



Java SE 11: Programming Complete

Activity Guide

D107120GC10 | 107911

Learn more from Oracle University at education.oracle.com



© 2018 International & Oracle Academy / Use Only

Author

Vasily Strelnikov

Technical Contributors and Reviewers

Joe Greenwald

Svetlana Savvina

Jacobo Marcos

Editors

Moushmi Mukherjee

Aju Kumar

Graphic Designer

Kavya Bellur

Publishers

Sujatha Nagendra

Veena Narasimhan

Pavithran Adka

Srividya Rameshkumar

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Disclaimer

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

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

Restricted Rights Notice

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

U.S. GOVERNMENT RIGHTS

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

Trademark Notice

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

1003242020

Table of Contents

Practices for Lesson 1: Introduction to Java	5
Practices for Lesson 1: Overview.....	6
Practice 1-1: Verify the JDK Installation.....	7
Practice 1-2: Create, Compile, and Execute a Java Application	9
Practices for Lesson 2: Primitive Types, Operators, and Flow Control Statements.....	13
Practices for Lesson 2: Overview.....	14
Practice 2-1: Manipulate with Primitive Types.....	15
Practice 2-2: Use the if/else and switch Constructs and a Ternary Operator	22
Practices for Lesson 3: Text, Date, Time, and Numeric Objects	31
Practices for Lesson 3: Overview.....	32
Practice 3-1: Explore String and StringBuilder Objects.....	33
Practice 3-2: Use BigDecimal Class and Format Numeric Values	38
Practice 3-3: Use and Format Date and Time Values.....	42
Practice 3-4: Apply Localization and Format Messages	46
Practices for Lesson 4: Classes and Objects.....	51
Practices for Lesson 4: Overview.....	52
Practice 4-1: Create the Product Management Application	53
Practice 4-2: Enhance the Product Class	66
Practice 4-3: Document Classes.....	73
Practices for Lesson 5: Improve Class Design	77
Practices for Lesson 5: Overview.....	78
Practice 5-1: Create Enumeration to Represent Product Rating.....	79
Practice 5-2: Add Custom Constructors to the Product Class.....	86
Practice 5-3: Make Product Objects Immutable	92
Practices for Lesson 6: Inheritance	95
Practices for Lesson 6: Overview.....	96
Practice 6-1: Create Food and Drink Classes That Extend Product	97
Practice 6-2: Override Methods and Use Polymorphism	106
Practice 6-3: Create Factory Methods	125
Practices for Lesson 7: Interfaces.....	131
Practices for Lesson 7: Overview.....	132
Practice 7-1: Design the Rateable Interface.....	133
Practice 7-2: Enable Products Review and Rating.....	141
Practice 7-3: Test the Product Review Functionality	152
Practices for Lesson 8: Arrays and Loops	155
Practices for Lesson 8: Overview.....	156
Practice 8-1: Allow Multiple Reviews for a Product.....	157

Practices for Lesson 9: Collections	163
Practices for Lesson 9: Overview.....	164
Practice 9-1: Organize Products and Reviews into a HashMap.....	165
Practice 9-2: Implement Review Sort and Product Search Features	174
Practices for Lesson 10: Nested Classes and Lambda Expressions.....	181
Practices for Lesson 10: Overview.....	182
Practice 10-1: Refactor ProductManger to use a Nested Class	183
Practice 10-1.A: Refactor ProductManger to use a Nested Class.....	184
Practice 10-1.B: Examine Solution for ProductManger refactored	194
Practice 10-2: Produce Customized Product Reports.....	197
Practices for Lesson 11: Java Streams API.....	201
Practices for Lesson 11: Overview.....	202
Practice 11-1: Modify ProductManager to Use Streams	203
Practice 11-2: Add Discount Per Rating Calculation	211
Practices for Lesson 12: Handle Exceptions and Fix Bugs	215
Practices for Lesson 12: Overview.....	216
Practice 12-1: Use Exception Handling to Fix Logical Errors.....	217
Practice 12-2: Add Text Parsing Operations	227
Practices for Lesson 13: Java IO API.....	239
Practices for Lesson 13: Overview.....	240
Practice 13-1: Print Product Report to a File.....	241
Practice 13-2: Bulk-Load Data from Files.....	248
Practice 13-3: Implement Memory Swap Mechanism	260
Practices for Lesson 14: Java Concurrency and Multithreading.....	267
Practices for Lesson 14: Overview.....	268
Practice 14-1: Redesign ProductManager as a Singleton.....	270
Practice 14-2: Ensure ProductManager Memory Safety	276
Practice 14-3: Simulate Concurrent Callers	286
Practices for Lesson 15: Java Modules	295
Practices for Lesson 15: Overview.....	296
Practice 15-1: Convert ProductManagement Application into a Module	297
Practice 15-2: Separate Application into Several Modules	303

Practices for Lesson 1: Introduction to Java

Practices for Lesson 1: Overview

Overview

In these practices, you explore your environment to develop Java applications. In Practice 1-1, you verify the Java Development Kit (JDK) installation and locate the practices folder structure. In Practice 1-2, you create, compile, and execute your first Java program.

```
JDK_HOME
  bin
    java
    javac
  conf
  include
  jmods
  legal
  lib
```



Java Development Kit (JDK) folder structure:

In this course environment, `JDK_HOME` refers to the `/usr/java/jdk-11.0.4` folder.

Practices folder structure:

`/home/oracle/labs/practice1/sources`

`/home/oracle/labs/practice1/classes`

Practice 1-1: Verify the JDK Installation

Overview

In this practice, you verify the Java Development Kit (JDK) installation and practices setup.

Assumptions

- JDK 11 is installed.
- JDK executables path is configured.
- The practices folder structure is created.

Tasks

1. Verify that JDK is installed and the jdk bin folder is added to the path.
 - a. Open Terminal.
 - b. Invoke Java Runtime and test the installed Java version: `java -version`.



- c. Invoke Java compiler: `javac`.

Notes

- JDK has already been installed on this computer. However, if you want to download and install JDK in your own environment, it is available from:
<https://www.oracle.com/technetwork/java/javase/downloads/index.html>
- JDK 11 is not the latest version of Java SE available. However, it is marked as (LTS) - Long Term Support, which means it is going to be supported by Oracle longer than other versions (12, 10, and 9).
- To invoke `java` or `javac` executables, your `JDK_HOME/bin` folder should be included in the `PATH` variable. This has already been configured on your practices machine.

2. Navigate to the course practices folder in the terminal window.
 - a. Change directory to the course practices folder: `cd labs/practice1`.
 - b. List the contents of the first lesson practice folder: `ls -al`.

The screenshot shows a Linux desktop environment. On the left is a Unity-style dock with icons for Home, Firefox Web Browser, Terminal (which is currently active), CDROM, and NetBeans. The terminal window is titled "oracle@edvmr1p0:~/labs/practice1". It displays Java compiler documentation and a command-line session:

```
Check that API used is available in the specified profile
--release <release>
    Compile for a specific VM version. Supported targets: 6, 7, 8, 9, 10, 11
-s <directory>           Specify where to place generated source files
-source <release>
    Provide source compatibility with specified release
--source-path <path>, -sourcepath <path>
    Specify where to find input source files
--system <jdk>|none      Override location of system modules
-target <release>        Generate class files for specific VM version
--upgrade-module-path <path>
    Override location of upgradeable modules
-verbose                  Output messages about what the compiler is doing
--version, -version       Version information
-Werror                   Terminate compilation if warnings occur

[oracle@edvmr1p0 ~]$ cd labs/practice1
[oracle@edvmr1p0 practice1]$ ls -al
total 16
drwxr-xr-x  4 oracle oracle 4096 Oct 22 10:41 .
drwxr-xr-x 25 oracle oracle 4096 Oct 22 10:25 ..
drwxr-xr-x  2 oracle oracle 4096 Oct 22 10:32 classes
drwxr-xr-x  2 oracle oracle 4096 Oct 22 10:32 sources
[oracle@edvmr1p0 practice1]$ █
```

Practice 1-2: Create, Compile, and Execute a Java Application

Overview

In this practice, you create, compile, and execute a Java application.

Assumptions

You have successfully completed Practice 1-1.

Tasks

1. Create the Java class `HelloWorld` in the `labs` package in the `sources` folder:

- a. Verify your current path in the terminal window using:

```
pwd
```

Note: Your current folder should be `/home/oracle/labs/practice1`. Change directory to this folder if this is not the case already.

- b. Create the Java class `HelloWorld` in the `labs` package in the `sources` folder:

```
gedit ./sources/labs/HelloWorld.java &
```

- c. Define the class structure of the `HelloWorld` class, as a member of the `labs` package in the editor:

```
package labs;
public class HelloWorld {
    // methods of this class will be added here
}
```

Note: The full path to the file containing the source code of the `HelloWorld` class is `/home/oracle/labs/practice1/source/labs/HelloWorld.java`. The first occurrence of the word "labs" is just a folder name that contains the entire set of practices for this course, while the second occurrence of the word "labs" is a package name. One is not in anyway linked to the other. This package name is not following the reverse-dns naming convention usually used for packages.

- d. Create a method to represent a default entry point within `HelloWorld` class:

```
public static void main(String[] args) {
    // logic of the method will be added here
}
```

- e. Add code to the `main` method to print a message "Hello " concatenated with the value of the first parameter:

```
System.out.println("Hello "+args[0]);
```

f. Save the HelloWorld.java file.



```
package labs;
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello " + args[0]);
    }
}
```

Java ▾ Tab Width: 8 ▾ Ln 4, Col 42 ▾ INS

2. Compile HelloWorld and store the compiled result in the classes folder.

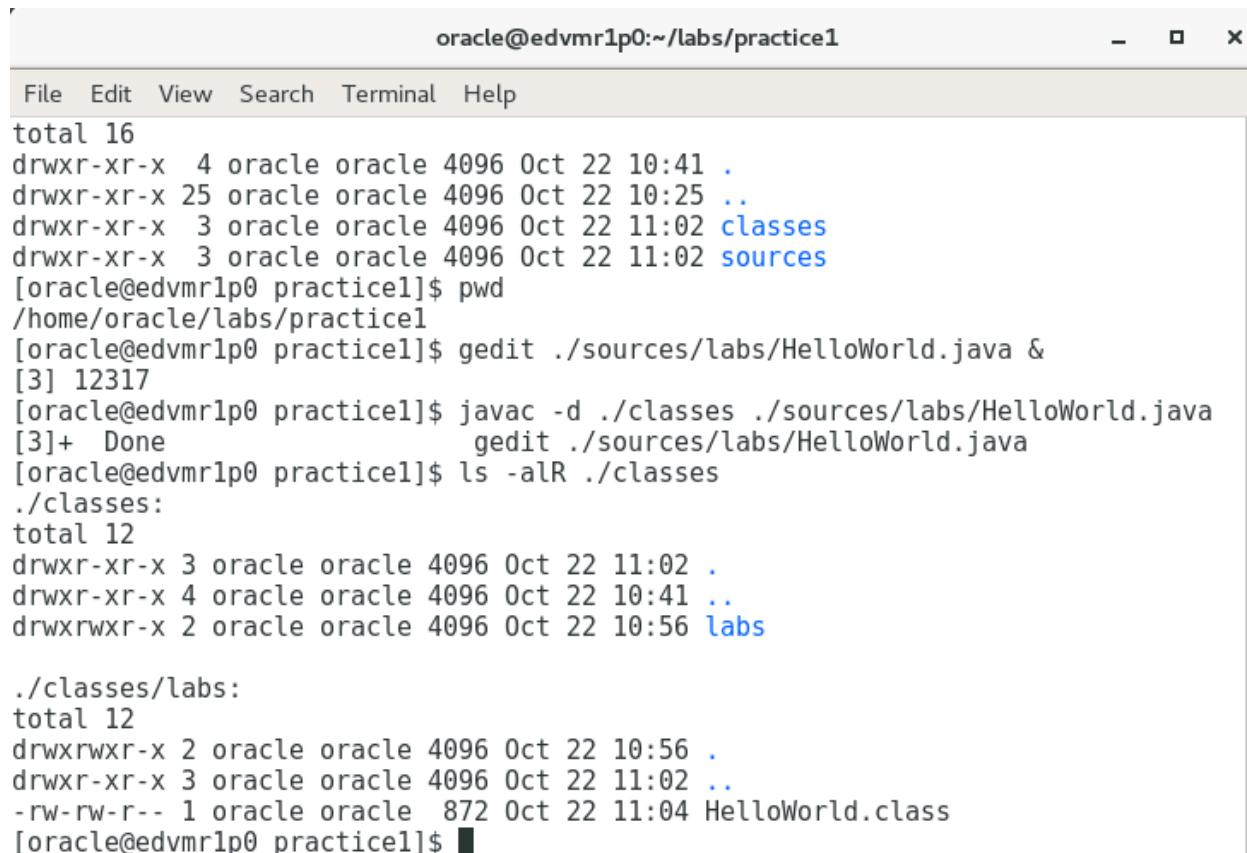
a. Use javac to compile the HelloWorld class:

```
javac -d ./classes ./sources/labs/HelloWorld.java
```

Note: You do not have to set the -cp (classpath) parameter for the compiler, because your class HelloWorld is not using any other classes, except those that are supplied with core JDK.

b. Verify the compilation results, stored under the classes folder:

```
ls -alR ./classes
```



```
oracle@edvmr1p0:~/labs/practice1
File Edit View Search Terminal Help
total 16
drwxr-xr-x 4 oracle oracle 4096 Oct 22 10:41 .
drwxr-xr-x 25 oracle oracle 4096 Oct 22 10:25 ..
drwxr-xr-x 3 oracle oracle 4096 Oct 22 11:02 classes
drwxr-xr-x 3 oracle oracle 4096 Oct 22 11:02 sources
[oracle@edvmr1p0 practice1]$ pwd
/home/oracle/labs/practice1
[oracle@edvmr1p0 practice1]$ gedit ./sources/labs/HelloWorld.java &
[3] 12317
[oracle@edvmr1p0 practice1]$ javac -d ./classes ./sources/labs/HelloWorld.java
[3]+ Done                  gedit ./sources/labs/HelloWorld.java
[oracle@edvmr1p0 practice1]$ ls -alR ./classes
./classes:
total 12
drwxr-xr-x 3 oracle oracle 4096 Oct 22 11:02 .
drwxr-xr-x 4 oracle oracle 4096 Oct 22 10:41 ..
drwxrwxr-x 2 oracle oracle 4096 Oct 22 10:56 labs

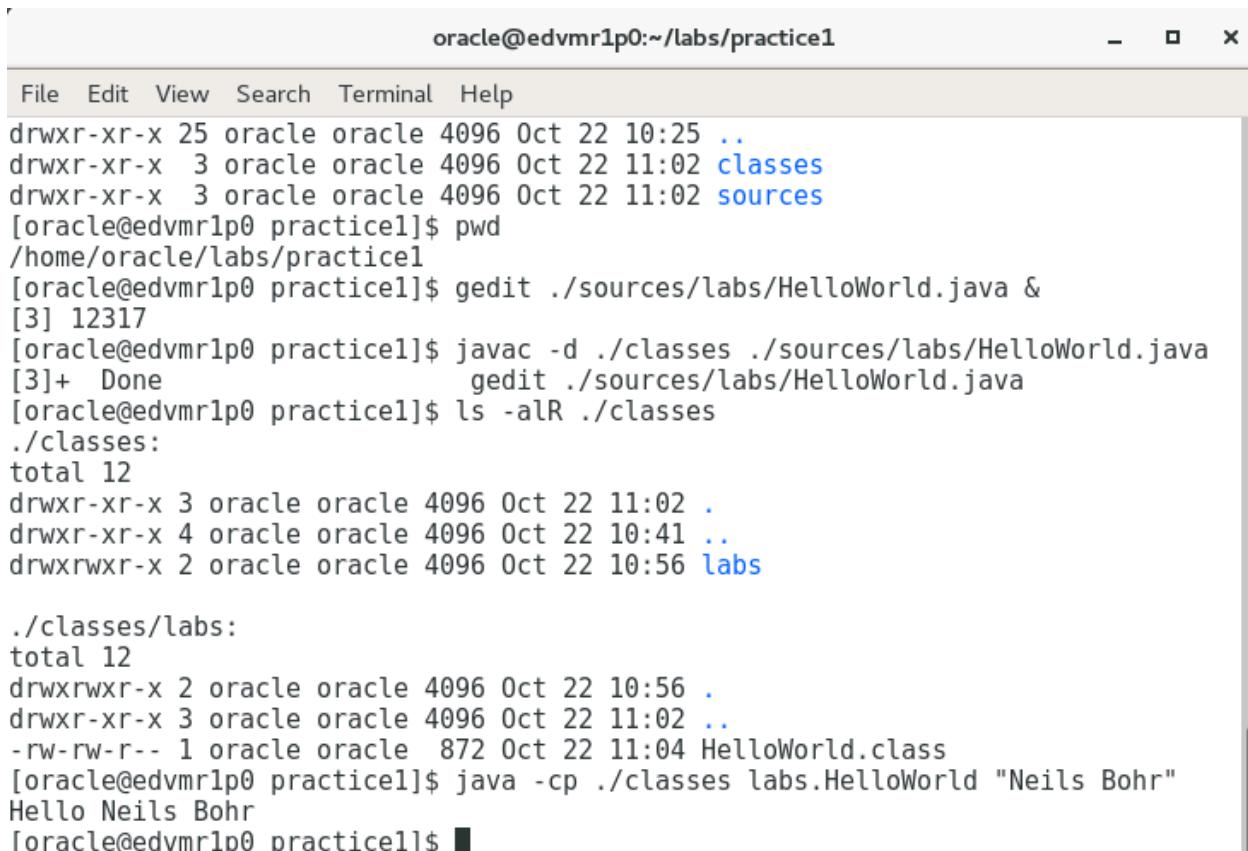
./classes/labs:
total 12
drwxrwxr-x 2 oracle oracle 4096 Oct 22 10:56 .
drwxr-xr-x 3 oracle oracle 4096 Oct 22 11:02 ..
-rw-rw-r-- 1 oracle oracle 872 Oct 22 11:04 HelloWorld.class
[oracle@edvmr1p0 practice1]$
```

Note: The `labs` folder representing your package has been created inside the `classes` folder, and `HelloWorld.class` has been placed inside that folder.

3. Run `HelloWorld` and print the `Hello Niels Bohr` message.

- a. Execute the `HelloWorld` class and pass `Niels Bohr` as a parameter.

```
java -cp ./classes labs.HelloWorld "Niels Bohr"
```



The screenshot shows a terminal window titled "oracle@edvmr1p0:~/labs/practice1". The window contains the following command-line session:

```

oracle@edvmr1p0:~/labs/practice1
File Edit View Search Terminal Help
drwxr-xr-x 25 oracle oracle 4096 Oct 22 10:25 ..
drwxr-xr-x  3 oracle oracle 4096 Oct 22 11:02 classes
drwxr-xr-x  3 oracle oracle 4096 Oct 22 11:02 sources
[oracle@edvmr1p0 practice1]$ pwd
/home/oracle/labs/practice1
[oracle@edvmr1p0 practice1]$ gedit ./sources/labs/HelloWorld.java &
[3] 12317
[oracle@edvmr1p0 practice1]$ javac -d ./classes ./sources/labs/HelloWorld.java
[3]+  Done                  gedit ./sources/labs/HelloWorld.java
[oracle@edvmr1p0 practice1]$ ls -alR ./classes
./classes:
total 12
drwxr-xr-x  3 oracle oracle 4096 Oct 22 11:02 .
drwxr-xr-x  4 oracle oracle 4096 Oct 22 10:41 ..
drwxrwxr-x  2 oracle oracle 4096 Oct 22 10:56 labs

./classes/labs:
total 12
drwxrwxr-x  2 oracle oracle 4096 Oct 22 10:56 .
drwxr-xr-x  3 oracle oracle 4096 Oct 22 11:02 ..
-rw-rw-r--  1 oracle oracle  872 Oct 22 11:04 HelloWorld.class
[oracle@edvmr1p0 practice1]$ java -cp ./classes labs.HelloWorld "Niels Bohr"
Hello Niels Bohr
[oracle@edvmr1p0 practice1]$ █

```

Notes

- Observe your program output printed on the console.
 - The reason the parameter value is enclosed in quotes is that space is used as a parameter separator.
- b. You can now close the `gedit` window where you have edited the `HelloWorld.java` source code.

Practices for Lesson 2: Primitive Types, Operators, and Flow Control Statements

Practices for Lesson 2: Overview

Overview

In these practices, you use JShell to explore Java primitive types and organize program flow with `if/else` and `switch` flow control constructs.

```
jshell> char x = 'a', y = 'b';
x ==> 'a'
y ==> 'b'

jshell> if (x < y) {
    ...>
```

Practice 2-1: Manipulate with Primitive Types

Overview

In this practice, you declare primitive variables, use assignment, and arithmetic operators.

Assumptions

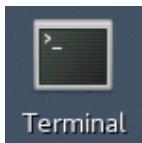
- JDK 11 is installed.
- JDK executables path is configured.
- The practices folder structure is created.

Tasks

1. Prepare the practice environment.

- a. Open the terminal window.

(You may continue to use the open terminal window from the previous practice.)



- b. Change folder to `/home/oracle/labs/practice2`.

(Your path for the `cd` command may vary depending on which folder is your current folder.)

```
cd /home/oracle/labs/practice2
```

- c. Invoke the JShell tool:

```
jshell
```

```
jshell> /exit
| Goodbye
[oracle@edvmr1p0 ~]$ jshell
| Welcome to JShell -- Version 11.0.4
| For an introduction type: /help intro

jshell>
```

Notes

- In case you need to leave JShell, enter the `/exit` command.
- You can also enter `/help` to get a list of JShell commands.

2. Declare numeric primitives and perform type-casting and arithmetic operations.

- a. Declare three `byte` variables `x`, `y`, and `z`. Immediately upon declaration, initialize these variables as first three prime numbers (2, 3, and 5):

```
byte x = 2, y = 3, z = 5;
```

- b. Recalculate a value of `z` by adding `x` and `y` values.

```
z = (byte) (x+y);
```

Note: Remember that any arithmetic operation on types smaller than `int` results in an `int` type. Therefore, type-casting will be required.

- c. Divide `x` by `y` and assign the result to a `float` variable `a`:

```
float a = (float)x/y;
```

Notes

- Without casting either `x` or `y` to `float` type, division operation would produce an `int` result.
- You need to cast not the overall result of the division `(float) (x/y)`, but rather any of the participants, in order to get actual floating point result.

- d. Divide `x` by `y` and assign a result to a `double` variable `b`:

```
double b = (double)x/y;
```

Note: Observe how `double` number 64-bit capacity allows for the allocation of more decimal digits of precision.

- e. Assign variable `a` value (which is a `float` type variable created in an earlier practice step) to variable `b` and observe the rounding side effects:

```
b = a;
```

Note: Although the assignment of a smaller type value (`float` variable) to a larger type (`double` variable) works and does not require type-casting, there is a rounding problem that occurs beyond the original 32-bit capacity of a number. This rounding problem is in fact already resolved by existing Java API classes (for example, `BigDecimal`). However, this is covered later in the lesson titled "Text, Date, Time, and Numeric Objects."

- f. Round variable `b` value to 3 decimal digits and assign the result to `float` variable `c`.

Hints

- Use the `round` method of a `Math` class to perform rounding.
- The `round` method always rounds values to a whole number (`int` or `long`). Therefore, to get 3 decimal digits, you need to multiply the number by a 10^3 (1000) and then divide the result of the rounding by the same amount.
- Remember that the result of rounding is a whole number (`int` or `long`). Therefore, make sure that when you divide it by the required amount, at least one participant of this operation is a `float` number, in order to ensure that you get a `float` type result.

```
float c = Math.round(b*1000)/1000F;
```

Note: Observe how `double` number 64-bit capacity allows for the allocation of more decimal digits of precision.

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

```
[oracle@edvmr1p0 practice2]$ jshell
| Welcome to JShell -- Version 11.0.4
| For an introduction type: /help intro

jshell> byte x = 2, y = 3, z = 5;
x ==> 2
y ==> 3
z ==> 5

jshell> z = (byte)(x+y);
z ==> 5

jshell> float a = (float)x/y;
a ==> 0.6666667

jshell> double b = (double)x/y;
b ==> 0.6666666666666666

jshell> b = a;
b ==> 0.6666666865348816

jshell> float c = Math.round(b*1000)/1000F;
c ==> 0.667
```

3. Declare char primitives and perform type-casting and arithmetic operations.

- Declare three `char` variables `a1`, `a2`, and `a3` initialized with 'a' character value as just a character literal, ASCII character code (141), and as unicode code (61).

```
char a1 = 'a', a2='\141', a3='\u0061';
```

Note: In fact all three variables have exactly the same value. They store the code for 'a' character. You can choose to represent this value in ASCII or unicode.

- Assign the `a1` variable value to a new `int` variable `i`:

```
int i = a1;
```

Notes

- No actual type casting is required to assign a `char` value to an `int` type variable, because `char` is merely storing a character code that is actually a number.
- It may be interesting to see what is the octal and hex equivalents of this integer value. You can convert the variable `i` to text that represents its value as octal (base 8) and hex (base 16) numbers:

```
Integer.toOctalString(i);
Integer.toHexString(i);
```

- These two statements produce an output of 141 and 61.
- Apparently an ASCII character code is an octal number and unicode is hex.
- `Integer` is one of primitive wrapper classes supplied with JDK. Details of what wrapper classes are and how to use them are covered in the lesson titled "Text, Date, Time, and Numeric Objects."

- c. Declare two `int` variables `i1` and `i2` initialized with `0141` and `0x61` numeric literal values.

```
int i1 = 0141, i2= 0x61;
```

Note: You can see that these values correspond to decimal value of 97.

- d. Assign `i1` variable value to a new `int` variable `a4`.

```
char a4 = (char)i1;
```

Note: This time type casting is required, because `int` value could be up to 32 bits in size and `char` capacity is only 16 bits.

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

```
jshell> char a1 = 'a', a2='\141', a3='\u0061';
a1 ==> 'a'
a2 ==> 'a'
a3 ==> 'a'

jshell> int i = a1;
i ==> 97

jshell> int i1 = 0141, i2= 0x61;
i1 ==> 97
i2 ==> 97

jshell> char a4 = (char)i1;
a4 ==> 'a'
```

4. Write a check using a remainder of division (modulus) operator to determine if a given `char` value is an even or odd character in the alphabet. Consider that character codes for '`a`' and '`z`' have decimal integer values of 97 and 122.

- a. Declare a variable `someChar` type of `char` and assign any character value to it between `a` and `z` inclusive.

```
char someChar = 'k';
```

- b. Apply remainder of division (modulus) operator to `someChar` to determine if it is divisible by 2 (even number) without remainder of division. Compare the remainder of division to zero. Assign the result of the calculation to a new boolean variable called `isEven`.

```
boolean isEven = (someChar%2 == 0);
```

Notes

- JShell allows you to write and evaluate expressions directly, without a need to create extra intermediate variables, so the following code would actually produce an identical verification result: `'k'%2 == 0`;
- Also, the use of round brackets `()` around the boolean expression is actually optional, but it may enhance code readability.

Note: The following screenshot (provided for verification purposes) shows all actions required by this practice segment:

```
jshell> char someChar = 'k';
someChar ==> 'k'

jshell> boolean isEven = (someChar%2 == 0);
isEven ==> false
```

5. Calculate the next and previous characters from a given character.

- a. Set variable `someChar` type of `char` to any char value between '`a`' and '`y`' inclusive (just before the last character in the alphabet).

```
someChar = 'k';
```

- b. Create variable `nextChar` type of `char` and set it to the value of the next character that you calculate from the value of the `someChar` variable.

```
char nextChar = (char)(someChar+1);
```

Notes

- It is interesting to see which character is going to be derived if the value of `someChar` is set to '`z`'. Your code basically goes beyond alphabetic characters and returns a next char symbol value, which in ASCII is a '`{`' symbol.
- You will get an error if you try to assign `someChar+1` to the `nextChar` variable without type casting. That is because any arithmetic operation on types smaller than `int` (in this case, a `char`) will result in an `int` value and the statement here is trying to assign it to a `char` variable.
- c. Try another approach at calculating the next character using an increment operator. Reassign the `nextChar` variable to the value of the next character that you calculate from the value of `someChar` variable.

```
char nextChar = ++someChar;
```

Notes

- No type casting is required this time, because operators such as `++` or `--` recursively modify same variable, without changing its type.
- If you repeat this line of code again, notice that a value of the variable keeps progressing to the next char. (You can simply press the up arrow on the keyboard to invoke the previously executed JShell commands.)
- d. Set `someChar` to the value of '`b`' .

```
someChar = 'b';
```

- e. Decrement a value of `someChar` variable.

```
--someChar;
```

Notes

- There is no need to create additional variables to hold the next or previous char values when you derive them using increment or decrement operators, because they recursively modify the original variable anyway.
- If you repeat this line of code again several times, notice that a value of the variable keeps progressing to the previous character and eventually arrives at uppercase 'Z'.

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

```
jshell> char someChar = 'k';
someChar ==> 'k'

jshell> boolean isEven = (someChar%2 == 0);
isEven ==> false

jshell> someChar = 'k';
someChar ==> 'k'

jshell> char nextChar = (char)(someChar+1);
nextChar ==> 'l'

jshell> char nextChar = ++someChar;
nextChar ==> 'l'

jshell> someChar = 'b';
someChar ==> 'b'

jshell> --someChar;
$22 ==> 'a'
```

6. Calculate the number of symbols (char codes) between a pair of characters.

- a. Define two char variables for the uppercase 'A' and the lowercase 'a'

```
char upperA = 'A', lowerA = 'a';
```

- b. Calculate how many symbols are between lower and uppercase a characters.

```
int distance = lowerA-upperA;
```

Note: Equivalent upper and lowercase letters are 32 symbols apart in the English alphabet, according to the ASCII character encoding.

- c. Set someChar to the value of 'h' .

```
someChar = 'h';
```

- d. Create another char variable called upperSomeChar to contain an uppercase equivalent of the someChar variable:

```
char upperSomeChar = (char)(someChar-32);
```

Note: Knowing a distance between upper and lowercase characters for a given alphabet (in this case, expressed as English ASCII char codes) can be helpful in writing character case conversions. Of course, such algorithms are already available in existing JDK classes. In this case, a String class has methods `toUpperCase` and `toLowerCase` that can convert text to a required case. However, the use of the String class and text formatting is covered in the lesson titled "Text, Date, Time, and Numeric Objects."

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

```
jshell> char upperA = 'A', lowerA = 'a';
upperA ==> 'A'
lowerA ==> 'a'

jshell> int distance = lowerA-upperA;
distance ==> 32

jshell> someChar = 'h';
someChar ==> 'h'

jshell> char upperSomeChar = (char)(someChar-32);
upperSomeChar ==> 'H'
```

Practice 2-2: Use the `if/else` and `switch` Constructs and a Ternary Operator

Overview

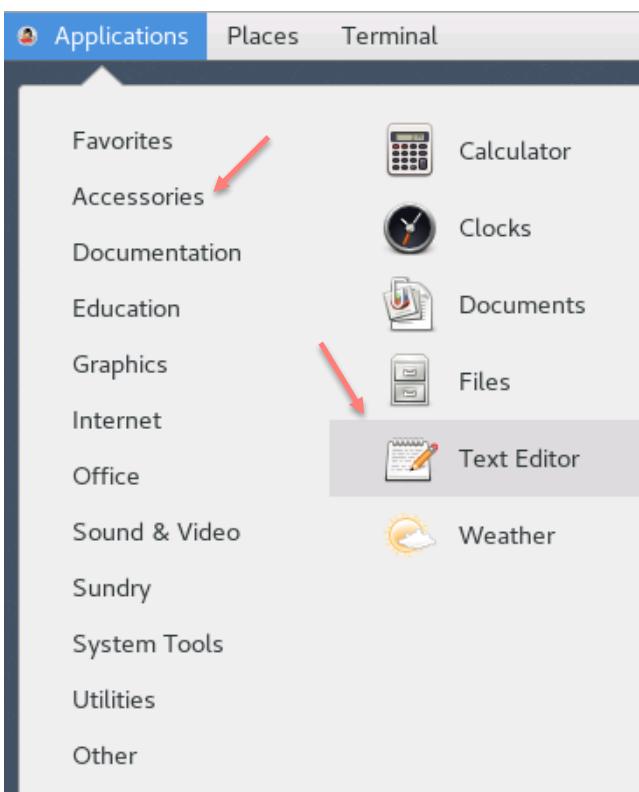
In this practice, you use `if/else` and `switch` constructs to control algorithm flow. Use the ternary operator to perform conditional assignments.

Tasks

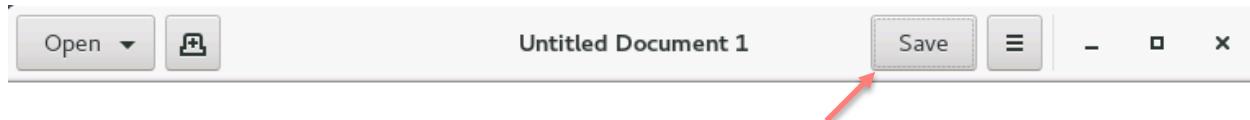
1. Write conditional logic using `if/else` statement.
 - a. Create a new text file, call it `script1.txt`, and save to the `/home/oracle/labs/practice2` folder.

Hints

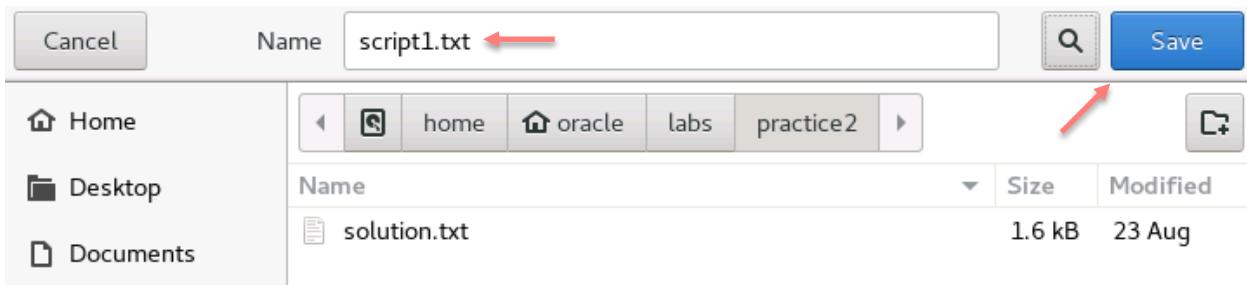
- Open Text Editor (available via Applications > Accessories menu).



- Click the "Save" button.



- Navigate to the /home/oracle/labs/practice2 folder and enter script1.txt as the filename and click the Save button.



- In this text file, create an `if/else` statement that checks if a given character is in lowercase (you could use previously defined `someChar` variable) and convert it to the opposite case.

Hints

- Remember the distance between upper and lowercase characters is 32 symbols.
- You need to test if a given character value is either within the `a-z` or the `A-Z` range and reassign the character value based on whenever this condition evaluates as true or false.
- You can use compound assignment to reassign the `someChar` variable.
- Carriage returns and code indentations in Java are not meaningful, but to make code more readable, you can write it over multiple lines and with indentations.

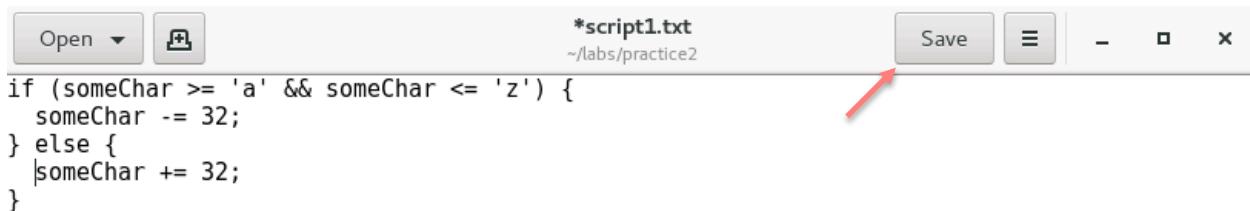
```
char someChar = 'q';
if (someChar >= 'a' && someChar <= 'z') {
    someChar -= 32;
} else {
    someChar += 32;
}
```

Notes

- If you choose to use a regular arithmetic operator, then you will have to perform the type-casting of the result of the operation back to char type:


```
someChar = (char) (someChar-32);
```
- It is rather convenient that Java compiler figures out that such type-casting should be performed implicitly in case you use one of the compound assignment operators.
- Code can be typed directly in JShell (rather than in a script file). In this case, if the line ends on the opening curly bracket `{` symbol, JShell would not immediately try to evaluate the code you've written, but would await further input and a closure of the code block with `}` curly bracket symbol.

- c. Save the file.



```
if (someChar >= 'a' && someChar <= 'z') {
    someChar -= 32;
} else {
    someChar += 32;
}
```

Note: Do not close this file; just switch back to the terminal window where you run JShell.

- d. Execute this code in the JShell:

```
/open script1.txt
```

Note: To observe what code was loaded, type from /list in JShell.

- e. To observe the result of the if/else construct execution, simply type someChar variable and JShell will echo its value back:

```
someChar
```

- f. Modify the script1.txt file and implement a more sophisticated logic that tests if the character is actually a letter between 'a' and 'z' and between 'A' and 'Z' to exclude all cases when the character is not a letter, but a special symbol or a numeric character instead. Only perform case conversion for those characters that are actually letters.

Hints

- Switch back to text editor to modify the existing script1.txt file.
- There is no need to declare the variable someChar again.
- An if statement can be embedded inside another if or an else block.

```
if (someChar >= 'a' && someChar <= 'z') {
    someChar -= 32;
} else {
    if (someChar >= 'A' && someChar <= 'Z') {
        someChar += 32;
    }
}
```

Note: There could be alternative solutions to this task. For example, you could check if the value is or is not a letter and only then proceed to perform checks of which kind of letter it actually is.

- g. Save the file.

- h. Execute this code in JShell.

```
/open script1.txt
```

- i. To observe the result, simply type someChar variable to echo its value back:

```
someChar
```

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

```
jshell> /open script1.txt
```

```
jshell> someChar
someChar ==> 'Q'
```

```
jshell> /open script1.txt
```

```
jshell> someChar
someChar ==> 'q'
```

2. Write a conditional variable assignment using a ternary operator.

- Repeat the same calculation as the earlier practice step (1.b), but using the ternary operator instead of the `if/else` statement.

Hints

- The use of compound assignment would not be possible with the ternary operator, so you would have to write explicit arithmetics with type-casting.
- There is no need to modify and reload the script file. You could just put the required code directly into JShell.

```
someChar = (someChar >= 'a' && someChar <= 'z') ?
    (char)(someChar-32) : (char)(someChar+32);
```

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

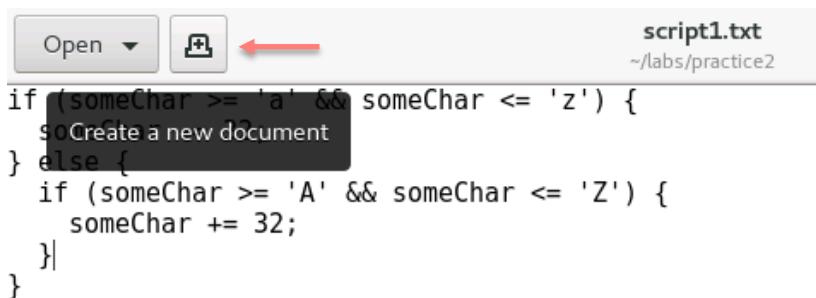
```
jshell> someChar = (someChar >= 'a' && someChar <= 'z') ?
    ...> (char)(someChar-32) : (char)(someChar+32);
someChar ==> 'Q'
```

3. Write the switch construct to calculate and add compound interest to the given amount of money for the specified period of time.

- a. Create a new text file, call it `script2.txt`, and save it to the `/home/oracle/labs/practice2` folder.

Hints

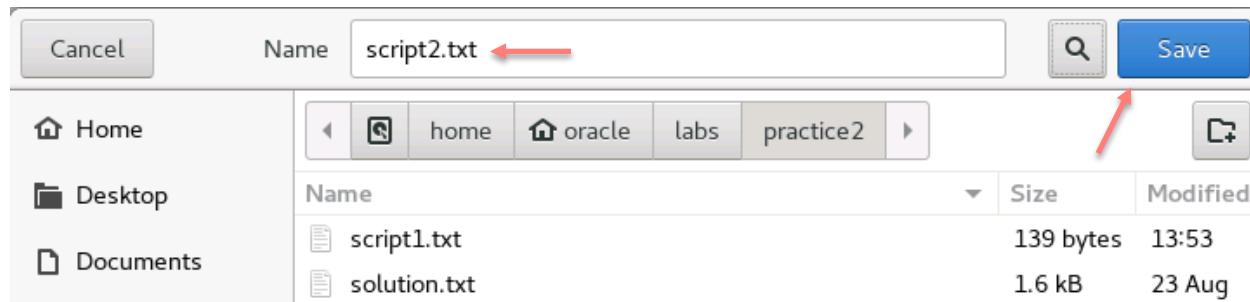
- Switch to the Text Editor window.
- Click the "New Document" button in Text Editor.



```
script1.txt
~/labs/practice2

if (someChar >= 'a' && someChar <= 'z') {
    someChar += 32;
} else {
    if (someChar >= 'A' && someChar <= 'Z') {
        someChar += 32;
    }
}
```

- Click the "Save" button.
- Type `script2.txt` as the file name.
- Make sure that the file is located in the `/home/oracle/labs/practice2` folder.



- Click the "Save" button again.
- b. In this text file, declare and initialize the following variables:
- An `int` variable to represent a period of time and set its value to 1
 - A `float` variable to represent monetary amount and set its value to 10
 - Another `float` variable to represent interest rate and set its value to `0.05F` indicating a 5% interest rate

```
int period = 1;
float amount = 10;
float rate = 0.05F;
```

- c. Create switch construct with three cases to represent periods of 3, 2, and 1 month in descending order.

```
switch (period) {
    case 3:
    case 2:
    case 1:
}
```

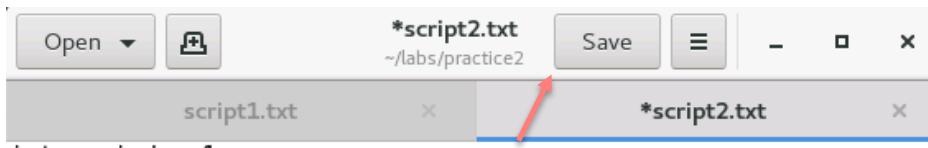
- d. To each of the cases, add a formula to calculate a new amount value with the interested added to it.

Hints

- Amount multiplied to the rate would produce an interest value, which then has to be added to the amount.
- Repeating the same formula in every case would repeat the calculation, with the growing amount figure for every subsequent period represented by consecutive cases.

```
amount += amount * rate;
```

- e. Save the file.



```
int period = 1;
float amount = 10;
float rate = 0.05F;
switch (period) {
    case 3:
        amount += amount * rate;
    case 2:
        amount += amount * rate;
    case 1:
        amount += amount * rate;
}
```

- f. Execute this code in JShell:

```
/open script2.txt
```

Note: To observe what code was loaded, type from /list in JShell.

- g. Output the current value of the amount variable:

```
amount
```

- h. Switch back to the Text Editor and modify the script2.txt file.

- i. Modify the period length in the script2.txt file to be equal to 3

```
int period = 3;
```

- j. Save the script2.txt file in Text Editor and reload this file in JShell.

```
/open script2.txt
```

- k. Output the current value of the amount variable:

```
amount
```

Note: Absence of break statements inside individual case statements means that once an algorithm jumps to a specific case, it just continues to subsequent cases until it reaches the end of the switch construct.

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

```
jshell> /open script2.txt

jshell> amount
amount ==> 10.5

jshell> /open script2.txt

jshell> amount
amount ==> 11.57625
```

3. Modify the switch statement so that for period of 4 months, no compound interest is to be applied, but instead the interest is to be calculated just once and should be set at 20% exactly.

Hints

- An extra case condition would be required.
- This new case should apply fix interest just once, so no subsequent cases are to be executed.
- a. Modify the `script2.txt` file to add an extra case to reflect a period of 4 months. This new case should be added at the start of the switch block, before other cases.

```
case 4:
    amount += amount * 0.2F;
    break;
```

- b. Modify the period length to be equal to 4:
- int period = 4;
- c. Save and reload the `script2.txt` file in JShell:
- `/open script2.txt`
- d. Output the current value of the amount variable:

```
amount
```

- e. This is the end of the practice; you may now close JShell:
/exit
- f. You can also close the Text Editor.

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

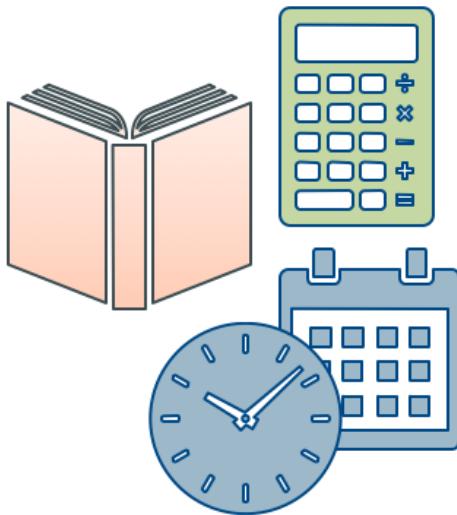
```
jshell> /open script2.txt  
  
jshell> amount  
amount ==> 12.0  
  
jshell> /exit  
| Goodbye  
[oracle@edvmr1p0 practice2]$ █
```


Practices for Lesson 3: Text, Date, Time, and Numeric Objects

Practices for Lesson 3: Overview

Overview

In these practices, you use JShell to explore handling of text, numeric, date, and time objects and apply localization and format values.



Practice 3-1: Explore String and StringBuilder Objects

Overview

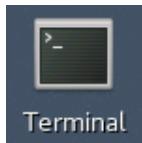
In this practice, you declare, initialize, and process String variables and explore String internment. You also create mutable text objects using StringBuilder.

Assumptions

- JDK 11 is installed.
- JDK executables path is configured.
- The practices folder structure is created.

Tasks

1. Prepare the practice environment.
 - a. Open the terminal window.
(You may continue to use the terminal window open from the previous practice.)



- b. Change folder to /home/oracle/labs/practice3.
cd /home/oracle/labs/practice3
- c. Invoke the JShell tool:
jshell

```
jshell> /exit
| Goodbye
[oracle@edvmr1p0 ~]$ jshell
| Welcome to JShell -- Version 11.0.4
| For an introduction type: /help intro

jshell>
```

2. Explore String internment.

- a. Declare the String object "Tea" and assign it to the teaTxt variable:

```
String teaTxt = "Tea";
```

- b. Declare the String object "Tea" and assign it to the variable b:

```
String b = "Tea";
```

- c. Test if both of these variables (teaTxt and b) are in fact referencing the same String object:

```
teaTxt == b;
```

Note: Because of automatic String internment, only one copy of a String object with a value of "Tea" has been placed in memory and both of the variables (teaTxt and b) are referencing the same object.

- d. Declare the String object "Tea" using the new operator and assign it to the variable c:

```
String c = new String("Tea");
```

- e. Test if the teaTxt variable references the same String object as the variable c:

```
teaTxt == c;
```

Note: Remember that using the new operator to create String objects is not recommended because it is disabling automatic String internment. Essentially by using the new operator, you have forced Java Runtime to allocate memory to store additional, separate copy of a "Tea" String object.

- f. Invoke the intern operation upon c object and then test if String c has been interned, by comparing teaTxt and c objects again:

```
c.intern();
```

```
teaTxt == c;
```

Note: The String referenced by the c variable has not been changed in any way, because all String objects are immutable.

- g. Declare a new String variable d and assign to it a value returned by the intern operation invoked upon the c object. Test if String d has been interned by comparing teaTxt and d objects:

```
String d = c.intern();
```

```
teaTxt == d;
```

Note: String referenced by the d variable has been interned. That is because intern operation simply returned a reference to the existing copy of a "Tea" object that is already referenced by variables teaTxt and b.

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

```
[oracle@edvmr1p0 practice3]$ jshell
| Welcome to JShell -- Version 11.0.4
| For an introduction type: /help intro

jshell> String teaTxt = "Tea";
teaTxt ==> "Tea"

jshell> String b = "Tea";
b ==> "Tea"

jshell> String c = new String("Tea");
c ==> "Tea"

jshell> teaTxt == b;
$4 ==> true

jshell> teaTxt == c;
$5 ==> false

jshell> c.intern();
$6 ==> "Tea"

jshell> teaTxt == c;
$7 ==> false

jshell> String d = c.intern();
d ==> "Tea"

jshell> teaTxt == d;
$9 ==> true
```

3. Use String operations to concatenate text values, create substrings, search through the text, and change text case.

- a. Concatenate the `teaTxt` object with a space (represented by a `char` primitive) and then concatenate with object `b`. Assign the result `String` variable `c`.

```
c = teaTxt+ ' '+b;
```

Note: Variable `c` has already been declared as a `String` in the earlier practice. You have just reassigned variable `c` to point to a new `String` reference (which contains the result of concatenation).

- b. Find the position (index) of the second letter '`T`' within the `c` `String` object.

```
c.indexOf('T', 1);
```

Notes

- The first occurrence of letter '`T`' is on position 0, so you need to start looking from the next position onward.
- Exactly the same result could have been achieved using `c.lastIndexOf('T');`. However, this works because there are only two letters '`T`' in this string, so the second one is also coincidentally the last.

- d. Find the last character in the string `c`:

```
c.charAt(c.length()-1);
```

Note: The length of the string "Tea Tea" is 7, but the last valid index position is 6.

- e. Convert String `c` to uppercase:

Hint: Remember that String objects are immutable. This means that any attempt to modify the String would simply produce a new String object. You may reassign a variable, so it would reference this new String object.

```
c = c.toUpperCase();
```

- f. Extract portion of a text from String `c`, starting from the last occurrence of letter 'T' plus one more character (the result should be 2 characters in length).

```
c.substring(c.lastIndexOf('T'), c.lastIndexOf('T')+2);
```

Note: The last index of letter 'T' is a lower boundary of the substring. Add 2 to this index to calculate the upper boundary index value. This is because the upper boundary is not inclusive of the result.

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

```
jshell> c = teaTxt+' '+b;
c ==> "Tea Tea"

jshell> c.indexOf('T',1);
$11 ==> 4

jshell> c.charAt(c.length()-1);
$12 ==> 'a'

jshell> c = c.toUpperCase();
c ==> "TEA TEA"

jshell> c.substring(c.lastIndexOf('T'), c.lastIndexOf('T')+2);
$14 ==> "TE"
```

4. Use `StringBuilder` to manipulate `txt`.

- a. Create new object called `txt` type of `StringBuilder` object and initialize it with text referenced by the variable `c`.

```
StringBuilder txt = new StringBuilder(c);
```

Note: Unlike Strings, `StringBuilder` objects must be created using new operator. Constructor of `StringBuilder` accepts `String` as an initial value.

- b. Find out the length and capacity of the `txt` object.

```
txt.length();
txt.capacity();
```

Note: Length represents an amount of characters stored in the `StringBuilder` and capacity is the size of internal storage within the `StringBuilder` object.

- c. Replace space first word "TEA" in the `txt` object with the text:

```
"What is the price of"
```

Hints

- Remember that `StringBuilder` objects are mutable, so you can modify text stored within them.
- Use the `replace` method that expects start and end positions as well as the replacement String to be supplied as parameters.

```
txt.replace(0,3,"What is the price of");
```

Note: Start position is 0, end position is 3 (number of characters in the word TEA).

Observe how replacement text has "pushed" the rest of this text value forward.

- d. Check the length and capacity of the `txt` object again.

```
txt.length();
txt.capacity();
```

Note: Internal storage within the `StringBuilder` has expanded once it has grown over the existing capacity.

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

```
jshell> StringBuilder txt = new StringBuilder(c);
txt ==> TEA TEA

jshell> txt.length();
$16 ==> 7

jshell> txt.capacity();
$17 ==> 23

jshell> txt.replace(0,3,"What is the price of");
$18 ==> What is the price of TEA

jshell> txt.length();
$19 ==> 24

jshell> txt.capacity();
$20 ==> 48
```

Practice 3-2: Use BigDecimal Class and Format Numeric Values

Overview

In this practice, you compare the use of primitives and objects to perform mathematical operations and format numeric values as text.

1. Compare mathematical operations performed on primitive and BigDecimal values.

- a. Using JShell, execute and observe behaviors of the following snippet of code:

```
double price = 1.85;
double rate = 0.065;
price -= price*rate;
price = Math.round(price*100)/100.0;
```

Notes

- Code above is designed to recalculate price by subtracting the discount, which is calculated based on 6.5% discount rate.
- Notice a need for the price double value rounding after the calculation.
- Your next task would be to perform same calculation using `java.math.BigDecimal` class instead of double primitives.

- b. Add following imports statements:

```
import java.math.BigDecimal;
import java.math.RoundingMode;
```

- c. Declare `BigDecimal` variables `price` and `rate`. Set their values to `1.85` and `0.065`, respectively:

```
BigDecimal price = BigDecimal.valueOf(1.85);
BigDecimal rate = BigDecimal.valueOf(0.065);
```

- d. Recalculate a value of `price` using the same formula as was used in set 1.a:

Hints

- Use methods of `BigDecimal` class for multiplication and subtraction.
- Instruct `BigDecimal` to round `price` with two fraction digit precision, using "half up" rounding mode.

```
price =
price.subtract(price.multiply(rate)).setScale(2,RoundingMode.HALF_UP);
```

Note: Notice the absence of double primitive rounding side effects.

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

```
jshell> double price = 1.85;
price ==> 1.85

jshell> double rate = 0.065;
rate ==> 0.065

jshell> price -= price*rate;
$32 ==> 1.7297500000000001

jshell> price = Math.round(price*100)/100.0;
price ==> 1.73

jshell> import java.math.BigDecimal;

jshell> import java.math.RoundingMode;

jshell> BigDecimal price = BigDecimal.valueOf(1.85);
price ==> 1.85

jshell> BigDecimal rate = BigDecimal.valueOf(0.065);
rate ==> 0.065

jshell> price = price.subtract(price.multiply(rate)).setScale(2,RoundingMode.HALF_UP);
price ==> 1.73
```

2. Format values for price and rate as text that represents currency and percentage.

- a. Add the following import statements:

```
import java.util.Locale;
import java.text.NumberFormat;
```

- b. Declare variable called `locale` type of `Locale` class. Initialize this variable to reference France locale object:

```
Locale locale = Locale.FRANCE;
```

Note: Alternatively, the same locale object can be initialized using:

```
new Locale("fr", "FR"); or Locale.forLanguageTag("fr-FR");
```

- c. Declare two variables type of `NumberFormat` called `currencyFormat` and `percentFormat`. Initialize these variables to reference currency and percentage format objects, using the `locale` object that you have just created:

```
NumberFormat currencyFormat =
NumberFormat.getCurrencyInstance(locale);
NumberFormat percentFormat =
NumberFormat.getPercentInstance(locale);
```

- d. Set `percentFormat` object to use two maximum fraction digits:

```
percentFormat.setMaximumFractionDigits(2);
```

Note: Default percent format object is formatting percent values discarding the floating point part.

- e. Format values of price and rate using currencyFormat and percentFormat objects:

```
currencyFormat.format(price);
percentFormat.format(rate);
```

Note: Observe France locale format rules using comma as decimal separator and Euro as the currency. Also notice that currency symbol is displayed after the value.

- f. Reassign locale variable to reference British English locale object:

Hint: You may reselect previous commands in JShell using arrow up key. Once required command is selected it can be modified and executed again.

```
locale = Locale.UK;
```

Note: Alternatively, same locale object can be initialized using:

```
new Locale("en", "GB"); or Locale.forLanguageTag("en-GB");
```

- g. Reinitialize both currency and percent format objects to apply new locale:

Hint: You have to re-create currency and percent format objects, to set them up to use another locale. This would dereference old format object. New percent format object has to be configured to use two fraction digits just like the previous one has.

```
currencyFormat = NumberFormat.getCurrencyInstance(locale);
percentFormat = NumberFormat.getPercentInstance(locale);
percentFormat.setMaximumFractionDigits(2);
```

- h. Declare two String variables called priceTxt, rateTxt. Repeat formatting of price and rate variables using currencyFormat and percentFormat objects and assign formatted result to these two newly declared variables :

```
String priceTxt = currencyFormat.format(price);
String rateTxt = percentFormat.format(rate);
```

Notes

- Observe format changes reflecting British English locale rules, using dot as decimal separator and pound as the currency. Also notice that currency symbol is displayed in front of the value.
- You will use the variables priceTxt and rateTxt in a later stage of the practice.

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

```
jshell> import java.util.Locale;
jshell> import java.text.NumberFormat;
jshell> Locale locale = Locale.FRANCE;
locale ==> fr_FR

jshell> NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
currencyFormat ==> java.text.DecimalFormat@674dc

jshell> NumberFormat percentFormat = NumberFormat.getPercentInstance(locale);
percentFormat ==> java.text.DecimalFormat@674dc

jshell> percentFormat.setMaximumFractionDigits(2);

jshell> currencyFormat.format(price);
$45 ==> "1,73 €"

jshell> percentFormat.format(rate);
$46 ==> "6,5 %"

jshell> locale = Locale.UK;
locale ==> en_GB

jshell> currencyFormat = NumberFormat.getCurrencyInstance(locale);
currencyFormat ==> java.text.DecimalFormat@6757f

jshell> percentFormat = NumberFormat.getPercentInstance(locale);
percentFormat ==> java.text.DecimalFormat@674dc

jshell> percentFormat.setMaximumFractionDigits(2);

jshell> String priceTxt = currencyFormat.format(price);
priceTxt ==> "£1.73"

jshell> String rateTxt = percentFormat.format(rate);
rateTxt ==> "6.5%"
```

Practice 3-3: Use and Format Date and Time Values

Overview

In this practice, you use local as well as zoned date and time objects and format date and time values as text.

1. Explore local date and time objects.

a. Add the following imports statements:

```
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.LocalDateTime;
import java.time.Duration;
```

b. Create a new variable called `today` of type `LocalDate` and initialize it to reference current date object:

```
LocalDate today = LocalDate.now();
```

c. Calculate which day of the week this date would be next year:

Hints

- You need to create a new `LocalDate` object by adding one to the value of year of the `today` object.

- After that, retrieve the value of the day of the week for this date.

```
today.plusYears(1).getDayOfWeek();
```

d. Create a new variable called `teaTime` of type `LocalTime` and initialize it to reference local time object that represents half past five in the afternoon:

```
LocalTime teaTime = LocalTime.of(17, 30);
```

e. Create a variable called `timeGap` of type `Duration`. Set this variable to a value of the time period between current point in time and the tea time.

Hint: Class `Duration` can be used to calculate distance between points in time. For calculating distance between dates, you could use class `Period`.

```
Duration timeGap = Duration.between(LocalTime.now(), teaTime);
```

f. Express the value of `timeGap` duration amount in minutes and in hours plus minutes:

Hint: Use the `toMinutes`, `toHours`, and `toMinutesPart` methods to express the duration amount in different ways:

```
timeGap.toMinutes();
timeGap.toHours();
timeGap.toMinutesPart();
```

Notes

- `Duration` can be positive or negative depending on the current time in relation to the tea time value.
- `toXXX` methods extract duration amount in different temporal units. Notice that the `toMinutes` method returns total amount of minutes between specific points in time, whereas `toMinutesPart` returns remaining amount of minutes past the nearest hour.

- g. Create a variable called `tomorrowTeaTime` of type `LocalDateTime`. Set this variable to a value of the combined value of the `today LocalDate` object, adjusted to represent the next date and the `teaTime LocalTime` object.

```
LocalDateTime tomorrowTeaTime =
LocalDateTime.of(today.plusDays(1), teaTime);
```

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

```
jshell> import java.time.LocalDate;
jshell> import java.time.LocalTime;
jshell> import java.time.LocalDateTime;
jshell> import java.time.Duration;
jshell> LocalDate today = LocalDate.now();
today ==> 2019-10-22
jshell> today.plusYears(1).getDayOfWeek();
$58 ==> THURSDAY
jshell> LocalTime teaTime = LocalTime.of(17,30);
teaTime ==> 17:30
jshell> Duration timeGap = Duration.between(LocalTime.now(), teaTime);
timeGap ==> PT-6H-17M-5.02316S
jshell> timeGap.toMinutes();
$61 ==> -377
jshell> timeGap.toHours();
$62 ==> -6
jshell> timeGap.toMinutesPart();
$63 ==> -17
jshell> LocalDateTime tomorrowTeaTime = LocalDateTime.of(today.plusDays(1), teaTime);
tomorrowTeaTime ==> 2019-10-23T17:30
```

2. Apply time zones to local date and time objects.

- a. Add the following import statements:

```
import java.time.ZoneId;
import java.time.ZonedDateTime;
```

- b. Declare two variables called london and katmandu of type ZoneId. Initialize these variables to reference the corresponding time zones:

```
ZoneId london = ZoneId.of("Europe/London");
ZoneId katmandu = ZoneId.of("Asia/Katmandu");
```

- c. Declare a variable called londonTime of type ZonedDateTime. Assign a result of conversion of tomorrowTeaTime object to London time zone to the londonTime variable:

```
ZonedDateTime londonTime =
ZonedDateTime.of(tomorrowTeaTime, london);
```

- d. Declare a variable called katmanduTime of type ZonedDateTime. Assign a result of conversion of londonTime object to Katmandu time zone to the katmanduTime variable:

```
ZonedDateTime katmanduTime =
londonTime.withZoneSameInstant(katmandu);
```

Note: Observe the difference in time between these two time zones. To find out the exact offset value between time in Katmandu and UTC/Greenwich time, use the katmanduTime.getOffset(); operation.

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

```
jshell> import java.time.ZoneId;
jshell> import java.time.ZonedDateTime;
jshell> ZoneId london = ZoneId.of("Europe/London");
london ==> Europe/London
jshell> ZoneId katmandu = ZoneId.of("Asia/Katmandu");
katmandu ==> Asia/Katmandu
jshell> ZonedDateTime londonTime = ZonedDateTime.of(tomorrowTeaTime, london);
londonTime ==> 2019-10-23T17:30+01:00[Europe/London]
jshell> ZonedDateTime katmanduTime = londonTime.withZoneSameInstant(katmandu);
katmanduTime ==> 2019-10-23T22:15+05:45[Asia/Katmandu]
jshell> katmanduTime.getOffset();
$71 ==> +05:45
```

4. Format date and time values.

a. Add the following import statement:

```
import java.time.format.DateTimeFormatter;
```

b. Declare a String variable called `datePattern` that describes the following format:

```
<Weekday>', '<Day>' of '<Month> <Year>' at '<Hours>:<Minutes> <Zone>'
```

Hints

- <Weekday> should be a name of the day of the week expressed as an abbreviation.
- <Day> should be the day of the month expressed without the leading zero.
- <Month> should be the full name of the month.
- <Year> should be expressed in 4 digits .
- <Hours> should represent the hour of the day between 0 and 23.
- <Minutes> should represent minutes within the hour.
- <Zone> should represent the time-zone name.
- Single quote symbol is an escape character within the pattern.

```
String datePattern = "EE', 'd' of 'MMMM yyyy' at 'HH:mm z";
```

c. Declare a new variable type of `DateTimeFormatter` called `dateFormat`. Initialize this variable to reference Date and Time format object, using the pattern defined in the previous step and a locale object that you have created in the earlier practice:

Hint: Your locale variable should be referencing British English locale object. You can always reset it with the flowing statement: `locale = Locale.UK;`

```
DateTimeFormatter dateFormat =
DateTimeFormatter.ofPattern(datePattern, locale);
```

d. Declare a String variable called `timeTxt`. Assign the result of formatting of the `katmanduTime` object to this variable:

```
String timeTxt = dateFormat.format(katmanduTime);
```

Note: You will use variable `timeTxt` in a later stage of the practice.

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

```
jshell> import java.time.format.DateTimeFormatter;
jshell> String datePattern = "EE', 'd' of 'MMMM yyyy' at 'HH:mm z";
datePattern ==> "EE', 'd' of 'MMMM yyyy' at 'HH:mm z"

jshell> DateTimeFormatter dateFormat = DateTimeFormatter.ofPattern(datePattern, locale);
dateFormat ==> Text(DayOfWeek,SHORT)', 'Value(DayOfMonth)' of 'T ... fHour,2)' 'ZoneText(SHORT)

jshell> String timeTxt = dateFormat.format(katmanduTime);
timeTxt ==> "Wed, 23 of October 2019 at 22:15 NPT"
```

Practice 3-4: Apply Localization and Format Messages

Overview

In this practice, you format messages using patterns, retrieved from the resource bundles.

1. Load resource bundle and format messages.

- a. Add the following imports statements:

```
import java.util.ResourceBundle;
import java.text.MessageFormat;
```

- b. Create a new variable called `msg` type of `ResourceBundle` and initialize it to reference resource bundle called `messages` using your existing `locale` variable:

```
ResourceBundle msg = ResourceBundle.getBundle("messages", locale);
```

Note: The resource bundle file (`messages.properties`) is located in the Practices/Lesson 3 folder and contains the following key-value pairs:

```
offer={0}, price: {1} (applied {2} discount), valid until {3}
dateFormat = EE', 'd' of 'MMMM yyyy' at 'HH:mm z
```

- c. Create a new variable called `offerPattern` of type `String` and initialize it to reference text that corresponds to the key "offer" fetched from the resource bundle.

```
String offerPattern = msg.getString("offer");
```

- d. Format the text message based on the offer pattern and substitute values of variables `teaTxt`, `priceTxt`, `rateTxt` and `timeTxt` (prepared in earlier steps of the practice):

Hint: Use the `MessageFormat` class to perform value substitutions.

```
MessageFormat.format(offerPattern, teaTxt, priceTxt, rateTxt,
timeTxt);
```

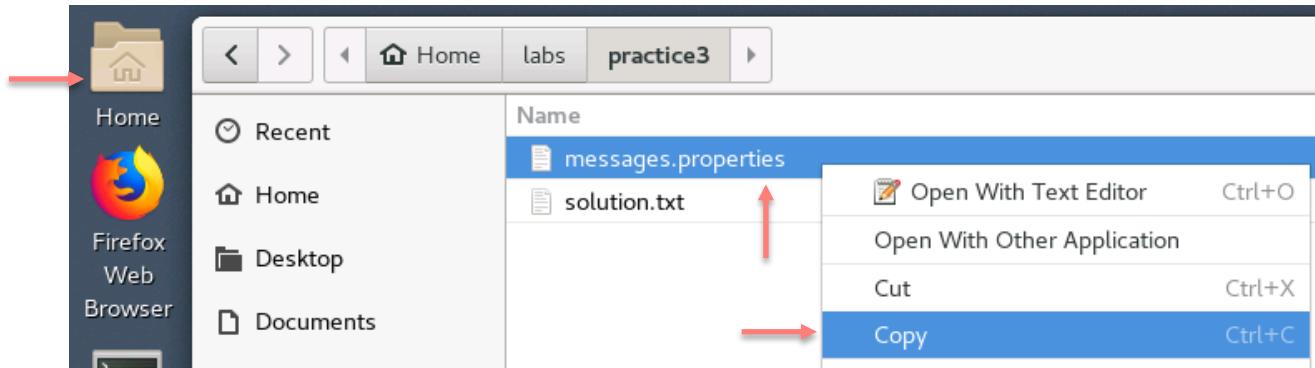
Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

```
jshell> import java.util.ResourceBundle;
jshell> import java.text.MessageFormat;
jshell> ResourceBundle msg = ResourceBundle.getBundle("messages", locale);
msg ==> java.util.PropertyResourceBundle@43bd930a
jshell> String offerPattern = msg.getString("offer");
offerPattern ==> "{0}, price: {1} (applied {2} discount), valid until {3}"
jshell> MessageFormat.format(offerPattern, teaTxt, priceTxt, rateTxt, timeTxt);
$80 ==> "Tea, price: £1.73 (applied 6.5% discount), valid until Wed, 23 of October 2019 at 22:15 NPT"
```

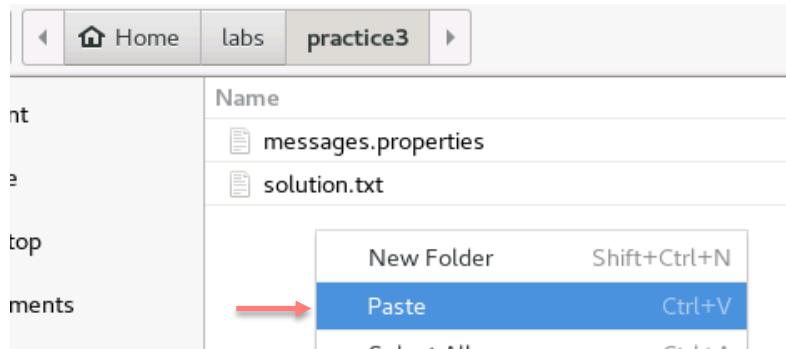
2. Create and use the translated version of the resource bundle.
 - a. Create a copy of the `message.properties` file located in the `/home/oracle/labs.practice3` folder. Rename this copy to match any language and country of your choice. This practice describes the creation of Russian language version of the properties file as an example.
`messages_ru_RU.properties`

Hints:

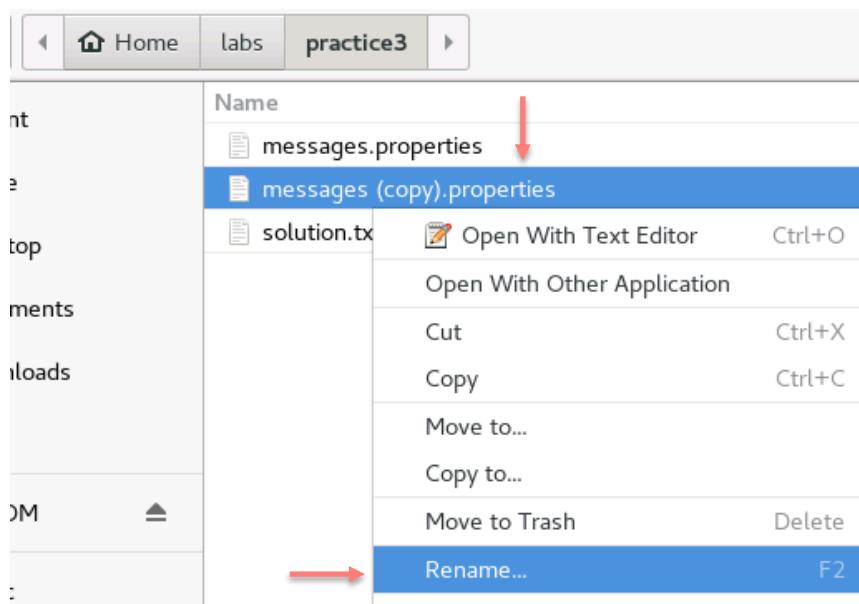
- Navigate to the `/home/oracle/labs/practice3` folder.



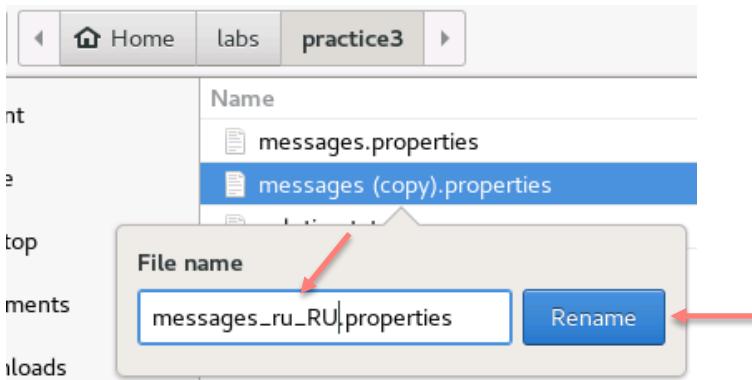
- Right-click anywhere inside the Practice 3 folder and select "Paste."



- Right-click the “messages (copy).properties” file and invoke the “Rename” menu.



- Modify the file name to match the language and country of your choice. (This example shows Russian.)



- Click the "Rename" button.
- Replace text values in this new resource file with the translated text. Use the language of your choice, so long as that matches the bundle file name.

Hints:

- Double-click the properties file you have just created to open it in Text Editor.
- Replace values for the `offer` and `dateFormat` keys.
- The Russian version is provided as an example.

`offer={0}, цена: {1} (с учетом {2} скидки), действительно до {3}`

`dateFormat = EE', 'd MMMM yyyy' в 'HH:mm z'`

- Click the "Save" button to save this file.

- e. Reassign the `locale` variable to reference the `Locale` object that matches required country and language to match the new resource bundle.

```
locale = new Locale("ru", "RU");
```

- f. Reload the resource bundle for the new locale:

Hint: Use the same bundle base name "messages" - the relevant version of the bundle will be selected for you based on a locale specified.

```
msg = ResourceBundle.getBundle("messages", locale);
```

- g. Reassign `datePattern` and `offerPattern` variables, getting their values from the bundle, using "offer" and "dateFormat" keys.

```
offerPattern = msg.getString("offer");
```

```
datePattern = msg.getString("dateFormat");
```

- h. Reassign `currencyFormat`, `percentFormat`, and `dateFormat` variable to use new locale:

```
currencyFormat = NumberFormat.getCurrencyInstance(locale);
```

```
percentFormat = NumberFormat.getPercentInstance(locale);
```

```
percentFormat.setMaximumFractionDigits(2);
```

```
dateFormat = DateTimeFormatter.ofPattern(datePattern, locale);
```

Note: You may wish to use default date/time format for the specific locale.

The following code is not a part of the practice, but a demonstration of an alternative way of formatting date and time values, without using custom format pattern:

```
import java.time.format.FormatStyle;
dateFormat =
DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM,
FormatStyle.MEDIUM).localizedBy(locale);
(Format styles are FULL, LONG, MEDIUM, SHORT)
```

- i. Reassign `teaTxt`, `priceTxt`, `rateTxt`, and `timeTxt` variables.

Hints

- Use the language of your choice to set the value of the `teaTxt` variable.
- Repeat invocations of format methods to format other variables.

```
teaTxt = "Чай";
priceTxt = currencyFormat.format(price);
rateTxt = percentFormat.format(rate);
timeTxt = dateFormat.format(katmanduTime);
```

- j. Repeat formatting of the text message based on the offer pattern and substitute values of variables `teaTxt`, `priceTxt`, `rateTxt` and `timeTxt`:

```
MessageFormat.format(offerPattern, teaTxt, priceTxt, rateTxt, timeTxt);
```

- k. This is the end of the practice. You can now close JShell:

```
/exit
```

- l. You can also close Text Editor.

Note: The following screenshot (provided for verification purposes) shows all JShell actions required by this practice segment:

```
jshell> locale = new Locale("ru","RU");
locale ==> ru_RU

jshell> msg = ResourceBundle.getBundle("messages",locale);
msg ==> java.util.PropertyResourceBundle@8e24743

jshell> offerPattern = msg.getString("offer");
offerPattern ==> "{0}, цена: {1} (с учетом {2} скидки), действительно до {3}"

jshell> datePattern = msg.getString("dateFormat");
datePattern ==> "EE', 'd MMMM yyyy' в 'HH:mm z"

jshell> currencyFormat = NumberFormat.getCurrencyInstance(locale);
currencyFormat ==> java.text.DecimalFormat@674dc

jshell> percentFormat = NumberFormat.getPercentInstance(locale);
percentFormat ==> java.text.DecimalFormat@674dc

jshell> percentFormat.setMaximumFractionDigits(2);

jshell> dateFormat = DateTimeFormatter.ofPattern(datePattern, locale);
dateFormat ==> Text(DayOfWeek,SHORT)', 'Value(DayOfMonth)' 'Text ... fHour,2)' 'ZoneText(SHORT)

jshell> teaTxt = "Чай";
teaTxt ==> "Чай"

jshell> priceTxt = currencyFormat.format(price);
priceTxt ==> "1,73 ₽"

jshell> rateTxt = percentFormat.format(rate);
rateTxt ==> "6,5 %"

jshell> timeTxt = dateFormat.format(katmanduTime);
timeTxt ==> "чт, 24 октября 2019 в 22:15 NPT"

jshell> MessageFormat.format(offerPattern, teaTxt, priceTxt, rateTxt, timeTxt);
$190 ==> "Чай, цена: 1,73 ₽ (с учетом 6,5 % скидки), действительно до чт, 24 октября 2019 в 22:15 NPT"

jshell> /exit
| Goodbye
[oracle@edvmr1p0 practice3]$ █
```

Practices for Lesson 4: Classes and Objects

Practices for Lesson 4: Overview

Overview

In these practices, you use NetBeans to create a project to contain classes for the Product Management application. You create Classes Product and Shop within this new project, document your code, and test your code by compiling and executing your application.



Practice 4-1: Create the Product Management Application

Overview

In this practice, you create a new project in NetBeans and use this project to create classes to implement the Product Management application.

Assumptions

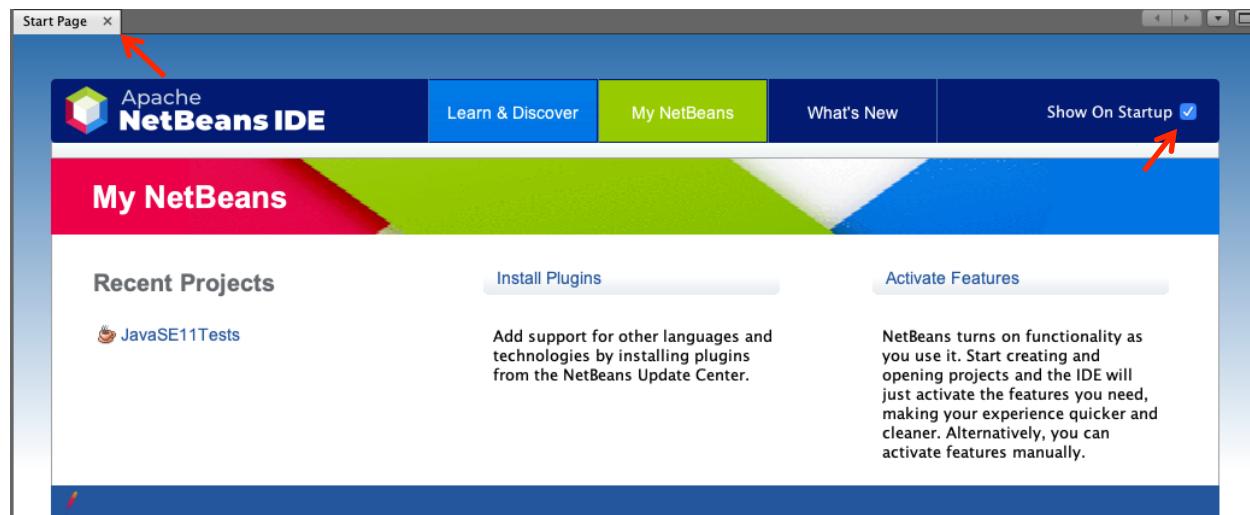
- JDK 11 is installed.
- NetBeans 11 is installed.
- The practices folder structure is created.

Tasks

1. Prepare the practice environment.
 - a. Open NetBeans. You can use the desktop shortcuts provided.

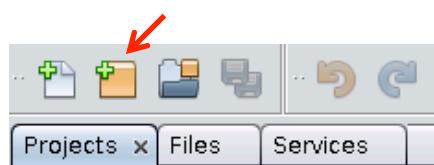


- b. Deselect the "Show On Startup" check box and close the "Start Page."

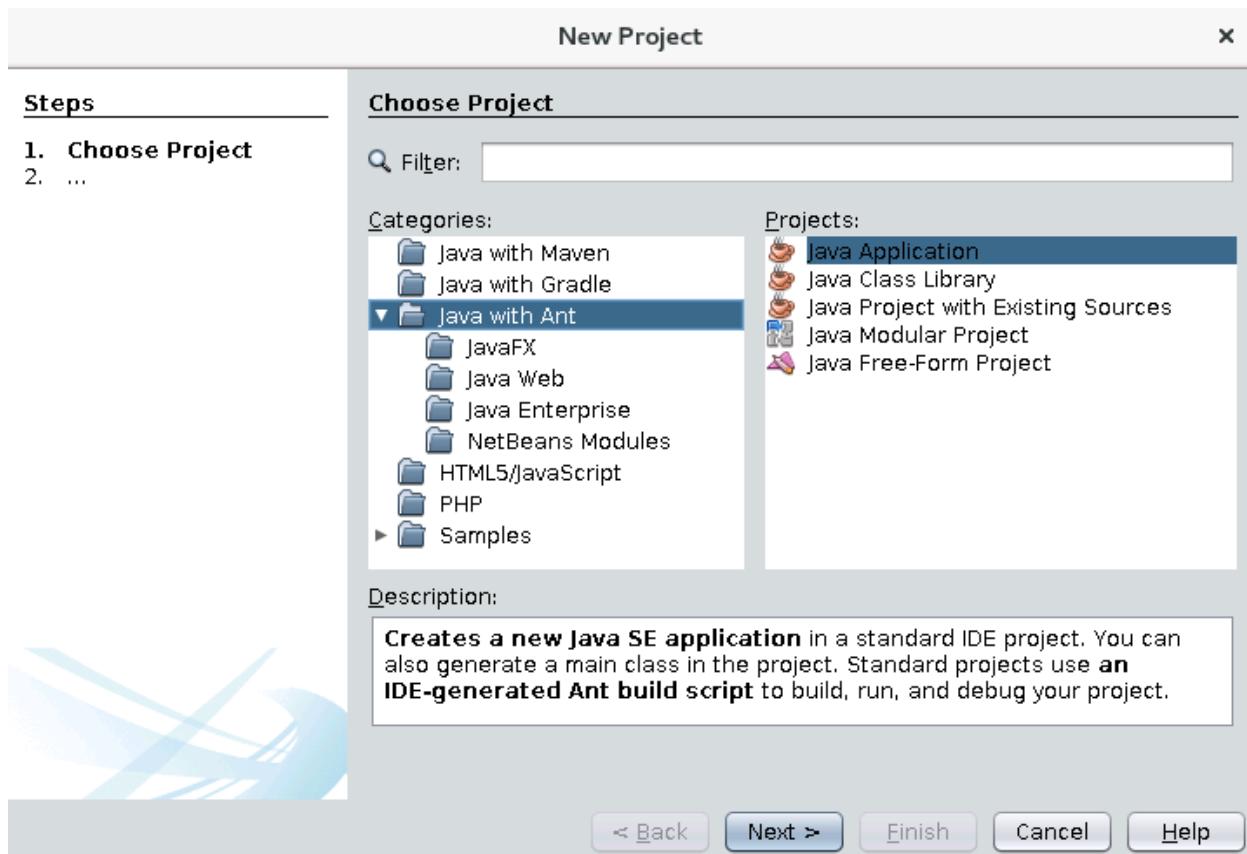


- c. Create a new project.

Hint: Use File > New Project menu or click the "New Project" toolbar button.

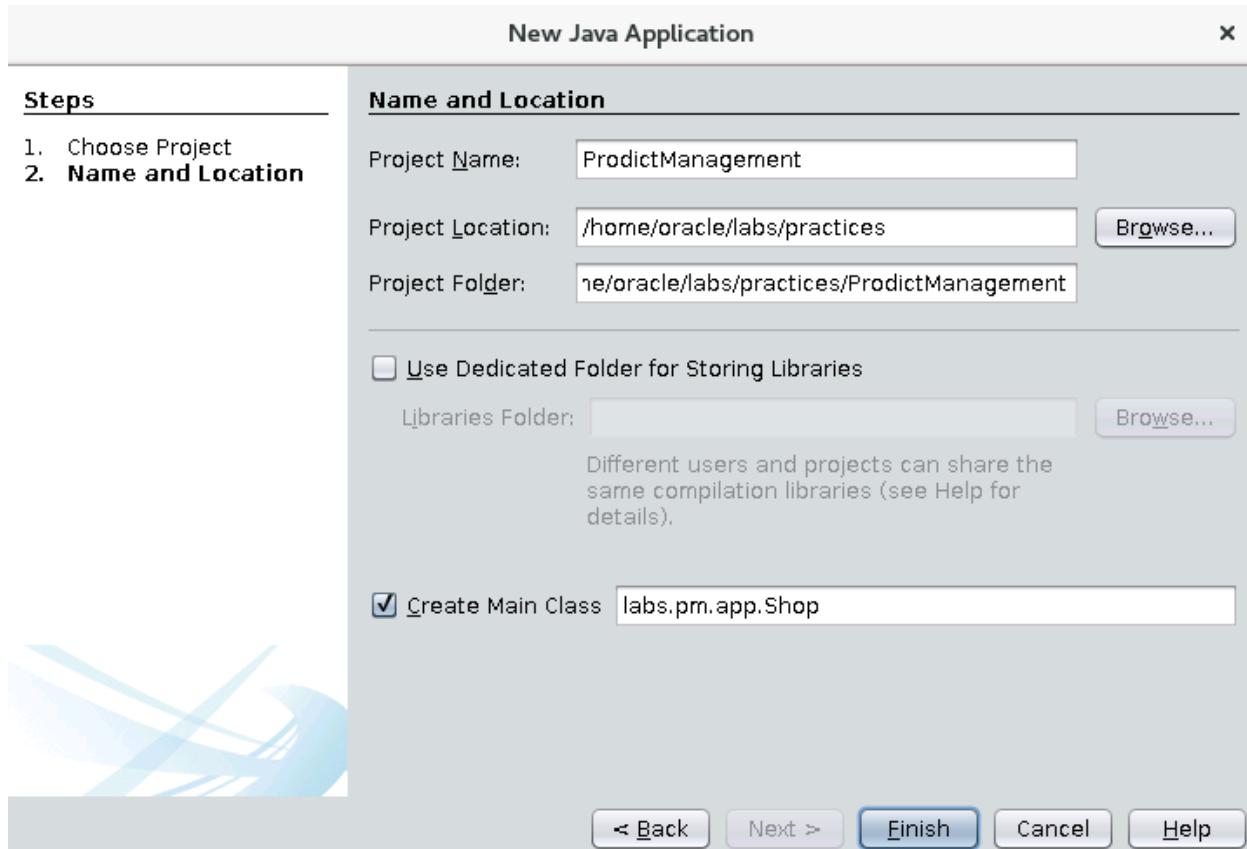


- d. Select the "Java with Ant" Category and "Java Application" as the project type.



- e. Click "Next."

- g. Set the following project properties:
- Project Name: ProductManagement
 - Project Location: labs/practices
 - Project Folder: /home/oracle/labs/practices/ProductManagement
 - Select the Create Main Class check box.
 - Use labs.pm.app.Shop as a main class package and class name.



- Note:** The purpose of the Shop class is to represent the entry point into your application.
- h. Click "Finish."

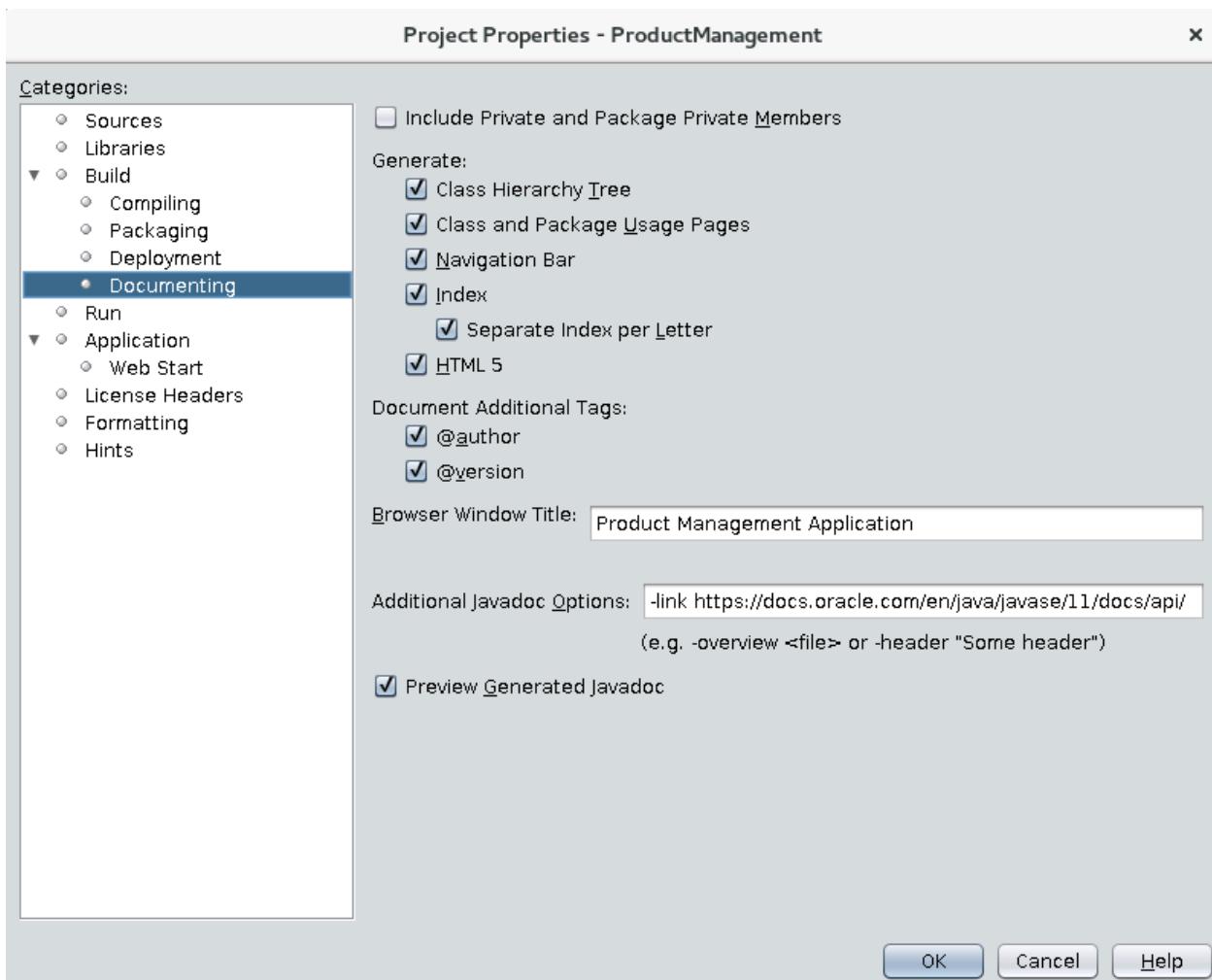
2. Set up the code documentation for the Product Management project.
 - a. Right-click a ProductManagement project and invoke the "Properties" menu.



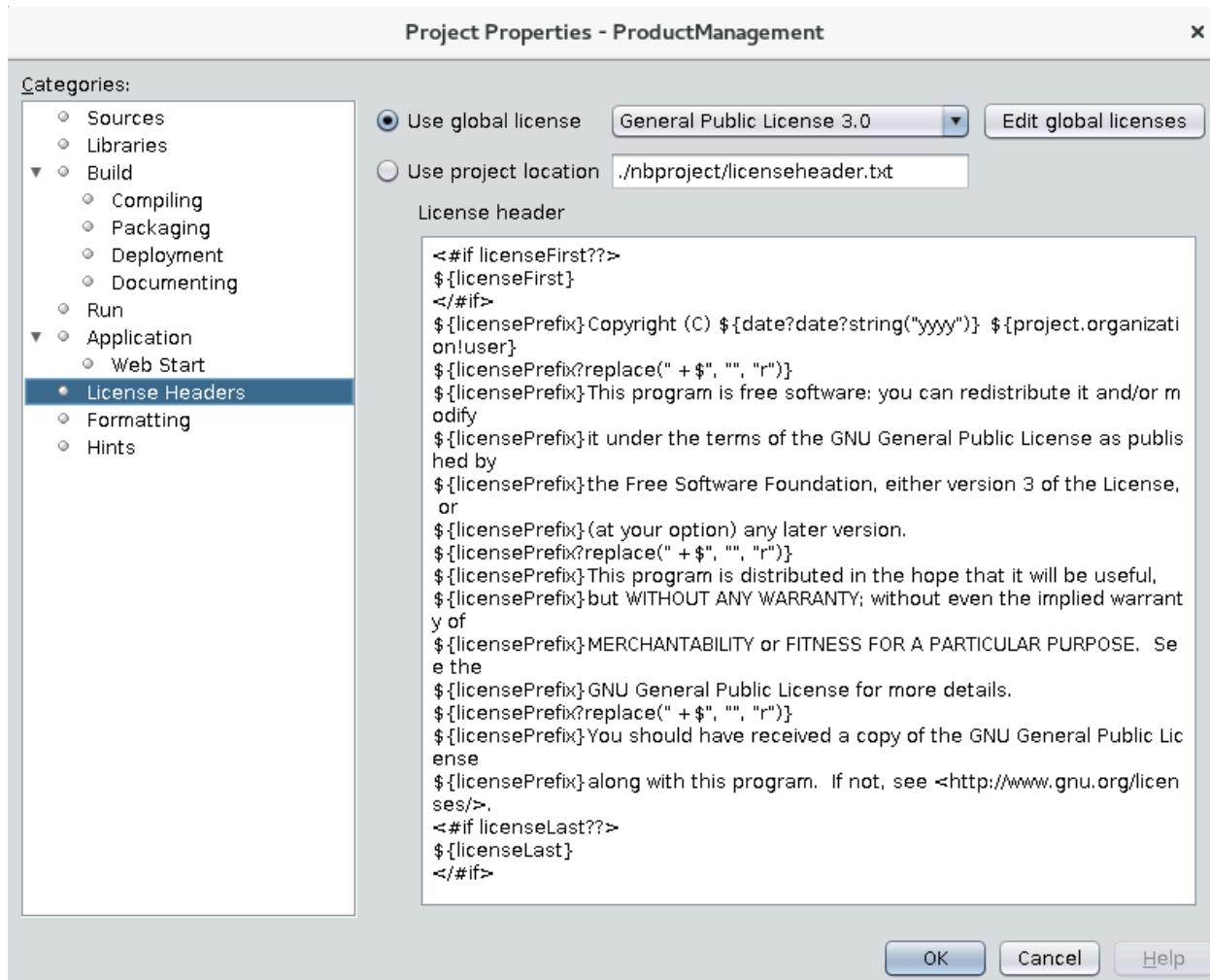
- b. Navigate to the Build->Documenting section and set the following documentation properties:

- Select HTML5, @author, and @version check boxes.
- Set Browser Window Title: Product Management Application
- Additional Javadoc Options:
 -link https://docs.oracle.com/en/java/javase/11/docs/api/
 -J-Dhttps.proxyHost=ges-proxy.us.oracle.com
 -J-Dhttps.proxyPort=80

Note: Proxy parameters in additional Javadoc options are required for this course environment.



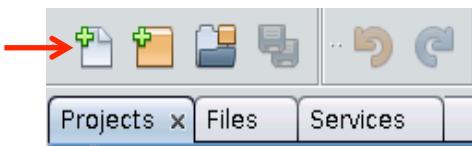
- Navigate to the Application->Licence Headers section.
- Select General Public Licence 3.0 from the drop-down list of global licenses.



- Click "OK."

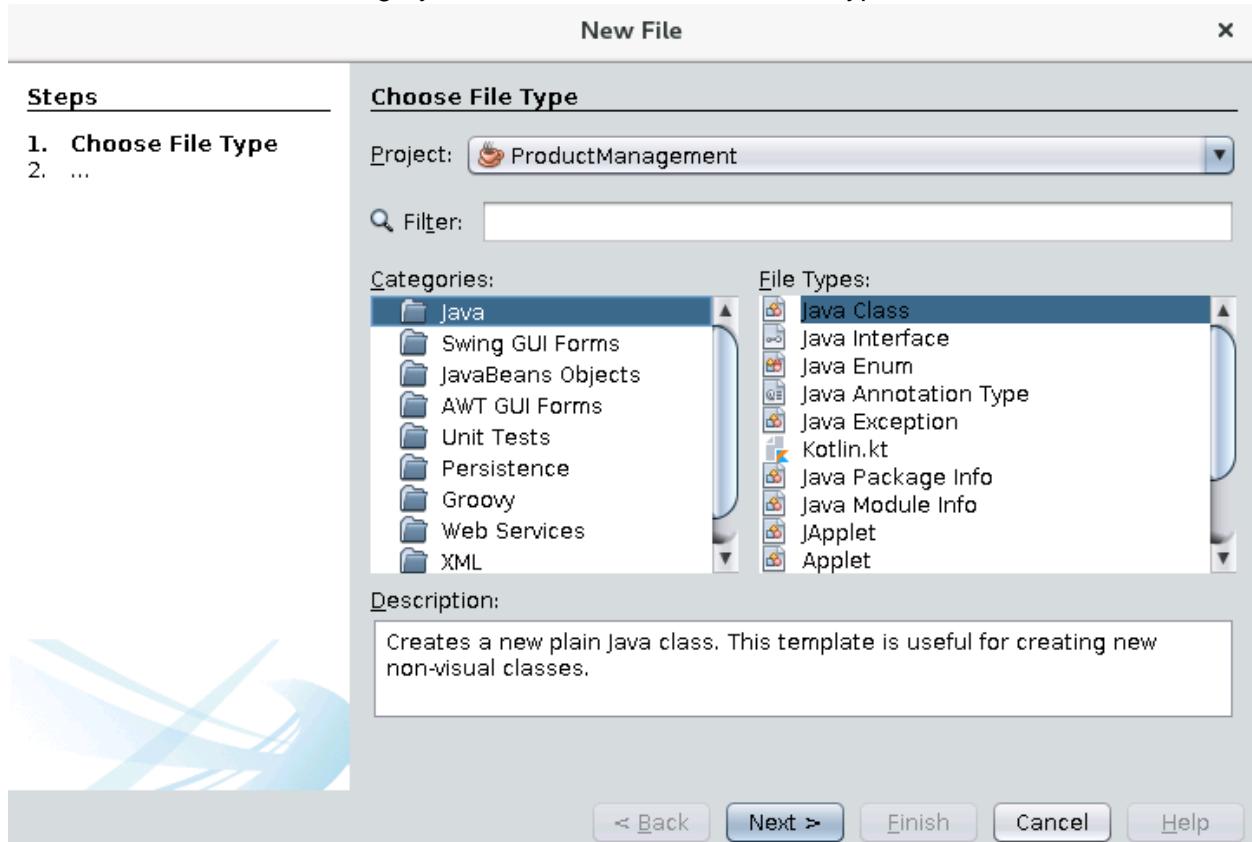
Note: The new classes created in the context of this Project would now be documented with the documentation settings that you have specified. However, existing classes (Shop) are not affected by these modifications of project settings.

3. Create the Product class to represent data and behaviors of product objects for the Product Management application.
 - a. Create new Java class.



Hint: Use the File -> New File menu or click the "New File" toolbar button.

- b. Select "Java" Category and "Java Class" as the file type.



- c. Click "Next."

d. Set the following class properties:

- **Class Name:** Product
- **Project:** ProductManagement
- **Location:** Source Packages
- **Package:** labs.pm.data



e. Click "Finish."

f. Add instance variables to Product class to represent the following information:

- **id** type of int
- **name** type of String
- **price** type of BigDecimal

Hint: Make all variables private to ensure proper encapsulation.

```
private int id;
private String name;
private BigDecimal price;
```

- g. Add an import statement for the `java.math.BigDecimal` class.

Hint: Click the left side of the line of code on a light bulb to invoke the "Add import" popup menu and select the "Add import for the `java.math.BigDecimal`" option.

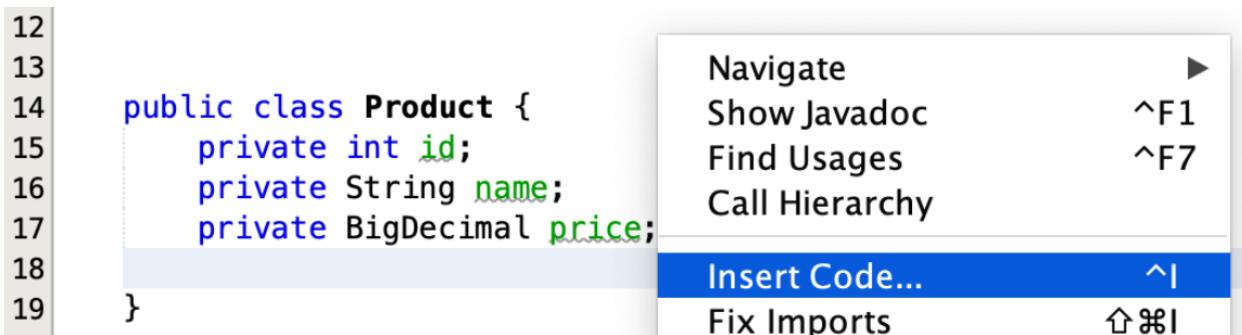
```
12 public class Product {  
13     private int id;  
14     private String name;  
15     private BigDecimal price;  
16      Add import for java.math.BigDecimal  
17      Create class "BigDecimal" in package labs.pm.data (Source Packages)  
18      Create class "BigDecimal" in labs.pm.data.Product
```

```
package labs.pm.data;  
import java.math.BigDecimal;  
public class Product {  
    private int id;  
    private String name;  
    private BigDecimal price;  
}
```

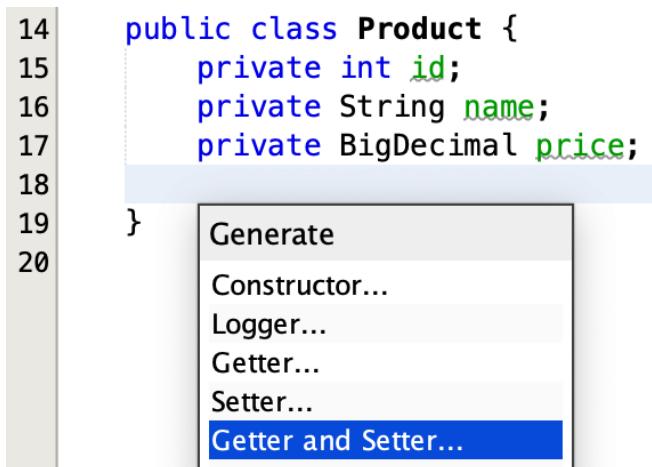
- h. Add accessor (getter and setter) methods for each instance variable of the Product.

Hints: Right-click on an empty line inside the Product class and select the "Insert Code..." menu.

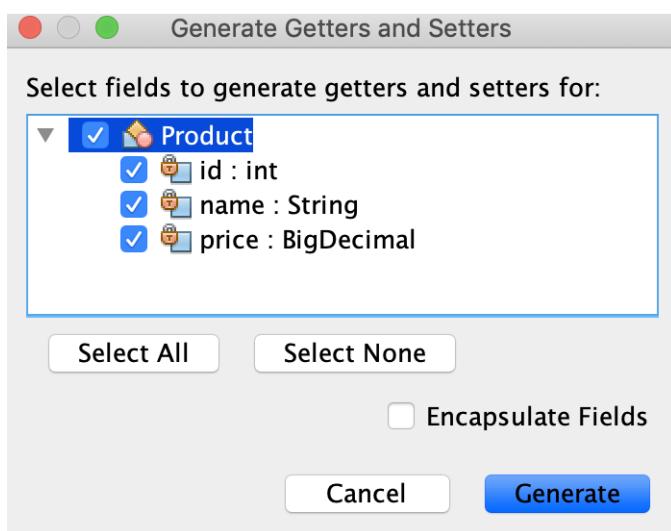
- Then select "Getter and Setter..." menu.



- Select the check box next to Product (which selects all of its attributes).



- Click the "Generate" button.



Note: There is no need to select the "Encapsulate Fields" check box if they are all already declared as private.

```
public class Product {
    private int id;
    private String name;
    private BigDecimal price;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public BigDecimal getPrice() {
        return price;
    }
    public void setPrice(BigDecimal price) {
        this.price = price;
    }
}
```

Note: NetBeans has generated parameter names for the setter methods that coincide with the instance variable names, thus "shadowing" these variables. This is why instance variables are prefixed with the "this" keyword inside these methods.

4. Add code to the main method of the Shop class to create Product instances, set values for their properties, retrieve these values back, and print them to the console.
 - a. Switch to the Shop class editor.
 - b. Inside the `main` method of the `Shop` class, add the following code:
 - Declare a variable called `p1` type of `Product`.
 - Create new instance of `Product`.
 - Initialize the `p1` variable to reference the `Product` object you have just created.

Hint: Replace "`// TODO code application logic here`" comment

```
Product p1 = new Product();
```

- c. Add an import of `labs.pm.data.Product` to the `Shop` class.

Hint: Click the right side of the line of code to invoke the "Add import" pop-up menu and select "Add import for `labs.pm.data.Product`" option.

```

17     public static void main(String[] args) {
18         Product p1 = new Product();
19         Add import for labs.pm.data.Product
20         Create class "Product" in package labs.pm.app (Source Packages)
21         Create class "Product" in labs.pm.app.Shop
22

```

```
import labs.pm.data.Product;
```

Note: The import above is added to the class `Shop` to make it more convenient to reference the `Product` class, because classes `Shop` and `Product` are located in different packages.

- d. Set ID, name, and price attributes of the `Product` instance reference via the `p1` variable. Use the following values: 101, Tea, 1.99

Hint: Use the setter methods, because all attributes of the `Product` are private and thus cannot be accessed from other classes.

```

p1.setId(101);
p1.setName("Tea");
p1.setPrice(BigDecimal.valueOf(1.99));

```

- e. Add an import statement for the `java.math.BigDecimal` class.

Hints:

- Click on the right side of the line of code to invoke the "Add import" pop-up menu and select "Add import for `java.math.BigDecimal`" option.
- Alternatively right-click anywhere inside the `Shop` class and invoke "Fix Imports" menu.

```

19     public static void main(String[] args) {
20         Product p1 = new Product();
21         p1.setId(101);
22         p1.setName("Tea");
23         p1.setPrice(BigDecimal.valueOf(1.99));
24     }
25
26 }
27

```

```
import java.math.BigDecimal;
```

- f. Print all attributes of the product object to the console as a single line of text.

Hints:

- Use the getter methods, because all attributes of the Product are private and thus cannot be accessed from other classes.
- Concatenate all attribute values using space between values.
- Print the resulting string to the console by using the System.out.println method.

```
System.out.println(p1.getId()+" "+p1.getName()+" "+p1.getPrice());
```

- g. Compile and run your application.

Hint: Click the "Run Project" toolbar button.



```

6 package labs.pm.app;
7
8 import java.math.BigDecimal;
9 import labs.pm.data.Product;
10
11 /**
12 * 
13 * @author oracle
14 */
15 public class Shop {
16
17     /**
18      * @param args the command line arguments
19     */
20     public static void main(String[] args) {
21         Product p1 = new Product();
22         p1.setId(101);
23         p1.setName("Tea");
24         p1.setPrice(BigDecimal.valueOf(1.99));
25         System.out.println(p1.getId()+" "+p1.getName()+" "+p1.getPrice());
26     }
27 }

```

The screenshot shows the Java code for the Shop class. The main() method creates a Product object p1 with ID 101, name "Tea", and price 1.99. It then prints these attributes to the console using System.out.println. The code is highlighted in blue and yellow, indicating syntax errors or warnings. The output window shows the successful compilation and execution of the code, printing "101 Tea 1.99" to the console.

Note: Observe the product details printed on to the console.

Practice 4-2: Enhance the Product Class

Overview

In this practice, you make improvements to the Product class design. Add a constant that represents discount rate and an operation that calculates discount for the Product class.

1. Ensure consistent assignment of parameters within the `Product` class.

Note: Your task in this part of the practice would be to "break" the `Product` class design and then add code to fix the problem and prevent similar accidental coding errors.

- a. Switch to the `Product` class editor, locate the `setPrice` method, and add the first line of code inside this method that assigns a value of `1` to the `price` variable.

Hint: Use public static final variable (constant) that represents a value of `1` available in the `BigDecimal` Class.

```
public void setPrice(BigDecimal price) {
    price = BigDecimal.ONE;
    this.price = price;
}
```

- b. Click the "Run Project" toolbar button.



Output - ProductManagement (run) x

```
ant -f /home/oracle/labs/practices/ProductManagement -Dnb.internal.action.name=run run
init:
Deleting: /home/oracle/labs/practices/ProductManagement/build/built-jar.properties
deps-jar:
Updating property file: /home/oracle/labs/practices/ProductManagement/build/built-jar.properties
Compiling 2 source files to /home/oracle/labs/practices/ProductManagement/build/classes
compile:
run:
101 Tea 1
BUILD SUCCESSFUL (total time: 3 seconds)
```

Note: You have just assigned another value to the variable that represents the method argument, which makes the `setPrice` method essentially ignore parameter value passed to it. This is not a recommended design practice because it can be very confusing to the invoker of this method. The `Shop` class sets product price to be `1.99`, yet it is actually reset to be `1` regardless of what the invoker actually specifies.

- c. Prevent reassignment of method arguments in the `Product` class by marking them all with the `final` keyword.

```
public void setId(final int id) { /* existing code */ }
public void setName(final String name) { /* existing code */ }
public void setPrice(final BigDecimal price) { /* existing code */ }
```

Note: The `Product` class would not compile, because its code attempts to assign new value to the `price` parameter, which is now marked as constant and thus cannot be reassigned.

- d. Remove the offending line of code from the Product class.

Hint: Delete `price = BigDecimal.ONE;` line of code inside the `setPrice` method

- e. Click the "Run Project" toolbar button again.



```
Output - ProductManagement (run) x
ant -f /home/oracle/labs/practices/ProductManagement -Dnb.internal.action.name=run run
init:
Deleting: /home/oracle/labs/practices/ProductManagement/build/built-jar.properties
deps-jar:
Updating property file: /home/oracle/labs/practices/ProductManagement/build/built-jar.properties
Compiling 1 source file to /home/oracle/labs/practices/ProductManagement/build/classes
compile:
run:
101 Tea 1.99
BUILD SUCCESSFUL (total time: 3 seconds)
```

Note: Now your code is compiling and parameter value reassignment is prevented by the Product class design.

2. Add logic to the Product class that calculates discount value.

- a. Inside the Product class, add a new constant to represent a `BigDecimal` discount rate of 10% and whose value is to be shared by all instances of the Product.

Hints:

- Make a new line just in front of existing variable declarations in the Product class to place the discount rate constant declaration.
- Make this variable `public, static, and final`.
- Use `BigDecimal` as the variable type.
- Use `DISCOUNT_RATE` as a variable name. (According to the Java naming convention, public static final constant names should be in uppercase, with underscores between words.)
- Initialize it to the value of `0.1`.
- No encapsulation is required in this case, because discount rate is marked as both static and final, is initialized immediately, and cannot be reassigned.

```
public static final BigDecimal DISCOUNT_RATE=BigDecimal.valueOf(0.1);
```

- b. Add a new method to the Product class to calculate and return discount value.

Hints:

- Make a new line just after the last method in the Product class to place the discount calculation method.
- Make this method `public`.
- Use `BigDecimal` as the return type.
- Use `getDiscount` as the method name.
- No parameters are required for this method.
- You will add actual discount calculation logic to the body of this new method in the next step of the practice.

```
public BigDecimal getDiscount() {
    // discount calculation logic will be added here
}
```

Note: At the moment, the `Product` class does not compile because the return statement is not yet present in the `getDiscount` method.

- Add logic to the `getDiscount` method that calculates and returns the discount value by applying the discount rate to product price. Ensure that calculated value is rounded to two decimal digits, using half up rounding mode.

```
return price.multiply(DISCOUNT_RATE).setScale(2, HALF_UP);
```

- Add static import of the `java.math.RoundingMode.HALF_UP` constant

Hint: Click on the right side of the line of code to invoke the "Add import" pop-up menu

The screenshot shows a Java code editor with the following code:

```
38     public BigDecimal getDiscount() {
40         return price.multiply(DISCOUNT_RATE).setScale(2, HALF_UP);
41     }
42 }
```

A tooltip box is open over the line `return price.multiply(DISCOUNT_RATE).setScale(2, HALF_UP);`. The tooltip contains four options:

- 💡 Add import for `java.math.RoundingMode.HALF_UP`
- 💡 Create local variable "HALF_UP"
- 💡 Create field "HALF_UP" in `labs.pm.data.Product`
- 💡 Create parameter "HALF_UP"

and select the "Add import for `java.math.RoundingMode.HALF_UP`" option.

```
import static java.math.RoundingMode.HALF_UP;
```

Note: This import allows referencing `HALF_UP` constant inside the `Product` class without having to prefix it with the package and class name that actually contains this content (`java.math.RoundingMode`).

- Add code to the `Shop` class to print discount value together with the rest of the Product details.

Hint: Invoke the `getDiscount` method at the end of the expression that concatenates Product attribute values.

```
System.out.println(p1.getId()+" "+p1.getName()
+" "+p1.getPrice()+" "+p1.getDiscount());
```

- f. Compile and run your application.

Hint: Click the "Run Project" toolbar button.



```

20   public static void main(String[] args) {
21     Product p1 = new Product();
22     p1.setId(101);
23     p1.setName("Tea");
24     p1.setPrice(BigDecimal.valueOf(1.99));
25     System.out.println(p1.getId()+" "+p1.getName()+" "+p1.getPrice()+" "+p1.getDiscount());
26   }
27 }
28

```

Output - ProductManagement (run) X

```

ant -f /home/oracle/labs/practices/ProductManagement -Dnb.internal.action.name=run run
init:
Deleting: /home/oracle/labs/practices/ProductManagement/build/built-jar.properties
deps-jar:
Updating property file: /home/oracle/labs/practices/ProductManagement/build/built-jar.properties
Compiling 2 source files to /home/oracle/labs/practices/ProductManagement/build/classes
compile:
run:
101 Tea 1.99 0.20
BUILD SUCCESSFUL (total time: 3 seconds)

```

Note: Observe the product details printed on to the console.

3. Explore instance variable default initialization.

- Open the `Shop` class editor.
- Inside the `main` method of a `Shop` class place, comment on the setter method calls that initialize `Product` object attributes `id`, `name`, and `price`.

Hints:

- Select three lines of code that invoke the setter methods on the `p1` variable.
- Press `CTRL+ /` keys to place comments on these lines of code.
- Alternatively you can simply type `//` in front of each of these lines or type `/*` before and `*/` after these lines of code.

```

// p1.setId(101);
// p1.setName("Tea");
// p1.setPrice(BigDecimal.valueOf(1.99));

```

Note: Pressing `CTRL+ /` comments and uncomments selected lines of code in NetBeans.

- c. Compile and run your application.

Hint: Click the "Run Project" toolbar button.



The screenshot shows the Oracle JDeveloper IDE interface. In the top left, there's a code editor with Java code. In the bottom right, there's an 'Output' window showing the build process and an error message. The code in the editor is as follows:

```

20  public static void main(String[] args) {
21      Product p1 = new Product();
22      // pl.setId(101);
23      // pl.setName("Tea");
24      // pl.setPrice(BigDecimal.valueOf(1.99));
25      System.out.println(p1.getId()+" "+p1.getName()+" "+p1.getPrice()+" "+p1.getDiscount());
26  }
27 }
28

```

The 'Output' window shows the following log:

```

ant -f /home/oracle/labs/practices/ProductManagement -Dnb.internal.action.name=run run
init:
Deleting: /home/oracle/labs/practices/ProductManagement/build/built-jar.properties
deps-jar:
Updating property file: /home/oracle/labs/practices/ProductManagement/build/built-jar.properties
Compiling 1 source file to /home/oracle/labs/practices/ProductManagement/build/classes
compile:
run:
Exception in thread "main" java.lang.NullPointerException
    at labs.pm.data.Product.getDiscount(Product.java:58)
    at labs.pm.app.Shop.main(Shop.java:25)
/home/oracle/labs/practices/ProductManagement/nbproject/build-impl.xml:1353: The following error occurred while executing this line:
/home/oracle/labs/practices/ProductManagement/nbproject/build-impl.xml:973: Java returned: 1
BUILD FAILED (total time: 2 seconds)

```

Notes

- Observe the statement that prints the product producing `NullPointerException`.
 - The `Product` object has been created successfully.
 - No values were assigned to any of the instance variables of this product object, because your code did not invoke any setter methods on this product instance.
 - Instance variables of this product object were not explicitly initialized, so default values were applied: `int id` was set to 0, `String name` to null, and `BigDecimal price` was also set to null.
 - An attempt to print information about this product results in a `NullPointerException`, because your code invokes the `getDiscount` method, which is trying to calculate discount value using `price`, which was not initialized to point to any `BigDecimal` object and was set to null. An attempt to invoke any operations or access any variables upon a null object reference will always result in a `NullPointerException`.
- d. Inside the `main` method of a `Shop` class, remove comments from the line of code that invokes the `setPrice` method.

Hints

- Select the line of code that invokes the `setPrice` method upon the `p1` variable.
- Press the `CTRL+ /` keys to remove comments from this line of code:

```
// p1.setId(101);
// p1.setName("Tea");
p1.setPrice(BigDecimal.valueOf(1.99));
```

- e. Compile and run your application.

Hint: Click the "Run Project" toolbar button.



```

20  public static void main(String[] args) {
21      Product p1 = new Product();
22      //          p1.setId(101);
23      //          p1.setName("Tea");
24      p1.setPrice(BigDecimal.valueOf(1.99));
25      System.out.println(p1.getId()+" "+p1.getName()+" "+p1.getPrice()+" "+p1.getDiscount());
26  }
27 }
28

```

labs.prn.app.Shop > main >

Output - ProductManagement (run) x

```

ant -f /home/oracle/labs/practices/ProductManagement -Dnb.internal.action.name=run run
init:
Deleting: /home/oracle/labs/practices/ProductManagement/build/built-jar.properties
deps-jar:
Updating property file: /home/oracle/labs/practices/ProductManagement/build/built-jar.properties
Compiling 1 source file to /home/oracle/labs/practices/ProductManagement/build/classes
compile:
run:
0 null 1.99 0.20
BUILD SUCCESSFUL (total time: 2 seconds)

```

Notes

- Observe 0 and null values printed for Product id and name on to the console.
- Although the name instance variable of the Product object is actually referencing null, it does not cause a NullPointerException, because you are not trying to invoke any methods on it.
- The variable price is now correctly initialized and thus can be used to calculate discount.

- f. Inside the `main` method of a `Shop` class, remove all remaining comments from the segment of code that invokes the setter methods upon the `p1` object reference.

Hints

- Select the lines of code that invoke `setId` and `setName` methods upon the `p1` variable.
- Press the `CTRL+ /` keys to remove comments from these lines of code.

- g. Compile and run your application.

Hint: Click the "Run Project" toolbar button.



```

20  public static void main(String[] args) {
21      Product p1 = new Product();
22      p1.setId(101);
23      p1.setName("Tea");
24      p1.setPrice(BigDecimal.valueOf(1.99));
25      System.out.println(p1.getId()+" "+p1.getName()+" "+p1.getPrice()+" "+p1.getDiscount());
26  }
27
28

```

Output - ProductManagement (run) x

```

ant -f /home/oracle/labs/practices/ProductManagement -Dnb.internal.action.name=run run
init:
Deleting: /home/oracle/labs/practices/ProductManagement/build/built-jar.properties
deps-jar:
Updating property file: /home/oracle/labs/practices/ProductManagement/build/built-jar.properties
Compiling 1 source file to /home/oracle/labs/practices/ProductManagement/build/classes
compile:
run:
101 Tea 1.99 0.20
BUILD SUCCESSFUL (total time: 3 seconds)

```

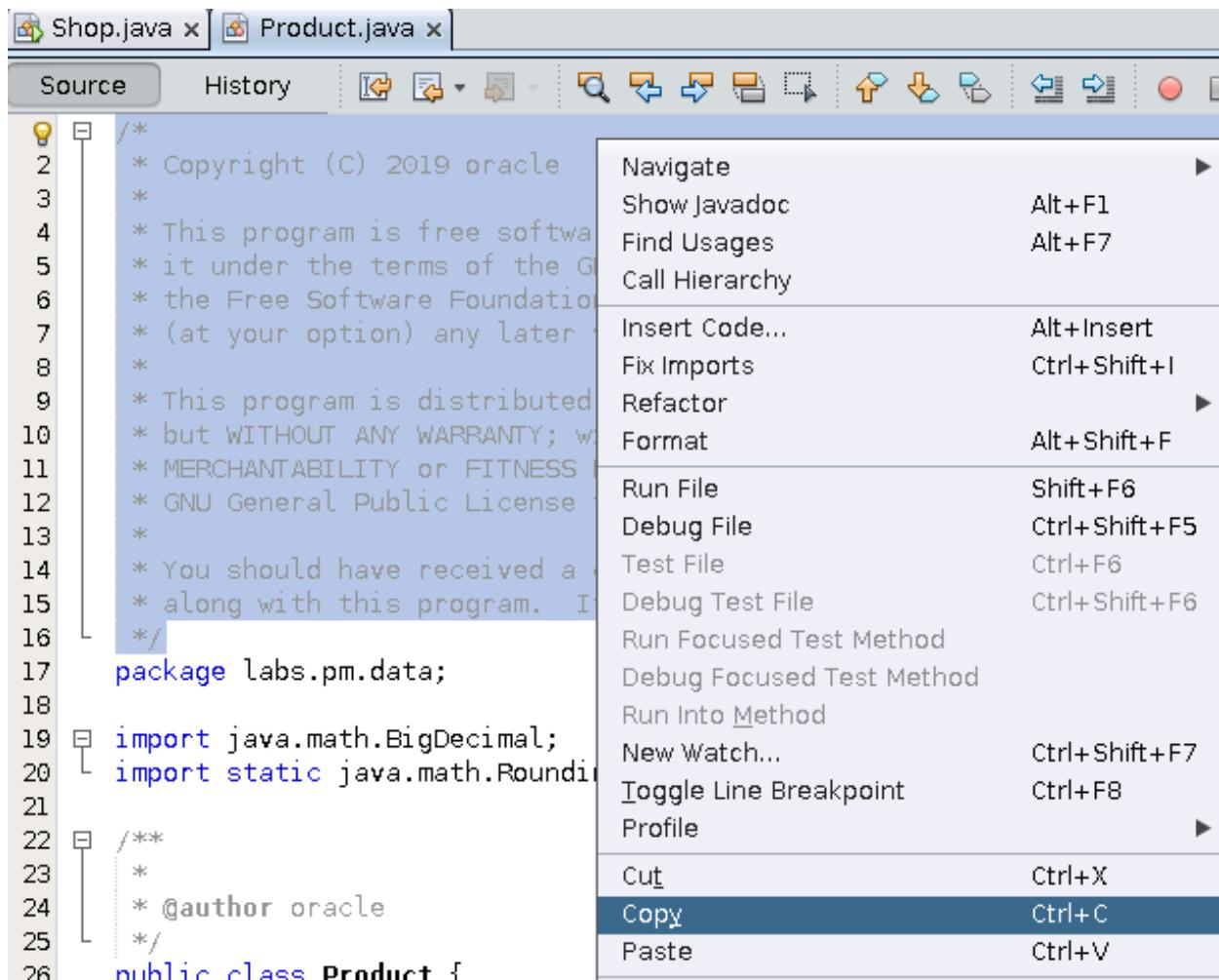
Note: Observe the product details printed on to the console.

Practice 4-3: Document Classes

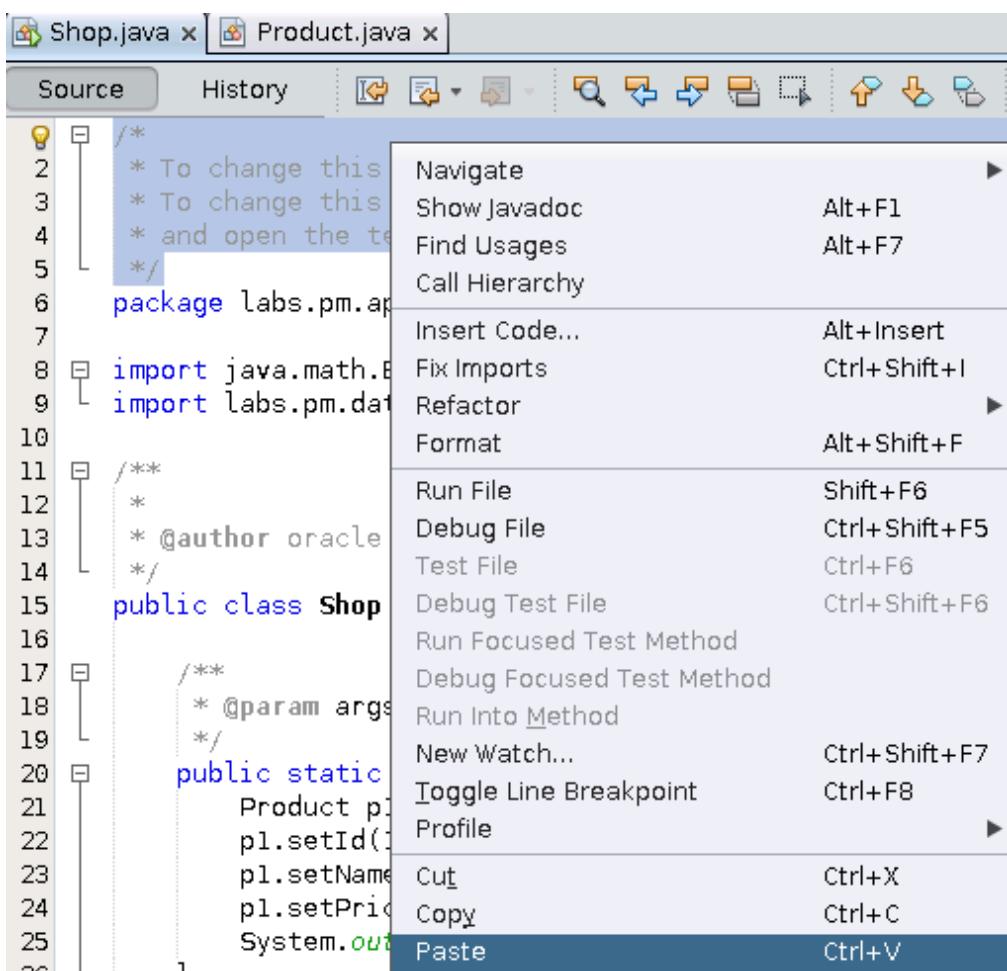
Overview

In this practice, you provide documentation comments and generate Javadoc.

1. Add documentation comments to Product and Shop classes.
 - a. Open the Product class editor.
 - b. Select license documentation comment section and copy it (CTRL+C).



- Open the `Shop` class editor.
- Select the license documentation comment section and replace this section with text copied from the `Product` class. Paste license text (CTRL+V).



- Add documentation describing the `Shop` class.

Hints

- Add text to the existing documentation comment that describes the `Shop` class.
- In one short sentence, describe the general purpose of the `Shop` class.
- Specify `version` and `author` attributes. (Consider using practice number as a version.)
- Optionally, format the word `Shop` in the description using code style.

```
/**
 * {@code Shop} class represents an application that manages Products
 * @version 4.0
 * @author oracle
 */
```

- g. Open the `Product` class editor.
- h. Add documentation describing the `Product` class:

Hints:

- Add text to the existing documentation comment that describes the `Product` class.
- Describe the general purpose of the `Product` class, its attributes (since the main purpose of this class is to represent information), and significant product specific features, such as ability to calculate discount.
- You may use HTML markups to format documentation text.
- Specify the `version` and `author` attributes. (Consider using practice number as a version.)
- Optionally, format the word `Product` in the description using code style.
- Describe `DISCOUNT_RATE` as a clickable link, which references the part of documentation that describes this constant.

```
/***
 * {@code Product} class represents properties and behaviours of
 * product objects in the Product Management System.
 * <br>
 * Each product has an id, name, and price
 * <br>
 * Each product can have a discount, calculated based on a
 * {@link DISCOUNT_RATE discount rate}
 * @version 4.0
 * @author oracle
 */
```

- i. Add documentation describing the `DISCOUNT_RATE` constant:

Hints:

- Add an empty line just above the line of code that describes `DISCOUNT_RATE` constant.
- Type `/**` and press Enter.
- Provide a description of the `DISCOUNT_RATE` constant, including its value.
- Optionally, describe a clickable link, which references the `BigDecimal` (type of this constant) documentation.

```
/***
 * A constant that defines a
 * {@link java.math.BigDecimal BigDecimal} value of the discount rate
 * <br>
 * Discount rate is 10%
 */
```

- j. Add documentation describing `getDiscount` method:

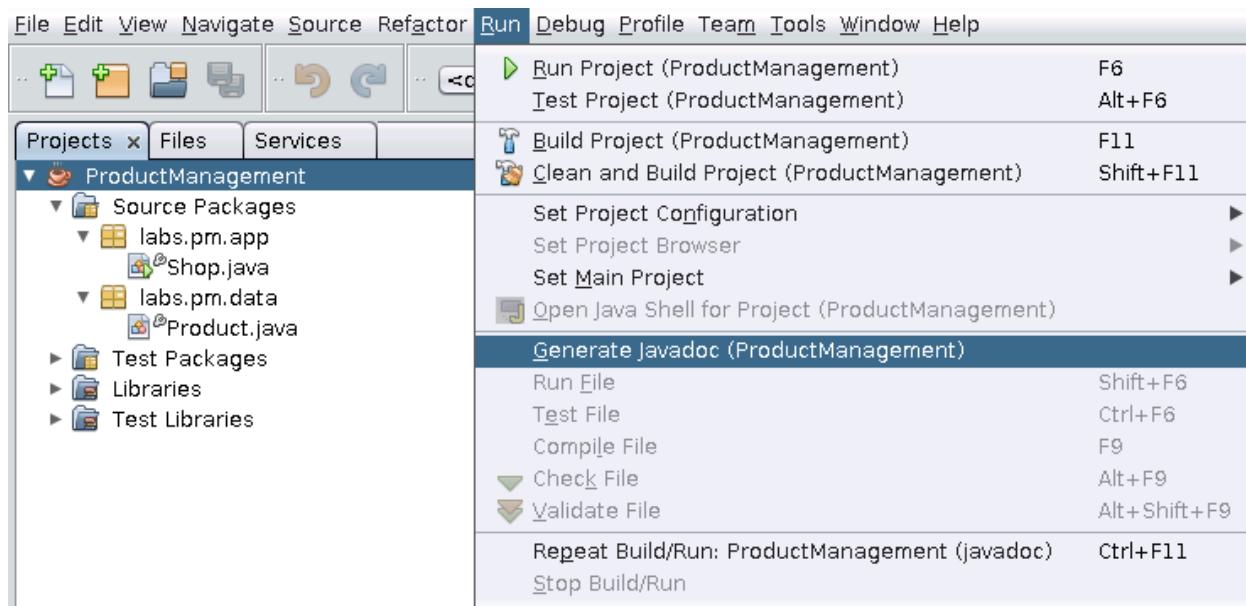
Hints:

- Add an empty line just above the line of code that describes the `getDiscount` method.
- Type `/**` and press Enter.
- Provide a description of the `getDiscount` method, including its returned value.
- Optionally, describe the clickable links, which reference `DISCOUNT_RATE` and `BigDecimal` documentation.

```
/**
 * Calculates discount based on a product price and
 * {@link DISCOUNT_RATE discount rate}
 * @return a {@link java.math.BigDecimal BigDecimal}
 * value of the discount
 */
```

2. Generate documentation for the ProductManagement project.

- a. Execute the Run->Generate Javadoc (ProductManagement) menu.



- b. Observe the documentation using the browser.

Hints

- After documentation generation is complete, NetBeans will invoke the browser to display your documentation.
- Navigate the documentation on classes `Shop` and `Product`.
- Observe the documentation on `DISCOUNT_RATE` constant and `getDiscount` method.
- Use clickable links embedded into the documentation.

Practices for Lesson 5: Improve Class Design

Practices for Lesson 5: Overview

Overview

In these practices, you improve Product class design by constraining values using Rating enumeration, ensuring consistent initialization of Product objects using constructors, and making Product objects immutable.



Practice 5-1: Create Enumeration to Represent Product Rating

Overview

In this practice, you create Enumeration to represent product ratings.

Assumptions

- JDK 11 is installed.
- NetBeans 11 is installed.
- You have completed Practice 4 or start with the solution for Practice 4 version of the application.

Tasks

1. Prepare the practice environment.

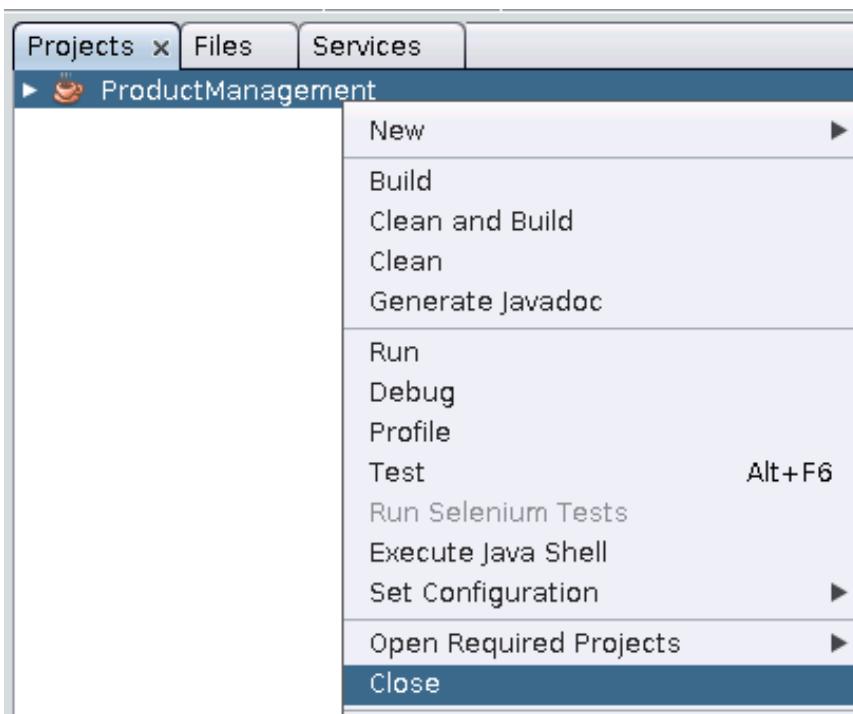
Notes

- You may continue to use the same NetBeans project as before, if you have successfully completed the previous practice. In this case, proceed directly to Practice 5-1, step 2.
- Alternatively, you can open a fresh copy of the NetBeans project, which contains the completed solution for the previous practice.
 - a. Open NetBeans (if it is not already running).



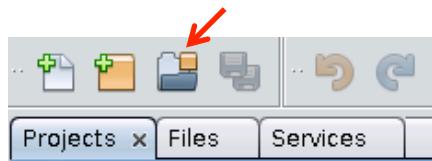
- b. Close the currently opened ProductManagement project.

Hint: Right-click the ProductManagement project and select "Close."



- c. Open the Solution for Practice 4 version of the ProductManagement project.

Hint: Use File -> Open Project menu or click the "Open Project" toolbar button.

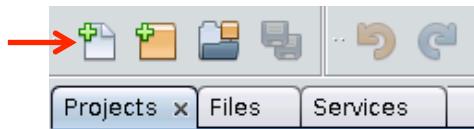


- d. Navigate to the /home/oracle/labs/solutions/practice4 folder and select ProductManagement project.
e. Click "Open Project."

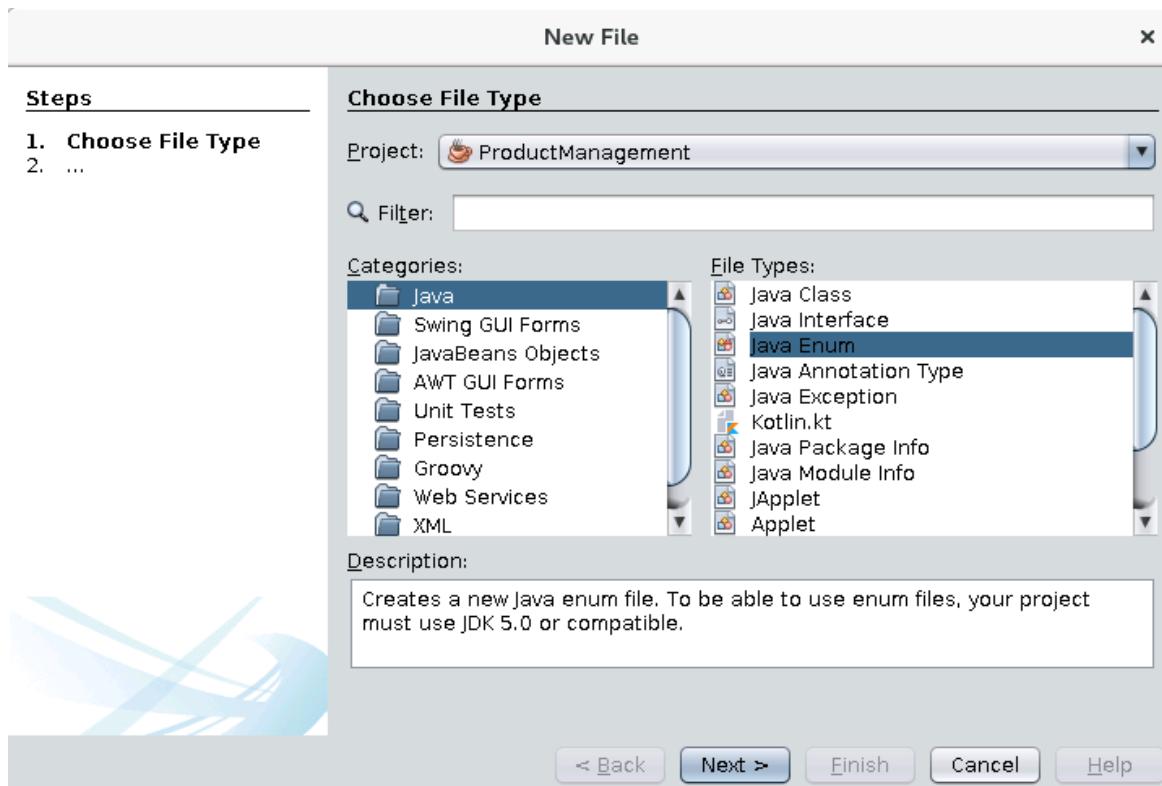
2. Create a new enumeration called Rating to represent product ratings.

- a. Create a new enumeration.

Hint: Use File -> New File menu or click the "New File" toolbar button.



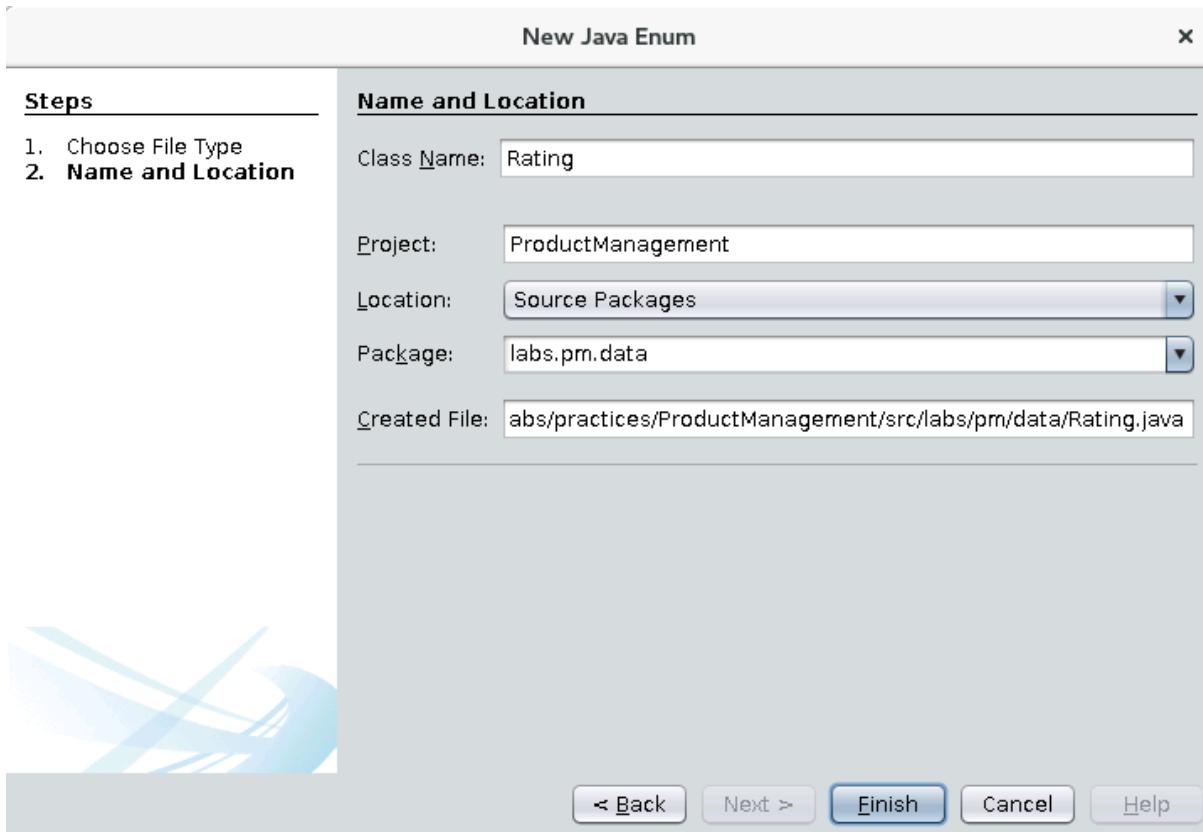
- b. Select "Java" Category and "Java Enum" as the file type.



- c. Click "Next."

d. Set the following enum properties:

- **Class Name:** Rating
- **Project:** ProductManagement
- **Location:** Source Packages
- **Package:** labs.pm.data



e. Click "Finish."

```
package labs.pm.data;  
public enum Rating {  
    // enumeration code will be added here  
}
```

- f. Inside the Rating enum, add six possible enumeration values to represent products that are not rated as well as products rated from 1 to 5 stars. Associate each rating with its own string to represent the number of stars:

Hints:

- Enumeration values are essentially constant. Therefore, a usual constant naming convention (uppercase with underscore between words) should be applied.
- Initialize text to represent the star rating of the product using unicode symbols for black \u2605 ★ and white stars \u2606 ☆

```
NOT_RATED("\u2606\u2606\u2606\u2606\u2606"),
ONE_STAR("\u2605\u2606\u2606\u2606\u2606"),
TWO_STAR("\u2605\u2605\u2606\u2606\u2606"),
THREE_STAR("\u2605\u2605\u2605\u2606\u2606"),
FOUR_STAR("\u2605\u2605\u2605\u2605\u2606"),
FIVE_STAR("\u2605\u2605\u2605\u2605\u2605");
```

Note: At this point, Rating enumeration would not compile, because it lacks actual instance variable declaration for the stars text, as well as appropriate constructor to initialize it.

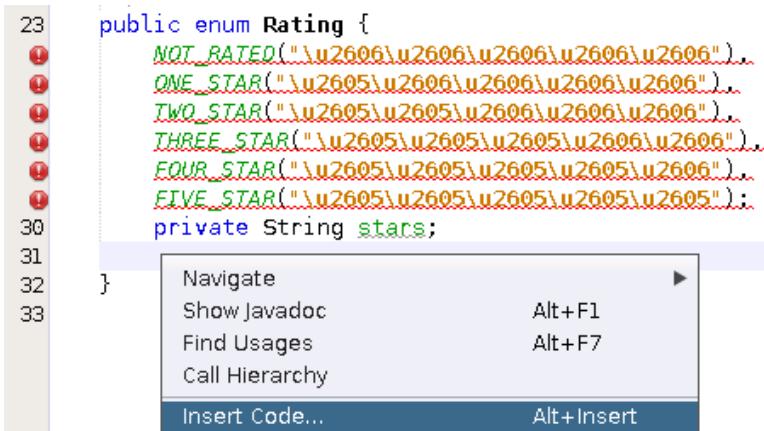
- g. Add a private instance variable called stars of type String to the Rating enumeration:

```
private String stars;
```

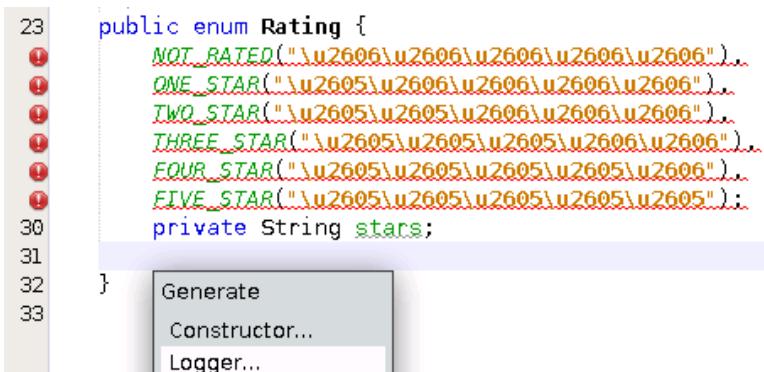
- h. Add constructor to the Rating enumeration, which sets stars variable value.

Hints

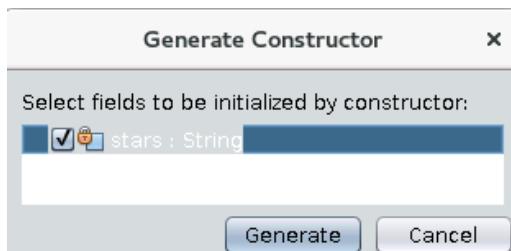
- Create a new line inside the Rating enum, just after the stars variable declaration.
- Right-click this empty line and select "Insert Code..." menu.



- Then select the "Constructor..." menu.



- Select the check box next to the stars variable.



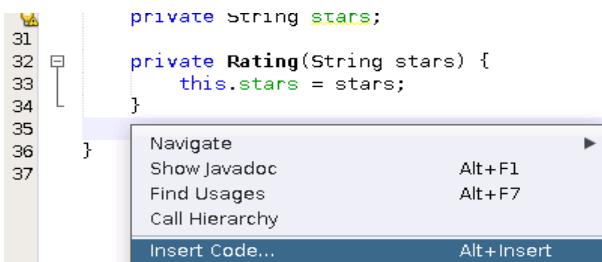
- Click the "Generate" button.

```
private Rating(String stars) {
    this.stars = stars;
}
```

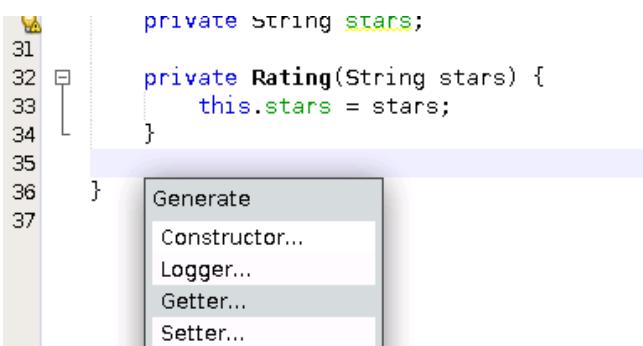
- i. Add the getter method to the Rating enumeration, which returns the stars variable value.

Hints:

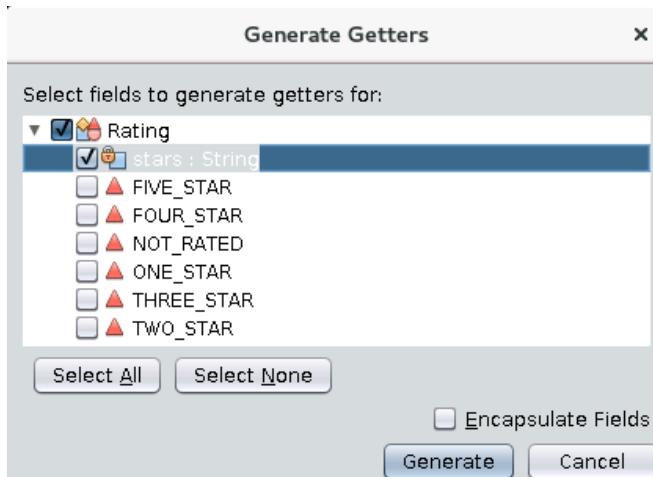
- Create new line inside Rating enum, just after the end of the constructor method body.
- Right-click this empty line and select "Insert Code..."



- Then select the "Getter..." menu



- Select the check box next to the stars variable.



- Click the "Generate" button.

```
public String getStars() {
    return stars;
}
```

3. Associate the Product class with the Rating enumeration.
- Open the Product class editor.
 - Add an instance variable called rating using the Rating enum as the variable type.

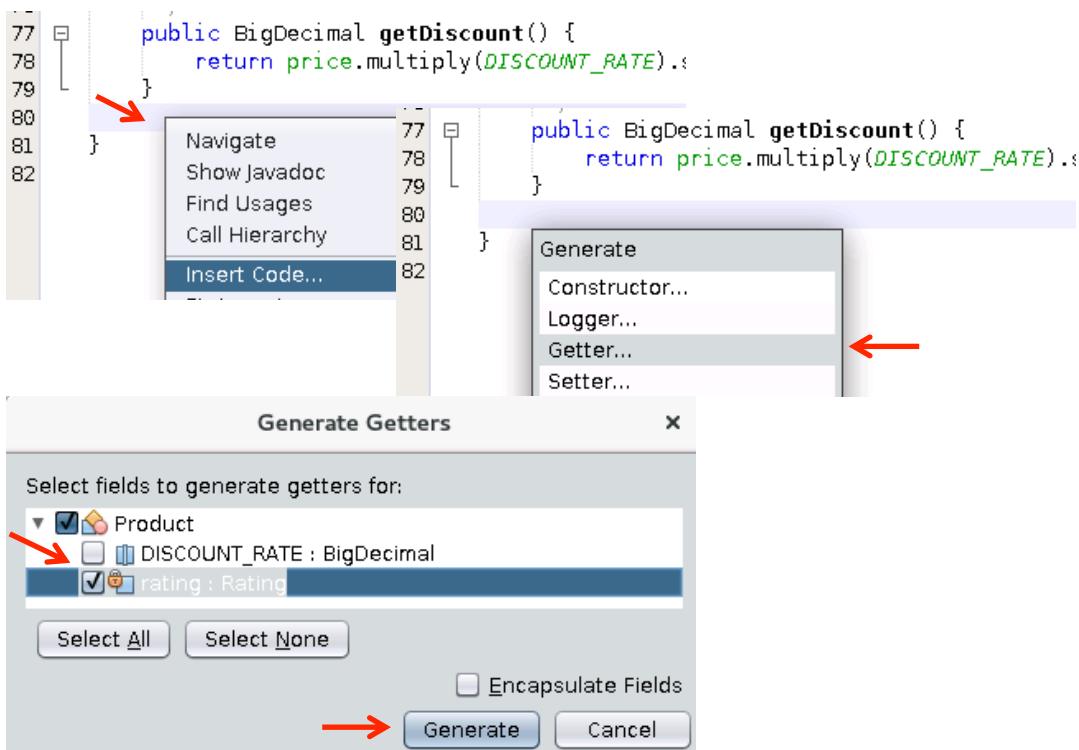
Hints:

- Add a new line after the last instance variable declaration inside the Product class.
 - Add the rating variable declaration to this new line of code.
 - Mark rating with private access modifier to ensure encapsulation.
- ```
private Rating rating;
```

- Add the getter method to the Product class to return the rating value.

**Hints:**

- Create a new line inside Product class, just after the last method.
- Right-click this empty line and select "Insert Code..."
- Then select the "Getter..." menu.
- Select the check box next to the rating variable.



- Click "Generate" button.

```
public Rating getRating() {
 return rating;
}
```

## Practice 5-2: Add Custom Constructors to the Product Class

### Overview

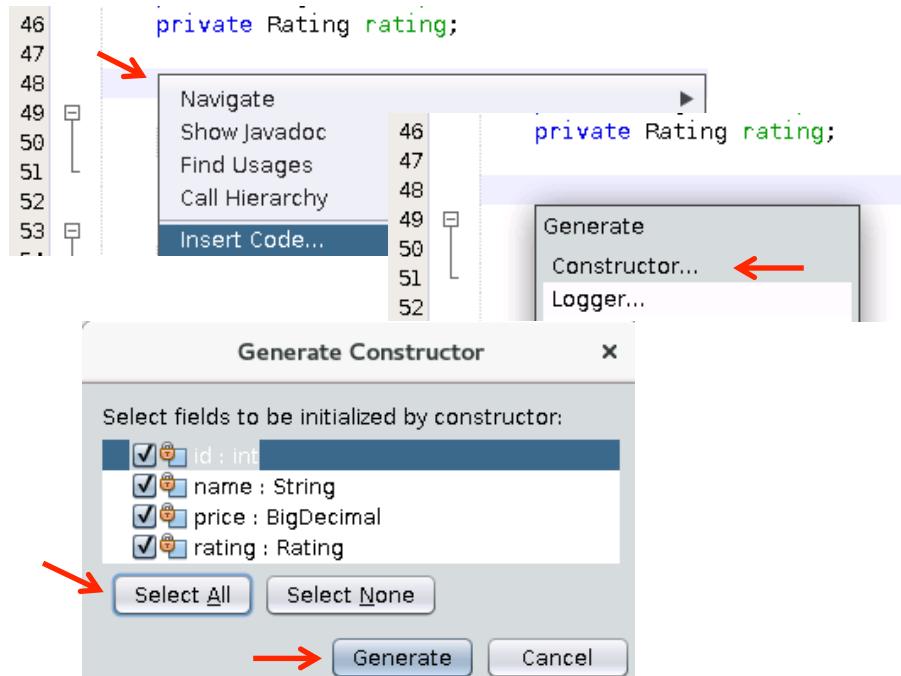
In this practice, you ensure consistent initialization of Product objects by adding custom constructors to the Product class.

#### 1. Add constructors to the Product class.

- a. Add a constructor to the `Product` class, which initializes `Product` using `id`, `name`, `price`, and `rating` values.

#### Hints:

- Add a new line after the last instance variable declaration inside the `Product` class.
- Right-click on this empty line and select "Insert Code..."
- Then select the "Constructor..." menu.
- Select all check boxes (against `id`, `name`, `price`, and `rating` variables)



- Click the "Generate" button.

```
public Product(int id, String name,
 BigDecimal price, Rating rating) {
 this.id = id;
 this.name = name;
 this.price = price;
 this.rating = rating;
}
```

- b. Add constructor to the `Product` class, which initializes `Product` using `id`, `name`, and `price` values.

#### Hints

- You can copy and paste the existing constructor code within the `Product` class.
- Remove the `rating` parameter from this newly pasted code fragment:

```
public Product(int id, String name, BigDecimal price) {
 // constructor logic will be placed here
}
```

- c. Reuse the existing constructor logic, by passing `id`, `name`, and `price` values, and then setting the default value for the `rating` argument as `Rating.NOT_RATED` from the newly created constructor.

**Hint:** Use the `this` keyword and matching constructor signature to invoke one constructor from the other.

```
public Product(int id, String name, BigDecimal price) {
 this(id, name, price, Rating.NOT_RATED);
}
```

#### Notes

- You have two alternative ways of creating `Product` objects, by overloading `Product` constructor.
- One of the constructors of the `Product` is using the code of another constructor.
- The default `no-arg` is no longer available in the `Product` class.
- Optionally, you could have also used static import of `Rating` enumeration, to avoid prefixing enum values: `import static labs.pm.data.Rating.*;`

2. Modify the class `Shop` to create the `Product` object using its constructors.

- a. Open the `Shop` class editor.

#### Notes:

- Class `Shop` is now broken. It relied upon the `no-arg` constructor that existed in the `Product` class implicitly, because no other constructor was provided. Now that you have actually added constructor with parameters to the `Product` class, this `no-arg` constructor is no longer automatically added to the `Product`. This prevents the `Shop` class from compiling because it is currently trying to create an instance of `Product` using the `no-arg` constructor: `new Product();`
- There are two ways of addressing this problem. The first option is to replace the `Shop` class code to use the constructor available in the `Product` class. The second option is to add constructor with no parameters to the `Product` class in addition to other constructors.

- b. Modify the way in which the `Product` object is created in the `Shop` class. Use `Product` constructor with `id`, `name`, and `price` parameters instead of the `no-arg` constructor. Use the following values: 101, Tea, 1.99

```
Product p1 = new Product(101, "Tea", BigDecimal.valueOf(1.99));
```

- c. Remove the setter method invocations from the `main` method of the `Shop` class.  
**Hint:** Remove invocations of `setId`, `setName`, and `setPrice` methods.
- d. Add code to the `Shop` class to print rating stars text together with the rest of the Product details.

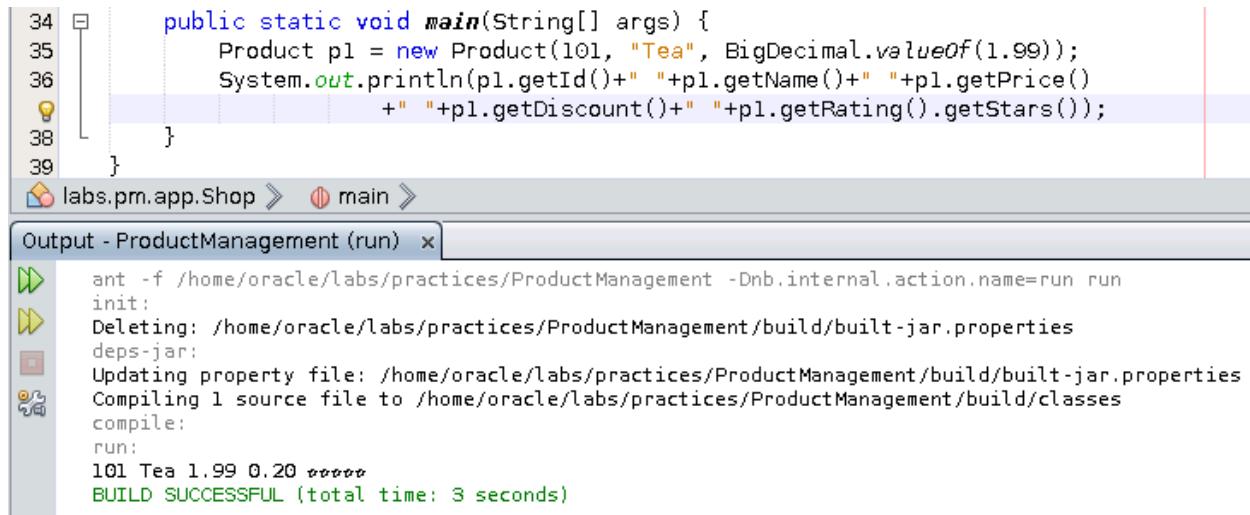
#### Hints

- Use chain invocation of `getter` methods to retrieve the `rating` attribute of the Product, followed by the stars text associated with this rating.
- Place this method call chain at the end of the expression that concatenates Product attribute values.

```
System.out.println(p1.getId()+" "+p1.getName()
+ " "+p1.getPrice()+" "+p1.getDiscount()
+ " "+p1.getRating().getStars());
```

- e. Compile and run your application.

**Hint:** Click the "Run Project" toolbar button.

The screenshot shows an IDE interface. In the top-left, there's a code editor with a snippet of Java code. Lines 34 through 39 are visible, showing the `main` method. The line `+ " "+p1.getRating().getStars());` is highlighted with a light purple background. In the bottom-right, there's a terminal window titled "Output - ProductManagement (run)" showing the results of a Maven build and the execution of the `main` method. The output includes:

```
ant -f /home/oracle/labs/practices/ProductManagement -Dnb.internal.action.name=run run
init:
Deleting: /home/oracle/labs/practices/ProductManagement/build/built-jar.properties
deps-jar:
Updating property file: /home/oracle/labs/practices/ProductManagement/build/built-jar.properties
Compiling 1 source file to /home/oracle/labs/practices/ProductManagement/build/classes
compile:
run:
101 Tea 1.99 0.20 *****
BUILD SUCCESSFUL (total time: 3 seconds)
```

**Note:** Observe product details printed on to the console.

3. Create two more instances of the `Product` using constructor with `id`, `name`, `price`, and `rating` parameters in the `main` method of the `Shop` class.
- Inside the `main` method of the `Shop` class, just after the line of code that instantiated first product object, add the following code:
    - Declare a variable called `p2` of type `Product`
    - Create a new instance `Product` using constructor with `id`, `name`, `price`, and `rating` parameters. Use the following values:  
`102, "Coffee", 1.99` and set rating to `FOUR_STAR`
    - Initialize the `p2` variable to reference the `Product` object you have just created.

```
Product p2 = new Product(102, "Coffee", BigDecimal.valueOf(1.99),
Rating.FOUR_STAR);
```

- b. Add an import statement for the `labs.pm.data.Rating` class.
- Hint:** Right-click anywhere inside the `Shop` class and invoke the "Fix Imports" menu.
- ```
import labs.pm.data.Rating;
```
- c. Inside the `main` method of the `Shop` class, just after the line of code that instantiated second product object, add the following code:
- Declare a variable called `p3` of type `Product`
 - Create a new instance of `Product` using constructor with `id`, `name`, `price`, and `rating` parameters. Use the following values:
`103, "Cake", 3.99` and set rating to `FIVE_STAR`
 - Initialize the `p3` variable to reference the `Product` object you have just created.

```
Product p3 = new Product(103, "Cake", BigDecimal.valueOf(3.99),  
Rating.FIVE_STAR);
```

- d. Print `id`, `name`, `price`, `discount`, and `rating stars` text of the second and third product objects to the console as a one line of text each.

Hints:

- Add this code just after the line of code that printed the first product object.
- You may copy and paste the existing line of code that prints product and then change the reference from `p1` to `p2` or `p3` in each subsequent printout.
- Concatenate all attribute values using space between values.
- Print the resulting string to the console using `System.out.println` method.

```
System.out.println(p2.getId() + " " + p2.getName() + " " + p2.getPrice()  
+ " " + p2.getDiscount() + " " + p2.getRating().getStars());  
System.out.println(p3.getId() + " " + p3.getName() + " " + p3.getPrice()  
+ " " + p3.getDiscount() + " " + p3.getRating().getStars());
```

4. Add the no-arg constructor to the `Product` class.

- a. Open the `Product` class editor and add the no-arg constructor, just after the last instance variable declaration.

```
public Product() {  
}
```

- b. Open the `Shop` class editor and add one more line of code that creates a new `Product` object using the no-arg constructor.

- Declare a variable called `p4` of type `Product`.
- Create new instance of `Product` using constructor with no parameters.
- Initialize the `p4` variable to reference the `Product` object you have just created.

```
Product p4 = new Product();
```

- d. Print id, name, price, discount, and rating stars text of the fourth product object to the console as a single line of text.

Hints:

- Add this code just after the line of code that printed the third product object.
- You may copy and paste an existing line of code that prints product and then change the reference to p4.
- Concatenate all attribute values using space between values.
- Print the resulting string to the console using `System.out.println` method.

```
System.out.println(p4.getId() + " " + p4.getName() + " " + p4.getPrice()
    + " " + p4.getDiscount() + " " + p4.getRating().getStars());
```

- e. Compile and run your application.



Hint: Click the "Run Project" toolbar button.

```
101 Tea 1.99 0.20 *****
102 Coffee 1.99 0.20 *****
103 Cake 3.99 0.40 *****
Exception in thread "main" java.lang.NullPointerException
at labs.pm.data.Product.getDiscount(Product.java:94)
at labs.pm.app.Shop.main(Shop.java:46)
/home/oracle/labs/practices/ProductManagement/nbproject/build-impl.xml:1353: The following error occurred while executing this line:
/home/oracle/labs/practices/ProductManagement/nbproject/build-impl.xml:973: Java returned: 1
BUILD FAILED (total time: 3 seconds)
```

Notes:

- Observe that the first three products printed successfully, but an attempt to print the fourth product produced `NullPointerException`.
- All product objects were created successfully.
- No values were assigned to any of the instance variables of the fourth product, because it was created using a no-arg constructor and your code did not invoke any setter methods upon this product instance.
- All instance variables of this last product object were not explicitly initialized, so default values were applied: `int id` was set to 0, `String name` to null, and `BigDecimal price` was also set to null.
- An attempt to print information about the fourth product results in a `NullPointerException`, because your code invokes the `getDiscount` method, which is trying to calculate the discount value using `price`, which was not initialized to point to any `BigDecimal` object and was set to null. An attempt to invoke any operations or access any variables upon a null object reference will always result in a `NullPointerException`.
- The execution path in this example has been terminated on the invocation of the `getDiscount` method, which has produced an exception. However, the invocation of the `getStars` method would have also produced the same exception type, because it is operating upon the `rating` variable, which is not initialized either and thus is set to null by default.

- f. Switch to the Product class editor.

- h. Modify `Product` class no-arg constructor to set default values for each instance variable.

Hints:

- Add the initialization code inside the no-arg constructor of the `Product` class.
- Use the following values:
set id to 0, name to "no name" and price to 0
- You can reuse existing constructors using this keyword followed by a matching constructor signature.
- Another constructor of the product class that you are invoking from the no-arg constructor is already coded to set rating to NOT_RATED.

```
public Product() {
    this(0, "no name", BigDecimal.ZERO);
}
```

Notes

- Alternatively, you could have assigned default variable to all instance variables of the `Product` directly, immediately when they are declared, rather than via the no-arg constructor.
- If an instance variable is already initialized, the constructor may still reassign its value, provided that such a variable is not marked with the `final` keyword (that is, it is not a constant).

- i. Compile and run your application.

Hint: Click the "Run Project" toolbar button.



```
101 Tea 1.99 0.20 *****
102 Coffee 1.99 0.20 *****
103 Cake 3.99 0.40 *****
0 no name 0 0.00 *****
BUILD SUCCESSFUL (total time: 2 seconds)
```

Note: Observe the product details printed on to the console.

Practice 5-3: Make Product Objects Immutable

Overview

In this practice, you ensure that `Product` objects are immutable by removing `setter` methods from the `Product` class. Provide an ability to create a replica `Product` with the adjusted price value.

1. Remove all `setter` methods from the `Product` class.
 - a. Open the `Product` class editor.
 - b. Remove all `setter` methods from the `Product` class

Hint: Simply delete the `setId`, `setName`, `setPrice` operations.

Notes

- Optionally, you may also mark instance variables `id`, `name`, and `price` with the `final` keyword to ensure that they can only be assigned once.
- Making the instance variable `final` will impose the following restrictions on your code:
 - Such a variable must be initialized immediately, or via instance initializer, or via all of these class constructors.
 - No other method may assign such a variable.

Your `Product` class design already satisfies these conditions.

2. Add an operation to the `Product` class that creates and returns a new `Product` object, which is a replica of the current `Product`, with the adjusted `rating` value.
 - a. Add a new method to the `Product` class to replicate, adjust, and return a new `Product` object.

Hints:

- Make a new line just after the last method in the `Product` class to place the new method.
- Make this method `public`.
- Use `Product` as the return type.
- Use `applyRating` as the method name.
- Method should expect new rating as an argument.
- You will add actual logic to the body of this new method in the next step of the practice.

```
public Product applyRating(Rating newRating) {
    // method logic will be added here
}
```

Note: At the moment, the `Product` class does not compile because the return statement is not yet present in the `applyRating` method.

- b. Add logic to the `applyRating` method that creates a new `Product` object, using all current product attributes, except `rating`, which must be set based on a parameter value. Return this product object.

```
return new Product(id, name, price, newRating);
```

Notes

- Product objects are immutable; their attributes cannot be changed. However, a new `Product` object can be created, which is a replica of the existing `Product` with any required attribute adjustments.
- This method is inside the `Product` class, so it can directly access any of its private variables or methods.
- Technically, your code is accessing current object variables except the `rating`:

```
return new Product(this.id, this.name, this.price, newRating);
```

However, inside the `applyRating` method, you did not write any code that would have "shadowed" any instance variables, so the use of `this` keyword is optional.

3. Use the `applyRating` operation to the `Product` class that creates and returns a new `Product` object, which is a replica of the current `Product`, with the adjusted `rating` value.

- a. Open the `Shop` class editor and add one more line of code that creates a new `Product` object as a result of adjusting the existing product.
- Create a new line of code immediately after the line of code that declares the `p4` variable.
 - On this new line, declare a variable called `p5` of type `Product`.
 - Initialize the `p5` variable to reference the `Product` object produced by invoking `applyRating` method on the `p3` variable.

```
Product p5 = p3.applyRating(Rating.THREE_STAR);
```

Note: You can reassign the `p3` variable to point to the newly created object instead of declaring additional variable, if you are no longer interested in the original `Product` object referenced by the `p3` variable:

```
p3 = p3.applyRating(Rating.THREE_STAR);
```

However, the reason you are asked to create additional variable is that it enables you to access both of the objects later, to point their details to the console and compare values. If you choose to reassign this reference, you would lose access to the original `Product` object referenced by the `p3` variable.

- b. Print `id`, `name`, `price`, `discount` and `rating stars` text of the fifth product object to the console as a single line of text.

Hints:

- Add this code just after the line of code that printed the fourth product object.
- You may copy and paste an existing line of code that prints product and then change the reference to `p5`.
- Concatenate all attribute values using space between values.
- Print the resulting string to the console by using `System.out.println` method.

```
System.out.println(p5.getId() + " " + p5.getName() + " " + p5.getPrice()
+ " " + p5.getDiscount() + " " + p5.getRating().getStars());
```

- c. Compile and run your application.

Hint: Click the "Run Project" toolbar button.



```
101 Tea 1.99 0.20 *****
102 Coffee 1.99 0.20 *****
103 Cake 3.99 0.40 *****
0 no name 0 0.00 *****
103 Cake 3.99 0.40 *****
BUILD SUCCESSFUL (total time: 0 seconds)
```

Note: Observe product details printed on to the console.

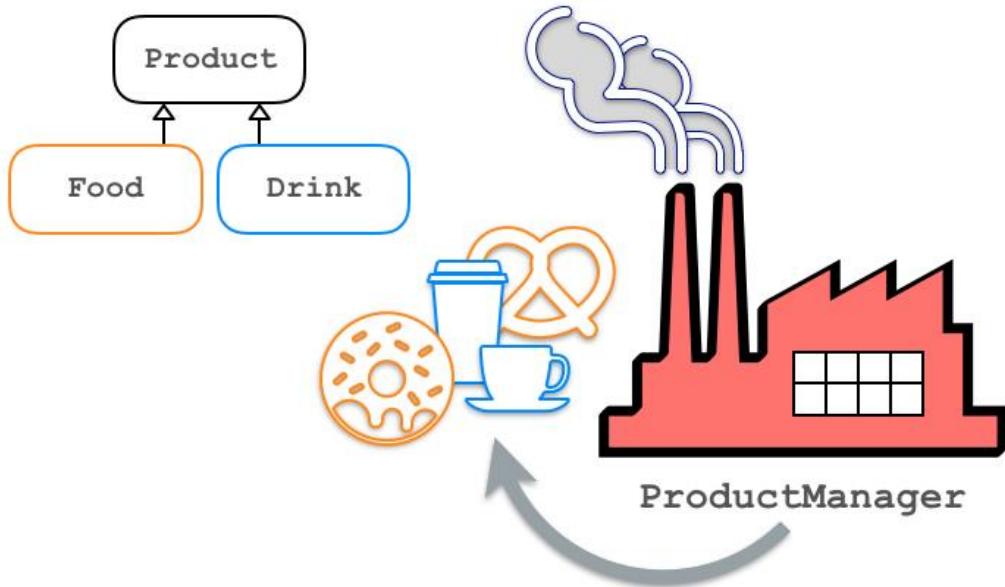
Practices for Lesson 6: Inheritance

Practices for Lesson 6: Overview

Overview

In these practices, you create the classes `Food` and `Drink` that extend the `Product` class, add subclass-specific attributes and methods, override parent class methods, and change the `Product` class from concrete to abstract. You also create the `ProductManager` class and add factory methods to it to create instances of `Food` and `Drink`. The `Shop` class is then modified to use these factory methods instead of using constructors of `Food` and `Drink` directly.

This practice is designed to demonstrate the design evolution of the Product Management application, explaining design decisions and alternative implementation options.



Practice 6-1: Create Food and Drink Classes That Extend Product

Overview

In this practice, you create the subclasses `Food` and `Drink` that extend the `Product` class.

Assumptions

- JDK 11 is installed.
- NetBeans 11 is installed.
- You have completed Practice 5 or start with the solution for Practice 5 version of the application.

Tasks

1. Prepare the practice environment.

Notes

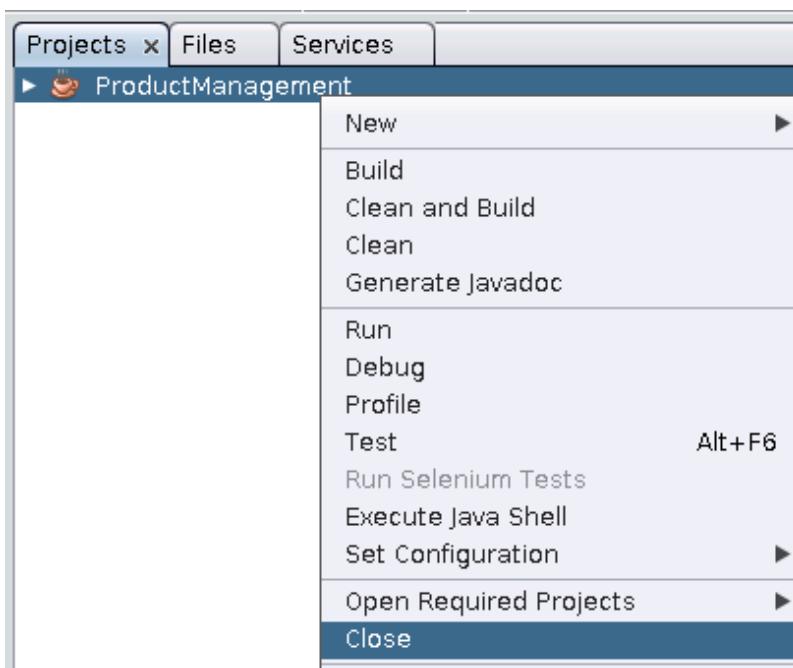
- You may continue to use the same NetBeans project as before, if you have successfully completed the previous practice. In this case, proceed directly to Practice 6-1, step 2.
- Alternatively, you can open a fresh copy of the NetBeans project, which contains the completed solution for the previous practice.

- a. Open NetBeans (if it is not already running).

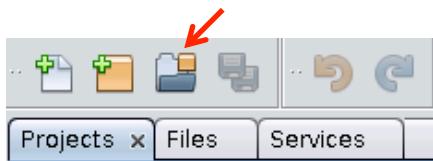


- b. Close the currently opened ProductManagement project.

Hint: Right-click the ProductManagement project and invoke the "Close" menu.

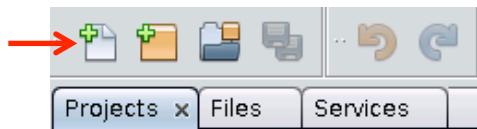


- c. Open the Solution for Practice 5 version of the ProductManagement project.
Hint: Use File -> Open Project menu or click the "Open Project" toolbar button.

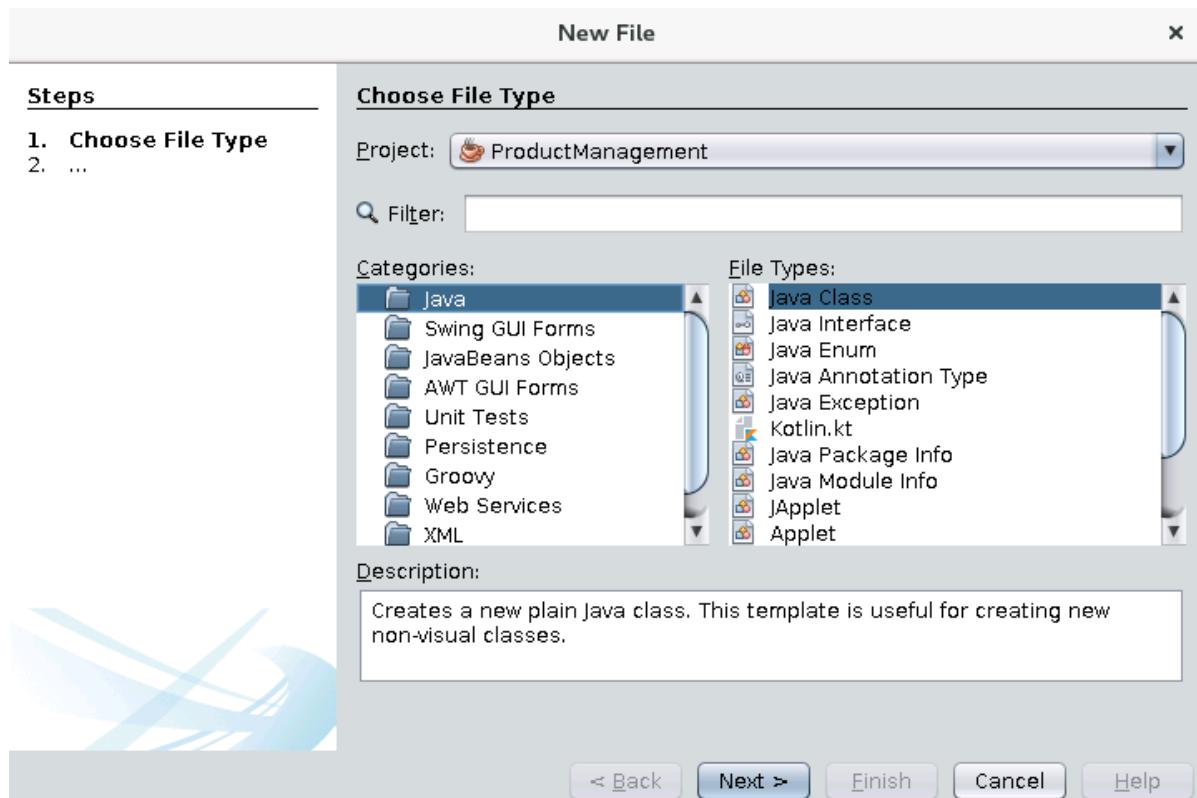


- d. Navigate to the /home/oracle/labs/solutions/practice5 folder and select the ProductManagement project.
e. Click "Open Project."
2. Create the classes Food and Drink that extend class Product.

- a. Create a new Java class.
Hint: Use File -> New File menu or click the "New File" toolbar button.



- b. Select "Java" Category and "Java Class" as the file type.



- c. Click "Next."

- d. Set the following class properties:

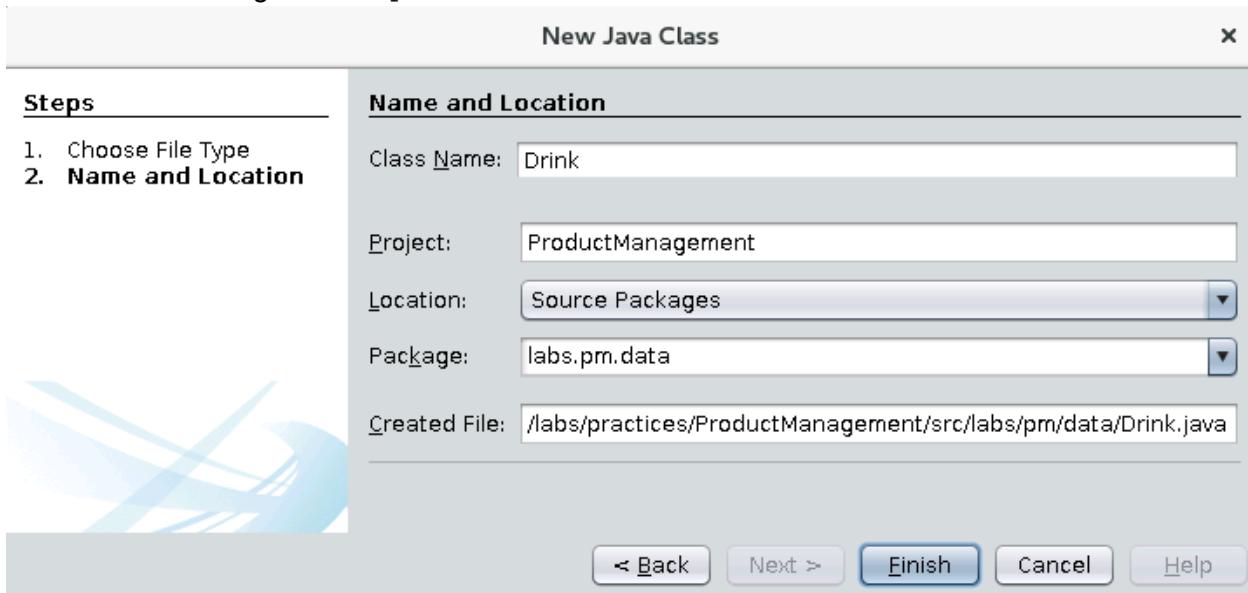
- **Class Name:** Food
- **Project:** ProductManagement
- **Location:** Source Packages
- **Package:** labs.pm.data



- e. Click "Finish."
- f. Add the extends clause to class definition, making class Food a subclass of Product:

```
package labs.pm.data;
public class Food extends Product {
    // more code will be added here
}
```

- g. Create a new Java class.
- Hint:** Use File -> New File menu or click the "New File" toolbar button.
- h. Select "Java" Category and "Java Class" as the file type.
 - i. Click "Next."
 - j. Set the following class properties:
 - Class Name: Drink
 - Project: ProductManagement
 - Location: Source Packages
 - Package: labs.pm.data



- k. Click "Finish."
 - l. Add the extends clause to class definition, making the class Drink a subclass of Product:


```
package labs.pm.data;
public class Drink extends Product {
    // more code will be added here
}
```
3. Explore the superclass constructor dependency.
- Notes**
- The next several steps of this practice are going to demonstrate the subclass constructor dependency on a superclass constructor.
 - You will be told to temporarily place comments on a no-arg superclass constructor and observe that the subclasses would fail to compile.
- a. Open the Product class editor.

- c. Place comments on a no-arg constructor of the Product class.

Hints

- Select three lines of code of the no-arg constructor of the Product class.
- Press **CTRL+ /** keys to place comments on these lines of code.
- Alternatively, you can simply enter `//` in front of each of these lines or enter `/*` before and `*/` after these lines of code.

```
//    public Product() {
//        this(0, "no name", BigDecimal.ZERO);
//    }
```

- d. Recompile the ProductManagement project.

Hint: Use Run->Clean and Build Project menu or a toolbar button



Notes:

- Classes Food, Drink, and Shop now fail to compile because they all use the no-arg constructor of a Product class.
- To fix this issue, you may either uncomment a no-arg constructor in a Product class or use constructors with parameters instead.

- e. Remove comments from the no-arg constructor of the Product class.

Hints

- Select three lines of code of the no-arg constructor of the Product class.
- Press **CTRL+ /** keys to remove comments these lines of code.

```
public Product() {
    this(0, "no name", BigDecimal.ZERO);
}
```

- f. Recompile ProductManagement project.

Hint: Use Run->Clean and Build Project menu or a toolbar button



Note: Classes Food, Drink, and Shop now successfully compile.

4. Add the best before date to the Food class and provide appropriate initialization logic

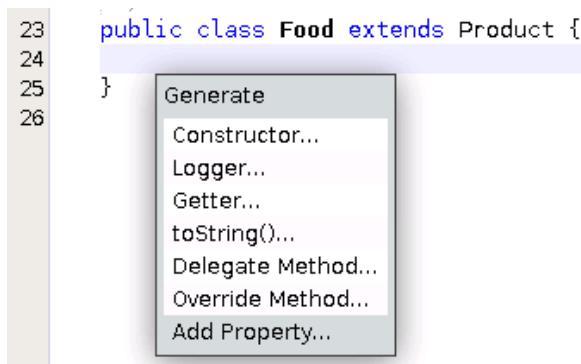
- a. Add an instance variable called `bestBefore` of type `LocalDate` to the Food class. Make this variable private to ensure proper encapsulation and add the `getter` method to provide access to the `bestBefore` variable.

Hints

- Right-click inside the Food class body and invoke the "Insert Code..." menu.

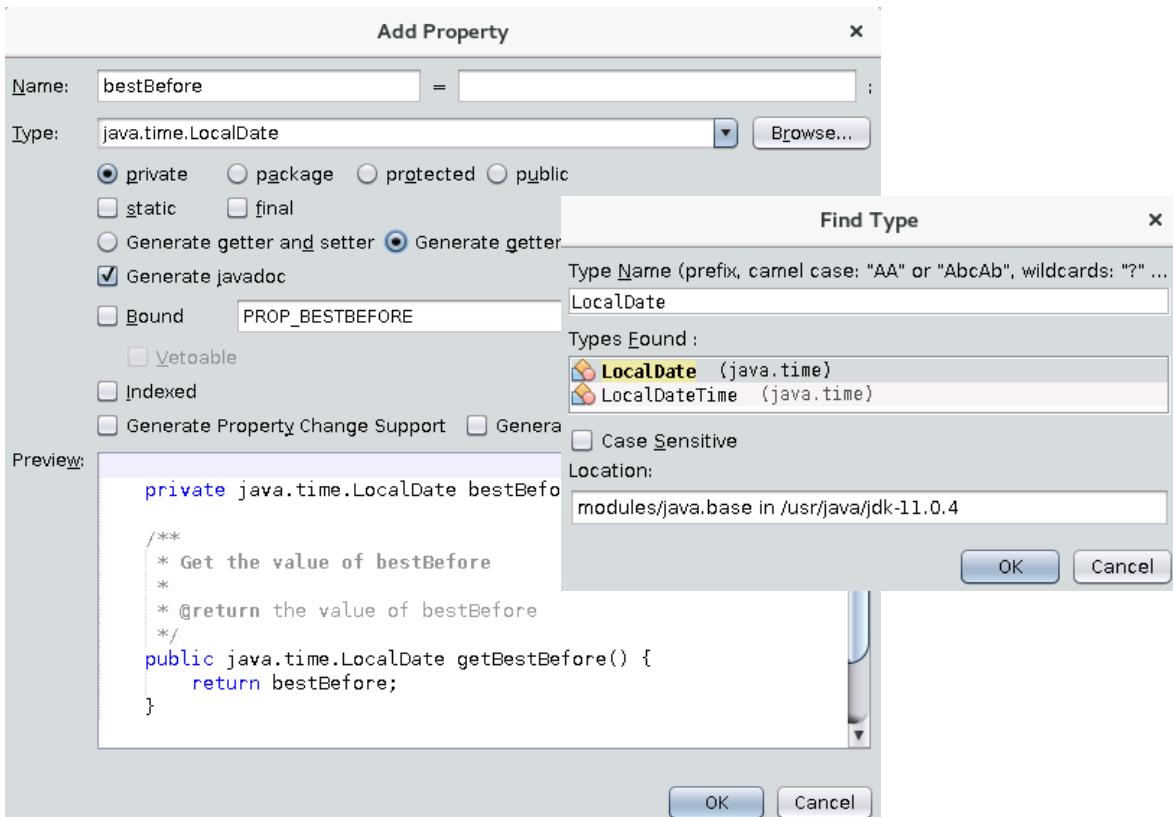


- Invoke the "Add Property..." menu.



- Set the following properties:

- Name: bestBefore
- Type: java.time.LocalDate
- (You can click the "Browse" button, type LocalDate in a search field, select LocalDate from the java.time package, and click "OK.")
- Select the "private" option.
- Select the "Generate getter" option.
- Select the "Generate javadoc" check box.



- Click "OK."

```

private LocalDate bestBefore;
/**
 * Get the value of bestBefore
 *
 * @return the value of bestBefore
 */
public LocalDate getBestBefore() {
    return bestBefore;
}

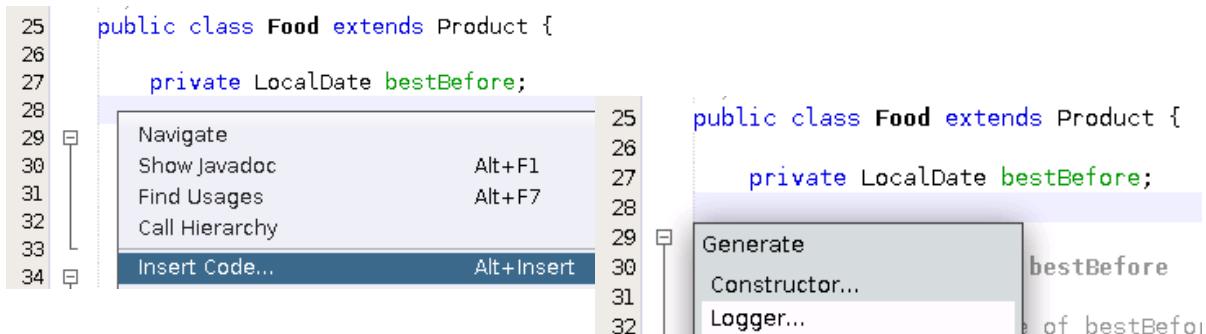
```

Note: NetBeans has generated an instance variable and a getter method and added an import for `java.time.LocalDate`.

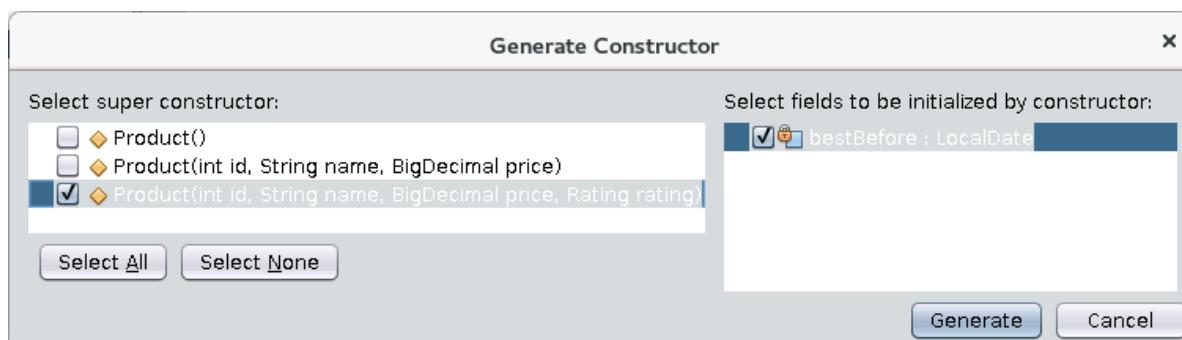
- Add constructor to the `Food` class, to initialize `id`, `name`, `price`, and `rating`, passing these values to the superclass (`Product`) constructor using superreference, and then initialize the `bestBefore` instance variable of the `Food` class.

Hints

- Right-click an empty line of code just after the declaration of the `bestBefore` variable, inside the `Food` class, and invoke "Insert Code..." and then "Constructor..." menu



- Select the following check boxes:
 - "`Product(int id, String name, BigDecimal price, Rating rating)`"
 - "`bestBefore: LocalDate`"



- Click "Generate".

```
public Food(LocalDate bestBefore, int id, String name,
           BigDecimal price, Rating rating) {
    super(id, name, price, rating);
    this.bestBefore = bestBefore;
}
```

- c. Reorder constructor arguments so that `bestBefore` appears after all arguments that are passed to the superclass.

Note: Strictly speaking, such reordering is not required, but it makes sense to provide constructor for subclasses with the same order of arguments as defined by the superclass.

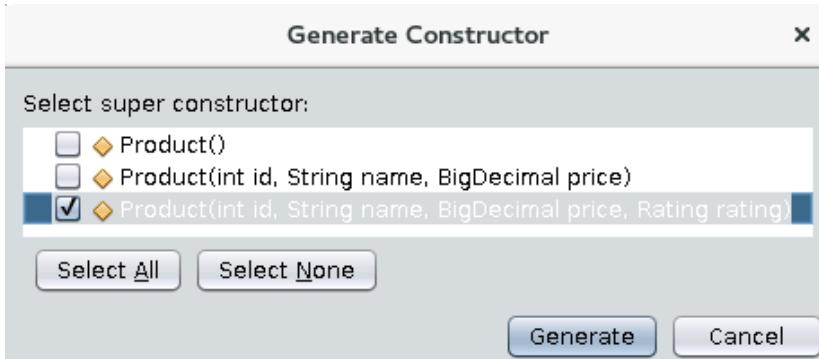
```
public Food(int id, String name, BigDecimal price,
           Rating rating, LocalDate bestBefore) {
    super(id, name, price, rating);
    this.bestBefore = bestBefore;
}
```

Note: If you would place comments on the no-arg constructor in the `Product` class again at this stage of the practice, the classes `Shop` and `Drink` would still fail to compile. However, the `Food` class would compile successfully, because it is now invoking another constructor on a `Product` class.

5. Provide appropriate initialization logic for the `Drink` class.
- Open `Drink` class editor.
 - Add constructor to the `Drink` class, to initialize `id`, `name`, `price`, and `rating`, passing these values to the superclass (`Product`) constructor using `super` reference.

Hints

- Right-click on an empty line of code inside the body of the `Drink` class and invoke "Insert Code..." and then "Constructor..." menu.
- Select the "Product(int id, String name, BigDecimal price, Rating rating)" check box.



- Click "Generate."

```

public Drink(int id, String name,
             BigDecimal price, Rating rating) {
    super(id, name, price, rating);
}

```

Note: The subclass does not have to define extra variables and methods in addition to those inherited from the superclass. The `Food` class provided such extra variables and methods, but the `Drink` class does not. However, each subclass is required to invoke the appropriate superclass constructor to initialize any superclass properties.

6. Modify the logic of a `main` method of a `Shop` class to use `Product` subclasses (`Food` and `Drink`).
 - a. Open the `Shop` class editor.
 - b. Modify a line of code that instantiates the second product (assigned to variable `p2`), to create an instance of `Drink`.

```
Product p2 = new Drink(102, "Coffee",
                       BigDecimal.valueOf(1.99), Rating.FOUR_STAR);
```

- c. Add an import statement for the `labs.pm.data.Drink` class.

Hint: Right-click anywhere inside the `Shop` class and invoke the "Fix Imports" menu.

```
import labs.pm.data.Drink;
```

- d. Modify a line of code that instantiates the third product (assigned to variable `p3`) to create an instance of `Food`.
- e. `Food` constructor requires an extra parameter of `LocalDate` type to indicate its best before date. Pass the extra parameter to the `Food` constructor to set the best before date as 2 days from today.

```
Product p3 = new Food(103, "Cake",
                      BigDecimal.valueOf(3.99), Rating.FIVE_STAR,
                      LocalDate.now().plusDays(2));
```

- f. Add an import statement for the `labs.pm.data.Food` and `java.time.LocalDate` classes.

Hint: Right-click anywhere in the source code of the `Shop` class and invoke "Fix Imports" menu.

```
import java.time.LocalDate;
import labs.pm.data.Food;
```

- g. Compile and run your application.

Hint: Click the "Run Project" toolbar button.



Note: Observe the product details printed on to the console.

Practice 6-2: Override Methods and Use Polymorphism

Overview

In this practice, you override `toString`, `equals`, and `hashCode` methods defined by the `Object` class and `getDiscount` and `applyDiscount` methods defined by the `Product` class. You make the `Product` class abstract and force the `applyDiscount` method to be overridden by child classes. The design assumption is that `applyDiscount` operation should be generally available for all `Products`, regardless of their specific subtype. It would be a bad design practice to add operations to the superclass, if they are not relevant to all of its subtypes.

1. Modify the logic of a `main` method of a `Shop` class to print information about products using the `toString` method.
 - a. Open the `Shop` class editor.
 - b. Replace code inside each `println` method call to simply print relevant product reference.

```
System.out.println(p1);
System.out.println(p2);
System.out.println(p3);
System.out.println(p4);
System.out.println(p5);
```

Notes

- The `println` method calls `String.valueOf(x)` (where `x` is the `println` method parameter) to get the printed object's string value.

```
public void println(Object x) {
    String s = String.valueOf(x);
    synchronized (this) {
        print(s);
        newLine();
    }
}
```

- The `valueOf` method checks if its parameter is `null`, in which case, it returns "null" String value; otherwise, it invokes the `toString` method upon this parameter.

```
public static String valueOf(Object obj) {
    return (obj == null) ? "null" : obj.toString();
}
```

- c. Compile and run your application.

Hint: Click the "Run Project" toolbar button.



```
labs.pm.data.Product@7382f612
labs.pm.data.Drink@1055e4af
labs.pm.data.Food@3caeaf62
labs.pm.data.Product@e6ea0c6
labs.pm.data.Product@6a38e57f
BUILD SUCCESSFUL (total time: 2 seconds)
```

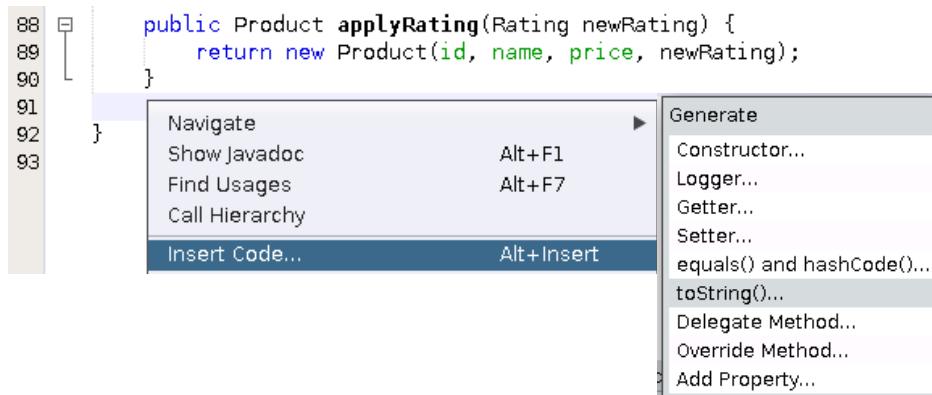
Notes

- Observe the `toString` method invocation results printed on to the console.
- These results are produced by the generic implementation of the `toString` method provided by the `Object` class.

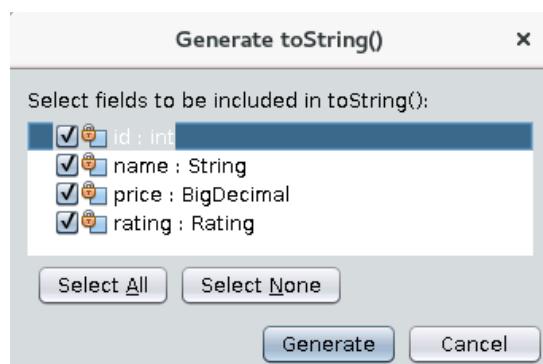
```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

2. Override the `toString` method of the `Object` class within the `Product` class.

- a. Open the `Product` class editor.
- b. Add an empty line just before the end of the `Product` class body.
- c. Right-click this new line and invoke "Insert Code..." and then "toString()..." menu.



- d. Select all the check boxes in the Generate `toString()` dialog box.



- e. Click "Generate."

```
@Override
public String toString() {
    return "Product{" + "id=" + id + ", name=" + name + ", price="
           + price + ", rating=" + rating + '}';
}
```

Notes

- To override a method, a child class (in this case Product) must provide a method with the signature that matches a relevant method signature in a parent class (in this case, Object).
 - The annotation `@Override` is optional and is not actually required to override a method.
 - The purpose of this annotation is to ensure that you would not accidentally create a method that you think overrides a method of a superclass, yet does not actually do it since you have not correctly matched the required method signature defined by the parent class.
 - You may try to change `toString` method signature (for example, call it `toStringX`) in a Product class and observe the `@Override` annotation causing an error.
 - Such an error would not have been produced if the annotation was not in place.
- f. Change the returned String to produce a simple comma-separated text comprising `id`, `name`, `price`, `discount`, and a `rating stars` text.

Hints

- `Product` defines instance variables for `id`, `name`, and `price`.
- Discount value is calculated by the `getDiscount` method.
- Stars is a text property of a `Rating` enumeration.

```
@Override
public String toString() {
    return id+", "+name+", "+price+", "
           +getDiscount()+", "+rating.getStars();
}
```

- h. Compile and run your application.

Hint: Click the "Run Project" toolbar button.



```
101, Tea, 1.99, 0.20, *****
102, Coffee, 1.99, 0.20, *****
103, Cake, 3.99, 0.40, *****
0, no name, 0, 0.00, *****
103, Cake, 3.99, 0.40, *****
BUILD SUCCESSFUL (total time: 3 seconds)
```

Notes

- Observe the product details printed on to the console.
- Due to polymorphism, the lowest available implementation of the method is automatically invoked, so you see results produced by the version of the `toString` method of the `Product` class.

3. Override the `toString` method within the `Food` class.

- a. Open the `Food` class editor.
- b. Add an empty line just before the end of the `Food` class body.
- c. Right-click this new line and invoke the "Insert Code..." and then "toString()..." menu
- d. Select all the check boxes in the Generate `toString()` dialog box.



- e. Click "Generate."

```
@Override
public String toString() {
    return "Food{" + "bestBefore=" + bestBefore + '}';
}
```

Notes

- The NetBeans utility that generates the `toString` method produces code that simply prints instance variables defined in a given class.

- f. Change the returned String to produce a simple comma-separated text, which comprises the output produced by the superclass version of the `toString` method with the added best before date attribute value.

Hint: Use the `super` keyword to invoke superclass version of the `toString` method to reuse the existing logic.

```
@Override
public String toString() {
    return super.toString() + ", " + bestBefore;
}
```

- g. Compile and run your application.

Hint: Click the "Run Project" toolbar button.



```
101, Tea, 1.99, 0.20, *****
102, Coffee, 1.99, 0.20, *****
103, Cake, 3.99, 0.40, *****, 2019-10-25
0, no name, 0, 0.00, *****
103, Cake, 3.99, 0.40, *****
BUILD SUCCESSFUL (total time: 2 seconds)
```

Notes

- Observe the product details printed on to the console and how they differ depending on the exact subtype of the product.
- Due to polymorphism, the lowest available implementation of the `toString` method is automatically invoked, which is adding a best before date for any `Food` object.
- No casting of the `Product` type variables to specific subtypes is required.

4. Compare the `Product` objects by using the `equals` method.

- Open the `Shop` class editor.
- Just before the end of the `main` method, declare two more variables of `Product` type `p6` and `p7`. Initialize the `p6` variable to reference a new instance of `Drink` and `p7` to reference a new instance of `Food`. Use the following values for `Drink` and `Food` constructors:
 - Both should use `id` of 104, a name of "Chocolate", and a price of 2.99.
 - Use `Rating.FIVE_STAR` for both `Drink` and `Food` instances.
 - Use current date plus two days as best before date for the `Food` instance.

```
Product p6 = new Drink(104, "Chocolate", BigDecimal.valueOf(2.99),
                      Rating.FIVE_STAR);
Product p7 = new Food(104, "Chocolate", BigDecimal.valueOf(2.99),
                      Rating.FIVE_STAR, LocalDate.now().plusDays(2));
c. Compare p6 and p7 objects by using the equals method and print the result to the
console.
System.out.println(p6.equals(p7));
```

- d. Compile and run your application.

Hint: Click the "Run Project" toolbar button.



Notes

- Observe that the result of comparing these objects is `false`. This is because at the moment your code is using the `equals` method logic provided by the `Object` class, which simply checks if these variables are pointing to the same object in the heap.

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

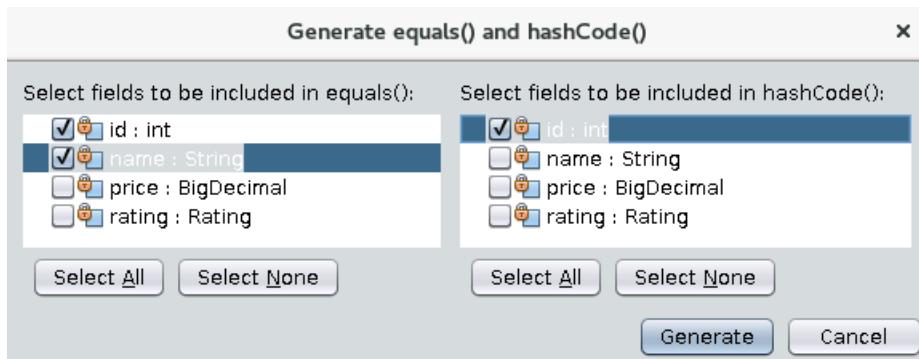
- Therefore, the only way this code would have returned true is if `p6` to `p7` would be referencing the same object, that is, `p6 = p7` or `p7 = p6`
- In the next step of this practice, you will override the `equals` method within the `Product` class and provide logic that compares product attributes. This practice assumes that products should be considered the same if they have the same id and name values. Use product id only to produce hash code value.

5. Override methods `equals` and `hashCode` to provide custom mechanism for comparing Product objects

- Open Product class editor.
- Add an empty line just before the end of the Product class body.
- Right-click this new line and invoke the "Insert Code..." and then "equals()" and "hashCode..." menus.



- Select `id` and `name` as fields to be included in the `equals` method and `id` only as to be included in the `hashCode` method.



- Click "Generate."

```

@Override
public int hashCode() {
    int hash = 7;
    hash = 89 * hash + this.id;
    return hash;
}

```

```

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Product other = (Product) obj;
    if (this.id != other.id) {
        return false;
    }
    if (!Objects.equals(this.name, other.name)) {
        return false;
    }
    return true;
}

```

Notes

- The `hashCode` method generates a new hash value based on a product id.
- The `equals` method first checks if the current object reference is the same as the parameter (which is actually the same as the generic algorithm of an `equals` method in the `Object` class). Essentially, in this case, `this == obj` is the same as `super.equals(obj)`
- Next the `obj` method parameter is compared to `null`, and if it is `null`, the `equals` method returns `false`.
- Next a check is performed to ensure that the parameter is of the same type (same class) as the current object.
- It is only safe to cast parameter to `Product` type if parameter is indeed of the same type as the current object (`Product`).
- The next two `if` statements could have been actually combined into a simple `return` statement, using short-circuit `&&` operator:

```
return this.id != other.id && !Objects.equals(this.name, other.name);
```

- The `equals` method of the `Objects` class (don't confuse this with the `Object` class) checks if the parameter it receives is not null before comparing its value to avoid producing `NullPointerException`.

```
public static boolean equals(Object a, Object b) {
    return (a == b) || (a != null && a.equals(b));
}
```

- Such a not null check could be considered excessive if the class is designed to guarantee that a certain attribute is never null (in your `Product` class design, the name is always initialized, since it is either explicitly set or defaulted by all constructors). This means that you can simply compare names directly, using the `equals` method provided by the `String` class:

`return this.id != other.id && this.name.equals(other.name);`

- f. Rewrite the logic of the `equals` method to shorten its algorithm:

```
@Override
public boolean equals(Object obj) {
    if (this==obj) { return true; }
    if (obj!=null && getClass()==obj.getClass()) {
        final Product other = (Product)obj;
        return this.id==other.id && Objects.equals(this.name, other.name);
    }
    return false;
}
```

- g. Compile and run your application.

Hint: Click the "Run Project" toolbar button.



Note: Observe that the result of comparing objects is still `false`. This is because the `equals` method uses the `getClass` operation to check if the parameter is of the same type as the current object. Comparing products in this example yields `false` even though they got identical id and name, because they are not of the same type - one is `Drink` and another is `Food`.

- h. Modify the algorithm of the `Product` `equals` method to use the `instanceof` operator to check if the parameter is of the same type as the current object:

```
@Override
public boolean equals(Object obj) {
    if (this==obj) { return true; }
    if (obj instanceof Product) {
        final Product other = (Product)obj;
        return this.id==other.id && Objects.equals(this.name, other.name);
    }
    return false;
}
```

Note: Not null check is no longer required, because the `instanceof` operator returns `false` if the parameter is null.

- j. Compile and run your application.

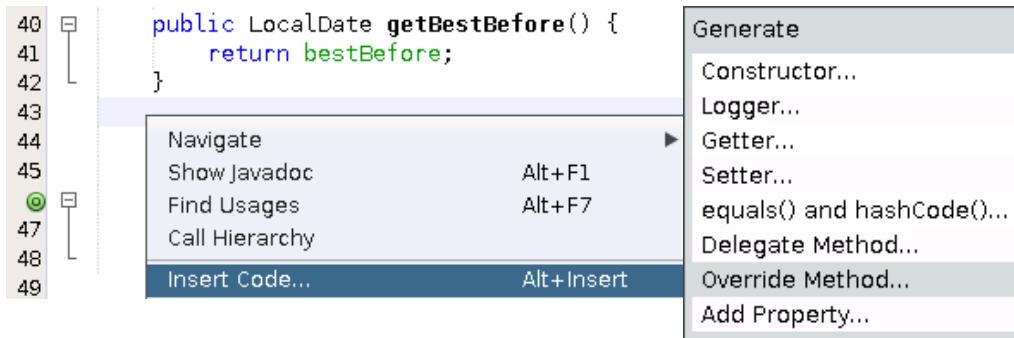
Hint: Click the "Run Project" toolbar button.



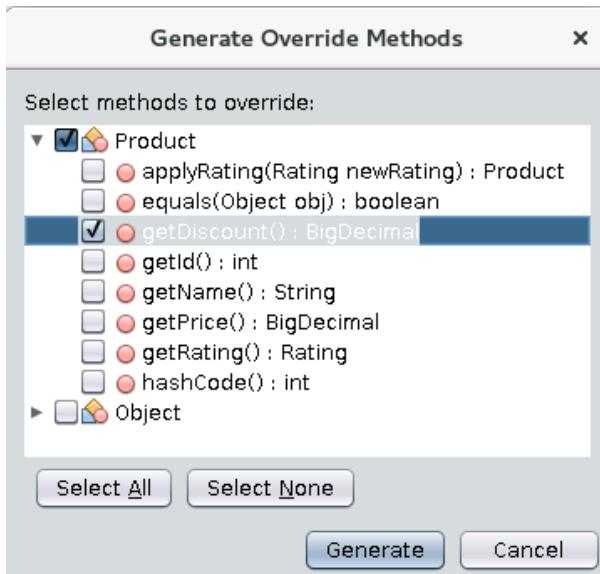
Notes

- In this case, the same compare products example is going to yield `true`, because the `equals` method algorithm now checks only if the parameter is a type of `Product` and ignores specific subtypes (`Food` or `Drink`).
- One way of implementing this logic is not necessarily better than the other. You may choose to allow products of different types to have the same id or not. This example essentially implies that product uniqueness verification requires to use a different id value for a chocolate that is a food and a chocolate that is a drink.
- The algorithm that is now implemented by the `equals` method of the `Product` class considers products with the same id and name to be the equal, regardless of a specific product subtype, price, or any other details.

7. Provide different algorithms to calculate discount for Food and Drink products.
- Open the `Food` class editor.
 - Add a new line just after the end of the `getBestBefore` method body.
 - Right-click this new empty line and invoke "Insert Code..." Override Method..." menu.



- Select the check box for the `getDiscount` method.



- Click "Generate."

```
@Override
public BigDecimal getDiscount() {
    return super.getDiscount();
}
```

- g. Modify logic of the `getDiscount` method in the `Food` class to calculate discount based on the best before date. Your algorithm should apply 10% discount if the product best before date is today.

Hints

- You can use the ternary operator `? :` to return different discount values based on the required condition.
- Reuse the existing logic of the superclass method `getDiscount` that returns a 10% discount value; otherwise, set discount to 0.

```
@Override
public BigDecimal getDiscount() {
    return (bestBefore.isEqual(LocalDate.now()))
        ? super.getDiscount() : BigDecimal.ZERO;
}
```

- h. Open the `Drink` class editor.
- i. Add a new line just before the end of the `Drink` class body.
- j. Right-click on this new empty line and invoke "Insert Code..." Override Method..." menu.
- k. Select the check box for the `getDiscount` method.
- l. Click "Generate."

```
@Override
public BigDecimal getDiscount() {
    return super.getDiscount();
}
```

- m. Modify the logic of the `getDiscount` method in the `Food` class to calculate discount based on the current time. Your algorithm should apply 10% discount between 16:30 and 17:30.

Hints

- Use `LocaleTime.now()` to get current time.
- You can use the ternary operator `? :` to return different discount values based on the required condition.
- Reuse existing logic of the superclass method `getDiscount` that returns a 10% discount value; otherwise, set discount to 0.

```
@Override
public BigDecimal getDiscount() {
    LocalTime now = LocalTime.now();
    return (now.isAfter(LocalTime.of(17,30)) &&
            now.isBefore(LocalTime.of(18,30)))
        ? super.getDiscount() : BigDecimal.ZERO;
}
```

- n. Add an import statement for the `java.time.LocalTime` class.

Hint: Right-click anywhere in the source code of the `Drink` class and invoke the "Fix Imports" menu.

```
import java.time.LocalTime;
```

- o. Compile and run your application.



Hints: Click the "Run Project" toolbar button.

Notes

- The actual values in your output may differ depending on the current time of the computer on which you perform this practice.
- Assuming that the current time is not between 16:30 and 17:30, any instance of `Drink` (objects referenced by `p2` and `p6` variables) would produce 0 discount.
- Any instances of `Product` (objects referenced by `p1` and `p4` variables) would produce 0 discount. Actually the `p4` variable is referencing a product with a 0 price anyway.
- Instances of `Food` that are going to expire later than today (objects referenced by `p3` and `p7` variables) would produce 0 discount.

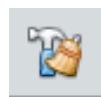
8. Fix logic of the `applyRating` operation to account for the existence of `Product` subclasses.

Notes

- There is a problem with the way in which the `applyRating` method works. It is designed to create a replica of the `Product` object with a different rating. However, it does not account for the existence of subclasses. If you invoke the `applyRating` method on an instance of `Food` or `Drink`, it would not return an instance of the relevant subclass, but an instance of `Product` instead, because at the moment it is hard-coded to do just that.
- To fix this issue, the `Product` class needs to treat this operation as something to be implemented by its subtypes. This could be achieved by making this operation abstract. This in turn would require the `Product` class itself to be marked as abstract to ensure that you only construct instances of its subtypes (`Food` and `Drink`), which would have to provide subtype specific implementations of the `applyRating` method.

- Open the `Product` class editor.
- Add the `abstract` keyword to the `Product` class definition.
- Recompile the `ProductManagement` project.

Hint: Use Run->Clean and Build Project menu or a toolbar button.



Notes

- The classes `Product` and `Shop` now fail to compile because they attempt to instantiate `Product` directly and that is no longer possible, because `Product` is now an abstract class.
- To fix this issue, you should replace instantiation of `Product` with instantiation of `Food` or `Drink` objects.

- d. Open the `Shop` class editor.
- e. Replace the initialization of the `p1` variable with the construction of a new `Drink` object using existing values and adding a three star rating to the constructor:

```
Product p1 = new Drink(101, "Tea", BigDecimal.valueOf(1.99),
                        Rating.THREE_STAR);
```

- f. Replace the initialization of the `p4` variable with the construction of a new `Food` object, setting id as 105, name as `Cookie`, price as 3.99, rating as two star, and best before date as today:

```
Product p4 = new Food(105, "Cookie", BigDecimal.valueOf(3.99),
                        Rating.TWO_STAR, LocalDate.now());
```

- g. Open the `Product` class editor.

- h. Mark the `applyRating` method as `abstract` and remove its body.

```
public abstract Product applyRating(Rating newRating);
```

- i. Recompile the `ProductManagement` project

Hint: Use Run->Clean and Build Project menu or a toolbar button

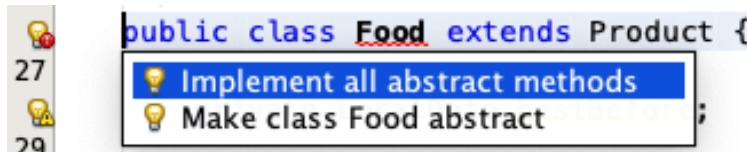


Note: Classes `Product` and `Shop` are now compiling successfully. However, classes `Food` and `Drink` are not. This is because any class that is not abstract and inherits abstract operations must override them.

- j. Open the `Food` class editor.

- k. Implement abstract methods in a `Food` class.

Hint: Click the right side of the line of code that defines `Food` class to invoke the "Implement all abstract method" pop-up menu.



```
@Override
public Product applyRating(Rating newRating) {
    throw new UnsupportedOperationException("Not supported yet.");
}
```

Note: It is possible to modify the return type of this method to actually return `Food` rather than `Product`, since this is the intention. However, generally it is recommended that method parameters and return types should use generic, parent types and only use specific subtypes if this is a necessary design restriction.

- I. Replace the `applyDiscount` operation logic with a code that creates a replica of `Food` object with modified rating based on the available parameter value.

Hints

- Create and return a new `Food` object.
- Use the `getter` method to retrieve id, name, and price values as they are private in the `Product` and, therefore, not visible to the `Food` object directly.
- `newRating` is an argument of the current method, so it can be accessed directly.
- The `bestBefore` field is declared in the `Food` class itself, so it can be accessed directly.

```
@Override
public Product applyRating(Rating newRating) {
    return new Food(getId(), getName(), getPrice(),
                    newRating, bestBefore);
}
```

- m. Open the `Drink` class editor.
n. Implement the abstract methods in a `Drink` class.

Hint: Click on the right side of the line of code that defines `Drink` class to invoke the "Implement all abstract method" pop-up menu.

```
@Override
public Product applyRating(Rating newRating) {
    throw new UnsupportedOperationException("Not supported yet.");
}
```

- o. Replace the `applyRating` operation logic with a code that creates a replica of the `Drink` object with modified rating based on the available parameter value.

Hints

- Create and return a new `Drink` object.
- Use the `getter` methods to retrieve id, name, and price values because they are private in the `Product` class and, therefore, not visible within the `Drink` class directly.

```
@Override
public Product applyRating(Rating newRating) {
    return new Drink(getId(), getName(), getPrice(),
                     newRating);
}
```

- p. Recompile the `ProductManagement` project.

Hint: Use Run->Clean and Build Project menu or a toolbar button



Note: All classes are now compiling successfully.

9. Explore the polymorphic behavior of the `applyDiscount` operation.
- Open the `Shop` class editor.
 - Create a new line of code immediately after the declaration of the `p5` variable.
 - On this new line, declare a new `Product` variable `p8` and initialize it to reference a replica of the product referenced by the `p4` variable, created as a result of a new five star rating being applied.
- ```
Product p8 = p4.applyRating(Rating.FIVE_STAR);
```
- Create a new line of code immediately after this and declare a new `Product` variable `p9` and initialize it to reference a replica of the product referenced by the `p1` variable, created as a result of a new two-star rating being applied.
- ```
Product p9 = p1.applyRating(Rating.TWO_STAR);
```
- Add two lines of code that print out objects referenced by the `p8` and `p9` variables immediately after the line that prints object referenced by the `p5` variable.
- ```
System.out.println(p8);
System.out.println(p9);
```
- Compile and run your application.
- Hints:** Click the "Run Project" toolbar button.
- 
- Note:** Because of polymorphism, the correct version of the `applyRating` method is automatically invoked.

10. Explore the non-polymorphic behavior of the `getBestBefore` operation.

#### Notes

- The `getBestBefore` operation is not declared at the `Product` class level and only exists in a `Food` class.
  - To invoke this operation from the `Shop` class, you have to either declare a variable as type of `Food` rather than `Product` (which means you would not be able to assign an instance of `Drink` to such a variable) or check if a variable is on a `Product` type, which is actually referencing an instance of `Food`. If that is the case, such a reference can be casted to `Food` type, so the invocation of the `getBestBefore` method can proceed.
- Open the `Shop` class editor.
  - Just before the end of the `main` method body, add code that checks if a variable `p3` is indeed referencing a `Food` object. Then cast this reference to `Food` type, invoke `getBestBefore` operation, and print the result to the console:

```
if (p3 instanceof Food) {
 System.out.println(((Food)p3).getBestBefore());
}
```

#### Notes:

- Performing such checks and castings every time you access operations that are only available in a specific subtype is necessary, but not convenient.

- However, you can make this operation polymorphic, by ensuring that it is declared in the Product class, overriding it in subclasses, and accessing it without any type-checks and castings.
  - There are two ways of implementing such a design:
    - Define this operation in a parent class as abstract and thus force it to be overridden in every product subclass. This is a way in which the `applyRating` operation is designed.
    - Define this operation in a parent class with some default behavior and only override it in some child classes as required. Your next step of the practice will be to design the `getBestBefore` operation in this way.
- c. Open `Food` class editor.
  - d. Select the `getBestBefore` operation together with its documentation comments section and copy it (CTRL+C).
  - e. Open the `Product` class editor.
  - f. Immediately after the the `applyRating` operation, paste the (CTRL+V) code copied from the `Food` class.
  - g. Click "OK" in the pop-up dialog box that offers you to add the required import of the `java.time.LocalDate` class.

```
/**
 * Get the value of bestBefore
 *
 * @return the value of bestBefore
 */
public LocalDate getBestBefore() {
 return bestBefore;
}
```

### Notes

- This operation does not compile at the moment because its logic is referencing a variable `bestBefore` that does not exist in a Product class.
  - Your next step of the practice will be to replace this line of code with a statement that returns a default value for the best before date.
- h. Modify return statement expression inside the `getBestBefore` method in a Product class, so it will always return the current date.

```
public LocalDate getBestBefore() {
 return LocalDate.now();
}
```

- i. Modify documentation comments to reflect the change of logic of this operation.

```

/**
 * Assumes that the best before date is today
 *
 * @return the current date
 */
public LocalDate getBestBefore() {
 return LocalDate.now();
}

```

### Notes

- Now the `Shop` class can use the `getBestBefore` operation on any `Product` without explicit type-checks or castings.
- `Food` class overrides this operation. You may even add `@Override` annotation (optional).
- `Drink` class does not override this operation, so for instances of `Drink` default behavior defined by the `Product` class is used.

- j. Open the `Shop` class editor.
- k. Modify code that prints best before date for product referenced by the `p3` variable, without performing type-check and typecasting.

```
System.out.println(p3.getBestBefore());
```

- l. Add another line of code that prints best before date for the product is referenced by the `p1` variable.

```
System.out.println(p1.getBestBefore());
```

**Note:** Using the `getBestBefore` operation on an instance type of `Drink` was not possible before you have promoted this operation to be a part of the `Product` class, as it was only available for instances of the `Food` class.

- m. Compile and run your application.

**Hints:** Click the "Run Project" toolbar button.



**Note:** Apparently the `toString` operation of the `Product` class still does not take an advantage of this operation. This is why only the best before date is printed only for instances of the `Food` class.

- o. Open the `Product` class editor.
- p. Append `bestBefore` date value at the end of the expression that returns `String` from the `toString` method:

```
@Override
public String toString() {
 return id+", "+name+", "+price+", "+getDiscount()+" , "
 +rating.getStars()+" "+getBestBefore();
}
```

**Note:** Due to the polymorphism, the lowest available implementation of the method is automatically invoked, so the `toString` method of the `Product` will use the `getBestBefore` operation defined by the `Food` class for the instances of `Food` and will use the `getBestBefore` operation defined by itself for any other child class instances that do not override this operation (class `Drink`).

- q. Open the `Food` class editor.
- r. Remove the `toString` method from the `Food` class.
- s. Compile and run your application.



**Hints:** Click the "Run Project" toolbar button.

**Note:** The version of the `toString` method available in a product now prints everything you need.

## Practice 6-3: Create Factory Methods

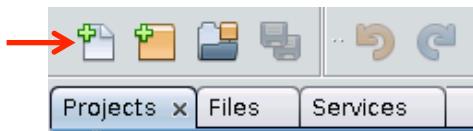
### Overview

In this practice, you create a new class called `ProductManager` and add factory methods to it, which create instances of Product subclasses (Food and Drink). You also modify the `Shop` class to use these factory methods instead of invoking Food and Drink constructors directly.

1. Create the `ProductManager` class.

- a. Create a new Java class.

**Hint:** Use File -> New File menu, or click the "New File" toolbar button.



- b. Select "Java" Category and "Java Class" as the file type.

- c. Click "Next."

- d. Set the following class properties:

- Class Name: `ProductManager`
- Project: `ProductManagement`
- Location: `Source Packages`
- Package: `labs.pm.data`



- e. Click "Finish."

2. Add the factory methods to the `ProductManager` class that create and return `Product` objects using `Food` and `Drink` subtypes.
  - a. Open the `ProductManager` class editor.
  - b. Add a new method to the `ProductManager` class to construct and return an object type of `Food`.

#### Hints

- Make this method `public`.
- Use `Product` as the return type.
- Use `createProduct` as the method name.
- Parameters for this method should match parameters of the `Food` constructor:
  - `id`, `name`, `price`, `rating`, and `best before date`
- Pass these parameters to the `Food` constructor.
- Return a new `Food` instance.

```
public Product createProduct(int id, String name, BigDecimal price,
 Rating rating, LocalDate bestBefore) {
 return new Food(id, name, price, rating, bestBefore);
}
```

- c. Add an import statement for the `java.math.BigDecimal` and `java.time.LocalDate` classes.

**Hint:** Right-click anywhere in the source code of the `ProductManager` class and invoke the "Fix Imports" menu.

```
import java.math.BigDecimal;
import java.time.LocalDate;
```

- d. Create an overloaded version of the `createProduct` method to construct and return an object type of `Food`.

#### Hints

- Copy/paste the existing `createProduct` method.
- Remove the `bestBefore` parameter.
- Pass remaining parameters to the `Drink` constructor.
- Return a new instance `Drink`.

```
public Product createProduct(int id, String name, BigDecimal price,
 Rating rating) {
 return new Drink(id, name, price, rating);
}
```

3. Change constructor access in the Product, Food, and Drink classes and remove unused constructors.

#### Notes

- Constructors of Product, Food, and Drink can be restricted to be used just within the package labs.pm.data.
- The `ProductManager` class is in the same package (`labs.pm.data`), so it can access operations that are restricted to be visible only to members of the same package.
- Classes in other packages (such as `Shop`) can use the public factory methods of the `ProductManager` class to gain access to products.
- `Product` class is abstract and, therefore, cannot be instantiated directly. `Food` and `Drink` classes only use one of the constructors of the `Product` class with `id`, `name`, `price`, and `rating` parameters. This constructor can be restricted to be used only by classes that are in the same package and other constructors of the `Product` class can be removed.
  - a. Open the `Product` class editor.
  - b. Restrict the `Product` class to constructor with `id`, `name`, `price`, and `rating` parameters to be visible only to members of the same package.

**Hint:** Simply remove the keyword `public` from this constructor.

```
Product(int id, String name, BigDecimal price, Rating rating) {
 this.id = id;
 this.name = name;
 this.price = price;
 this.rating = rating;
}
```

#### Notes:

- Default (absent) access modifier indicates current package only visibility of a method or a variable to which it is applied.
- This constructor is used by `Food` and `Drink` classes, but they are in the same package as the `Product` class. They can access this constructor even if it has default access modifier.
- You can impose additional restrictions on the code visibility with the use of Java Modules. This topic is covered in the lesson titled "Modules" of this course.
- c. Remove other constructors from the `Product` class. These are the `no-arg` constructor and constructor with `id`, `name`, and `price` arguments.

- d. Open the `Food` class editor.
- e. Restrict the `Food` class constructor to be visible only to members of the same package.

**Hint:** Simply remove the `public` keyword from this constructor:

```
Food(int id, String name, BigDecimal price,
 Rating rating, LocalDate bestBefore) {
 super(id, name, price, rating);
 this.bestBefore = bestBefore;
}
```

#### Notes

- Default (absent) access modifier indicates current package only visibility of a method or a variable to which it is applied.
  - This constructor is used by `ProductManager` class, which is in the same package as the `Product` class, so it can access this constructor even if it has a default access modifier.
  - This constructor is also used by the `Shop` class; however, we intend to remove these usages and replace them with factory method invocations.
- f. Open the `Drink` class editor.
  - g. Restrict `Drink` class constructor to be visible only to members of the same package.

**Hint:** Simply remove the keyword `public` from this constructor.

```
Drink(int id, String name, BigDecimal price, Rating rating) {
 super(id, name, price, rating);
}
```

#### Notes:

- Default (absent) access modifier indicates current package only visibility of a method or a variable to which it is applied.
  - This constructor is used by the `ProductManager` class, which is in the same package as the `Product` class, so it can access this constructor even if it has a default access modifier.
  - This constructor is also used by the `Shop` class, but we intend to remove these usages and replace them with factory method invocations.
- h. Recompile `ProductManagement` project.

**Hint:** Use Run->Clean and Build Project menu or a toolbar button



**Note:** Class Stop fails to compile because it is still using constructor on `Drink` and `Food` that are now restricted to be only visible to members of the same package.

4. Replace direct constructor invocations to create instances of `Food` or `Drink` from the `Shop` class with calls upon `ProductManager` factory methods.
  - a. Open `Shop` class editor.
  - b. Create a new line of code at the start of the main method.

- c. Declare a new variable called `pm` of type `ProductManager`. Initialize this variable to reference a new instance of `ProductManager`:

```
ProductManager pm = new ProductManager();
```

- d. Add an import statement for the `labs.pm.data.ProductManager` class.

**Hint:** Right-click anywhere in the source code of the `Shop` class and invoke "Fix Imports" menu.

```
import labs.pm.data.ProductManager;
```

- e. Replace all references to `Food` or `Drink` constructors with invocations of the `createProduct` method using the `pm` object reference.

**Hint:** You can simply copy/paste `pm.createProduct` indent of new `Food` or new `Drink` statements.

```
Product p1 = pm.createProduct(101, "Tea",
 BigDecimal.valueOf(1.99), Rating.THREE_STAR);
Product p2 = pm.createProduct(102, "Coffee",
 BigDecimal.valueOf(1.99), Rating.FOUR_STAR);
Product p3 = pm.createProduct(103, "Ice Cream",
 BigDecimal.valueOf(3.99), Rating.FIVE_STAR,
 LocalDate.now().plusDays(2));
Product p4 = pm.createProduct(105, "Cake",
 BigDecimal.valueOf(3.99), Rating.TWO_STAR,
 LocalDate.now());
Product p5 = p3.applyRating(Rating.THREE_STAR);
Product p8 = p4.applyRating(Rating.FIVE_STAR);
Product p9 = p1.applyRating(Rating.TWO_STAR);
System.out.println(p1);
System.out.println(p2);
System.out.println(p3);
System.out.println(p4);
System.out.println(p5);
System.out.println(p8);
System.out.println(p9);
Product p6 = pm.createProduct(104, "Chocolate",
 BigDecimal.valueOf(2.99), Rating.FIVE_STAR);
Product p7 = pm.createProduct(104, "Chocolate",
 BigDecimal.valueOf(2.99), Rating.FIVE_STAR,
 LocalDate.now().plusDays(2));
System.out.println(p6.equals(p7));
System.out.println(p3.getBestBefore());
System.out.println(p1.getBestBefore());
```

**Note:** This is a very important design change. It allows you to create other subclasses of the `Product` class, override any of its operations as necessary, and add extra overloaded versions of the `createProduct` factory method to the `ProductManager` class, without affecting the `Shop` class design in any way. This design helps you to achieve better long-term application flexibility and extensibility, as well as isolate business logic and data management from the front-end part of the application.

- f. Compile and run your application.



**Note:** Observe product details printed on to the console.

5. (Optional) Prevent classes `Food` and `Drink` from being extended .

**Note:** This is a design choice that treats `Food` and `Drink` as leaf classes in your application class hierarchy.

- a. Open the `Food` class editor and make this class a leaf class in the class hierarchy.

**Hint:** Apply the keyword `final` to the class definition.

```
public final class Food extends Product {
```

- b. Open the `Drink` class editor and make this class a leaf class in the class hierarchy.

**Hint:** Apply the keyword `final` to the class definition.

```
public final class Drink extends Product {
```

**Note:** These changes affect only future application extensibility and have no impact on the way in which `Shop` front-end works.

## **Practices for Lesson 7: Interfaces**

## Practices for Lesson 7: Overview

### Overview

In these practices, you design the Rateable interface to represent an ability to associate different classes with Rating objects. At the moment, your `Product` class is designed to handle such association. Changing design to use an interface opens up a possibility to apply this feature to any other class as required. Your next assignment is to design a `Review` class to contain information about rating and review comments. Then you change the `ProductManager` class to enable it to handle product reviews, as well as formatting and printing product review report.

**Tea, Price £1.99, Rating: ★★★★☆, Best Before 23/10/2019**

## Practice 7-1: Design the Rateable Interface

---

### Overview

In this practice, you create a Rateable interface and modify Product class, so it implements this interface. Your existing code (from which you start this practice) already defines required capabilities in the Product class. However, by describing such abilities using an interface, you make this a generic feature applicable not just to the Product class, but to any other class in which you choose to implement the Rateable interface.

### Assumptions

- JDK 11 is installed.
- NetBeans 11 is installed.
- You have completed Practice 6 or start with the solution for Practice 6 version of the application.

### Tasks

1. Prepare the practice environment.

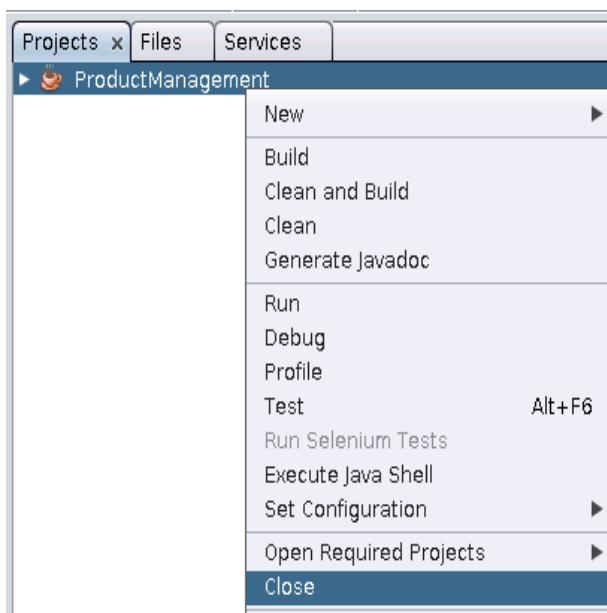
#### Notes

- You may continue to use the same NetBeans project as before, if you have successfully completed the previous practice. In this case, proceed directly to Practice 7-1, step 2.
  - Alternatively, you can open a fresh copy of the NetBeans project, which contains the completed solution for the previous practice.
- a. Open NetBeans (if it is not already running).



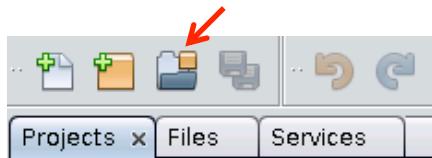
- b. Close the currently open ProductManagement project.

**Hint:** Right-click the ProductManagement project and invoke the "Close" menu.



- c. Open the Solution for Practice 6 version of the ProductManagement project.

**Hint:** Use File -> Open Project menu or click the "Open Project" toolbar button.



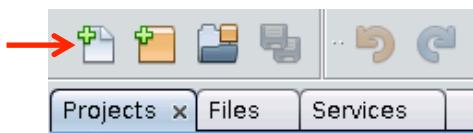
- d. Navigate to the /home/oracle/labs/solutions/practice6 folder and select the ProductManagement project.
- e. Click "Open Project."
2. Create the Rateable interface to implement generic ability to apply rating to various objects.

#### Notes

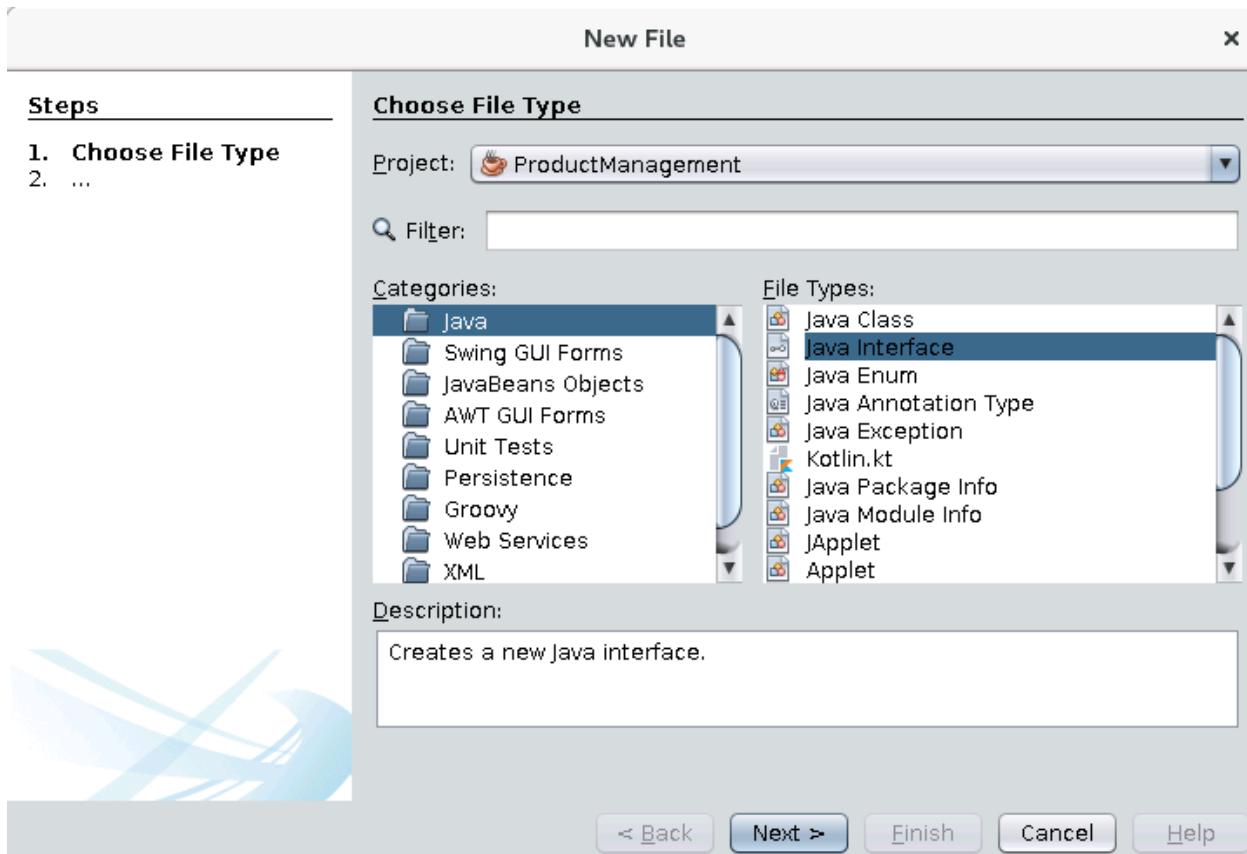
- The Product class is already designed to have an ability to apply rating, provided by the code in its constructor as well as applyRating and getRating methods. However, this capability can be considered as a more widely applicable feature—to be able to apply rating to many more classes and not just to products. For example, you can imagine instances of Shop also being rated.
- The purpose of this part of the practice is to describe an interface that would enable other classes to implement the ability to be rated in a consistent way.

- a. Create a new Java interface.

**Hint:** Use File -> New File menu or click the "New File" toolbar button.



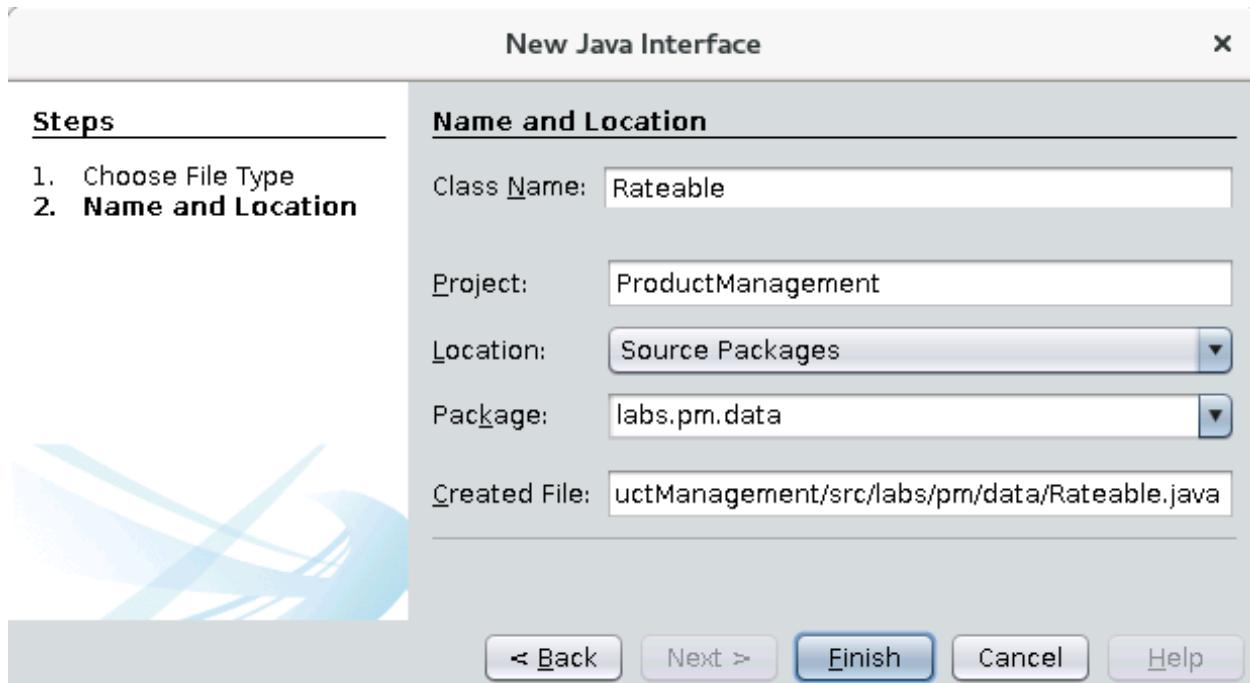
- b. Select "Java" as the Category and "Java Interface" as the file type.



- c. Click "Next."

- d. Set the following class properties:

- Interface Name: Rateable
- Project: ProductManagement
- Location: Source Packages
- Package: labs.pm.data



- e. Click "Finish."
- f. Add generics to the Rateable interface definition, to enable capability to apply rating to other various types.
- ```
package labs.pm.data;
public interface Rateable<T> {
    // more code will be added here
}
```
3. Add constants, abstract, static, and default methods to the Rateable interface.

- a. Add constant to the Rateable interface, which defines a default rating value. Set it to reference the NOT_RATED value of the Rating enum.

Reminders

- Interfaces are allowed to define constants, but not allowed to contain instance variable.
- The naming convention for constants is all uppercase with underscore symbols between words.

```
public static final Rating DEFAULT_RATING = Rating.NOT_RATED;
```

- b. Add an abstract method to the `Rateable` interface, which specifies how ratings should be applied to whatever objects that would implement this interface.

Hints

- This method should be added right after the `DEFAULT_RATING` constant declaration.
- Method should be `public` and `abstract` (although these keywords could be omitted as they are implied in interfaces anyway).
- Method should return generic type.
- Use `applyRating` as a method name.
- Method should accept `Rating` enumeration as an argument.

```
public abstract T applyRating(Rating rating);
```

Notes

- This method signature actually coincides with the definition of an `applyRating` method already provided by the `Product` class.
 - This is not just a simple coincidence as the purpose of this interface is to expand the feature currently defined by the `Product` class to any other classes.
- c. Add a default method to the `Rateable` interface, which returns the `DEFAULT_RATING` value.

Hints

- This method should be added right after the `applyRating` method declaration.
- Method should be `public` and `default` (concrete nonprivate instance methods are not allowed in interfaces).
- Method should return a value type of `Rating` enumeration.
- Use `getRating` as a method name.
- Method should accept no arguments.

```
public default Rating getRating() {
    return DEFAULT_RATING;
}
```

Notes

- With an interface, such methods can only return a predefined value for the `Rating`, because an interface cannot possibly have any instance variables. However, classes that implement this interface may override this method and return any other specific values of `Rating`.
- This method is not abstract, so classes that implement this interface do not have to override it, in which case, the default value would be used.
- This method signature actually coincides with the definition of a `getRating` method already provided by the `Product` class.
- This is not just a simple coincidence as the purpose of this interface is to expand the feature currently defined by the `Product` class to be available to any other class.

- d. Add a static method to the `Rateable` interface, which converts a numeric stars value to the Rating enumeration value.

Hints

- This method should be added right after the `getRating` method.
- Method should be `public` and `static`. (static methods are allowed in interfaces.)
- Method should return a value type of `Rating` enumeration.
- Use `convert` as a method name.
- Method should accept an argument of an `int` type to represent number of `stars` for the rating.

```
public static Rating convert(int stars) {
    // method logic will be added here
}
```

Note: Alternatively this method could have been implemented as a private method, depending on whenever the ability to perform such conversion is needed outside of the `Rateable` interface.

- e. Add implementation logic to the `convert` method to return a `Rating` enum value for the corresponding number of stars.

Hints

- Any Java enumeration provides a method called `values` that returns all corresponding enumeration objects as an array.
- Arrays are covered later in this course. However, for the purposes of this practice, all you need to know about arrays is that they are indexed using `int`, that the index starts from `0`, and that to access array element you need to specify an index value in square brackets: `someArray[index]`
- You need to write an expression that checks if the arguments star value is between `0` and `5` and pick up a corresponding Rating value. Otherwise, you need to return a default rating value.
- Use the ternary operator `? :` to determine the returned value.

```
public static Rating convert(int stars) {
    return (stars>=0&&stars<=5) ? Rating.values() [stars] : DEFAULT_RATING;
}
```

Note: The logic of the statement assumes that `Rating` enum defines its values in a specific order as `NOT_RATED`, `ONE_STAR`, `TWO_STAR`, `THREE_STAR`, `FOUR_STAR` and `FIVE_STAR`, so that they would correspond to array indexes from `0` to `5`.

- f. Add another default method to the `Rateable` interface, which applies rating as an `int` value. This should be an overloaded version of the abstract `applyRating` method.

Hints

- This method should be added right after the `applyRating` method declaration.
- The method should be `public` and `default` (concrete non-private instance methods are not allowed in interfaces).
- The method should return generic type (just like the existing `applyRating` method).

- Use `applyRating` as a method name. (This is an overloaded version of the `applyRating` method.)
- The method should accept an argument of an `int` type to represent number of stars for the rating.

```
public default T applyRating(int stars) {
    // method logic will be added here
}
```

- g. Add implementation logic to the `applyRating` default method to use the `convert` method to supply a `Rating` value that corresponds to the number of `stars`. Make default method invoke abstract method.

```
public default T applyRating(int stars) {
    return applyRating(convert(stars));
}
```

Note: Any class that implements this interface would have to override its abstract method `applyRating`. This default method provides an additional ability to accept the value of rating as an `int` number of stars as well as the actual `Rating` enum value.

- h. (Optionally) Add an annotation that restricts the `Rateable` interface to describe only one abstract method:

```
@FunctionalInterface
public interface Rateable<T> {
    // the rest of this interface code
}
```

Notes:

- Annotation `@FunctionalInterface` prevents interface from compiling if it defines more than one abstract method.
- The `Rateable` interface has only one abstract method, so it should compile without any problems.

4. Implement the `Rateable` interface in the `Product` class.

Notes

- When implementing an interface, you must override its abstract methods. However, since `Product` is an abstract class you don't actually have to override interface abstract methods in the `Product` class itself, but can choose to delegate this to `Product` class children (`Food` and `Drink` classes).
 - You do not have to override default methods if there is no conflict between default methods defined by different interfaces. However, you may choose to override default methods if you wish anyway.
- a. Open the `Product` class editor.

- b. Add implements clause to the Product class definition to implement the Rateable interface. Ensure that this Rateable interface implementation specifies Product as a generic type.

```
public abstract class Product implements Rateable<Product> {  
}
```

Notes

- The Rateable interface was designed to define an ability that Product class has defined anyway. Rateable defines a single abstract method applyRating with the signature that coincidentally is already present in the Product class as an abstract method. You may now remove this abstract method definition from the Product class, because it is essentially a duplicate and thus is no longer required.
- c. Remove the public abstract Product applyRating(Rating newRating); method definition from the Product class.

Notes

- The Product class is abstract, so it does not have to implement all abstract methods of the interfaces it implements. However, its subclasses (Food and Drink) are concrete classes, and thus they must override all abstract method defined by their parent class or any interfaces that they implement directly or via their parent class.
- They already implement the required operation applyDiscount, simply because the signature of the abstract applyDiscount method that has been defined by the Product class happened to be identical to the applyDiscount method defined by the Rateable interface.
- Product class already provides its own implementation of the getRating method, which is no longer considered to be just another method defined by the Product class, but in fact overrides the default getRating method defined by the Rateable interface.
- d. (Optionally) Add an annotation to ensure that a method signature of the getRating method in the Product class does indeed correspond to the method signature of the getRating method described by the Rateable interface:

```
@Override  
public Rating getRating() {  
    return rating;  
}
```

Note: Annotation @Override can be added in front of the getRating method in the Product class to prevent accidental changes to this method signature that put it out of sync with the corresponding interface method.

- e. Recompile ProductManagement project.

Hint: Use Run->Clean and Build Project menu or a toolbar button.



Practice 7-2: Enable Products Review and Rating

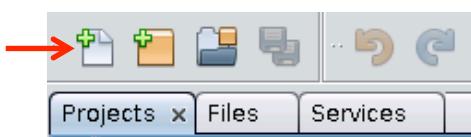
Overview

In this practice, you create a class to represent product reviews. Each review is associated with the rating and comments. Your next task is to provide methods in a ProductManager class to write reviews. At this stage of the practices, ProductManager would only store information on a single Product and a Review for this product. This is going to be changed in the practice for the next lesson, where multiple reviews would be enabled with the use of arrays.

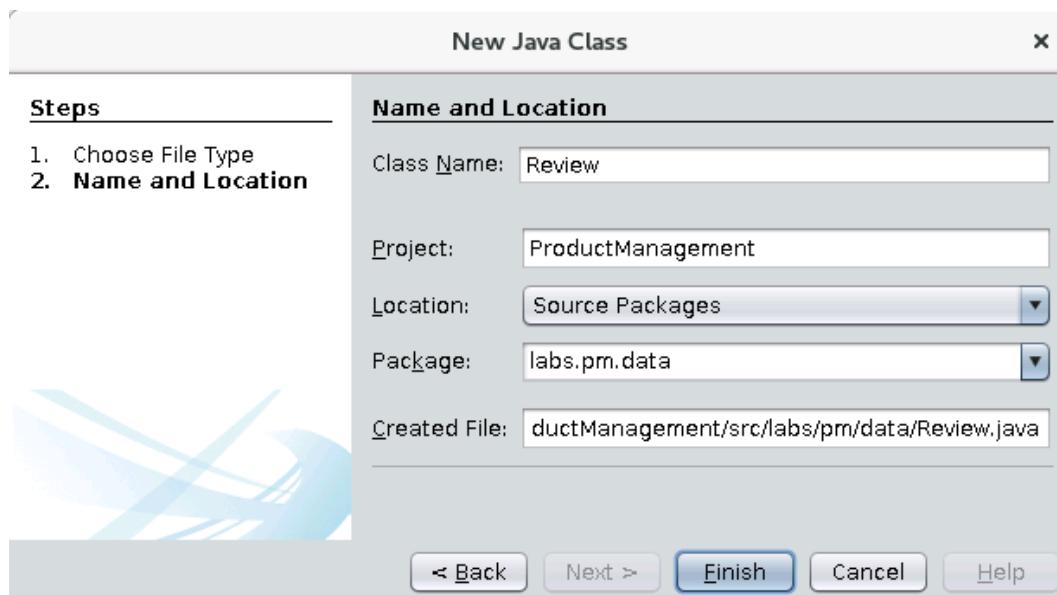
1. Create the class Review.

- a. Create a new Java class.

Hint: Use File -> New File menu or click the "New File" toolbar button.



- b. Select "Java" as the Category and "Java Class" as the file type.
- c. Click "Next."
- d. Set the following class properties:
 - Class Name: Review
 - Project: ProductManagement
 - Location: Source Packages
 - Package: labs.pm.data



- e. Click "Finish."

```
package labs.pm.data;
public class Review {
}
```

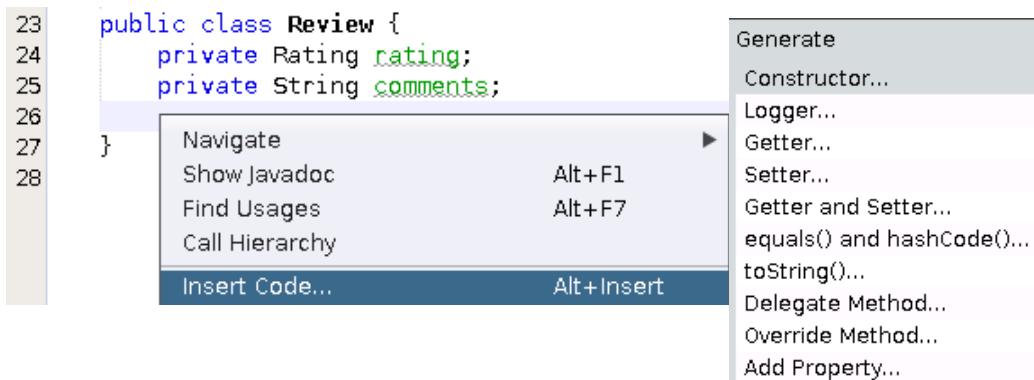
2. Add attributes and operations to the `Review` class to represent rating and comments.
 - a. Add instance variables to the `Review` class. The `rating` attribute should be declared using the `Rating` enumeration as a type and `comments` should be of type `String`. Make these variables private.

```
package labs.pm.data;
public class Review {
    private Rating rating;
    private String comments;
}
```

- b. Add constructor to the `Review` class, which sets all of its attributes.

Hints

- Create a new line inside the `Review` class, just after the `comments` variable declaration.
- Right-click on this empty line and select "Insert Code..." menu and then "Constructor..." menu.



- Select all check boxes.



- Click the "Generate" button.

```

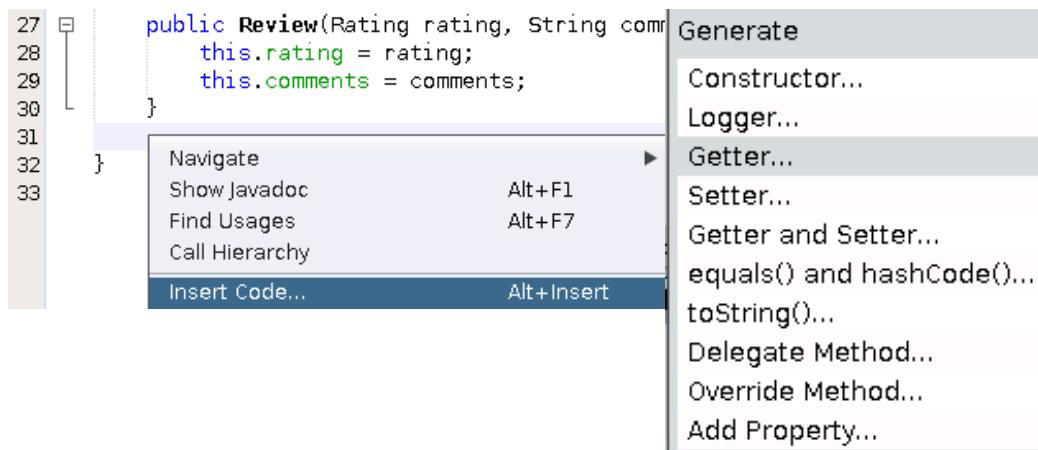
public Review(Rating rating, String comments) {
    this.rating = rating;
    this.comments = comments;
}

```

- c. Add getter methods to the `Review` class, which returns the `rating` and `comments` variables value.

Hints:

- Create a new line inside the `Review` class, just after the constructor.
- Right-click on this empty line and select "Insert Code..." menu and then "Getter..." menu.



- Select the check boxes next to all the variables.



- Click the "Generate" button.

```

public Rating getRating() {
    return rating;
}

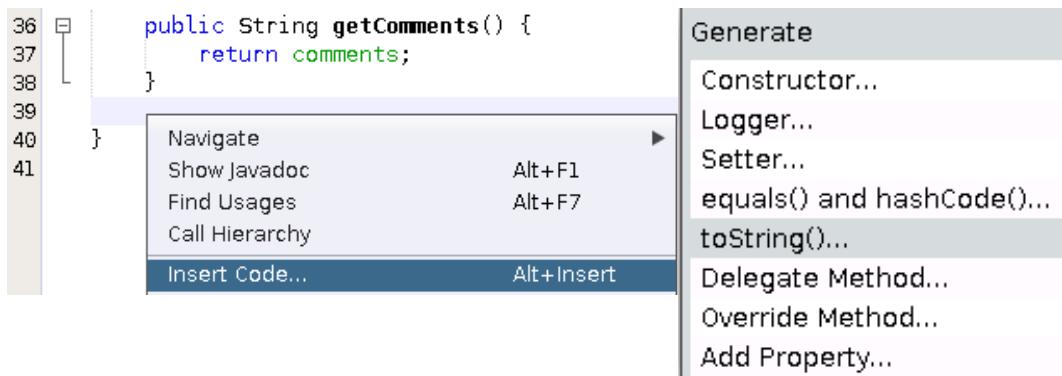
public String getComments() {
    return comments;
}

```

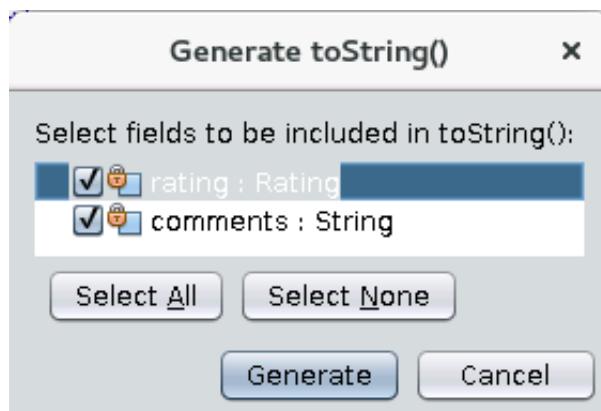
- d. Override the `toString` method within the `Review` class.

Hints:

- Create a new line inside `Review` class, just after the constructor.
- Right-click on this new line and invoke "Insert Code..." and then "toString()..." menu.



- Select the check boxes next to all the variables.



- Click the "Generate" button.

```
@Override
public String toString() {
    return "Review{"+"rating="+rating+", comments="+comments + '}';
}
```

3. Modify the `ProductManager` class to enable it to store information about a product and a review.

Note: At this stage of the practices, `ProductManager` would only store information on a single Product and a Review for this product. This is going to be changed in the practice for the next lesson, where multiple reviews would be enabled with the use of arrays.

- a. Open the `ProductManager` class editor.

- c. Add two instance variables to the `ProductManager` class: attribute `product` of the `Product` class type and `review` of the `Review` class type. Make these variables private.

```
package labs.pm.data;
public class ProductManager {
    private Product product;
    private Review review;
    // the rest of the ProductManager class code
}
```

- d. Modify both versions of the `createProduct` method to set `product` variable before returning it to the invoker.

```
public Product createProduct(int id, String name,
        BigDecimal price, Rating rating, LocalDate bestBefore) {
    product = new Food(id, name, price, rating, bestBefore);
    return product;
}
public Product createProduct(int id, String name,
        BigDecimal price, Rating rating) {
    product = new Drink(id, name, price, rating);
    return product;
}
```

- e. Add a method that creates a new `Review` object, assigns a reference to it to the `review` variable, applies new rating value to the `product`, reassigned the `product` instance variable, and returns it to the invoker.

Hints

- This method should be added right after the last `createProduct` method.
- The method should be `public`.
- The method return type should be `Product`.
- Use `reviewProduct` as a method name.
- The method should accept `Product`, `Rating`, and `comments` arguments.

```
public Product reviewProduct(Product product, Rating rating,
        String comments) {
    review = new Review(rating, comments);
    this.product = product.applyRating(rating);
    return this.product;
}
```

Notes

- `Rating` and `comments` arguments should be used to construct a new `Review` object.
- This method should also apply `Rating` to the `Product` based on a `Rating` value supplied as a part of the review.
- In a later practice, product rating would be calculated as an average rating value of all reviews.

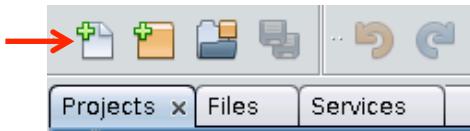
- The reason this method is declared to return Product, rather than just be defined with a void return type, is because of the immutable nature of the Product class design. Method `applyRating` is designed to create a new Product object.
- The Rateable interface provides an overloaded version of the `applyRating` method that can use a simple int value to set number of stars and converts it into the Rating enum object.

Note: A more generic (alternative) design of this application could have used a type Rateable instead of Product for both instance variable and method argument, to enable application to create reviews of any other objects that implement the Rateable interface.

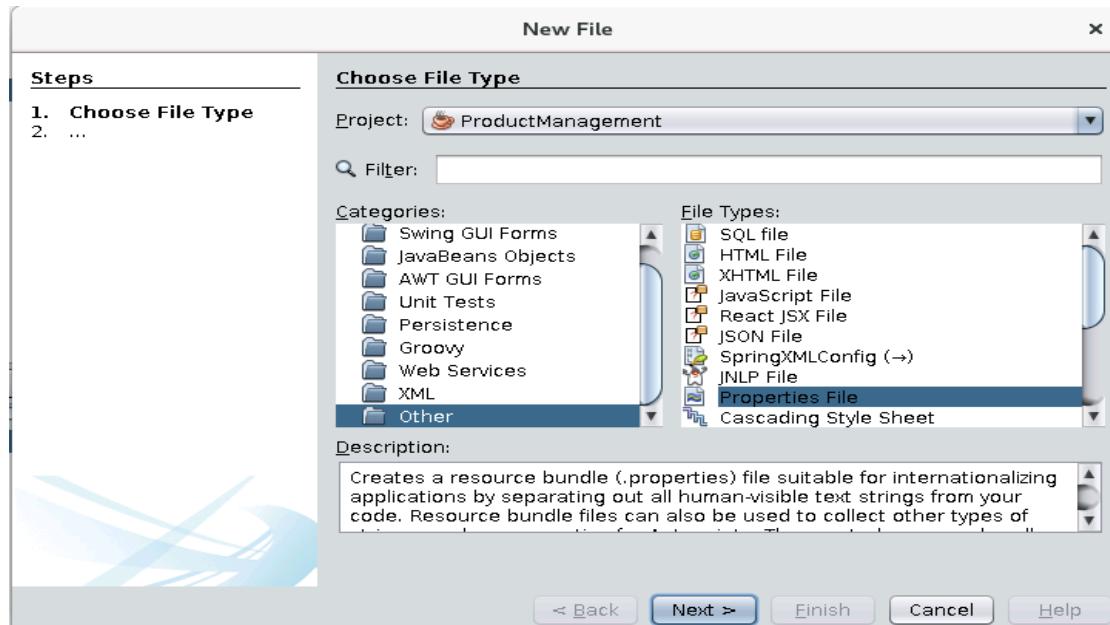
- Create resource bundle to support formatting of product and review report.
 - Create a new resource bundle called `resources`.

Hints:

- Use File -> New File menu or click the "New File" toolbar button.

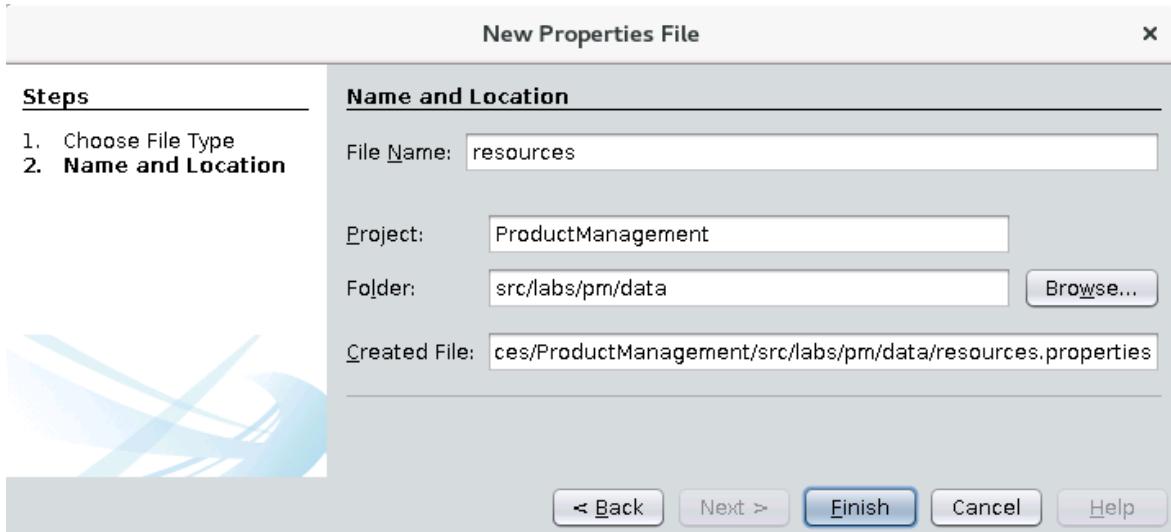


- Select "Other" Category and "Properties File" as the file type.



- Click "Next."

- Set the following class properties:
 - File Name: resources
 - Project: ProductManagement
 - Folder: src/labs/pm/data



- Click "Finish."
- b. Add three properties to the `resources.properties` file to represent format patterns for the product and review and a message to be produced when no reviews are available.

```
product={0}, Price: {1}, Rating: {2}, Best Before: {3}
review=Review: {0}\t{1}
no.reviews=Not reviewed
```

Notes

- Substitution parameters of the product are intended to represent product name, price, average rating, and the best before date.
- Substitution parameters of the review are intended to represent review rating and comments.
- Review rating and comments values are to be separated by the tab \t character.
- You may (optionally) create alternative language representations for these text values, if you want to try using alternative locales in later stages of this practice.

5. Modify the `ProductManager` class to enable it to format and print a report on the product and its reviews (just one review at this point).

Note: All of your classes already override the `toString` method, which is great for writing logs or printing technical debugging information on the console. However, in this practice you need to create code in the `ProductManager` class that would format product and review values in a way that is suitable for the end-user consumption. This means that you should use appropriate localization, resource bundles, as well as number and date formatters. For more information about text formatting and localization, refer to the lesson titled "Text, Date, Time and Numeric Objects."

- Open the `ProductManager` class editor.
- Add four more instance variables to the `ProductManager` class to represent `Locale`, `ResourceBundle`, `DateTimeFormatter`, and `NumberFormat` objects.

Hints

- These variables should be added right after the declaration of the `review` variable.
- Give these attributes names: `locale`, `resources`, `dateFormat`, and `moneyFormat`, respectively.
- Make these variables private:

```
private Locale locale;
private ResourceBundle resources;
private DateTimeFormatter dateFormat;
private NumberFormat moneyFormat;
```

- Add an import statement for the `java.util.Locale` `java.util.ResourceBundle` `java.time.format.DateTimeFormatter` and `java.text.NumberFormat` classes.

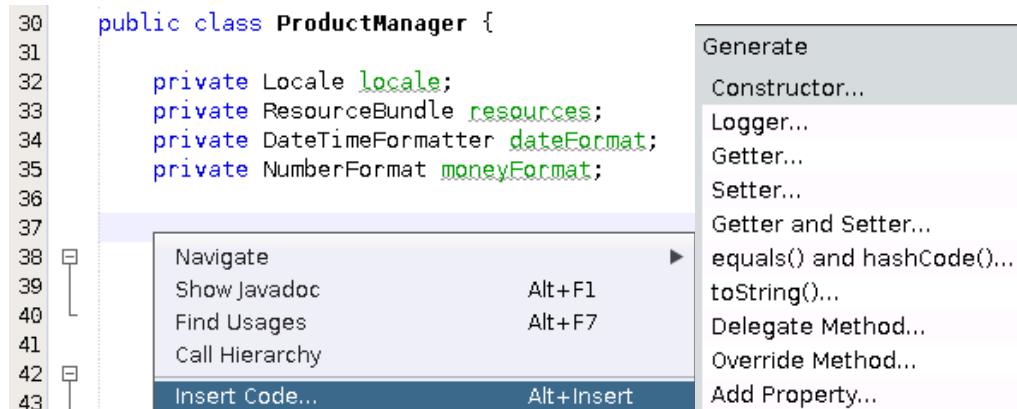
Hint: Right-click anywhere in the source code of the `ProductManager` class and invoke the "Fix Imports" menu.

```
import java.util.Locale;
import java.util.ResourceBundle;
import java.time.format.DateTimeFormatter;
import java.text.NumberFormat;
```

- Add constructor to the `ProductManager` class, which accepts `Locale` as an argument.

Hints:

- Create a new line inside the `ProductManager` class, just before the first `createProduct` method.
- Right-click on this empty line and select "Insert Code..." menu and then "Constructor..." menu.



- Select only the check box next to the locale field.



- Click the "Generate" button.

```
public ProductManager(Locale locale) {
    this.locale = locale;
}
```

- Add code to the `ProductManager` constructor to load resource bundle and initialize date and money format instance variables.

Hints

- This code should be added right after the initialization of the `locale` variable, inside the `ProductManager` constructor.
- Use the `getBundle` method of the `ResourceBundle` class, which accepts bundle name and locale as arguments.
- The name of the resource bundle should be `resources`, and it is located in the `labs.pm.data` package.
- Use the `ofLocalizedDate` method of the `DateTimeFormatter` class and set `SHORT` as the `FormatStyle` and associate it with `locale` using the `localizedBy` method.
- Use the `getCurrencyInstance` method of the `NumberFormat` class that accepts `locale` as an argument.

```
public ProductManager(Locale locale) {
    this.locale = locale;
    resources = ResourceBundle.getBundle("labs.pm.data.resources", locale);
    dateFormat = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT)
        .localizedBy(locale);
    moneyFormat = NumberFormat.getCurrencyInstance(locale);
}
```

- f. Add an import statement for the `java.time.format.FormatStyle` enum.
- Hint:** Right-click anywhere in the source code of the `ProductManager` class and invoke the "Fix Imports" menu.
- ```
import java.time.format.FormatStyle;
```
- g. Add a method that creates, prepares, and prints a report on a product and its review.

#### Hints

- This method should be added right after the `reviewProduct` method.
- Method should be `public`.
- Method return type should be `void`.
- Use `printProductReport` as a method name.
- Method should accept no arguments.

```
public void printProductReport() {
 // formatting and printing logic will be added here
}
```

- h. Inside the `printProductReport` method, create a new `StringBuilder` object called `txt` and initialize it to reference a new empty `StringBuilder` object.
- ```
StringBuilder txt = new StringBuilder();
```
- i. In the next line of code, append a formatted message to the `txt` object using the `format` method of the `MessageFormat` class:
- The first parameter of the `format` method is the text pattern into which values should be substituted.
 - Get the `product` message pattern from the resource bundle using the `getString` method.
 - Other parameters of the `format` method are values of product name, price, stars rating, and best before date, which you need to substitute into this pattern.
 - Format price using `moneyFormat` formatter object and best before date using `dateFormat` object.

```
txt.append(MessageFormat.format(resources.getString("product"),
    product.getName(),
    moneyFormat.format(product.getPrice()),
    product.getRating().getStars(),
    dateFormat.format(product.getBestBefore())));
```

- j. Add an import statement for the `java.text.MessageFormat` class.
- Hint:** Right-click anywhere in the source code of the `ProductManager` class and invoke "Fix Imports" menu.
- ```
import java.text.MessageFormat;
```
- k. Continue writing formatting logic in the next line of code inside the `printProductReport` method and append new line to the `txt` object.
- ```
txt.append('\n');
```

- i. Add an `if` statement to check if review object is not null.

```
if (review != null) {
    // format review text here
}
```

- m. Inside the body of the `if` block, add a line of code that appends a formatted message to the `txt` object using the `format` method of the `MessageFormat` class:

- The first parameter of the `format` method is the text pattern into which values should be substituted.
- Get the review message pattern from the resource bundle using `getString` method.
- Other parameters of the `format` method are values of review stars rating and comments, which you need to substitute into this pattern.

```
if (review != null) {
    txt.append(MessageFormat.format(resources.getString("review"),
        review.getRating().getStars(),
        review.getComments()));
}
```

- n. Add an `else` clause, which prints a message to indicate that product was not reviewed. The message text should be picked up from the resource bundle by using the "no.reviews" key.

```
else {
    txt.append(resources.getString("no.reviews"));
}
```

- o. Append another new line to the `txt` object.

```
txt.append('\n');
```

- p. Print the `txt` object to the console

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

- q. Recompile the ProductManagement project.

Hint: Use Run->Clean and Build Project menu or a toolbar button.



Notes

- All classes except Shop should compile successfully.
- The `Shop` class is still using the no-arg constructor on `ProductManager`, which is no longer available. This will be fixed in the next stage of the practice.

Practice 7-3: Test the Product Review Functionality

Overview

In this practice, you modify logic of the `main` method of the `Shop` class to test new functionalities on writing product reviews provided by the `ProductManager` class.

1. Modify logic of the `main` method of the `Shop` class to change the way you initialize `ProductManager` object and create `Product` objects.
 - a. Open the `Shop` class editor.
 - b. Add the `Locale` parameter to the constructor of the `ProductManager` object. Use British English locale.

```
ProductManager pm = new ProductManager(Locale.UK);
```

 - c. Add an import statement for the `java.util.Locale` class.
- Hint:** Right-click anywhere in the source code of the `Shop` class and invoke the "Fix Imports" menu.
- ```
import java.util.Locale;
```
- d. Change initialization of the first product to use the `NOT_RATED` value for the Rating:
- ```
Product p1 = pm.createProduct(101, "Tea",
                               BigDecimal.valueOf(1.99), Rating.NOT_RATED);
```
- e. Inside the `main` method of a `Shop` class, place comments on all lines of code after the first product initialization.

Hints

- Select all lines of code from the line that initialized product `p2`, until the end of the `main` method.
- Press the `CTRL+/` keys to place comments on these lines of code.
- Alternatively you can simply type `//` in front of each of these lines or type `/*` before and `*/` after these lines of code.

```

37  public static void main(String[] args) {
38      ProductManager pm = new ProductManager(Locale.UK);
39      Product p1 = pm.createProduct(101, "Tea", BigDecimal.valueOf(1.99), Rating.NOT_RATED);
40      pm.printProductReport();
41      // Product p2 = pm.createProduct(102, "Coffee", BigDecimal.valueOf(1.99), Rating.FOUR_STAR);
42      // Product p3 = pm.createProduct(103, "Cake", BigDecimal.valueOf(3.99), Rating.FIVE_STAR);
43      // Product p4 = pm.createProduct(105, "Cookie", BigDecimal.valueOf(3.99), Rating.TWO_STAR);
44      // Product p5 = p3.applyRating(Rating.THREE_STAR);
45      // Product p8 = p4.applyRating(Rating.FIVE_STAR);
46      // Product p9 = p1.applyRating(Rating.TWO_STAR);
47      // System.out.println(p1);
48      // System.out.println(p2);
49      // System.out.println(p3);
50      // System.out.println(p4);
51      // System.out.println(p5);
52      // System.out.println(p8);
53      // System.out.println(p9);
54      // Product p6 = pm.createProduct(104, "Chocolate", BigDecimal.valueOf(2.99), Rating.FIVE_STAR);
55      // Product p7 = pm.createProduct(104, "Chocolate", BigDecimal.valueOf(2.99), Rating.FIVE_STAR);
56      // System.out.println(p6.equals(p7));
57      // System.out.println(p3.getBestBefore());
58      // System.out.println(p1.getBestBefore());
59  }

```

- f. Invoke the `printProductReport` method. This code should be placed on a new line of code immediately after the initialization of the `p1` variable.

```
pm.printProductReport();
```

- g. Compile and run your application.



Hint: Click the "Run Project" toolbar button.

```
Tea, Price: £1.99, Rating: *****, Best Before: 23/10/2019
Not reviewed
```

```
BUILD SUCCESSFUL (total time: 4 seconds)
```

Note: Observe the product details and a not reviewed message printed on to the console.

2. Add a product review to the first product created in the `main` method of the `Shop` class.
 - a. Invoke the `reviewProduct` method, passing first product, four star rating, and any comments text you like as method parameters. The result returned by the `reviewProduct` method can be used to reassign the `p1` variable. This code should be placed on a new line of code immediately after the invocation of the `printProductReport` method.

```
p1 = pm.reviewProduct(p1, Rating.FOUR_STAR, "Nice hot cup of tea");
```

- b. On the next line of code, invoke the `printProductReport` method:

```
pm.printProductReport();
```

- c. Compile and run your application.



Hint: Click the "Run Project" toolbar button.

```
Tea, Price: £1.99, Rating: *****, Best Before: 23/10/2019
Not reviewed
```

```
Tea, Price: £1.99, Rating: *****, Best Before: 23/10/2019
Review: ***** Nice hot cup of tea
```

```
BUILD SUCCESSFUL (total time: 3 seconds)
```

Note: Observe the product details and a product review printed on to the console.

Practices for Lesson 8:

Arrays and Loops

Practices for Lesson 8: Overview

Overview

In these practices, you create an array of review objects inside the `ProductManager` class instead of using a single review. You also change the way in which the Product rating is applied, to compute such a rating based on the average value of ratings in all reviews.

Process array of reviews
Calculate product rating

Review[★★★★☆
★☆☆☆☆☆
★★★☆☆☆
★☆☆☆☆☆
★★☆☆☆☆
★★★★★]



Practice 8-1: Allow Multiple Reviews for a Product

Overview

In this practice, you modify the `ProductManager` class design to allow multiple reviews to be stored for a product. You also add code to calculate Product rating based on an average score of individual ratings from reviews. Test this new functionality by creating multiple reviews with different ratings in the `Shop` class.

Assumptions

- JDK 11 is installed.
- NetBeans 11 is installed.
- You have completed Practice 7 or start with the solution for Practice 7 version of the application.

Tasks

1. Prepare the practice environment.

Notes

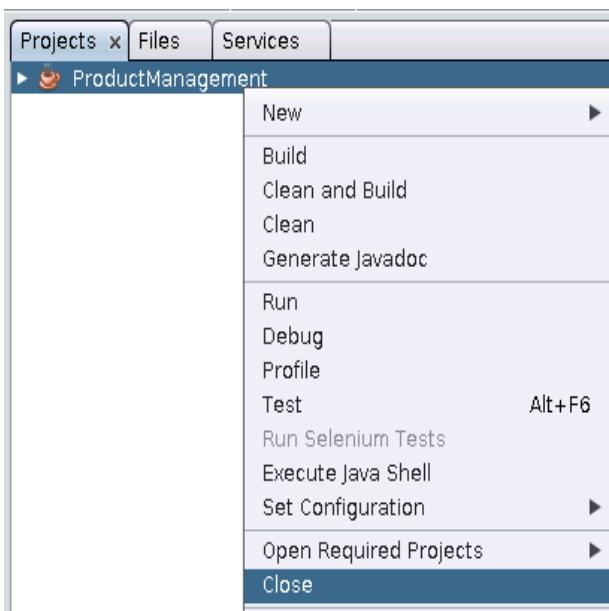
- You may continue to use the same NetBeans project as before, if you have successfully completed the previous practice. In this case, proceed directly to Practice 8-1, step 2.
- Alternatively, you can open a fresh copy of the NetBeans project that contains the completed solution for the previous practice.

- a. Open NetBeans (if it is not already running).



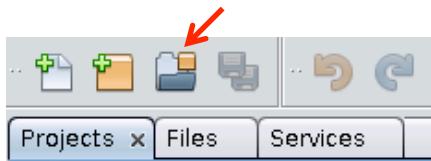
- b. Close the currently opened ProductManagement project.

Hint: Right-click the ProductManagement project and invoke the "Close" menu.



- c. Open the Solution for Practice 7 version of the ProductManagement project.

Hint: Use File -> Open Project menu or click the "Open Project" toolbar button.



- d. Navigate to the `/home/oracle/labs/solutions/practice7` folder and select ProductManagement project.
- e. Click "Open Project."
2. Modify the `ProductManager` class to store `Review` objects in the array.
- Open the `ProductManager` class editor.
 - Modify declaration and initialization of the `review` variable.
 - Change its type to the array of `Review` objects.
 - Change its name to `reviews`.
 - Add the initialization that references a new array of `Review` objects that can hold up to five values.

```
private Review[] reviews = new Review[5];
```

3. Modify the logic of the `reviewProduct` method of the `ProductManager` class to place the new `Review` object into the `reviews` array. This algorithm should replace the line of code that assigns the `review` variable to reference the new instance of the `Review` object.
- Remove the line of code that assigns new `Review` instance to a `review` variable in the `reviewProduct` method.
 - Add an `if` statement that determines if the array of reviews is full and re-create `reviews` array by copying the existing content into the array of a larger size.

Hints

- Check if the last element in the array is not null.
- Use the `Arrays.copyOf` method.
- Increase the size of the array by five elements:

```
if (reviews[reviews.length-1] != null) {
    reviews = Arrays.copyOf(reviews, reviews.length+5);
}
```

- c. Add a new line of code immediately after the end of the `if` block and declare two `int` variables called `sum` and `i`, both set to 0. The purpose of these variables is to compute the total of stars in all ratings and to count a number of ratings so that the average rating value can be determined.

```
int sum = 0, i = 0;
```

- e. Add a new line of code and declare a new boolean variable called `reviewed` and set it to `false`. The purpose of this variable is to indicate if the review was successfully added to the array of reviews and use it as a condition to terminate iteration through this array.

```
boolean reviewed = false;
```

- f. Next create a `while` loop, which should continue iterating until it reaches the end of the array and that the review has not yet been added to the array.

```
while (i < reviews.length && !reviewed) {
    // review array handling logic will be added here
}
```

- g. Inside this `while` loop, add an `if` statement that checks if an element in the array is `null`. In which case create a new `Review` object passing `rating` and `comments` parameters to the constructor, assign this review to the current element in the `reviews` array, and set the `reviewed` variable to `true`, to indicate that no more iterations are required.

```
if (reviews[i] == null) {
    reviews[i] = new Review(rating, comments);
    reviewed = true;
}
```

- h. After the end of this `if` block, inside the `while` loop, add a calculation of the total sum value of rating stars.

Hints:

- Add `int` stars value of Rating to the `sum` variable.
- Invoke the `getRating` method on a current `reviews` array object.
- To find the `int` value of the `Rating` enum, use the `ordinal` method that is available for any enumeration.

```
sum += reviews[i].getRating().ordinal();
```

- i. After this calculation, inside the `while` loop increment the value of `i` variable so that the loop would progress to the next iteration.

```
i++;
```

- j. After the end of the `while` loop, modify a line of code that applies rating to a product.

Hints:

- You should replace a simple assignment of rating with a calculation of the rating value based on the total number of stars (held in a variable `sum`) and a total number of reviews (held in a variable `i`).
- When dividing `sum` by `i`, remember that they are both of an `int` type, and the result you want to achieve should be `float`, so you will have to cast any one of these variables to `float` type.
- You then need to round the result back to the `int` value using `Math.round` method.
- Use the static `convert` method of a `Rateable` interface to convert the `int` value into a `Rating` enum object reference.

```

this.product =
product.applyRating(Rateable.convert(Math.round((float)sum/i)));

```

Note: This is the overall result that you should have achieved modifying the logic of the `reviewProduct` method:

```

public Product reviewProduct(Product product,
                               Rating rating, String comments) {
    if (reviews[reviews.length-1] != null) {
        reviews = Arrays.copyOf(reviews, reviews.length+5);
    }
    int sum = 0, i = 0;
    boolean reviewed = false;
    while (i < reviews.length && !reviewed) {
        if (reviews[i] == null) {
            reviews[i] = new Review(rating, comments);
            reviewed = true;
        }
        sum += reviews[i].getRating().ordinal();
        i++;
    }
    this.product =
product.applyRating(Rateable.convert(Math.round((float)sum/i)));
    return this.product;
}

```

4. Modify logic of the `printProductReport` method of the `ProductManager` class to iterate through the arrays of reviews.

- a. Add a `forEach` loop statement that iterates through the `reviews` array.

- The loop should begin just before the `if` statement that checks if review is not null.
- The loop body should end just before the line of code that prints the `txt` object.

```

for (Review review : reviews) {
    // existing logic that appends review information to the txt object
}

```

- b. Modify the `if` condition so it will check if review is null and break out of the loop, if this is the case.

```

for (Review review : reviews) {
    if (review == null) {
        break;
    }
    // existing logic that appends review information to the txt object
}

```

- c. Append no.reviews message to the txt object, if there are no reviews available at all in the array.
- Create an `if` statement to check if there were no reviews available in the array.
 - Insert this statement immediately after the end of the `forEach` loop body, just before the printout of the `txt` object.
 - Cut and paste the line of code that appends not reviewed message inside this `if` statement body.
 - Append a new line '\n' to the `txt` object.

```
if (reviews[0] == null) {
    txt.append(resources.getString("no.reviews"));
    txt.append('\n');
}
```

- d. Remove the entire `else` clause from side of the loop.

Note: This is the overall result that you should have achieved modifying the logic of the `printProductReport` method:

```
public void printProductReport() {
    StringBuilder txt = new StringBuilder();
    txt.append(MessageFormat.format(resources.getString("product"),
        product.getName(),
        moneyFormat.format(product.getPrice()),
        product.getRating().getStars(),
        dateFormat.format(product.getBestBefore())));
    txt.append('\n');
    for (Review review : reviews) {
        if (review == null) {
            break;
        }
        txt.append(MessageFormat.format(resources.getString("review"),
            review.getRating().getStars(),
            review.getComments()));
        txt.append('\n');
    }
    if (reviews[0] == null) {
        txt.append(resources.getString("no.reviews"));
        txt.append('\n');
    }
    System.out.println(txt);
}
```

- e. Recompile ProductManagement project.

Hint: Use Run->Clean and Build Project menu or a toolbar button.



5. Modify method main of the Shop class to test multiple review capabilities.
- Open the Shop class editor.
 - Add five more reviews to the p1 product object. Use different ratings and comments.
 - This logic should be added immediately after the first review application, just before the final printing of the product report.

```
ProductManager pm = new ProductManager(Locale.UK);
Product p1 = pm.createProduct(101, "Tea",
                               BigDecimal.valueOf(1.99), Rating.NOT_RATED);
pm.printProductReport();
p1 = pm.reviewProduct(p1, Rating.FOUR_STAR, "Nice hot cup of tea");
p1 = pm.reviewProduct(p1, Rating.TWO_STAR, "Rather weak tea");
p1 = pm.reviewProduct(p1, Rating.FOUR_STAR, "Fine tea");
p1 = pm.reviewProduct(p1, Rating.FOUR_STAR, "Good tea");
p1 = pm.reviewProduct(p1, Rating.FIVE_STAR, "Perfect tea");
p1 = pm.reviewProduct(p1, Rating.THREE_STAR, "Just add some lemon");
pm.printProductReport();
```

- c. Compile and run your application.

Hint: Click the "Run Project" toolbar button.



Note: Observe product details and product reviews printed on to the console. You may add intermediate printings of the product report if you want to observe changes in the average rating calculation.

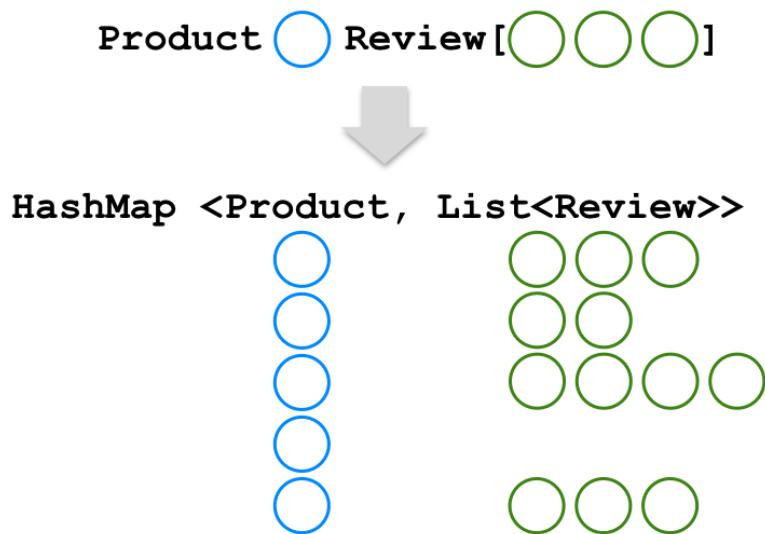
Note: At this stage of the course, the ProductManager class is only storing information on a single product. You could have designed it to store an array of Product objects instead. In which case you can move the array of Review objects inside the Product, together with all the code that manages these reviews. This would allow you to manage multiple reviews in the context of specific Product instance and multiple products in a context of a ProductManager instance. However, in the practice for the lesson titled “Collections,” a different design approach will be elected—using Java Collections API instead of arrays.

Practices for Lesson 9: Collections

Practices for Lesson 9: Overview

Overview

In these practices, you replace a single Product object reference and an array of Review objects in the ProductManager class with a HashMap that will store a Set of Product objects and a List of Review objects per each product. Product object will be used as map key to identify individual map entries and a List of Review objects will be a map entry value. Provide a sorting mechanism for reviews and a searching mechanism for products.



Practice 9-1: Organize Products and Reviews into a HashMap

Overview

In this practice, you modify ProductManager class design to allow it to store and manage multiple products and multiple reviews per product.

Assumptions

- JDK 11 is installed.
- NetBeans 11 is installed.
- You have completed Practice 8 or start with the solution for Practice 8 version of the application.

Tasks

1. Prepare the practice environment.

Notes:

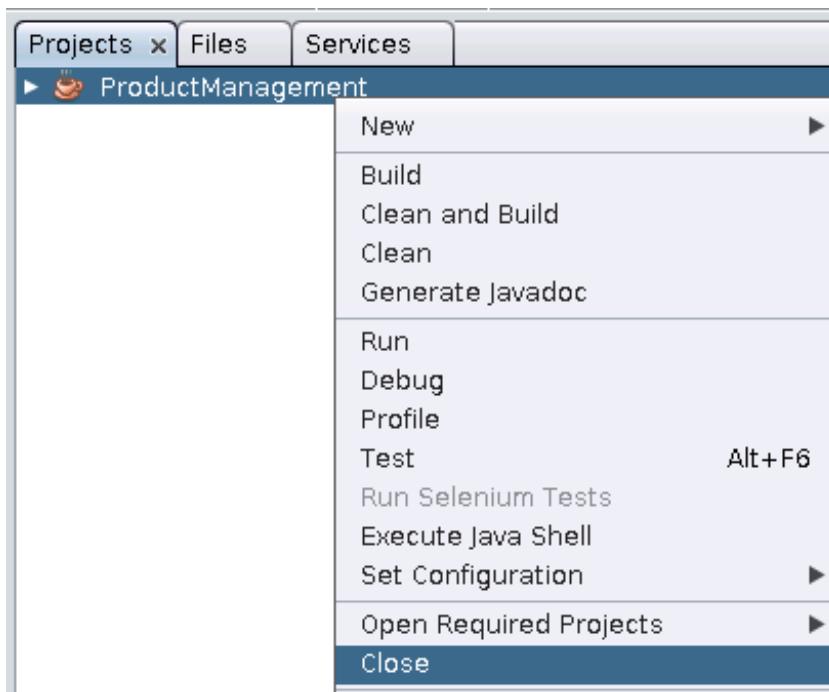
- You may continue to use the same NetBeans project as before, if you have successfully completed the previous practice. In this case, proceed directly to Practice 9-1, step 2.
- Alternatively, you can open a fresh copy of the NetBeans project, which contains the completed solution for the previous practice.

- a. Open NetBeans (if it is not already running).



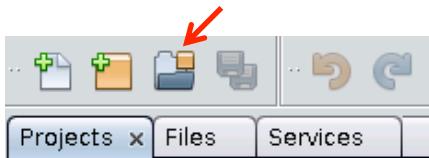
- b. Close the currently opened ProductManagement project.

Hint: Right-click on the ProductManagement project and invoke "Close" menu.



- c. Open Practice 8 version of the ProductManagement project.

Hint: Use File -> Open Project menu or click the "Open Project" toolbar button.



- d. Navigate to /home/oracle/labs/practice8 folder and select ProductManagement project.
- e. Click "Open Project".
2. Modify ProductManager class to store Product and Review objects in the HashMap.
- Open ProductManager class editor.
 - Replace declarations of the product and reviews variables with a new private instance variable called products of Map type that should be declared and initialized to use Product objects as keys and a List of Review objects per product as values.
 - This Map declaration should use generics.
 - Use HashMap implementation of Map interface.
 - The key in the map should be type of Product.
 - The value in the map should be type of a List of Review objects.

```
private Map<Product, List<Review>> products = new HashMap<>();
```
 - Add an import statement for java.util.Map, java.util.List, and java.util.HashMap.

Hint: Right-click anywhere in the source code of the ProductManager class and invoke "Fix Imports" menu.

```
import java.util.Map;
import java.util.List;
import java.util.HashMap;
```

3. Modify both versions of createProduct method to put a new instance of ArrayList of Review objects together with the Product object into a map as a new entry. Make sure you do not try to put Product into a map if such a product is already present in it.

- a. Change the line of code that creates a new Product instance in both createProduct methods so that it assigns this new Product to a local product variable, since you have already removed an instance variable from the ProductManager class editor.

```
Product product = new Food(id, name, price, rating, bestBefore);  
and
```

```
Product product = new Drink(id, name, price, rating);
```

- b. Before returning product object from these methods, place it into a products map, together with a new ArrayList of Review objects. Do not replace map entry for this product if it already exists in the map.

Hint: Use putIfAbsent method of a Map interface to perform conditional insertion of a new entry into a map.

```
public Product createProduct(int id, String name, BigDecimal price,
                             Rating rating, LocalDate bestBefore) {
    Product product = new Food(id, name, price, rating, bestBefore);
    products.putIfAbsent(product, new ArrayList<>());
    return product;
}

public Product createProduct(int id, String name, BigDecimal price,
                             Rating rating) {
    Product product = new Drink(id, name, price, rating);
    products.putIfAbsent(product, new ArrayList<>());
    return product;
}
```

Notes: Using a simple put method would result in a replacement of map entry value for a given product. In this case, map entry value is a new list of reviews. This could lead to the loss of all reviews associated with a map entry for a specific product, in case this product will be accidentally attempted to be placed into a map more than once. The algorithm that determines if this product object is the same as some other is based on a logic of the equals method. Currently the logic of the Product equals method assumes that any two products with the same id and name should be considered the same product.

- c. Add an import statement for the java.util.ArrayList.

Hint: Right-click anywhere in the source code of the ProductManager class and invoke "Fix Imports" menu.

```
import java.util.ArrayList;
```

4. Modify logic of the reviewProduct method of the ProductManager class to use a Map of products and a List of reviews instead of a simple single product variable and an array of reviews.
- a. Remove the block of code from the beginning of the reviewProduct method that verifies that array is full and recreates the array of a larger capacity. This code is no longer required, because an ArrayList would automatically extend to accommodate more Review objects if required.

- b. Now that the `ProductManager` class stores more than one product, you need to write code in the `reviewProduct` method to the locale and get the Map entry for the corresponding product. Place this code in the beginning of the `reviewProduct` method. You need to get the List of the Review objects for the corresponding product from the map.

```
List<Review> reviews = products.get(product);
```

- c. Next, you need to remove this entire entry from the map. This entry will be added back to the map after you will add a review and apply new Rating to the Product object.

```
products.remove(product, reviews);
```

Notes:

- Product object is used as a key in the products Map.
 - Method `applyRating` creates a new Product object, which has to be placed into a map instead of the previous version of this product.
 - Map interface provides methods to replace map values or put new entries.
 - Map does not provide a method to replace the key itself. This can only be achieved by removing and recreating the map entry.
- d. Next, you need to create a new `Review` object and append it to the `reviews` list. Use `rating` and `comments` parameters for the `Review` constructor.
- ```
reviews.add(new Review(rating, comments));
```
- e. Next, you need to iterate through the list of reviews and calculate the total sum of all ratings. Your algorithm would still require the `sum` variable, but you would not need to keep counting iterations or detect a next available slot in the array to insert the review. Remove declaration of variables `i` and `reviewed`. Keep declaration and initialization of variable `sum`.
- f. Replace the `while` loop with the `forEach` loop to iterate through the list of `reviews`.
- Remove all logic related to array iteration handling, verification that array element is null, and construction of a new `Review` object from the loop.
  - Keep only the actual calculation of the rating sum.
  - You are no longer using an int index to access array elements, so change the way you're referencing each `Review` object to use `review` variable provided by the `forEach`, instead of an array reference with an index.

```
for (Review review: reviews) {
 sum += review.getRating().ordinal();
}
```

- g. After the end of the loop, modify a line of code that applies rating to a product.

**Hints:**

- Instance variable representing a product has been removed from the `ProductManager` class. You should replace it with the use of a local variable that refers to the product for which you are computing a new rating value.
- You are no longer using the `i` variable to determine a number of reviews in the array. Instead, use the `size` method provided by the `List` to determine a number of reviews.

```
product = product.applyRating(Rateable.convert(
 Math.round((float)sum/reviews.size())));
```

- h. Next, put the product and the list of reviews back into the products map.

```
products.put(product, reviews);
```

- i. Modify the return statement to return the product local variable instead of an instance variable.

```
return product;
```

**Notes:**

- This is the overall result that you should have achieved by modifying the logic of the `reviewProduct` method:

```
public Product reviewProduct(Product product,
 Rating rating, String comments) {
 List<Review> reviews = products.get(product);
 products.remove(product, reviews);
 reviews.add(new Review(rating, comments));
 int sum = 0;
 for (Review review: reviews) {
 sum += review.getRating().ordinal();
 }
 product = product.applyRating(Rateable.convert(
 Math.round((float)sum/reviews.size())));
 products.put(product, reviews);
 return product;
}
```

- This code appears to be shorter and more "automated" compared to the previous version of the code that used arrays, even though the new version of the code is handling an entire map of products and associated lists of reviews, rather than just one array of reviews for a single product.

5. Modify the logic of the `printProductReport` method of the `ProductManager` class to iterate through the list of reviews for the specific product.

- a. Add an argument type of `Product` called `product` to the `printProductReport` method.

```
public void printProductReport(Product product) {
 // existing logic
}
```

- b. At the beginning of the `printProductReport` method, add code that locates a list of reviews for the corresponding product in the map of products.

```
List<Review> reviews = products.get(product);
```

**Note:** No changes are required to the code that formats and appends product information. However, adjustments should be applied to the code that iterates through reviews, since you are no longer using reviews array, but a reviews list instead.

- c. Remove entire `if` block from inside the `forEach` loop that iterates through reviews, because you no longer need to check if you encounter an empty array element.

- d. Modify the `if` condition that checks if there were no reviews available. Use `isEmpty` method provided by the `List` to determine if `no.reviews` text needs to be appended.

```
if (reviews.isEmpty()) {
 txt.append(resources.getString("no.reviews"));
 txt.append('\n');
}
```

**Notes:**

- This is the overall result that you should have achieved by modifying the logic of the `printProductReport` method:

```

public void printProductReport(Shop product) {
 List<Review> reviews = products.get(product);
 StringBuilder txt = new StringBuilder();
 txt.append(MessageFormat.format(resources.getString("product"),
 product.getName(),
 moneyFormat.format(product.getPrice()),
 product.getRating().getStars(),
 dateFormat.format(product.getBestBefore())));
 txt.append('\n');
 for (Review review : reviews) {
 txt.append(MessageFormat.format(resources.getString("review"),
 review.getRating().getStars(),
 review.getComments())));
 txt.append('\n');
 }
 if (reviews.isEmpty()) {
 txt.append(resources.getString("no.reviews"));
 txt.append('\n');
 }
 System.out.println(txt);
}
}

```

- This code appears to be shorter and more "automated" compared to the previous version of the code that used arrays.
- Class Shop no longer compiles, because you have changed `printProductReport` method signature to include additional parameter. You next task will be to fix class Shop to actually pass the `product` parameter to the `printProductReport` method.

6. Modify main method of the Shop class so it will pass product as an argument to the printProductReport method.

- Open Shop class editor.
- Pass p1 product object to the printProductReport method on both occasions.

```
ProductManager pm = new ProductManager(Locale.UK);
Product p1 = pm.createProduct(101, "Tea",
 BigDecimal.valueOf(1.99), Rating.NOT_RATED);
pm.printProductReport(p1);
p1 = pm.reviewProduct(p1, Rating.FOUR_STAR, "Nice hot cup of tea");
p1 = pm.reviewProduct(p1, Rating.TWO_STAR, "Rather weak tea");
p1 = pm.reviewProduct(p1, Rating.FOUR_STAR, "Fine tea");
p1 = pm.reviewProduct(p1, Rating.FOUR_STAR, "Good tea");
p1 = pm.reviewProduct(p1, Rating.FIVE_STAR, "Perfect tea");
p1 = pm.reviewProduct(p1, Rating.THREE_STAR, "Just add some lemon");
pm.printProductReport(p1);
```

- Compile and run your application.

**Hint:** Click a "Run Project" toolbar button.



**Note:** Observe product details and product reviews printed on the console. You may add intermediate printings of the product report if you want to observe changes in the average rating calculation.

7. Create more product objects and more reviews for these products to test multi-product capabilities of the ProductManager class.

**Notes:**

- You may use existing values from the commented segments of the Shop class code to construct new products and add these products to the map maintained by the ProductManager.
  - Use NOT\_RATED Rating enum value for all products at the point when they are just created. This value will be recalculated as you add reviews for the product.
- Add more instances of Product and Review objects and print reports for each of the products.

```
Product p2 = pm.createProduct(102, "Coffee",
 BigDecimal.valueOf(1.99), Rating.NOT_RATED);
p2 = pm.reviewProduct(p2, Rating.THREE_STAR, "Coffee was ok");
p2 = pm.reviewProduct(p2, Rating.ONE_STAR, "Where is the milk?!?");
p2 = pm.reviewProduct(p2, Rating.FIVE_STAR,
 "It's perfect with ten spoons of sugar!");
pm.printProductReport(p2);
```

```

Product p3 = pm.createProduct(103, "Cake",
 BigDecimal.valueOf(3.99), Rating.NOT_RATED,
 LocalDate.now().plusDays(2));
p3 = pm.reviewProduct(p3, Rating.FIVE_STAR, "Very nice cake");
p3 = pm.reviewProduct(p3, Rating.FOUR_STAR,
 "It good, but I've expected more chocolate");
p3 = pm.reviewProduct(p3, Rating.FIVE_STAR, "This cake is perfect!");
pm.printProductReport(p3);

Product p4 = pm.createProduct(104, "Cookie",
 BigDecimal.valueOf(2.99), Rating.NOT_RATED,
 LocalDate.now());
p4 = pm.reviewProduct(p4, Rating.THREE_STAR, "Just another cookie");
p4 = pm.reviewProduct(p4, Rating.THREE_STAR, "Ok");
pm.printProductReport(p4);

Product p5 = pm.createProduct(105, "Hot Chocolate",
 BigDecimal.valueOf(2.50), Rating.NOT_RATED);
p5 = pm.reviewProduct(p5, Rating.FOUR_STAR, "Tasty!");
p5 = pm.reviewProduct(p5, Rating.FOUR_STAR, "No bad at all");
pm.printProductReport(p5);

Product p6 = pm.createProduct(106, "Chocolate",
 BigDecimal.valueOf(2.50), Rating.NOT_RATED,
 LocalDate.now().plusDays(3));
p6 = pm.reviewProduct(p6, Rating.TWO_STAR, "Too sweet");
p6 = pm.reviewProduct(p6, Rating.THREE_STAR, "Better then cookie");
p6 = pm.reviewProduct(p6, Rating.TWO_STAR, "Too bitter");
p6 = pm.reviewProduct(p6, Rating.ONE_STAR, "I don't get it!");
pm.printProductReport(p6);

```

- b. You may remove the remaining comments from the `main` method of the `Shop` class.
- c. Compile and run your application.

**Hint:** Click the "Run Project" toolbar button.



**Note:** Observe product details and product reviews printed on the console.

## Practice 9-2: Implement Review Sort and Product Search Features

---

### Overview

In this practice, you implement Comparable interface in the Review class to allow reviews to be ordered by rating. You also add sorting logic to the printProductReport method of the ProductManager class. Another task is to implement an ability to search for the specific Product object in the products map maintained within the ProductManager class.

1. Implement Comparable interface in the Review class to order reviews by their ratings.

- a. Open Review class editor.
- b. Add implements clause to the Review class definition to implement Comparable interface, using Review as a generic type.

```
public class Review implements Comparable<Review> {
 // Review class body
}
```

- c. Override compareTo method of the Comparable interface within the Review class.

**Hint:** Right-click the left-hand side of the line of code that declares Review class and invoke "Implement all abstract methods" menu.

```
@Override
public int compareTo(Review arg0) {
 throw new UnsupportedOperationException("Not supported yet.");
}
```

- d. Rename compareTo method argument to other to indicate the nature of the compareTo method logic, which is supposed to compare properties of the current object to properties of the other object represented by this parameter.

```
public int compareTo(Review other)
```

- e. Replace throw exception clause in the body of the `compareTo` method with review comparing algorithm that arranges reviews in the order from the highest number of stars to the lowest.

**Hints:**

- Use method `ordinal` to get the numeric value of the `Rating` enumeration object for each `Review` object.
- `compareTo` method is expected to return `-1` if the current object is less than a parameter, `0` if they are the same, or `+1` if the current object is greater than a parameter.
- However, actual values do not have to be `-1` or `+1`, but rather any negative or positive number that can indicate the ordering. Consider calculating this number as a difference of ordinal values of `Rating` objects.

```
@Override
public int compareTo(Review other) {
 return other.rating.ordinal() - this.rating.ordinal();
}
```

2. Add sorting capability to `printProductReport` method of the `ProductManager` class.

- a. Open `ProductManager` class editor
- b. Order the content of the `reviews` list using `sort` method of the `Collections` class. This ordering should be performed on the next line of code immediately after the declaration and initialization of the `reviews` list.

```
Collections.sort(reviews);
```

- c. Add an import statement for the `java.util.Collections` class.

**Hint:** Right-click anywhere in the source code of the `ProductManager` class and invoke "Fix Imports" menu.

```
import java.util.Collections;
```

- d. Compile and run your application.

**Hint:** Click the "Run Project" toolbar button.



**Note:** Observe product details and product reviews printed on the console.

3. Implement a method that searches for specific product object in the collection of Products maintained within ProductManager class.
  - a. Open ProductManager class editor
  - b. Add a new method called `findProduct` that performs Product search based on a product id value.

**Hints:**

- Method `findProduct` should have `public` access.
- It should return `Product` object.
- It should accept `int id` as an argument.
- Add this method just before the `reviewProduct` method.

```
public Product findProduct(int id) {
 // product search logic will be added here
}
```

- c. Inside the `findProduct` method, add a declaration of the variable called `result`, type of `Product`, and set this variable to null.

```
Product result = null;
```

- d. Next, add a `forEach` loop that iterates through the set of products. Use `keySet` method of a `products` map to get the set of products out of the map.

```
for (Product product : products.keySet()) {
 // product id comparing will be added here
}
```

- e. Inside this loop, add an if statement that compares each product id to the value of parameter. If these values match, assign `result` variable to reference the product that you have located and break from the loop.

```
if (product.getId() == id) {
 result = product;
 break;
}
```

- f. After the end of the `forEach` loop, add return clause that returns the `result` object.

```
return result;
```

4. Create overloaded versions of `reviewProduct` and `printProductReport` methods that use `int id` parameter and locate the required product using `findProduct` method.
  - a. Just in before the `reviewProduct` method, add an additional overloaded version of the `reviewProduct` method.

**Hints:**

- The first parameter of this newly added method should accept `int id` value, instead of the Product reference.
- All other parameters should remain the same as in the existing version of the `reviewProduct` method.
- This method should invoke the existing version of the `reviewProduct` method and return its result.
- To locate the product, use `findProduct` method.
- Pass other parameters as is.

```
public Product reviewProduct(int id, Rating rating, String comments) {
 return reviewProduct(findProduct(id), rating, comments);
}
```

- b. Just in before the `printProductReport` method, add an additional overloaded version of the `printProductReport` method.

**Hints:**

- The parameter of the this newly added method should accept `int id` value, instead of the Product reference.
- This method should invoke the existing version of the `printProductReport` method and return its result.
- To locate the product, use `findProduct` method.

```
public void printProductReport(int id) {
 printProductReport(findProduct(id));
}
```

5. Change the way Product objects are compared.

**Notes:**

- Set uses `equals` method to ensure the uniqueness of objects stored inside it. So no two products that return true when compared with the `equals` method should ever be inside the Set.
- If you look at how Product class implements `equals` method, you will notice that it compares both product ids and names. Therefore, it is possible that there could be two products with the same id, so long as they have different names.
- In this part of the practice, you modify Product `equals` method to compare only product ids, in order to guarantee that no two products could have the same id and be stored in the products Map.

- Open Product class editor.
- Remove name comparing from the equals method.

```

@Override
public boolean equals(Object obj) {
 if (this == obj) {
 return true;
 }
 if (obj instanceof Product) {
 final Product other = (Product) obj;
 return this.id == other.id;
 }
 return false;
}

```

- Recompile ProductManagement project.

**Hint:** Use Run->Clean and Build Project menu or a toolbar button.



- Change the way Shop class is referencing Product objects.

**Notes:**

- In this practice, you have provided capability to uniquely identify and locate Product objects using relevant product id value.
  - This means that the Shop class is no longer required to keep up-to-date product references, but can instead locate required product at any time, with the simple id lookup.
- Open Shop class editor
  - Remove all Product object references: p1, p2, p3, p4, p5, and p6. Replace these references with corresponding product id values: 101, 102, 103, 104, 105, and 106.

```

ProductManager pm = new ProductManager(Locale.UK);
pm.createProduct(101, "Tea", BigDecimal.valueOf(1.99),
 Rating.NOT_RATED);
pm.printProductReport(101);
pm.reviewProduct(101, Rating.FOUR_STAR, "Nice hot cup of tea");
pm.reviewProduct(101, Rating.TWO_STAR, "Rather weak tea");
pm.reviewProduct(101, Rating.FOUR_STAR, "Fine tea");
pm.reviewProduct(101, Rating.FOUR_STAR, "Good tea");
pm.reviewProduct(101, Rating.FIVE_STAR, "Perfect tea");
pm.reviewProduct(101, Rating.THREE_STAR, "Just add some lemon");
pm.printProductReport(101);
pm.createProduct(102, "Coffee", BigDecimal.valueOf(1.99),
 Rating.NOT_RATED);

```

```

pm.reviewProduct(102, Rating.THREE_STAR, "Coffee was ok");
pm.reviewProduct(102, Rating.ONE_STAR, "Where is the milk?!");
pm.reviewProduct(102, Rating.FIVE_STAR,
 "It's perfect with ten spoons of sugar!");
pm.printProductReport(102);
pm.createProduct(103, "Cake", BigDecimal.valueOf(3.99),
 Rating.NOT_RATED, LocalDate.now().plusDays(2));
pm.reviewProduct(103, Rating.FIVE_STAR, "Very nice cake");
pm.reviewProduct(103, Rating.FOUR_STAR,
 "It good, but I've expected more chocolate");
pm.reviewProduct(103, Rating.FIVE_STAR, "This cake is perfect!");
pm.printProductReport(103);
pm.createProduct(104, "Cookie", BigDecimal.valueOf(2.99),
 Rating.NOT_RATED, LocalDate.now());
pm.reviewProduct(104, Rating.THREE_STAR, "Just another cookie");
pm.reviewProduct(104, Rating.THREE_STAR, "Ok");
pm.printProductReport(104);
pm.createProduct(105, "Hot Chocolate", BigDecimal.valueOf(2.50),
 Rating.NOT_RATED);
pm.reviewProduct(105, Rating.FOUR_STAR, "Tasty!");
pm.reviewProduct(105, Rating.FOUR_STAR, "No bad at all");
pm.printProductReport(105);
pm.createProduct(106, "Chocolate", BigDecimal.valueOf(2.50),
 Rating.NOT_RATED, LocalDate.now().plusDays(3));
pm.reviewProduct(106, Rating.TWO_STAR, "Too sweet");
pm.reviewProduct(106, Rating.THREE_STAR, "Better than cookie");
pm.reviewProduct(106, Rating.TWO_STAR, "Too bitter");
pm.reviewProduct(106, Rating.ONE_STAR, "I don't get it!");
pm.printProductReport(106);

```

**Note:** Both styles of invocation (using product id and an object reference) are still available via the ProductManagement class, because you did not remove the existing methods, but added extra overloaded versions.

- Compile and run your application.

**Hint:** Click the "Run Project" toolbar button.



**Note:** Observe the product details and product reviews printed on the console.



## **Practices for Lesson 10: Nested Classes and Lambda Expressions**

## Practices for Lesson 10: Overview

### Overview

In these practices, you change the design of the ProductManagement class, creating static nested helper class to encapsulate management of text resources and localization. This new design helps to separate presentation and business logic of the application. Another task is to provide various product-sorting options using lambda expressions that implement Comparator interface.



## Practice 10-1: Refactor ProductManger to use a Nested Class

---

### Overview

In this practice, you put all localization and text formatting capabilities into a static nested class inside the ProductManager.

This design change does not require you to write a lot of new code, but rather move existing code, which involves mostly cutting and pasting code from ProductManager class to the ResourceFormatter static nested class.

You now have a choice:

- A. Follow Practice 10-1.A instructions on how to refactor the code.
- B. Open solution for Practice 10-1 where code has already been refactored and read through the practice 10-1.B to understand design changes and then proceed to practice 10-2

In any case you must either complete practice 10-1 A or B, or use solution for it, before proceeding to practice 10-2.

## Practice 10-1.A: Refactor ProductManger to use a Nested Class

### Assumptions

- JDK 11 is installed.
- NetBeans 11 is installed.
- You have completed Practice 9 or start with the solution for Practice 9 version of the application.

### Tasks

1. Prepare the practice environment.

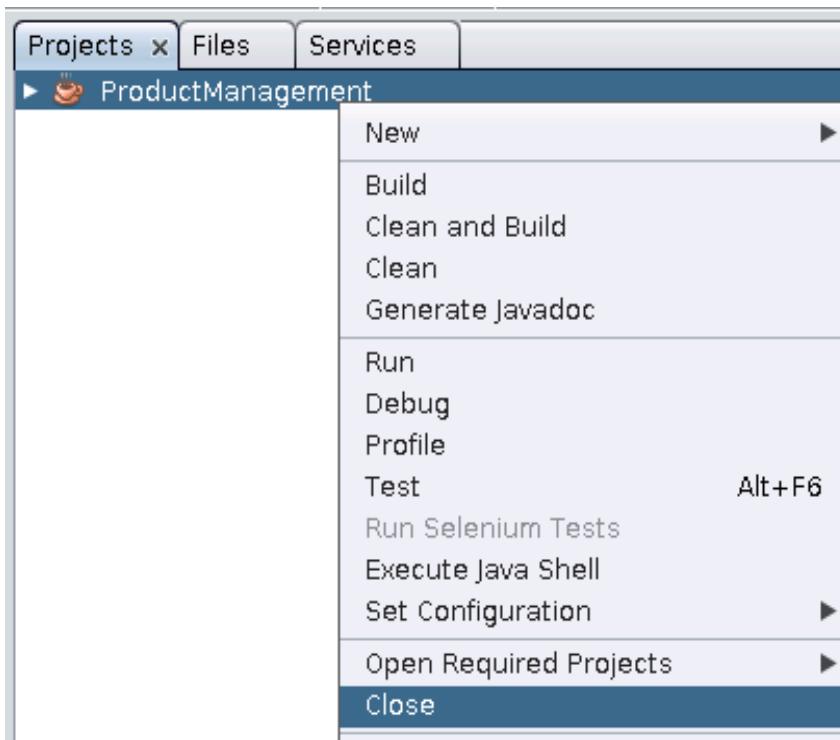
#### Notes:

- You may continue to use the same NetBeans project as before, if you have successfully completed the previous practice.  
In this case, proceed directly to Practice 10-1.A, step 2.
- Alternatively, you can open a fresh copy of the NetBeans project, which contains the completed solution for the previous practice.
- a. Open NetBeans (if it is not already running).



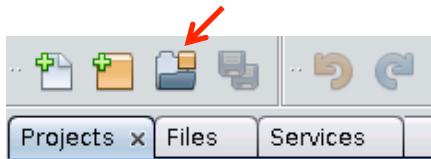
- b. Close the currently opened ProductManagement project.

**Hint:** Right-click on the ProductManagement project and invoke "Close" menu.



- c. Open solution for Practice 9 version of the ProductManagement project.

**Hint:** Use File -> Open Project menu or click the "Open Project" toolbar button.



- d. Navigate to /home/oracle/labs/solutions/practice9 folder and select ProductManagement project.  
e. Click "Open Project".
2. Create a static inner class called ResourceFormatter inside the ProductManager class and move all logic related to localization, resource management, and formatting into this nested class.
- Open ProductManager class editor.
  - Just before the end of the ProductManager class body, add a new static nested class definition. This nested class should be called ResourceFormatter, and it should be made private.

```
private static class ResourceFormatter {
 // more code will be added here
}
```

- c. Move declarations of locale, resources, dateFormat, and moneyFormat fields into the ResourceFormatter class.

**Hints:**

- Select lines of code that declare these variables in ProductManager class.
- Cut these lines of code **CTRL+X**.
- Move your cursor inside ResourceFormatter class.
- Paste these lines of code **CTRL+V**.

```
private static class ResourceFormatter {
 private Locale locale;
 private ResourceBundle resources;
 private DateTimeFormatter dateFormat;
 private NumberFormat moneyFormat;
 // more code will be added here
}
```

- d. Add constructor to ResourceFormatter class that accepts Locale as an argument. You can make this constructor **private**, because it can still be accessed by the outer class, and no other class should be able to access it anyway.

```
private ResourceFormatter(Locale locale) {
 // field initialisation code will be added here
}
```

- e. Move initialization code of `locale`, `resources`, `dateFormat`, and `moneyFormat` fields from `ProductManager` constructor to the `ResourceFormatter` constructor.

**Hints:**

- Select lines of code that initialize these variables in `ProductManager` constructor.
- Cut these lines of code `CTRL+X`.
- Move your cursor inside `ResourceFormatter` constructor.
- Paste these lines of code `CTRL+V`.

```
private ResourceFormatter(Locale locale) {
 this.locale = locale;
 resources =
 ResourceBundle.getBundle("labs.pm.data.resources", locale);
 dateFormat = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT)
 .localizedBy(locale);
 moneyFormat = NumberFormat.getCurrencyInstance(locale);
}
```

- f. Add `formatProduct` operation to the `ResourceFormatter` class.

- This operation should have `private` access.
- It should accept `Product` as an argument.
- It should return `String` value.
- Add this operation just before the end of `ResourceFormatter` class body.

```
private String formatProduct(Product product) {
 // product text formatting all be added here
}
```

g. Add return statement into to the `formatProduct` method.

- This return statement should format the product message taken from the resource bundle, substituting values of product name, formatted value of price, stars rating, and formatted value for the best before date.
- Cut the required expression from the `append` operation inside the `printProductReport` method and paste it as a return value inside the `formatProduct` method.

```
private String formatProduct(Product product) {
 return MessageFormat.format(resources.getString("product"),
 product.getName(),
 moneyFormat.format(product.getPrice()),
 product.getRating().getStars(),
 dateFormat.format(product.getBestBefore()));
}
```

**Note:** This would leave the append operation inside the `printProductReport` method temporarily broken. You will fix it in a later step of this practice.

h. Add `formatReview` operation to the `ResourceFormatter` class.

- This operation should have `private` access.
- It should accept `Review` as an argument.
- It should return `String` value.
- Add this operation just before the end of `ResourceFormatter` class body.

```
private String formatReview(Review review) {
 // review text formatting all be added here
}
```

i. Add return statement into the `formatReview` method.

- This return statement should format review message taken from the resource bundle, substituting values of star rating and comments.
- Cut the required expression from the `append` operation inside the `printProductReport` method and paste it as a return value inside the `formatReview` method.

```
private String formatReview(Review review) {
 return MessageFormat.format(resources.getString("review"),
 review.getRating().getStars(), review.getComments());
}
```

**Note:** This would leave the append operation inside the `printProductReport` method temporarily broken. You will fix it in a later step of this practice.

- j. Add `getText` operation to the `ResourceFormatter` class.
- This operation should have `private access`.
  - It should accept `String` argument, which represents resource key value.
  - It should return `String` value fetched from the `resources` bundle using the key provided.
  - Add this operation just before the end of `ResourceFormatter` class body.

```
private String getText(String key) {
 return resources.getString(key);
}
```

3. Write code inside the `ProductManager` class that creates `ResourceFormatter` instances and performs a selection of a specific `ResourceFormatter` instance.
- a. Add a new static variable to the `ProductManager` class that will represent a `Map` of `ResourceFormatter` instances indexed by a `String` value.
- Place this variable declaration just before the `ProductManager` constructor.
  - This variable should have `private access`.
  - Use generics to restrict `Map` value types—`String` as the key and `ResourceFormatter` as the value.
  - Use `formatters` as the variable name.
  - Initialize this variable using `Map.of` method to construct as map of `ResourceFormatter` instances, using locales that represent various locales (in the example code, these are: UK, USA, France, Russia, and China).
  - The later code examples assume that UK locale is present in this map, so it would be best if you put it in, even though you are free to choose other locales.
  - The key value for map entries should be the language tag of the corresponding locale (for example, "en-GB" for the UK locale).

```
private static Map<String, ResourceFormatter> formatters =
 Map.of("en-GB", new ResourceFormatter(Locale.UK),
 "en-US", new ResourceFormatter(Locale.US),
 "fr-FR", new ResourceFormatter(Locale.FRANCE),
 "ru-RU", new ResourceFormatter(new Locale("ru", "RU")),
 "zh-CN", new ResourceFormatter(Locale.CHINA));
```

**Note:** If you wish, you may use less obvious locale choices.

For example: `new Locale{"es", "US"}` represents Spanish American locale or `new Locale{"fr", "CA"}` represents French Canadian locale. More information can be found in the lesson titled "Text, Date, Time and Numeric Objects" of this course.

- b. Add a new instance variable to the `ProductManager` class, which will represent a specific `ResourceFormatter` instance that should be used by a specific `ProductManager` instance.

- Place this variable declaration just before the static `formatters` variable.
- This variable should have `private` access.
- Use `formatter` as the variable name.
- This variable will be initialized later.

```
private ResourceFormatter formatter;
```

- c. Add a new operation to the `ProductManager` class that changes the selection of the `ResourceFormatter` for the current instance of the `ProductManager` based on a locale language tag provided as an argument.

- Place this method declaration just after the end of the `ProductManager` constructor body.
- This method should have `public` access.
- This method should be `void`.
- Use `changeLocale` as the method name.
- Accept `String languageTag` as an argument.
- This method should initialize `formatter` variable based in a matching `ResourceFormatter` object from the `formatters` map.
- Use `getOrDefault` method of the map interface to pick up the matching `ResourceFormatter` or the `ResourceFormatter` for the "en-GB" language tag.

```
public void changeLocale(String languageTag) {
 formatter = formatters.getOrDefault(languageTag,
 formatters.get("en-GB"));
}
```

- d. Add a new operation to the `ProductManager` class that returns a set of all supported locales.

- Place this method declaration just after the end of the `changeLocale` method body.
- This method should have `public` access.
- This method should be marked with `static` keyword.
- Use `getSupportedLocales` as the method name.
- This method should return a Set of `String` objects representing language tag values from the `formatters` Map.
- No arguments are required for this method.
- This method should return a set of keys from the `formatters` Map.

```
public static Set<String> getSupportedLocales() {
 return formatters.keySet();
}
```

**Note:** Method `getSupportedLocales` is marked with the `static` keyword because its code retrieves the Set of `Locale` objects from the `ResourceFormatter` objects Map, which is accosted with the class (`static`) context of the `ProductManager` class.

- e. Add an import statement for the `java.util.Set` class.

**Hint:** Right-click anywhere in the source code of the `ProductManager` class and invoke "Fix Imports" menu.

```
import java.util.Set;
```

- f. Add additional (overloaded) version of the `ProductManager` constructor, which accepts `String languageTag` argument, and pass it on to the `changeLocale` method. Add this constructor immediately after the end of the existing constructor body.

```
public ProductManager(String languageTag) {
 changeLocale(languageTag);
}
```

- g. Modify `ProductManager` constructor, which accepts `Locale` argument, so that it invokes the other constructor using `this` keyword.

```
public ProductManager(Locale locale) {
 this(locale.toLanguageTag());
}
```

4. Modify logic inside the `printProductReport` method to utilize `ResourceFormatter` to format values.

- a. Add parameter to the first `append` method invocation inside `printProductReport` method to format the text for the `product` object.

```
txt.append(formatter.formatProduct(product));
```

- b. Add parameter to the `append` method invocation inside the `forEach` loop that iterates through `reviews` to format the text for the `review` object.

```
txt.append(formatter.formatReview(review));
```

- c. Modify parameter value of the `append` method invocation inside the `if` block that prints a message, which indicates no reviews to request relevant text from the `formatter` object.

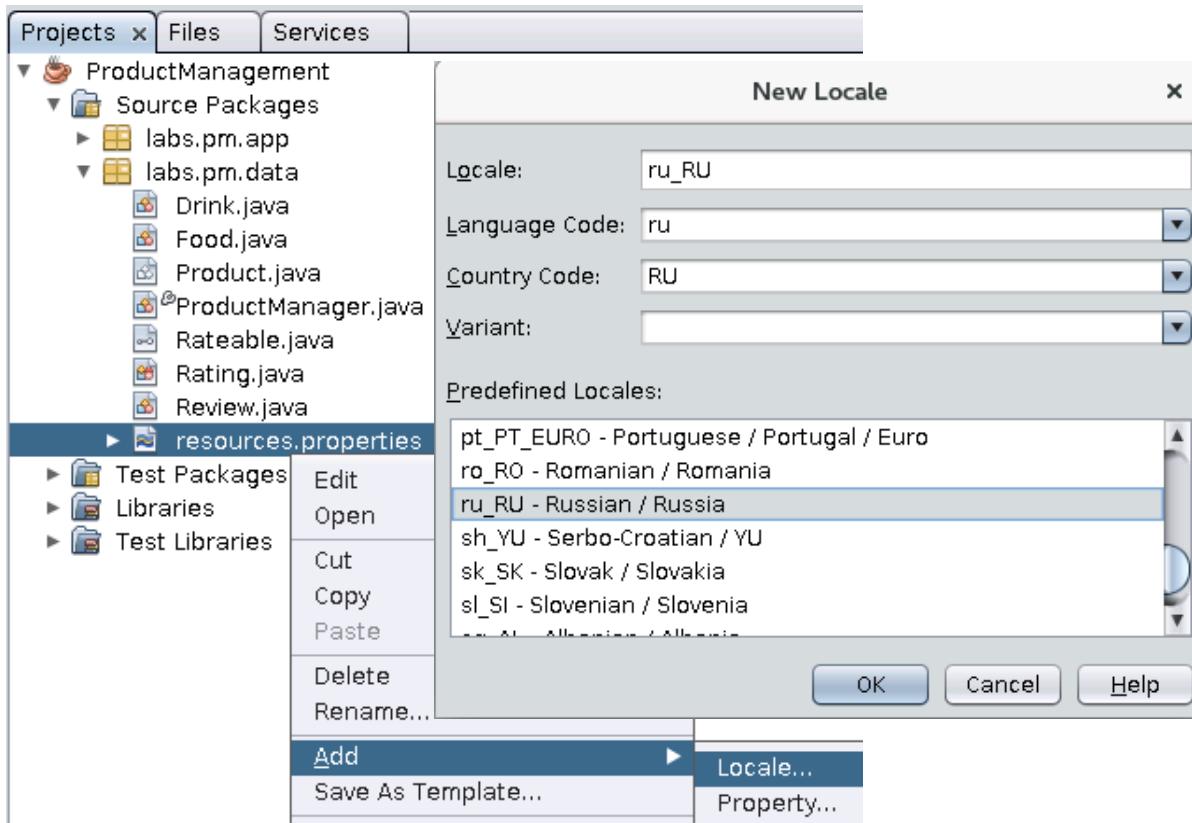
```
txt.append(formatter.getText("no.reviews"));
```

**Notes:**

- This is the overall result that you should have achieved by modifying the logic of the `printProductReport` method (parts modified in this practice are highlighted):

```
public void printProductReport(Product product) {
 List<Review> reviews = products.get(product);
 Collections.sort(reviews);
 StringBuilder txt = new StringBuilder();
 txt.append(formatter.formatProduct(product));
 txt.append('\n');
 for (Review review : reviews) {
 txt.append(formatter.formatReview(review));
 txt.append('\n');
 }
 if (reviews.isEmpty()) {
 txt.append(formatter.getText("no.reviews"));
 txt.append('\n');
 }
 System.out.println(txt);
}
```

5. Create a copy of the `resources.properties` file to support a different language and country.
  - a. Right-click on a `resources.properties` file in the ProductManagement project navigator and invoke "Add" and then "Locale..." menu.
  - b. Select locale for Russian/Russia from the list (this could be a different locale, if you have placed such an option into the map of formatter objects earlier in this practice).



- c. Click "OK."
  - d. Open `resources_ru_RU.properties` file editor (or another locale version, if that was your choice in the earlier step).
  - e. Provide translations for all text resources in this file.
- ```
product={0}, Цена: {1}, Рейтинг: {2}, Употребить до: {3}
review=Отзыв: {0}\t{1}
no.reviews=Нет отзывов
```
- f. (Optionally) repeat this process to provide other translations.

6. Test locale switching capabilities provided by the nested inner class inside the `ProductManager` from the `main` method of the `Shop` class.
 - a. Open `Shop` class editor.
 - b. Replace parameter value in the invocation of the `ProductManager` constructor with the `String` value representing the relevant locale tag; instead do an actual `Locale` object.

```
ProductManager pm = new ProductManager("en-GB");
```
 - Note:** This change is actually not required as both versions of constructor are available in the `ProductManager` class.
 - c. Switch locale to Russian (or whatever other locale you have configured earlier) just before the line of code that prints second product report (for product p2).

```
pm.changeLocale("ru-RU");
```
 - d. (Optionally) Repeat the switch to other locales before printing other product reports. "en-US", "fr-FR", "zh-CN" or whatever other locales you have configured in the `ProductManager` earlier.
 - e. Compile and run your application.

Hint: Click a "Run Project" toolbar button.



Notes:

- Observe product details and product reviews printed on the console.
- Format of prices and dates changes for every locale switch. Alternative text values are picked up from the resource bundle only if a bundle for the corresponding country and language is actually provided, otherwise the default bundle is used.

Practice 10-1.B: Examine Solution for ProductManger refactored

Assumptions

- JDK 11 is installed.
- NetBeans 11 is installed.
- You have decided not to complete practice 10-1.A and instead want to analyze design changes using solution for Practice 10-1 without making these changes yourself.

Tasks

1. Prepare the practice environment.

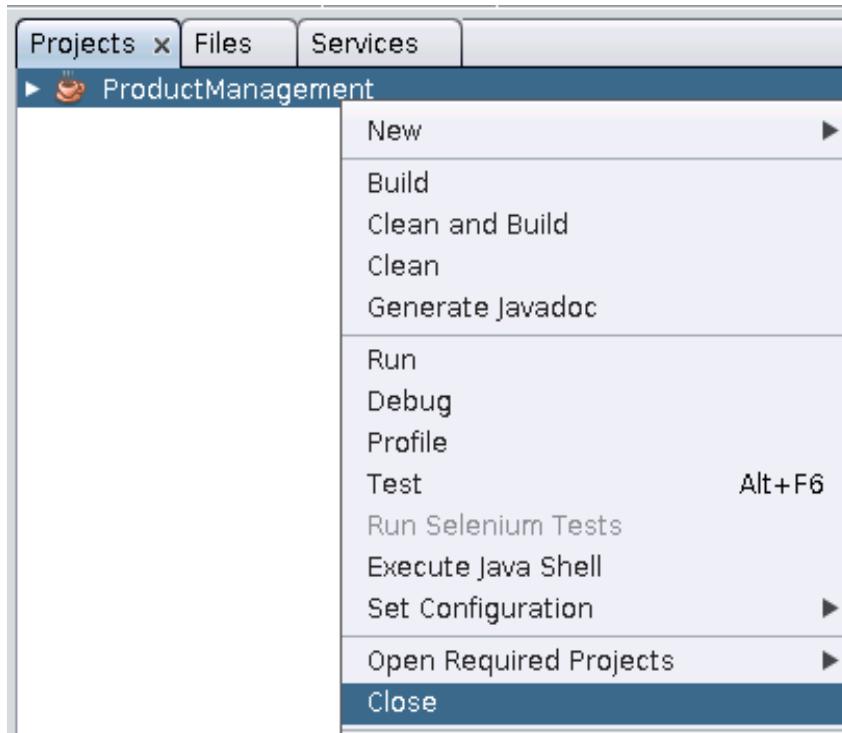
You must open the solution project for Practice 10-1.

- a. Open NetBeans (if it is not already running).



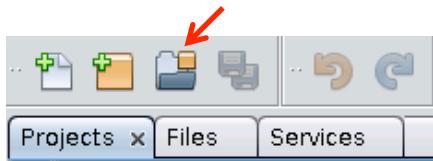
- b. Close the currently opened ProductManagement project.

Hint: Right-click on the ProductManagement project and invoke "Close" menu.



- c. Open solution for Practice 10-1 version of the ProductManagement project.

Hint: Use File -> Open Project menu or click the "Open Project" toolbar button.



- d. Navigate to /home/oracle/labs/solutions/practice10.1 folder and select ProductManagement project.
e. Click "Open Project".

2. Overview of key design changes in Practice 10-1

- ProductManager class version 9

Formatting and localization resources were defined in the ProductManager class

```
public class ProductManager {
    private Locale locale;
    private ResourceBundle resources;
    private DateTimeFormatter dateFormat;
    private NumberFormat moneyFormat;
    ...
}
```

- ProductManager class version 10.1

Formatting and localization resources were moved into the ResourceFormatter static nested class and ProductManager class now provides a way of selecting desired formatter object from the collection.

```
public class ProductManager {
    private ResourceFormatter formatter;
    private static Map<String, ResourceFormatter> formatters =
        Map.of("en-GB", new ResourceFormatter(Locale.UK),
               "en-US", new ResourceFormatter(Locale.US),
               "fr-FR", new ResourceFormatter(Locale.FRANCE),
               "ru-RU", new ResourceFormatter(new Locale("ru", "RU"))),
               "zh-CN", new ResourceFormatter(Locale.CHINA));
    ...
    private static class ResourceFormatter {
        private Locale locale;
        private ResourceBundle resources;
        private DateTimeFormatter dateFormat;
        private NumberFormat moneyFormat;
        ...
    }
}
```

- Because of this change, formatting and localization resource initializations were moved from ProductManager class constructor to ResourceFormatter constructor.
- Methods to get supported locales and to change a locale were added to the ProductManager class.
- Methods to get messages from resource bundle and to format products and reviews were added to the ResourceFormatter class.
- Throughout the rest of the ProductManager class all formatting code is refactored to use ResourceFormatter methods.

Practice 10-2: Produce Customized Product Reports

Overview

In this practice, you add ability to the `ProductManager` class to generate product reports based on flexible sorting conditions provided using lambda expressions that implement `Comparator` interface.

1. Add method to the `ProductManager` class to print a number of products with different sorting options.

- a. Open `ProductManager` class editor.
- b. Add new method called `printProducts` right after the end of the `printProductReport` method body.
 - This method should have `public` access.
 - Does not need to return any value
 - Accept `Comparator` of `Product` type as an argument.

```
public void printProducts(Comparator<Product> sorter) {
    // method logic will be added here
}
```

- c. Add an import statement for the `java.util.Comparator` class.

Hint: Right-click anywhere in the source code of the `ProductManager` class and invoke "Fix Imports" menu.

```
import java.util.Comparator;
```

- d. Inside the `printProducts` method, declare a variable to store a list of products you intend to sort.

- This variable should be type of `List` of `Product` objects.
- Give this variable a name `productList`.
- To initialize this variable, create a new `ArrayList` object passing a set of `Product` key objects from the `products` map to the list constructor.
- Accept `Comparator` of `Product` type as an argument.

```
List<Product> productList = new ArrayList<>(products.keySet());
```

- e. Next, apply `sorter` parameter to this list.

```
productList.sort(sorter);
```

- f. Declare and initialize a new `StringBuilder` object reference.

```
StringBuilder txt = new StringBuilder();
```

- g. Add a `forEach` loop to iterate through the list of products, format each product using `ResourceFormatter formatProduct` method, and append formatted result to the `StringBuffer` together with the new line '`\n`' character.

```
for (Product product : productList) {
    txt.append(formatter.formatProduct(product));
    txt.append('\n');
}
```

- h. After the end of the loop body, print the `StringBuffer` object to the console.

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

3. Pass different lambda expressions to the `printProducts` method, from the `main` method of the `Shop` class to implement alternative sorting.

- a. Open `Shop` class editor.

- b. Place comments on all lines of code that invoke `changeLocale` and `printProductReport` methods.

- c. Compile and run your application.



Hint: Click a "Run Project" toolbar button.

Note: Nothing should be printed by your program at this stage.

- d. Invoke `printProducts` operation, passing a lambda expression that orders products based on their ratings.

Hints:

- Place this code just before the end of the `main` method.
- Products should be sorted from the highest to the lowest rating value.
- You are implementing method `compare` of the `Comparator` interface that accepts two parameters and is expected to return `-1` if the first object is less than a second object, `0` if they are the same, or `+1` if the first object is greater than a second object.
- Parameter type is inferred from the context (in this case it is `Product`).
- Use method `ordinal` to get a numeric value of the `Rating` enumeration object for each `Review` object.
- Actual values do not have to be `-1` or `+1`, but rather any negative or positive number that can indicate the ordering. Consider calculating this number as a difference of ordinal values of `Rating` objects.
- The swapping of the order of products (ascending or descending) can simply be achieved by swapping `p2` and `p1` object around within the expression.

```
pm.printProducts((p1, p2) ->
    p2.getRating().ordinal() - p1.getRating().ordinal());
```

- e. Compile and run your application.



Hint: Click a "Run Project" toolbar button.

Note: Observe a list of products that are sorted by rating printed to the console.

- f. Make another call to the `printProducts` operation, passing a lambda expression that orders products based on their price.

Hints:

- Place this code just before the end of the `main` method.
- Products should be sorted from the highest to the lowest rating value.
- You are implementing method `compare` of the `Comparator` interface that accepts two parameters and is expected to return `-1` if the first object is less than a second object, `0` if they are the same, or `+1` if the first object is greater than a second object.
- Parameter type is inferred from the context (in this case it is `Product`).
- Actual values do not have to be `-1` or `+1`, but rather any negative or positive number that can indicate the ordering. Class `BigDecimal` already implements `Comparable` interface and provides `compareTo` with identical semantics.
- The swapping of the order of products (ascending or descending) can simply be achieved by swapping `p2` and `p1` object around within the expression.

```
pm.printProducts((p1, p2) ->
    p2.getPrice().compareTo(p1.getPrice()));
```

- g. Compile and run your application.



Hint: Click a "Run Project" toolbar button.

Note: Observe a list of products that are sorted by price printed to the console.

2. Combine multiple Comparator objects.

- a. Create `Comparable` object reference to sort products by `rating`.

- Just before the end of the method `main`, create a new variable called `ratingSorter`.
- Make this variable type of `Comparable` use `Product` as a generic type restriction.
- Initialize this new variable to reference the lambda expression that sorts products by their rating value. You may copy and paste the existing expression.

```
Comparator<Product> ratingSorter = (p1, p2) ->
    p2.getRating().ordinal() - p1.getRating().ordinal();
```

- b. Add an import statement for the `java.util.Comparator` interface.

Hint: Right-click anywhere in the source code of the `Shop` class and invoke "Fix Imports" menu.

```
import java.util.Comparator;
```

- c. Create Comparable object reference to sort products by price.
- Just before the end of the method main, create a new variable called priceSorter.
 - Make this variable type of Comparable use Product as generic-type restriction.
 - Initialize this new variable to reference the lambda expression that sorts products by their price value. You may copy and paste the existing expression.

```
Comparator<Product> priceSorter = (p1,p2) ->
    p2.getPrice().compareTo(p1.getPrice());
```

- d. Make another call to the printProducts operation, passing a lambda expression that combines ratingSorter and priceSorter using thenComparing method.

```
pm.printProducts(ratingSorter.thenComparing(priceSorter));
```

- e. Compile and run your application.

Hint: Click a "Run Project" toolbar button.

Note: Observe a list of products ordered by their ratings and prices printed on to the console.

- f. Make another call to the printProducts operation, passing a lambda expression that combines ratingSorter and priceSorter using thenComparing method and reverses the sorting order.

```
pm.printProducts(ratingSorter.thenComparing(priceSorter).reversed());
```

- g. Compile and run your application.

Hint: Click a "Run Project" toolbar button.



Note: Observe a list of products ordered by their ratings and prices in the reverse order printed on to the console.

Practices for Lesson 11: Java Streams API

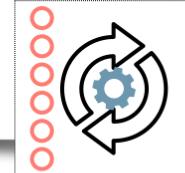
Practices for Lesson 11: Overview

Overview

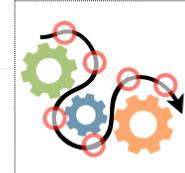
In these practices, you replace all loops that process products and reviews collections with Streams in the ProductManagement class. You also add a method that calculates a total discount per product rating. The result of this calculation should be formatted and assembled into a Map object.

Migrate from Loops to Streams

```
List<Product> productList = new ArrayList<>(products.keySet());
productList.sort(sorter);
for (Product p: productList) {
    txt.append(formatter.formatProduct(p) + '\n')
}
```



```
products.keySet()
    .stream()
    .sorted(sorter)
    .forEach(p -> txt.append(formatter.formatProduct(p) + '\n'));
```



Practice 11-1: Modify ProductManager to Use Streams

Overview

In this practice, you change code in the `ProductManagement` class to use Streams instead of loops to process collections of `Product` and `Review` objects.

Assumptions

- JDK 11 is installed.
- NetBeans 11 is installed.
- You have completed Practice 10 or have started with the Solution for Practice 10 version of the application.

Tasks

1. Prepare the practice environment.

Notes:

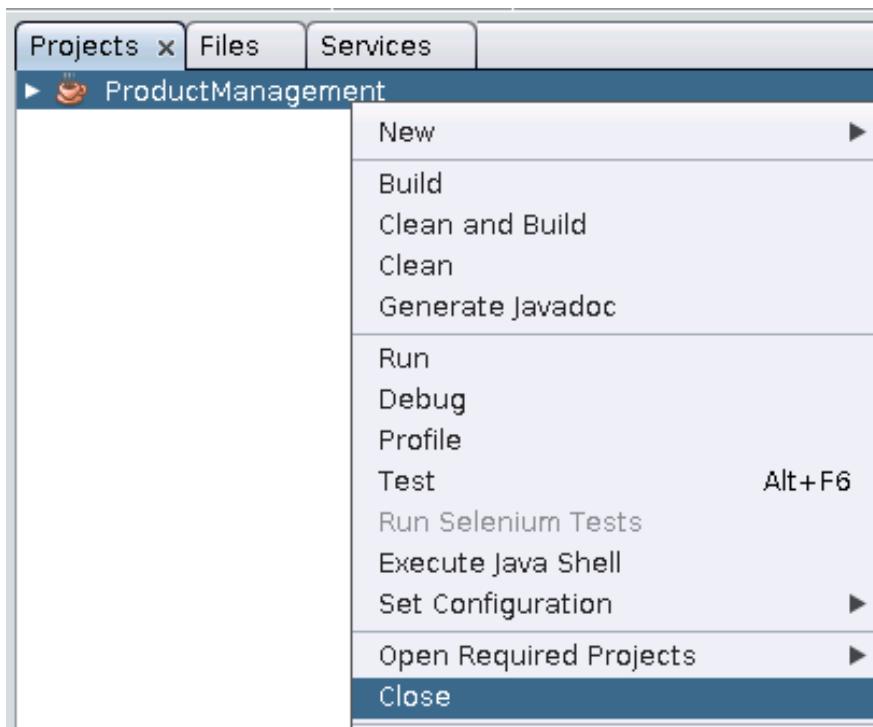
- You may continue to use the same NetBeans project as before, if you have successfully completed the previous practice. In this case, proceed directly to Practice 11-1, step 2.
- Alternatively, you can open a fresh copy of the NetBeans project, which contains the completed solution for the previous practice.

- a. Open NetBeans (if it is not already running).



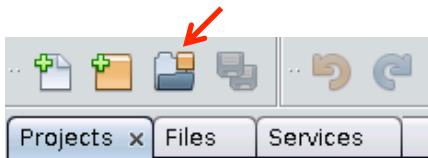
- b. Close the currently opened `ProductManagement` project.

Hint: Right-click on the `ProductManagement` project and invoke "Close" menu.



- c. Open the Solution for Practice 10 version of the ProductManagement project.

Hint: Use File -> Open Project menu or click the "Open Project" toolbar button.



- d. Navigate to the /home/oracle/labs/solutions/practice10 folder and select ProductManagement project.
e. Click "Open Project."
2. Modify logic in the `findProduct` method of the `ProductManagement` class to use Streams.

- a. Open `ProductManager` class editor.
b. Place comments on all existing code inside `findProduct` method body.

Hints:

- Select all lines of code inside `findProduct` method body.
- Press `CTRL+/-` to place comments on these lines.

```
public Product findProduct(int id) {
    // Product result = null;
    // for (Product product : products.keySet()) {
    //     if (product.getId() == id) {
    //         result = product;
    //         break;
    //     }
    // }
    // return result;
}
```

- c. Reproduce the same algorithm as the one you just placed comments on, using streams and lambda expressions.

Hints:

- Place this algorithm just before the commented section inside the `findProduct` method body.
- Use `keySet` method to obtain a Set of Product objects from the `products` Map.
- Use `stream` method to obtain Stream from this Set.
- Use `filter` method to look for product objects with the same id as the method parameter.
- Use Lambda expression that implements the `Predicate` interface to provide the filter condition.
- Use `findFirst` method to find the first element that matches the Predicate condition.
- `findFirst` method returns an `Optional` object.

- Use `orElseGet` method to get this product from the `Optional` object or return `null` if the products is not found.
- Use lambda expression that implements `Supplier` interface to provide the `orElseGet` logic.
- This stream returns either a `Product` with the matching id or `null`, if such a `Product` is not found. Return this value from the `findProduct` method.

```
return products.keySet()
    .stream()
    .filter(p -> p.getId() == id)
    .findFirst()
    .orElseGet(() -> null);
```

Notes:

- The new version of code that uses stream logic and lambda expressions looks shorter and is probably more readable, compared to the old version of code that is now commented out.
 - This approach may also improve performance using parallel stream processing in case you have to handle a very large collection of products.
- d. (Optional) You may remove the commented section of code from the `findProduct` method body.
3. Modify logic in the `reviewProduct` method of the `ProductManagement` class to use Streams.
- Locate the version of the `reviewProduct` method with `Product`, `Rating`, and `String` arguments.
 - Place comments on the section of code that calculates an average rating value for this reviews and applies the new rating to the product.

Hints:

- Select all lines of code starting from the `sum` variable declaration and ending on the `applyRating` method call.
- Press `CTRL+/` to place comments on these lines.

```
// int sum = 0;
// for (Review review : reviews) {
//     sum += review.getRating().ordinal();
// }
// product = product.applyRating(Rateable.convert(
//     Math.round((float) sum / reviews.size())));

```

- c. Reproduce the same algorithm as the one you just placed comments on, using streams and lambda expressions.

Hints:

- Place this algorithm just before the commented section inside the `reviewProduct` method body.
- Use `stream` method to obtain a Stream from the list of `reviews` for the given product.

- Use `mapToInt` method to convert each `Review` object to an `int` value of `Rating`.
- Use `lambda` expression that implements the `ToIntFunction` interface to provide conversion of each review object to `int` value for its `Rating`.
- Use `average` method to calculate the average rating for reviews in the stream.
- `Average` method returns an `OptionalDouble` object.
- Use `orElse` method to get the double value from the `OptionalDouble` object or return `0` if no reviews were present in the stream.
- This stream returns a double number that represents an average rating value.
- Convert this double number to `int` using `Math.round` method and cast returned result to an `int` value.
- Invoke `convert` method provided by the `Rateable` interface to convert the average value of stars into a `Rating` enum value.
- Pass this `Rating` to the `applyRating` method and reassign the `product` object reference.

```
product = product.applyRating(
    Rateable.convert(
        (int) Math.round(
            reviews.stream()
                .mapToInt(r -> r.getRating().ordinal())
                .average()
                .orElse(0))));
```

Notes:

- New version of code that uses stream logic and `lambda` expressions looks shorter and is probably more readable, compared to the old version of code that is now commented out.
 - This approach may also improve performance using a parallel stream processing in case you may have to handle a very large collection of reviews.
- d. (Optional) You may remove the commented section of code from the `reviewProduct` method body.

4. Modify logic in the `printProductReport` method of the `ProductManagement` class to use Streams.
- Locate the version of the `printProductReport` method with `Product` argument.
 - Place comments on the section of code that handles the list of reviews.

Hints:

- Select all lines of this `forEach` loop.
- Press `CTRL+/` to place comments on these lines.

```
// for (Review review : reviews) {
//   txt.append(formatter.formatReview(review));
//   txt.append('\n');
// }
// if (reviews.isEmpty()) {
//   txt.append(formatter.getText("no.reviews"));
//   txt.append('\n');
// }
```

- c. Just before this commented section, create an `if/else` statement that checks if the list of reviews is empty and appends reviews text together with a new line '`\n`' character to the `StringBuilder` object.

```
if (reviews.isEmpty()) {
    txt.append(formatter.getText("no.reviews") + '\n');
} else{
    // review stream handling will be added here
}
```

- d. Inside the `else` block, place an algorithm that produces a string object out of every review, joins these strings together, and appends the result to the `StringBuilder` object, using streams and lambda expressions.

Hints:

- Use `stream` method to obtain a Stream from the `reviews` List.
- Use `map` method to convert each Review into String using `formatReview` method and add a new line '`\n`' character.
- Use `collect` and `Collectors.joining` methods to assemble formatted lines of text together.
- Append the result to the `StringBuilder` object.

```
if (reviews.isEmpty()) {
    txt.append(formatter.getText("no.reviews") + '\n');
} else{
    txt.append(reviews.stream()
        .map(r->formatter.formatReview(r) + '\n')
        .collect(Collectors.joining()));
}
```

Notes:

- New version of code that uses stream logic and lambda expressions looks shorter and is probably more readable, compared to the old version of code that is now commented out.
- This approach may also improve performance using parallel stream processing in case you have to handle a very large collection of reviews.
- Alternatively, you could have written similar algorithm appending text elements in a `forEach` stream method to the mutable `StringBuilder` object. However, this way the logic would not correctly work in parallel stream handling mode.

```
reviews.stream()
    .forEach(r -> txt.append(formatter.formatReview(r) + '\n'));
```

- e. (Optional) You may remove the commented section of code from the `printProductReport` method body.
5. Modify logic in the `printProducts` method of the `ProductManagement` class to use Streams.
- a. Place comments on the section of code inside the `printProducts` method that creates a List of products out of the Set and applies sorter object to this list.

Hints:

- Select lines of code that create the List object and invoke `sort` method.
- Press `CTRL+ /` to place comments on these lines.

```
// List<Product> productList = new ArrayList<>(products.keySet());
// productList.sort(sorter);
```

- b. Place comments on the section of code inside the `printProducts` method that iterates through the list of products.

Hints:

- Select all lines of this `forEach` loop.
- Press `CTRL+ /` to place comments on these lines.

```
// for (Product product : productList) {
//     txt.append(formatter.formatProduct(product));
//     txt.append('\n');
// }
```

- c. Reproduce the same algorithm as the one you just placed comments on, using streams and lambda expressions.

Hints:

- Place this algorithm just after the line of code that creates a `StringBuilder` object inside the `printProducts` method body.
- Use `keySet` method to obtain a Set of Product objects from the `products` Map.
- Use `stream` method to obtain a Stream from this Set.

- Use `sorted` method, passing sorter object as parameter to order the stream.
- Use `forEach` method to append each formatted Product object to the `StringBuilder` and a new line '`\n`' character.

```
products.keySet()
    .stream()
    .sorted(sorter)
    .forEach(p -> txt.append(formatter.formatProduct(p) + '\n'));
```

Notes:

- New version of code that uses stream logic and lambda expressions looks shorter and is probably more readable, compared to the old version of code that is now commented out.
- This algorithm cannot be correctly parallelized, because it requires to maintain the order of text elements that are combined into a single text, and it appends these text elements to a mutable `StringBuilder` object, instead of using `collect` and `Collectors.joining` methods to assemble formatted lines of text together and only then append a result to the `StringBuilder`.

6. Add a `Predicate` parameter to the `printProducts` method and use it to filter the stream content.
 - Add `Predicate` parameter called `filter` to the `printProducts` method. Use `Product` as a generic type for this `Predicate`.

```
public void printProducts(Predicate<Product> filter,
                           Comparator<Product> sorter) {
```

- Add an import statement for the `java.util.function.Predicate` interface.

Hint: Right-click anywhere in the source code of the `ProductManager` class and invoke "Fix Imports" menu.

```
import java.util.function.Predicate;
```

- Add an invocation of the `filter` method to the stream pipeline inside the `printProducts` method. Pass `filter` `Predicate` object as an argument.

```
products.keySet()
    .stream()
    .sorted(sorter)
    .filter(filter)
    .forEach(p -> txt.append(formatter.formatProduct(p) + '\n'));
```

- (Optional) You may remove the commented sections of code from the `printProducts` method body.

7. Test the updated logic from the `main` method of the `Shop` class.

- Open `Shop` class editor.

- b. Uncomment last invocation of the `printProductReports` method (for product with id 106) and place comments on all `printProducts` method invocations except the first one.

Hints:

- Select relevant lines of code inside `main` method body.

```
pm.printProductReport(106);
pm.printProducts((p1,p2)->p2.getRating().ordinal()-p1.getRating().ordinal());
// pm.printProducts((p1,p2)->p2.getPrice().compareTo(p1.getPrice()));
// Comparator<Product> ratingSorter = (p1,p2)->p2.getRating().ordinal()-p1.getRating().ordinal();
// Comparator<Product> priceSorter = (p1,p2)->p2.getPrice().compareTo(p1.getPrice());
// pm.printProducts(ratingSorter.thenComparing(priceSorter));
// pm.printProducts(ratingSorter.thenComparing(priceSorter).reversed());
}
```

- Press `CTRL+ /` to put or remove comments on these lines.
- c. Pass extra parameter to the `printProducts` method that implements a Predicate interface using lambda expression and describes a filtering condition that selects only the products that have a price less than 2.

```
pm.printProducts(p->p.getPrice().floatValue() < 2,
                (p1,p2)->p2.getRating().ordinal()-p1.getRating().ordinal());
```

- d. Compile and run your application.

Hint: Click the "Run Project" toolbar button.



Notes: Observe product details and product reviews printed on the console.

Practice 11-2: Add Discount Per Rating Calculation

Overview

In this practice, you add code to the `ProductManager` class to calculate a total of all discount values for each group of products that have the same rating. The result of this calculation should be formatted and assembled into a `Map` object.

1. Add new operation to the `ProductManager` class to calculate a sum discount value for products that have the same rating.
 - a. Open `ProductManager` class editor.
 - b. Add new `public` method called `getDiscounts` that should return a `Map` object that uses `String` stars value of product rating as a key and calculated discount figure formatted as `String` as a value. This method should accept no arguments. Place this new method definition immediately after the end of the `printProducts` method body.

```
public Map<String, String> getDiscounts() {
    // method logic will be added here
}
```

- c. Add `return` statement inside the `getDiscount` method that is going to return the result of the calculation.

```
public Map<String, String> getDiscounts() {
    return /*products stream processing will be added here*/;
}
```

- d. Add a total sum of discounts per rating calculation logic to compute the value that you return from the `getDiscounts` method.

Hints:

- Use `keySet` method to obtain a `Set` of `Product` objects from the `products` `Map`.
- Use `stream` method to create a stream of `Product` objects.
- Use `collect` method to assemble your calculation results into a `Map`. (You will need to pass two parameters to this `collect` operation—the first one will be a grouping collector to create a map entry per each rating and the second one will be the calculation, followed by formatting of the total discount value for every rating.)
- Use `Collectors.groupingBy` method to group discount values by ratings. Extract the `stars` property from the `rating` of every product to create key value for the results `Map`.
- Use `Collectors.collectingAndThen` method to produce the formatted value of the total discount per rating. (You will need to pass two parameters to this operation—the first one performing the sum discount calculation and the second one to format this discount value.)
- Use `Collectors.summingDouble` method to perform discount calculation, extracting each product discount as a `Double` value.

- Invoke `format` method upon the `moneyFormat` variable available via the `formatter` object reference to format the calculated `discount` value as String.

```
return products.keySet()
    .stream()
    .collect(
        Collectors.groupingBy(
            product -> product.getRating().getStars(),
            Collectors.collectingAndThen(
                Collectors.summingDouble(
                    product -> product.getDiscount().doubleValue(),
                    discount -> formatter.moneyFormat.format(discount))));
```

Notes:

- The `summingDouble` method produces a `Double` value. A `collectingAndThen` operation allows you to add a finisher logic in order to present this value in a nicely formatted way.
- Using streams to implement such a calculation, formatting and data regrouping logic may improve performance by merging a number of data manipulations into a single pass on data and potentially benefiting from the parallel stream processing capabilities in case you may have to handle a very large collection of products.

2. Test new logic from the `main` method of the `Shop` class.
 - a. Open `Shop` class editor.
 - b. Add an invocation of the `getDiscounts` method and print each combination of rating and discount returned from it.

Hints:

- Place this method call immediately after the line of code that invokes `printProductReport` method.
- Use `forEach` method to iterate through all Map entries returned by the `getDiscounts` method.
- You need to write a lambda expression that implements `BiConsumer` interface to handle the key-value pairs for each of the map entries, where the key is the rating and the value is the discount.
- Print each rating value concatenated with tab "\t" character and then concatenated with the value of `discount`.

```
pm.getDiscounts().forEach(
    (rating, discount) -> System.out.println(rating + "\t" + discount));
```

- c. Compile and run your application.

Hint: Click the "Run Project" toolbar button.



Notes:

- Observe discount per rating information printed on the console.
- Not all products may be eligible for discounts, and it is possible that some rating categories will have a total discount of zero.
- The results are not ordered. You may order results, but this may have an adverse effect on the performance when handling very large collection of products using parallel streams mode.

Practices for Lesson 12: Handle Exceptions and Fix Bugs

Practices for Lesson 12: Overview

Overview

In these practices, you change values used in a Shop class to create circumstances in which exceptions will be thrown from ProductManagement class operations. You then write exception handling and propagation code to mitigate these errors. You also write code to parse text, numeric, and date values and handle related exceptions.



Practice 12-1: Use Exception Handling to Fix Logical Errors

Overview

In this practice, you identify potential issues related to the use of erroneous product id value and then fix these issues with appropriate exception handling.

Assumptions

- JDK 11 is installed.
- NetBeans 11 is installed.
- You have completed Practice 11 or start with the solution for Practice 11 version of the application.

Tasks

1. Prepare the practice environment.

Notes:

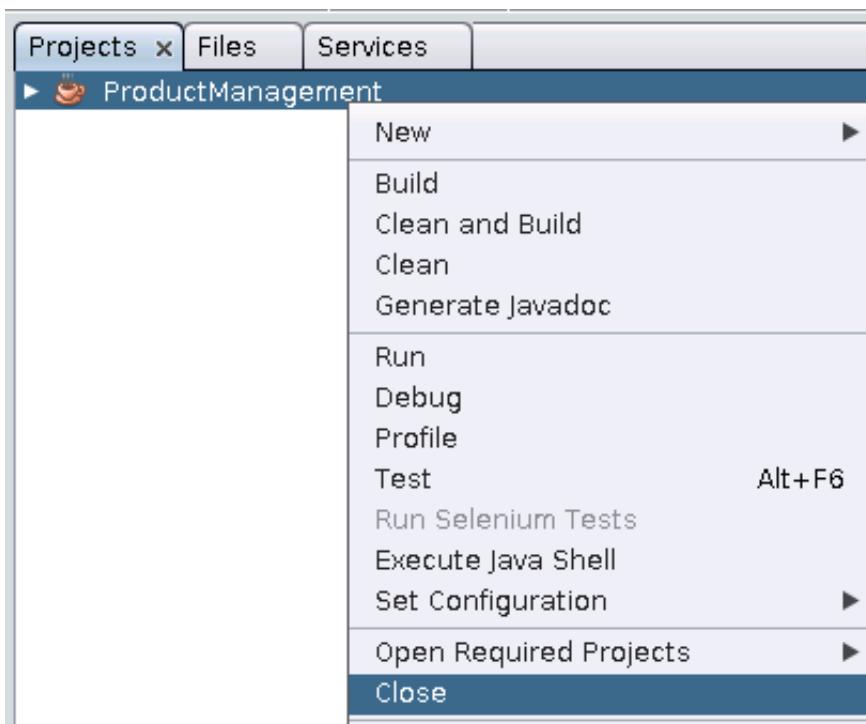
- You may continue to use same NetBeans project as before, if you have successfully completed the previous practice. In this case, proceed directly to Practice 12-1, step 2.
- Alternatively, you can open a fresh copy of the NetBeans project, which contains the completed solution for the previous practice.

- a. Open NetBeans (if it is not already running).



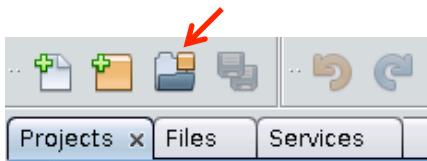
- b. Close the currently opened ProductManagement project.

Hint: Right-click on the ProductManagement project and invoke "Close" menu.



- c. Open solution for Practice 11 version of the ProductManagement project.

Hint: Use File -> Open Project menu or click the "Open Project" toolbar button.



- d. Navigate to /home/oracle/labs/solutions/practice11 folder and select ProductManagement project.
e. Click "Open Project".
2. Make main method of the Shop class request print the product report using non-existent product id.
- Open Shop class editor.
 - Uncomment first invocation of the `printProductReport` method in the `main` method of the `Shop` class.
- Hints:**
- Select relevant line of code inside the `main` method body.
 - Press `CTRL+/-` to remove comments from this line.
- ```
pm.printProductReport(101);
```
- Modify parameter value to a non-existent product id.
  - Compile and run your application.
- Hint:** Click the "Run Project" toolbar button.
- Note:** Observe exception stack trace printed on the console.
- Investigate the case of error by observing the error trace.

```
Exception in thread "main" java.lang.NullPointerException
| at java.base/java.util.Collections.sort(Collections.java:145)
| at labs.pm.data.ProductManager.printProductReport(ProductManager.java:102)
| at labs.pm.data.ProductManager.printProductReport(ProductManager.java:97)
| at labs.pm.app.Shop.main(Shop.java:42)
```

- NullPointerException was produced by the Collections.sort method.
- The actual exception is only produced when your algorithm attempts to use a null reference, trying to sort elements in a non-initialized list of reviews.
- You need to find the reason the reviews object is not properly initialized.
- Variable reviews is null because products.get method returns null.
- products.get method returns null because it receives a null product object as an argument.
- The product object is null because the findProduct method returns null if the product with a given id is not found.
- Therefore, the issue starts with the findProduct method not indicating that the product has not been found.
- Now that the root cause of the NullPointerException is discovered, you need to address the problem.

3. Make `findProduct` method throw an exception and interrupt the rest of this algorithm if the product with a given id is not found.

**Note:** At the moment, that last action in the product stream handling inside the `findProducts` method is a call to `orElseGet` method that returns null value if the product is not found. You may replace this call with either:

- An invocation of the `get` method that throws `NoSuchElementException` exception
- Or an invocation of the `orElseThrow` method that throws any exception you supply

- a. Open `ProductManager` class editor and locate `findProduct` method.
- b. Replace `orElseGet` with the `get` method invocation.

```
return products.keySet()
 .stream()
 .filter(p -> p.getId() == id).findFirst()
 .get();
```

- c. Compile and run your application.

**Hint:** Click the "Run Project" toolbar button.



**Note:** Observe the exception stack trace printed on the console.

- d. Understand error trace

```
Exception in thread "main" java.util.NoSuchElementException: No value present
| at java.base/java.util.Optional.get(Optional.java:148)
| at labs.pm.data.ProductManager.findProduct(ProductManager.java:80)
| at labs.pm.data.ProductManager.printProductReport(ProductManager.java:97)
| at labs.pm.app.Shop.main(Shop.java:42)
```

- `NoSuchElementException` was produced by the `Optional.get` method.
- This indicates the real cause of the problem immediately—no extra investigations are required to pinpoint the cause of the problem.
- However, `NoSuchElementException` is an unchecked exception (it descends from the `RuntimeException` class); thus, you are not required to intercept it or explicitly declare it to be thrown from a method. You may choose to throw checked exception instead to force it to be reported and caught by the invoicing operations.

4. Test other execution paths that lead to an invocation of the `findProduct` method.

**Note:** Method `findProduct` is also invoked from the `reviewProduct` method.

- a. Open `Shop` class editor.
- b. Place comments on the line of code that attempts to print reviews for non-existent product with id 42.

**Hints:**

- Select this line of code inside `main` method body.
- Press `CTRL+ /` to place comments on this line.

```
// pm.printProductReport(42);
```

- c. Modify one of the calls to the `reviewProduct` method to make it use a non-existent product id. For example:

```
pm.reviewProduct(42, Rating.FOUR_STAR, "Nice hot cup of tea");
```

- d. Compile and run your application.



**Hint:** Click the "Run Project" toolbar button.

**Note:** Observe the exception stack trace printed on the console.

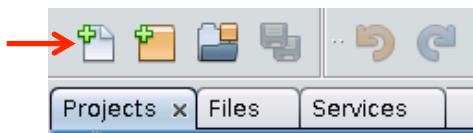
- e. Understand the error trace.

```
Exception in thread "main" java.util.NoSuchElementException: No value present
| at java.base/java.util.Optional.get(Optional.java:148)
| at labs.pm.data.ProductManager.findProduct(ProductManager.java:80)
| at labs.pm.data.ProductManager.reviewProduct(ProductManager.java:84)
| at labs.pm.app.Shop.main(Shop.java:43)
```

- The exact same error in the exact same place has been produced—`NoSuchElementException` was produced by the `Optional.get` method.

5. Create custom exception class.

- a. Create new Java class.



**Hint:** Use File -> New File menu or click the "New File" toolbar button.

- Select "Java" Category and "Java Class" as the file type.
- Click "Next".

- d. Set the following class properties:

- **Class Name:** ProductManagerException
- **Project:** ProductManagement
- **Location:** Source Packages
- **Package:** labs.pm.data



- e. Click "Finish".  
f. Add extends clause to class definition, making class `ProductManagerException` a subclass of `Exception`.

```
package labs.pm.data;
public class ProductManagerException extends Exception {
 // constructor will be added here
}
```

- g. Add three constructors to the `ProductManagerException` class.
- The first constructor should accept arguments.
  - The second should accept a String that represents an error message and pass it to the superclass.
  - The third should accept an error message and a `Throwable` case, which should also be passed to the superclass constructor.

```
public ProductManagerException() {
 super();
}

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

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

6. Produce `ProductManagerException` when product is not found:
- a. Open `ProductManager` class editor.
  - b. Locate `findProduct` method.
  - c. Replace invocation of method `get` with `orElseThrow` method call and provide a new instance of the `ProductManagerException` with a custom error message indicating that a product with specific id is not found

**Hint:** Operation `orElseThrow` accepts a lambda expression (implementing `Supplier` interface) that produces an exception object. You do not need to write `throw` clause yourself because `orElseThrow` method does it for you—all you need to do is supply the actual exception object to throw.

```
return products.keySet()
 .stream()
 .filter(p -> p.getId() == id)
 .findFirst()
 .orElseThrow(() ->
new ProductManagerException("Product with id "+id+" not found"));
```

**Note:** Class `ProductManager` no longer compiles. That is because you now throw a checked exception that must be either caught or be thrown to the invoking method.

- d. Add throws class to the `findProduct` method definition to propagate this exception to any invoking operations.

**Hint:** Right-click on the left side of the line of code that is producing the exception and invoke "Add throws clause for labs.pm.date.ProductManagerException" menu.

The screenshot shows a Java code editor with the following code:

```
public Product findProduct(int id) {
 return products.keySet().stream().filter(p -> p.getId() == id).findFirst().orElseThrow(() ->
 new ProductManagerException("Product with id "+id+" not found"));
}
```

A code completion tooltip is displayed at the end of the `.orElseThrow()` call, listing three suggestions:

- Add throws clause for labs.pm.data.ProductManagerException
- Surround Statement with try-catch
- Surround Block with try-catch

The first suggestion, "Add throws clause for labs.pm.data.ProductManagerException", is highlighted with a red arrow pointing to the "throws" keyword in the tooltip.

## Notes:

- The issue is now shifted to reviewProduct and printProductReport methods, because these are the invokers of the findProduct method.
  - You need to decide if you want to catch this exception in these methods or propagate it further up the invocation chain. This question can be answered by looking at where do you want to interrupt your algorithm and handle this exception. Your options are:
    - To add throws clause to the reviewProduct and printProductReport method definitions to propagate this exception further to the Shop class and place the try-catch block there. In this scenario, a single erroneous product id value would interrupt the entire main method sequence of the Shop class.
    - Or place try-catch blocks into the reviewProduct and printProductReport methods. In this scenario, an erroneous product id value would only interrupt an individual attempt to review or print a non-existent product, not affecting the rest of the Shop main method logic.

- ## 7. Add error handlers to reviewProduct and printProductReport methods

- a. Add try-catch block around the invocation of the `reviewProduct` method, which uses `findProduct` method.

**Hint:** Right-click on the left side of the line of code that is producing the exception in the `reviewProduct` method and invoke "Surround Statement with try-catch" menu.

```
84 public Product reviewProduct(int id, Rating rating, String comments) {
85 return reviewProduct(findProduct(id), rating, comments);
86 }
87 💡 Add throws clause for labs.pm.data.ProductManagerException
88 💡 Surround Statement with try-catch
```

```

public Product reviewProduct(int id, Rating rating, String comments) {
 try {
 return reviewProduct(findProduct(id), rating, comments);
 } catch (ProductManagerException ex) {
 Logger.getLogger(ProductManager.class.getName())
 .log(Level.SEVERE, null, ex);
 }
}

```

**Notes:**

- NetBeans has auto-generated an invocation Logger inside the catch block.
- Operation reviewProduct was designed to return actual Product object after the review has been applied to it. However, class Shop is not actually using this returned object in any way. So you may either return null when exception occurs or make this method void and not return anything.

- b. Add return null clause to the catch block inside the reviewProduct method.

```

try {
 return reviewProduct(findProduct(id), rating, comments);
} catch (ProductManagerException ex) {
 Logger.getLogger(ProductManager.class.getName())
 .log(Level.SEVERE, null, ex);
 return null;
}

```

- c. Add try-catch block around the invocation of the printProductReport method, which uses findProduct method

**Hint:** Right-click on the left side of the line of code that is producing the exception in the printProductReport method and invoke "Surround Statement with try-catch" menu.

```

public void printProductReport(int id) {
 try {
 printProductReport(findProduct(id));
 } catch (ProductManagerException ex) {
 Logger.getLogger(ProductManager.class.getName())
 .log(Level.SEVERE, null, ex);
 }
}

```

8. Customize Logger setup and use in the ProductManager class.

- a. Define Logger as a class scope content in the ProductManager class.

**Hints:**

- Add a private static final constant called logger type of Logger to the ProductManager class.
- Initialize this variable to reference a logger using ProductManager class name.
- Add this constant after the declaration and initialization of the static variable called formatters.

```
private static final Logger logger =
 Logger.getLogger(ProductManager.class.getName());
```

- b. Customize Logger behaviors in both `reviewProduct` and `printProductReport` methods.

**Notes:**

- The error you are handling is ultimately caused by the wrong product id value supplied by the Shop app. It should not be considered as a severe problem. All your application needs to do is not to attempt to add reviews or print non-existent products.
- You may also consider customizing the amount of details recorded by the Logger. You should print your exception error message, but there is no need to actually print the entire stack trace for such a trivial error, unless you are still debugging your code.

**Hints:**

- Change the Logger level to INFO in both `reviewProduct` and `printProductReport` methods.
- Modify Logger parameters in both `reviewProduct` and `printProductReport` methods to use an exception error message instead of null.
- Remove the detailed exception trace printing (the last parameter) from both log method invocations.
- Use class-level constant that references a logger object.

```
logger.log(Level.INFO, ex.getMessage());
```

9. Test the updated logic from the `main` method of the `Shop` class.

- a. Open `Shop` class editor.  
b. Uncomment the first invocation of the `printProductReports` method (for product with id 42) and the second call to the `printProductReport` method (with id 101).

**Hints:**

- Select relevant lines of code inside `main` method body.
- Press `CTRL+ /` to add or remove comments on these lines.

```
ProductManager pm = new ProductManager("en-GB");
pm.createProduct(101, "Tea", BigDecimal.valueOf(1.99),
 Rating.NOT_RATED);
pm.printProductReport(42);
pm.reviewProduct(42, Rating.FOUR_STAR, "Nice hot cup of tea");
pm.reviewProduct(101, Rating.TWO_STAR, "Rather weak tea");
pm.reviewProduct(101, Rating.FOUR_STAR, "Fine tea");
pm.reviewProduct(101, Rating.FOUR_STAR, "Good tea");
pm.reviewProduct(101, Rating.FIVE_STAR, "Perfect tea");
pm.reviewProduct(101, Rating.THREE_STAR, "Just add some lemon");
pm.printProductReport(101);
```

- c. Place comments on all remaining lines of code inside main method so that you'll only see the output generated by the handling of the first product and associated reviews and error messages related to the use of wrong product id values.

**Hints:**

- Select relevant lines of code inside `main` method body.
- Press `CTRL+/` to add comments to these lines.

- d. Compile and run your application.

**Hint:** Click the "Run Project" toolbar button.



**Notes:**

- By default, logger is configured to print messages to the console.
  - Observe the error messages as well as the product details and product reviews printed on the console.
  - Individual attempts to review or print non-existent products now fail, but the rest of program logic is not affected.
- e. Apply correct values—replace product id 42 with 101 in both `printProductReport` and `reviewProduct` methods.

## Practice 12-2: Add Text Parsing Operations

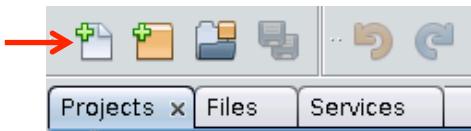
### Overview

In this practice, you add code to the ProductManager class to parse Strings and extract information to create Product and Review objects. Your code should be designed to parse text values as if you've loaded them from the delimited files. Assume that data is presented in a non-locale-specific, portable format. Parse operations can throw various exceptions related to text, numeric, and date format issues. Your task is to handle these different exceptions in a consistent fashion.

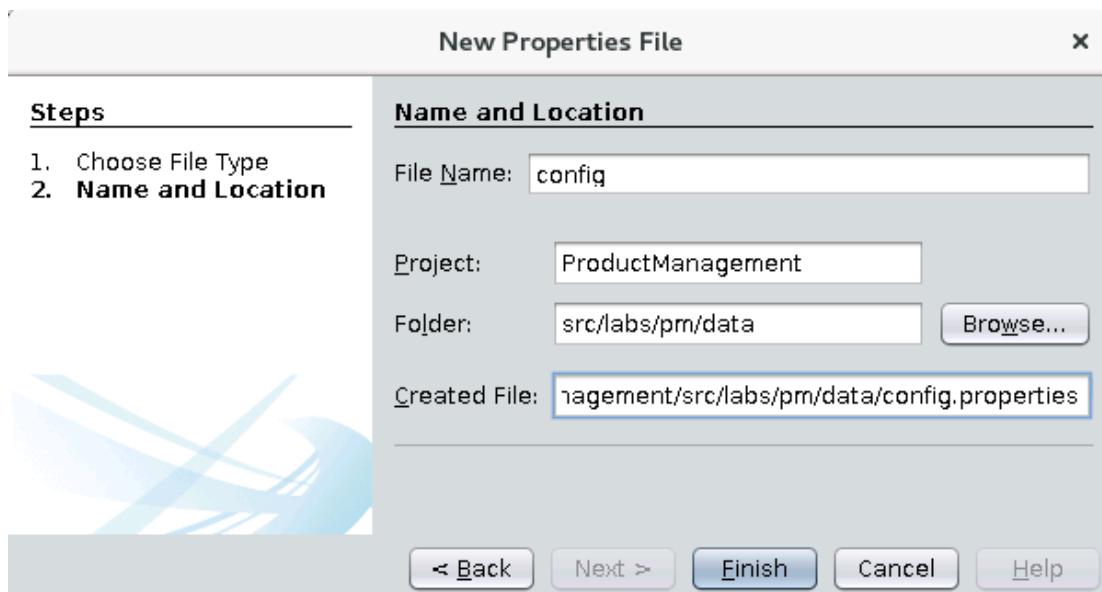
1. Create configuration file for storing program settings.

- a. Create new property file called config.

**Hints:**



- Use File -> New File menu or click the "New File" toolbar button.
- Select "Other" Category and "Properties File" as the file type.
- Click "Next".
- Set the following class properties:
  - File Name: config
  - Project: ProductManagement
  - Folder: src/labs/pm/data



- Click "Finish".

- b. Add two properties to the `config.properties` file, to represent format patterns for the product and review, intended to be loaded from delimited files.

```
product.data.format={0},{1},{2},{3},{4},{5}
review.data.format={0},{1},{2}
```

**Notes:**

- The review pattern represents the following structure of text elements:
  - Product Id (integer number)
  - Review Rating (integer number)
  - Review comments (text)
 For example: "101,4,Nice hot cup of tea"
- The product pattern represents the following structure of text elements:
  - Product Type (F for Food or D for Drink)
  - Product Id (integer number)
  - Product Name (text)
  - Product Price (floating-point number)
  - Product Rating (integer number)
  - Product Best Before Date (date in ISO format: yyyy-mm-dd)
 For example: "D,101,Tea,1.99,0,2019-09-19"
 or: "F,103,Cake,3.99,0,2019-09-19"
- Java Object-Oriented features such as inheritance or polymorphism are not applicable to such simple delimited text data. Therefore, you have to explicitly indicate a product type as part of the text data.
- Simple comma-delimited format may not be suitable to handle numeric or date values formatted using specific locales, because comma can be a part of the value itself.
- Message format class understands more sophisticated patterns, for example, {3, number, ###.##} for setting up a numeric pattern or {5, date} for fields that should represent dates. However, in this practice you are instructed to extract all values as simple text and then apply numeric, date, etc. parsing to these values.

2. Load configuration file and initialize `MessageFormat` objects within the `ProductManager` class.
- a. Open `ProductManager` class editor.
  - b. Add new instance variable called `config` and initialize it to reference your config file represented as `ResourceBundle`.

**Hints:**

- Add this new variable just before the declaration of the formatter variable.
- Use `getBundle` method of the `ResourceBundle` class to initialize this config variable.
- Reference config file using full package prefix.
- Make this variable private.

```
private ResourceBundle config =
 ResourceBundle.getBundle("labs.pm.data.config");
```

**Notes:**

- This configuration file is intended to be used in a locale-independent way.
  - `java.util.ResourceBundle` class treats it as a part of your program, i.e., loads this config from the Java class-path.
  - An alternative design of the configuration file is to use `java.util.Properties` class, which reads property files from any stream, usually a file located somewhere on your filesystem.
- c. Add two new instance variables called `productFormat` and `reviewFormat` both types of `MessageFormat` class and initialize these variables to contain review and product format patterns retrieved from the config file.

**Hints:**

- Add these new variables after the declaration of the config variable.
- Use `getString` method of the `ResourceBundle` class to initialize each of the `MessageFormat` instances.
- Make these variables private.

```
private MessageFormat reviewFormat =
 new MessageFormat(config.getString("review.data.format"));
private MessageFormat productFormat =
 new MessageFormat(config.getString("product.data.format"));
```

3. Add an operation to the `ProductManager` class that parses a comma-separated text and constructs a `Review` object using parsed values.
- a. Open `ProductManager` class editor.
  - b. Add `public` method called `parseReview` immediately after the end of the `printProducts` method body. This new method should accept `String` parameter and does not need to return any value.

```
public void parseReview(String text) {
 // review parsing logic will be added here
}
```

- c. Inside the `parseReview` method, declare a new variable type of `Object` array called `values`. Invoke `parse` method upon the `reviewFormat` object, passing `text` parameter to it. Assign the result returned by the `parse` method to the `values` variable.

```
Object[] values = reviewFormat.parse(text);
```

**Notes:**

- You have now extracted three expected elements from the text as an array of objects.
- You will have cast each array value to `String` and then converted it to the relevant Java type.
- `Parse` method of the `MessageFormat` throws `ParseException`; this is a checked exception and must be either caught or declared to be thrown to the invoker explicitly.

- d. Add try-catch block around the invocation of the `parse` method

**Hint:** Right-click on the left side of the line of code that is producing the exception in the `parseReview` method and invoke "Surround Statement with try-catch" menu

```
try {
 Object[] values = reviewFormat.parse(text);
 // parse values and create review object
} catch (ParseException ex) {
 Logger.getLogger(ProductManager.class.getName())
 .log(Level.SEVERE, null, ex);
}
```

**Notes:**

- If you have followed the hint for this practice step, NetBeans has not only generated the required try-catch block but also added an import of the `ParseException` class.

- e. Customize Logger behavior in the `parseReview` method by changing the severity level to warning and supplying a short descriptive error message.

**Hints:**

- Change logger level to `WARNING`.
- Modify Logger parameter to use a concatenation of the "Error parsing review:" string and the `text` parameter instead of `null`.
- Use class-level constant that references a logger object.

```
logger.log(Level.WARNING, "Error parsing review "+text, ex);
```

- f. Create new review object using values extracted from the text.

**Hints:**

- Invoke operation `reviewProduct` and pass the following parameters:
  - First parameter is an `int` value of the product id.
    - Cast the first element in the `values` array to `String`.
    - Use method `parseInt` of the `Integer` class to convert this value to an `int`.
  - Second parameter is a `Rating` value.
    - Cast the first element in the `values` array to `String`.
    - Use `parseInt` method of the `Integer` class to convert this value to an `int`.
    - Use `convert` method of the `Rateable` interface to convert this value to `Rating` object.
  - Third parameter is a `String` value of the review comments.
    - Cast the first element in the `values` array to `String`.
- Place this code immediately after the invocation of the `parse` method inside the try block.

```
reviewProduct(Integer.parseInt((String)values[0]),
 Rateable.convert(Integer.parseInt((String)values[1])),
 (String)values[2]);
```

4. Test the updated logic from the `main` method of the `Shop` class.

- Open `Shop` class editor.
- Set up a test scenario for the `parseReview` method.

**Hints:**

- Replace invocation of the first `reviewProduct` method with `parseReview`.
- Use the following text as parameter: "101,4,Nice hot cup of tea".
- Apply comments to all other invocations of the `reviewProduct` method for the product 101.
- Select relevant lines of code inside `main` method body.
- Press `CTRL+ /` to add comments to these lines.

```
ProductManager pm = new ProductManager("en-GB");
pm.createProduct(101, "Tea", BigDecimal.valueOf(1.99),
 Rating.NOT_RATED);
pm.printProductReport(101);
pm.parseReview("101,4,Nice hot cup of tea");
// pm.reviewProduct(101, Rating.TWO_STAR, "Rather weak tea");
// pm.reviewProduct(101, Rating.FOUR_STAR, "Fine tea");
// pm.reviewProduct(101, Rating.FOUR_STAR, "Good tea");
// pm.reviewProduct(101, Rating.FIVE_STAR, "Perfect tea");
// pm.reviewProduct(101, Rating.THREE_STAR, "Just add some lemon");
pm.printProductReport(101);
```

- Compile and run your application.



**Hint:** Click the "Run Project" toolbar button.

**Note:** Observe the product details and product reviews printed on the console.

- Trigger `ParseException` by providing erroneous text value for the `parseReview` method invocation.

**Hints:** Use the following text as parameter: "1014,Nice hot cup of tea"—it is missing a comma and thus would not match the expected pattern.

```
pm.parseReview("1014,Nice hot cup of tea");
```

- Compile and run your application.



**Hint:** Click the "Run Project" toolbar button.

**Note:** Observe the error information printed on the console.

- f. Trigger `NumberFormatException` by providing an erroneous text value for the `parseReview` method invocation.

**Hints:**

- Use the following text as parameter: "101,X,Nice hot cup of tea".
- The overall format of this message is fine.
- However, one of the values that is expected to be an int has a value of X:  
`pm.parseReview("101,x,Nice hot cup of tea");`

- g. Compile and run your application.



**Hint:** Click the "Run Project" toolbar button.

**Notes:**

- Observe error information printed on the console.
- The actual exception that was produced was a `NumberFormatException`.
- Compiler did not force you to provide a catch block for it, because it is an unchecked exception.
- Consider catching this exception anyway to allow your program to handle the erroneous number format and allow the rest of your program to proceed.

5. Add `NumberFormatException` handling logic to the `parseReview` method/
- a. Open `ProductManager` class editor.
  - b. Locate `catch(ParseException ex)` statement inside `parseReview` method.
  - c. Allow `NumberFormatException` to be handled by the same catch block as the one that is already handling the `ParseException`.

**Hint:** Use `|` operator to add extra catch parameter.

```
catch (ParseException | NumberFormatException ex)
```

**Notes:**

*The following notes (including code examples) are not actual instructions that you should perform as part of this practice. Their purpose is to explain possible design improvements of the error handling process.*

- Capturing several exceptions in the same catch has become possible in Java since version 7. In earlier versions, you had to write separate catch blocks—one per exception.
- You can still provide separate catch blocks if you like. However, this is the choice you can make based on whenever you want to perform the same or different exception handling actions.
- You may also wish to report a problem to the invoker, such as a `Shop` class. You may capture a number of relevant exceptions in the `ProductManager` class and then throw your custom exception to the invoker instead.
  - To achieve that, use extra `Throwable` parameter of the `ProductManagerException` constructor, to carry information about the cause of the error.
  - Capture any number of exceptions within a given operation.

- Create an instance of your custom exception passing an error message and an original cause of the problem as parameters to your exception class constructor.
- Throw this custom exception to the invoker.

```
public void parseReview(String text)
 throws ProductManagerException {
 try {
 // preform required actions
 } catch (ParseException | NumberFormatException ex) {
 // perform required error handling
 throw
 new ProductManagerException("Unable to parse review",ex);
 }
}
```

- This way the invoker such as a Shop class would only have to capture one exception type, yet it would be able to find the original cause for the problem if required.

```
try {
 pm.parseReview("whatever text to parse");
} catch (ProductManagerException ex) {
 Throwable cause = ex.getCause();
}
```

*This is the end of the notes section.*

d. Compile and run your application.

**Hint:** Click the "Run Project" toolbar button.



**Notes:**

- Observe error information printed on the console.
- All exceptions are now caught within the `parseReview` method, so the Shop application may proceed to other actions, even though a specific review has failed to be applied.

6. Clean up test code.

a. Open `Shop` class editor.

b. Clean up test code in the `main` method.

**Hints:**

- Replace erroneous text with "101,4,Nice hot cup of tea" value.
- Remove comments to all other invocations of the `reviewProduct` method for the product 101.
- Select relevant lines of the code inside `main` method body.

- Press **CTRL+ /** to add comments to these lines.
- Replace these calls with invocations of the `parseReview` method, passing appropriate text as an argument for each of these calls.

```
ProductManager pm = new ProductManager("en-GB");
pm.createProduct(101, "Tea", BigDecimal.valueOf(1.99),
 Rating.NOT_RATED);
pm.printProductReport(101);
pm.parseReview("101,4,Nice hot cup of tea");
pm.parseReview("101,2,Rather weak tea");
pm.parseReview("101,4,Fine tea");
pm.parseReview("101,4,Good tea");
pm.parseReview("101,5,Perfect tea");
pm.parseReview("101,3,Just add some lemon");
pm.printProductReport(101);
c. Open ProductManager class editor.
d. Locate parseReview method.
e. Remove last parameter (printing detailed stack trace) from the logger invocation and add exception error message to the printed text.
logger.log(Level.WARNING,
 "Error parsing review "+text+" "+ex.getMessage());
```

7. Add an operation to the `ProductManager` class that parses a comma-separated text and constructs a `Product` object using parsed values.

- a. Open `ProductManager` class editor.
- b. Add `public` method called `parseProduct` immediately after the end of the `parseReview` method body. This new method should accept `String` parameter and return does not need to return any value.

```
public void parseProduct(String text) {
 // product parsing logic will be added here
}
```

- c. Inside the `parseProduct` method declare a new variable type of `Object` array called `values`. Invoke method `parse` upon the `productFormat` object, passing `text` parameter to it. Assign the result returned by the `parse` method to the `values` variable.

```
Object[] values = productFormat.parse(text);
```

**Notes:**

- You have now extracted six elements from the text as an array of objects.
- You will have cast each array value to `String` and then converted it to the relevant Java type.
- Parse method of the `MessageFormat` throws `ParseException`. This is a checked exception and must be either caught or declared to the thrown to the invoker explicitly.

- d. Add a try-catch block around the invocation of the parse method that catches ParseException, NumberFormatException, and DateTimeParseException using the same handler. This exception handler should produce WARNING level log message with the text that indicates the nature of the error, a value of the text that was parsed, and an exception error message

**Hints:**

- Right-click on the left side of the line of code that is producing the exception in the parseProduct method and invoke "Surround Statement with try-catch" menu.
- Modify logger logic to use logger constant, produce WARNING level message, and format the error message.

```
try {
 Object[] values = productFormat.parse(text);
 // parse values and create product object
} catch (ParseException |
 NumberFormatException |
 DateTimeParseException ex) {
 logger.log(Level.WARNING,
 "Error parsing product "+text+" "+ex.getMessage());
}
```

- e. Add an import statement for the java.time.format.DateTimeParseException class.

**Hint:** Right-click anywhere in the source code of the ProductManager class and invoke "Fix Imports" menu.

```
import java.time.format.DateTimeParseException;
```

- f. Inside the try block, extract int id value from the second element in the values array.

**Hints:**

- Declare int id variable.
- Cast the second element in the values array to String.
- Use parseInt method of the Integer class to convert its value to int and assign the result of this conversion to the id variable.

```
int id = Integer.parseInt((String)values[1]);
```

- g. Next, extract String name value from the third element in the values array.

**Hints:**

- Declare String name variable.
- Cast the third element in the values array to String and assign it the name variable.

```
String name = (String)values[2];
```

- h. Next, extract `BigDecimal` price value from the fourth element in the values array.

**Hints:**

- Declare `BigDecimal` price variable.
- Cast the fourth element in the values array to `String`.
- Use `parseDouble` method of the `Double` class to convert `String` value to `double`.
- Use `valueOf` method of the `BigDecimal` class to convert `double` value to `BigDecimal`.
- Assign the result of the conversion to the `price` variable.

```
BigDecimal price =
 BigDecimal.valueOf(Double.parseDouble((String)values[3]));
```

- i. Next, extract Rating value from the fifth element in the values array.

**Hints:**

- Declare the Rating rating variable.
- Cast the fifth element in the values array to `String`.
- Use `parseInt` method of the `Integer` class to convert its value to `int`.
- Use `convert` method of the `Rateable` interface to convert `int` value to `Rating` enum value.
- Assign the result of the conversion to the `rating` variable.

```
Rating rating =
 Rateable.convert(Integer.parseInt((String)values[4]));
```

- j. Create switch construct that tests the String value of the first element in the values array and expects two cases "D" and "F", indicating if product is a Drink or a Food.

```
switch ((String)values[0]) {
 case "D":
 // add code to create drink object
 case "F":
 // add code to parse best before date and create food object
}
```

**Note:** ProductManager application assumes that products type of Food has a specific expiry date and products type of Drink are assumed to expire on the same day. Therefore, the best before date for the Drink can be ignored.

- k. Inside the "D" case, invoke `createProduct` operation, passing `id`, `name`, `price`, and `rating` parameters. After that, add a `break` statement.

```
createProduct(id, name, price, rating);
break;
```

- I. Inside the "F" case, extract LocalDate bestBefore value from the sixth element in the values array.

**Hints:**

- Declare the LocalDate bestBefore variable.
- Cast the sixth element in the values array to String.
- Use parse method of the LocalDate class to convert this String value to LocalDate.
- Assign the result of the conversion to the bestBefore variable.

```
case "F":
 LocalDate bestBefore = LocalDate.parse((String)values[5]);
 // next you add code to create food object
 m. Next, invoke createProduct operation, passing id, name, price, rating, and
 bestBefore parameters.
 createProduct(id, name, price, rating, bestBefore);
```

8. Test the updated logic from the main method of the Shop class.

- a. Open Shop class editor.
- b. Set up a test scenario for the parseProduct method.

**Hints:**

- Replace invocation of the first createProduct method with the parseProduct.
- Use the following text as parameter: "D,101,Tea,1.99,0,2019-09-19"  
pm.parseProduct("D,101,Tea,1.99,0,2019-09-19");

- c. Compile and run your application.

**Hint:** Click the "Run Project" toolbar button. 

**Note:** Observe product and review details printed on the console.

- d. Trigger ParseException by providing erroneous text value for the parseProduct method invocation.

**Hints:** Use the following text as parameter: "D,101,Tea,1.99,0"—it is missing a value and thus does not match the expected pattern.

```
pm.parseProduct("D,101,Tea,1.99,0");
```

- e. Compile and run your application.

**Hint:** Click the "Run Project" toolbar button. 

**Notes:**

- Observe error information printed on the console.
- The Product object was not actually created because of ParseException.
- However, because this exception has been caught inside the parse method, the rest of the program continued to run.
- As a result, your application tried to apply reviews and print product reports for the products that were never created in the first place.
- Error-handling code managing this kind of eventuality, using ProductManagerException, was added to the ProductManager class in the previous practice (12-1).

- f. Trigger `DateTimeParseException` by providing erroneous text value for the `parseProduct` method invocation.

**Hints:**

- Add additional invocation of the `parseProduct` method using the following parameters:
- It should be a data set that triggers the creation of `Food` object.
- It should use impossible date value "D,101,Tea,1.99,0,2019-09-49"  
`pm.parseProduct("F,103,Cake,3.99,0,2019-09-49");`

- g. Compile and run your application.



**Hint:** Click the "Run Project" toolbar button.

**Note:** Observe error information printed on the console.

- h. Clean up test code in the `main` method.

**Hints:**

- Replace erroneous text with "F,103,Cake,3.99,0,2019-09-19" value.

```
ProductManager pm = new ProductManager("en-GB");
pm.parseProduct("D,101,Tea,1.99,0,2019-09-19");
pm.printProductReport(101);
pm.parseReview("101,4,Nice hot cup of tea");
pm.parseReview("101,2,Rather weak tea");
pm.parseReview("101,4,Fine tea");
pm.parseReview("101,4,"Good tea");
pm.parseReview("101,5,Perfect tea");
pm.parseReview("101,3,Just add some lemon");
pm.printProductReport(101);
pm.parseProduct("F,103,Cake,3.99,0,2019-09-19");
```

- i. Compile and run your application.



**Hint:** Click the "Run Project" toolbar button.

**Note:** Observe the product and review details printed on the console.

## **Practices for Lesson 13: Java IO API**

## Practices for Lesson 13: Overview

---

### Overview

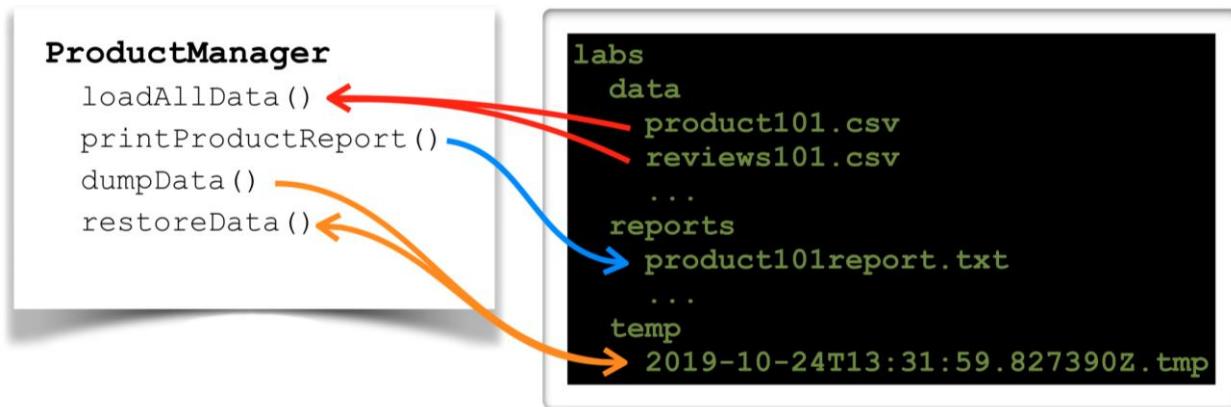
In these practices, you write products and reviews reports to text files, bulk-load data from comma-delimited files, as well as serialize and deserialize Java objects.

You are not very likely to use such primitive form of data storage as a comma-delimited file. Many modern Java programs use XML or JSON data formats and of course may use databases to store information. All of these approaches are much more consistent and practical and require you to write a way less code compared to the comma-delimited format used in this practice.

You can get more information on these topics from:

"Appendix B: JDBC" of this course, which covers basic database access

"Developing Applications for the Java EE 7 Platform" course that covers database access using JPA API, as well as XML and JSON mappings API.



## Practice 13-1: Print Product Report to a File

### Overview

In this practice, you modify printProductReport method to produce text files containing formatted products and reviews.

### Assumptions

- JDK 11 is installed.
- NetBeans 11 is installed.
- You have completed Practice 12 or have started with the solution for Practice 12 version of the application tasks.

#### 1. Prepare the practice environment.

##### Notes:

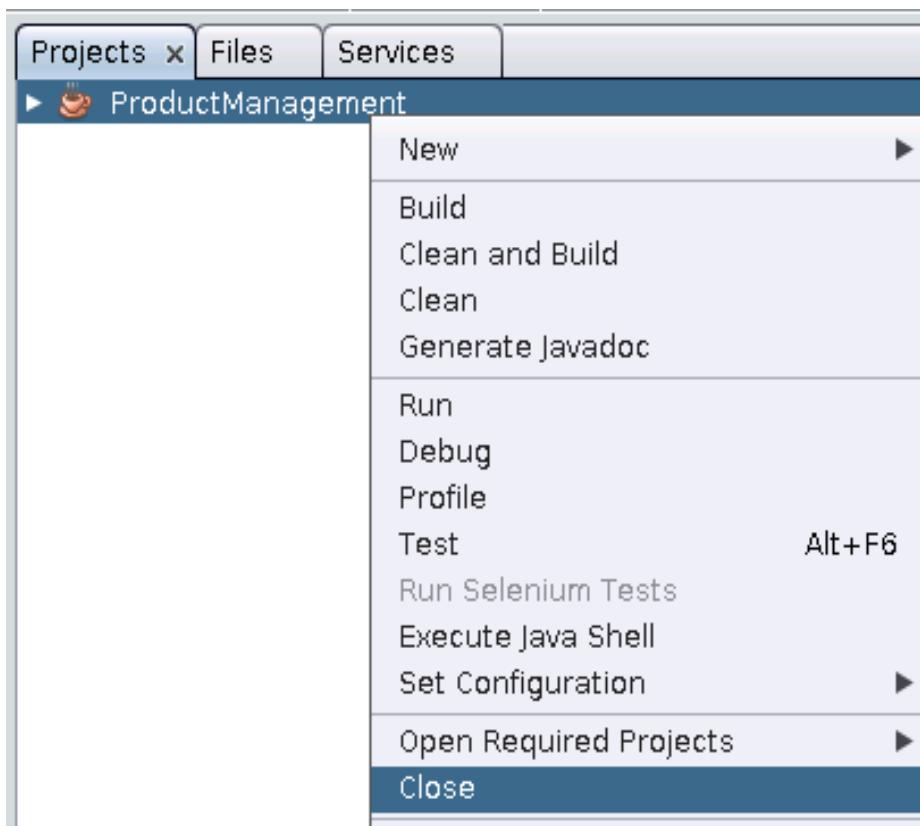
- You may continue to use same NetBeans project as before, if you have successfully completed the previous practice. In this case, proceed directly to Practice 13-1, step 2.
- Alternatively, you can open a fresh copy of the NetBeans project, which contains the completed solution for the previous practice.

- a. Open NetBeans (if it is not already running).

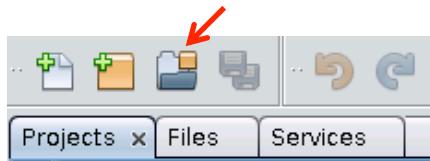


- b. Close the currently opened ProductManagement project.

**Hint:** Right-click on the ProductManagement project and invoke "Close" menu.



- c. Open the solution for Practice 12 version of the ProductManagement project.  
**Hint:** Use File -> Open Project menu or click the "Open Project" toolbar button.



- d. Navigate to /home/oracle/labs/solutions/practice12 folder and select ProductManagement project.  
e. Click "Open Project".
2. Modify configuration properties file.
- Open config.properties file.
  - Add three properties that represent different paths that you will use during this practice. These properties should be configured as following:  
**Hints:**
    - Property reports.folder set to /home/oracle/labs/reports value
    - Property data.folder set to /home/oracle/labs/data value
    - Property temp.folder set to /home/oracle/labs/temp value

```
reports.folder=/home/oracle/labs/reports
data.folder=/home/oracle/labs/data
temp.folder=/home/oracle/labs/temp
```
  - Add three properties that represent filename formats for report files, data files, and temporary files. These properties should be configured as following:  
**Hints:**
    - Property report.file set to product{0}report.txt value
    - Property product.data.file set to product{0}.csv value
    - Property reviews.data.file set to reviews{0}.csv value
    - Property temp.file set to {0}.tmp value

```
report.file=product{0}report.txt
product.data.file=product{0}.csv
reviews.data.file=reviews{0}.csv
temp.file={0}.tmp
```

**Note:** Substitution parameters will be used to incorporate relevant identity values into these filenames.
3. Prepare ProductManager class to access filesystem paths required for this set of practices.
- Open ProductManager class editor.

- b. Add three new instance variables type of `Path` called `reportsFolder`, `dataFolder`, and `tempFolder` to reference a path where report files will be created.

**Hints:**

- Use `getString` method of the config resource bundle to retrieve value to initialize these variables using "reports.folder", "data.folder", and "temp.folder" properties.
- Place these new variable definitions just after the declaration of the `productFormat` variable.
- Make these variables private.

```
private Path reportsFolder =
 Path.of(config.getString("reports.folder"));
private Path dataFolder =
 Path.of(config.getString("data.folder"));
private Path tempFolder =
 Path.of(config.getString("temp.folder"));
```

- c. Add an import statement for the `java.nio.file.Path` class.

**Hint:** Right-click anywhere in the source code of the `ProductManager` class and invoke "Fix Imports" menu.

```
import java.nio.file.Path;
```

4. Modify `printProductReport` method of the `ProductManager` class to print product report to a file instead of the console.
- Locate a version of the `printProductReport` method that accepts `Product` object as an argument.
  - Replace code at the line code that declares a `StringBuilder` object with an initialization of a new `Path` variable called `productReportFile`, which represents a file where product report will be written.

**Hints:**

- Remove code that created and initialized `StringBuilder` variable.
- In its place, declare a new variable called `productFile` type of `Path` class.
- Initialize this variable using `resolve` method to combine a `reportsFolder` path with the filename for this product report.
- Produce a filename for this product report using a `MessageFormat.format` method, passing a value of "report.file" key from the config file and `product id` as parameters.

```
Path productFile =
reportsFolder.resolve(
MessageFormat.format(
config.getString("report.file"), product.getId())
);
```

**Note:** Ignore errors displayed by NetBeans, informing you that the `StringBuilder` txt variable is no longer defined. These issues will be fixed soon.

- c. Next, add a `try-with-parameters` block, which should be placed around the remaining lines of code in the `printProductReport` method.

```
try /* output stream initialisation will be added here */ {
 txt.append(formatter.formatProduct(product));
 txt.append('\n');
 if (reviews.isEmpty()) {
 txt.append(formatter.getText("no.reviews") + '\n');
 } else {
 txt.append(reviews.stream()
 .map(r -> formatter.formatReview(r) + '\n')
 .collect(Collectors.joining())));
 }
 System.out.println(txt);
}
```

- d. Inside the parameter section of the try block, describe an output stream that will be used to write to the file.

**Hints:**

- Define the new variable called `out` type of `PrintWriter` and initialize this variable to reference a new `PrintWriter` object.
- Pass a new instance of `OutputStreamWriter` object as an argument to the `PrintWriter` constructor.
- Initialize `OutputStreamWriter` to reference an output stream that is referencing `productFile` Path object and the "UTF-8" character encoding for this file.
- Use `newOutputStream` method of the `Files` class to create an output stream pointing to the required file.
- Pass a reference to the required `Path` object and a `CREATE` `StandardCopyOption` enum value to the `newOutputStream` method.

```
try (PrintWriter out = new PrintWriter(
 new OutputStreamWriter(
 Files.newOutputStream(productFile,
 StandardOpenOption.CREATE),
 "UTF-8"))) {
 // remaining logic of the try block
}
```

**Note:** Specifying the `StandardOpenOption.CREATE` option allows a file to be created and overwritten if required.

- e. Add an import statement for classes and enums used in this method.

**Hint:** Right-click anywhere in the source code of the `ProductManager` class and invoke "Fix Imports" menu.

```
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.nio.file.Files;
import java.nio.file.StandardOpenOption;
```

- f. Inside the try block body, adjust the code to use `PrintWriter` instead of `StringBuilder`.

**Hints:**

- Replace all references to the `txt` variable with references to the `out` variable.
- Use system-specific line separator instead of '`\n`' character.
- Remove the line of code that prints `txt` object to the console.

```
try (PrintWriter out = ...) {
 out.append(formatter.formatProduct(product)
 +System.lineSeparator());
 if (reviews.isEmpty()) {
 out.append(formatter.getText("no.reviews")
 +System.lineSeparator());
 } else {
 out.append(reviews.stream()
 .map(r -> formatter.formatReview(r)
 +System.lineSeparator())
 .collect(Collectors.joining())));
 }
}
```

**Note:** Operation `System.lineSeparator()` returns platform-specific codes for ending a line of text. The ending of the line of text is controlled using two symbols:

- Carriage Return (CR): Java escape sequence '`\r`', ASCII code '`\015`'
- or Line Feed (LF): Java escape sequence '`\n`', ASCII code '`\012`'
- UNIX-based systems use LF and Windows CR+LF.

It is recommended to use `System.lineSeparator()` when writing files to automatically pick up correct current OS style of managing new lines of text.

- g. Add throws clause to the printProductReport method to propagate IOException to the invoker.

**Hint:** Right-click on the left side of the line of code that is producing the IOException and invoke "Add throws clause for java.io.IOException" menu.

```
public void printProductReport(Product product) throws IOException {
```

**Notes:**

- Inside the printProductReport method, you are using try-with-resources construct, which adds an implicit finally block to the try block.
- Although IOException must be caught, because it is a checked exception, it does not mean that it has to be caught immediately.
- It is possible to write a try-finally or try-with-resources constructs without immediately adding catch blocks, but propagating exceptions to the invoker instead.

5. Modify the version of the printProductReport method with int parameter to capture the IOException.

- a. Add additional catch block to the existing-try-catch construct.

**Hint:** Right-click on the left side of the line of code that is producing the exception in the printProductReport method and invoke "Add catch Clause" menu.

```
public void printProductReport(int id) {
 try {
 printProductReport(findProduct(id));
 } catch (ProductManagerException ex) {
 logger.log(Level.INFO, ex.getMessage());
 } catch (IOException ex) {
 Logger.getLogger(ProductManager.class.getName())
 .log(Level.SEVERE, null, ex);
 }
}
```

- b. Customize Logger behavior for the IOException handler. Produce "Error printing product report" message concatenated with the exception error message, instead of null parameter. Use class-level constant logger.

```
public void printProductReport(int id) {
 try {
 printProductReport(findProduct(id));
 } catch (ProductManagerException ex) {
 logger.log(Level.INFO, ex.getMessage());
 } catch (IOException ex) {
 logger.log(Level.SEVERE,
 "Error printing product report "+ex.getMessage(), ex);
 }
}
```

6. Test printing product reports into files functionality using `findProduct` method.
  - a. Open `Shop` class editor.
  - b. Remove the line of code that represents the first invocation of the `printProductReport` for product 101.

**Notes:**

- You don't want to print this product report before reviews are added to it.
- You will print two reports, for product 101 and 103. The first report will be with reviews and the second one without it.

```
ProductManager pm = new ProductManager("en-GB");
pm.parseProduct("D,101,Tea,1.99,0,2019-09-19");
pm.parseReview("101,4,Nice hot cup of tea");
pm.parseReview("101,2,Rather weak tea");
pm.parseReview("101,4,Good tea");
pm.parseReview("101,5,Perfect tea");
pm.parseReview("101,3,Just add some lemon");
pm.printProductReport(101);
pm.parseProduct("F,103,Cake,3.99,0,2019-09-19");
pm.printProductReport(103);
```

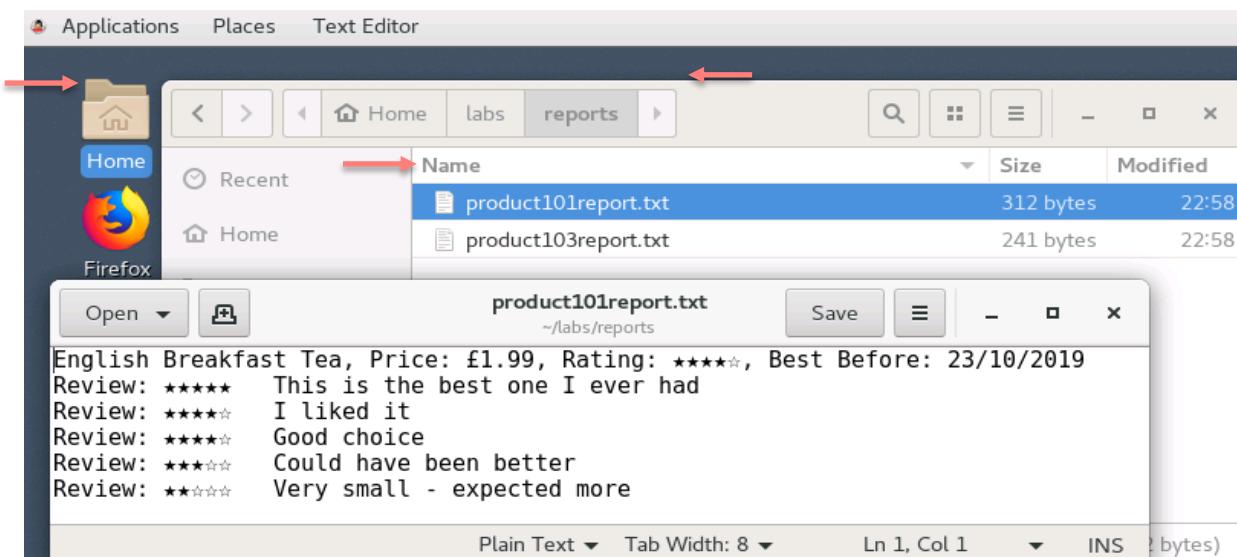
- c. Compile and run your application.

**Hint:** Click the "Run Project" toolbar button.



**Notes:**

- Your program no longer prints product report to the console.
- Navigate to the `/home/oracle/labs/reports` folder.
- Double-click on `product101report.txt` and `product103report.txt` files to open with Text Editor.



- Close these files after you completed inspecting them.
- Printing report twice for the same product will simply overwrite the file.

## Practice 13-2: Bulk-Load Data from Files

---

### Overview

In this practice, you write logic that loads products and reviews from comma-delimited files. You need to modify `parseProduct` and `parseReview` methods and add extra operations that actually read data from files and supply this data to parsing methods.

#### Notes:

- Existing algorithms of the `parseReview` and `parseProduct` methods automatically add `Review` and `Product` objects to the `products` Map collection.
- Such an approach is not suitable for a bulk data load scenario, because every time you invoke the `writeReview` method it recalculates the average rating value for all reviews of this product and recreates products by applying the new rating value.
- You also need to be able to use Java Stream API Collectors to improve bulk load performance and possibly allow parallel processing of information.
- You need to load data for your products and reviews from files as is, without constantly recalculating values.
- Data is stored in two type of files:
  - A `product<id>.csv` – containing a product record. For example:  
`product101.csv` contains:  
D,101,Tea,1.99,0,2019-09-19
  - A `reviews<id>.csv` – containing a review record. For example:  
`reviews101.csv` contains:  
5,This is the best one I ever had  
2,Very small - expected more  
4,I liked it  
4,Good choice  
3,Could have been better
- Review records do not contain product id; instead, it is the actual filename that links these reviews to a specific product.

1. Change the review record format in the config file:
  - a. Open `config.properties` file editor.
  - b. Modify `review.data.format` property to expect only two values (rating and comments).

```
review.data.format={0},{1}
```

2. Modify `parseReview` operation in the `ProductManager` class to construct a new `Review` object and return it to the invoker.

- Open `ProductManager` class editor.
- Change `parseReview` method signature to return `Review` object.

```
public Review parseReview(String text) {
 // existing method logic
}
```

- Inside the `parseReview` method, just before the try block, declare a new variable type of `Review` called `review` and initialize this variable to reference `null` value.

```
Review review = null;
```

- At the end of the method, after the end of the catch block, add a statement to return this `review` variable.

```
return review;
```

- Inside the try block, replace the invocation of the `reviewProduct` method with an invocation of the `Review` constructor.

**Hints:**

- Assign newly created `Review` object to the `review` variable.
- Remove the first parameter that represents the product id, leaving only rating and comments parameters.
- Change index values for the rating and comments to point to first and second elements in the values array.

```
review = new Review(
 Rateable.convert(Integer.parseInt((String)values[0])),
 (String)values[1]);
```

3. Modify `parseProduct` operation in the `ProductManager` class to construct a new `Product` object and return it to the invoker.

- Change `parseProduct` method signature to return `Product` object.

```
public Product parseProduct(String text) {
 // existing method logic
}
```

- Inside the `parseProduct` method, just before the try block, declare a new variable type of `Product` called `product` and initialize this variable to reference `null` value.

```
Product product = null;
```

- At the end of the method, after the end of the catch block, add a statement to return this `product` variable.

```
return product;
```

- d. Inside the try block, replace both invocations of the `createProduct` method with invocations of the `Drink` and `Food` constructors.

**Hints:**

- Assign newly created `Drink` or `Food` object to the `product` variable.

```
switch ((String)values[0]) {
 case "D":
 product = new Drink(id, name, price, rating);
 break;
 case "F":
 LocalDate bestBefore = LocalDate.parse((String)values[5]);
 product = new Food(id, name, price, rating, bestBefore);
}
```

4. Add an operation to the `ProductManager` class that loads reviews data for a given product from the file.

- a. Add `private` method called `loadReviews` just before `parseReview` method. This new method should accept `Product` parameter and return a `List` of `Review` objects.

```
private List<Review> loadReviews(Product product) {
 // reviews loading logic will be added here
}
```

- b. Inside the `loadReviews` method, declare a new variable called `reviews` type of `List` using `Review` as a generic type and initialize this variable to reference `null` value.

```
List<Review> reviews = null;
```

- c. Next, declare a new variable called `file` type of `Path` and initialize this variable to reference a file that contains reviews for a given product.

**Hints:**

- Initialize this variable using `resolve` method to combine a `dataFolder` path with the filename containing reviews for a given product.
- Produce a filename using a `MessageFormat.format` method, passing a value of "reviews.data.file" key from the config file and product id as parameters.

```
Path file =
 reportsFolder.resolve(
 MessageFormat.format(
 config.getString("reviews.data.file"), product.getId())
);
```

- d. Not all products have reviews, so your next step is to check if such reviews file actually exists.

**Hints:**

- Construct an if statement that checks the absence of the file using `Files.notExists` method.
- Inside this if block, assign a new empty `ArrayList` of `Review` objects to the `reviews` variable.

```
if (Files.notExists(file)) {
 reviews = new ArrayList<>();
}
```

- e. Add an `else` block, which contains logic that reads all lines from the reviews file, parses them one by one, and collects results into a List.

**Hints:**

- Use `Files.lines` method to read all lines of text from the review file.
- Specify "UTF-8" as expected character encoding, using `Charset.forName` method.
- Operation `Files.lines` returns a stream of `String` objects.
- Use method `map` to convert each `String` in a stream into a `Review` object with the help of the `parseReview` method.
- In case there is an issue parsing a `Review` method, `parseReview` captures and logs the error and returns null. Therefore, you need to apply a `filter` condition that discards all null objects from the stream.
- Use `collect` method and `Collectors.toList` to assemble a stream of `Review` objects into the list.
- Assign the result to the `reviews` variable.

```
else{
 reviews = Files.lines(file, Charset.forName("UTF-8"))
 .map(text -> parseReview(text))
 .filter(review -> review != null)
 .collect(Collectors.toList());
}
```

- f. Add an import statement for the `Charset` class.

**Hint:** Right-click anywhere in the source code of the `ProductManager` class and invoke "Fix Imports" menu.

```
import java.nio.charset.Charset;
```

- g. Add try-catch block around the invocation of the `Files.lines` method.

**Hint:** Right-click on the left side of the line of code that is producing the exception in the `loadReviews` method and invoke the "Surround Statement with try-catch" menu.

```
try {
 reviews = Files.lines(file, Charset.forName("UTF-8"))
 .map(text -> parseReview(text))
 .filter(review -> review != null)
 .collect(Collectors.toList());
} catch (IOException ex) {
 Logger.getLogger(ProductManager.class.getName())
 .log(Level.SEVERE, null, ex);
}
```

- h. Customize Logger behavior to use the existing logger object and a message that explains the nature of the problem.

**Hints:**

- Use class-level constant that references a logger object.
- Change logger level to WARNING.
- Modify Logger parameters to produce a message indicating an error in loading reviews concatenated with the error message.
- Remove the detailed exception trace printing (the last parameter).

```
try {
 // reviews loading logic
} catch (IOException ex) {
 logger.log(Level.WARNING, "Error loading reviews "+ex.getMessage());
}

i. Add a return statement to return reviews object to the invoker. Place this return statement just before the end of the loadReviews method body.

return reviews;
```

5. Add an operation to the `ProductManager` class that loads product data from a given file.

- a. Add private method called `loadProduct` just before `loadReviews` method. This new method should accept `Path` parameter and return a `Product` object.

```
private Product loadProduct(Path file) {
 // product loading logic will be added here
}
```

- b. Inside the `loadProduct` method, declare a new variable called `product` type of `Product` and initialize this variable to reference `null` value.

```
Product product = null;
```

- c. Parse product data file and assign result to the product variable.

**Hints:**

- Invoke `parseProduct` operation, which accepts a line of text containing product information that you need to retrieve from the file.
- Use `Files.lines` method to read all lines of text from the product file.
- Use `resolve` method to construct a reference for a product file in a data folder.
- Specify "UTF-8" as expected character encoding, using `Charset.forName` method.
- Operation `Files.lines` returns a stream of `String` objects. Only one product per file is expected.
- Use `findFirst` method to get the first line of text from the file.
- Use `orElseThrow` method to raise an exception in case no actual product text is present in the file.

```
product = parseProduct(
 Files.lines(dataFolder.resolve(file), Charset.forName("UTF-8"))
 .findFirst().orElseThrow());
```

**Notes:**

- `orElseThrow` method produces `NoSuchElementException`.
- You were instructed to use this method rather than `get` method to ensure that you would be able to log the issue if the product cannot be located and parsed.

- d. Add try-catch block around the invocation of this product parsing logic

**Hint:** Right-click on the left side of the line of code that is producing the exception in the `loadProduct` method and invoke "Surround Statement with try-catch" menu.

```
try {
 product = parseProduct(
 Files.lines(dataFolder.resolve(file), Charset.forName("UTF-8"))
 .findFirst().orElseThrow());
} catch (IOException ex) {
 Logger.getLogger(ProductManager.class.getName())
 .log(Level.SEVERE, null, ex);
}
```

- e. Customize catch block and the Logger behavior to use existing logger object and a message that explains the nature of the problem.

**Hints:**

- Change catch clause to catch all exceptions and not just IOException to account for a possible NoSuchElementException that can also be produced in this method.
- Use class-level constant that references a logger object.
- Change logger level to WARNING.
- Modify Logger parameters to produce a message indicating an error in loading product concatenated with the error message.
- Remove the detailed exception trace printing (the last parameter).

```
try {
 // product loading logic
} catch (Exception ex) {
 logger.log(Level.WARNING, "Error loading product "+ex.getMessage());
}

f. Add a return statement to return product object to the invoker. Place this return
statement just before the end of the loadProduct method body.

return product;
```

6. Add an operation to the ProductManager class that loads all products and reviews from files located in the data folder.

- a. Add private method called loadAllData just before loadProduct method. This new method does not require any parameters and does not return any value.

```
private void loadAllData() {
 // bulk data load logic will be added here
}
```

- b. Inside the `loadAllData` method, list all product files from the data folder, load each product and reviews for this product, and collect results into the `products` Map object.

**Hints:**

- Use `Files.list` method to list all files in a `dataFolder`.
- Use `filter` method to only list files with name starting with "product".
- Use `map` method to transform the stream of files to the stream of Product objects with the help of the `loadProduct` method.
- Use `filter` method to remove null product entries from the stream.
- Use `collect` method to assemble products and reviews into a Map.
- Use `Collectors.toMap` method to create a map indexed by `Product`, containing a `List` of `Review` objects as a value. This list can be obtained using `loadReviews` method.
- Assign the result to the `products` variable.

```
products = Files.list(dataFolder)
 .filter(file -> file.getFileName().toString().startsWith("product"))
 .map(file -> loadProduct(file))
 .filter(product -> product!=null)
 .collect(Collectors.toMap(product -> product,
 product -> loadReviews(product))) ;
```

**Notes:**

- This code fuses the entire product and review data loading into a single stream pass. All products and corresponding reviews are loaded from the data folder into the memory map referenced by the `products` variable.
  - This algorithm can be parallelized using `toConcurrentMap` instead of `toMap` method.
- c. Add try-catch block around the invocation of this data loading logic

**Hint:** Right-click on the left side of the line of code that is producing the exception in the `loadAllData` method and invoke "Surround Statement with try-catch" menu.

```
try {
 // data loading logic
} catch (IOException ex) {
 Logger.getLogger(ProductManager.class.getName())
 .log(Level.SEVERE, null, ex);
}
```

- d. Customize catch block and the Logger behavior to use existing logger object and a message that explains the nature of the problem.

**Hints:**

- Use class-level constant that references a logger object.
- Use SEVERE logging level, since the failure of this operation would result in general data unavailability for the entire application.
- Modify Logger parameters to produce a message indicating an error in loading product concatenated with the error message.
- Use detailed exception trace printing (the last parameter).

```
try {
 // data loading logic
} catch (IOException ex) {
 logger.log(Level.SEVERE, "Error loading data "+ex.getMessage(), ex);
}
```

7. Adjust ProductManager class design to benefit from the code added in this practice.
- a. Add an invocation of the `loadAllData` method to the `ProductManager` constructor.

**Hints:**

- Find `ProductManager` constructor that accepts `String languageTag` as an argument.
- Add an invocation of the `loadAllData` method after the `changeLocale` method call.

```
public ProductManager(String languageTag) {
 changeLocale(languageTag);
 loadAllData();
}
```

- b. Mark `parseProduct` and `parseReview` operations as `private` to prevent any access to these operations from outside the `ProductManager` class.

```
private Review parseReview(String text)
private Product parseProduct(String text)
```

**Notes:**

- The idea is that an invoker of the `ProductManager` operates on a fully initialized object, with all the required data. However, the exact way in which these products and reviews are parsed is really none of the invokers' business.
- You could make a decision to make `loadProduct` and `loadReviews` operations public, which would allow the invoker to request to load an additional Product from file or a List of Review objects for a given product. However, the application should not normally expose its internal mechanisms of storing information.

- The last change (making parse methods private) prevents class Shop from compiling. Your next task will be to adjust logic in the Shop class so it will only use publically available ProductManager operations and test the application.
8. Test the updated logic from the `main` method of the `Shop` class.
- Open `Shop` class editor.
  - Place comments on all parse method calls and test the print product report functionality.

**Hints:**

- Select relevant lines of code inside the `main` method body.
- Press `CTRL+ /` to add comments to these lines.
- Your main method should simply create a new `ProductManager` object, and it can print product reports immediately.

```
ProductManager pm = new ProductManager("en-GB");
pm.printProductReport(101);
pm.printProductReport(103);
```

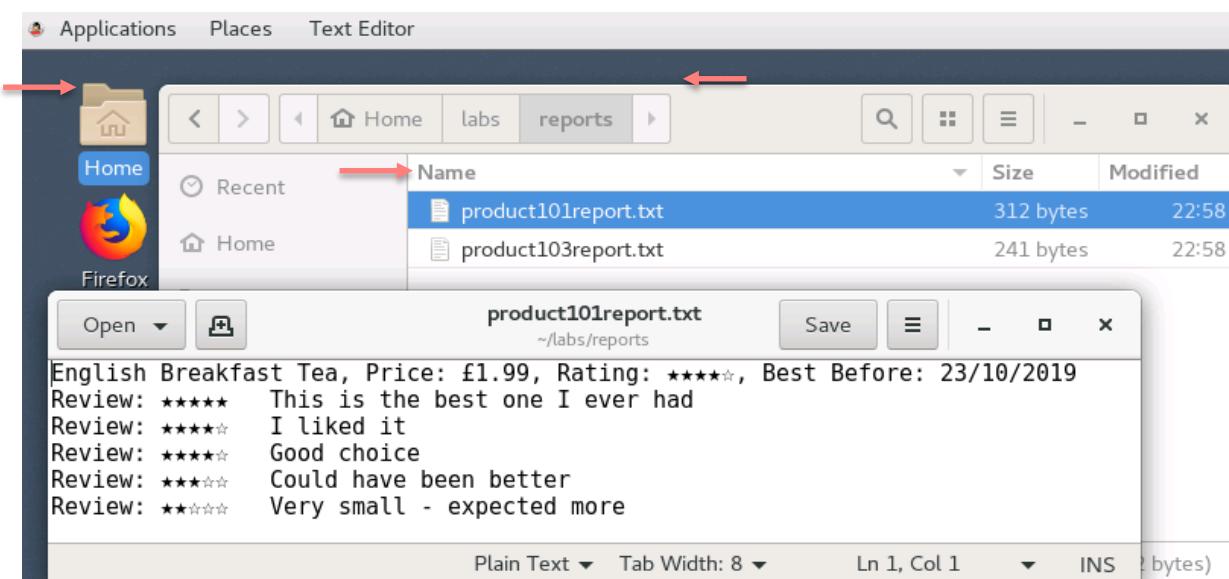
**Note:** You may choose to print reports in a product id range from 101 to 163.

- Compile and run your application.

**Hint:** Click the "Run Project" toolbar button.

**Notes:**

- Your program no longer prints the product report to the console.
- Navigate to the `/home/oracle/labs/reports` folder.
- Double-click on `product101report.txt` and `product103report.txt` files to open with Text Editor.
- Close these files after you have completed inspecting them.



- Printing report for the same product twice will simply overwrite the file.

- d. Attempt to print the product report for the non-existent product. You can use any product id outside the 101–163 range.

```
pm.printProductReport(42);
```

- e. Compile and run your application.

**Hint:** Click the "Run Project" toolbar button.



**Note:** Observe the error information.

- f. Remove the offending line of code that prints non-existent product.

- g. Remove comments from the line of code that prints products using a Predicate condition to look for products with price less than 2 and string than by rating.

**Hints:**

- Select relevant lines of code inside the `main` method body.
- Press `CTRL+ /` to add comments to these lines.

```
pm.printProducts(
 p -> p.getPrice().floatValue() < 2,
```

```
(p1,p2) -> p2.getRating().ordinal()-p1.getRating().ordinal());
```

- h. Compile and run your application.

**Hint:** Click the "Run Project" toolbar button.



**Note:** Observe a list of products printed to the console.

- i. Add a new Product using the `createProduct` method and some reviews using the `reviewProduct` method. Use product id 164 and price below 2, so it will appear in the list of products produced with the `printProducts` method invocation. Also, print the product report for this new product.

```
pm.createProduct(164, "Kombucha", BigDecimal.valueOf(1.99),
 Rating.NOT_RATED);

pm.reviewProduct(164, Rating.TWO_STAR, "Looks like tea but is it?");
pm.reviewProduct(164, Rating.FOUR_STAR, "Fine tea");
pm.reviewProduct(164, Rating.FOUR_STAR, "This is not tea");
pm.reviewProduct(164, Rating.FIVE_STAR, "Perfect!");

pm.printProductReport(164);
```

- j. Compile and run your application.

**Hint:** Click the "Run Project" toolbar button.



**Notes:**

- Observe this product information in the output produced by the product report.
- You may also look at the `product164report.txt` file, located in the reports folder: `/home/oracle/labs/reports`.
- You may also add reviews to any existing products.
- However, such new products and new reviews only exist in the Map of products managed by the `ProductManager` object, because no code has been written to actually save these objects into the data folder as comma-delimited files.
- This practice does not cover writing logic to save data to files, because of the complexity of this task. You will have to track changed, added, or removed products and reviews, as well as actually write this data to files and add transactional logic to verify if data has been saved successfully. You would not normally write such code, but use some system (such as a database) that provides appropriate implementation of transactional functionalities for you.

- k. (Optional) Clean up test code in the `main` method of the `Shop` class.

**Hint:** Remove all commented lines of code from the `main` method, leaving only the following:

```
ProductManager pm = new ProductManager("en-GB");
pm.printProductReport(101);
pm.createProduct(164, "Kombucha", BigDecimal.valueOf(1.99),
 Rating.NOT_RATED);
pm.reviewProduct(164, Rating.TWO_STAR, "Looks like tea but is it?");
pm.reviewProduct(164, Rating.FOUR_STAR, "Fine tea");
pm.reviewProduct(164, Rating.FOUR_STAR, "This is not tea");
pm.reviewProduct(164, Rating.FIVE_STAR, "Perfect!");
pm.printProductReport(164);
pm.printProducts(p->p.getPrice().floatValue() < 2,
 (p1,p2)->p2.getRating().ordinal()-p1.getRating().ordinal());
pm.getDiscounts().forEach(
 (rating, discount)->System.out.println(rating+"\t"+discount));
```

## Practice 13-3: Implement Memory Swap Mechanism

---

### Overview

In this practice, you write logic that dumps all product and review objects into temporary swap file and restore objects from this file back into memory.

1. Permit objects type of Product and Review to be written and read to and from data streams.
  - a. Open `Product` class editor.
  - b. Make `Product` class implement the `Serializable` interface.

**Hints:**

- Add `Serializable` interface to the list of interfaces in the `implements` clause of the `Product` class definition.

```
public abstract class Product
 implements Rateable<Product>, Serializable {
 // existing code of the product class
}
```

- c. Add an import statement for the `java.io.Serializable` interface.

**Hint:** Right-click anywhere in the source code of the `Product` class and invoke "Fix Imports" menu.

```
import java.io.Serializable;
```

- d. Open `Review` class editor.

- e. Make `Review` class implement the `Serializable` interface.

**Hints:**

- Add `Serializable` interface to the list of interfaces in the `implements` clause of the `Product` class definition.

```
public class Review
 implements Comparable<Review>, Serializable {
 // existing code of the review class
}
```

- f. Add an import statement for the `java.io.Serializable` interface.

**Hint:** Right-click anywhere in the source code of the `Review` class and invoke "Fix Imports" menu.

```
import java.io.Serializable;
```

2. Add operation to the ProductManager class to write an entire content of the products Map to a file.
  - a. Open ProductManager class editor.
  - b. Add new public method called `dumpData` that should accept no arguments and does not need to return a value. Add this method just before the `loadAllData` method.

```
public void dumpData() {
 /*
 Write products map object into a file.
 Reassign products variable to reference new empty HashMap object.
 */
}
```

- c. Inside `dumpData` method, add a try-catch construct that is expecting `IOException` and is using the class-level logger to write an exception message and stack trace to the log.

```
try {
 // the rest of the method logic will be written here
} catch(IOException ex) {
 logger.log(Level.SEVERE,
 "Error dumping data " + ex.getMessage(), ex);
}
```

**Note:** This try block is intended to contain code that serializes Java objects. Such a code may throw `java.io.NotSerializableException`. However, this exception is a subtype of a `IOException`, so the catch block that you provided would be capable of handling it as well.

- d. Inside this try block, add if statement that creates a directory for `tempFolder` path if it does not exist.

**Hints:**

- Use `Files.notExists` method to verify the absence of the temp folder.
- Use `Files.createDirectory` method to create the temp folder.

```
if (Files.notExists(tempFolder)) {
 Files.createDirectory(tempFolder);
}
```

**Hint:** `Files.createDirectory` method verifies if the path does not exist before trying to create it.

- e. After the end of the if block, add a declaration of a new variable called `tempFile` type of Path and initialize this variable to reference a temporary file object that uses timestamp as a file name.

**Hints:**

- Invoke `resolve` method upon the `tempFolder` Path object to construct a path to the temporary file.
- Use `MessageFormat.format` method, passing a value of "temp.file" key from the config file, and substitute a timestamp as a file name.
- Use `Instant.now()` method to generate timestamp.
- Create a file if it does not exist, otherwise overwrite the existing file.

```
Path tempFile = tempFolder.resolve(
 MessageFormat.format(config.getString("temp.file"), Instant.now()));
```

- f. Add an import statement for the `java.time.Instant` class.

**Hint:** Right-click anywhere in the source code of the `ProductManager` class and invoke "Fix Imports" menu.

```
import java.time.Instant;
```

- g. Next, inside the same try block, immediately after the declaration of the `tempFile` variable, add a `try-with-resources` block that declares and initializes a new `ObjectOutputStream` object that is connected to the output stream referencing your file.

**Hints:**

- Inside this inner `try` block parameter section, declare a new variable called `out` type of `ObjectOutputStream`.
- Use the `ObjectOutputStream` constructor to initialize the `out` variable.
- Constructor of the `ObjectOutputStream` should accept a new output stream pointing to the required file.
- Use `Files.newOutputStream` method pointing to the `tempFile`.
- Use `StandardOpenOption.CREATE` to create a file if it does not exist, otherwise overwrite the existing file.

```
try (ObjectOutputStream out = new ObjectOutputStream(
 Files.newOutputStream(tempFile, StandardOpenOption.CREATE))) {
 // data writing logic will be added here
}
```

- h. Add an import statement for the `java.io.ObjectOutputStream` class.

**Hint:** Right-click anywhere in the source code of the `ProductManager` class and invoke "Fix Imports" menu.

```
import java.io.ObjectOutputStream;
```

- i. Inside the inner try block, write `products` map to the output stream and reset `products` object to reference new empty `HashMap` object.

**Hints:**

- Use `writeObject` method to serialize data.
- Don't forget to use `<>` indicator when creating an instance of the `HashMap` to acknowledge the use of generics (variable `products` is declared as a map of products and lists of review objects: `Map<Product, List<Review>>`)

```
out.writeObject(products);
products = new HashMap<>();
```

3. Add operation to the `ProductManager` class to restore the entire content of the `products` Map back into memory.
  - a. Add new `public` method called `restoreData` that should accept no arguments and does not need to return a value. Add this method just before the `saveChanges` method.

```
public void restoreData() {
 /*
 * Read products map object from a file.
 * Reassign products variable to reference this restored HashMap.
 */
}
```

- b. Inside `restoreData` method, add a try-catch construct that is expecting any `Exception` and is using the class-level logger to write an exception message and stack trace to the log.

```
try {
 // the rest of the method logic will be written here
} catch(Exception ex) {
 logger.log(Level.SEVERE,
 "Error restoring data " + ex.getMessage(), ex);
}
```

**Notes:** Code that will be placed inside this try block can produce several exception types:

- `IOException` to indicate any filesystem access issues
- `ClassNotFoundException` to indicate that serialized version of code does not match current versions of code
- `NoSuchElementException` to indicate that no temp file was found

- c. Inside the try block, add a declaration of a new variable called `tempFile` type of `Path` and initialize this variable to reference a temporary file.

**Hints:**

- Invoke `Files.list` method to get the listing of all files in a temp folder.
- Use `filter` operation on a directory listing stream to only select files whose name ends with word "tmp".
- Use `findFirst` method to get the first temp file in this folder.
- Use `orElseThrow` method to either get the path object or throw a `NoSuchElementException` if no matching file is found.

```
Path tempFile = Files.list(tempFolder)
 .filter(path->
 path.getFileName().toString().endsWith("tmp"))
 .findFirst().orElseThrow();
```

**Note:** This logic assumes that there should be exactly one temporary file in the designated folder

- d. Next, inside the same try block, immediately after the declaration of the `tempFile` variable add a `try-with-resources` block that declares and initializes a new `ObjectInputStream` object that is connected to the input stream referencing your file.

**Hints:**

- Inside the `try` parameter section, declare a new variable called `in` type of `ObjectInputStream`.
- Use `ObjectInputStream` constructor to initialize the `in` variable.
- Constructor of the `ObjectInputStream` should accept a new input stream pointing to the required file.
- Use `Files.newInputStream` method, pointing to the `tempFile` and instructing it to delete the file upon closure.

```
try (ObjectInputStream in = new ObjectInputStream(
 Files.newInputStream(tempFile,
 StandardOpenOption.DELETE_ON_CLOSE))) {
 // logic that restores products object from the temp file
}
```

- e. Add an import statement for the `java.io.ObjectInputStream` class.

**Hint:** Right-click anywhere in the source code of the `ProductManager` class and invoke "Fix Imports" menu.

```
import java.io.ObjectInputStream;
```

- f. Inside the inner try block, reassign the `products` map using a value restored from the input stream.

**Hints:**

- Use `readObject` method to read back previously serialized data.
- Cast the restored object to `HashMap` type.

```
products = (HashMap) in.readObject();
```

- g. Recompile ProductManagement project.

**Hint:** Use Run->Clean and Build Project menu or a toolbar button.



**Notes:**

- There is a warning message produced by the compiler that tells you that you are using unchecked or unsafe operations.
- The reason for this message is the inevitable loss of generics information during the serialization-deserialization process.
- For more information on why this is the case and how it can be mitigated, refer to the lessons titled "Appendix A: Annotations" and "Appendix D: Generics" of this course.

- h. Add annotation that informs the compiler that you consider your code that is restoring the products `HashMap` to be in line with the way it is intended to be used. Basically, you are sure that the serialized content in the file indeed contains a map of products and reviews and thus is fully compatible with the generics expectations set by the declaration of the `product` variable.

**Hints:**

- To tell the compiler that you are safely using generics, add the following annotation in front of the `restoreData` method definition:

```
@SuppressWarnings("unchecked")
public void restoreData() {
 // the rest of the method logic
}
```

- i. Recompile ProductManagement project.

**Hint:** Use Run->Clean and Build Project menu or a toolbar button.



**Note:** Compiler is no longer producing the warning message.

4. Test the new logic from the `main` method of the `Shop` class.
  - a. Open `Shop` class editor.
  - b. Place invocations of `dumpData` and `restoreData` methods just before you invoke `printProductReport` method for the product with 164 id.

```
ProductManager pm = new ProductManager("en-GB");
pm.printProductReport(101);
pm.createProduct(164, "Kombucha", BigDecimal.valueOf(1.99),
 Rating.NOT_RATED);
pm.reviewProduct(164, Rating.TWO_STAR, "Looks like tea but is it?");
pm.reviewProduct(164, Rating.FOUR_STAR, "Fine tea");
pm.reviewProduct(164, Rating.FOUR_STAR, "This is not tea");
pm.reviewProduct(164, Rating.FIVE_STAR, "Perfect!");
pm.dumpData();
pm.restoreData();
pm.printProductReport(164);
pm.printProducts(p->p.getPrice().floatValue() < 2,
 (p1,p2)->p2.getRating().ordinal()-p1.getRating().ordinal());
pm.getDiscounts().forEach(
 (rating, discount)->System.out.println(rating+"\t"+discount));
```

- c. Compile and run your application.

**Hint:** Click the "Run Project" toolbar button.



**Notes:**

- Observe the products printed to the console.
- No errors indicating that products with a given id are not found were produced.
- This would not be the case if you place comments on the line of code that invokes `restoreData` method and run the program again.
- If you are interested to see the actual format of the serialization data file, you may change the setting in the `restoreData` method in the `ProductManager` class from `StandardOpenOption.DELETE_ON_CLOSE` to `StandardOpenOption.READ`, which will not delete the file once you have executed your program. Don't forget to revert this change once you have investigated the file.

- d. Change access modifier for `dumpData` and `restoreData` methods to `private` within the `ProductManager` class.

```
private void dumpData() { /* existing method implementation */ }
private void restoreData() { /* existing method implementation */ }
```

**Notes:** You may wish to trigger `dumpData` and `restoreData` methods from within the `ProductManager` class based on some internally managed conditions, such as requirements to conserve memory. However, it is not safe to allow other invokers to trigger such memory dump and restore operation directly.

- e. Remove test code from the `main` method of the `Shop` class, except the line of code that declares and initializes the `ProductManager` object.

**Hint:** This is the only code that should be left in the `main` method:

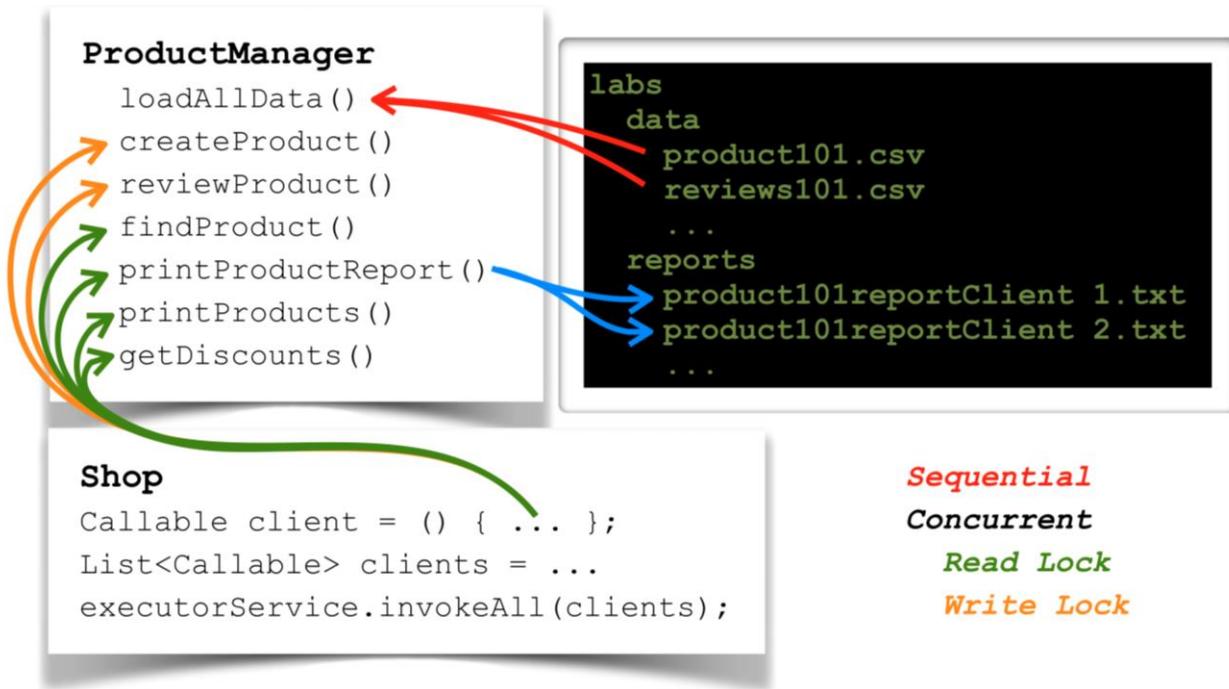
```
ProductManager pm = new ProductManager("en-GB");
```

## **Practices for Lesson 14: Java Concurrency and Multithreading**

## Practices for Lesson 14: Overview

### Overview

In these practices, you will write code that simulates multiple concurrent callers that are going to share a single instance of the ProductManager. You will need to decouple locale management from the instance of the ProductManager to allow different concurrent callers to set their own locales without affecting the others. You also need to protect your data cache (map of products and reviews) from corruption using an appropriate locking strategy. Finally, you will have to resolve filesystem path clashes that may occur when concurrent callers attempt to write the



same files (such as a report for the same product) at the same time.

## Concurrency Design Analysis of the ProductManagement Application

There could be different scenarios of how exactly multiple concurrent callers may use ProductManager class. You need to perform analysis of what are the exact resources that would be shared between callers, before deciding upon the best design approach.

1. Each concurrent caller creates its own instance of the ProductManager.
  - In this scenario, data cache (map of products and reviews) stored inside the ProductManager as an instance variable would not be shared between different threads. Having a separate copy of a data cache created for each concurrent caller would consume memory.
  - Each invoker would be able to assign its own locale for the purposes of formatting resources.
2. All concurrent callers share a single instance of the ProductManager.
  - In this scenario, data cache (map of products and reviews) stored inside the ProductManager as an instance variable becomes shared between different threads. Having a shared data cache would require to adjust logic in the ProductManager class to prevent potential data corruption.
  - Invokers would not be able to use different locales, unless locale management is redesigned in a way that does not use instance variables.

In any of these scenarios, concurrent callers would still clash on access to folders, such as reports or temp folders. For example, if they attempt to print product report on the same product at the same time.

Proposed designs solutions are relatively similar to the Stateful and Singleton EJBs (Enterprise Java Beans) as described by the Java EE specification. Using Java EE significantly automates all concurrency and transaction management tasks. You can learn more about this from the "Developing Applications for the Java EE 7 Platform" course.

## Practice 14-1: Redesign ProductManager as a Singleton

---

### Overview

In this practice, you modify the design of a ProductManager, making it singleton. You also allow a selection of different ResourceFormatter objects for individual method invocations.

### Assumptions

- JDK 11 is installed.
- NetBeans 11 is installed.
- You have completed Practice 13 or have started with the solution for Practice 13 version of the application.

### Tasks

1. Prepare the practice environment.

#### Notes:

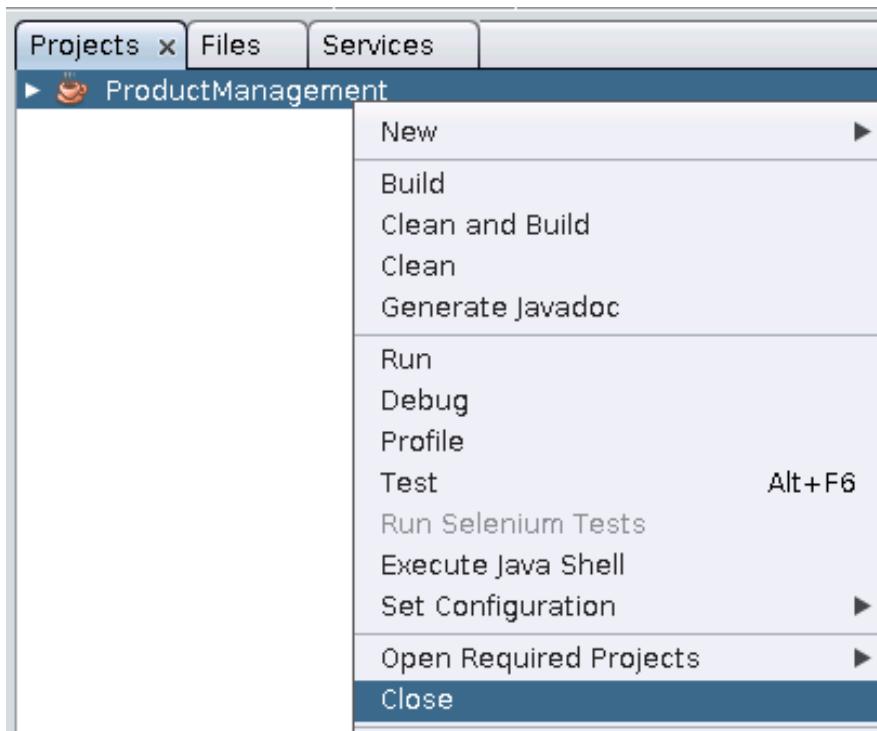
- You may continue to use the same NetBeans project as before, if you have successfully completed the previous practice. In this case, proceed directly to Practice 14-1, step 2.
- Alternatively, you can open a fresh copy of the NetBeans project, which contains the completed solution for the previous practice.

- a. Open NetBeans (if it is not already running).

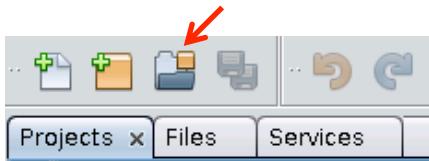


- b. Close the currently opened ProductManagement project.

**Hint:** Right-click on the ProductManagement project and invoke "Close" menu.



- c. Open the Solution for Practice 13 version of the ProductManagement project.  
**Hint:** Use File -> Open Project menu or click the "Open Project" toolbar button.



- d. Navigate to the /home/oracle/labs/solutions/practice13 folder and select ProductManagement project.  
e. Click "Open Project".
2. Modify the way in which ResourceFormatter and ProductManager instances are related.

**Notes:**

- Currently, ResourceFormatter is selected and associated with each ProductManager instance. This means that a number of concurrent callers sharing the same instance of ProductManager would have to use the same ResourceFormatter instance as well.
  - Your task is to change this design in a way that allows the invoker to select the required ResourceFormatter at the moment of specific operation invocation.
- Open ProductManager class editor.
  - Place comments on the line of code that declares the formatter instance variable:

**Hints:**

- Select relevant line of code inside ProductManager class.
- Press **CTRL+ /** to add comments to this line.

```
// private ResourceFormatter formatter;
```

**Notes:**

- All operations in the ProductManager class that used this variable now fail to compile and must be adjusted.
- These are: getDiscounts, printProducts, printProductReport, and changeLocale.

3. Modify getDiscounts and printProducts operations and both versions of printProductReport to use an extra language tag parameter.
- Make getDiscounts method use an additional parameter to designate the required language tag to select the ResourceFormatter instance to be used within this method.

**Hints:**

- Locate getDiscounts method inside the ProductManager class.
- Add an additional parameter called languageTag of String type.

```
public Map<String, String> getDiscounts(String languageTag) {
 // initialisation of the ResourceFormatter will be added here
 // the rest of the method body
```

- b. Inside `getDiscounts` method, create the local variable called `formatter` type of `ResourceFormatter` and initialize this variable to reference `ResourceFormatter` object from the `formatters` Map, using the supplied `languageTag` parameter value. The "en-GB" language tag should be used if the supplied parameter did not match any value in the `formatters` map.

**Hints:**

- This must be the first line of code inside the `getDiscounts` method body.
- This new local variable should be initialized in the same way as the `formatter` instance variable.
- You can copy-paste the required initialization logic from the `changeLocale` method.

```
ResourceFormatter formatter =
```

```
 formatters.getOrDefault(languageTag, formatters.get("en-GB"));
```

- c. Make `printProducts` method use an additional parameter to designate the required language tag to select the `ResourceFormatter` instance to be used within this method.

**Hints:**

- Locate the `printProducts` method inside the `ProductManager` class.
- Add additional parameter called `languageTag` of `String` type.

```
public void printProducts(Predicate<Product> filter,
 Comparator<Product> sorter, String languageTag) {
 // initialisation of the ResourceFormatter will be added here
 // the rest of the method body
```

- d. Inside `printProducts` method, create a local variable called `formatter` type of `ResourceFormatter` and initialize this variable to reference `ResourceFormatter` object from the `formatters` Map, using the supplied `languageTag` parameter value. The "en-GB" language tag should be used if the supplied parameter does not match any value in the `formatters` map.

**Hints:**

- This must be the first line of code inside the `printProducts` method body.
- This new local variable should be initialized in the same way as the `formatter` instance variable.
- You can copy-paste the required initialization logic from the `changeLocale` or `getDiscounts` method.

```
ResourceFormatter formatter =
```

```
 formatters.getOrDefault(languageTag, formatters.get("en-GB"));
```

- e. Make both versions of the `printProductReport` method use an additional parameter to designate the required language tag to select ResourceFormatter instance to be used within this method.

**Hints:**

- Locate both versions of the `printProductReport` method inside the `ProductManager`.
- Add additional parameter called `languageTag` of `String` type.

```
public void printProductReport(int id, String languageTag) {
 // the rest of the method body
}

public void printProductReport(Product product, String languageTag)
 throws IOException {
 // initialisation of the ResourceFormatter will be added here
 // the rest of the method body
}
```

- f. Inside `printProductReport` method with the `int id` argument, modify the invocation of the `printProductReport` method with the `Product` argument to pass the `languageTag` parameter.

```
printProductReport(findProduct(id), languageTag);
```

- g. Inside `printProductReport` method with the `Product` argument, create a local variable called `formatter` type of `ResourceFormatter` and initialize this variable to reference `ResourceFormatter` object from the `formatters` Map, using the supplied `languageTag` parameter value. The "en-GB" language tag should be used if the supplied parameter does not match any value in the `formatters` map.

**Hints:**

- This must be the first line of code inside the `printProductReport` method body.
- This new local variable should be initialized in the same way as the `formatter` instance variable.
- You can copy-paste the required initialization logic from the `changeLocale`, `getDiscounts`, or `printProducts` method.

```
ResourceFormatter formatter =
 formatters.getOrDefault(languageTag, formatters.get("en-GB"));
```

4. Remove all remaining references to the formatter instance variable.

- a. Place comments on the `changeLocale` method.

**Hints:**

- Select relevant lines of code inside `ProductManager` class.
- Press `CTRL+ /` to add comments to these lines,

```
// public void changeLocale(String languageTag) {
// formatter = formatters.getOrDefault(languageTag,
// formatters.get("en-GB"));
// }
```

- b. Place comments on the invocation of the `changeLocale` method from `ProductManager` constructor and remove the `languageTag` argument.

**Hints:**

- Select relevant lines of code inside `ProductManager` class.
- Press `CTRL+ /` to add comments to these lines.

```
public ProductManager() {
 // changeLocale(languageTag);
 loadAllData();
}
```

- c. Place comments on the invocation of the `ProductManager` constructor with `Locale` argument.

**Hints:**

- Select relevant lines of code inside `ProductManager` class.
- Press `CTRL+ /` to add comments to these lines.

```
// public ProductManager(Locale locale) {
// this(locale.toLanguageTag());
// }
```

**Notes:**

- Now all references to the instance variable `formatter` were removed from the `ProductManager` class and replaced with the use of equivalent local variables within relevant methods.

- d. You may now remove all commented lines of code related to the `formatter` object from the `ProductManager` class.

5. Modify `ProductManager` class to make it use a Singleton design pattern. Singleton is a design pattern that guarantees a creation of exactly one instance of a given class.

To make your class a singleton, you need to:

- Make its constructor private so that no instances of this class can be created anywhere but in this class itself
- Create and initialize a private, static, and final variable that references the only instance of this class you'll ever going to create
- Create public static method that returns this only instance to the invoker

- Apply private access modifier to the `ProductManager` constructor.

```
private ProductManager() {
 loadAllData();
}
```

- Add new `private static` and `final` variable called `pm` type of `ProductManager` and initialize it to reference new instance of `ProductManager`. Place this variable declaration immediately after the `logger` variable declaration.

```
private static final ProductManager pm = new ProductManager();
```

- Add new `public static` method called `getInstance` that should accept no arguments and return `ProductManager` object. Add this operation just before `getSupportedLocales` method.

```
public static ProductManager getInstance() {
 return pm;
}
```

- Recompile `ProductManagement` project.

**Hint:** Use Run->Clean and Build Project menu or a toolbar button.



**Note:** Class `Shop` will not compile, because it is currently using `ProductManager` constructor, which is no longer available.

- Modify the way in which the main method of the `Shop` class obtains `ProductManager` instance.

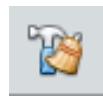
- Open `Shop` class editor.

- Replace initialization of the `pm` variable with an invocation of the `getInstance` method upon the `ProductManager` class.

```
ProductManager pm = ProductManager.getInstance();
```

- Recompile `ProductManagement` project.

**Hint:** Use Run->Clean and Build Project menu or a toolbar button.



**Notes:**

- All your code should now successfully compile.
- You have redesigned `ProductManager` class as a singleton and allowed a selection of different `ResourceFormatter` objects for individual method invocations.

## Practice 14-2: Ensure ProductManager Memory Safety

---

### Overview

In this practice, you modify the code in the `ProductManager` class to ensure safe and consistent access to its data and operations by multiple concurrent callers. There are different design cases presented in this practice:

- Some data can be made immutable and therefore safe for concurrent callers to access.
  - Some data can be made thread specific instead of shared.
  - Some data has to stay mutable, so access to it has to be guarded with read and write locks.
1. Make `ProductManager` class as immutable as possible to prevent accidental collisions between concurrent callers.

**Notes:** You need to consider if it is safe to concurrently access variables (both instance and static) defined within the `ProductManager` class

- The class variable `formatters` referencing the Map of `ResourceFormatter` objects is actually already safe to use with concurrent access for the following reasons:
  - Its value is constructed using `Map.of` method that produces an immutable Map object.
  - `ResourceFormatter` objects are themselves immutable as well.
- Instance variables `config`, `reviewFormat`, `productFormat`, `reportsFolder`, `dataFolder`, and `tempFolder` all represent `ProductManager` configuration properties. Once initialized, they are never reassigned, so they are effectively final.
- All these variables can be marked as `final` to ensure they are not accidentally reassigned.
  - a. Open `ProductManager` class editor.
  - b. Make `config` variable a constant.

```
private final ResourceBundle config =
ResourceBundle.getBundle("labs.pm.data.config");
 c. Make reviewFormat variable a constant.
private final MessageFormat reviewFormat = new
MessageFormat(config.getString("review.data.format"));
 d. Make productFormat variable a constant.
private final MessageFormat productFormat = new
MessageFormat(config.getString("product.data.format"));
 e. Make reportsFolder variable a constant.
private final Path reportsFolder =
Path.of(config.getString("reports.folder"));
 f. Make dataFolder variable a constant.
private final Path dataFolder =
Path.of(config.getString("data.folder"));
 g. Make tempFolder variable a constant.
private final Path tempFolder =
Path.of(config.getString("temp.folder"));
```

h. Make formatters variable a constant.

```
private static final Map<String, ResourceFormatter> formatters =
 Map.of("en-GB", new ResourceFormatter(Locale.UK),
 "en-US", new ResourceFormatter(Locale.US),
 "fr-FR", new ResourceFormatter(Locale.FRANCE),
 "ru-RU", new ResourceFormatter(new Locale("ru", "RU"))),
 "zh-CN", new ResourceFormatter(Locale.CHINA));
```

2. Analyze requirements to ensure safe access to the Map of Product and Review objects referenced by the products variable.

Instance variable `products` is referencing a mutable object. You need to analyze which operations require read or write access to this variable:

`loadAllData`, `loadProduct`, `loadReview`, and `restoreData` operations and both versions of `createProduct` and `reviewProduct` require write access to the `products` `HashMap`.

- `loadAllData`, `loadProduct`, and `loadReview` methods populate `products` `HashMap` with data loaded from files.
  - At the moment, this process is only triggered once, when the instance of `ProductManager` is created. Because `ProductManager` class is now designed as a singleton, these operations are not supposed to be triggered again.
  - You would have to consider applying locking or synchronization mechanisms to these methods if you change your design and allow data reload at any stage other than the initialization of the `ProductManager` single instance.
- `restoreData` method recreates the entire Map of Product and Review objects and reassigned the `products` variable.
  - Only one thread can be permitted to call such an operation at any time.
  - All other threads must be blocked from accessing `ProductManager` object during this operation.
- `createProduct` and `reviewProduct` methods add, remove, and update entries in the `products` Map.
  - Only one thread can be permitted to call such operations at any time.
  - All other threads must be blocked from accessing `ProductManager` object during the execution of one of these operations.

`dumpData`, `findProduct`, `getDiscounts`, and `printProducts` operations and both versions of `printProductReport` only require read access to the `products` `HashMap`.

- They do not cause any modifications to the `products` Map and therefore are not at risk of corrupting memory if invoked concurrently.
- However, you still must ensure consistent state of the `products` Map when these methods read data from it.

Safe access to the `products` map may be achieved in several alternative ways:

- Change type of products Map to ConcurrentHashMap. This option is suitable only if mutator methods perform simple atomic actions. Unfortunately, in the ProductManager class reviewProduct method goes beyond such simple actions, because it is reconstructing the product object and thus it has to remove the entire entry from the map and then replace it with the new entry.
- Make all operations that access this map synchronized, preventing any concurrent callers from either reading or writing to it simultaneously. This option would cause significant performance degradation.
- Apply read locks to all operations that read from the products map and write locks to all operations that modify this map. This option is most flexible as it allows mutator methods to perform complex actions upon shared data and perform read operations without blocking concurrent callers.

The fact that `dumpData`, `restoreData`, `loadAllData`, `loadProduct`, and `loadReview` operations are `private` does not make them thread-safe. They could still be invoked from within the `ProductManager` class concurrently with other invokers calling other operations at the same time. However, for the purposes of this practice, you should assume that no concurrent calls to private methods of the `ProductManager` class are going to be performed.

3. Provide read-write lock mechanism to ensure safe access to the products map.
  - a. Add new private instance constant called `lock` type of `ReentrantReadWriteLock` and initialize it to reference new `ReentrantReadWriteLock` instance. Place this constant declaration immediately after the products Map declaration.

```
private final ReentrantReadWriteLock lock =
 new ReentrantReadWriteLock();
```

- b. Add an import statement for the `ReentrantReadWriteLock` class.

**Hint:** Right-click anywhere in the source code of the `ProductManager` class and invoke "Fix Imports" menu.

```
import java.util.concurrent.locks.ReentrantReadWriteLock;
```

- c. Add two new private instance constants called `writelock` and `readLock` both type of `Lock` and initialize them to reference read and write lock object obtained from the `lock` variable. Place these constant declarations immediately after the `lock` variable declaration.

```
private final Lock writeLock = lock.writeLock();
private final Lock readLock = lock.readLock();
```

- d. Add an import statement for the `Lock` interface.

**Hint:** Right-click anywhere in the source code of the `ProductManager` class and invoke "Fix Imports" menu.

```
import java.util.concurrent.locks.Lock;
```

**Notes:**

- Any operation that requires write access to products map should use writeLock.
- Any operation that requires read access to products map should use readLock.
- To implement lock management, you should use the following code pattern:

```
try {
 writeLock.lock(); // or readLock.lock();
 // perform actions upon products map
} catch(Exception ex) {
 // perform error handling actions
} finally {
 writeLock.unlock(); // or readLock.unlock();
}
```

4. Change logic in both versions of `createProduct` method to manage product creation and addition to the `HashMap` using the write lock.

**Note:** You can perform all required changes in one of the versions of `createProduct` method and then replicate these changes to the other version, or if you wish, you may update both versions of this method step by step.

- a. Separate `product` variable declaration and initialization in both versions of `createProduct` method.

```
Product product = null;
product = new Food(id, name, price, rating, bestBefore);
return product;
```

Same change is applied to the other version of `createProduct` method:

```
Product product = null;
product = new Drink(id, name, price, rating);
return product;
```

- b. Inside both versions of the `createProduct` method, add a `try/catch/finally` block that captures any exceptions and writes logger message indicating that the product failed to be added.

**Hints:**

- Include product initialization and addition of product into the map inside the try block.
- Obtain write lock inside the try block and release it in the finally block.
- Use info logger level.
- Once logger message is produced, return null from the catch block.

```
Product product = null;
```

```
try {
 writeLock.lock();
 product = new Food(id, name, price, rating, bestBefore);
 products.putIfAbsent(product, new ArrayList<>());
} catch(Exception ex) {
 logger.log(Level.INFO,
 "Error adding product "+ex.getMessage());
 return null;
} finally {
 writeLock.unlock();
}
return product;
```

Same change is applied to the other version of `createProduct` method:

```
Product product = null;
try {
 writeLock.lock();
 product = new Drink(id, name, price, rating);
 products.putIfAbsent(product, new ArrayList<>());
} catch(Exception ex) {
 logger.log(Level.INFO,
 "Error adding product "+ex.getMessage());
 return null;
} finally {
 writeLock.unlock();
}
return product;
```

**Note:** The way logic of the `createProduct` methods is written presents an issue. It is possible that product object will be created, but not added to the map, for example, if such a product is already present in the map. This is why the catch block should discard the product object reference.

5. Modify `findProduct` method to use the read lock.
- Surround the existing `return` statement inside the `findProduct` method with a `try` block. Initiate read lock as the first line of code inside this `try` block. Add a `finally` block and release the read lock.

```
try {
 readLock.lock();
 return products.keySet()
 .stream()
 .filter(p -> p.getId() == id)
 .findFirst()
 .orElseThrow(() ->
 new ProductManagerException(
 "Product with id " + id + " not found"));
} finally {
 readLock.unlock();
}
```

**Note:** `findProduct` method is declared to propagate exception to the invoker. That is why catch block is not required in this case.

6. Modify `reviewProduct` method to use the write lock.
- Locate the version of `reviewProduct` method that accepts `int`, `Rating`, and `String` arguments.
- Note:** This version of the `reviewProduct` method already contains try/catch construct, and it is actually invoking the other version of the `reviewProduct`.
- Add a `finally` block after the catch in the `reviewProduct` method. Acquire a write lock as the first action within the `try` block inside the `reviewProduct` method. Release this lock in the `finally` block.

```
public Product reviewProduct(int id, Rating rating,
 String comments) {
 try {
 writeLock.lock();
 return reviewProduct(findProduct(id), rating, comments);
 } catch (ProductManagerException ex) {
 logger.log(Level.INFO, ex.getMessage());
 return null;
 } finally {
 writeLock.unlock();
 }
}
```

- c. Apply `private` access modifier to the other version of `reviewProduct` method.

**Note:** This practice makes an assumption that private methods of the `ProductManager` would not be used by concurrent invokers. This is not a realistic design, as this cannot really be guaranteed. If you want, you can add read and write locks to private methods as well, but it would be quite a repetitive task.

```
private Product reviewProduct(Product product, Rating rating,
 String comments) {
 // existing method logic unchanged
}
```

- 7. Modify `printProductReport` method to use the read lock and prevent concurrent invokers clashing on attempts to write the same file.
  - a. Locate the version of `printProductReport` method that accepts `int` and `String` arguments.

**Note:** This version of the `printProductReport` method already contains try/catch construct, and it is actually invoking the other version of the `printProductReport`.

  - b. Add a `finally` block after the catch in the `printProductReport` method. Acquire a read lock as the first action within the `try` block inside the `printProductReport` method. Release this lock in the `finally` block.

```
public void printProductReport(int id, String languageTag) {
 try {
 writeLock.lock();
 printProductReport(findProduct(id), languageTag);
 } catch (ProductManagerException ex) {
 logger.log(Level.INFO, ex.getMessage());
 } catch (IOException ex) {
 logger.log(Level.SEVERE,
 "Error printing product report " + ex.getMessage(), ex);
 } finally {
 writeLock.unlock();
 }
}
```

- c. Apply `private` access modifier to the other version of `printProductReport` method.

```
private void printProductReport(Product product,
 String languageTag) throws IOException {
 // existing method logic unchanged
}
```

**Note:** Read lock used by the `printProductReport` method guarantees consistent access to Products map. However, it does nothing to resolve the following problem—what if more than one concurrent invoker attempts to print report for the same product? This would cause concurrent callers to clash. There are several ways of resolving this problem:

- Make this method synchronized so that only one caller would be able to invoke it at any time. However, this would mean that the next invoker would simply overwrite the file produced by a previous caller.
  - Change the logic of the method, so it formats and returns product report as text to the invoker and lets the invoker decide where to print it out.
  - Add an additional parameter that would allow invokers to create filenames that are different for each concurrent caller.
- d. Add an additional `String` parameter called `client` to both versions of the `printProductReport` method.

```
public void printProductReport(int id,
 String languageTag, String client) {
 // existing method logic
}

private void printProductReport(Product product,
 String languageTag, String client) throws IOException {
 // existing method logic
}
```

- e. Pass this new parameter when invoking the `printProductReport` method.

```
printProductReport(findProduct(id), languageTag, client);
```

- f. Open `config.properties` file editor.

- g. Modify `report.file` pattern in the `config.properties` file to utilize extra parameter.

```
report.file=product{0}report{1}.txt
```

- h. Go back to the `ProductManager` class editor.

- i. Continue editing the `printProductReport` method that uses `product`, `languageTag`, and `client` parameters.

- j. Add this `client` parameter to the `format` method call, when constructing a product report file path inside this `printProductReport` method.

```
Path productFile = reportsFolder.resolve(
 MessageFormat.format(config.getString("report.file"),
 product.getId(), client));
```

**Note:** This additional parameter allows product reports to use different names for different concurrent callers and thus prevents these callers to write the same file at the same time.

8. Modify `printProducts` method to use the read lock.

- a. Locate the `printProducts` method.

- b. Surround all existing code inside the `printProducts` method with a `try` block.

Initiate read lock as the first line of code inside this `try` block. Add a `finally` block and release the read lock. No catch block is required, because this method is not expected to throw any exceptions.

```
public void printProducts(Predicate<Product> filter,
 Comparator<Product> sorter,
 String languageTag) {
 try {
 readLock.lock();
 ResourceFormatter formatter =
 formatters.getOrDefault(languageTag,
 formatters.get("en-GB"));
 StringBuilder txt = new StringBuilder();
 products.keySet()
 .stream()
 .sorted(sorter)
 .filter(filter)
 .forEach(p -> txt.append(formatter.formatProduct(p) + '\n'));
 System.out.println(txt);
 } finally {
 readLock.unlock();
 }
}
```

**Note:** This method prints the result it produces to the console. Therefore, even if it is invoked by concurrent callers, the actual printing would be sequential. However, the way this method assembles text to be printed works perfectly fine in the concurrent mode. Potentially design of this method can be improved, so it will format the required text and return it to the invoker instead of directly printing it to the console. Different concurrent invokers can then independently decide what to do with this text.

9. Modify `getDiscounts` method to use the read lock.
  - a. Locate `getDiscounts` method.
  - b. Surround all existing code inside the `getDiscounts` method with a `try` block. Initiate read lock as the first line of code inside this `try` block. Add a `finally` block and release the read lock. No catch block is required, because this method is not expected to throw any exceptions.

```
public Map<String, String> getDiscounts(String languageTag) {
 try {
 readLock.lock();
 // existing method code
 } finally {
 readLock.unlock();
 }
}
```

**Note:** This method returns its result to the invoker instead of directly printing to the console or a file. No concurrent performance bottleneck or shared resource clash is produced. Different concurrent invokers can independently decide what to do with these results.

## Practice 14-3: Simulate Concurrent Callers

---

### Overview

In this practice, you write code in the main method of a Shop class that simulates several concurrent callers that share a single instance of the ProductManager.

Each concurrent client should perform the following operations:

- Generate unique client id, which can be used to distinguish concurrent callers
- Generate random product id
- Select random locale
- Invoke getDiscounts, reviewProduct, and printProductReport operations
- Produce a log of these actions

You also create an ExecutorService to launch these concurrent callers. After that, you need to retrieve action logs produced by each concurrent client and print these logs to the console.

Printing logs to the console is a sequential activity; this is why your concurrent client should not try to print directly to the console.

1. Create a unique identity generator for each concurrent client.

- a. Open `Shop` class editor.
- b. Declare a new variable called `clientCount` type of `AtomicInteger` and initialize it to reference new `AtomicInteger` object with the value of 0. This variable should be declared inside `main` method, on the next line of code after the initialization of the `ProductManager` object.

```
AtomicInteger clientCount = new AtomicInteger(0);
```

**Note:**

- Just like the `ProductManager` object, this variable will be shared between multiple concurrent callers.
- Each concurrent client will increment and get the value of the `clientCount` variable.
- c. Add an import statement for the `AtomicInteger` class.

**Hint:** Right-click anywhere in the source code of the `Shop` class and invoke "Fix Imports" menu.

```
import java.util.concurrent.atomic.AtomicInteger;
```

2. Create new Callable object to represent a concurrent client.

- a. Create new variable called `client`, type of `Callable` interface. Use `String` as generic type for this `Callable` object. Assign a lambda expression that implements `Callable` interface `call` method that returns `String` object. Place this code after the declaration of the `clientCount` variable, inside `main` method.

```
Callable<String> client = () -> {
 // concurrent client logic will be placed here
 return "";
};
```

- b. Add an import statement for the `Callable` interface.

**Hint:** Right-click anywhere in the source code of the `Shop` class and invoke "Fix Imports" menu.

```
import java.util.concurrent.Callable;
```

3. Organize code inside the lambda expression that implements the `Callable` object to represent a concurrent client.

- a. Declare the new `String` variable called `clientId` and initialize it to reference the concatenation of the word "Client" and the next incremented int value produced by the `clientCount` object. Place this code inside the body of the lambda expression.

**Hint:** Use `incrementAndGet` method provided by the `AtomicInteger` object to retrieve the next client id int value.

```
String clientId = "Client "+clientCount.incrementAndGet();
```

- b. Next, declare the new `String` variable called `threadName` and initialize it to reference the current thread name.

**Hints:**

- Use `Thread.currentThread` method to get the current thread object.
- Use `getName` method provided by the `Thread` object to retrieve its name.

```
String threadName = Thread.currentThread().getName();
```

- c. Next, declare new `int` variable called `productId` and assign a randomly generated int number between 101 and 163 to this variable.

**Hints:**

- Use `ThreadLocalRandom.current` method to obtain a reference to the random number generator object that generates independent random values for each thread.
- Use `nextInt` method of the `ThreadLocalRandom` object to generate a random number between 0 and 63.
- Add 101 to the random number.

```
int productId = ThreadLocalRandom.current().nextInt(63)+101;
```

- d. Add an import statement for the `ThreadLocalRandom` class.

**Hint:** Right-click anywhere in the source code of the `Shop` class and invoke "Fix Imports" menu.

```
import java.util.concurrent.ThreadLocalRandom;
```

- e. Next, declare new `String` variable called `languageTag` and assign a randomly selected value from the set of supported locales.

**Hints:**

- There should be at least five supported language tags registered in the `ProductManager`.
- Use `ProductManager.getSupportedLocales` method to retrieve the Set of supported language tags.
- Produce `stream` from this Set.
- Use `skip` method to get random language tag from the stream.
- To determine a number of elements to skip in the stream, use `ThreadLocalRandom.current` method to obtain a reference to the random number generator object that generates independent random values for each thread and `nextInt` method of the `ThreadLocalRandom` object to generate a random number between 0 and 4.
- After skipping a random number of elements in the stream, use `findFirst` method to get the element you currently skipped to.
- `findFirst` method returns an `Optional` object. Use `get` method to retrieve the actual `languageTag` value from it.

```
String languageTag =
 ProductManager.getSupportedLocales()
 .stream()
 .skip(ThreadLocalRandom.current().nextInt(4))
 .findFirst().get();
```

- f. Next, declare new `StringBuilder` variable called `log` and initialize it to reference a new instance of `StringBuilder`.

```
StringBuilder log = new StringBuilder();
```

**Notes:**

- This object represents an activity log that each concurrent client populates with the log of its actions.
  - The contenders of this log must be returned the end of the block of code that implements this lambda expression.
- g. Modify `return` statement in this lambda expression to return `log` content as `String` instead of an empty String.

```
return log.toString();
```

- h. Append text that identifies the start and the end of the log for each concurrent client.

**Hints:**

- Start of the log text should be:

```
clientId+" "+threadName+"\n-\tstart of log\t-\n"
```

- End of the log text should be:

```
"\n-\tend of log\t-\n"
```

- Use append method of the log object to add this text.

- Place this code between the declaration of the `log` variable and the `return` statement.

```
StringBuilder log = new StringBuilder();
log.append(clientId+" "+threadName+"\n-\tstart of log\t-\n");
// calls to ProductManager methods will be added here
log.append("\n-\tend of log\t-\n");
return log.toString();
```

4. Invoke operations upon the `ProductManager` object from the lambda expression that implements the `Callable` object to represent a concurrent client.

**Note:** All code in this part of the practice must be added inside the lambda expression that represents concurrent client, between statements that append the start and the end log messages.

- a. Invoke `getDiscounts` operation upon the `ProductManager` object. This operation uses read lock. It expects you to pass `languageTag` parameter and returns a Map of star rating values and sum of discounts per rating formatted as text. Your task is to join all elements in the returned map into a single string and append this result to the log.

**Hints:**

- Invoke `getDiscounts` method upon the `pm` object, passing randomly selected `languageTag` as parameter.
- Use `entrySet` method to retrieve the Set of keys from the Map returned by the `getDiscounts` method.
- Get `stream` from this set.
- Use `map` method to concatenate each key from the stream with "\t" tab character and the value for each map entry.
- Use `collect` method and a joining `Collector` to assemble stream elements to a single String, using "\n" character between text elements.
- Use `append` method to add result of the stream processing to the `log`.

```
log.append(pm.getDiscounts(languageTag)
 .entrySet()
 .stream()
 .map(entry->entry.getKey()+"\t"+entry.getValue())
 .collect(Collectors.joining("\n")));
```

- b. Invoke `reviewProduct` operation upon the `ProductManager` object. This operation uses write lock. You are going to update a randomly selected product by adding a new review.

**Hints:**

- Invoke `reviewProduct` method upon the `pm` object, passing randomly selected `productId` as parameter.
- Other parameters are the `Rating` and `review comments` `String`.
- You may use any valid rating value and comments.
- Assign the value returned by the `reviewProduct` method to a new `Product` variable.

```
Product product = pm.reviewProduct(productId, Rating.FOUR_STAR, "Yet another review");
```

- c. Append message to the log indicating if the product has been reviewed or not. This message should contain product id value. It should start and end with the "`\n`" new line character.

**Hints:** Use ternary `? :` operator to determine if the `reviewProduct` method has returned the actual product, rather than `null`. This would indicate that the review has been applied successfully.

```
log.append((product != null)
 ? "\nProduct "+productId+" reviewed\n"
 : "\nProduct "+productId+" not reviewed\n");
```

- d. Invoke `printProductReport` operation upon the `ProductManager` object. This operation uses read lock. You are going to request to print the product report for this client, for the randomly selected product and using a randomly selected language tag.

**Hint:** Invoke `printProductReport` method upon the `pm` object, passing `productId`, `languageTag`, and `clientId` parameters.

```
pm.printProductReport(productId, languageTag, clientId);
```

- e. Append message to the log indicating which product report file you have produced. This message should contain client id and product id values.

```
log.append(clientId+" generated report for "+productId+" product");
```

**Note:** You have completed coding the concurrent client activities. Your next tasks will be to trigger these client invocations and process the generated logs.

5. Prepare to invoke five concurrent client Callable objects.
  - a. Declare a new variable called `clients`, type of `List`. Use generics to parameterize this `List` to contain `Callable<String>` objects, matching the definition of your `client` variable. This list should be populated with five references to the `client` variable.

**Hints:**

- To populate `clients` variable, use `Stream.generate` method to produce infinite stream populated with `client` object references.
- Use `limit` method to take only first 5 elements from this stream.
- Use `collect` method and `Collectors.toList` to convert this stream into a list.
- This code should be added after the end of the concurrent `client` lambda expression body, inside the `main` method of the `Shop` class.

```
List<Callable<String>> clients = Stream.generate(() -> client)
 .limit(5)
 .collect(Collectors.toList());
```

- b. Add import statements for the `List` interface and `Stream` class.

**Hint:** Right-click anywhere in the source code of the `Shop` class and invoke "Fix Imports" menu.

```
import java.util.List;
import java.util.stream.Stream;
```

- c. Declare a new variable called `executorService`, type of `ExecutorService`. Initialize this variable to reference a new fixed thread pool of three threads.

```
ExecutorService executorService =
 Executors.newFixedThreadPool(3);
```

**Note:** You have created a list of 5 Callable objects that you are going to execute using a pool of 3 threads. This means that you will force the thread scheduler to make these Callable objects to time-share the same threads.

- d. Add import statements for the `ExecutorService` and `Executors` classes.

**Hint:** Right-click anywhere in the source code of the `Shop` class and invoke "Fix Imports" menu.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

6. Trigger concurrent execution of all client objects in the clients list and process the resulting logs, produced by each Callable client.
  - a. Trigger concurrent execution of all client objects using `invokeAll` method provided by the executor service. `invokeAll` method accepts the list of Callable objects as parameter and produces a list of `Future` objects, which contains results of concurrent clients executions. Assign the value returned by the `invokeAll` method to a new variable called `results`, type of `List`. Use generics to parameterize this `List` to contain `Future<String>` objects.

```
List<Future<String>> results = executorService.invokeAll(clients);
```

- b. Add an import statement for the `Future` class.

**Hint:** Right-click anywhere in the source code of the `Shop` class and invoke "Fix Imports" menu.

```
import java.util.concurrent.Future;
```

**Note:** `invokeAll` method can produce `InterruptedException`. You need to create a `try-catch` construct around this method call to intercept this exception.

- c. Add `try-catch` construct around the line of code that calls the `invokeAll` method.

**Hint:** Right-click on the left side of the line of code that is producing the exception in the `reviewProduct` method and invoke "Surround Statement with try-catch" menu.

```
try {
 List<Future<String>> results = executorService.invokeAll(clients);
 // code that handles results will be added here
} catch (InterruptedException ex) {
 Logger.getLogger(Shop.class.getName()).log(Level.SEVERE, null, ex);
}

d. Customize exception handling logic inside the catch block. Replace null value in the log method call with the message that indicates a failure in the executor service clients invocation.

Logger.getLogger(Shop.class.getName())
 .log(Level.SEVERE, "Error invoking clients", ex);

e. Inside the try block, after the execution of invokeAll method, add code that prevents executor service from accepting any further execution requests.

executorService.shutdown();

f. Next, process the concurrent execution results by creating a stream from the results list, stepping through all elements in this stream using forEach method.

results.stream().forEach(result->
 // result processing will be added here
});
```

**Notes:**

- Your Callable client objects were designed to concurrently access ProductManager object.
  - Each one of these clients has independently generated a String containing the log of its activities.
  - ExecutorService has placed these String logs into a List of Future objects.
  - You are now going to process these logs sequentially.
  - Your next task will be to extract String value out of every Future object returned from the executor service and print it on the console.
- g. Extract String value from every future result object and print it on the console.

```
results.stream().forEach(result->{
 System.out.println(result.get());
});
```

**Notes:**

- This code does not yet compile, because `get` method may produce `InterruptedException` and `ExecutionException` that you are not yet intercepting.
  - These exceptions can be produced if you attempt to get results that were supposed to be produced by callable objects that were aborted or interrupted.
  - This may look a little strange—it seems like your code is inside the try block that actually has a catch that intercepts the `InterruptedException`. However, what you need to realize is that the `get` method is invoked inside a lambda expression, which technically is another object (in this case, it is an implementation of `Consumer` interface). So whatever code is in the body of this lambda expression, it is really part of the `accept` method that you are implementing here.
  - You need to add another try-catch construct inside the lambda expression body to intercept relevant checked exceptions.
- h. Add `try-catch` construct around the line of code that calls the `get` method.

**Hint:** Right-click on the left side of the line of code that is producing the exception in the `reviewProduct` method and invoke the "Surround Statement with try-catch" menu.

```
try {
 System.out.println(result.get());
} catch (InterruptedException ex) {
 Logger.getLogger(Shop.class.getName()).log(Level.SEVERE, null, ex);
} catch (ExecutionException ex) {
 Logger.getLogger(Shop.class.getName()).log(Level.SEVERE, null, ex);
}
```

- i. Customize exception handling logic. Merge these two exception handlers into one catch block and replace `null` value in the `log` method call with the message that indicates a failure to retrieve the result of the concurrent client execution.

```
try {
 System.out.println(result.get());
} catch (InterruptedException | ExecutionException ex) {
 Logger.getLogger(Shop.class.getName())
 .log(Level.SEVERE, "Error retrieving client log", ex);
}
```

- j. Compile and run your application.

**Hint:** Click the "Run Project" toolbar button.



**Notes:**

- Product id values and locales were selected at random.
- The order of concurrent execution is stochastic.
  - Logs appear in unpredictable order.
  - Threads are allocated to your clients in unpredictable order.
- You may repeat the execution of the application several times, and each time the order of logs and relationship between threads and clients may be different.
- You may also navigate to the `/home/oracle/labs/reports` folder and observe product report files that now contain client id as part of the name.
- Every time you run application client, enumeration starts from 1.
- On different application runs, client and product ids usage may coincide. In this case, a client in a later application execution would simply overwrite previously generated product report file. However, this cannot actually happen on the same program run, because client id values are unique within a given program execution.
- When a client adds a new review to a product, you can see this review printed into a product report file. However, this change is not saved into the product and reviews csv files located in the data folder.

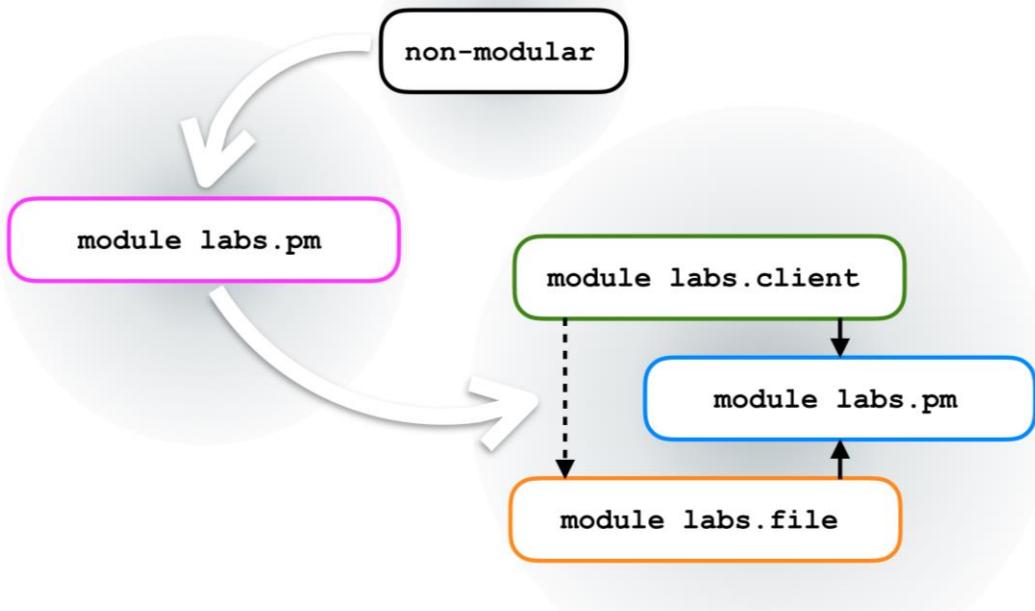
## **Practices for Lesson 15: Java Modules**

## Practices for Lesson 15: Overview

### Overview

In these practices, you will deploy ProductManagement Application in non-modular and modular formats. You then create another version of the ProductManagement Application that splits its code into several modules, to improve application structure and allow it to be extended.

### Product Management Application Migration



## Practice 15-1: Convert ProductManagement Application into a Module

### Overview

In this practice, you will deploy ProductManagement Application using a non-modular format. Then convert ProductManagement Application into a module, by providing a module-info descriptor and establishing the required dependencies. You also change NetBeans project settings to enable deployment of this application as a jimage. Deploy and execute this modular jimage version of the application.

### Assumptions

- JDK 11 is installed.
- NetBeans 11 is installed.
- You have completed Practice 14 or start with the solution for Practice 14 version of the application.

### Tasks

1. Prepare the practice environment.

#### Notes:

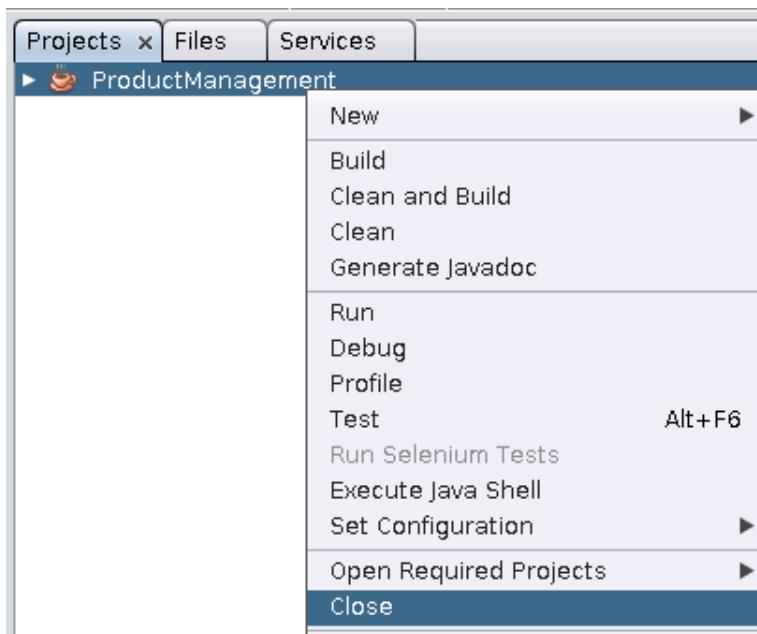
- You may continue to use same NetBeans project as before, if you have successfully completed the previous practice. In this case, proceed directly to Practice 15-1, step 2.
- Alternatively, you can open a fresh copy of the NetBeans project, which contains the completed solution for the previous practice.

- a. Open NetBeans (if it is not already running).

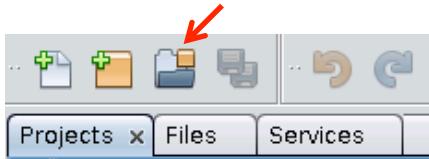


- b. Close the currently opened ProductManagement project.

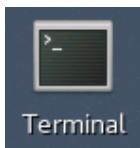
**Hint:** Right-click on the ProductManagement project and invoke "Close" menu.



- c. Open the Solution for Practice 14 version of the ProductManagement project.
- Hint:** Use File -> Open Project menu or click on the "Open Project" toolbar button.



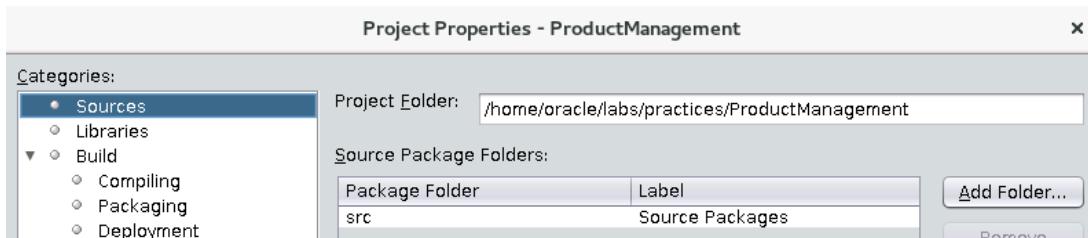
- d. Navigate to /home/oracle/labs/solutions/practice14 folder and select ProductManagement project.
- e. Click "Open Project".
2. Execute non-modular version of the ProductManagement application.
- Note:** Currently, your application is not using Java modules. NetBeans automatically produces non-modular jar deployment every time you build your application project.
- a. Open the terminal window.  
(You may continue to use the opened terminal window from the earlier practice.)



- b. In the terminal window, change the directory to the root folder of your project.

**Hints:**

- The following steps of the practice assume that your project folder is: /home/oracle/labs/practices/ProductManagement .
- You may check the actual location of your project folder by right-clicking on the ProductManagement project and invoke Properties menu. The Project folder field indicates the correct location.



- Change the directory to your project folder.
- ```
cd /home/oracle/labs/practices/ProductManagement
```

c. Run the non-modular version of your application.

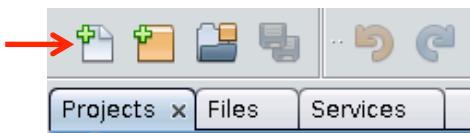
```
java -jar ./dist/ProductManagement.jar
```

Note:

- This application can be deployed and distributed by copying this jar file to target computers.
- You also need to install JDK on these machines.

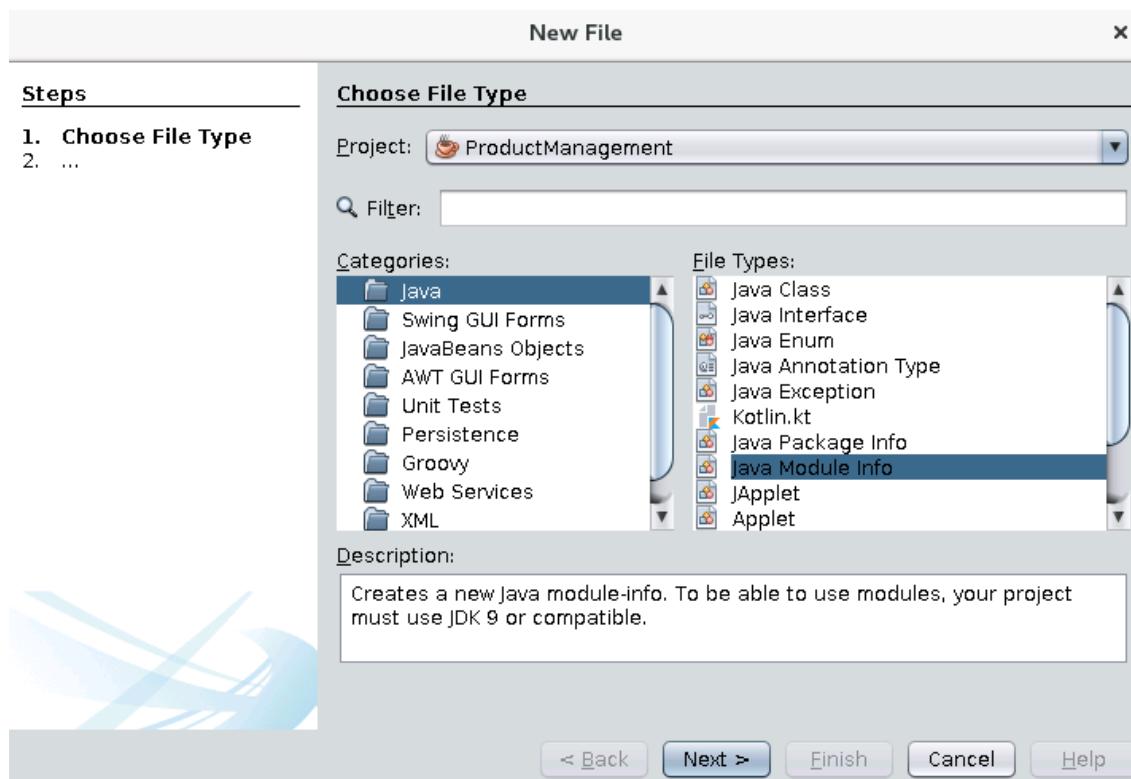
3. Create module descriptor for the ProductManager application.

a. Create new Java Module info class.



Hint: Use File -> New File menu or click the "New File" toolbar button.

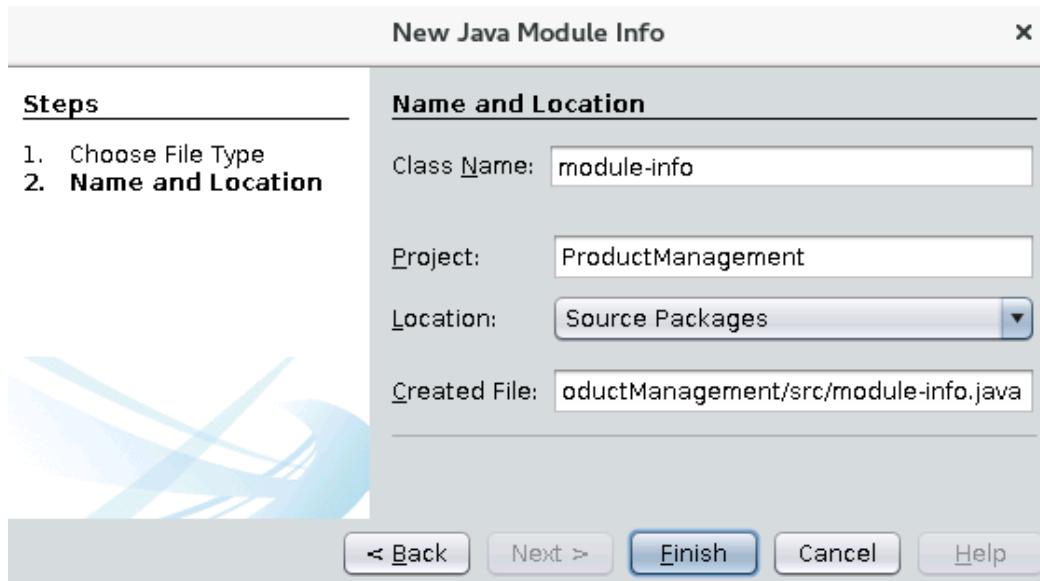
b. Select "Java" Category and "Java Module Info" as the file type.



c. Click "Next".

- d. Set the following class properties:

- Class Name: module-info
- Project: ProductManagement
- Location: Source Packages



- e. Click "Finish."

Notes:

- Once you have created a `module-info` class, classes `Shop` and `ProductManager` will no longer compile.
- This is because both of these classes use Java Logging API, which is not part of the `java.base` module.
- You will have to declare a dependency with the `java.logging` module.

4. Modify module descriptor for the `ProductManager` application.

- a. Open the `module-info` class editor.
- b. Change the module name to `labs.pm`.

Note: It is considered to be a good practice to name the module after the root package contained within the module.

- c. Add a dependency with Java Logging API to the `module-info`.

```
module labs.pm {
    requires java.logging;
}
```

- d. Recompile `ProductManagement` project.

Hint: Use Run->Clean and Build Project menu or a toolbar button.

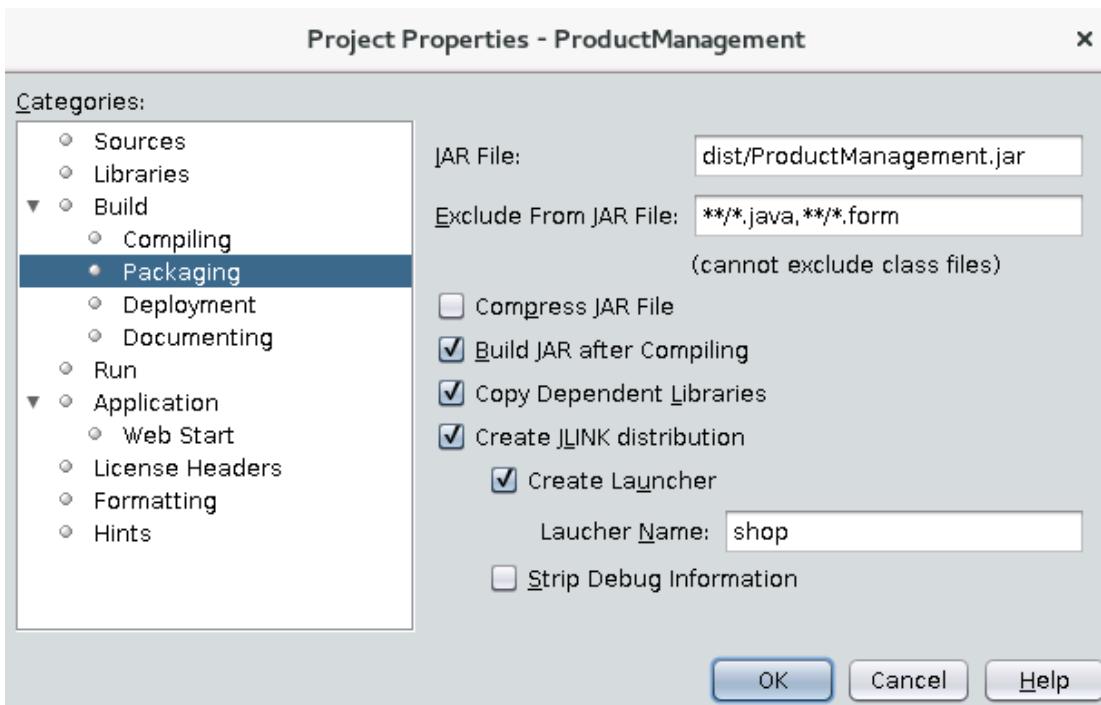


Note: All classes should now successfully compile.

5. Deploy ProductManager application in jimage format.
 - a. Enable jlink distribution for the ProductManagement application and define application launcher.

Hints:

 - Right-click on the ProductManagement project and invoke the Properties menu.
 - Navigate to Build > Packaging section.
 - Select Create JLINK distribution check box.
 - Set Launcher name property: shop.
 - Optionally, you may select Compress Jar File and Strip Debug Information check boxes as well.



- b. Click "OK".
- c. Recompile ProductManagement project.

Hint: Use Run->Clean and Build Project menu or a toolbar button.

Note: NetBeans automatically creates jar files and jimage distribution when you execute Clean and Build action.

6. Test ProductManager application deployment.
 - a. Open Terminal. (You may use the already opened terminal window.)
 - b. In the terminal window, change the directory to the root folder of your project.

Hint: The following steps of the practice assume that

```
/home/oracle/labs/practices/ProductManagement is your project folder
cd /home/oracle/labs/practices/ProductManagement
```

- c. Check the size of the jimage folder.

```
du -sh ./dist/jlink/
```

Notes:

- The size of the entire application deployment is around 48 megabytes, which includes the actual java runtime for Linux 64 bit platform.
- It could be even smaller if you have selected Compress Jar File and Strip Debug Information check boxes in project properties.

- d. List modules included in the jimage deployment.

```
./dist/jlink/ProductManagement/bin/java --list-modules
```

Note: Your deployment comprises `java.base`, `java.logging`, and `labs.pm` modules.

- e. Execute ProductManagement application using the launcher.

```
./dist/jlink/ProductManagement/bin/shop
```

Notes:

- You have used `shop` launcher script to execute this application.
- Alternatively, you may run this application using java executable embedded in the module directly, passing the module name that contains your main class:

```
./dist/jlink/ProductManagement/bin/java -m labs.pm
```

Note:

- This application can be deployed and distributed by copying the `ProductManagement` folder to target computers.
- There is no need to install JDK on these machines.

Practice 15-2: Separate Application into Several Modules

Overview

In this practice, you break ProductManagement application into several modules, to accommodate a more flexible design for the purposes of future extensibility.

The proposed change in application design breaks ProductManagement application into the following modules:

- Application Client module that deals with the front-end part of the application
- Service module that describes business logic capabilities and data structures
- Service implementation module that implements business logic using filesystem storage

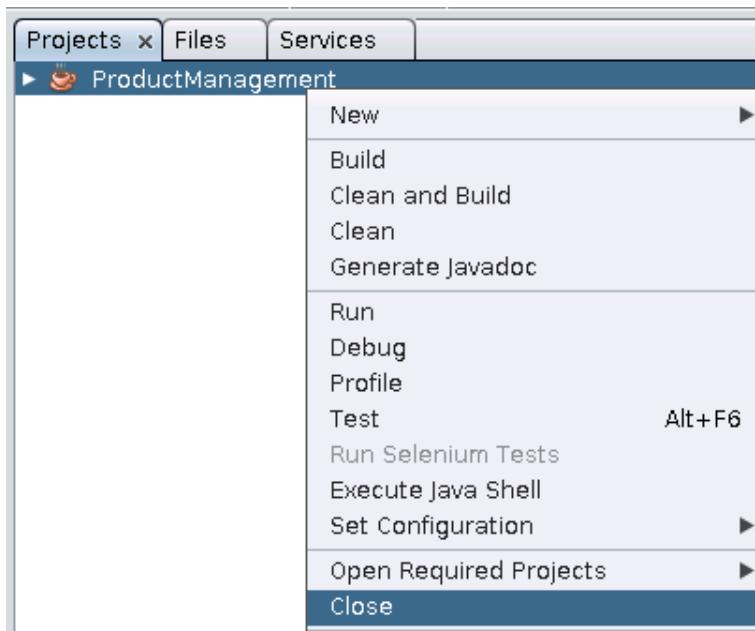
This design allows production of alternative business service implementation modules, for example, using database instead of filesystem storage.

1. Open the refactored version of the ProductManagement project.

Note: This is a mandatory practice step—you cannot continue to use your current project, because application classes were refactored to allow application to be divided into several modules with improved design extensibility.

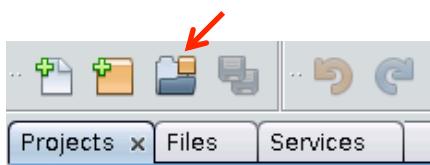
- a. Close the currently opened ProductManagement project.

Hint: Right-click on the ProductManagement project and invoke "Close" menu.

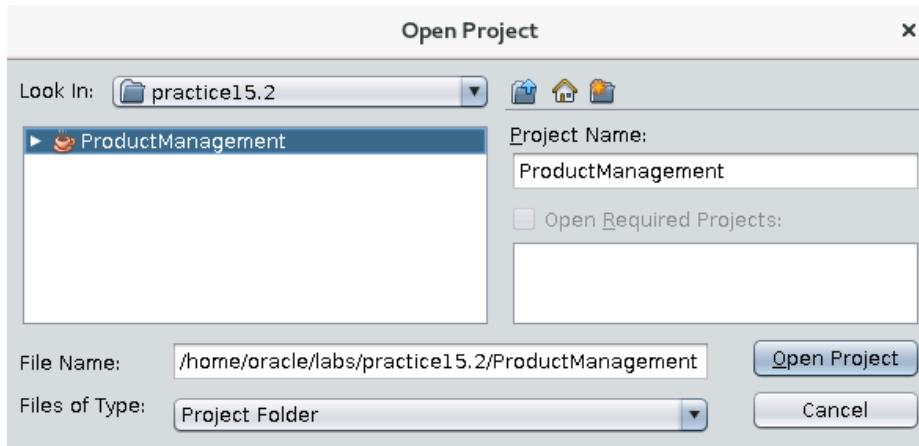


- b. Open Practice 15.2 version of the ProductManagement project.

Hint: Use File -> Open Project menu or click the "Open Project" toolbar button.



- c. Navigate to /home/oracle/labs/practice15.2 folder and select ProductManagement project.



- d. Click "Open Project".
2. Analyze changes that were made to the ProductManagement application structure.
- A number of changes had to be applied to the ProductManagement application structure to allow a more flexible design and potential extensibility. The version of the application that you just opened has been modified in the following steps:
- ProductManager class has been repurposed to contain only the code related to the business logic of the application, and all data presentation code has been moved to the front-end part of the application.
 - ResourceFormatter class has been refactored as a normal class instead of the static nested. This allows its logic to be reused within the front-end tier of the application.
 - All formatting capabilities were removed from the ProductManager class. This allows client to choose the way in which information is presented.
 - Product report printing functionality has been generalized to print any formatted text, and this logic has been moved from ProductManager class to the client part of the application, represented by the Shop class.
 - All operations that were printing directly to the console were removed from the ProductManager, since this can now be achieved by the client.
 - ProductManager class has been split into an interface called ProductManager that describes the required business methods and an implementation class called ProductFileManager that contains the filesystem-based implementation of these methods.
 - Interface ProductManager now describes the following public abstract methods:

```
Product createProduct(int id, String name, BigDecimal price,
                      Rating rating) throws ProductManagerException;
Product createProduct(int id, String name, BigDecimal price,
                      Rating rating, LocalDate bestBefore) throws ProductManagerException;
```

```

Product reviewProduct(int id, Rating rating, String comments)
                    throws ProductManagerException;
Product findProduct(int id) throws ProductManagerException;
List<Product> findProducts(Predicate<Product> filter)
                    throws ProductManagerException;
List<Review> findReviews(int id) throws ProductManagerException;
Map<Rating, BigDecimal> getDiscounts() throws ProductManagerException;

```

All these methods indicate that they may produce a `ProductManagerException`. This allows different implementations of the `ProductManager` interface to generate and capture different exceptions (such as IO or SQL exceptions), but throw the same exception type to the client so that the client remains unaware of the specific nature of the implementation of this interface that the client may choose to use. Also, notice that these methods do not operate on formatted text, leaving such presentation functionality to the client.

- Class `ProductFileManager` is designed to implement these methods and in addition provides operations that access the filesystem storage of products and reviews.
- Application code was separated into the following packages:
 - Package `labs.client` containing `ResourceFormatter` and `Shop` classes as well as resource bundles needed for formatting and localisation purposes
 - Package `labs.pm.service` containing `ProductManager` interface and `ProductManagerException` class
 - Package `labs.pm.data` containing `Product`, `Food`, `Drink`, `Review` classes, `Rating` enumeration, and a `Rateable` interface
 - Package `labs.file.service` containing `ProductFileManager` class and a config file needed to implement filesystem-based management of application data

Notes:

- You may open editors for these files to study code modifications described above.
- This application is still described as a single module, but it is now ready to be segregated into several modules. Your next task will be to perform this segregation.

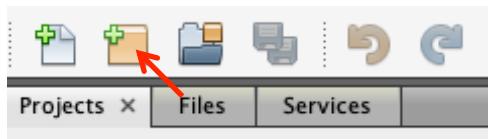
3. Create a new Java Modular Project that will contain three new modules for the `ProductManagement` application.

Notes:

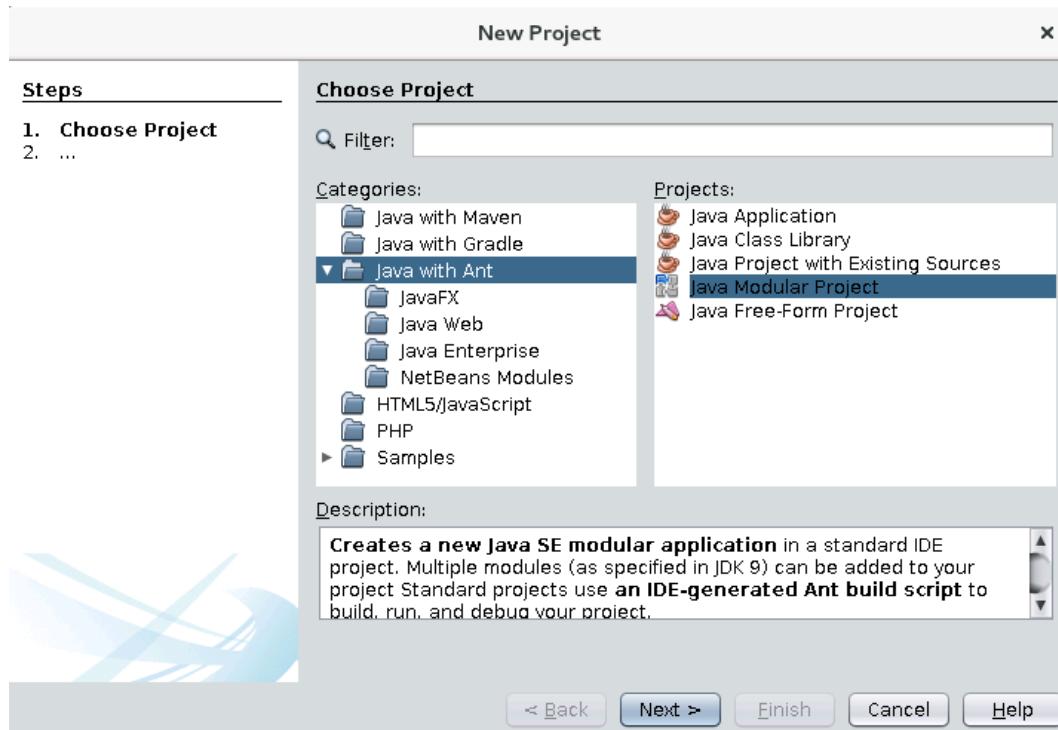
- You can create a regular NetBeans projects—one per module, and link them together using module path settings in the libraries section of the project properties. NetBeans can even automatically generate a jimage based on these projects, except linking dependent service implementation modules, which would require a manual deployment.
- NetBeans also supports Java Modular Project type, which allows you to create one or more modules inside the same project and fully supports jimage deployment.
- Your next task is to produce a new modular project that is designed to contain several modules and deploy them as jimage.

- a. Create new Java Modular Project.

Hint: Use File -> New Project menu or click the "New Project" toolbar button.

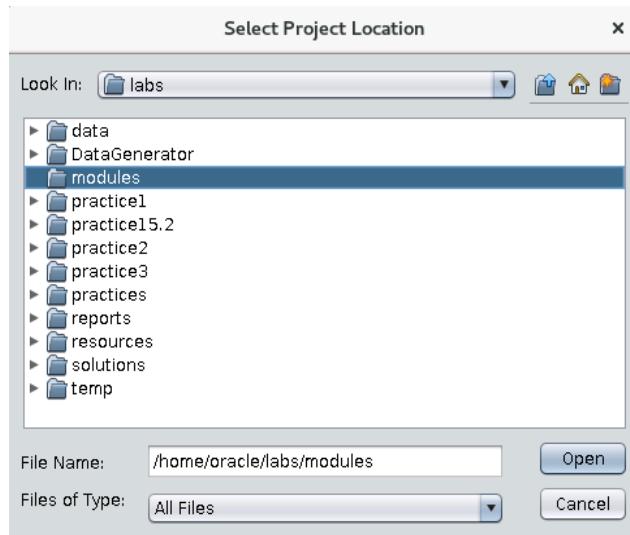


- b. Select "Java with Ant" Category and "Java Modular Project" as the project type.



- c. Click "Next".

- d. Set the following project properties:
- Project Name: ProductManagement
 - Project Location: /home/oracle/labs/modules
(use Browse button to select folder)

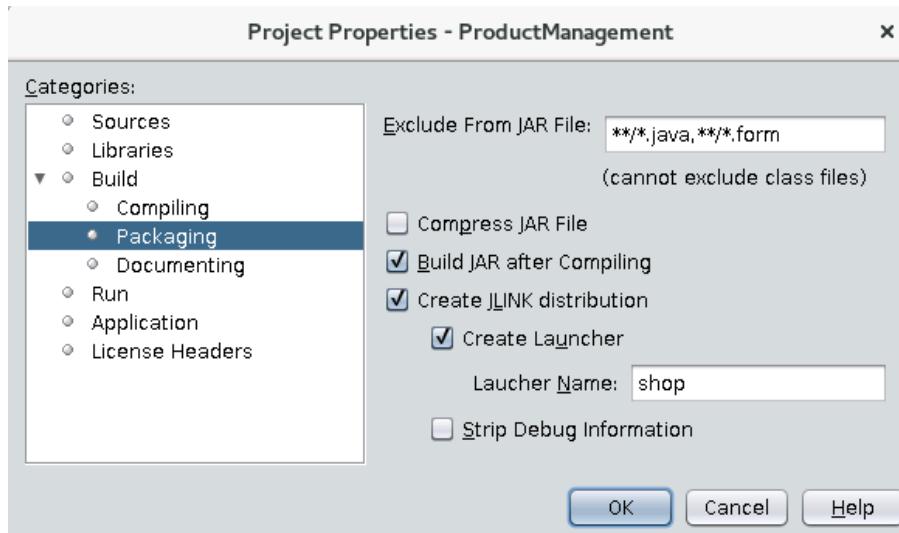


- Project Folder: /home/oracle/labs/modules/ProductManagement
- Platform: JDK 11 (use drop-down list to select platform)



- e. Click "Finish".
4. Set up ProductManagement Modular Project properties.
- Enable jlink distribution for the ProductManagement Modular project and define application launcher
- Hints:**
- Right-click on the ProductManagement modular project and invoke Properties menu.
 - Navigate to the Build > Packaging section.
 - Select Create JLINK distribution check box.

- Set Launcher name property: shop.
- Optionally, you may select Compress Jar File and Strip Debug Information check boxes as well.



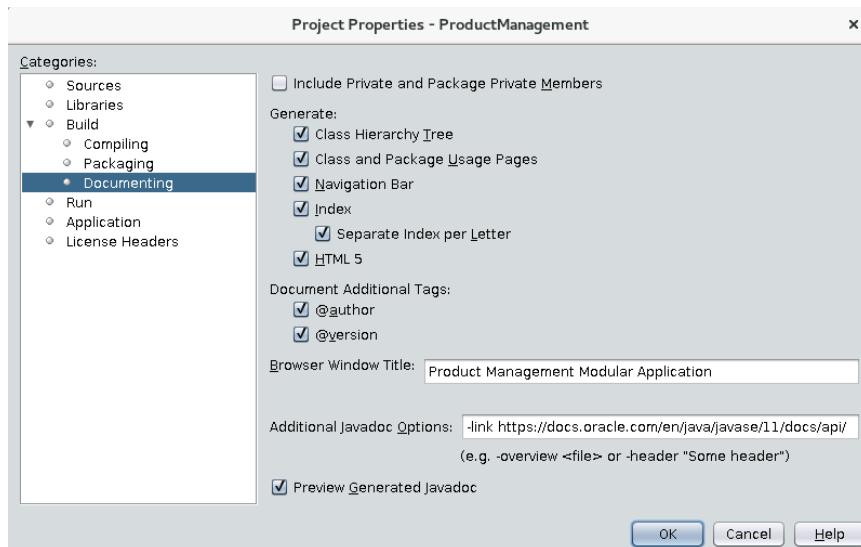
- b. Continue editing ProductManagement modular project properties. Configure documentation settings.

Hints:

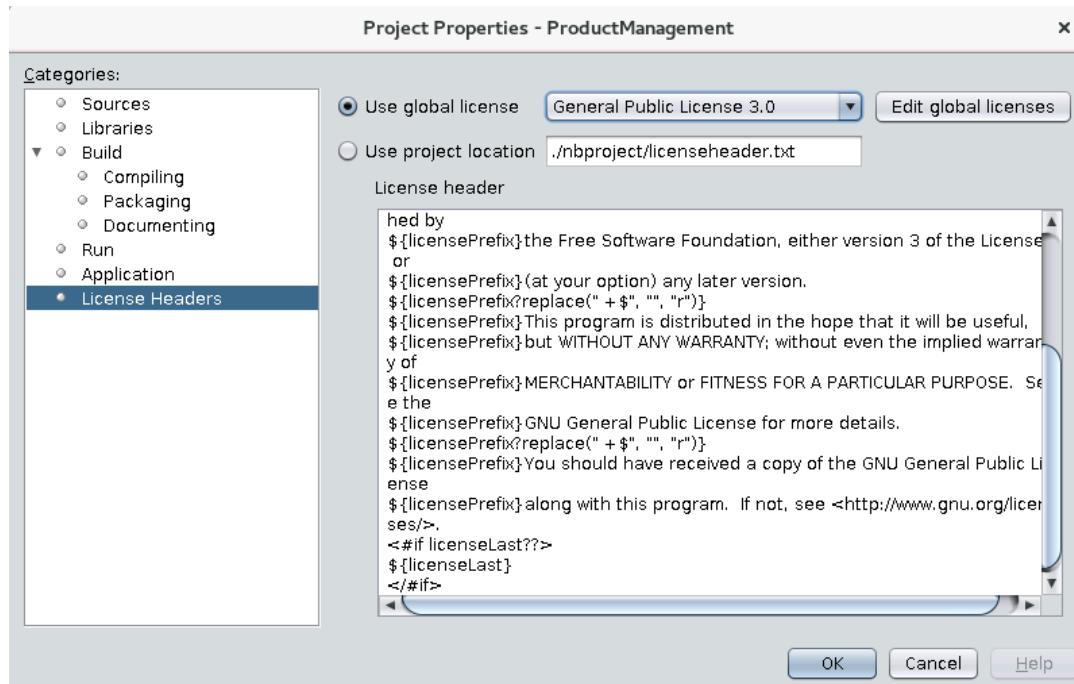
- Navigate to Build->Documenting section.
- Select HTML5, @author, and @version check boxes.
- Set Browser Window Title: Product Management Modular Application.
- Additional Javadoc Options:

```
-link https://docs.oracle.com/en/java/javase/11/docs/api/  
-J-Dhttps.proxyHost=ges-proxy.us.oracle.com  
-J-Dhttps.proxyPort=80
```

Note: Proxy parameters in additional Javadoc options are required for this course environment.

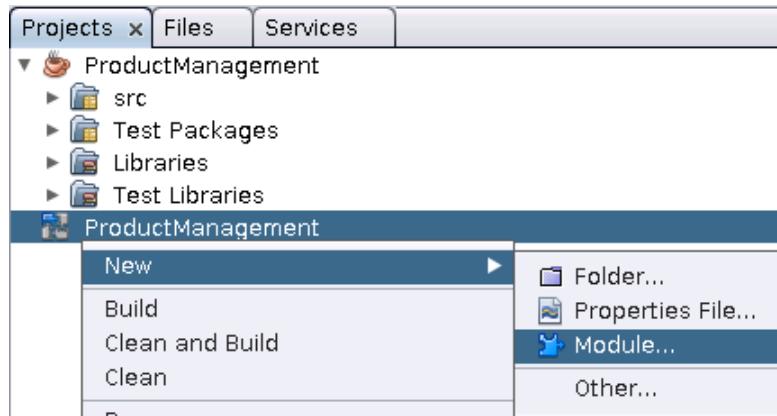


- c. Navigate to the Licence Headers section.
- d. Select General Public Licence 3.0 from the drop-down list of global licenses.
- e. Click "OK".



- 5. Create three new modules for the ProductManagement Modular Project.
 - a. Create new labs.client Java Module inside the ProductManagement Modular Project.

Hint: Right-click on the ProductManagement Modular Project and invoke "New Module" menu.

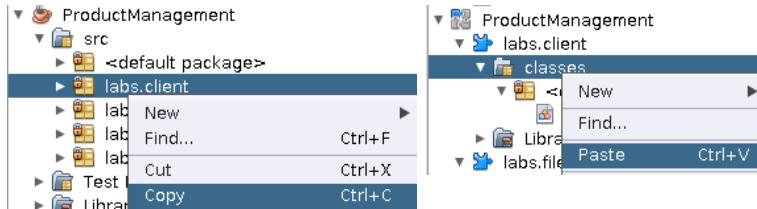


- b. Set the following project properties:
- **Module Name:** labs.client
 - **Project:** ProductManagement
 - **Location:** Source Modules
 - **Created Module:**
/home/oracle/labs/modules/ProductManagement/src/labs.client

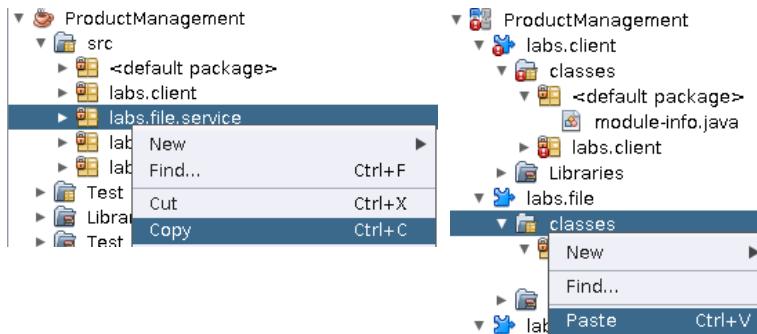


- c. Click "Finish".
- d. Create new `labs.pm` Java Module inside the ProductManagement Modular Project.
Hint: Right-click on the ProductManagement Modular Project and invoke "New Module" menu.
- e. Set the following project properties:
- **Module Name:** labs.pm
 - **Project:** ProductManagement
 - **Location:** Source Modules
 - **Created Module:**
/home/oracle/labs/modules/ProductManagement/src/labs.pm
- f. Click "Finish".
- g. Create new `labs.file` Java Module inside the ProductManagement Modular Project.
Hint: Right-click on the ProductManagement Modular Project and invoke "New Module" menu.
- h. Set the following project properties:
- **Module Name:** labs.file
 - **Project:** ProductManagement
 - **Location:** Source Modules
 - **Created Module:**
/home/oracle/labs/modules/ProductManagement/src/labs.file
- i. Click "Finish".

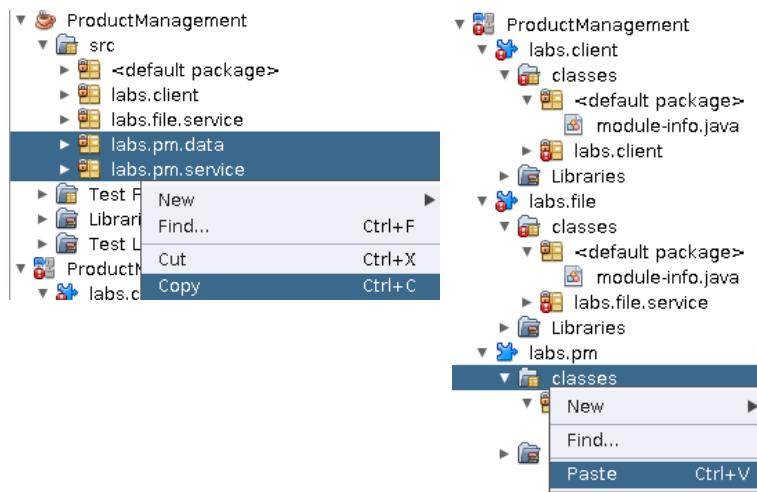
6. Copy code from **ProductManagement** project into **ProductManagement Modular** project.
 - a. Right-click on the `labs.client` package in the **ProductManagement** project and invoke "Copy" menu.
 - b. Right-click on the `classes` node located inside the `labs.client` module in the **ProductManagement Modular** project and invoke "Paste" menu.



- c. Right-click on the `labs.file.service` package in the **ProductManagement** project and invoke "Copy" menu.
- d. Right-click on the `classes` node located inside the `labs.file` module in the **ProductManagement Modular** project and invoke "Paste" menu.



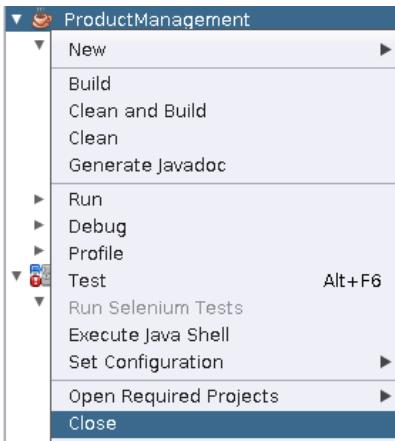
- e. Select `labs.pm.data` and `labs.pm.service` packages in the **ProductManagement** project (use **CTRL+click**) Right-click on these selected packages and invoke "Copy" menu.
- f. Right-click on the `classes` node located inside the `labs.pm` module in the **ProductManagement Modular** project and invoke "Paste" menu.



Note: You have copied all of your classes into the ProductManagement Modular project. You may close this project now.

- g. Close ProductManagement project.

Hint: Right-click on the ProductManagement project and invoke "Close" menu.



7. Describe module exports and dependencies in the ProductManagement Modular Project.
 - a. Open `module-info` file located inside the `labs.pm` module.
 - b. Export two packages `labs.pm.service` and `labs.pm.data`

```
module labs.pm {
```

```
    exports labs.pm.service;
    exports labs.pm.data;
```

```
}
```

- c. Open `module-info` file located inside the `labs.file` module.

- d. Describe that this module requires `java.logging` and `labs.pm` modules and that it provides an implementation for the `ProductManager` interface with `ProductFileManager` class. Use full package prefixes to qualify interface and class names.

```
module labs.file {
```

```
    requires java.logging;
    requires labs.pm;
```

```
    provides labs.pm.service.ProductManager
        with labs.file.service.ProductFileManager;
```

```
}
```

- e. Open `module-info` file located inside the `labs.client` module.
- f. Describe that this modules requires `java.logging` and `labs.pm` modules and that it uses the `ProductManager` service. Use full package prefixes to qualify the interface name.

```
module labs.client {
    requires java.logging;
    requires labs.pm;
    uses labs.pm.service.ProductManager;
}
```

8. Enable Shop class lookup ProductManager service.
 - a. Open `Shop` class editor.
 - b. Remove import statement for `labs.file.service.ProductFileManager` class.
 - c. Add a new line of code, just before the line of code that creates an instance of the `ProductManager` object inside the `main` method of the `Shop` class.
 - d. On this new line of code, create a new variable called `serviceLoader` type of `ServiceLoader`. Use `ProductManager` interface as a generic type for the `ServiceLoader` object. Initialize this variable using `ServiceLoader.load` method, passing `ProductManager.class` as an argument.

```
ServiceLoader<ProductManager> serviceLoader =
    ServiceLoader.load(ProductManager.class);
```

- e. Add an import statement for the `java.util.ServiceLoader` class.

Hint: Right-click anywhere in the source code of the `Shop` class and invoke "Fix Imports" menu.

```
import java.util.ServiceLoader;
```

- f. Modify initialization of the `ProductManager pm` variable, using `serviceLoader` object.

Hints:

- Use `findFirst` operation to get an `Optional` object that contains the first available implementation of the `ProductManager` service.
- Use `get` operation to get the actual implementation from the `Optional` object.
- Assign the result to the `pm` variable.

```
ProductManager pm = serviceLoader.findFirst().get();
```

Notes:

- At the moment, only one implementation of the `ProductManager` service is provided by the `ProductFileManager` class located in the `labs.file` module.
- Additional implementations may be supplied later.
- `ServiceLoader` provides an ability to get a stream of all available service implementations.
- The following code example demonstrates the way you may retrieve the complete list of all available implementations of a service:

```
serviceLoader.stream().forEach(x->System.out.println(x));
```

- g. Recompile ProductManagement Modular project.

Hint: Use Run->Clean and Build Project menu or a toolbar button.



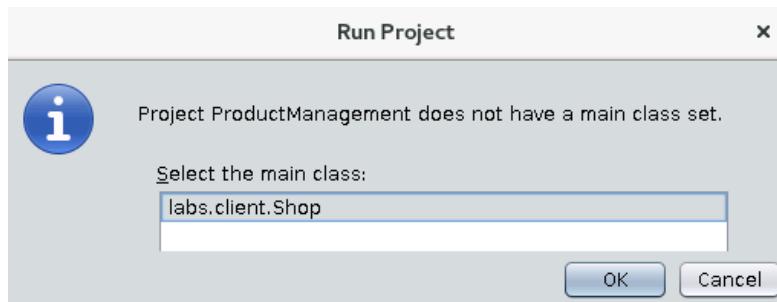
Notes:

- All classes should now successfully compile.
- You may get warning indicators on the classes inside this project. If you do, simply open such classes in the editor and close the editor. The warning should disappear. This issue is caused by NetBeans not parsing files that were copied from another project and singling errors where there are none.
- Modular project rebuild may also produce warning messages in the NetBeans log: "Warning: Nashorn engine is planned to be removed from a future JDK release." This is a bug in the nb-javac plugin, which is used to assemble and execute modular projects. However, this issue does not cause any ill effects and can be ignored.

- h. Compile and run your application.

Hints:

- Click on the "Run Project" toolbar button.
- Run Project Dialog box will pop up and request to confirm the name of the main class.
- Confirm that you are using class labs.client.Shop as your main class by clicking "OK" button in this dialog.



Notes:

- Observe the program output printed to the console.
- Rebuild project deployment again, once the main class has been selected.

- i. Recompile ProductManagement Modular project.

Hint: Use Run->Clean and Build Project menu or a toolbar button.



9. Test ProductManager Modular project deployment.
 - a. Open Terminal. (You may use the already opened terminal window.)
 - b. In the terminal window, change the directory to the root folder of your project:
`cd /home/oracle/labs/modules/ProductManagement`
 - c. Check the size of the jimage folder.
`du -sh ./dist/jlink/`

Notes:

- The size of the entire application deployment is around 48 megabytes, which includes the actual java runtime for Linux 64 bit platform.
 - It could be even smaller if you have selected Compress Jar File and Strip Debug Information check boxes in project properties.
 - There is no tangible size difference between packaging all classes to same or different modules.
 - This application can be deployed and distributed by copying the ProductManagement folder located under this path
`/home/oracle/labs/modules/ProductManagement/dist/jlink/`
 to target computers.
- There is no need to install JDK on these machines.
- However, in addition, a small change in the way this application manages its configuration would be required. At this point, configuration is stored in properties files inside actual deployed modules and is loaded using ResourceBundle class. This should really be changed to a properties file located in a regular folder outside the module itself and loaded using Properties class. This was the design decision taken as part of Practice 12. See notes in Practice 12.2, step 2.e, for more information.

- d. List the modules included in the jimage deployment.

```
./dist/jlink/ProductManagement/bin/java --list-modules
```

Note: Your deployment comprises `java.base`, `java.logging`, `labs.client`, `labs.file`, and `labs.pm` modules.

- e. Execute ProductManagement application using the launcher.

```
./dist/jlink/ProductManagement/bin/shop
```

Notes:

- You have used `shop` launcher script to execute this application.
- Alternatively, you may run this application using java executable embedded in the module directly, passing the module name that contains your main class, followed by the main class package and class name:

```
./dist/jlink/ProductManagement/bin/java -m
labs.client/labs.client.Shop
```

10. Investigate module dependencies using visual graphs.
 - a. Return to NestBeans.
 - b. Open `module-info.java` files from `labs.client`, `labs.pm`, and `labs.file` modules.
 - c. Click the "Graph" button on each of these files to observe module dependencies for each of your modules.

