     public static void main(String[] args) {
13
14         List<Employee> eList = Employee.createShortList();
15
16         Optional<Employee> e1 = eList.stream()
17             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18             .sequential().findAny();
19
20         Optional<Employee> e2 = eList.stream()
21             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
22             .parallel().findAny();
23
24         System.out.println("The same: " +
25             e1.get().getEmail().equals(e2.get().getEmail()));
26
27     }
28 }
```

The data set used in the example is fairly small therefore the two different approaches will often produce the same result. However, with a larger data set, it becomes more likely that the results produced will not be the same.

Reduction

The reduce method performs reduction operations for the stream libraries. The following example sums numbers 1 to 5.

A08Reduction.java

```
 9 public class A08Reduction {
10
11     public static void main(String[] args) {
12
13         int r1 = IntStream.rangeClosed(1, 5).parallel()
14             .reduce(0, (a, b) -> a + b);
15
16         System.out.println("Result: " + r1);
17
18         int r2 = IntStream.rangeClosed(1, 5).parallel()
19             .reduce(0, (sum, element) -> sum + element);
20
21         System.out.println("Result: " + r2);
22
23     }
```

Two examples are shown. The second example started on line 18 uses more description variables to show how the two variables are used. The left value is used as an accumulator. The value on the right is added to the value on the left. Reductions must be associative operations to get a correct result.

The output from both expressions should be the following:

Output

```
Result: 15
Result: 15
```

Practices for Lesson 17: Parallel Streams

Practice 17-1: Calculate Total Sales without a Pipeline

Overview

In this practice, calculate the sales total for Radio Hut using the Stream class and normal Java statements.

Assumptions

You have completed the lecture portion of this lesson and the previous practice.

Tasks

1. Open the `SalesTxn17-01Prac` project.

   - Select File > Open Project.
   - Browse to `/home/oracle/labs/ 17-ParallelStreams /practices/practice1`.
   - Select `SalesTxn17-01Prac` and click the Open Project button.

2. Expand the project directories.

3. Edit the `CalcTest` class to perform the steps in this practice.

4. Calculate the total sales for Radio Hut using the Stream class and Java statements.

Create a stream from `tList` and assign it to: `Stream<SalesTxn> s1`

Create a second stream and assign the results of the `filter` method for Radio Hut transactions: `Stream<SalesTxn> s2`

Create a third stream and assign the results from a `mapToDouble` method that returns the transaction total: `DoubleStream s3`

Sum the final stream and assign the result to: `double t1`.

5. Print the results.

   Hint: Be mindful of the method return types. Use the API doc to ensure that you are using the correct methods and classes to create and store results.

6. The output from your test class should be similar to the following:

```
=== Transactions Totals ===
Radio Hut Total: $3,840,000.00
```

Practice 17-2: Calculate Sales Totals using Parallel Streams

Overview

In this practice, calculate the sales totals from the collection of sales transactions.

Assumptions

You have completed the lecture portion of this lesson and the previous practice.

Tasks

1. Open the `SalesTxn17-02Prac` project.

    *   Select File > Open Project.
    *   Browse to `/home/oracle/labs/ 17-ParallelStreams /practices/practice2`.
    *   Select `SalesTxn17-02Prac` and click the Open Project button.

2. Expand the project directories.

3. Edit the `CalcTest` class to perform the steps in this practice.

4. Calculate the total sales for Radio Hut, PriceCo, and Best Deals.

    a. Calculate the Radio Hut total using the `parallelStream` method. The pipeline should contain the following methods: `parallelStream`, `filter`, `mapToDouble`, and `sum`.

    b. Calculate the PriceCo total using the `parallel` method. The pipeline should contain the following methods: `filter`, `mapToDouble`, `parallel`, and `sum`.

    c. Calculate the Best Deals total using the `sequential` method. The pipeline should contain the following methods: `filter`, `mapToDouble`, `sequential`, and `sum`.

5. Print the results.

6. The output from your test class should be similar to the following:

```
=== Transactions Totals ===
Radio Hut Total: $3,840,000.00
PriceCo Total: $1,460,000.00
Best Deals Total: $1,300,000.00
```

Practice 17-3: Calculate Sales Totals Using Parallel Streams and Reduce

Overview

In this practice, calculate the sales totals from the collection of sales transactions using the `reduce` method.

Assumptions

You have completed the lecture portion of this lesson and the previous practice.

Tasks

1. Open the `SalesTxn17-03Prac` project.

    *   Select File > Open Project.

    *   Browse to `/home/oracle/labs/ 17-ParallelStreams /practices/practice3`.

    *   Select `SalesTxn17-03Prac` and click the Open Project button.

2. Expand the project directories.

3. Edit the `CalcTest` class to perform the steps in this practice.

4. Calculate the total sales for PriceCo using the `reduce` method instead of `sum`.

    a. Your pipeline should consist of: `filter`, `mapToDouble`, `parallel`, and `reduce`.

    b. The reduce function can be defined as: `reduce(0, (sum, e) -> sum + e)`

5. In addition, calculate the total number of transactions for PriceCo using `map` and `reduce`.

    a. Your pipeline should consist of: `filter`, `mapToInt`, `parallel`, and `reduce`.

    b. To count the transactions, use: `mapToInt(t -> 1)`

    c. The `reduce` function can be defined as: `reduce(0, (sum, e) -> sum + e)`.

6. Print the results.

7. The output from your test class should be similar to the following:

```
=== Transactions Totals ===

PriceCo Total: $1,460,000.00
PriceCo Transactions:      4
```

Practices for Lesson 18: Building Database Applications with JDBC

Chapter 18

Practices for Lesson 18: 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 18-1: 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.

Note

The first time you run the JavaDB server you will get the following error message:
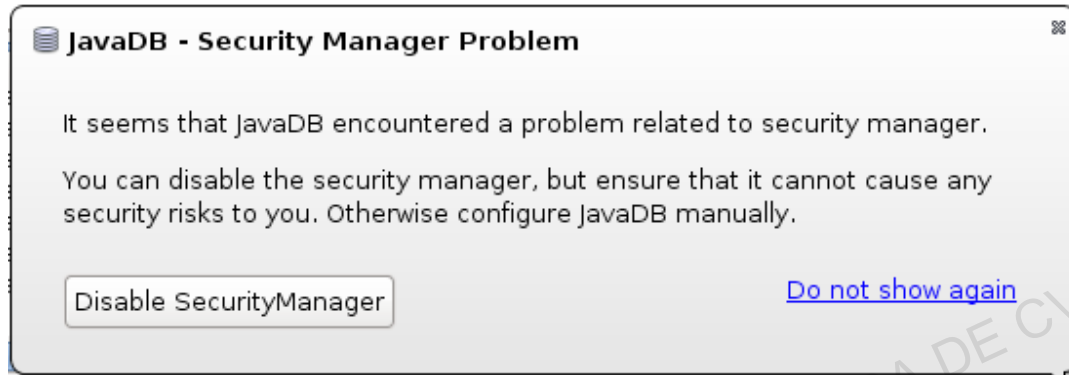
Go ahead and click the **Disable SecurityManager** button. This will enable you to complete the practices.

Tasks

1. Create the Employee Database by using the SQL script provided in the resource directory. Perform the following steps in NetBeans:
    a. Click the Services tab.
    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	tiger
Password	scott
Confirm Password	scott

    f. Click OK
    g. Right-click the connection that you created:
        `jdbc:derby://localhost:1527/EmployeeDB[tiger on TIGER]` and select Connect.
    h. Select File > Open File.
    i. Browse to `/home/oracle/labs/resources/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 `TIGER` schema is now created. Expand the `TIGER` 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 `SimpleJDBC18-01Prac` project and run it.

a. You should see all the records from the Employee table displayed.

Note: In case you get a broken reference link to Java DB driver error, perform the following steps:

    I. Right-click on the project and select properties.

    II. In the categories column select Libraries.

    III. Click Add Library and select Java DB Driver from the Available libraries

    IV. Click Add Library.

    V. Click OK.

3. Add a SQL command to add a new Employee record.

a. Modify the `SimpleJDBCExample` class to add a new Employee record to the database.

Note: If you run the application again, it will throw an exception, because this key already exists in 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 18-1: 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. Perform the following steps in NetBeans:

    a. Click Services tab.
    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	tiger
Password	scott
Confirm Password	scott

    f. Click OK.

    g. Right-click the connection that you created:
       `jdbc:derby://localhost:1527/EmployeeDB[tiger on TIGER]` and select Connect.

    h. Select File > Open File.

    i. Browse to `/home/oracle/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 `TIGER` schema is now created. Expand the `TIGER` 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 `SimpleJDBC18-01Prac` Project and run it.

   a. Select File > Open Project.

   b. Select `/home/oracle/labs/18-JDBC/practices/practice1/SimpleJDBC18-01Prac`.

   c. Click Open Project.

   d. Expand the Source Packages and look at the `SimpleJDBCExample.java`

   e. Run the project: Right-click the project and select Run, or click the Run icon, or press F6.

   f. You should see all the records from the Employee table displayed.

Note: In case you get a broken reference link to Java DB driver error, perform the following steps:

   VI. Right-click on the project and select properties.

   VII. In the categories column select Libraries.

   VIII. Click Add Library and select Java DB Driver from the Available libraries.

   IX. Click Add Library.

   X. Click OK.

3. 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 (400, '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.

Practices for Lesson 19: Localization

Chapter 19

Practices for Lesson 19: 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 19-1: 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 JDK8 API documentation.

Summary

Create a simple text-based date application that displays the following date information for today:

- Full date
- Long date
- Short date
- Medium date/time
- Medium time

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 ===
Full Date is: Tuesday, June 17, 2014
Long Date is: June 17, 2014
Short Date is: 6/17/14
Medium Date and Time is: Jun 17, 2014 10:51:09 AM
Medium Time is: 10:51:09 AM

--- 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

1. Open the `Localized19-01Prac` project in NetBeans.

    - Select File > Open Project.

    - Browse to `/home/oracle/labs/19-Localization/practices/practice1`.

    - Select `Localized19-01Prac` and click Open Project.

2. Edit the `DateApplication.java` file.

3. 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.

4. Add code to display the specified date formats (indicated with comments) and localized text.

5. Add code to change the `Locale` based on the user input.

6. Run the `DateApplication.java` file and verify that it operates as described.

Practice 19-1: 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 JDK8 API documentation.

Summary

Create a simple text-based date application that displays the following date information for today:

- Full date
- Long date
- Short date
- Medium date/time
- Medium time

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 ===
Full Date is: Tuesday, June 17, 2014
Long Date is: June 17, 2014
Short Date is: 6/17/14
Medium Date and Time is: Jun 17, 2014 10:51:09 AM
Medium Time is: 10:51:09 AM

--- 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

1. Open the `Localized19-01Prac` project in NetBeans.

   a. Select File > Open Project.

   b. Browse to `/home/oracle/labs/19-Localization/practices/practice1`.

   c. Select `Localized19-01Prac` and click Open Project.

2. Expand the project directories.
3. Edit the `DateApplication.java` file.

4. Open the `MessagesText.txt` file found in the `practices` directory for this practice in a text editor.

5. 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.

6. 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.

7. 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);
    }
```

Practices for Lesson 19: Localization

8. Add the code that displays the date information to the `printMenu` method.

```
    public void printMenu(){
        pw.println("=== Date App ===");

        // Full Date
        df =
DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL).withLocale(c
urrentLocale);
        pw.println(messages.getString("date1") + " " +
today.format(df));

        // Long Date
        df =
DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG).withLocale(c
urrentLocale);
        pw.println(messages.getString("date2") + " " +
today.format(df));

        // Short Date
        df =
DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT).withLocale(
currentLocale);
        pw.println(messages.getString("date3") + " " +
today.format(df));

         // Medium Date/Time
        df =
DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM).withLo
cale(currentLocale);
        pw.println(messages.getString("date4") + " " +
today.format(df));

         // Medium Time
        df =
DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM).withLocale
(currentLocale);
        pw.println(messages.getString("date5") + " " +
today.format(df));

            pw.println("\n--- Choose Language Option ---");
            pw.println("1. " + messages.getString("menu1"));
            pw.println("2. " + messages.getString("menu2"));
            pw.println("3. " + messages.getString("menu3"));
            pw.println("4. " + messages.getString("menu4"));
            pw.println("q. " + messages.getString("menuq"));
```

Practices for Lesson 19: Localization

```
           System.out.print(messages.getString("menucommand") + "
");
}
```

9. Run the `DateApplication.java` file and verify that it operates as described.

Practices for Lesson 19: Localization