ORACLE University

Integrated Cloud Applications & Platform Services

Java SE: Programming II

Activity Guide

D102474GC10

Edition 1.0 | December 2018 | D105724

Learn more from Oracle University at **education.oracle.com**

ORACLE

Authors

Kenny Somerville
Anjana Shenoy
Nick Ristuccia

Technical Contributors and Reviewers

Joe Greenwald
Jeffrey Picchione
Joe Boulenouar
Steve Watts
Pete Iaseau
Henry Jen
Nick Ristuccia
Alex Buckley
Vasily Strelnikov
Aurelio García-Ribeyro
Stuart Marks
Geertjan Wielenga
Mike Williams

Editors

Moushmi Mukherjee
Raj Kumar

Graphic Designers

Anne Elizabeth
Yogita Chawdhary
Kavya Bellur

Publishers

Asief Baig
Jayanthy Keshavamurthy

Table of Contents

Practices for Lesson 1: Introduction

Practices for Lesson 1: Overview

Overview

In these practices, you explore the systems and primary tools that are used throughout the course. This is to ensure the tools and your Oracle Linux lab environment are working properly.

Practice 1-1: Log In to Oracle Linux

Overview

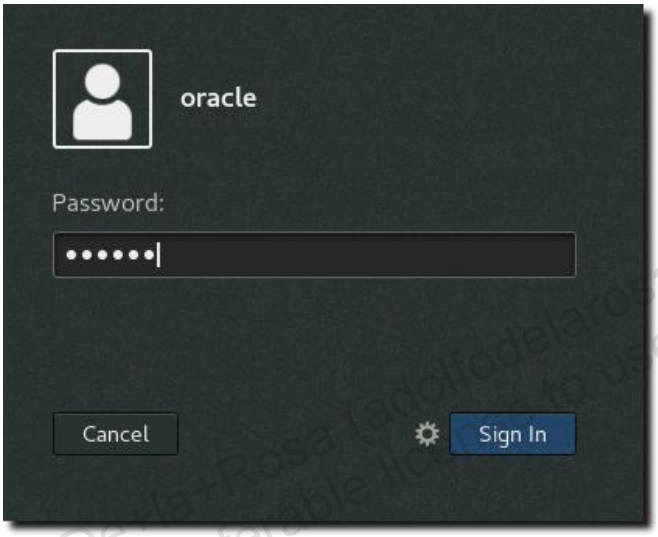
In this practice, you log in to the Oracle Linux operating system.

Assumptions

Oracle Linux 7.5 is installed on your system, and it is on and functioning.

Tasks

1. At the login screen, click the username: ***oracle***
2. Your instructor will give you the password to use. Enter the password and click **Sign In**.



3. On login you should see a screen similar to the one shown below.

Practice 1-2: Open Terminal Windows in Oracle Linux

Overview

In this practice, you open a terminal window in Oracle Linux.

Assumptions

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

Tasks

1. Double click the **Terminal** on the desktop. Alternatively, you may select from the menu **Applications > System Tools > Terminal**.
2. A terminal session should start.
3. Keep the terminal open. You'll need it for the next practice.

Note: For quick reference, we've compiled a list comparing Windows/DOS commands with UNIX commands for the terminal window.

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

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

Practice 1-3: Verify the Version of Java

Overview

In this practice, you use the terminal window to verify the version of Java and check to see it's installed properly.

Assumptions

You are logged in to Oracle Linux, are running a Gnome Desktop, and completed the previous practice.

Tasks

1. In the terminal, type:

```
java -version
```

2. Observe the output. The terminal should report back a Java run time and HotSpot compiler of Java SE 11.0.1 or later.

Practice 1-4: Open a Text File in Oracle Linux

Overview

In this practice, you will use the Nautilus file manager to explore directories and open a text file.

Assumptions

You are logged in to Oracle Linux.

Tasks

1. Open the **oracle's Home** directory by double-clicking its folder on the desktop. This directory will display in Linux's Nautilus file manager window.
2. Open the **labs** directory. This is where you'll find all the labs for this course.
3. Open the **01_Intro** directory.
4. Open the **practices** directory.
5. Open the **read.txt** file by double-clicking the icon in this folder. The file opens in a text editor.
6. Close the text editor and any Nautilus windows.

Practice 1-5: Start NetBeans and Open a Project

Overview

In this practice, you launch NetBeans and open a NetBeans project to ensure NetBeans is functioning properly.

Assumptions

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

Tasks

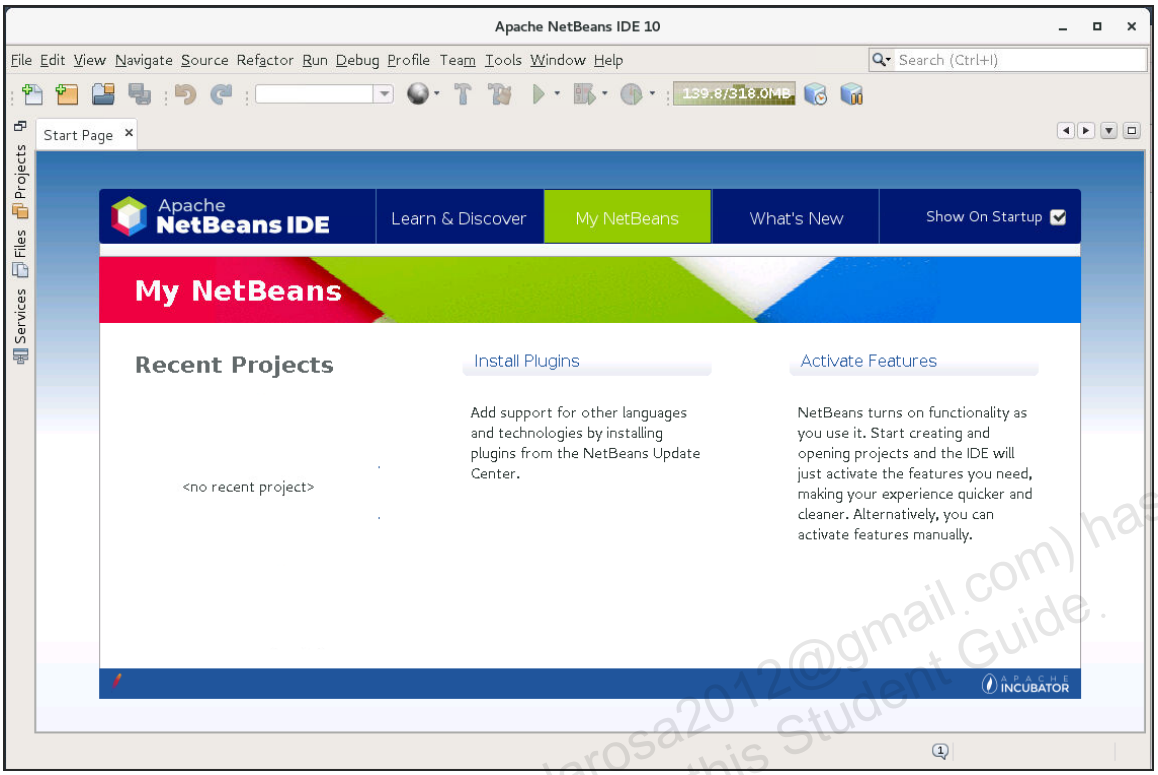
1. Open **NetBeans 10** from the desktop.

   Note: This is a release candidate 3 build of NetBeans. It's a precursor leading up the official release of NetBeans 10. We'll use this build for most of the practices.

2. You'll notice NetBeans begin to load:

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

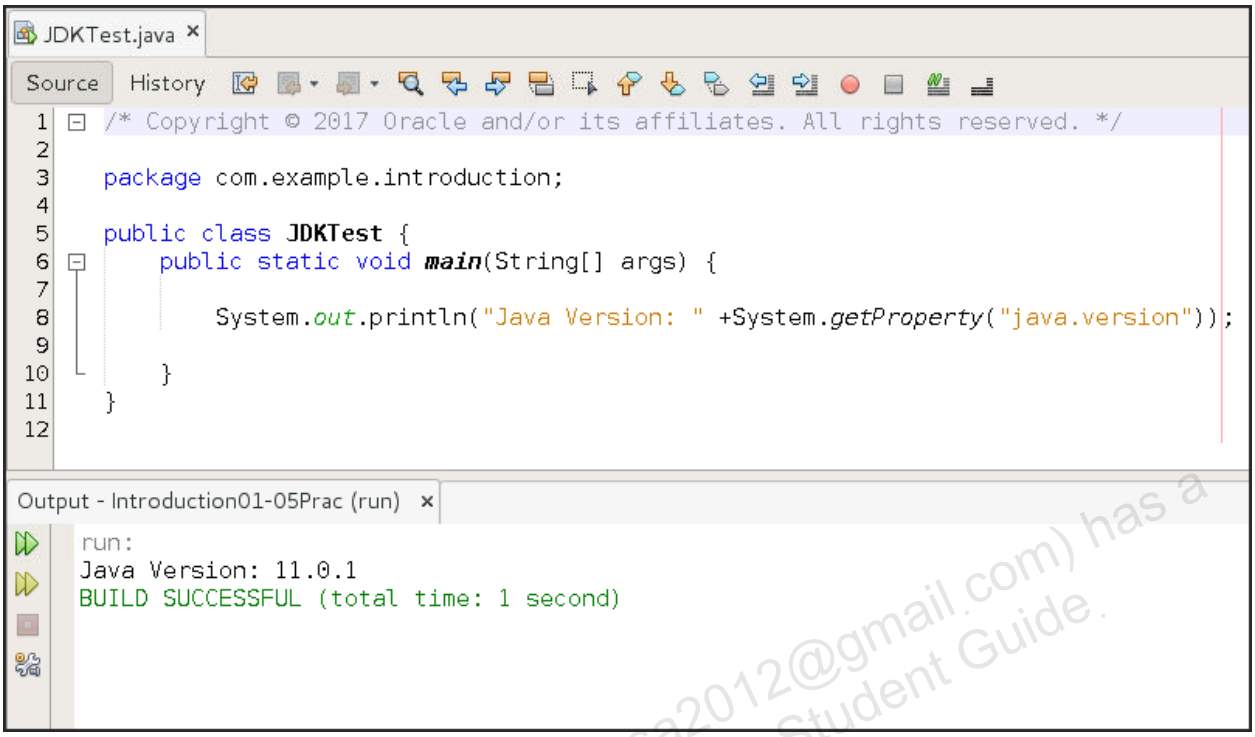
3. After it launches, NetBeans should look like this:



4. Open a sample NetBeans project by selecting **File** > **Open Project**.
5. Navigate to the **`labs/01_Intro/practices`** directory.
6. Select **`Introduction01-05Prac`** and then click **Open Project**. You'll notice the Projects window now appears in NetBeans.
7. Expand the project and double-click **`JDKTest.java`** to view the code.
8. Run the project. Right-click the project from the Projects window and select **Run**. You may alternatively press the Run button ( ).

   Note: This program prints the current version of Java to the output window. It should indicate you're using Java SE 11.

9. NetBeans should look like this:



10. Close NetBeans.

Practices for Lesson 2: Java: OOP Concepts and Review

Practices for Lesson 2: Overview

Overview

In these practices, you will use the abstract, final, and static Java keywords. You will also learn to refactor existing application using Java enums.

Practice 2-1: Overriding and Overloading Methods

Overview

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

Tasks

1. Open the `Employee02-01Prac` project in the `practices` directory.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/02-Review/practices/practice1`.
   c. Select `Employee02-01Prac` and click Open Project.
2. Edit the `Employee` class: to override the `toString()` method from the `Object` class. `Object`'s `toString` method returns a `String`.
   a. Delete the instance method `printEmployee()` from the `Employee` class.

```
public  void printEmployee() {

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

   b. Add the `toString` method to the `Employee` class with the following signature:
   `public String toString() {`
   c. Add a `return` statement that returns a string that includes the employee information: ID, name, Social Security number, and a formatted salary like this:

```
return "Employee ID:     " + getEmpId() + "\n" +
       "Employee Name:   " + getName() + "\n" +
       "Employee SSN:    " + getSsn() + "\n" +
       "Employee Salary: " +
NumberFormat.getCurrencyInstance().format(getSalary());
```

   d. Save the `Employee` class.

Override the `toString` method in the `Manager` class to include the `deptName` field value.

e. Open the `Manager` class.

f. Add a `toString` method with the same signature as the `Employee` `toString` method:

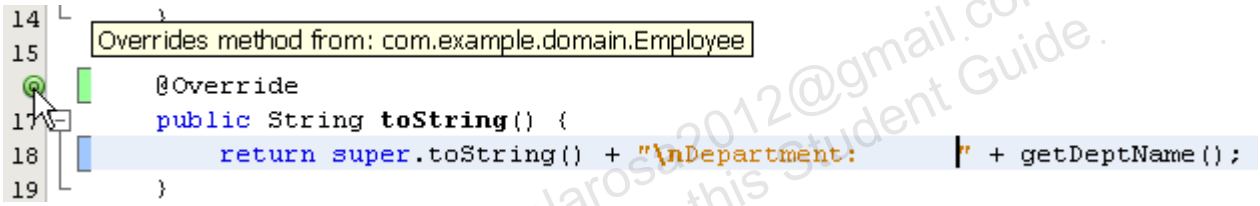
```
public String toString() {
```

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

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

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

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



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

h. Save the `Manager` class.

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

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

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

a. Create the new package and class in one step by right-clicking Source Package, and then selecting New > Java Class.

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

1) Enter the class name as `EmployeeStockPlan`.

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

3) Click Finish.

c. Add fields to the `EmployeeStockPlan` class to define the stock levels, like this:

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

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

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

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

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

f. Resolve any missing import statements.

g. Save the `EmployeeStockPlan` class.

5. Modify the `EmployeeTest` class:

a. Add a static `printEmployee` method.

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

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

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

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

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

1) Create another `printEmployee` method that takes an instance of the `EmployeeStockPlan` class:

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

2) This method first calls the original `printEmployee` method:

```
a. printEmployee(emp);
```

3) Add a print statement to print out the number of stock options that the employee is entitled to:

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

c. Resolve any missing import statements.

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

```
EmployeeStockPlan esp = new EmployeeStockPlan();
printEmployee(eng, esp);
```

e. Modify the remaining `printEmployee` invocations.

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

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

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

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

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

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

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

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

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

Practice 2-2: Using Java Enumerations

Overview

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

Assumptions

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

Summary

You have been given a project that implements the logic for a bank. By creating a new Java enum you will modify the application to hold various branch locations of the bank. By using enum to store the branch details, in the future it is easy to add more branch locations to the bank, it is easy to validate branch information.

Tasks

1. Open the `EnumBanking02-02Prac` project as the main project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/02-Review/practices/practice2`.
   c. Select `EnumBanking02-02Prac`.
   d. Click Open Project.
2. Expand the project directories.
3. Run the project. You should see a report of all customers and their accounts.
4. Create a new Java enum, `Branch` in the `com.example` package, by performing the following steps:
   a. In NetBeans, right-click on the project, select New > Other.
   b. Select **Java** from Categories column
   c. Select **Java Enum** from File Types column
   d. Click **Next**.
5. In the Name and Location dialog box, enter the following details:
   a. Class: `Branch`
   b. Package: `com.example`
   c. Click Finish.
6. Modify the enum, `Branch.java`. The `Branch enum` stores the location at which the customer banks at. In addition, information about the types of services offered by the bank are also stored.
   a. Create `Branch` instances, `LA, BOSTON, BANGALORE, MUMBAI` that call the `Branch` constructor with values "Basic", "Loan", "Full", and "Full", respectively.

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

```
public enum Branch {

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

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

        public String getServiceLevel(){
          return serviceLevel;
        }


}
```

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

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

   b. Declare a variable of type Branch.

```
      private Branch branch;
```

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

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

    }
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Customer: Smith, Will
Branch: LA, Basic
```

```
    Checking Account: current balance is 500.0

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

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

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

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

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

Practices for Lesson 3: Exceptions and Assertions

Practices for Lesson 3: Overview

Practices Overview

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

Practice 3-1: Extending `Exception` and `Throwing Exception`

Overview

In this practice, you will take an existing application and refactor the code to make use of a custom exception class and throwing exception using `throw` and `throws`.

Assumptions

You have reviewed the exception handling section of this lesson.

Summary

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

Tasks

1. Open the `CustomExceptions03-01Prac` project as the main project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/03-Exceptions/practices/practice1`
   c. Select `CustomExceptions03-01Prac` and Click Open Project.
2. Expand the project directories.
3. Create a `InvalidOperationException` class in the `com.example` package.
4. Complete the `InvalidOperationException` class. The `InvalidOperationException` class should:
   - Extend the `Exception` class.
   - Create four public constructors with parameters matching those of the four public constructors present in the `Exception` class. For each constructor, use `super()` to invoke the parent class constructor with matching parameters.

```
package com.example;

public class InvalidOperationException extends Exception {

    public InvalidOperationException() {
        super();
    }

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

    public InvalidOperationException(Throwable cause) {
        super(cause);
```

```
    }

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

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

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

6. Modify the `delete` method within the `EmployeeImpl` class to:
   - Declare that a `InvalidOperationException` may be produced during execution of this method.
   - Use an if statement to validate that an existing employee is being deleted. If one would not be, generate a `InvalidOperationException` and deliver it to the caller of the method. The `InvalidOperationException` should contain a message String indicating what went wrong and why.

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

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

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

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

8. Modify the `EmployeeTest` class to handle the `InvalidOperationException` objects that are thrown by the `EmployeeImpl`
   a. .Modify the `main` method:

Add the `throws` statement from the `main` method.

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

9. Run the project. Test all the operations by invoking the methods: `add`,`delete` and `findById`.

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

```
Exception in thread "main"
com.example.InvalidOperationException: Error deleting employee,
no such employee 7
```

Practices for Lesson 4: Java Interfaces

Practices for Lesson 4: Overview

Overview

In these practices, you will explore how Java interfaces have evolved and what effects this evolution has on your code. Practices for this lesson deal with a subset of financial products. Financial products include savings accounts, checking accounts, savings bonds, or certificates of deposits. All financial products could arguably inherit from a `FinancialProduct` super class; however, you won't need to write or extend this super class in this lesson's practice. Instead, you'll focus on writing the `Accessible` interface. The goal of implementing this interface is to enable customers to access the money within a financial product through deposits and withdrawals. A checking account, for example, would be accessible. A savings bond would not. In this practice, you'll implement and evolve the `Accessible` interface, `BasicChecking` class, and `RestrictedChecking` class.

Practice 4-1: Java SE 8 Default Methods

Overview

In this practice, you begin correcting some of the code duplication. For example, the implementations of the `verifyDeposit` and `verifyWithdraw` methods are nearly identical across the `BasicChecking` and `RestrictedChecking` classes. It would be great if this duplicated code could be written only once. As of Java SE 8, default methods provide a solution. Default methods in interfaces may contain an implementation.

Assumptions

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

Tasks

1. Open the `Interfaces04-01Prac` project as the main project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/04-Interfaces/practices/practice1`
   c. Select `Interfaces04-01Prac` and Click Open Project.
2. Navigate to the `Accessible` interface.
3. Provide an implementation for the `verifyDeposit` method.
   a. Change the `verifyDeposit` method from `abstract` to `public default`.
   b. Study the implementation of this method in the `BasicChecking` and `RestrictedChecking` classes. Copy over any duplicated code.
      1) In the case of `verifyDeposit`, the entire method is duplicated. Write this implementation once in the interface, and remove the method entirely from the `BasicChecking` and `RestrictedChecking` classes. There's no longer a need to override this method.
      2) You'll notice NetBeans complains that the `verifyPIN` method cannot be found. To fix this, create an abstract `verifyPIN` method in the `Accessible` interface.

         Note: Although the implementation for the `verifyPIN` method is also duplicated between checking accounts, you can't use default methods to resolve this. The `verifyPIN` method requires access to instance fields. But interfaces only support static fields, not instance fields. This design decision prevents potential issues around multiple inheritance of state.
4. Provide an implementation for the `verifyWithdraw` method.
   a. Change the `verifyWithdraw` method from `abstract` to `public default`.
   b. Study the implementation of this method found in the `BasicChecking` and `RestrictedChecking` classes. Copy over any duplicated logic.
      1) Everything that's found in the `BasicChecking` implementation of `verifyWithdraw` is also found in the `RestrictedChecking` version. When

you write the common parts of the implementation once in the interface, you'll remove the method entirely from the `BasicChecking` class.

2) The `RestrictedChecking` class still needs to override `verifyWithdraw`. Remove any duplicated code and replace it with a call to the method found in the interface:

```
Accessible.super.verifyWithdraw(amount, pin)
```

3) NetBeans complains that the `getBalance` method cannot be found. To fix this, create an abstract `getBalance` method in the `Accessible` interface.

Note: Just like the `verifyPIN` method, you can't use default methods to resolve code duplication with methods that access an instance's state. The `getBalance` method requires access to instance fields. But interfaces only support static fields, not instance fields. This design decision prevents potential issues around multiple inheritance of state.

5. Use the `main` method to test your changes. Do the checking account instances still behave as expected?

Practice 4-2: Java SE 9 Private Methods

Overview

In this practice, you'll continue refining your code to minimize code duplication. You'll notice the `verifyDeposit` and `verifyWithdraw` methods duplicate some of the same logic. Both methods verify the PIN number and check that the amount requested is greater than 0. This kind of code duplication can be safely minimized in Java SE 9 with the introduction of `private` methods in interfaces.

Assumptions

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

Tasks

1. Open the `Interfaces04-02Prac` project as the main project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/04-Interfaces/practices/practice2`
   c. Select `Interfaces04-02Prac` and Click Open Project.
2. Navigate to the `Accessible` interface.
3. Create a `default boolean` method `verifyTransaction`. This method should implement the logic that is duplicated between the `verifyDeposit` and `verifyWithdraw` methods. It must accept a `double amount` and `int pin` arguments.
   a. If the `pin` cannot be verified, return `false` instead of 0.
   b. If the `amount` is less than 0, return `false` instead of 0.
   c. Otherwise, return `true`.
   d. Leave this method `default` for now. You'll notice this design decision has consequences, which you'll correct later.
4. Modify the `verifyDeposit` and `verifyWithdraw` methods so that instead of duplicating code, they call the `verifyTransaction` method. If the call to `verifyTransaction` returns `false`, these either method should return 0.
5. Use the `main` method to test your changes. Do the checking account instances still behave as expected?
6. Try calling `verifyTransactions` from the `main` method.

   Note: It's possible to make this call. The information returned from the `verifyTransactions` method doesn't have much meaning when it's separated from the rest of the logic found in the `verifyDeposit` and `verifyWithdraw` methods. This utilization of default methods could be problematic if it leads to the creation of methods that you don't want callable. Preventing this issue is the motivation behind `private` methods in Java SE 9 interfaces.

7. Change the `verifyTransactions` method from `default` to `private`.
8. Observe that it's no longer possible to call this `private` method from the `main` method. It's also not possible to override a `private` method or call it from the classes which implement the interface! For this reason, `private` interface methods require you to think carefully about how you design interfaces.

Practices for Lesson 5: Generics and Collections

Practices for Lesson 5: Overview

Practices Overview

In these practices, use generics and collections to practice the concepts covered in the lecture.

Practice 5-1: Counting Part Numbers by Using HashMaps

Overview

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

Assumptions

You have reviewed the collections section of this lesson.

Summary

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

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

Once complete, your report should look like this:

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

Tasks

1. In NetBeans, open the `GenericsHashMap05-01Prac` project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/05-Generics_Collections/practices/practice1`
   c. Select `GenericsHashMap05-01Prac` and click Open Project.
2. Expand the project directories.

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

   a. Add two private map fields- `productCountMap` and `productNames`. The first map counts part numbers. The order of the keys does not matter. The second map stores the mapping of product description to part number. The keys should be sorted alphabetically by description for the second map.

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

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

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

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

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

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

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

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

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

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

```
run:
=== Product Report ===
Name: Black Polo Shirt          Count: 6
Name: Blue Polo Shirt           Count: 7
Name: Duke Mug                  Count: 3
Name: Red Ball Cap              Count: 5
BUILD SUCCESSFUL (total time: 0 seconds)
```

Practice 5-2: Using Convenience Method `of`

Overview

In this and the next practice, you will explore the use of convenience methods `of` and `ofEntry` as they apply to the `Set`, `List`, and `Map` interfaces. You'll see the convenience these methods provide in setting up collections and explore their immutability.

The following are examples of calling convenience methods on collections:

Set:

```
List<String> testList = List.of("A", "B", "C", "D", "E");
```

List:

```
Set<String> testSet = Set.of("A", "B", "C", "D", "E");
```

Map:

```
Map<String, Integer> testMap = Map.of(
     "A", 1,
     "B", 2,
     "C", 3,
     "D", 4,
     "E", 5
);
```

```
Map<String, Integer> testMap2 = Map.ofEntries(
     entry("A", 1),
     entry("B", 2),
     entry("C", 3),
     entry("D", 4),
     entry("E", 5)
);
```

Assumptions

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

Tasks

This practice creates a list of `Integer` objects for simplicity. But the same work can be done with collections of any object type.

1. In NetBeans, open the project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/05-Generics_Collections/practices/practice2`

c. Select `ConvenientCollection05-02Prac` and click Open Project.

d. Expand the project directories.

e. Open `TestClass.java` in the editor and make the following changes:

2. Import the `java.util.List` package.

3. Create a `List` called `testList`.

   a. Use generics to specify that this is a collection of `Integer` objects.

   b. Use the `of` method from the `List` interface to set the values of `testList`.

   c. Set `testList`'s elements to contain the first six powers of two: 1, 2, 4, 8, 16, 32.

4. Explore what immutability means in the context of collections. Just how immutable is this collection?

   a. Try calling the `add` method on `testList` to add the next power of two, 64, to the collection.

   b. Try calling the `set` method on `testList` to replacing an element in the collection with 0.

   c. Try replacing the `List` altogether. Set `testList` equal to a new `List` that contains a single integer: 0. Create this new List by calling the `of` method of the `List` interface.

   Note: Although you can outright replace the collection by pointing your reference variable elsewhere, any other attempt to modify the collection results in an error message. Before you create a collection using these convenience methods, it may be beneficial to first consider how these lists will be treated by your program and if the data may need to be modified later.

Practice 5-3: Using Convenience Method `ofEntries`

Overview

In this practice, you will see how the `ofEntry` convenience method can be used to create a `Map`. This method is very similar to the `of` method. They're both static factory methods which produce immutable collections. However, the `ofEntry` method is designed for creating Maps with indeterminate number of elements and requires each `Map` entry to be boxed.

This practice examines fake holidays and their associated dates.

Assumptions

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

Tasks

1. In NetBeans, open the project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/05-Generics_Collections/practices/practice3`
   c. Select `ConvenientCollection05-03Prac` and click Open Project.
   d. Expand the project directories.
   e. Open `TestClass.java` in the editor and make the following changes:
2. Create a `Map` called `testMap`.
   a. Use generics to specify that this `Map` requires a `String` key and `MonthDay` value.
   b. Use the `ofEntry` method from the `Map` interface to set the values of `testMap`.
   c. Set the `Map`'s keys to five fake holidays. Represent each key as a `String`:
      1) Bologna Day
      2) Opposite Day
      3) Panic Day
      4) Raymond Day
      5) Knight Day
   d. Set this map's values to the day the holiday occurs. Represent these as a `MonthDay`.
      1) January 16
      2) February 13
      3) March 9
      4) October 20
      5) December 1

      Hint: An example of creating a `MonthDay` is:

```
MonthDay.of(Month.APRIL, 1)
```

3. Did NetBeans complain? It may be because you need to import the following packages before creating the `Map`:

```
import java.time.MonthDay;
import java.time.Month;
import java.util.Map;
import static java.util.Map.entry;
```

4. With the imports taken care of, you should be able to create the `Map` now.
5. Add the following code to print out the contents of the `Map`. Don't worry that you haven't really looked in detail at using `forEach` in this way. You'll learn all about `forEach` and lambda later in the course.

```
testMap.forEach((key,value) -> System.out.println(key +": "
+value +", "));
```

6. Run `TestClass.java` and verify the output.

```
run:
0,
Bologna Day: --01-16,
Opposite Day: --02-13,
Raymond Day: --10-20,
Knight Day: --12-01,
Panic Day: --03-09,
```

Practices for Lesson 6: Functional Interfaces and Lambda Expressions

Practices for Lesson 6: Overview

Practices Overview

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

Practice 6-1: Refactor Code to Use Lambda Expressions

Overview

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

Assumptions

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

Tasks

1. In NetBeans, open the `EmployeeSearch06-01Prac` project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/06-Lambda/practices/practice1`
   c. Select `EmployeeSearch06-01Prac` and click Open Project.
2. Open the `Employee.java` file and become familiar with the code included in the file.
3. Open the `RoboMailOldStyle.java` file and review the code in the file.
4. Open the `RoboTest01.java` file and review the code there.
5. Run the project and examine the output.
6. Update the `RoboMailTest01.java` file and replace the anonymous inner classes with lambda expressions.
7. Your program should perform the following tasks to the following groups.
   - Email all HR employees: `e.getDept().equals("HR")`
   - Text all sales employees: `e.getDept().equals("Sales")`
   - Text all sales executives: `e.getRole().equals(Role.EXECUTIVE) && e.getDept().equals("Sales")`
   - Email all sales older than 50: `e.getAge() >= 50 && e.getDept().equals("Sales")`

Your output should look similar to the following:

```
==== RoboMail 01

=== Members of HR ===
Emailing: John Doe age 28 at john.doe@example.com
Emailing: Phil Smith age 55 at phil.smith@examp;e.com

=== All Sales ===
Texting: Jane Doe age 25 at 202-123-4678
Texting: John Adams age 52 at 112-111-1111
Texting: Betty Jones age 65 at 211-333-1234

=== All Sales Execs
Texting: Betty Jones age 65 at 211-333-1234
```

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

Practice 6-2: Refactor Code to Reuse Lambda Expressions

Overview

In this practice, continue with the previous project or use the project provided in the following.

Assumptions

You have completed the lecture and completed the previous practice.

Tasks

1. In NetBeans, open the `EmployeeSearch06-02Prac` project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/06-Lambda/practices/practice2`
   c. Select `EmployeeSearch06-02Prac` and click Open Project.
2. Open the `RoboMailTest01.java` file and review the code there.
3. Update the `RoboMailTest01.java` file to store lambda expressions in variables.
   a. Create a variable `salesExecutives` that stores a lambda expression that selects all sales employees: `e.getDept().equals("Sales")`
   b. Create a variable `salesEmployeesOver50` that stores a lambda expression that selects sales employees over the age of 50: `e.getAge() >= 50 && e.getDept().equals("Sales")`
4. Your program should perform the following tasks to the following groups.
   - Email all sales executives
   - Text all sales executives
   - Email all sales employees older than 50
   - Text all sales employees older than 50

Your output should look similar to the following:

```
==== RoboMail 01

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

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

Practices for Lesson 7: Collections Streams, and Filters

Practices for Lesson 7: Overview

Practice Overview

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

The RoboCall App

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

- **Drivers:** Persons over the age of 16
- **Draftees:** Male persons between the ages of 18 and 25
- **Pilots (specifically commercial pilots):** Persons between the ages of 23 and 65

Person

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

First, private fields for each Person are as follows:

Person.java

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

These will be the fields that our application can search.

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

Person.java

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

```
178             .address("44 4th St")
179             .city("Smallville")
180             .state("KS")
181             .code("12333")
182             .build()
183       );
184
```

forEach

All collections have a `forEach` method.

RoboCallTest06.java

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

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

Stream and Filter

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

RoboCallTest07.java

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

The `stream` method creates a pipeline of immutable `Person` elements and access to methods that can perform actions on those elements. The `filter` method takes a lambda expression as a parameter and filters on the logical expression provided. This indicates that a `Predicate` is the target type of the filter. The elements that meet the filter criteria are passed to the `forEach` method, which does a `roboCall` on matching elements.

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

RoboCallTest08.java

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

Method References

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

A03aMethodReference.java

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

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

Chaining and Pipelines

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

A04IterationTest.java

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

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

Practice 7-1: Update RoboCall to Use Streams

Overview

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

Assumptions

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

Tasks

1. Open the `EmployeeSearch07-01Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/07-CollectionsStreamsFilters /practices/practice1`.
   - Select `EmployeeSearch07-01Prac` and click Open Project.
2. Open the `RoboMail01.java` file and remove the `mail` and `text` methods. They are no longer needed since a `stream` will be used to filter the employees and a `forEach` will call the required communication task.
3. Open the `RoboMailTest01.java` file and review the code there.
4. Update `RoboMailTest01.java` to use `stream`, `filter`, and `forEach` to perform the mailing and texting tasks of the previous program.
5. Your program should continue to perform the following tasks to the following groups.
   - Email all sales executives using `stream`, `filter`, and `forEach`.
   - Text all sales executives using `stream`, `filter`, and `forEach`.
   - Email all sales employees older than 50 using `stream`, `filter`, and `forEach`.
   - Text all sales employees older than 50 using `stream`, `filter`, and `forEach`.
6. To mail or text a group in the `forEach` method, use a lambda expression for each task.
   a. Mail example: `p -> robo.roboMail(p)`
   b. Text example: `p -> robo.roboText(p)`

Your output should look similar to the following:

```
==== RoboMail 01

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

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

Practice 7-2: Mail Sales Executives Using Method Chaining

Overview

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

Assumptions

You have completed the lecture and completed the previous practice.

Tasks

1. Open the `EmployeeSearch07-02Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/07-CollectionsStreamsFilters /practices/practice2`.
   - Select `EmployeeSearch07-02Prac` and click Open Project.
2. Open the `RoboMailTest01.java` file and review the code there.
3. Update the `RoboMailTest01.java` file to mail all sales executives. Use two filter methods to select the recipients of the mail.

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

```
==== RoboMail 01

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

Practice 7-3: Mail Sales Employees over 50 Using Method Chaining

Overview

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

Assumptions

You have completed the lecture and completed the previous practice.

Tasks

1. Open the `EmployeeSearch07-03Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/07-CollectionsStreamsFilters /practices/practice3`.
   - Select `EmployeeSearch07-03Prac` and click Open Project.
2. Open the `RoboMailTest01.java` file and review the code there.
3. Update the `RoboMailTest01.java` file to mail all sales employees over 50. Use two filter methods to select the recipients of the mail.

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

```
==== RoboMail 01

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

Practice 7-4: Mail Male Engineering Employees Under 65 Using Method Chaining

Overview

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

Assumptions

You have completed the lecture and completed the previous practice.

Tasks

1. Open the `EmployeeSearch07-04Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/07-CollectionsStreamsFilters /practices/practice4`.
   - Select `EmployeeSearch07-04Prac` and click Open Project.
2. Open the `RoboMailTest01.java` file and review the code there.
3. Update the `RoboMailTest01.java` file to mail all male engineering employees under 65. Use three filter methods to select the recipients of the mail.

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

```
==== RoboMail 01

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

Practices for Lesson 8: Lambda Built-in Functional Interfaces

Practices for Lesson 8: Overview

Practice Overview

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

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

Predicate

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

A01Predicate.java

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

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

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

Consumer

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

A02Consumer.java

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

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

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

Function

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

A03Function.java

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

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

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

Supplier

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

A04Supplier.java

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

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

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

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

Primitive Types - ToDoubleFunction and AutoBoxing

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

A05PrimFunction.java

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

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

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

A05PrimFunction.java

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

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

Primitive Types -– DoubleFunction

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

A06DoubleFunction.java

```
 5 public class A06DoubleFunction {
 6
 7   public static void main(String[] args) {
 8
```

```
 9     A06DoubleFunction test = new A06DoubleFunction();
10
11     DoubleFunction<String> calc =
12           t -> String.valueOf(t * 3);
13
14     String result = calc.apply(20);
15     System.out.println("New value is: " + result);
16   }
17 }
```

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

Binary Interfaces – BiPredicate

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

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

A07Binary.java

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

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

UnaryOperator

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

A08Unary.java

```
10 public class A08Unary {
11
12   public static void main(String[] args){
13
14     List<SalesTxn> tList = SalesTxn.createTxnList();
15     SalesTxn first = tList.get(0);
16
17     UnaryOperator<String> unaryStr =
18         s -> s.toUpperCase();
19
20       System.out.println("== Upper Buyer");
21       System.out.println(
22           unaryStr.apply(first.getBuyerName()));
23   }
24 }
```

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

API Docs

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

Practice 8-1: Creating Consumer Lambda Expression

Overview

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

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

Assumptions

You have completed the lecture portion of the course.

Tasks

1. Open the `EmployeeSearch08-01Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/08-LambdaBuiltIns/practices/practice1`.
   - Select `EmployeeSearch08-01Prac` and click Open Project.
2. Open the `Employee.java` file and become familiar with the code included in the file.
3. Open the `ConsumerTest.java` file and make the following updates.
4. Write a `Consumer` lambda expression to print data about the first employee in the list.
   a. The data printed should be the following: `"Name: " + e.getSurName() + " Role: " + e.getRole() + "  Salary: " + e.getSalary()`
5. Write a statement to execute the lambda expression on the `first` variable.
6. Your output should look similar to the following:

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

Practice 8-2: Creating a Function Lambda Expression

Overview

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

Assumptions

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

Tasks

1. Open the `EmployeeSearch08-02Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/08-LambdaBuiltIns/practices/practice2`.
   - Select `EmployeeSearch08-02Prac` and click Open Project.
2. Open the `Bonus.java` file and review the code included in the file.
3. Open the `FunctionTest.java` file and make the following updates.
4. Write a `ToDoubleFunction` lambda expression to calculate the bonus for the first employee in the list.
   a. The bonus can be calculated as follows: `e.getSalary() * Bonus.byRole(e.getRole())`
5. Write a statement to execute the lambda expression on the `first` variable.
6. Your output should look similar to the following:

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

Practice 8-3: Creating a Supplier Lambda Expression

Overview

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

Assumptions

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

Tasks

1. Open the `EmployeeSearch08-03Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/08-LambdaBuiltIns/practices/practice3`.
   - Select `EmployeeSearch08-03Prac` and click Open Project.
2. Open the `SupplierTest.java` file and make the following updates.
3. Write a `Supplier` lambda expression to add a new employee to the list. The employee data is as follows:

   Given name: Jill
   SurName: Doe
   Age: 26
   Gender: Gender.FEMALE
   Role: Role.STAFF
   Dept: Sales
   StartDate: LocalDate.of(2012, 7, 14)
   Salary: 45000
   Email: jill.doe@example.com
   PhoneNumber: 202-123-4678
   Address: 33 3rd St
   City: Smallville
   State: KS
   Code: 12333

   Hint: Her data is almost exactly the same as her sister Jane and can be found in the `Employee.java` file.
4. Write a statement to add the new employee to the employee list.

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

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

Practice 8-4: Creating a BiPredicate Lambda Expression

Overview

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

Assumptions

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

Tasks

1. Open the `EmployeeSearch08-04Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/08-LambdaBuiltIns/practices/practice4`.
   - Select `EmployeeSearch08-04Prac` and click Open Project.
2. Open the `BiPredicateTest.java` file and make the following updates.
3. Write a `BiPredicate` lambda expression to compare a field in the employee class to a string.
   a. The `searchState` variable should be compared to the state value in the employee element.
4. Write an expression to perform the logical test in the `for` loop.
5. Your output should look similar to the following:

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

Practices for Lesson 9: More Lambda Operations

Practices for Lesson 9: Overview

Practice Overview

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

Employee List

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

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

Map

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

A01MapTest.java

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

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

Output

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

Peek

The `peek` method of the `Stream` class allows you to perform an operation on an element in the stream. The elements are returned to the stream and are available to the next stream in the

pipeline. The `peek` method can be used to read or change data in the stream. Any changes will be made to the underlying collection.

A02MapPeekTest.java

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

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

Output

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

Find First

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

A03FindFirst.java

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

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

Output

```
== First CO Bonus ==

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

Find First Lazy

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

A04FindFirstLazy.java

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

```
33              .filter(e -> e.getState().equals("CO"))
34              .peek(e -> System.out.println("CO Executives"))
35              .findFirst();
36
37          if (result.isPresent()){
38              result.get().printSummary();
39          }
40      }
41 }
```

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

Output

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

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

anyMatch

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

A05AnyMatch.java

```
10 public class A05AnyMatch {
```

```
11
12      public static void main(String[] args) {
13
14        List<Employee> eList = Employee.createShortList();
15
16        System.out.println("\n== First CO Bonus ==");
17        Optional<Employee> result;
18
19        if (eList.stream().anyMatch(
20            e -> e.getState().equals("CO"))){
21
22          result = eList.stream()
23            .peek(e -> System.out.println("Stream"))
24            .filter(e -> e.getRole().equals(Role.EXECUTIVE))
25            .filter(e -> e.getState().equals("CO"))
26            .findFirst();
27
28          if (result.isPresent()){result.get().printSummary();}
29        }
```

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

Count

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

A06StreamData.java

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

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

Output

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

Max

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

A06StreamData.java

```
23      System.out.println("Exec count: " + execCount);
24
25      System.out.println("\n== Highest Paid Exec ==");
26      Optional highestExec =
27        eList.stream()
28          .filter(e -> e.getRole().equals(Role.EXECUTIVE))
29          .max(Employee::sortBySalary);
30
31      if (highestExec.isPresent()){
32        Employee temp = (Employee) highestExec.get();
33        System.out.printf(
34            "Name: " + temp.getGivenName() + " "
35            + temp.getSurName() + "   Salary: $%,6.2f %n ",
36            temp.getSalary());
37      }
```

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

Output

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

Min

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

A06StreamData.java

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

In this example, a different `Comparator` is used. The `comparingDouble` static method is called to make the comparison. Notice that the example uses a lambda expression to specify

the comparison field. If you look at the code closely, a method reference could be substituted instead: `Employee::getSalary`. More discussion on this subject follows in the `Comparator` section.

Output

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

Sum

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

A07CalcSum.java

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

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

Output

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

Average

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

A08CalcAvg.java

```
28      System.out.println("\n== Average CO Bonus Details ==");
29
30      result = eList.stream()
31        .filter(e -> e.getRole().equals(Role.EXECUTIVE))
32        .filter(e -> e.getState().equals("CO"))
33        .peek(e -> System.out.print("Name: " + e.getGivenName()
```

```
34            + " " + e.getSurName() + " "))
35        .mapToDouble(e -> e.getSalary() * Bonus.byRole(e.getRole()))
36        .peek(d -> System.out.printf("Bonus paid: $%,6.2f %n", d))
37        .average();
38
39        if (result.isPresent()){
40          System.out.printf("Average Bonuses paid: $%,6.2f %n",
41            result.getAsDouble());
42        }
43    }
```

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

Output

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

Sorted

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

A09SortBonus.java

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

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

Output

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

Sorted with Comparator

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

A10SortComparator.java

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

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

Output

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

Reversed

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

A10SortComparator.java

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

Output

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

Two Level Sort

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

A10SortComparator.java

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

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

Output

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

Collect

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

A11Collect.java

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

```
29          nList.stream()
30              .forEach(Employee::printSummary);
31
32      }
33
34  }
```

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

Output

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

Collectors and Math

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

A12CollectMath.java

```
12 public class A12CollectMath {
13
14     public static void main(String[] args) {
15
16         List<Employee> eList = Employee.createShortList();
17
18         // Collect CO Executives
19         double avgSalary = eList.stream()
20             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
21             .filter(e -> e.getState().equals("CO"))
22             .collect(
23                 Collectors.averagingDouble(Employee::getSalary));
24
25         System.out.println("\n== CO Exec Avg Salary ==");
26         System.out.printf("Average: $%,9.2f %n", avgSalary);
27
28     }
29
30 }
```

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

Output

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

Collectors and Joining

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

A13CollectJoin.java

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

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

Output

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

Collectors and Grouping

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

A14CollectGrouping.java

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

```
26              System.out.println("\nDept: " + k);
27              v.forEach(Employee::printSummary);
28          });
29
30      }
31
32  }
```

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

Output

```
== Employees by Dept ==

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

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

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

Collectors, Grouping, and Counting

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

A15CollectCount.java

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

```
27          gMap.forEach((k,v) ->
28              System.out.println("Dept: " + k + " Count: " + v)
29          );
30
31      }
32
33  }
```

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

Output

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

Collectors and Partitioning

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

A16CollectPartition.java

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

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

Output

```
== Employees by Dept ==

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

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

Practice 9-1: Using Map and Peek

Overview

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

Assumptions

You have completed the lecture portion of this course.

Tasks

1. Open the `SalesTxn09-01Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/09-LambdaOperations /practices/practice1`.
   - Select `SalesTxn09-01Prac` and click Open Project.
2. Review the code for the `SalesTxn` class. Note that enumerations exist for `BuyerClass`, `State`, and `TaxRate`.
3. Modify the `MapTest` class to create a sales tax report.
   a. Filter the transactions for the following.
      – Transactions from the state of CA: `t.getState().equals(State.CA)`
      – Transactions for the Widget Pro product: `t.getProduct().equals("Widget Pro")`
   b. Use the `map` method to calculate the sales tax. The calculation is as follows: `t.getTransactionTotal() * TaxRate.byState(t.getState())`
   c. Print a report similar to the following:

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

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

   `System.out.printf("Txn tax: $%,9.2f%n", amt`
4. Copy the main method from the `MapTest` class to the `PeekTest` class.
5. Update your code to print more detailed information about the matching transaction using the `peek` method. A `Consumer` is provided for you that adds the following:
   - Transaction ID
   - Buyer
   - Total Transaction amount
   - Sales tax amount

6. The output should look similar to the following:

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

Practice 9-2: FindFirst and Lazy Operations

Overview

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

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

Assumptions

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

Tasks

1. Open the `SalesTxn09-02Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/09-LambdaOperations /practices/practice2`.
   - Select `SalesTxn09-02Prac` and click Open Project.
2. Edit the `LazyTest` class to perform the steps in this practice.
3. Using `stream` and lambda expressions print out a list of transactions that meet the following criteria.
   a. Create a filter to select all "Widget Pro" sales.
   b. Create a filter to select transactions in the state of Colorado (CO).
   c. Iterate through the matching transactions and print a report similar to the following using `quantReport` in the `forEach`.

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

4. Modify the search in the previous step. This time use the `peek` method to display each step in the process. Put a `peek` method call in the following places.
   a. Add a `peek` method after the `stream()` method that uses the `streamStart` as its parameter.
   b. Add a `peek` method after the `filter` for state that uses `stateSearch` as its parameter.
   c. Add a `peek` method after the `filter` for product that uses `productSearch` as its parameter.
   d. Print the final result using `forEach` as in the previous step.

e. The output should look similar to the following.

```
=== Widget Pro Quantity in CO ===
Stream start: Jane Doe ID: 11
Stream start: Jane Doe ID: 12
Stream start: Jane Doe ID: 13
Stream start: John Smith ID: 14
Stream start: Betty Jones ID: 15
State Search: Betty Jones St: CO
Product Search
Seller: Betty Jones-- Buyer: Radio Hut -- Quantity:    20,000
Stream start: Betty Jones ID: 16
State Search: Betty Jones St: CO
Stream start: Dave Smith ID: 17
State Search: Dave Smith St: CO
Product Search
Seller: Dave Smith-- Buyer: PriceCo -- Quantity:     6,000
Stream start: Dave Smith ID: 18
State Search: Dave Smith St: CO
Stream start: Betty Jones ID: 19
State Search: Betty Jones St: CO
Product Search
Seller: Betty Jones-- Buyer: Best Deals -- Quantity:    20,000
Stream start: John Adams ID: 20
Stream start: John Adams ID: 21
Stream start: Samuel Adams ID: 22
Stream start: Samuel Adams ID: 23
```

5. Copy the code from the previous step so you can modify it, and comment out the previous steps so it doesn't print out anything.
6. Replace the `forEach` with a `findFirst` method.
7. Add the following code to the search (remember you'll need to import `Optional`):
   a. Use an `Optional<SalesTxn>` named `ft` to store the result.
   b. Write an `if` statement to check to see if `ft.isPresent()`.
   c. If a value is returned, call the `accept` method of `quantReport` to display the result.
   d. Your output should look similar to the following:

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

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

Practice 9-3: Analyzing Transactions with Stream Methods

Overview

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

Assumptions

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

Tasks

1. Open the `SalesTxn09-03Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/09-LambdaOperations /practices/practice3`.
   - Select `SalesTxn09-03Prac` and click Open Project.
2. Edit the `RadioHutTest` class to perform the steps in this practice.
3. Using `stream` and lambda expressions print out all the transactions involving Radio Hut.
   a. Use a `filter` to select all "Radio Hut" transactions.
   b. Use the `radioReport` variable to print the matching transactions.
   c. Your output should look similar to the following:

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

4. Use `stream`, `filter`, and lambda expressions to calculate and print out the total number of transactions involving Radio Hut. (**Hint:** Use the `count` method.)
5. Use `stream` and lambda expressions to calculate and print out the largest transaction based on the total transaction amount involving Radio Hut. Use the `max` function with a `Comparator`. Here's an example (remember you'll need to import `Optional` and `Comparator`):

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

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

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

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

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

Practice 9-4: Performing Calculations with Primitive Streams

Overview

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

Assumptions

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

Tasks

1. Open the `SalesTxn09-04Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/09-LambdaOperations /practices/practice4`.
   - Select `SalesTxn09-04Prac` and click Open Project.
2. Edit the `CalcTest` class to perform the steps in this practice.
3. Calculate the total sales for "Radio Hut", "PriceCo", and "Best Deals" and print the results.
   - For example, filter Radio Hut with a lambda like this:
     ```
     t -> t.getBuyerName().equals("Radio Hut")
     ```
   - For example, get the transaction total with:
     ```
     .mapToDouble( t -> t.getTransactionTotal())
     ```
4. Calculate the average number of units sold for the "Widget" and "Widget Pro" products and print the results.
   - For example, the Widget Pro code looks like the following:

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

   Hint: Be mindful of the method return types. Use to the API doc to ensure you are using the correct methods and classes to create and store results.
5. The output from your test class should be similar to the following:

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

Practice 9-5: Sorting Transactions with Comparator

Overview

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

Assumptions

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

Tasks

1. Open the `SalesTxn09-05Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/09-LambdaOperations /practices/practice5`.
   - Select `SalesTxn09-05Prac` and click Open Project.
2. Edit the `SortTest` class to perform the steps in this practice.
3. Use streams and lambda expressions to print out all the PriceCo transactions by transaction total in ascending order.
   - The sorted method should look something like this:

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

   - Use the `transReport` variable to print the results.
4. Use the same data from the previous step to print out the PriceCo transactions in descending order.
5. Print out all the transactions sorted using the following sort keys.
   - Buyer name
   - Sales person
   - Transaction total
6. When complete, the output should look similar to the following:

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

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

=== Triple Sort Transactions ===
Id: 11 Seller: Jane Doe Buyer: Acme Electronics Amt: $60,000.00
```

```
Id: 12 Seller: Jane Doe Buyer: Acme Electronics Amt: $400,000.00
Id: 16 Seller: Betty Jones Buyer: Best Deals Amt: $500,000.00
Id: 19 Seller: Betty Jones Buyer: Best Deals Amt: $800,000.00
Id: 14 Seller: John Smith Buyer: Great Deals Amt: $100,000.00
Id: 22 Seller: Samuel Adams Buyer: Mom and Pops Amt: $60,000.00
Id: 17 Seller: Dave Smith Buyer: PriceCo Amt: $240,000.00
Id: 18 Seller: Dave Smith Buyer: PriceCo Amt: $300,000.00
Id: 20 Seller: John Adams Buyer: PriceCo Amt: $280,000.00
Id: 21 Seller: John Adams Buyer: PriceCo Amt: $640,000.00
Id: 15 Seller: Betty Jones Buyer: Radio Hut Amt: $800,000.00
Id: 13 Seller: Jane Doe Buyer: Radio Hut Amt: $2,000,000.00
Id: 23 Seller: Samuel Adams Buyer: Radio Hut Amt: $1,040,000.00
```

Practice 9-6: Collecting Results with Streams

Overview

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

Assumptions

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

Tasks

1. Open the `SalesTxn09-06Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/09-LambdaOperations /practices/practice6`.
   - Select `SalesTxn09-06Prac` and click Open Project.
2. Edit the `CollectTest` class to perform the steps in this practice.
3. Filter the transaction list to only include transactions greater than $300,000 sorted in ascending order.
4. Store the results in a new list using the `collect` method. For example:

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

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

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

Practice 9-7: Joining Data with Streams

Overview

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

Assumptions

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

Tasks

1. Open the `SalesTxn09-07Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/09-LambdaOperations /practices/practice7`.
   - Select `SalesTxn09-07Prac` and click Open Project.
2. Edit the `JoinTest` class to perform the steps in this practice.
3. Get a list of unique buyer names in a sorted order. Follow these steps to accomplish the task:
   a. Use `map` to get all the buyer names.
   b. Use `distinct` to remove duplicates.
   c. Use `sorted` to sort the names.
   d. Use `joining` to join the names together in the output you see in the following.
4. When complete, your output should look similar to the following:

```
=== Sorted Buyer's List ===
Buyer list: Acme Electronics, Best Deals, Great Deals, Mom and Pops,
PriceCo, Radio Hut
```

Practice 9-8: Grouping Data with Streams

Overview

In this practice, create a `Map` of transaction data using the `groupingBy` method from the `Collectors` class.

Assumptions

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

Tasks

1. Open the `SalesTxn09-08Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/09-LambdaOperations /practices/practice8`.
   - Select `SalesTxn09-08Prac` and click Open Project.
2. Edit the `GroupTest` class to perform the steps in this practice.
3. Populate the `Map` by using the stream `collect` method to return the list elements grouped by buyer name.
   a. Use `Collectors.groupingBy()` to group the results.
   b. Use `SalesTxn::getBuyerName` to determine what to group by.
4. Print out the result.
5. Use the `printSummary` method of the `SalesTxn` class to print individual transactions.
6. Your output should look similar to the following:

```
=== Transactions Grouped by Buyer ===

Buyer: PriceCo
ID: 17 - Seller: Dave Smith - Buyer: PriceCo - Product: Widget Pro - ST: CO - Amt:
240000.0 - Date: 2013-03-20
ID: 18 - Seller: Dave Smith - Buyer: PriceCo - Product: Widget - ST: CO - Amt:
300000.0 - Date: 2013-03-30
ID: 20 - Seller: John Adams - Buyer: PriceCo - Product: Widget - ST: MA - Amt:
280000.0 - Date: 2013-07-14
ID: 21 - Seller: John Adams - Buyer: PriceCo - Product: Widget Pro - ST: MA - Amt:
640000.0 - Date: 2013-10-06

Buyer: Acme Electronics
ID: 11 - Seller: Jane Doe - Buyer: Acme Electronics - Product: Widgets - ST: CA -
Amt: 60000.0 - Date: 2013-01-25
ID: 12 - Seller: Jane Doe - Buyer: Acme Electronics - Product: Widget Pro - ST: CA
- Amt: 400000.0 - Date: 2013-04-05

Buyer: Radio Hut
ID: 13 - Seller: Jane Doe - Buyer: Radio Hut - Product: Widget Pro - ST: CA - Amt:
2000000.0 - Date: 2013-10-03
ID: 15 - Seller: Betty Jones - Buyer: Radio Hut - Product: Widget Pro - ST: CO -
Amt: 800000.0 - Date: 2013-02-04
ID: 23 - Seller: Samuel Adams - Buyer: Radio Hut - Product: Widget Pro - ST: MA -
Amt: 1040000.0 - Date: 2013-12-08
```

```
Buyer: Mom and Pops
ID: 22 - Seller: Samuel Adams - Buyer: Mom and Pops - Product: Widget - ST: MA -
Amt: 60000.0 - Date: 2013-10-02

Buyer: Best Deals
ID: 16 - Seller: Betty Jones - Buyer: Best Deals - Product: Widget - ST: CO - Amt:
500000.0 - Date: 2013-03-21
ID: 19 - Seller: Betty Jones - Buyer: Best Deals - Product: Widget Pro - ST: CO -
Amt: 800000.0 - Date: 2013-07-12

Buyer: Great Deals
ID: 14 - Seller: John Smith - Buyer: Great Deals - Product: Widget - ST: CA - Amt:
100000.0 - Date: 2013-10-10
```

Practices for Lesson 10: The Module System

Practices for Lesson 10: Overview

Overview

In these practices, you will explore how to create Java modular applications using both the command line and NetBeans. You'll also create and optimize a runtime image by using `jlink`.

Summary of Keywords:

Keyword and syntax	Description
`export <package>`	Declares which package is eligible to be read.
`export <package> to <module>`	Declares which package is eligible to be read by a specific module.
`requires <module>`	Specifies another module to read from.
`requires transitive <module>`	Specifies another module to read from. The relationship in transitive in that indirect access is given to modules requiring the current module.

Practice 10-1: Creating a Modular Application from the Command Line

Overview

In this practice, you create a simple single-class Java application and convert it to a modular application.

Assumptions

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

Tasks

1. Login to the Oracle Linux lab environment.
2. Open a terminal in the directory `/home/oracle/labs/10_ModularSystem/practices`.
3. Create a project directory, `Prac_10_01_ModularSystem`.

```
mkdir Prac_10_01_ModularSystem
```

4. Change to this directory

```
cd Prac_10_01_ModularSystem
```

5. Your main class will be in the package `com.greetings`. Create a source directory, `src`, and within it the directory structure to match the package name.

```
mkdir -p src/com/greeting
```

6. Within `src/com/greeting` create a new main class, `Main.java`.
   a. Change directories to `src/com/greeting`

```
cd src/com/greeting
```

   b. Create the `Main.java` class

```
touch Main.java
```

   c. Open `Main.java` in gedit and add the following code. Remember to save your changes.

```
package com.greeting;

public class Main {
    public static void main(String[] args) {
            System.out.println("Hello World!");
    }
}
```

7. From the `~/labs/10_ModularSystem/practices/Prac_10_01_ModularSystem` directory, create a `classes` folder to hold the compiled classes. Then compile the Main.java class to that folder.

   a. Change directories to the `Prac_10_01_ModularSystem` folder. You can back out to this directory level by entering:

   ```
   cd ../../..
   ```

   b. Create the `classes` directory.

   ```
   mkdir classes
   ```

   c. Compile your code.

   ```
   javac -d classes src/com/greeting/Main.java
   ```

   d. You can check the file layout by running the `tree` command from the command line. The output should look like this:

   ```
   .
   ├── classes
   │   └── com
   │       └── greeting
   │           └── Main.class
   └── src
       └── com
           └── greeting
               └── Main.java
   ```

   e. Test the application from the same location. Java needs to know the location of the class. To do specify this, use the `-cp (classpath)` parameter.

   ```
   java -cp classes com.greeting.Main
   ```

   f. You should get the following output:

   ```
   Hello World!
   ```

8. Make this simple application a modular Java application.

   a. Create a module directory in the `src` folder, just above the package folder `com`. Both the module and directory will be named `hello`.

   ```
   cd src
   mkdir hello
   ```

   b. Move `com` to the `hello` directory.

   ```
   mv com hello
   ```

   c. View the file structure by using the command `tree ..`
   (note the double dots after the tree command).

   ```
   .
   ├── classes
   │   └── com
   │       └── greeting
   │           └── Main.class
   └── src
   ```

```
    └── hello
        └── com
            └── greeting
                └── Main.java
```

d. Create a file `module-info.java` and put it in the root directory of the module (parallel with `com`).

```
cd hello
touch module-info.java
```

e. Open the file in gedit and add the following code. Remember to save your work.

```
module hello{


}
```

Note: This code names the module `hello`. It's also located within the `hello` folder. You'll see in the next practice why this naming consistency is very important.

Note: The module is empty. This very simple application uses only one class, `System`, which is included in the `java.base` module. The `java.base` module doesn't need to be explicitly specified as a required module because it is implicitly always present.

f. Change to the `Prac_10_01_ModularSystem` directory and compile the application.

```
cd ..
cd ..
javac -d mods/hello src/hello/module-info.java
src/hello/com/greeting/Main.java
```

Note: `module-info.java` is specified first so the compiler knows this is modular application and will inform you accordingly if any problems exist with the code. Also note how you must specify the `hello` directory as part of the destination.

g. Check the directory structure. Your results will look like this:

```
├── mods
│   └── hello
│       ├── com
│       │   └── greeting
│       │       └── Main.class
│       └── module-info.class
└── src
    └── hello
        ├── com
        │   └── greeting
        │       └── Main.java
        └── module-info.java
```

Note: The `classes` directory is not shown. This directory still exists, but is no longer necessary to run the modular application.

h. Run your new modular application.

```
java -p mods -m hello/com.greeting.Main
```

i. You should get the following output:

```
Hello World!
```

Practice 10-2: Compiling Modules from the Command Line

Overview

In this practice, you see a shortcut to compile all modules at once. You don't need to specify each individual module and class for compilation like you saw in the previous practice. For this shortcut to work, it's important to name modules and their directories consistently.

Assumptions

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

Tasks

1. Ensure that your terminal is open in the directory `~/labs/10_ModularSystem/practices/Prac_10_01_ModularSystem`.
2. Delete your compiled code by removing the `mods` directory.

```
rm –r mods
```

3. Compile all modules at once.

```
javac -d mods --module-source-path src $(find src -name
"*.java")
```

Note: This automatically creates a directory that is the same as the name given in the `module-info` file.

4. Run the application. It should work.

```
java -p mods -m hello/com.greeting.Main
```

5. In the file explorer, navigate to the `src/hello` directory and open the `module-info` class in gedit. Change the name of the class from `hello` to `test`. Remember to save your work.

```
module test{


}
```

Note: Now the directory and `module-info` class have different names.

6. Try compiling all modules at again.

```
javac -d mods --module-source-path src $(find src -name
"*.java")
```

Note: You should get an error message. Below is shown the last 5 lines of this message.

```
module test{
^
error: cannot access module-info
  cannot resolve modules
3 errors
```

7. Close the terminal. The remaining practices are completed in NetBeans.

Practice 10-3: Creating a Modular Application from NetBeans

Overview

In this practice, you create a modular Java application using NetBeans. You'll begin seeing how modules read from each other and how NetBeans allows you to compile many modules at once.

Assumptions

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

Tasks

1. Open NetBeans and create a new project for a modular Java application.
   a. Select **File > New Project**.
   b. Select **Java Modular Project** from the list of project types and click **Next**.
   c. Name the project `HelloNetBeans`
   d. Set the project's location to `Prac_10_03_ModularSystem`.
   e. Click **Finish**.
2. Create the `hello` module within the project.
   a. Right-click on the project in NetBeans.
   b. Select **New > Module**.
   c. Name the module `hello`.
   d. Click **Finish**.

   Note: The `module-info` class is automatically created within the module and displayed for you. Expand the project to see this class in the default package.
3. Create the package `com.greeting` within the `hello` module.
4. Create a new Java Main Class within the `com.greeting` package called `Main`. Add a `main` method that prints out "Hello NetBeans" to the class:

```
package com.greeting;

public class Main {
    public static void main(String[] args) {
          System.out.println("Hello NetBeans!");
    }
}
```

5. Compile and Run the program. Remember to specify `com.greeting.Main` as the main class. You should get the following output:

```
Hello NetBeans!
```

6. Create a new module called `people`.
7. Create the package `com.name` within the `people` module.

8. Create a new Java Class within the `com.name` package called `Names`. Add the following code to the class. Replace Duke's name with your own:

```
package com.name;

public class Names {
    public static String getName(){
        return "Duke!";
    }
}
```

9. Modify the `module-info` class of the `people` module so that it exports the `com.name` package.
10. Modify the `module-info` class of the `hello` module so that it requires the `people` module.
11. Modify the greeting in the `main` method to include the name found in the `Names` class. The print statement will look like this:

```
System.out.println("Hello " +Names.getName());
```

12. You'll notice NetBeans will complain about not finding the `Names` package. In addition to setting up the correct requires and exports statements, you also need to import the relevant packages into your classes. Add this line of code to the Main class.

```
import com.name.Names;
```

13. Build the project.

    Note: NetBeans' output window will report that two JARs are created. NetBeans uses an ant script to find and compile all modules associated with the project.

14. Run the project. Your output window will look like this:

```
Hello Duke!
```

Practice 10-4: Requiring a Module Transitively

Overview

In this practice, you continue working on the modular Java application from the previous practice. You'll create a new module and examine the effects of requiring a module transitively.

Assumptions

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

Tasks

1. Continue editing your NetBeans project from the previous practice. Alternatively, you can open the project `Prac_10_04_ModularSystem`.
2. Create a new module called `conversation`.
3. Create the package `com.question` within the `conversation` module.
4. Create a new Java Class within the `com.question` package called `Questions`. Add the following code to the class:

```
package com.question;

public class Questions {
    public static String getQuestion(){
        return "How are you?";
    }
}
```

5. Append the returned `String` from calling `getQuestion()` to the print statement in the main method. This means the `hello` module must read from the `conversation` module.
   a. The print statement will look like this:

```
System.out.println("Hello " +Names.getName() +" "
+Questions.getQuestion());
```

   b. Add an import statement to the `Main` class.

```
import com.question.Questions;
```

Note: When you save your progress, you'll notice NetBeans complains about the import statement. Although NetBeans acknowledges that the packages exist, the `modules-info` classes must be edited to allow the modules to read from each other.

   c. Modify the `module-info` class of the `conversation` module so that it exports the `com.question` package.

d. Modify the `module-info` class of the `people` module so that it requires transitively the `conversation` module.

Note: Although the `hello` module doesn't explicitly require `conversation`, classes within this module can still read. Transitivity allows for readability up the requirement chain. `hello` requires `people`. `people` requires `conversation` transitively.

6. Run the project. Your output will look like this

```
Hello Duke! How are you?
```

7. What if `people` no longer requires `conversation` transitively?
   a. Remove the keyword `transitive` from the `requires` statement of the `module-info` class of the `people` module.
   b. Try running the project and observe NetBeans' error message.

   Note: Without transitivity, the `hello` module cannot read from `conversation`.

8. What if `conversation` exports to only `people`?
   a. Add the keyword `transitive` back to the `requires` statement of the `module-info` class of the `people` module.
   b. Modify the `module-info` class of the `conversation` module. Specify that `conversation` exports `com.question` to only `people`.
   c. Try running the project and observe NetBeans' error message.

   Note: The `to` keyword breaks transitivity in this scenario.

Practice 10-5: Beginning to Modularize an Older Java Application

Overview

In this practice, you see how modularization affects a program's readability of certain Java APIs. Your application focuses on Java's `Logger` class. More information on migrating a Java Application to work with Java SE 9 and modularization is discussed in a later lesson.

Assumptions

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

Tasks

1. Open the NetBeans project `Prac_10_05_ModularSystem`.
2. You'll notice the program won't run because it doesn't yet recognize the `Logger` object type. Make the necessary import in the `Main` class to get the program to run.

```
import java.util.logging.Logger;
```

3. Run the program. Your output will look like this:

```
Nov 15, 2018 2:36:26 PM com.example.Main main
INFO: HelloWorld App says hello!
```

4. A simple way to start modularizing the project is to introduce a `module-info` class into the default package.
   a. Right-click on the **Source Packages** folder and select **New > Other > Java Module Info**.
   b. Click **Next**.
   c. Click **Finish**.
5. Try running the program again. You'll notice you can't.

   Note: With the modularization of Java, Java packages in the JDK have been sorted into modules. Not all Java packages are automatically accessible to a modular application. A great many packages are stored in the `java.base` module. There's no need to explicitly require this module because it's automatically accessible to all modules. The `Logger` class is stored in the `java.logging` module. This module does need to be explicitly required.
6. Open the `module-info class` and require the `java.logging` module.
7. Run the program again. It should work this time.

Practice 10-6: Creating and Optimizing a Custom Runtime Image by Using `jlink`

Overview

In this practice, you will create and optimize a custom runtime image in Java SE 9 by using the `jlink` tool.

Assumptions

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

Tasks

1. To create a custom runtime image, perform the following tasks:
   a. Open a terminal and navigate to the `Hello` directory and execute the following command:

```
$ cd
/home/oracle/labs/10_ModularSystem/practices/Prac_10_06/Hello

$ jlink --module-path  dist/Hello.jar:/usr/java/jdk-11.0.1/jmods
--add-modules com.greeting --output myimage
```

   The command creates a new directory, `myimage`, which contains a Java Runtime Environment customized to run the `Hello` application.

2. Examine the contents of the `myimage` directory.

   Execute the following commands to navigate to the `myimage` directory and observe the directories:

```
$ cd myimage

$ ls
```

3. Check the runtime of the custom runtime image created.
   a. Execute the following commands:

```
$ cd bin

$ ./java –version
java version "11.0.1" 2018-10-16 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.1+13-LTS)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.1+13-LTS,
mixed mode)
```

   Observe that it shows that it's a JDK 11 runtime.

4. Check the modules that are part of the `myimage` custom runtime image by executing the following command:

```
$ ./java --list-modules
com.greeting
java.base@11.0.1
java.logging@11.0.1
```

Observe that `com.greeting`,`java.base` and `java.logging` are listed as three modules that are part of `Hello`.

5. Examine the footprint of the custom runtime image, and execute the following two commands:
   a. Execute the following command to know the size of the JDK 11 installed in your machine:

```
$ du –sh /usr/java/jdk-11.0.1
290 M
```

   b. Execute the following commands to know the size of the custom runtime image created (the `cd` is to return to the `myimage` folder):

```
$ cd ..

$ du –sh
  48 M
```

   Observe the reduced size of the custom runtime image.

6. You can run an application from the custom runtime image.

   The following commands show the execution of the `Hello` application from its custom runtime image, `myimage`:

   a. Execute the following commands:

```
$ cd bin
$ ./java --module com.greeting
```

   b. The following output is displayed:

```
Nov 17, 2018 5:44:57 AM com.greeting.Main main
INFO: HelloWorld App says hello!
```

7. You can optimize the custom runtime image by using a `jlink` plug-in, and `compress` to create a reduced JDK 11 runtime image. You can also enable compression and remove some debugging features on a production system.

   The following command creates a runtime image in the `compressmyimage` directory that is stripped of debug symbols, and uses compression to reduce space:

   a. Execute the following commands to create a custom runtime image, `compressmyimage`:

```
$ cd
/home/oracle/labs/10_ModularSystem/practices/Prac_10_06/Hello
$ jlink --module-path dist/Hello.jar:/usr/java/jdk-11.0.1/jmods
--add-modules com.greeting --output compressmyimage
--strip-debug  --compress=2
```

8. Compare the size of the `myimage` and `compressmyimage` custom runtime images:

   a. Execute the following command:

```
$ du -sh ./compressmyimage ./myimage

33M ./compressmyimage
48M ./myimage
```

   Note: The output generated is an example and may not be exactly the same on your system.

Practice 10-7: Using NetBeans to Create and Optimize a Runtime Image

Overview

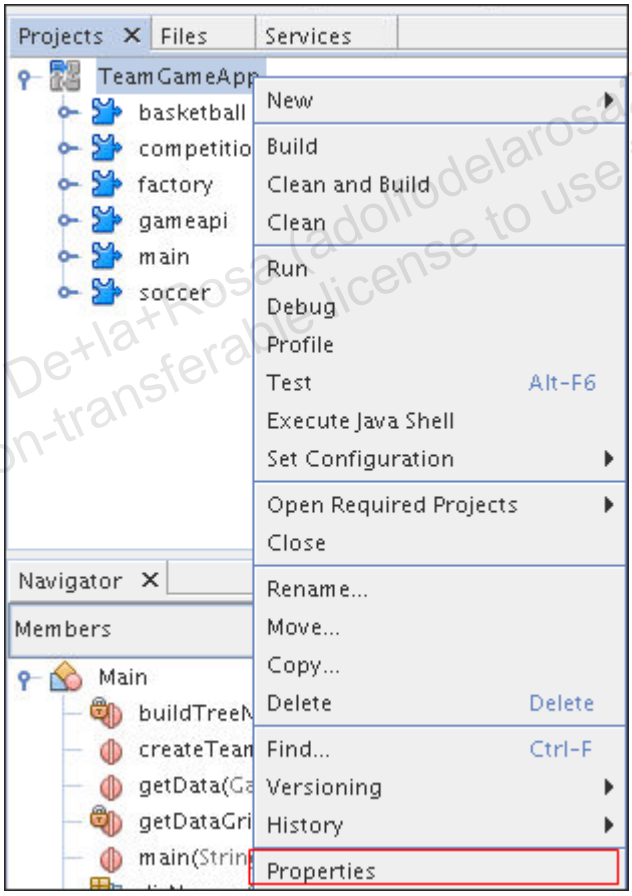
In this practice, in NetBeans, you will create and optimize a custom runtime image in JDK 11 by using the `jlink` tool.

Assumptions

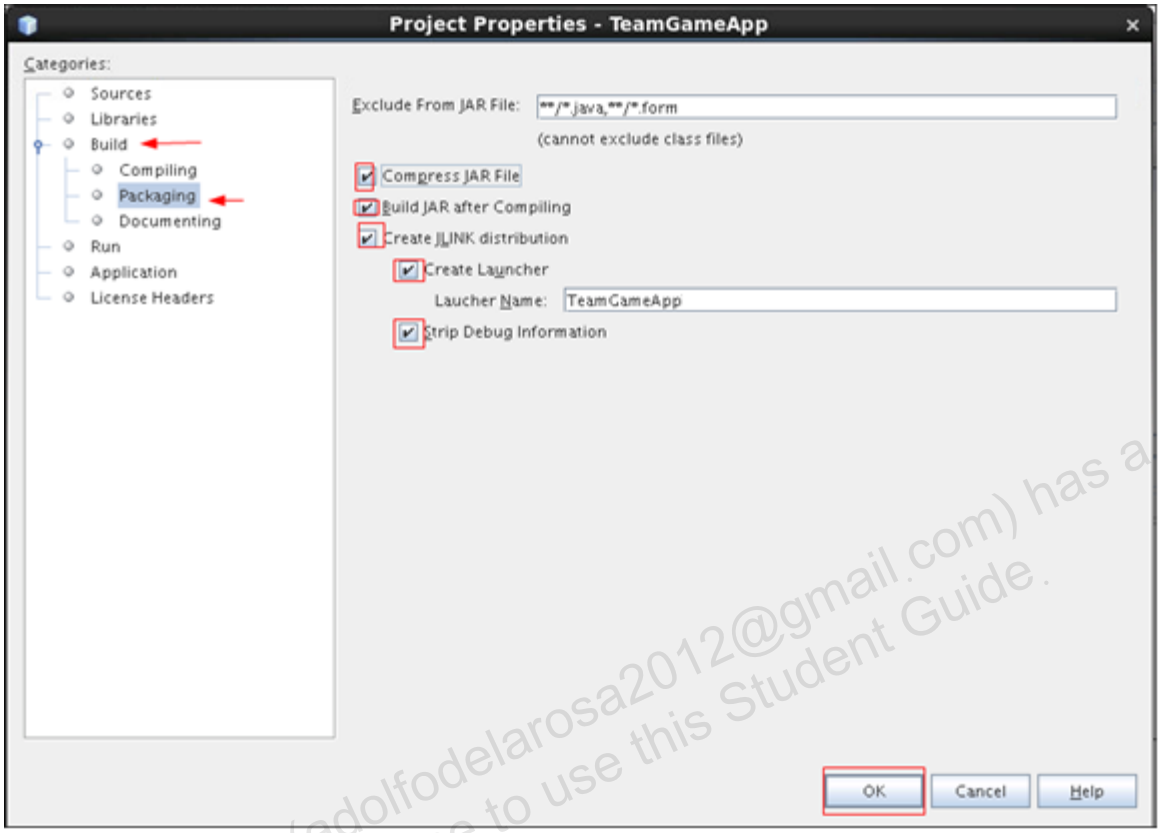
You have completed the lecture and reviewed the overview for this practice.

Tasks

1. In NetBeans, open the `TeamGameApp` project from `/home/oracle/labs/10_ModularSystem/practices/Prac_10_07/TeamGameApp` and perform the following tasks:
   a. Right-click the project and select Properties.



In the Project Properties window, select build in the Categories column, and then select Packaging.



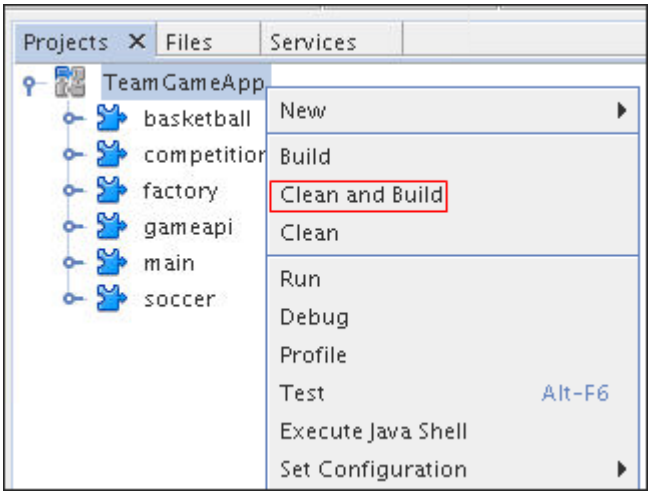
b. Select the following options:

1) Compress JAR File
2) Build JAR after Compiling
3) Create JLINK distribution and then Create Launcher
    a) Launcher Name: Accept the default name.
4) Strip Debug Information

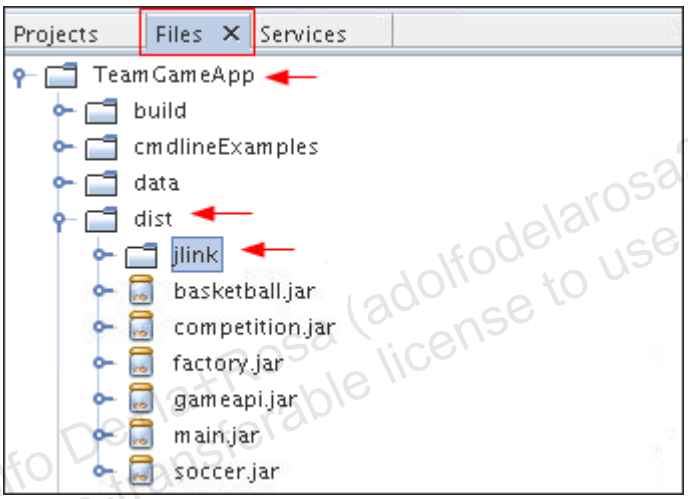
c. Click OK.

The options Compress JAR File and Strip Debug Information are used to compress and optimize the custom runtime image created.

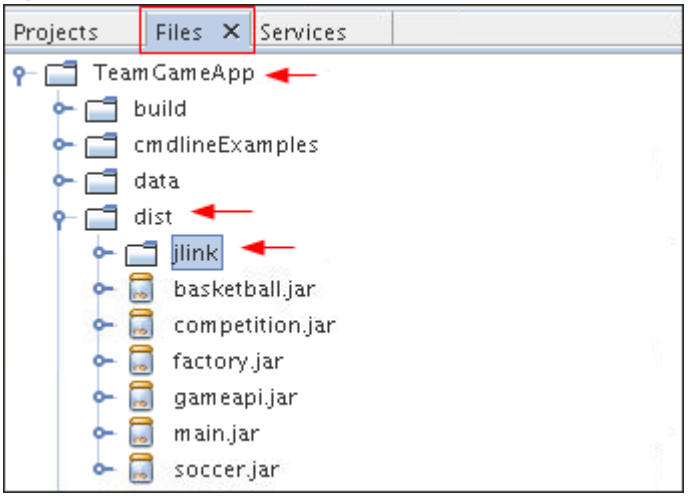
2. Right-click the `TeamGameApp` project and select “Clean and Build.”



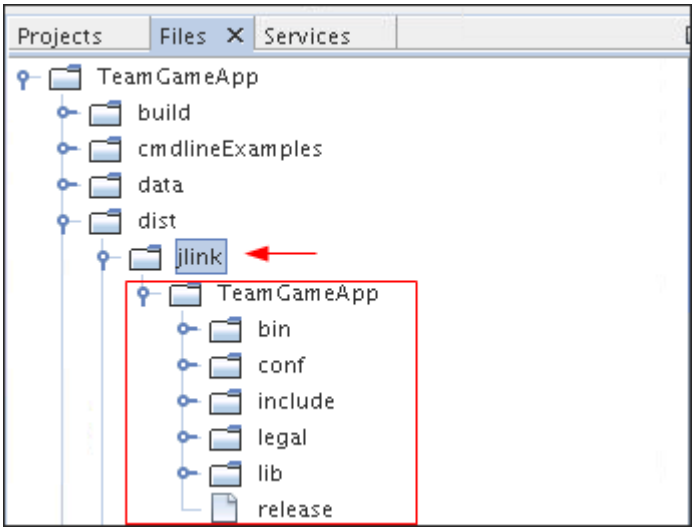
3. In NetBeans, click the **Files** tab and in `TeamGameApp`, expand the `dist` folder.



4. Observe the `jlink` folder created; it contains the custom runtime image, `TeamGameApp`.



5. Expand the `jlink` folder and observe the `TeamGameApp` folder.



6. Check the modules that are part of the custom runtime image by executing the following commands:

```
$ cd
/home/oracle/labs/10_ModularSystem/practices/Prac_10_07/TeamGame
App/dist/jlink/TeamGameApp/bin/

$ ./java --list-modules
basketball
competition
display.ascii
factory
gameapi
java.base@11.0.1
main
soccer
```

7. To run the application from the custom runtime image created. Execute the following steps:
   a. Browse to the `TeamGameApp` root folder:

```
$ cd /home/oracle/labs/
10_ModularSystem/practices/Prac_10_07/TeamGameApp
```

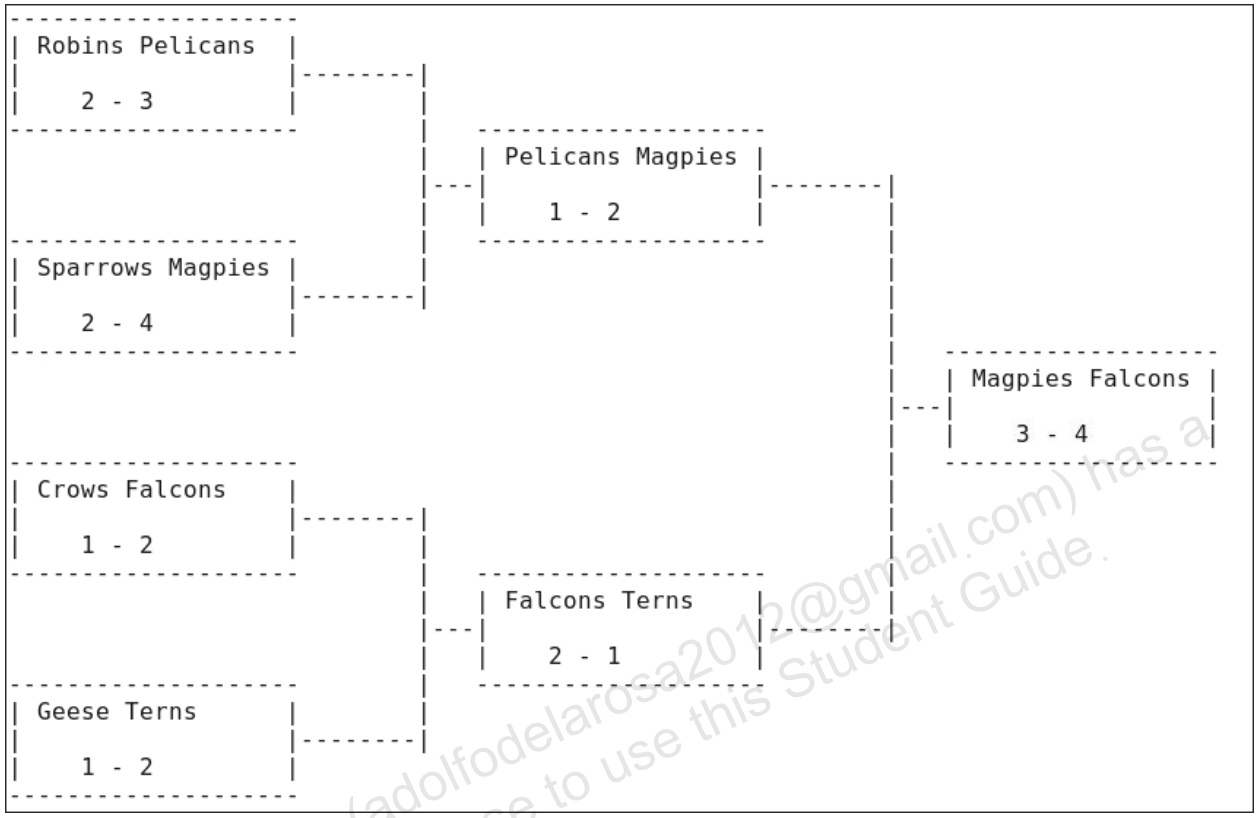
   To execute the `TeamGameApp` application, you require the `Authors.txt` file, which is located in `data` directory.

   b. Execute the following command:

```
$ ./dist/jlink/TeamGameApp/bin/java --module
main/tournament.Main
```

c. The following output is displayed:

Note: This is an example output and it changes each time you execute the command.



Practices for Lesson 11: Migration

Practices for Lesson 11: Overview

Overview

In these practices, you will migrate an application written in SE 8 to a modular application. Along the way, you'll deal with many of the issues that may arise when making this transition.

Practice 11-1: Examining the League Application

Overview

In this practice, you examine an application written in Java SE 8 and assess how to migrate it to a modular application. You'll see some of the pitfalls of trying unorthodox approaches to achieve encapsulation at the JAR level in Java SE 8, and how the class path can exacerbate some of the problems.

Assumptions

Tasks

1. Examine the application to be migrated. Open the following projects in `~/labs/11_Migration/practices`.
   - League
   - Soccer_1pkg
   - Basketball_1pkg

   Here is a description of each of these projects.

League

The main project contains the classes that organize a league style competition.

League contains three packages: `game`, `main`, and `util`.

The `game` package - contains the API for the application, including all the interfaces used for managing games, and the Factory class for creating objects based on these interfaces. The Factory class returns implementations of the API for a particular game.

The `utils` package - contains a Settings class with values for running the league.

The `main` package - contains the Main class that runs the application.

League also contains the following libraries:

- `Soccer_1pkg.jar` - an implementation of a soccer game.
- `Basketball_1pkg.jar` - an implementation of a basketball game.
- `display-ascii-0.1b.jar` - a library for displaying data in grids or trees using just ASCII characters.

Soccer_1pkg

The Soccer_1pkg project is one of two currently available projects to represent types of games that can be used by the League class.

Soccer_1pkg contains two packages: `game`, and `util`.

The `game` package - contains the classes that implement the interfaces in the League project. Notice that the package name is the same as the package name in League; this is to allow all these classes to be package protected, so that they can only be accessed through classes in the game package in League.

As you'll see, although this arrangement can provide a weak form of encapsulation at the JAR level, it introduces other problems.

The `util` package - contains a `Settings` class with values for running the `Soccer` class.

Basketball_1pkg

Contains exactly the same organization as Soccer_1pkg, but represents a basketball game.

2. Run the application by running the League project. You should see a grid something like the following that shows the results of running a league competition. Note that the results will not be identical as the League project generates game results randomly each time it is run.

```
-------------------------------------------------------------------------------------------------
|          | Sparrows | Falcons | Crows | Pelicans | Robins | Magpies | League Pts | Total Goals |
-------------------------------------------------------------------------------------------------
| Sparrows |  X       | 5 - 2   | 2 - 4 | 3 - 2    | 4 - 3  | 1 - 0   | 15         | 31          |
-------------------------------------------------------------------------------------------------
| Falcons  | 3 - 4    |  X      | 2 - 1 | 1 - 1    | 1 - 1  | 2 - 2   | 10         | 23          |
-------------------------------------------------------------------------------------------------
| Crows    | 2 - 5    | 3 - 3   |  X    | 2 - 4    | 1 - 0  | 1 - 1   | 10         | 21          |
-------------------------------------------------------------------------------------------------
| Pelicans | 1 - 1    | 2 - 2   | 1 - 2 |  X       | 4 - 4  | 2 - 0   | 10         | 21          |
-------------------------------------------------------------------------------------------------
| Robins   | 3 - 2    | 1 - 5   | 0 - 5 | 3 - 3    |  X     | 2 - 0   | 8          | 19          |
-------------------------------------------------------------------------------------------------
| Magpies  | 3 - 4    | 2 - 2   | 1 - 0 | 1 - 1    | 2 - 2  |  X      | 7          | 12          |
```

In the grid heading, League Pts represents the number of points based on games won, drawn, or lost andTotal Goals represents the goals scored over all games. Total Goals is used to break any ties on League Pts.

3. Look at the log for the application (players are based on a list of famous authors and are randomly chosen each time).
   a. Open a terminal and navigate to `labs/11_Migration/practices/League/data`
   a. Use `grep` to show just the relevant lines in the log.

```
grep FINE soccer.log | more
FINE: ****** New game ******
FINE: Kickoff
FINE: Stephen King of the Pelicans team -- Receive pass
FINE: Stephen King of the Pelicans team -- Pass attempt
FINE: Rudyard Kipling of the Pelicans team -- Receive pass
FINE: Rudyard Kipling of the Pelicans team -- Pass attempt
FINE: Marguerite Yourcenar of the Pelicans team -- Receive pass
FINE: Marguerite Yourcenar of the Pelicans team -- Dribble
FINE: Marguerite Yourcenar of the Pelicans team -- SHOOTS
FINE: Marguerite Yourcenar of the Pelicans team -- GOAL!
FINE: Samuel Beckett of the Robins team -- Kickoff
...
```

4. Examine the application.
   a. Open the Soccer_1pkg project and expand the `game` package.
   b. Open the class `Dribble`. This class implements a type of event in a soccer game. Notice that the class is not `public`. This makes it more difficult to access these classes except via the public API in League. However, the drawback is that the package name is, and must be, the same as that in the League project. It also means that all of these classes must be in the `game` package, even though it may well make sense to organize them into two or more packages, say, `soccer.play`, and `soccer.event`.
5. Open the `Main` class in the package `main` of the League project.
6. Try running the application with a different library, this time the `Basketball_1pkg.jar` library.
   a. At the start of the `main` method, change the value in the `String gameType` to `"basketball"` and rerun the application. You will see an error similar to the screenshot below (some line numbers may be slightly different).

```
run:
Exception in thread "main" java.lang.ArrayStoreException: game.ReceivePass
        at game.StartPlay.getNextEvents(StartPlay.java:26)
        at game.StartPlay.getNextEvents(StartPlay.java:15)
        at game.BasketballTeam.getNextPlayAttempt(BasketballTeam.java:216)
        at game.Basketball.playGame(Basketball.java:123)
        at game.League.createAndPlayAllGames(League.java:380)
        at main.Main.main(Main.java:72)
/home/kenny/.cache/netbeans/8.2/executor-snippets/run.xml:53: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

   This error illustrates one of the dangers of splitting package names across different JAR files. If you look in the `game` package in the Basketball_1pkg project and the Soccer_1pkg project, you will see that some class names exist in both packages. This is a problem, because these identically named classes are different and contain different code, each pertaining to the game in question. The way the class path works is involved here too because when a class is found on the class path, it will not be reloaded, even if found again.
   b. Right click on the League project and select Properties.
   c. Select Libraries in the left panel.

      Note that there are three libraries on the class path. Also, note that `Soccer_1pkg.jar` appears before `Basketball_1pkg.jar`.
   d. Click on `Soccer_1pkg.jar` to highlight it, then click Remove.
   e. Now click Add JAR/Folder to add it again. Navigate to `~/labs/11_Migration/practices/League/lib`, then select `Soccer_1pkg.jar`, select Relative Path, and click Open.

      Notice that `Soccer_1pkg.jar` now occurs after `Basketball_1pkg.jar` in the class path.
   f. Click OK to close the Project Properties.

7. Try running the application again (by running the League project). It should work and now the grid shows more basketball-like scores.

   This is an example of just how problematic the class path can be. If a class is implemented more than once, it's very easy to load the wrong one. Moreover, there is no warning that there is a problem, and as you saw, the error given is not necessarily very helpful.

8. Change the `gameType` field back to `"soccer"`, then change the position of the JAR files in the library so that `Soccer_1pkg.jar` is listed first, and rerun the application again.

9. Look at the Library JDK used by League, it's JDK 1.8. As the first step of migration, try running the application with JDK 11. You can use the command line to run the application under Java SE 11.

   a. In a terminal, navigate to `~/labs/11_Migration/practices/League/`.

   b. Check that the default version of Java run from the command line is SE 11.

   ```
   java --version
   ```

   c. Run the application with the command:

   ```
   java -cp dist/*:lib/Soccer_1pkg.jar:lib/display-ascii-0.1b.jar
   main.Main
   ```

   Note that you could use an asterisk instead of specifying `Soccer_1pkg.jar` and `display-ascii-0.1b.jar` by name, but then you'd run a risk of loading the Basketball_1pkg.jar library first. This is an example of the care which is sometimes needed in setting up class paths.

   You should see the application running without any modifications. However, it is not a modular application.

10. As a first move towards modularizing this application, a useful approach is to make each of the projects a separate module. However, we can predict that there will be a problem because of the common package names and classes in the League Project and in the Soccer_1pkg and Basketball_1pkg projects. This configuration is called a split package and is not permitted between modules. Additionally, the command line tool `jdeps` will not give us useful information about dependencies with this configuration, as it won't see the JARs as distinct entities.

11. Modify the League project to use new basketball and soccer JARS that do not share a package name between the API and its implementation. These projects, named Basketball and Soccer, have been set up so that all implementations are public. This is also not ideal, but we will address this issue later using the module system rather than attempting to address it using the class access modifiers.

    a. Close the projects Basketball_1pkg and Soccer_1pkg

    b. Open the projects Basketball and Soccer (also in `~/labs/11_Migration/practices`)

    c. Right click the League project and click Properties. Select Libraries and then remove `Soccer_1pkg.jar` and `Basketball_1pkg.jar` and click OK. This will cause the League project to show errors - you can ignore them for now.

    d. Select the Files tab to display the files in each project.

    e. Expand Basketball, Soccer, and League

f. In the lib folder of League, delete `Soccer_1pkg.jar` and `Basketball_1pkg.jar`. (Select them, highlight, then right click and select delete).

g. Copy the Soccer.jar file from `Soccer/dist` to `League/lib`. You do this in the Files tab by dragging `Soccer.jar` using the mouse, but pressing Ctrl before releasing the mouse button.

h. Copy the Basketball.jar file from `Basketball/dist` to `League/lib`.

i. Select the Projects tab then right click on Libraries (in League project) and select Add JAR/folder...

j. In the dialog, navigate to `League/lib` and select `Soccer.jar` and `Basketball.jar`, then click Open. Use the Relative Path setting.

12. Open the `Factory` class in League to address the errors in League.

a. Click the topmost of the errors (red marks to the right of the editor scroll bar. This will take you to the first error. This error is due to the implementation of the `Game` interface no longer being in the `game` package.

b. With the cursor on the same line as the error, press Alt-Enter (or click the suggestion icon on the left of the editor pane). Then, in the menu, select add import for `soccer.SoccerTeam`. This should clear this error. If Alt-Enter doesn't give you this prompt, simply add import `soccer.SoccerTeam` to the class.

c. Repeat this process for all the errors in the `Factory` class.

d. When you have addressed all the errors, save the file. This should clear all errors from the project.

13. Test the application

a. Clean and build the application, then try running it. It should run successfully as before.

b. Try changing between "soccer" and "basketball" as before (change the value of `gameType` in the Main class. It should work correctly for either "soccer" or "basketball".

c. Open a terminal and navigate to the League folder.

d. As before, try running the application from the command line with the command:

```
java -cp dist/*:lib/* main.Main
```

Note how you can now use an asterisk to specify all JARs in lib, as the order the JARs are loaded in doesn't matter. The application should work correctly. (If there is a problem, make sure you rebuilt the League project).

Practice 11-2: Using jdeps to Determine Dependencies

Overview

In this practice, you use `jdeps` to examine the application and plan how you will complete a top-down migration to a modular implementation. As each JAR file will become a module, you determine the dependencies for each JAR.

Assumptions

Tasks

1. Run `jdeps` on the League application.
   a. In a terminal open in `~/labs/11_Migration/practices/League`, run the following command:

```
jdeps -summary -cp
lib/Basketball.jar:lib/Soccer.jar:lib/display-ascii-0.1b.jar
dist/League.jar


League.jar -> lib/Basketball.jar
League.jar -> lib/Soccer.jar
League.jar -> lib/display-ascii-0.1b.jar
League.jar -> java.base
League.jar -> java.logging
```

   (Note that you could substitute `build/classes` for `dist/League.jar` and get essentially the same result).

```
jdeps -summary -cp
lib/Basketball.jar:lib/Soccer.jar:lib/display-ascii-0.1b.jar
build/classes


classes -> lib/Basketball.jar
classes -> lib/Soccer.jar
classes -> lib/display-ascii-0.1b.jar
classes -> java.base
classes -> java.logging
```

   This is a simple application, but you can see that the main project, League, requires `Basketball.jar`, `Soccer.jar`, `display-ascii-0.1b.jar, java.base`, and `java.logging`.

   b. What about `Soccer.jar`, `Basketball.jar`, and `display-ascii-0.1b.jar`? By adding -R (or -recursive) requirements for these JARs will be listed also. Try this.

```
jdeps -R -s -cp lib/Basketball.jar:lib/Soccer.jar:lib/display-
ascii-0.1b.jar dist/League.jar


Basketball.jar -> dist/League.jar
```

```
Basketball.jar -> java.base
Basketball.jar -> java.logging
League.jar -> lib/Basketball.jar
League.jar -> lib/Soccer.jar
League.jar -> lib/display-ascii-0.1b.jar
League.jar -> java.base
League.jar -> java.logging
Soccer.jar -> lib/Basketball.jar
Soccer.jar -> dist/League.jar
Soccer.jar -> java.base
Soccer.jar -> java.logging
display-ascii-0.1b.jar -> java.base
```

Note that using dist/* in the class path has the same effect as –R.

c. Now you see that `Basketball.jar` and `Soccer.jar` in turn require `League.jar`, `java.base` and `java.logging`, while `display-ascii-0.1b.jar` requires only `java.base`. One problem for future modularization is that `League.jar` requires `Soccer.jar`, and `Soccer.jar` requires `League.jar`. This is called a cyclic dependency and if these JARs are to become modules, this will not be permitted by the module system. You deal with this later when you create a modular application. Java SE 8 or earlier permits this cyclic dependency. For the Java SE 8 or earlier runtime, the contents of all JARs just become a list of classes on the class path. (However, many IDEs recognize the problematic aspect of this arrangement, and do not allow projects to depend mutually on each other. If we had tried to do this in NetBeans by using projects instead of adding JAR files to the libraries of each project, the IDE would have prohibited this approach).

2. `jdeps` can also write out the `module-info.java` file for you.

a. Try this now by using this command from the command line at `~/labs/11_Migration/practices/League`. (In the example below, `module-info-files` is the name of the directory to be created).

```
 jdeps --generate-module-info module-info-files lib/* dist/*

writing to module-info-files/display.ascii/module-info.java
writing to module-info-files/League/module-info.java
writing to module-info-files/Soccer/module-info.java
writing to module-info-files/Basketball/module-info.java
```

You may get an error here if you didn't previously delete `Basketball_1pkg.jar` and `Soccer_1pkg.jar` from the lib folder of League.

b. Take a look in the `module-info-files` directory using the `tree` command.

```
tree module-info-files

module-info-files
├── Basketball
│   └── module-info.java
├── display.ascii
│   └── module-info.java
├── League
│   └── module-info.java
└── Soccer
    └── module-info.java
```

c. Look in the `module-info.java` file generated for the League project.

```
more module-info-files/League/module-info.java

module League {

    requires Basketball;
    requires Soccer;
    requires display.ascii;
    requires java.logging;

    exports game;
    exports main;
    exports util;

}
```

This can provide useful information for setting up modules later.

Practice 11-3: Migrating the Application

Overview

In this practice, you begin the process of top down migration. For this practice and the remainder of these practices, you use a version of NetBeans that supports Java SE 9 and its module system.

Assumptions

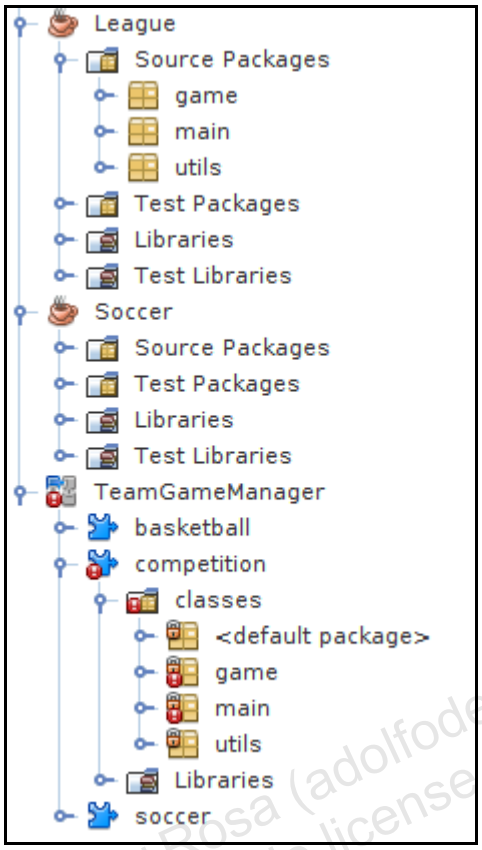
Tasks

1. Create a new modular Java project.
   a. Create a project (File > New Project) and choose Java Modular Project, then click Next.
   b. Give the project the name TeamGameManager, select JDK 11 as the platform, and make sure that it is being created in `~/labs/11_Migration/practices`.
   c. Right click on TeamGameManager, and select Properties, then:
      - In the Libraries category set Java Platform to JDK 11
      - In the Sources category set Source/Binary format to JDK 11

   Because the Soccer and Basketball projects implement interfaces defined in the League project, you need to migrate `League.jar`, `Soccer.jar`, and `Basketball.jar` at the same time.

   Later you migrate the library, `display-ascii-0.1b.jar`. Starting with the main application and eventually migrating the libraries (where possible) is top-down migration.
   d. Right click on TeamGameManager and select New > Module. Name it `competition`. It will contain the code for running the league. In the Java SE 8 version of the application, this functionality was in the League project, but later you add more functionality for other types of competition, so for the new module name `competition` is more appropriate.
   e. Create two more modules named `soccer` and `basketball`. Modules `soccer` and `basketball` are for the functionality that was previously in the JAR files, `Soccer.jar` and `Basketball.jar`.
2. Copy the various packages from the Java SE 8 version of the League project into the module `competition` of the modular application TeamGameManager. As you do so, a number of errors will appear that you can ignore for now.
   a. Expand the League project and the folder Source Packages within it.
   b. Expand the `competition` module in the TeamGameManager project.

c. Drag the three packages, `game`, `main`, and `utils` to the classes folder in the `competition` module. Before releasing the mouse button, press Ctrl so that the packages are copied rather than moved.



Note that sometimes packages other than those shown in the screenshot may also show errors. It can take NetBeans some time to resolve all the class and module dependencies.

3. Copy the `soccer` and `util` packages from the SE8 Soccer project to the `soccer` module.
   a. Expand the `soccer` module, and then expand Source Packages in the Soccer project.
   b. Drag the `soccer` and `util` packages into the classes folder in the `soccer` module. Press Ctrl before releasing the mouse button so that the packages are copied rather than moved.
4. Repeat this process with the Basketball project and the `basketball` module.
5. Examine the errors that highlighted by the IDE.
   a. Expand the `soccer` package in the soccer module. Notice that the errors are all in the implementations of the various interfaces.
   b. Examine the errors in the `competition` module. Note that they are in the `Factory` and `Main` classes. (They may show up elsewhere also, but NetBeans will resolve these as time passes).

   Classes in the `soccer` module are dependent on the interfaces they implement that are in the `competition` module. However, classes in the `competition` module are dependent on the implementations that are in the `soccer` module (and those in the `basketball`

module too). It is not possible to set up these dependencies as to attempt to do so would be to attempt to implement cyclic dependency. Cyclic dependencies are not permitted.

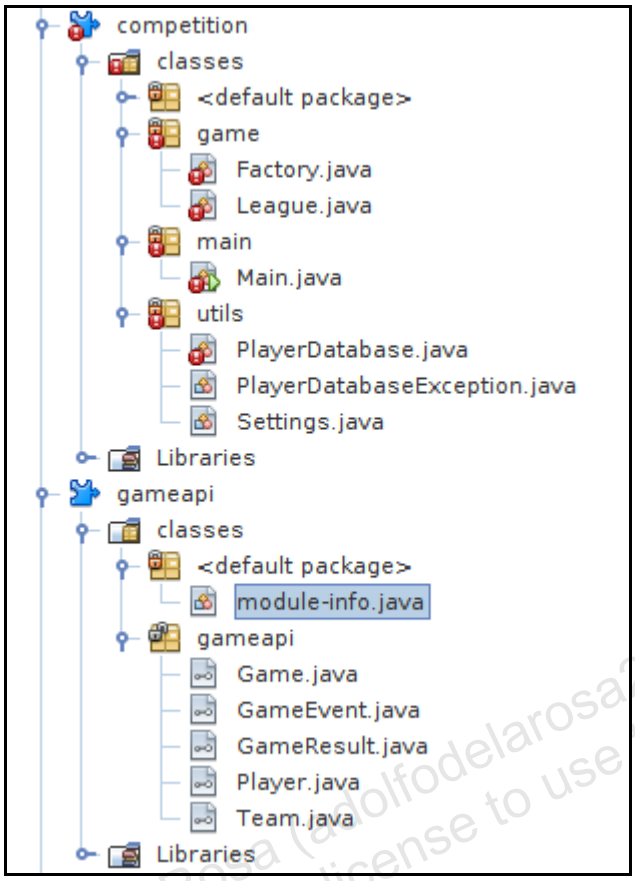
There are several ways to address this, but the simplest is to move the interfaces to their own module, thus breaking the cycle. Another, usually superior, approach is to use services. The next lesson is about services.

6. Create a new module named `gameapi`.
   a. Within this `gameapi` module create a package named `gameapi` (Note that you cannot have a package named `game` as this would be a split package. As you saw, duplicate packages names are possible in Java SE 8 across JARs, but are not permitted in the Java SE 9 module system.
   b. Follow these instructions very carefully!
      Move the interface classes from the `game` package in competition to the `gameapi` package in the module `gameapi`. Make sure you move them to the `gameapi` ***package*** of the `gameapi` module. In this case, do not copy, but move the classes. When you release the mouse button, click Refactor (the package name is different and the IDE can help make the change). The interface classes to move are these five:

      ```
      Game
      GameEvent
      GameResult
      Player
      Team
      ```

      It may be helpful to note that in NetBeans interfaces are recognizable as they have a different icon than classes.

Note that you may see an error dialog, as dependencies are still not set up. If you do see a warning, click Refactor again.



Next, you begin to set up dependencies.

You could look at the `module-info.java` files you created using `jdeps`, and if the application were more complex, that could be helpful. For this application, it should be easy to decide what is needed by considering one module at a time, and adding the necessary imports to fix the various errors that show up in the IDE. You can always refer to the `jdeps` created `module-info.java` files for help.

7. Both the `competition` and `soccer` modules need access to the API so you set up that dependency first.
   a. Open `module-info.java` for the `gameapi` module and add a line to export the package `gameapi`. The `gameapi` module doesn't require any other modules so you should see no errors.

```
module gameapi {
    exports gameapi;
}
```

   b. Open the `module-info.java` file of the `soccer` module. As this module contains implementations of the API, it requires the `gameapi` module. Add this to the `module-info.java` file.

c. You also need access to the `java.logging` module so add that as well.

```
module soccer {
    requires gameapi;
    requires java.logging;
}
```

8. This will allow you to fix some of the errors. Previously all the classes with "Soccer" in their title (`SoccerEvent`, `SoccerPlayer`, etc.) imported their interfaces from the `game` package. This import will now need to be from the `gameapi` package. Fix this now.
   a. Open the `Soccer` class, in the `soccer` package, and you will see imports with errors. Modify each of the imports that use the package `game` so the package name is now `gameapi`. Find and Replace may be helpful for this, but be careful to change only `import` statements.
   Note that this won't fix all the errors in the `Soccer` class.
   b. Repeat this process for each of the other `Soccer*` classes (all the classes whose names begin with "Soccer") and save the project.
   This should fix all the errors in the `soccer` module. Note that even when you have fixed all the errors it may take NetBeans a short while to reconcile everything and it shows no errors for the `soccer` module.
9. Now all the errors in the `soccer` module should have gone. However, the soccer classes are not visible in the `competition` module. All the implementations in the `soccer` module of the interfaces in the `gameapi` module are needed in the `competition` module.
   a. Add `exports soccer;` to the `module-info.java` for the `soccer` module.

```
module soccer {
    requires gameapi;
    requires java.logging;
    exports soccer;
}
```

   b. Now set up the `basketball` module's `module-info.java` file in a similar way:

```
module basketball {
    requires gameapi;
    requires java.logging;
    exports basketball;
}
```

   c. Fix the import errors in the `Basketball*` classes in the basketball module, just as you did for the `Soccer*` classes.
10. Open the `Factory` class in the `game` package of the `competition` module. At the top of the class, you can see that the implementations of the interfaces are not available.
    a. Add `requires gameapi;` to the `module-info.java` for the competition module.

11. Next, fix the error on the imports in the basketball and soccer packages.
    a. Add `requires basketball;` and `requires soccer;` to the `module-info.java` file for the `competition` module.

```
module competition {
    requires gameapi;
    requires soccer;
    requires basketball;
}
```

12. There may be another error in `Factory` class (depending on whether refactoring has been done successfully). There may be no `import` for `Team`, `Player` or `Game`. Originally, these were not necessary because they were in the same JAR. Now the API interfaces are in the `gameapi` module.
    a. If you need to, use the suggest icon to add imports for `gameapi.Team`, `gameapi.Player` and `gameapi.Game`.

    (You may need to do this for the `League` class too).

    There should be only one class remaining that has errors, the `Main` class.
13. Open the `Main` class and fix the logging issue.
    a. Fix these import problems by adding `requires java.logging;` to the `module-info.java` file for the `competition` module, then save the project.
14. Now, in the `Main` class, fix the issue with the error on importing the `displayDiagram` package (the problem is because the library `display-ascii-0.1b.jar` is not available).
    a. Select the Files tab of the IDE and create a `lib` folder in TeamGameManager (Right click on TeamGameManager and select New > Folder).
    b. Copy `display-ascii-0.1b.jar` from `League/lib` to `TeamGameManager/lib`
    c. On the projects tab, add `display-ascii-0.1b.jar` as an available JAR for the `competition` module (use Relative Path). (Right click on Libraries, and select Add JAR/Folder. Navigate to `~/labs/11_Migration/practices/TeamGameManager/lib`. Select `display_ascii-0.1b.jar`).
    d. Go to the `module-info.java` file for the `competition` module and add `requires display.ascii;` (the automatic module name) if it has not already been done (NetBeans usually does this automatically).

    The entire modular application TeamGameManager should now be free of errors.
15. Test TeamGameManager.
    a. Compile TeamGameManager (Right click on TeamGameManager and select Clean and Build). It should compile successfully. Now try running the application in the IDE. In the dialog, accept `main.Main` as the main class.

    You get an error, as there is a split package between the `soccer` and `basketball` modules. Both have a `util` package.

b. Use Refactor > Rename to refactor `util` in the soccer module to `soccer.util` and `util` in the `basketball` module to `basketball.util`.

c. Clean and build the application.

16. Try running the application in the IDE again. You will find there's one final error, but it is not related to the application itself. The error is that a needed file is not available. Use the Files tab to:

    a. Create a folder `data` in TeamGameManager.

    b. Copy `authors.txt` from `League/data` to `TeamGameManager/data`.

    c. Try running the application in the IDE again.

    The application should now work correctly.

17. Run the application on the command line.

    a. Clean and build the project using the IDE

    b. Open a terminal at `~/labs/11_Migration/practices/TeamGameManager`.

    c. Run the command:

    ```
    java -p dist:lib -m competition/main.Main
    ```

    The application should work correctly.

    Note: Remember that to add JARs in `<foldername>` to the class path, use `< foldername >/*`. To add JARs in `< foldername >` to the module path, use `< foldername >` (without the `*`).

Practice 11-4: Adding a Main Module

Overview

In this practice, you add another module named `main` so that the `competition` module represents the API of the application and the `main` module calls that API. You also look at the module graph for the application and see how transitive dependencies might improve the design.

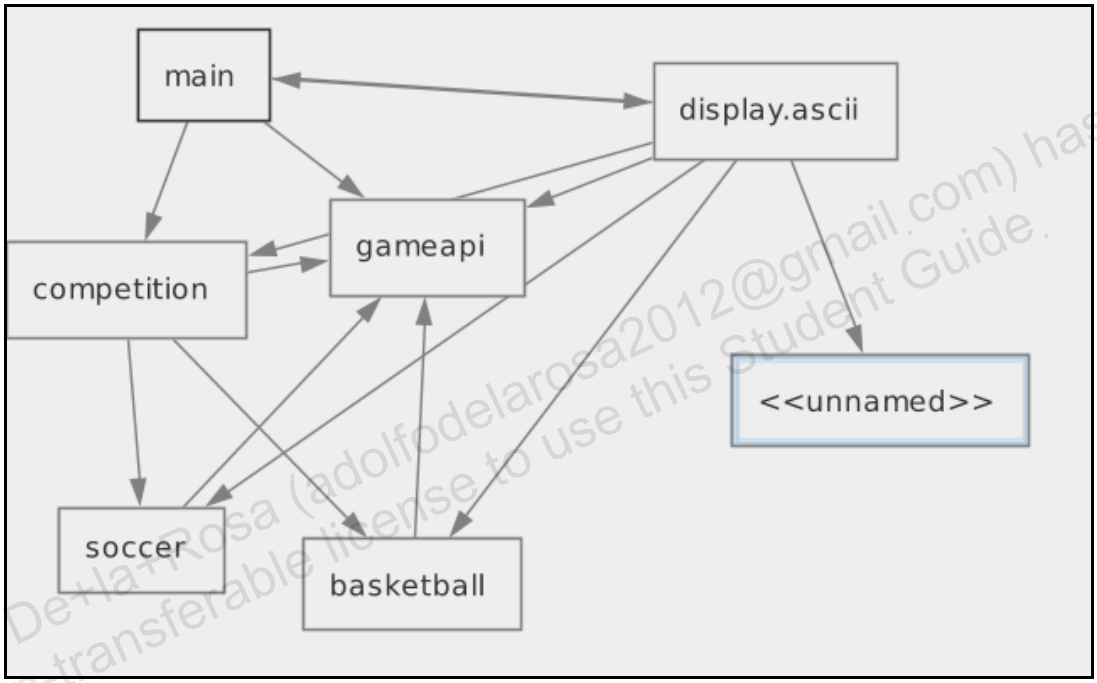
Assumptions

Tasks

1. Set up a new module, `main`, that calls the `competition` module.
   a. Create a new module and name it `main`.
   b. Move (by dragging) the `main` package from the `competition` module to the `main` module.
   c. Add the following dependencies to the `module-info.java` file for the `main` module.
      ```
      requires java.logging;
      requires competition;
      requires gameapi;
      requires display.ascii;
      ```
      You'll notice that even after adding these, some of the packages in the `competition` module are still not visible.
   d. In the `module-info.java` file for the `competition` module, add the following exports for the `game` and `utils` packages.
      ```
      exports game;
      exports utils;
      ```
      The application should now run.
   e. Remove the `requires java.logging;`, and the `requires display.ascii;` lines from the `module-info.java` file in the `competition` module. These are no longer needed.
2. Move the `PlayerDatabase*` classes to the `main` module. You'll do this by copying first, then deleting as NetBeans works better this way in this case.
   a. Create a new package in the `main` module named `database`.
   b. Copy the `PlayerDatabase` and `PlayerDatabaseException` classes from the `utils` package of the `competition` module to the `database` package.
   c. Delete the `PlayerDatabase` and `PlayerDatabaseException` classes from the `utils` package of the `competition` module.
   d. If necessary, change the imports to `utils.PlayerDatabaseException` and `utils.PlayerDatabaseException` in the `main.Main` class so that they now point to these same classes in the `database` package.

e. Clean and build the project. It should now have no errors and run correctly. It would make sense here to move some of the `utils.Settings` class to the `main` module, but don't worry about it for now.

3. Examine the module graph for the application.
   a. Open the `module-info.java` file for the `main` module and select Graph to view the module graph for the application.
   b. It will initially look confusedly complicated, but you can simplify it by omitting the JDK modules. Select Hide all JDK modules in the menu at the top right of the editor panel.
   c. You should now see something like this. You can move the modules around to get a clear picture.



Notice that `display.ascii`, being an automatic module, requires all other modules. Obviously, this is not ideal and later you modularize this library. Another interesting point about the depiction of the `display.ascii` module is that it (and only it) also requires the `unnamed` module. Only automatic modules do this. But the `display.ascii` module doesn't really depend on any other modules, and only the `main` module depends on it. These dependencies gained automatically are unnecessary and a good reason why automatic modules are an aid to migration but not intended to be a permanent solution.

The module that has most dependencies from other modules is the `gameapi` module. This makes sense, as it is the API. Is the current setup best though?

For example, does the `main` module have any reason to access the `gameapi` module directly? It would make more sense that the `competition` module depends on the `gameapi` module transitively as the `main` module only requires the `gameapi` module as part of the API offered by the `competition` module.

d. Change the `competition` module's dependence on the `gameapi` module to `transitive` dependence, and remove the `main` module's direct dependence on the `gameapi` module. The `module-info.java` file for `competition` will now look like this.

```
module competition {
    requires transitive gameapi;
    requires soccer;
    requires basketball;

    exports game;
    exports utils;
}
```

e. View the graph again; the `main` module should no longer depend on the `gameapi` module directly.

f. Another small improvement would be to limit the `soccer` and `basketball` modules to export only to the `competition` module. Make this change to the `module-info.java` files for these modules. For example, in the `module-info.java file` for the `soccer` module, you put the following. Note the “to competition” part of the exports line.

```
module soccer {
    requires gameapi;
    requires java.logging;
    exports soccer to competition;
}
```

Make a similar change to the `module-info.java` file of the `basketball` module.

Practice 11-5: Migrating a Library

Overview

In this practice, you migrate the library `display-ascii-0.1b.jar` to a modular library. You use the same name for the modular version of this library as was chosen automatically by the system when the JAR was loaded as an automatic module, that is, `display.ascii` (its automatic name was based on the name of the JAR).

In previous practices, you treated `display-ascii-0.1b.jar` as a library JAR, but in this practice assume that you have access to this JAR and can modularize it. This may be true for some external libraries if they are open source like Jackson, but it is not usually a good idea to modularize an open source library. Instead, you should contact the maintainer of the library and find out when a modularized version may become available.

In this practice you work with a very simple library, `display-ascii-0.1b.jar`.

Assumptions

Tasks

1. In NetBeans, navigate to `~/labs/11_Migration/practices` and open the project display-ascii-0.1b. This was created in Java SE 8.
   a. Create a new modular project named `display.ascii`.
   b. Create a new module in this project also named `display.ascii`.
   c. Copy the two packages from `display-ascii-0.1b` to the `display.ascii` module. There should be no errors.

      Now you need to create the `module-info.java` file for the `display.ascii` module. As it's a very simple module, it's obvious that exporting the `displayDiagram` package is all that is necessary.

      However, if you were to look in the `module-info.java` files you generated earlier using jdeps (in `TeamGameManager/generated-module-info`) you'd see that `jdeps` suggested:

```
display.ascii {
    exports displayDiagram;
    exports test;
}
```

      In this case, `jdeps` suggests exporting a package that you know to be unnecessary, so you can omit exporting the test package. This is quite common when you use `jdeps` to suggest a `module-info.java` file – it will err on the side of ensuring that all packages that *may* be needed are exported. You may therefore find you can omit some that you know to be unnecessary.

d. Add the necessary exports to the `module-info.java` file for the `display.ascii` module.

e. Build this module.

2. Replace the unmodularized `display-ascii-0.1b.jar` with `display.ascii.jar` for the `main` module in TeamGameManager.

   a. Use the files tab to copy `display-ascii.jar` (from the `dist` folder of the display.ascii project) to the `lib` folder of the TeamGameManager project.

   b. In the Projects tab, right click Libraries > Properties in the `main` module of the TeamGameManager project.

   c. Remove the JAR file `display-ascii-0.1b.jar`.

   d. Click the … icon for Modulepath, then click Add JAR/folder, and navigate the `lib` folder of the TeamGameManager project.

   e. Select `display.ascii.jar` and click OK.

   f. Test TeamGameManager. It should work correctly.

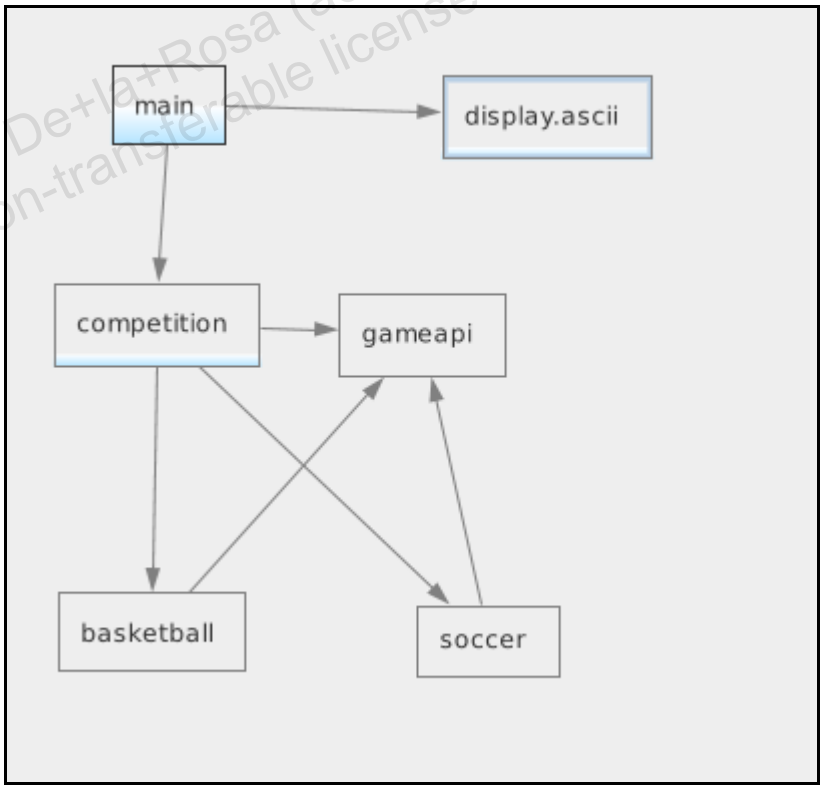
   You now have a completely modularized application.

3. View the module graph for TeamGameManager in NetBeans. You may need to do the following before the change is visible in the graphing display.

   a. Shut down NetBeans.

   b. Restart NetBeans, and open the `module-info.java` file for the main module.

   c. Set the JDK scope to Hide all JDK modules



Notice how this is very different from the previous module graph where `display.ascii` was an automatic module. Now the `display.ascii` module has no dependencies on any other modules and the entire application is much simpler and more secure.

You can also clearly see how having `gameapi` as a separate module ensures that there are no cycles in the module graph. However, for this application there is a better option for breaking cyclic dependencies. In the next lesson on services you explore this option.

Practice 11-6: Bottom-up Migration

Overview

In this practice, you look at how bottom-up migration differs from top-down migration. Bottom-up migration is possible by being able to use the class path and the module path together in a Java SE 9 application.

You have completed the migration from the top down, starting by modularizing the application itself, and eventually modularizing the display.ascii library.

However, given that the class path and the module path can both be used to run an application, it is possible to work in the other direction. For example, you could first modularize `display-ascii-0.1b.jar` and run it on the module path while running `League.jar`, `Soccer.jar`, and `Basketball.jar` on the class path. Then, you could continue the migration process by modularizing `Basketball.jar`, `Soccer.jar`, and finally `League.jar`.

As you have already done all the modularization of these JAR files, it's not necessary to modularize them again. You can see how this bottom-up migration would work by simply running the application with some of its classes on the class path and some on the module path.

You could do this in the IDE, by specifying which modules to place on the module path, and which JARs to place on the class path. But it's much easier to use the command line to explore different options, and that's what you do in this practice.

Assumptions

Tasks

All of the commands given below are available as scripts in `~/labs/11_Migration/practices/League`. They are `ex1`, `ex2`, `ex3`, `ex4`, `ex5`, and `ex6`, and refer in turn to each of the commands given in this practice. The scripts save some typing, but do try to understand how they work and feel free to experiment with them. You run the scripts by entering them on the command line. For example, to run the first command listed below enter the following at the command line.

```
./ex1
```

1. Run the application, using the class path for everything except the modularized library, `display.ascii`. You do this by using:

   `-cp` to specify that some JARs are loaded onto the classpath

   `-p` to specify that the `display.ascii.jar` modular JAR is loaded onto the module path.

   a. Navigate to the League directory in practices.

   b. Try this command (it will fail but in an instructive manner). Note that `Basketball.jar` and `Soccer.jar` are explicitly placed on the class path rather than using an asterisk. This is so that the `display-ascii-0.1b.jar` is not loaded. Otherwise, even if the modularized `display.ascii` module did not load, `display-ascii-0.1b.jar` would load, and the application would still work.

Here is the command. (This is script `ex1`).

```
java -cp dist/*:lib/Basketball.jar:lib/Soccer.jar -p
../display.ascii/dist/* main.Main


Exception in thread "main" java.lang.NoClassDefFoundError:
[[LdisplayDiagram/DisplayDetail;
    at main.Main.getDataGrid(Main.java:87)
        at main.Main.main(Main.java:74)
        Caused by: java.lang.ClassNotFoundException:
displayDiagram.DisplayDetail
        …
```

What is happening here? Remember that here we are attempting to run all the code except the `display.ascii` module on the class path. In other words, everything except `display.ascii` runs as an unnamed module. Normally the process of resolving modules is to follow the requirements until all modules have been loaded. But the unnamed module doesn't have any explicit requirements, instead it implicitly requires all other modules. You need to add `display.ascii` module so that it will be loaded.

c. Try this command. It should work.

(This is script `ex2`).

```
java -cp dist/*:lib/Basketball.jar:lib/Soccer.jar -p
../display.ascii/dist --add-modules display.ascii main.Main
```

2. Try running some other commands that combine the use of the class path and the module path.

a. Try this command. Can you see how it works?

(This is script `ex3`).

```
java -cp dist/*:lib/Basketball.jar:lib/Soccer.jar -p
lib/display-ascii-0.1b.jar --add-modules display.ascii main.Main
```

Instead of putting the modularized JAR `display.ascii.jar` on the module path, you've put `display-ascii-0.1b.jar` on the class path! This causes it to become an automatic module named `display.ascii` that is then loaded using `--add-modules` to add its automatic name.

You can see more of what is happening by adding `--show-module-resolution` to the command. But you'll need to limit the modules, otherwise you will see a very long list of platform modules.

b. For example, try the following command (for clarity the command and its output are in separate code boxes.

(This is script `ex4`).

```
java --limit-modules java.base,java.logging --show-module-
resolution -cp dist/*:lib/Basketball.jar:lib/Soccer.jar -p
lib/display-ascii-0.1b.jar --add-modules display.ascii main.Main
```

```
root display.ascii
file:///home/oracle/labs/11_Migration/practices/League/lib/displ
ay-ascii-0.1b.jar automatic
root java.logging jrt:/java.logging
root java.base jrt:/java.base
java.base binds java.logging jrt:/java.logging
```

c. Try this command.

(This is script `ex5`).

```
java --limit-modules java.base,java.logging --show-module-
resolution -cp dist/*:lib/Basketball.jar:lib/Soccer.jar -p
../display.ascii/dist --add-modules display.ascii main.Main
```

```
root display.ascii
file:///home/oracle/labs/11_Migration/practices/League/../displa
y.ascii/dist/display.ascii.jar
root java.logging jrt:/java.logging
root java.base jrt:/java.base
java.base binds java.logging jrt:/java.logging
```

The resolution for these two commands is very similar, and in both cases the resolution of `display.ascii` shows the file it came from. However, in the first instance, where that file is an unmodularized JAR file, it is marked as an automatic module.

Could you add `Soccer.jar` or `Basketball.jar` as named modules? Think about that for a moment.

This can't be done because of the dependency where classes in modules `soccer` and `basketball` implement interfaces in the `gameapi` module. These are not the interfaces used in the League project (in `League.jar`). The `competition` module must be combined with the `soccer` and `basketball` modules and the `League.jar` must be combined with `Soccer.jar` and `Basketball.jar`.

The following is possible.

`League.jar` - loaded on the class path, therefore an unnamed module

`Basketball.jar` - loaded on the class path, therefore an unnamed module

`Soccer.jar` - loaded on the module path therefore an automatic module named `Soccer`.

`display.ascii` - a named module loaded on the module path

d. Now try this command:

(This is script ex6).

```
java --limit-modules java.base,java.logging --show-module-
resolution -cp dist/League.jar:lib/Basketball.jar -p
../display.ascii/dist/:lib/Soccer.jar --add-modules
display.ascii,Soccer  main.Main
```

Notice how `–add-modules` needs to be used with the automatic module `Soccer` now as well as the `display.ascii` module.

This last example is not a very realistic way to run the application, but it does illustrate the how the class path and module path can be combined to run a named module, an automatic module and unnamed modules in the same application.

Practice 11-7: Adding the Jackson Library

Overview

In this practice, you add another library to the application, this time a commonly used library used for JSON support called Jackson.

You put the JSON support code in its own module, as it may be possible in future that the application may support several types of storage. For example: SQL database, XML files, JSON files.

Tasks

1. Create a new module in TeamGameManager called `storage` to handle storing and retrieving games.

   The code to interact with JSON to store all the games in a competition has already been written for you.

   a. In the file browser, or from a command line, copy the folder storage in `~/labs/11_Migration/practices` to the classes directory in `~/lab/11_Migration/practices/TeamGameManager/src/storage`

   From the command line the command is:

   ```
   cp -r ~/labs/11_Migration/practices/storage
   ~/labs/11_Migration/practices/TeamGameManager/src/storage/classes/
   ```

   This will add a package `storage` to the `storage` module.

   b. Now copy the Jackson library classes from `~/labs/11_migration/practices` to the `lib` directory of the TeamGameManager application.

   The command line command is:

   ```
   cp -r ~/labs/11_Migration/practices/jackson*
   ~/labs/11_Migration/practices/TeamGameManager/lib
   ```

   c. Right click on Libraries in the `storage` module and click Add JAR/Folder

   d. In the dialog, select the three Jackson libraries you just added to the lib folder (`labs/11_Migration/practices/TeamGameManager/lib`) and click Open (Use Relative path).

2. Add the remaining dependencies to the `module-info.java` file to clear up the remaining errors.

   a. As the JSON code will be working to store an array of Game types, add

   ```
   requires gameapi;
   ```

   to the `module-info.java` file for storage. This should clear any errors (the Jackson library JARs should already have been added to the `module-info.java` file by NetBeans.

   b. As the `main` module will need access to the classes in the `storage` package add

   ```
   exports storage;
   ```

   to the `module-info.java` file for the `storage` module.

3. Now add the commands to the `Main` class to use the `storage` module to store games as a JSON file.
   a. First, add a dependence on the `storage` module to the `module-info.java` file of the `main` module. (The necessary line is `requires storage;`)
   b. Open the `Main` class and find the point where the `createAndPlayAllGames` method has just been called on the `League` object (`theLeagueCompetition`).
   c. Add a line below this with the necessary code to save all the games as JSON data.

```
storage.JacksonUtil.saveToJSONFile( dirName,
"leagueSoccer01.json", theLeagueCompetition.getGames());
```

   You'll get an error for `dirName`, so use the suggestion to import `utils.Settings.dirName` (this is set to `data`).

4. Try running the application.

   The application should run and produce the league results, but you will see an error that states:

   "Unable to make field private gameapi.Team soccer.Soccer.homeTeam accessible: module soccer does not "opens soccer" to module jackson.databind (through reference chain: soccer.Soccer[0])"

   This is because the `jackson.databind` module is using reflection to examine a field of the `Soccer` class. The field is private so `open` is required to allow this to happen. If the field were public, then `exports` would work.

5. Modify the `module-info.java` file for the `soccer` module so that it permits reflection on the `soccer` package.
   a. Open the `module-info.java` file for the module `soccer` and add the line:

```
opens soccer to jackson.databind;
```

   b. Now try running the application again. It should run correctly.
   c. Look in `~/labs/11_Migration/practices/TeamGameManager/data`. You should see that the `leagueSoccer01.json` file has been added. Look at it in a text editor if you like. You could also use the linux commands `less` or `more` to look at the file.
   d. Make the same change to the `basketball` module by adding a line to the basketball `module-info.java` file.

```
opens basketball to jackson.databind;
```

6. Check that loading games works by loading a set of games and using those games to instantiate a `League` object.
   a. Add these lines just below the line you just added in the `Main` class.

```
Game[] newGames = storage.JacksonUtil.getGamesFromJSONFile(
dirName, "leagueSoccer01.json");
League retrievedLeagueCompetition = new League(newGames);
```

b. In order that a new `League` object can be instantiated using a `Game` array, add the following constructor to the `League` class in the `game` package of the `competition` module.

```
public League(Game[] allGames) {
    this.games = allGames;
    this.teams = getTeamsFromGames(allGames);
}
```

c. In the `Main` class, change the reference in the `getDataGrid` method call to use this new `League` object. The line will now look like this:

```
DisplayDetail[][] dataGrid =
getDataGrid(retrievedLeagueCompetition.getGames(),
retrievedLeagueCompetition.getTeams());
```

Make sure to use `retreivedLeagueCompetition` in both references in the method call.

You could of course use the `Game` array `newGames` as the first parameter in this method, but you also need to pass in the `Team` array, and for that you need to create a new `League` as its constructor will derive the `Team` array from the `Game` array passed in to its constructor.

d. Try running the application a few times. Notice that the results are different every time.

e. Now comment out the line that saves the JSON file and try running the application again. You should see that it returns the same results every time.

f. Uncomment the line that saves the JSON file, so that the results once again change each time. (This is necessary in preparation for the practices in the next lesson).

That's all the practices for Lesson 11. There's a lot of material here, but you've covered many issues that are important considerations in migrating an application from SE 8 or earlier to a modular application.

Practices for Lesson 12: Services

Practices for Lesson 12: Overview

Overview

In these practices, you will migrate soccer and basketball so they become services. This makes the application more easily extensible and more robust.

Practice 12-1: Creating Services

Overview

In this practice, you will modify the TeamGameManager application so that the basketball and soccer module provide services. This is a superior approach to that currently taken and allows the addition of further team games without recompiling TeamGameManager (except for the Main class which most likely would be created in its own project).

The first thing to consider is how to acquire the necessary Java objects to create a league of soccer games or basketball games.

In the TeamGameManager application, the creator of a new implementation of a `Game` type can also create unique implementations of `Team` and `Player` to work with that `Game` type. These new `Team` and `Player` types must implement `Team` and `Player` but they could have other methods that available to the `Game` type object, or, after casting, to the class that sets up and runs the league, in this case, the `Main` class. For this reason, the `Factory` method for creating, say, a `Soccer` object, either needs to provide functionality to also create `Player` and `Team` objects that are compatible with that `Soccer` object, or needs to provide a proxy or factory object to do this.

In this practice, you use the latter approach. It is simpler as you are only creating one object, which in turn creates the `Soccer`, `Player`, and `Team` objects. It also allows you to call the normal constructors on `Soccer`, `Team` and `Game`, rather than having to call a no-argument constructor and then provide a method for populating the object in a second step.

Assumptions

You completed practice 11, or you have opened the solution to practice 11 and plan to work with that. If you use the solution to practice 11 as your starting point:

1. Load the project in NetBeans (you'll find it at `/home/oracle/labs/11_Migration/solutions/Sol_11_07_TeamGameManager`.)
2. Right click on the solution project and click Copy.
3. Change the name by removing the "`Sol_`" part of the current name.
4. Change the Location to: `/home/oracle/labs/11_Migration/practices`

Tasks

1. Create the interface for the service.
   a. Create an interface in the `gameapi` package of the `gameapi` module that will be implemented in both `soccer` and `basketball` modules (and in subsequent game implementations). Call the interface `GameProvider` and add methods to create a `Game`, `Team`, or `Player`. The method signature for each should match the method signature of the constructor in each class.

b.

```
public interface GameProvider {

    Game getGame(Team homeTeam, Team awayTeam, LocalDateTime
plusDays);
    Player getPlayer(String playerName);
    Team getTeam(String teamName, Player[] players);

}
```

c. Add an import for `java.time.LocalDateTime` to clear the error.

d. Add an abstract method to the `GameProvider` interface to identify the game being provided by the service. The code in the `Factory` class will look at this value to determine if this is the game to create.

`String getType();`

This will be used to return "soccer" if the provider is for soccer, and "basketball" if the provider is for basketball.

e. Create an implementation of this interface in the `soccer` package of the `soccer` module. Call this class `SoccerProvider`. For each of the methods, call the constructor on the class that is to be returned, and return the object created. For the `getType` method, return the `String` "`soccer`". You'll need to add imports for the various types used.

```
public class SoccerProvider implements GameProvider {

    public Game getGame(Team homeTeam, Team awayTeam,
LocalDateTime plusDays) {
        return new Soccer(homeTeam, awayTeam, plusDays);
    }
    public Player getPlayer(String playerName){
        return new SoccerPlayer(playerName);
    }
    public Team getTeam(String teamName, Player[] players){
        return new SoccerTeam(teamName, players);
    }
    public String getType() {
        return "soccer";
    }
}
```

f. Create a similar class named `BasketballProvider` in the basketball package of basketball to provide the various necessary objects for basketball.

2. Modify the `Factory` class to use services instead of instantiating objects by using the keyword `new` as it currently does.

   a. Open the `Factory` class in the `game` package in the `competition` module. You will recode this class to provide the service object `GameProvider`, and then use that object to provide `Game`, `Player`, and `Team` objects as needed.

   b. Create a new method in the `Factory` class called `getProvider`. Start with a simple version of this method. You will improve it later. As you add functionality to the method, you'll need to import the types you're using.

```
public static GameProvider getProvider(String gameType) {
}
```

   c. Within the `getProvider` method, create a variable for the `GameProvider`.

```
GameProvider theProvider = null;
```

   d. Use the `load` method of the `ServiceLoader` class to get the `GameProvider`.

```
ServiceLoader<GameProvider> loader =
ServiceLoader.load(GameProvider.class);
```

   e. Write a loop to iterate through the loader

```
for (GameProvider currProvider: loader){

}
```

   f. Within the for loop, check what provider you're currently referencing in `currProvider`. When it's the one you want, that is when `gameType`, the type you're requesting, is the same as the type referenced in `currProvider`, break out of the loop so you don't continue searching.

```
if (currProvider.getType().equalsIgnoreCase(gameType)){
theProvider = currProvider;
break;

}
```

   g. Below the for block, add code to return the the `GameProvider` you found.

```
return theProvider;
```

   You'll use this provider in the `getTeam`, `getPlayer`, and `getGame` methods to load the appropriate types.

The code written so far will work but there are some issues you need to address later. (For example, what happens if no suitable service is found?).

3. Code the `createTeam` method.
   a. At the top of the `createTeam` method add a line to load a `GameProvider` using the method you just created.

```
GameProvider theProvider = getProvider(gameType);
```

   b. Call the `getTeam` method on the `GameProvider` object, and return the `Team` thus created.

```
return theProvider.getTeam(teamName.trim(), thePlayers);
```

   c. Comment out (or delete) all of the switch block in this method.
   d. Update the `createGame` method in the same way.

```
GameProvider theProvider = getProvider(gameType);

return theProvider.getGame(homeTeam, awayTeam, dateOfGame);
```

   Remember to comment out (or delete) the switch block.
   e. Update the `createPlayer` method in the same way.

```
        GameProvider theProvider = getProvider(gameType);

        return theProvider.getPlayer(playerName);
```

4. Add `uses` and `provides` statements in the appropriate `module-info.java` files; a `uses` statement in the `module-info.java` file in the `competition` module, and a `provides` statement in the `module-info.java` file in the `soccer` and `basketball` modules.
   a. Add the `uses` statement to the `module-info.java` file in the `competition` module.

```
uses gameapi.GameProvider;
```

b. Add the `provides` statements to the `module-info.java` file in the `soccer` and `basketball` modules You can also remove the `exports soccer;` line as that is no longer necessary. Here's the appropriate line for soccer; use a similar approach for basketball.

```
provides gameapi.GameProvider with soccer.SoccerProvider;
```

c. That should be enough to get things working. No longer is the instantiation of the types hard-coded, instead, you are using a service to load an object that can provide them. When a new game is added, the code in the `getProvider` method will return its `GameProvider` when requested.

d. You should have no errors at this point if you have been adding import statements for new classes used in the code. If there are errors at this point, check to make sure you have added all necessary imports.

e. Try running the application. It should work correctly and is now using services!

You're not quite finished. If new games can be added to the application at any time without recompiling, that means that no modules in the application should be dependent on any of the game services. Previously the `competition` module had dependencies on `soccer` and `basketball` modules in order to instantiate the types in them. These dependencies can now go!

5. Remove any unnecessary dependencies.

a. Remove the following statements from the `module-info.java` file of the `competition` module. They are no longer required as soccer and basketball are now implemented as services.

```
requires soccer;
requires basketball;
```

b. This will cause errors in Factory. Remove the imports that are showing an error, as they are no longer needed.

c. Run the application again and try changing from soccer to basketball. It should work. If it doesn't, make sure you're using a newly created set of games and not a stored set of games.

The `Factory` class currently works but it is not ideal. For every call to `createTeam`, `createGame`, or `createPlayer`, a new `GameProvider` object is being created.

6. Determine how many times `GameProvider` is being created.

- In the Factory class, just inside the loop that iterates through the available `GameProvider` services, add a line that prints the currProvider to the console.

```
System.out.println(currProvider.getClass() + " : " +
currProvider.hashCode());
```

a. Run the application with "soccer" as the gameType.

You will see a lot of output showing that a new SoccerProvider is provided for every call. Also, each time, the basketball provider is looked at first - a lot of unnecessary work.

In this application this still doesn't take much time, but it could be improved. An obvious approach would be to use a singleton of some kind. For this practice, you take an even simpler approach by making GameProvider a static field. Then, as long as a GameProvider exists and it's of the correct type, you can just return it without calling the ServiceLoader at all.

7. Add code to ensure GameProvider is acquired when needed.

a. Declare a static GameProvider field. Use the same name, theProvider, as before.

```
static GameProvider theProvider;
```

b. Delete the declaration of GameProvider within the getProvider method.

c. In the getProvider method, but before the call to the ServiceLoader.load method, add a line that tests if the correct GameProvider already exists.

```
if ((theProvider!=null) &&
    theProvider.getType().equals(gameType)) {
    return theProvider;
}
```

d. In each of the three methods for returning Game, Team, and Player objects, remove the type GameProvider so that theProvider refers to the static GameProvider. The first line in each of the methods will look like this.

```
theProvider = getProvider(gameType);
```

e. Run the code again, you should see a SoccerProvider object is acquired only once.

8. Address what happens if there is no suitable service.

a. Change the gameType again, this time to a non-existent service, neither "soccer" or "basketball".

```
String gameType = "hockey"; // Or anything that is not "soccer"
    or "basketball"
```

b. Run the application.

You will get a NullPointerException. You need to add some code to check if any valid GameProducer has been found.

c. In the `getProvider` method of the `Factory` class, add some code under the for loop to check if `theProvider` is still `null` after all available services have been examined.

```
if (theProvider == null) {
    throw new RuntimeException("No suitable service provider
        found!");
}
```

d. Now run the application again. This time the `Exception` will give the more useful information that no suitable service provider has been found. This `Exception` could be a custom one and could be handled more elegantly, but here we just want to show that the code is capable of discovering that there is no suitable service.

e. Set `gameType` back to `"soccer"` so the application works correctly.

That's it for this practice. The code in `getProvider` could be tidied up a little, but it is doing what is necessary, providing a `GameProvider` of the appropriate type.

Practice 12-2: More Services

Overview

In this practice, you examine how services could be used for other aspects of the application's functionality. For example, in the TeamGameManager application, services could be used to provide additional:

- Competition types – for example, knockout competitions.
- Display types – for example, a graphical display.
- Storage types – for example, storage to a database.

In this practice, you look at how TeamGameManager could be extended to provide another type of competition. As you've already coded and configured services for the type of game to be managed, this time you won't do any coding. Instead you'll examine a completed application to see how the service implementation differs from the one you just created.

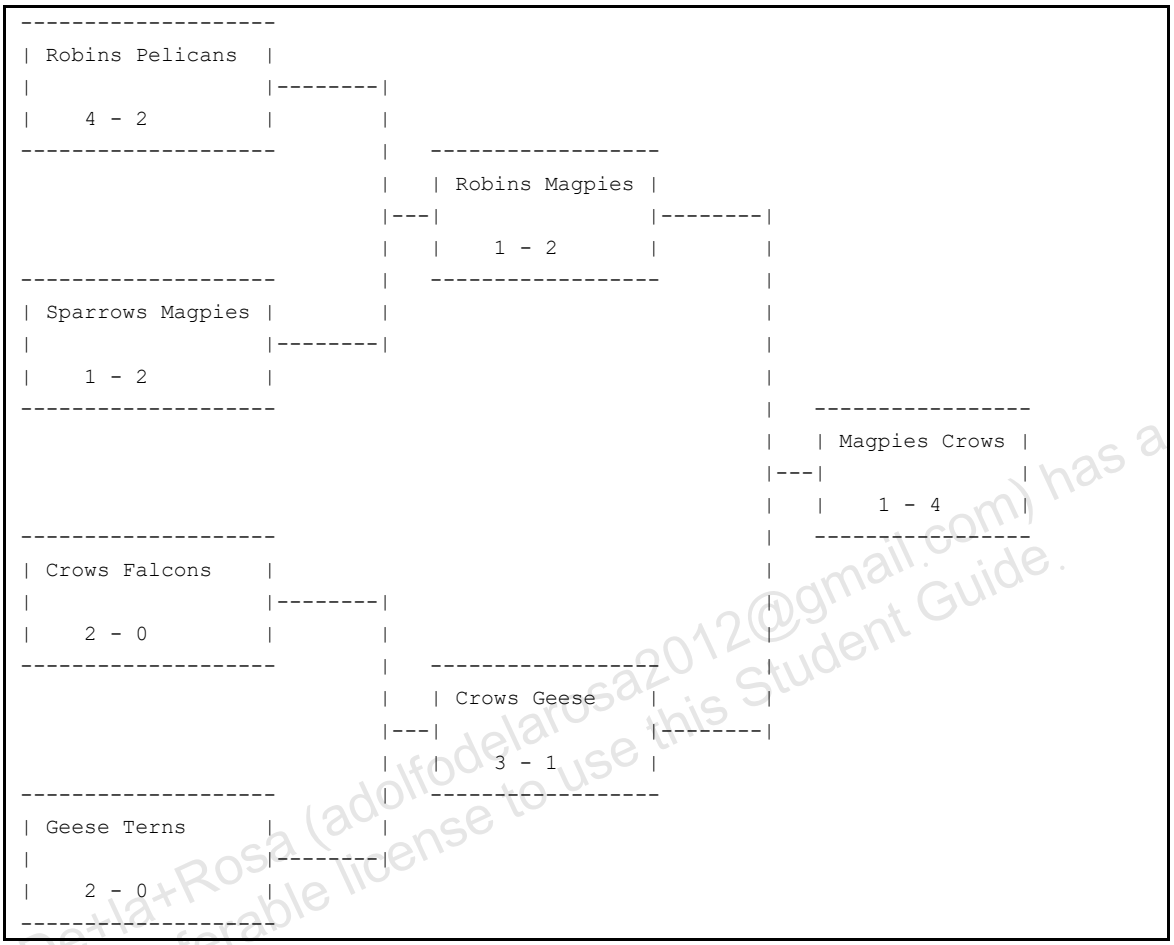
Assumptions

Tasks

1. Open the application Sol_12_02_TeamGameManager that is in the folder `~/labs/12_Services/solutions`.
2. Examine the functionality in the Main class.
   a. Open the `Main` class in the `main` package of the `main` module.
   b. Look at the beginning of the `main` method. The two variables, `competitionType` and `gameType`, determine the type of competition and the type of game. The variable `gameType` you are familiar with, but `competitionType` is new; it is used to select the type of competition that will be used.

```
public static void main(String[] args) {
    String gameType = "soccer"; // "soccer" or "basketball"
    String competitionType = "knockout"; // "knockout" or
                                         // "league"
```

c. Try running the application. You should see output for a knockout style competition for the game of soccer.



d. Try changing the competition type and/or the game type. Run the application again. It should respond with the appropriate results.

e. Scroll down the main method. Notice the line that creates the competition.

```
TournamentType theCompetition =
    TournamentFactory.getTournament(competitionType, gameType,
        theTeams);
```

Unlike previously, the type is not `League`, rather it is `TournamentType`, and `TournamentFactory` is used to create the competition based the type of the competition requested, the type of the game requested, and the teams.

3. Examine the functionality in the `TourrnamentFactory` class.

a. Open the `TournamentFactory` class in the `game` package of the `competition` module.

b. Notice that it is simpler than the `Factory` class used to get the `GameProvider` object to create the needed objects for a particular game. For a `TournamentType`, the API is very simple. Only one class is required so the class implementing `TournamentType` is

not a proxy, as with `GameProvider`, but the actual class needed to create the necessary functionality.

c. Notice that in the `getTournament` method with the method signature `String, String, Team[]`, the `TournamentType theTourney` has its `populate` method called to populate it with the teams needed and set the type of game (shown bolded below).

```
public static TournamentType getTournament(String tourneyType,
    String gameType, Team[] theTeams) {

        TournamentType theTourney = getTournament(tourneyType);
        theTourney.populate(gameType, theTeams);

        return theTourney;

    }
```

4. Examine the functionality of the `Knockout` class.
   a. Open the `Knockout` class.
   b. Notice that unlike the three classes that implemented a game (`Game`, `Team`, and `Player`), `Knockout`, as it is returned directly by the `ServiceLoader` class, must have a no argument constructor. That's why there is also a method, `populate`, to populate a newly created `Knockout` object.
5. Examine the `module-info.java` file for the `competition` module.
   a. Open the `module-info.java` file for the `competition` module
   b. Look at the final two lines of the file. First, it specifies that this module `uses` classes that implement `GameProvider`, and `TournamentType`. However, it also specifies that this module also `provides` two implementations of `TournamentType`.

      This is an unusual construction. In some cases, as here, the module that "uses" a type may also provide that type. This can be useful to provide a default implementation. However, in this application the implementations of `TournamentType` should really be in their own modules. That way, a new implementation can be added to the application at any time.

6. If you would like to, you can now go back to the applications in `~/10_IntroducingModules/examples`.
   a. Look at TeamGameManager application. You'll see that it is a further development of the TeamGameManager application here in practice 7 and has two improvements:
      - The competition implementations are now in their own modules.
      - The `gameapi` module is gone and its interfaces are now in the competition module. It is no longer needed as a separate module to solve the split package problem now that the application is using services.

This is the end of this practice.

Practices for Lesson 13: Concurrency

Practices for Lesson 13: Overview

Practices Overview

In these practices, you will use the `java.util.concurrent` package and sub-packages of the Java programming language.

Practice 13-1: Summary Level: Using the `java.util.concurrent` Package

Overview

In this practice, you will modify an existing project to use an `ExecutorService` from the `java.util.concurrent` package.

Assumptions

You have reviewed the sections covering the use of the `java.util.concurrent package`.

Summary

You will create a simple multithreaded counting application. Instead of manually creating threads, you will leverage an `ExecutorService` from the `java.util.concurrent` package.

Tasks

1. Open the `ConCount13-01Prac` project as the main project.
   a. Select File > Open Project.
   b. Browse to /home/oracle/labs/13-Concurrency/practices/practice1.
   c. Select ConCount13-01Prac and click the Open Project button.
2. Expand the project directories.
3. Open the `CountRunnable` class in the `com.example` package.
4. Create a constructor to initialize the `count` and `threadName` variables.
5. Uncomment the `count` and `threadName` variables.
6. In the run method, set up a `for` loop to print out the thread name and each number counted.
7. Open the `Main` class in the `com.example` package.
8. Set up the `ExecutorService` in the main method using the `Executors` class and the `newCachedThreadPool` method.
9. Setup three `CountRunnable` objects to count to 20, named threads A, B, and C.
10. Shut down the `ExecutorService`.
11. Run the project. You should see each thread count to 20. Because of out of order processing, the counts of the three threads should be all jumbled together.

Practice 13-1: Detailed Level: Using the `java.util.concurrent` Package

Overview

In this practice, you will modify an existing project to use an `ExecutorService` from the `java.util.concurrent` package.

Assumptions

You have reviewed the sections covering the use of the `java.util.concurrent package`.

Summary

You will create a simple multithreaded counting application. Instead of manually creating threads, you will leverage an `ExecutorService` from the `java.util.concurrent` package.

Tasks

1. Open the `ConCount13-01Prac` project as the main project.
   a. Select File > Open Project.
   b. Browse to /home/oracle/labs/13-Concurrency/practices/practice1.
   c. Select ConCount13-01Prac and click the Open Project button.
2. Expand the project directories.
3. Open the `CountRunnable` class in the `com.example` package.
4. Create a constructor to initialize the `count` and `threadName` variables.

```
public CountRunnable(int count, String name){
  this.count = count;
  this.threadName = name;
}
```

5. Uncomment the `count` and `threadName` variables.

```
final int count;
final String threadName;
```

6. In the run method, set up a `for` loop to print out the thread name and each number counted.

```
  for (int i = 1; i <= count; i++){
    System.out.println("Thread " + threadName +
      ": " + i);
  }
```

7. Open the `Main` class in the `com.example` package.
8. Set up the `ExecutorService` in the main method using the `Executors` class and the `newCachedThreadPool` method.

```
  ExecutorService es = Executors.newCachedThreadPool();
```

9. Setup three `CountRunnable` objects to count to 20, named threads A, B, and C.

```
    es.submit(new CountRunnable(20,"A"));
    es.submit(new CountRunnable(20,"B"));
    es.submit(new CountRunnable(20,"C"));
```

10. Shut down the `ExecutorService`.

```
    es.shutdown();
```

11. Run the project. You should see each thread count to 20. Because of out of order processing, the counts of the three threads should be all jumbled together.

Practice 13-2: Summary Level: Creating a Network Client using the `java.util.concurrent` Package

Overview

In this practice, you will modify an existing project to use an `ExecutorService` from the `java.util.concurrent` package.

Assumptions

You have reviewed the sections covering the use of the `java.util.concurrent` package.

Summary

You will create a multithread networking client that will rapidly read the price of a shirt from several different servers. Instead of manually creating threads, you will leverage an `ExecutorService` from the `java.util.concurrent` package.

Tasks

1. Open the `ExecutorService13-02Prac` project as the main project.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/13-Concurrency/practices/practice2`.
   c. Select `ExecutorService13-02Prac` and click the Open Project button.
2. Expand the project directories.
3. Run the `NetworkServerMain` class in the `com.example.server` package by right-clicking the class and selecting Run File.
4. Open the `NetworkClientMain` class in the `com.example.client` package.
5. Run the `NetworkClientMain` class package by right-clicking the class and selecting Run File. Notice the amount of time it takes to query all the servers sequentially.
6. Create a `NetworkClientCallable` class in the `com.example.client` package.
   a. Add a constructor and a field to receive and store a `RequestResponse` reference.
   b. Implement the `Callable` interface with a generic type of `RequestResponse`.

```
public class NetworkClientCallable implements
Callable<RequestResponse>
```

   c. Complete the `call` method by using a `java.net.Socket` and a `java.util.Scanner` to read the response from the server. Store the result in the `RequestResponse` object and return it.

      Note: You may want to use a `try-with-resource` statement to ensure that the `Socket` and `Scanner` objects are closed.
7. Modify the `main` method of the `NetworkClientMain` class to query the servers concurrently by using an `ExecutorService`.
   a. Comment out the contents of the `main` method.

b. Obtain an `ExecutorService` that reuses a pool of cached threads.

c. Create a Map that will be used to tie a request to a future response.

```
Map<RequestResponse, Future<RequestResponse>> callables = new
HashMap<>();
```

d. Code a loop that will create a `NetworkClientCallable` instance for each network request.

e. The servers should be running on localhost, ports 10000–10009.

f. Submit each `NetworkClientCallable` to the ExecutorService. Store each `Future` in the `Map` created in step 7c.

g. Shut down the `ExecutorService`.

h. Await the termination of all threads within the `ExecutorService` for 5 seconds.

i. Loop through the `Future` objects stored in the `Map` created in step 7c. Print out the servers' response or an error message with the server details if there was a problem communicating with a server.

8. Run the `NetworkClientMain` class by right-clicking the class and selecting Run File. Notice the amount of time it takes to query all the servers concurrently.

9. When done testing your client, be sure to select the ExecutorService output tab and terminate the server application.

Practice 13-2: Detailed Level: Creating a Network Client using the `java.util.concurrent` Package

Overview

In this practice, you will modify an existing project to use an `ExecutorService` from the `java.util.concurrent` package.

Assumptions

You have reviewed the sections covering the use of the `java.util.concurrent` package.

Summary

You will create a multithread networking client that will rapidly read the price of a shirt from several different servers. Instead of manually creating threads, you will leverage an `ExecutorService` from the `java.util.concurrent` package.

Tasks

1. Open the `ExecutorService13-02Prac` project as the main project.
   a. Select File > Open Project.
   b. Browse to /home/oracle/labs/13-Concurrency/practices/practice2.
   c. Select ExecutorService13-02Prac and click the Open Project button.
2. Expand the project directories.
3. Run the `NetworkServerMain` class in the `com.example.server` package by right-clicking the class and selecting Run File.
4. Open the `NetworkClientMain` class in the `com.example.client` package.
5. Run the `NetworkClientMain` class package by right-clicking the class and selecting Run File. Notice the amount of time it takes to query all the servers sequentially.
6. Create a `NetworkClientCallable` class in the `com.example.client` package that implements the Callable interface. Use the notation for generics to define the Callable as of type RequestResponse.

```
public class NetworkClientCallable implements
Callable<RequestResponse>
```

   NetBeans shortcut: Right-click and select Fix Imports to add the necessary import statement.

   a. Add a constructor and a field named lookup of type `RequestResponse` to receive and store a `RequestResponse` reference during construction.

   NetBeans shortcut: Add the field first, as a private class field, then right-click and select Insert Code. Then Select Constructor. Select the lookup field and click Generate.

   b. Implement the `Callable` interface with a generic type of `RequestResponse`.

NetBeans shortcut: Select the light bulb beside the class signature and click Implement all abstract methods.

c. Remove the line of code in the generated `call` method.

d. Complete the `call` method by using a `java.net.Socket` and a `java.util.Scanner` to read the response from the server. Store the result in the `RequestResponse` object and return it.

Note: You may want to use a `try-with-resource` statement to ensure that the `Socket` and `Scanner` objects are closed.

```
try (Socket sock = new Socket(lookup.host, lookup.port);
  Scanner scanner = new Scanner(sock.getInputStream())) {
  lookup.response = scanner.next();
  return lookup;
}
```

e. Use the NetBeans hint above to add the necessary import statements.

f. **Note:** Click the lightbulb with the caution triangle next to the class field to add `final` to the class field instance.

g. Save the file.

7. Modify the `main` method of the `NetworkClientMain` class to query the servers concurrently by using an `ExecutorService`.

a. Comment out the contents of the `main` method.

b. Obtain an `ExecutorService` that reuses a pool of cached threads.

```
ExecutorService es = Executors.newCachedThreadPool();
```

c. Create a `Map` that will be used to tie a request to a future response.

```
Map<RequestResponse, Future<RequestResponse>> callables = new
HashMap<>();
```

d. Copy the following lines of the `for` loop and code that creates an instance of a `RequestResponse` from the commented out code:

```
String host = "localhost";
for (int port = 10000; port < 10010; port++) {
  RequestResponse lookup = new RequestResponse(host, port);
```

e. Add a line of code that creates an instance of a `NetworkClientCallable` and passes the instance of the `RequestResponse` object to it for each network request.

f. Submit each `NetworkClientCallable` to the ExecutorService. Store each `Future` in the `Map` created above.

g. Your complete for loop should look like this:

```
for (int port = 10000; port < 10010; port++) {
  RequestResponse lookup = new RequestResponse(host, port);
  NetworkClientCallable callable =
    new NetworkClientCallable(lookup);
```

```
  Future<RequestResponse> future = es.submit(callable);
  callables.put(lookup, future);
}
```

h. Shut down the `ExecutorService`.

i. Await the termination of all threads within the `ExecutorService` for 5 seconds. Recall from the lesson that `awaitTermination` method throws an `InterruptedException`, so use a try-catch block.

```
es.shutdown();

try {
  es.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException ex) {
  System.out.println("Stopped waiting early");
}
```

j. Loop through the `Future` objects stored in the `Map` created above. Use the `keyset` method to return and `Iterable` that contains the `RequestResponse` object.

k. Get the `Future<RequestResponse>` object from the `RequestResponse` object retrieved from the `Map`.

l. Print out the servers' response or an error message with the server details if there was a problem communicating with a server.

m. Your code should look similar to this:

```
for (RequestResponse lookup : callables.keySet()) {
  Future<RequestResponse> future = callables.get(lookup);
  try {
    lookup = future.get();
    System.out.println(lookup.host + ":" + lookup.port + " " +
                       lookup.response);
  } catch (ExecutionException | InterruptedException ex) {
    System.out.println("Error talking to " + lookup.host +
                       ":" + lookup.port);
  }
}
```

8. Run the `NetworkClientMain` class by right-clicking the class and selecting Run File. Notice the amount of time it takes to query all the servers concurrently.

9. When done testing your client, be sure to select the `ExecutorService` output tab and terminate the server application.

Practices for Lesson 14: Parallel Streams

Practices for Lesson 14: Overview

Practice Overview

In these practices, explore the parallel stream options available in Java.

Old Style Loop

The following example iterates through an `Employee` list. Each member who is from Colorado and is an executive has their information printed out. In addition, the `sum` mutator is used to calculate the total amount of executive pay for the selected group.

A01OldStyleLoop.java

```
 9 public class A01OldStyleLoop {
10
11     public static void main(String[] args) {
12
13         List<Employee> eList = Employee.createShortList();
14
15         double sum = 0;
16
17         for(Employee e:eList){
18             if(e.getState().equals("CO") &&
19                 e.getRole().equals(Role.EXECUTIVE)){
20                 e.printSummary();
21                 sum += e.getSalary();
22             }
23         }
24
25         System.out.printf("Total CO Executive Pay: $%,9.2f %n", sum);
26     }
27
28 }
```

There are a couple of key points that can be made about the above code.

- All elements in the collections must be iterated through every time.
- The code is more about "how" information is obtained and less about "what" the code is trying to accomplish.
- A mutator must be added to the loop to calculate the total.
- There is no easy way to parallelize this code.

The output from the program is as follows.

Output

```
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Total CO Executive Pay: $370,000.00
```

Lambda Style Loop

The following example shows the new approach to obtaining the same data using lambda expressions. A stream is created, filtered, and printed. A `map` method is used to extract the salary data, which is then summed and returned.

A02NewStyleLoop.java

```
public class A02NewStyleLoop {

    public static void main(String[] args) {

        List<Employee> eList = Employee.createShortList();

        double result = eList.stream()
            .filter(e -> e.getState().equals("CO"))
            .filter(e -> e.getRole().equals(Role.EXECUTIVE))
            .peek(e -> e.printSummary())
            .mapToDouble(e -> e.getSalary())
            .sum();

        System.out.printf("Total CO Executive Pay: $%,9.2f %n", result);
    }

}
```

There are also some key points worth pointing out for this piece of code as well.

- The code reads much more like a problem statement.
- No mutator is needed to get the final result.
- Using this approach provides more opportunity for lazy optimizations.
- This code can easily be parallelized.

The output from the example is as follows.

Output

```
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Total CO Executive Pay: $370,000.00
```

Streams with Code

So far all the examples have used lambda expressions and stream pipelines to perform the tasks. In this example, the `Stream` class is used with regular Java statements to perform the same steps as those found in a pipeline.

A03CodeStream.java

```
11 public class A03CodeStream {
12
13     public static void main(String[] args) {
14
15         List<Employee> eList = Employee.createShortList();
16
17         Stream<Employee> s1 = eList.stream();
18
19         Stream<Employee> s2 = s1.filter(
20             e -> e.getState().equals("CO"));
21
22         Stream<Employee> s3 = s2.filter(
23             e -> e.getRole().equals(Role.EXECUTIVE));
24         Stream<Employee> s4 = s3.peek(e -> e.printSummary());
25         DoubleStream s5 = s4.mapToDouble(e -> e.getSalary());
26         double result = s5.sum();
27
28         System.out.printf("Total CO Executive Pay: $%,9.2f %n", result);
29     }
30
31 }
```

Even though the approach is possible, a stream pipeline seems like a much better solution.

The output from the program is as follows.

Output

```
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Total CO Executive Pay: $370,000.00
```

Making a Stream Parallel

Making a stream run in parallel is pretty easy. Just call the `parallelStream` or `parallel` method in the stream. With that call, when the stream executes it uses all the processing cores available to the current JVM to perform the task.

A04Parallel.java

```
 9 public class A04Parallel {
10
11     public static void main(String[] args) {
12
13         List<Employee> eList = Employee.createShortList();
14
15         double result = eList.parallelStream()
16             .filter(e -> e.getState().equals("CO"))
17             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18             .peek(e -> e.printSummary())
19             .mapToDouble(e -> e.getSalary())
20             .sum();
21
22         System.out.printf("Total CO Executive Pay: $%,9.2f %n", result);
23
24         System.out.println("\n");
25
26         // Call parallel from pipeline
27         result = eList.stream()
28             .filter(e -> e.getState().equals("CO"))
29             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
30             .peek(e -> e.printSummary())
31             .mapToDouble(e -> e.getSalary())
32             .parallel()
33             .sum();
34
35         System.out.printf("Total CO Executive Pay: $%,9.2f %n", result);
36
37         System.out.println("\n");
38
39         // Call sequential from pipeline
40         result = eList.stream()
41             .filter(e -> e.getState().equals("CO"))
42             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
43             .peek(e -> e.printSummary())
44             .mapToDouble(e -> e.getSalary())
45             .sequential()
46             .sum();
47
48         System.out.printf("Total CO Executive Pay: $%,9.2f %n", result);
49     }
50 }
```

Remember, the last call wins. So if you call the sequential method after the parallel method in your pipeline, the pipeline will execute serially.

The following output is produced for this sample program.

Output

```
Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Total CO Executive Pay: $370,000.00


Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Total CO Executive Pay: $370,000.00


Name: Joe Bailey Role: EXECUTIVE Dept: Eng St: CO Salary: $120,000.00
Name: Phil Smith Role: EXECUTIVE Dept: HR St: CO Salary: $110,000.00
Name: Betty Jones Role: EXECUTIVE Dept: Sales St: CO Salary: $140,000.00
Total CO Executive Pay: $370,000.00
```

Stateful Versus Stateless Operations

You should avoid using stateful operations on collections when using stream pipelines. The `collect` method and `Collectors` class have been designed to work with both serial and parallel pipelines.

A05AvoidStateful.java

```
11 public class A05AvoidStateful {
12
13     public static void main(String[] args) {
14
15         List<Employee> eList = Employee.createShortList();
16         List<Employee> newList01 = new ArrayList<>();
17         List<Employee> newList02 = new ArrayList<>();
18
19         eList.parallelStream() // Not Parallel. Bad.
20             .filter(e -> e.getDept().equals("Eng"))
21             .forEach(e -> newList01.add(e));
22
23         newList02 = eList.parallelStream() // Good Parallel
24             .filter(e -> e.getDept().equals("Eng"))
25             .collect(Collectors.toList());
26
27     }
28 }
```

Lines 19 to 21 show you how NOT to extract data from a pipeline. Your operations may not be thread safe. Lines 23 to 25 demonstrate the correct method for saving data from a pipeline using the `collect` method and `Collectors` class.

Deterministic and Non-Deterministic Operations

Most stream pipelines are deterministic. That means that whether the pipeline is processed serially or in parallel the result will be the same.

A06Determine.java

```
10 public class A06Determine {
11
12     public static void main(String[] args) {
13
14         List<Employee> eList = Employee.createShortList();
15
16         double r1 = eList.stream()
17             .filter(e -> e.getState().equals("CO"))
18             .mapToDouble(Employee::getSalary)
19             .sequential().sum();
20
21         double r2 = eList.stream()
22             .filter(e -> e.getState().equals("CO"))
23             .mapToDouble(Employee::getSalary)
24             .parallel().sum();
25
26         System.out.println("The same: " + (r1 == r2));
27     }
28 }
```

The example shows that the result for a sum is the same that is processed using either highlighted method.

The output from the sample is as follows:

Output

```
The same: true
```

However, some operations are not deterministic. The `findAny()` method is a short-circuit terminal operation that may produce different results when processed in parallel.

A07DetermineNot.java

```
10 public class A07DetermineNot {
11
12     public static void main(String[] args) {
13
14         List<Employee> eList = Employee.createShortList();
15
16         Optional<Employee> e1 = eList.stream()
17             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18             .sequential().findAny();
19
20         Optional<Employee> e2 = eList.stream()
21             .filter(e -> e.getRole().equals(Role.EXECUTIVE))
22             .parallel().findAny();
23
24         System.out.println("The same: " +
```

```
25              e1.get().getEmail().equals(e2.get().getEmail()));
26
27      }
28  }
```

The data set used in the example is fairly small therefore the two different approaches will often produce the same result. However, with a larger data set, it becomes more likely that the results produced will not be the same.

Reduction

The reduce method performs reduction operations for the stream libraries. The following example sums numbers 1 to 5.

A08Reduction.java

```
 9 public class A08Reduction {
10
11     public static void main(String[] args) {
12
13         int r1 = IntStream.rangeClosed(1, 5).parallel()
14             .reduce(0, (a, b) -> a + b);
15
16         System.out.println("Result: " + r1);
17
18         int r2 = IntStream.rangeClosed(1, 5).parallel()
19             .reduce(0, (sum, element) -> sum + element);
20
21         System.out.println("Result: " + r2);
22
23     }
```

Two examples are shown. The second example started on line 18 uses more description variables to show how the two variables are used. The left value is used as an accumulator. The value on the right is added to the value on the left. Reductions must be associative operations to get a correct result.

The output from both expressions should be the following:

Output

```
Result: 15
Result: 15
```

Practice 14-1: Calculate Total Sales Without a Pipeline

Overview

In this practice, calculate the sales total for Radio Hut using the Stream class and normal Java statements.

Assumptions

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

Tasks

1. Open the `SalesTxn14-01Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/ 14-ParallelStreams /practices/practice1`.
   - Select `SalesTxn14-01Prac` and click the Open Project button.
2. Expand the project directories.
3. Edit the `CalcTest` class to perform the steps in this practice.
4. Calculate the total sales for Radio Hut using the Stream class and Java statements.

   Create a stream from `tList` and assign it to: `Stream<SalesTxn> s1`

   Create a second stream and assign the results of the `filter` method for Radio Hut transactions: `Stream<SalesTxn> s2`

   Create a third stream and assign the results from a `mapToDouble` method that returns the transaction total: `DoubleStream s3`

   Sum the final stream and assign the result to: `double t1`.
5. Print the results.

   Hint: Be mindful of the method return types. Use the API doc to ensure that you are using the correct methods and classes to create and store results.
6. The output from your test class should be similar to the following:

```
=== Transactions Totals ===
Radio Hut Total: $3,840,000.00
```

Practice 14-2: Calculate Sales Totals Using Parallel Streams

Overview

In this practice, calculate the sales totals from the collection of sales transactions.

Assumptions

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

Tasks

1. Open the `SalesTxn14-02Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/ 14-ParallelStreams /practices/practice2`.
   - Select `SalesTxn14-02Prac` and click the Open Project button.
2. Expand the project directories.
3. Edit the `CalcTest` class to perform the steps in this practice.
4. Calculate the total sales for Radio Hut, PriceCo, and Best Deals.
   a. Calculate the Radio Hut total using the `parallelStream` method. The pipeline should contain the following methods: `parallelStream`, `filter`, `mapToDouble`, and `sum`.
   b. Calculate the PriceCo total using the `parallel` method. The pipeline should contain the following methods: `filter`, `mapToDouble`, `parallel`, and `sum`.
   c. Calculate the Best Deals total using the `sequential` method. The pipeline should contain the following methods: `filter`, `mapToDouble`, `sequential`, and `sum`.
5. Print the results.
6. The output from your test class should be similar to the following:

```
=== Transactions Totals ===
Radio Hut Total: $3,840,000.00
PriceCo Total: $1,460,000.00
Best Deals Total: $1,300,000.00
```

Practice 14-3: Calculate Sales Totals Using Parallel Streams and Reduce

Overview

In this practice, calculate the sales totals from the collection of sales transactions using the `reduce` method.

Assumptions

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

Tasks

1. Open the `SalesTxn14-03Prac` project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/ 14-ParallelStreams /practices/practice3`.
   - Select `SalesTxn14-03Prac` and click the Open Project button.
2. Expand the project directories.
3. Edit the `CalcTest` class to perform the steps in this practice.
4. Calculate the total sales for PriceCo using the `reduce` method instead of `sum`.
   a. Your pipeline should consist of: `filter`, `mapToDouble`, `parallel`, and `reduce`.
   b. The reduce function can be defined as: `reduce(0, (sum, e) -> sum + e)`
5. In addition, calculate the total number of transactions for PriceCo using `map` and `reduce`.
   a. Your pipeline should consist of: `filter`, `mapToInt`, `parallel`, and `reduce`.
   b. To count the transactions, use: `mapToInt(t -> 1)`
   c. The `reduce` function can be defined as: `reduce(0, (sum, e) -> sum + e).`
6. Print the results.
7. The output from your test class should be similar to the following:

```
=== Transactions Totals ===

PriceCo Total: $1,460,000.00
PriceCo Transactions:      4
```

Practices for Lesson 15: Terminal Operations: Collectors

Practices for Lesson 15: Overview

Overview

In these practices, you work through a number of examples of using predefined collectors.

This is the last practice working with lambda and streams to summarize and process data, so you first do some review exercises that summarize the advantages of streams, and especially of using the `collect` method to work with the data in aggregate.

You will work with a class `ComplexSalesTxn` that is very similar to the class `SalesTxn` you used in earlier lessons, but it permits a single transaction to be for any number of items (using the `lineItems` field of type `List<LineItem>`). This will allow you to explore `groupingBy` and `flatMapping` collectors in detail.

Practice 15-1: Review: A Comparison of Iterative Approach, Streams, and Collectors

Overview

In this practice you use a number of different ways to extract and display information in a collection. You'll use: iterative, aggregate `List`, aggregate `Stream`, and finally a collector. Because the data requirements are very straightforward at first, all four methods work. As the queries become more complex, you see how collectors support the aggregate programming paradigm.

Assumptions

Tasks

1. In NetBeans, open the Prac_15_Collectors project from `/home/oracle/labs/15_Collectors/practices`

   As you work through the instructions, you'll frequently add classes that require an import statement. NetBeans will often flag this with a small lightbulb icon that you can click to have NetBeans do the import for you.

2. Display a summary list of all transactions using an iterative approach.

   a. Enter and run the following code in the file `CollectorExamples.java`:

   ```
   List<ComplexSalesTxn> tList = ComplexSalesTxn.createTxnList();
   for (ComplexSalesTxn currTxn: tList) {
      System.out.println(currTxn.getSummaryStr());
   }
   ```

   b. You should see something like the following. Notice how some transactions contain more than one item.

   ```
   ID: 11 - Seller: Jane Doe - Buyer: Acme Electronics - Line
   items: [Widget, Widget Pro II] - ST: CA - Date: 2013-01-25
   ID: 12 - Seller: Jane Doe - Buyer: Acme Electronics - Line
   items: [Widget Pro] - ST: CA - Date: 2013-04-05
   ID: 13 - Seller: Rob Doe - Buyer: Radio Hut - Line items:
   [Widget Pro II, Widget] - ST: CA - Date: 2013-10-03
   ...
   ```

   c. Comment out the your code (or just the `System.out.println()` line) before doing the next task but make sure to leave the line that populates `tList`, as all your later examples will use it.

3. Display a summary list of all transactions using a List `forEach` approach.

   a. Now enter and run the following code:

   ```
   tList.forEach(a -> System.out.println(a.getSummaryStr()));
   ```

b. You should see the same output as before.

This is neat and convenient, but if anything it's more inflexible than the `for` loop. Nevertheless, let's see if it's possible to modify the output with this approach.

4. Modify this code to output results for only the buyer PriceCo. How could you do this?

a. One way could be to use a ternary expression. Try the following:

```
tList.forEach(a -> System.out.println(a.getBuyerName()
   .equals("PriceCo")?a.getSummaryStr():""));
```

b. You should see output something like the following:

```
ID: 17 - Seller: Dave Smith - Buyer: PriceCo - Line items:
[Widget] - ST: CO - Date: 2013-03-20
ID: 18 - Seller: Dave Smith - Buyer: PriceCo - Line items:
[Widget Pro II] - ST: CO - Date: 2013-03-30

ID: 20 - Seller: John Adams - Buyer: PriceCo - Line items:
[Widget Pro II] - ST: MA - Date: 2013-07-14
ID: 21 - Seller: John Adams - Buyer: PriceCo - Line items:
[Widget] - ST: MA - Date: 2013-10-06
```

This sort of works but you're getting blank lines for the buyers other than PriceCo. Again this is fixable, perhaps like this:

```
tList.forEach(a -> System.out.print(a.getBuyerName()
   .equals("PriceCo")?a.getSummaryStr() + "\n":""));
```

```
tList.forEach(a -> System.out.print(a.getBuyerName()
   .equals("PriceCo")?a.getSummaryStr() + "\n":""));
```

But it's hardly elegant, readable, or fluent. Let's see how it would look in the more fluent programming style of using streams. Using streams will allow modifying the `List` however we choose, then printing out the `List` or passing it to another method as needed.

c. Comment out this line and move on the next task.

5. Use a `Stream` and some of its methods to achieve the same output.

a. Enter and run the following code.

```
tList.stream()
  .filter(f -> f.getBuyerName().equals("PriceCo"))
  .map(ComplexSalesTxn::getSummaryStr)
  .forEach(System.out::println);
```

b. You should see the output just for PriceCo.

```
ID: 17 - Seller: Dave Smith - Buyer: PriceCo - Line items:
[Widget] - ST: CO - Date: 2013-03-20
```

```
ID: 18 - Seller: Dave Smith - Buyer: PriceCo - Line items:
[Widget Pro II] - ST: CO - Date: 2013-03-30
ID: 20 - Seller: John Adams - Buyer: PriceCo - Line items:
[Widget Pro II] - ST: MA - Date: 2013-07-14
ID: 21 - Seller: John Adams - Buyer: PriceCo - Line items:
[Widget] - ST: MA - Date: 2013-10-06
```

This is very clear, and allows further modifications to the output in the same fluent style. But there are still some issues.

- Often printing to the console is not what is required; rather, a summary collection that can be passed to another object may be needed. And `forEach()` is not the recommended way to create and populate such a collection.
- Creating any kind of classification using stream methods is not possible. For example, instead of list of all PriceCo transactions, you might want a list of transactions grouped by buyer.

  The `collect` method can achieve these needs, and the predefined collectors are designed to be a standard way to terminate a stream. (Using the `reduce` method is also a standard way to terminate a stream, but it is less flexible than `collect`).

6. Use a collector to summarize the result.
   a. Replace the forEach method with a collect method, and use the `toList` collector to store the result.

```
tList.stream()
  .filter(f -> f.getBuyerName().equals("PriceCo"))
  .map(ComplexSalesTxn::getSummaryStr)
  .collect(toList());
```

   You may need to add a static import:

```
static java.util.stream.Collectors.toList;
```

   b. Assign the result of the collect method to a variable of type `List<String>` and add `a forEach` on the `List` to see its contents.

```
List<String> priceCoSummary = tList.stream()
  .filter(f -> f.getBuyerName().equals("PriceCo"))
  .map(ComplexSalesTxn::getSummaryStr)
  .collect(toList());

priceCoSummary.forEach(System.out::println);
```

This looks just like when you printed the stream. But it's more useful as printing to the console is not likely to be the reason you manipulate data; it's likely that you need the data in a form that can be stored and possibly passed to other methods and objects. In this example, `priceCoSummary` stores the modifications.

Practice 15-2: Using Collectors for Grouping

Overview

In this practice, you work with operations that are more complex and explore grouping transactions in many different ways.

Assumptions

Tasks

1. Use the `collect` method and predefined collectors to list all transactions grouped by buyer.

   You need to set up the collection that will result from a simple group by. (It'll be a `Map`, and as before you can use a `forEach` on the `Map` to display its contents, but the `forEach` will require a `BiConsumer` as `Map` types have both keys and values).

   a. Comment out the previous code that prints out the `List`, and add the following code (as you don't add a semi-colon yet, you may see errors in the IDE even after you add necessary imports):

   ```
   Map<Buyer,List<ComplexSalesTxn>> txns =
      tList.stream()
   ```

   b. Now add the necessary collect and appropriate collector to group the transactions.

   ```
   .collect(groupingBy(ComplexSalesTxn::getBuyer));
   ```

   c. Add `System.out.println` to print out the collection. It's a Map and has a toString method so it'll print out a readable result.

   d. `System.out.println(txns);`

   e. You should see something like this (your output may be different as `Map` does not guarantee order, and much is omitted):

   ```
   {Great Deals=[Transaction id: 14Sales person: John SmithBuyer
   name: Great DealsBuyer class: BASIC Line items: [Widget, Widget
   Pro, Widget Pro II]Tax rate: 0.09Discount rate: 0.0Transaction
   date: 2013-10-10City: San JoseState: CACode: 95101
   ], PriceCo=[Transaction id: 17Sales person: Dave SmithBuyer
   name: PriceCoBuyer class:
   ...
   ```

   It may be hard to see that the transactions are grouped by buyer. Summary output would be more suitable.

2. Note that you can't add:

   ```
   .map(ComplexSalesTxn::getSummaryStr)
   ```

   before the `collect` method, as the `groupingBy` needs a transaction to do its job. You have to use the mapping collector instead, downstream of the `groupingBy`. Note that some of the methods of `Stream` reappear as collectors so they can be combined with `groupingBy` in this way (map and mapping, filter and filtering, etc.)

a. Modify the collect method to include the mapping collector as shown here (you'll get an error even after doing the necessary imports):

```
.collect(groupingBy(ComplexSalesTxn::getBuyer,
      mapping(ComplexSalesTxn::getSummaryStr, toList())));
```

b. The IDE shows an error. By changing the mapping, you've changed the return type of the `collect` method. You'll find this happens often as you build up more complex pipelines. In this case the ToolTip is fairly easy to understand. The Map will now contain `List<String>` as the value type as `ComplexSalesTxn` has been mapped to `String`.

c. Change the txns type to reflect this:

```
Map<Buyer,List<String>> txns =
```

d. Run the code. You should see something like this (order may be different):

```
{PriceCo=[ID: 17 - Seller: Dave Smith - Buyer: PriceCo -  s:
[Widget] - ST: CO - Date: 2013-03-20, ID: 18 - Seller: Dave
Smith - Buyer: PriceCo - Line items: [Widget Pro II] - ST: CO -
Date: 2013-03-30, ID: 20 - Seller: John Adams - Buyer: PriceCo -
Line items: [Widget Pro II] - ST: MA - Date: 2013-07-14, ID: 21
- Seller: John Adams - Buyer: PriceCo - Line items: [Widget] -
ST: MA - Date: 2013-10-06], Mom and Pops=[ID: 22 - Seller:
Samuel Adams - Buyer: Mom and Pops - Line items: [Widget Pro II]
- ST: MA - Date: ...
```

e. Still pretty hard to decipher! It may be better to use `forEach` with the `Map` collection instead of System.out.println. That will enable inserting returns and that will make things much more readable.

3. Use `forEach` instead of `System.out.println` to display the output `Map`.

a. Comment out the `System.out.println` command and instead call `forEach` on `txns`. The `forEach` method for a `Map` is a `BiConsumer` so there are two parameters. You need to declare both, but you only need to print the value parameter, the `List`, as the summary includes the buyer name anyway.

```
txns.forEach((a, b) -> b.forEach(System.out::println));
```

b. However, by using the first paramenter, `a`, you can make things much more readable.

```
txns.forEach((a, b) -> { System.out.println("<< " + a + " >> ");
                         b.forEach(System.out::println);
        });
```

c. The output will be like this:

```
<< Great Deals >>
ID: 14 - Seller: John Smith - Buyer: Great Deals - Line items:
[Widget, Widget Pro, Widget Pro II] - ST: CA - Date: 2013-10-10
<< Acme Electronics >>
ID: 11 - Seller: Jane Doe - Buyer: Acme Electronics - Line
items: [Widget, Widget Pro II] - ST: CA - Date: 2013-01-25
ID: 12 - Seller: Jane Doe - Buyer: Acme Electronics - Line
items: [Widget Pro] - ST: CA - Date: 2013-04-05
```

```
... < remainder of output omitted >
```

Note that you could just tack the `forEach` method call just after the `collect` method, though you would have to remove the assignment to `txns`.

4. Now that you have grouped transactions by buyer, try grouping `LineItem` elements by salesperson. This time you put them into a collection and just use `System.out.println` to print out the collection.

   What should be the type? Each transaction consists of a number of number of `LineItem` elements stored in a `List`. Using `groupingBy` ensures the transactions will be in a `List`, and each of those transactions will contain a `List` of `LineItem` elements. Therefore the type is:

   ```
   Map<String, List<List<LineItem>>>
   ```

   a. Create the first line of code to declare a collection variable txnDetails.

   ```
   Map<String, List<List<LineItem >>> txnDetails
                   = tList.stream()
   ```

   b. Now create a collection just as you did for transactions by buyer, but instead for `lineItems` by `salesperson`.

   ```
   .collect(groupingBy(ComplexSalesTxn::getSalesPerson,
   mapping(ComplexSalesTxn::getLineItems, toList())));
   ```

   c. Finally add a line to print the collection and run the code:

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

   d. You should see something like this:

   ```
   {Samuel Adams=[[Widget Pro II], [Widget Pro, Widget]], John
   Smith=[[Widget, Widget Pro, Widget Pro II]], Rob Doe=[[Widget
   Pro II, Widget]], John Adams=[[Widget Pro II], [Widget]], Betty
   Jones=[[Widget Pro, Widget], [Widget Pro II], [Widget Pro II]],
   Dave Smith=[[Widget], [Widget Pro II]], Jane Doe=[[Widget,
   Widget Pro II], [Widget Pro]]}
   ```

   What is interesting here is that for each salesperson we can see their list of transactions, and for each transaction, we can see the individual `LineItem` elements. For example, Samual Adams (first in the output) has two transactions, the first of consisting of one `LineItem` element, the second of which consisted of two `LineItem` elements.

   This information may be exactly what you want, but if you want to aggregate `LineItem` elements in some way per sales person irrespective of the transactions, you don't want this level of nesting.

5. Remove a level of nesting so that we get only the `lineItem` elements for each salesperson. Given the code you have already that uses mapping, this is quite straightforward. You only need to exchange mapping for flatmapping (and make some adjustments to conform with this).

a. Modify the code you've written to change "mapping" to "flatMapping". You will see some errors – don't worry about this. The collect method will now be:

```
.collect(groupingBy(ComplexSalesTxn::getSalesPerson,
     flatMapping(ComplexSalesTxn::getLineItems, toList())));
```

b. Why are there errors? (there are actually two reasons). Look at the API docs for the `flatMapping` collector factory method. Notice it requires a `Stream` and a collector. To fix this modify the collect method again to create a `Stream` from the `List` of `LineItems`. Note you won't be able to use a method reference any more.

```
.collect(groupingBy(ComplexSalesTxn::getSalesPerson,
   flatMapping(t -> t.getLineItems().stream(), toList())));
```

c. There's still an error. Remember, a level of nesting is going away so the type of the resultant collection must adjust. Change the type accordingly and run the code.

```
Map<String, List<LineItem>>
```

d. On running the code you should see:

```
{Samuel Adams=[Widget Pro II, Widget Pro, Widget], John
Smith=[Widget, Widget Pro, Widget Pro II], Rob Doe=[Widget Pro
II, Widget], John Adams=[Widget Pro II, Widget], Betty
Jones=[Widget Pro, Widget, Widget Pro II, Widget Pro II], Dave
Smith=[Widget, Widget Pro II], Jane Doe=[Widget, Widget Pro II,
Widget Pro]}
```

e. Notice that now all `LineItem` elements for a salesperson are in the same `List`.

6. With a few simple modifications of your code so far, you can now produce various counts related to a salesperson's `LineItem` elements.

a. Count the numbers of `LineItem` elements for each salesperson. Modify the collect method to use `counting` instead of `toList()` as the final collector:

```
.collect(groupingBy(ComplexSalesTxn::getSalesPerson,
flatMapping(t -> t.getLineItems().stream(), counting())));
```

b. and modify the result type:

```
Map<String, Long> txnDetails
```

c. The result should be:

```
{Samuel Adams=3, John Smith=3, Rob Doe=2, John Adams=2, Betty
Jones=4, Dave Smith=2, Jane Doe=3}
```

7. Now return the total sales each salesperson has achieved. Note that each `LineItem` element, for, say, a Widget, has a quantity.

a. You only need to substitute counting with `summingDouble` (and change the type of `txnDetails` accordingly). Here's the code for the collect method:

```
.collect(groupingBy(ComplexSalesTxn::getSalesPerson,
   flatMapping(t -> t.getLineItems().stream(),
   summingDouble(s -> s.getQuantity() * s.getUnitPrice()))));
```

b. Here is the result.

```
{Samuel Adams=87600.0, John Smith=116000.0, Rob Doe=58500.0,
John Adams=57000.0, Betty Jones=171700.0, Dave Smith=34500.0,
Jane Doe=71000.0}
```

It would be nice to have this result in order but the `Map` type is not ordered. But there is a way!

8. Order the total sales for each salesperson. To do this, you need to create a new type based on the `Map` that you can order. You can use the `Map` method `entrySet()`, to create a `Set` based on the `Map`, create a stream from that, and finally order the stream. Using `entrySet()` to work with `Map` collections in this way is quite common when the processing cannot be done in a single pipeline.
   a. Remove or comment out the result type from the previous code and remove or comment out the System.out.println. You will perform the sorted output without creating a collection reference.
   b. Between the final ")" of the collect method and the semicolon add the following code putting it on a new line for readability:

```
.entrySet()
.stream()
.sorted()
```

   c. Now add a forEach to print out the sorted stream. The entire code should look like this:

```
tList.stream()
   .collect(groupingBy(ComplexSalesTxn::getSalesPerson,
         flatMapping(t -> t.getLineItems().stream(),
         summingDouble(s -> s.getQuantity() *
s.getUnitPrice()))))
                  .entrySet()
                  .stream()
                  .sorted()
                  .forEach(System.out::println);
```

   d. Run the code (there will be errors!) You will see something like this:

```
Exception in thread "main" java.lang.ClassCastException:
java.base/java.util.HashMap$Node cannot be cast to
java.base/java.lang.Comparable
…
```

   e. What's happening is that you're trying to sort the `Entry` type in the `Set` and it does not implement `Comparable`! You need to use the sorted method that takes a `Comparator`.
   f. Add `Comparator.comparing(Entry::getValue)` as a parameter to sorted. Run the code.

g. It should look like this:

```
Dave Smith=34500.0
John Adams=57000.0
Rob Doe=58000.0
Jane Doe=71000.0
Samuel Adams=87600.0
John Smith=116000.0
Betty Jones=171700.0
```

h. How would you reverse the order? Just use `Comparator.reversed()`. Try this if you like, but you'll find there's a problem. If you use `reversed()`, the compiler struggles with inferring the types and you'll have to explicitly declare the lambda type like this:

```
.sorted(Comparator.comparing((Entry<String, Double> a) ->
a.getValue()).reversed())
```

i. Finally, try making the stream parallel. Change the first line this code fragment to:

```
tlist.parallelStream()
```

j. **IMPORTANT – NOTE!**

Does it work? It should, but there are two pipelines here! Change the `stream()` method after `entrySet()` to `parallelStream()`. Does it still work? You'll see it's no longer ordered. `forEach()` does not respect `sorted()`.

You can get around this by using `forEachOrdered()`, but this may have a performance hit that may in some cases negate the parallel performance advantage. Ordering and parallelism don't always coexist well. If you *do* need it sorted and (as you should) wish to make it parallel-ready, use a collector here. The `forEach()` for displaying data in these exercises should be taken only as a demonstration.

9. For the final task, something that sounds quite difficult but is really quite straightforward. List all transactions ordered by City, SalesPerson, and Buyer. To do this you use a nested groupingBy.

a. Enter the following code:

```
tList.stream()
.collect(groupingBy(ComplexSalesTxn::getCity,
                    groupingBy(ComplexSalesTxn::getSalesPerson,
                    groupingBy(ComplexSalesTxn::getBuyer,

mapping(ComplexSalesTxn::getLineItems,
                              toList()))))));
```

Note how each groupingBy is contained within its parent groupingBy.

b. Use forEach to display the result.

```
.forEach((a, b) -> b.forEach((c, d) -> d.forEach((e, f) ->
System.out.println(a + " : " + c + " : " + e + " : " + f) )));
```

The " : " are included to make the data more readable.

c. Try running the code. You should see something like the following:

```
Denver : Betty Jones : Best Deals : [[Widget Pro II], [Widget
Pro II]]
Denver : Betty Jones : Radio Hut : [[Widget Pro, Widget]]
Denver : Dave Smith : PriceCo : [[Widget], [Widget Pro II]]
San Jose : John Smith : Great Deals : [[Widget, Widget Pro,
Widget Pro II]]
San Jose : Rob Doe : Radio Hut : [[Widget Pro II, Widget]]
San Jose : Jane Doe : Acme Electronics : [[Widget, Widget Pro
II], [Widget Pro]]
Boston : Samuel Adams : Radio Hut : [[Widget Pro, Widget]]
Boston : Samuel Adams : Mom and Pops : [[Widget Pro II]]
Boston : John Adams : PriceCo : [[Widget Pro II], [Widget]]
```

This is the end of this practice.

Practices for Lesson 16: Custom Streams

Practices for Lesson 16: Overview

Overview

In this practice, you look at some examples of the use of parallel streams. You look at:

- A program to calculate how many prime numbers exist within a range of numbers.
- An example of the prime number program running using JMH (Java Microbench Harness) to illustrate a more rigorous benchmarking process.
- A program to play Tictactoe that calculates for an entire 4X4 game exhaustively. This includes a version that has a custom spliterator and one that just uses streams.
- A custom Stream example that illustrates a custom Spliterator for an nary tree (a tree where each node can have, no, one, or many children.

Practice 16-1: Examine the PrimeNumbersExample Application

Overview

This program you examine in this practice calculates the number of primes in a specific range of numbers. As such, it's a good example of a program where parallelizing helps. The program consists of a nested loop where the inner loop takes a single number and determines if it is a prime, and the outer loop repeat that test for each number in the range. Because determining a prime for large numbers requires more processing, the test range is set to 100,000 to 1,000,000. This will produce results easily observable.

Assumptions

Tasks

1. Examine the program.
   a. Open the project PrimeNumbersExample in the folder: `/home/oracle/labs/16_CustomStreams/examples`, and open the `Primenumbers.java` file.

      Each of the loops, outer and inner, are implemented in three distinct ways:

   - Using a for loop
   - Using a sequential stream
   - Using a parallel stream

   These six possibilities are set up in the code and you can choose between them by commenting or uncommenting out various options.
2. Run the code that uses for loops for both outer and inner loops several times.
   a. The program is set up initially in this way, so just run the project a few times. You'll see that though it varies, the program typically takes less than a second to run.
3. Try increasing the range to 100,000 to 10,000,000
   a. Change the line:

      ```
      long window = 1_000_000;
      ```

      to

      ```
      long window = 10_000_000;
      ```

   b. Run the program a few times. You will now see it takes about 10 times longer – around 10 seconds.
4. Now try making the outer loop run by using a Stream.
   a. Comment out the line:

      ```
      int howMany = outerPlainLoop(a -> isPrimePlainLoop(a), begin, end);
      ```

      and uncomment the line:

   b. `int howMany = outerStreamSequential(a -> isPrimePlainLoop(a), begin, end);`

c. Run the program a few times. You will find it runs in roughly the same amount of time as the previous example or takes a little longer. This is normal. Streams do add some overhead over a simple for loop. In some cases the optimization done by Streams can offset this or even give better performance, but in this example there isn't a set of intermediate operations that could be optimized.

5. Now try using parallel Streams for the outer loop.
   a. Comment out:

```
int howMany = outerStreamSequential(a -> isPrimePlainLoop(a),
begin, end)
```

   Uncomment:

```
int howMany = outerStreamParallel(a -> isPrimePlainLoop(a),
begin, end)
```

   b. Run the program a few times. You should find that it is faster. Your work machine has only two cores and it is a virtual machine so this negates some of the advantages or parallel, but it should still be visible. One a four core physical system, the increase in performance is typically X 2 or better.
6. Examine the work distribution. You use top for this. It's just an indication of what is happening.
   a. Open a terminal.
   b. Enter top to start the top program. You will see a list of the top programs currently running.
   c. Press 1. At the top of the screen you'll see the usage shown by individual CPU.
   d. Run the prime numbers program again while keeping an eye on top. You should see that for a few seconds the CPUs will go to the high nineties in usage, indicating that the prime number program is using parallel processing.
7. Now try some examples where the outer loop runs as a plain loop but the inner loop runs as a Stream.
   a. Change the range back to 100,000 to 1,000,000 (performance will be worse so you can have it do less work and still see the effects.)
   b. Comment out the current example and uncomment:

```
int howMany = outerPlainLoop(a -> isPrimeStreamSequential(a),
begin, end);
```

   Run the program a few times. You will find it to be somewhat slower than the first example (where both inner and outer loops were for loops). The combination of using Streams with a short-circuit terminal operation is not so efficient as just breaking out of a for loop.
8. Finally, try the parallel version of the inner loop.
   a. Comment out the current example and uncomment:

```
int howMany = outerPlainLoop(a -> isPrimeStreamParallel(a),
begin, end);
```

You will find this to be much slower! Again the use of Streams with a short-circuit terminal operation is not so efficient, but consider also the number of splits. This affects the efficiency of the noneMatch operation further.

9. To summarize, the best solution is to use a parallel stream for the outer loop and a standard for loop for the inner loop. This experience once again reaffirms the need to benchmark performance if one is to use parallel streams to attempt to improve performance.

   However the program is not rigorous in its performance measurement.

   In the next practice, you run this code in the Java Microbench Harness, a benchmarking system.

Practice 16-2: Using JMH (Java Microbench Harness)

Overview

In this practice, you use JMH to look at the performance of the prime numbers program.

Assumptions

Tasks

1. Run the prime number program in JMH.
   a. Open the ~/test, folder in a terminal.
   b. Enter the tree command so you can see the layout of the directory structure for the source code:

```
tree src
src
└── main
    └── java
        └── org
            └── sample
                └── MyBenchmark.java
```

   `MyBenchmark.java` contains the source code of the prime numbers program.
   c. Run the `tree` command on its own to see the entire directory structure. The JMH uses maven to create a JAR file that contains all the necessary code to test the code in `MyBenchmark.java`. This code is in `~/test/target/benchmarks.jar`.
   d. Run the benchmark code. `MyBenchmark.java` is set so that it is using standard for loops to determine the number of primes.

```
java –jar target/benchmarks.jar
```

   You will immediately see something like the following:

```
 [test]$ java -jar target/benchmarks.jar
# VM invoker: /usr/java/jdk-11.0.1/bin/java
# VM options: <none>
# Warmup: 20 iterations, 1 s each
# Measurement: 20 iterations, 1 s each
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: org.sample.MyBenchmark.testMethod

# Run progress: 0.00% complete, ETA 00:06:40
# Fork: 1 of 10
# Warmup Iteration   1: There are 76122 primes  between 100000
and 1100000
Took 0.46976206 seconds
```

```
There are 76122 primes  between 100000 and 1100000
Took 0.46976206 seconds
There are 76122 primes  between 100000 and 1100000
Took 0.46976206 seconds
2.109 ops/s
# Warmup Iteration   2: There are 76122 primes  between 100000
and 1100000
Took 0.5368709 seconds
There are 76122 primes  between 100000 and 1100000
Took 0.46976206 seconds
There are 76122 primes  between 100000 and 1100000
Took 0.46976206 seconds

<... further output omitted ...>
```

You can see from this that the code is being run many times. It will run for about 9 minutes and will then provide a summary of the performance of the code.

```
  Statistics: (min, avg, max) = (1.988, 2.118, 2.203), stdev =
0.046
  Confidence interval (99.9%): [2.107, 2.129]



# Run complete. Total time: 00:09:56

Benchmark                        Mode  Samples  Score  Score error
Units
o.s.MyBenchmark.testMethod      thrpt      200  2.118        0.011
ops/s
```

Note that the result is 2.118 operations per second. This gives you some well tested results to compare with, should you change the code.

2. Modify the code in `MyBenchmarks.java` and run the test again.
   a. Use the text editor to open the file `src/main/java/org/sample/MyBenchmark.java`
   b. Comment out the currently uncommented line, and uncomment
      `int howMany = outerStreamParallel(a -> isPrimePlainLoop(a), begin, end);`
   c. Recompile and recreate `benchmarks.jar` by running `./recompile` from the command line. You should see some output that eventually indicates that the build has been successful.
   d. Run the benchmark again. (`java –jar target/benchmarks.jar`).

e. Again, after about 9 minutes you'll see a summary. This time it'll show something like:

```
Result: 3.214 ±(99.9%) 0.043 ops/s [Average]
  Statistics: (min, avg, max) = (2.151, 3.214, 3.713), stdev =
0.183
  Confidence interval (99.9%): [3.171, 3.257]



# Run complete. Total time: 00:08:46

Benchmark                       Mode  Samples  Score  Score error
Units
o.s.MyBenchmark.testMethod     thrpt      200  3.214        0.043
ops/s
```

Note that it is significantly faster. Now you can be sure that when running in parallel on the two core image you're using, the prime numbers program runs about 50% faster than when running in sequential mode.

Practice 16-3: Run the Tictactoe Game Engine

Overview

In this practice, you begin the process of top down migration. For this practice and the remainder of these practices, you use a version of NetBeans that supports Java SE 9 and its module system.

Assumptions

Tasks

1. Examine the Tictactoe program.
   a. Open the project TictactoeExample in the folder `/home/oracle/16_CustomStreams/examples`.

      This program is a tictactoe engine that determines the best move on a 4 X 4 tictactoe game. Move calculation is exhaustive (calculating to the end of the game), and using a 4 X 4 board ensures that the calculation is deep enough to take a measurable time.

   b. Open the file Tictactest.java

      The code is currently set up to play tictactoe with the computer playing both sides. One side, the player of 'X' in this code example, plays using the tictactoe engine, while the player 'O' . A number of games are played, until the tictactoe engine wins.

   c. Look at the following parts of the code.

      Lines 41 to 45

      These lines set up a starting position with 5 moves already made. This makes the engine evaluation fast enough for use in this exercise. (If the tictactoe engine must move after only one move of the opponent it will take about 90 seconds.

      Line 90

      ```
      calculateMove(myTic, playerOne, true, true);
      ```

      This method call is to the method that sets up the `Stream` for evaluating moves. The third parameter determines if the `Stream` will run in parallel, and the fourth parameter affects the algorithm. This is included so that you can see that in some cases, it may well be more important to work on fine-tuning the algorithm than to make the program run in parallel.

      Lines 157 - 159

      These lines create a `Stream` that is simply a `List` of possible moves, and by using `map`, transforms each move into an evaluation. Then lines 164 - 170 choose the move to make based on the evaluations.

2. Run the program.
   a. Change line 90 so that the engine runs in sequential mode.

      ```
      calculateMove(myTic, playerOne, false, true);
      ```

b. Run the project.

The output will show the details of a number of games, the last one of which is won by the tictactoe engine.

```
IN_PROGRESS
 O X O -
 O O X O
 - - X X
 X X O O
Before checking move
The list: [TIE_GAME, TIE_GAME, PLAYER_WINS]
Before beginning calculations: IN_PROGRESS

Moves remaining before testing next move: 1



IN_PROGRESS
 O X O -
 O O X O
 O X X X
 X X O O
Before checking move
The list: [PLAYER_WINS]

PLAYER_WINS
 O X O X
 O O X O
 O X X X
 X X O O

*** Game completed ***

Took 1.2079595 seconds
```

You'll see that the games, especially the winning one tend to complete in a little over a second.

3. Run the Tictactoe program in sequential mode.

   a. Change line 90 so that the engine runs in parallel mode.

   ```
   calculateMove(myTic, playerOne, true, true);
   ```

   You will see that typically games complete somewhat quicker, usually under a second though the times may vary on separate runs . On a standalone machine with four cores it's more than twice as fast in parallel mode.

This example also shows an example that uses a custom Spliterator, `TicTacToeSpliterator`. For the functionality currently in the program, there is no real advantage to using a custom `Spliterator`, but in a more sophisticated example where you may wish to control exactly how deep to calculate you may need to do this. For example, in many games, exhaustive calculation of every possibility (as used here) is not possible. Even a 5 X 5 tictactoe game would make exhaustive search quite impossible.

In the next practice you see the creation of a custom `Spliterator` for a custom datatype. This is a typical reason why you might want to create a custom `Spliterator`.

Practice 16-4: Examine a Custom Spliterator

Overview

In this practice, you add another module named `main` so that the `competition` module represents the API of the application and the `main` module calls that API. You also look at the module graph for the application and see how transitive dependencies might improve the design.

Assumptions

Tasks

1. Examine the code that uses the custom `Spliterator`.
   a. Open the project TreeExample in the folder `/home/oracle/labs/16_CustomStreams/examples`.
   b. Open the file Test.java
   c. Look at the code. It does the following:
      1) Creates a tree of nodes. The tree can contain elements of any type but the sample data is of type `String` so the `createSampleData` method creates a `Tree<String>`.
      2) Prints out the tree in outline mode so you can see the sample data. The tree has been set up as a simple organization structure.
      3) Creates a custom `Spliterator` by passing the `tree` to the custom `Spliterator` constructor.
      4) Creates a `Stream` from the `Spliterator`. Note the second parameter, `false`, indicates the `Spliterator` should not support parallel operations.
      If this custom tree data type was intended for wide use, you would probably put this code into the `Tree` type so that a user could call a static `stream` or `parallelStream` method to get the `Stream`.

   The remainder of the code is just normal code you'd use to work with any Stream.
2. Run the code in sequential mode. You should see a display of the entire organization followed by a listing of the non-manager employees.

```
'- Owner
     |- Head of Installation Dept
     |    |- Manager
     |    |    |- Fitter 1
     |    |    |- Fitter 2
     |    |    '- Fitter 3
     |    |- Estimator
     |    |    '- Scheduler
     |    '- Custom Fitter
```

```
     |- Head of Sales Dept
     |   |- North and East Sales
     |   '- South Sales
     '- Head of Manufacturing Dept
         |- Head of Design
         |- Manufacture Supervisor
         |   |- Machine Operator 1
         |   |- Machine Operator 2
         |   '- Machine Operator 3
         '- Repairs and Upkeep
Fitter 1
Fitter 2
Fitter 3
Scheduler
Custom Fitter
North and East Sales
South Sales
Head of Design
Machine Operator 1
Machine Operator 2
Machine Operator 3
Repairs and Upkeep
```

3. Run the code in parallel mode.
   a. Make the operation parallel by changing `false` to `true` in the `stream` method of `StreamSupport`.
   b. Run the project again.

      Notice that the output is the same, except that the order is quite different. This is because it's running in parallel and the employee printed could be from any of the sub-Spliterators.
4. Examine the custom `Spliterator`.
   a. Open `NTreeSpliterator`. Don't worry about understanding the details of the code. The point here is just to look at the major methods and see what they do.
   b. Examine the following methods:

      `tryAdvance`

      It must go forward through the tree by one Node on each call, so it keeps track of the current node (`currNode`)

      `trySplit`

      This is a little more complex, but its purpose is to split the tree into subtrees so that each subtree can be traversed separately by `tryAdvance`.

5. Determine how many splits are occurring.
   a. In the trySplit method, uncomment the code:
      ```
      System.out.println(">>>> Splitting at: " + myNode);
      ```
   b. Now run the program again. You'll see that the tree is being turned into subtrees with the following root nodes:
      Manager
      Estimator
      Manufacture Supervisor

That's the end of this practice.

Practices for Lesson 17: Java I/O Fundamentals

Practices for Lesson 17: Overview

Practices Overview

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

Practice 17-1: Writing a Simple Console I/O Application

Overview

In this practice, you will write a simple console-based application that reads from and writes to the system console. In NetBeans, the console is opened as a window in the IDE.

Tasks

1. Open the project `FileScanner17-01Prac`.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/17-IO_Fundamentals/practices/practice1`.
   c. Select `FileScanner17-01Prac` and select the "Open as Main Project" check box.
   d. Click the Open Project button.
2. Open the file `FileScanInteractive.java`.

   Notice that the class has a method called `countTokens` already written for you. This method takes a String `file` and String `search` as parameters. The method will open the file name passed in and use an instance of a `Scanner` to look for the search token. For each token encountered, the method increments the integer field `instanceCount`. When the file is exhausted, it returns the value of `instanceCount`. Note that the class rethrows any `IOException` encountered, so you will need to be sure to use this method inside a try-catch block.
3. Code the `main` method to check the number of arguments passed.

   The application expects at least one argument (a string representing the file to open). If the number of arguments is less than one, exit the application with an error code (-1).
   a. The `main` method is passed an array of Strings. Use the `length` attribute to determine whether the array contains less than one argument.
   b. Print a message if there is less than one argument, and use `System.exit` to return an error code. (-1 typically is used to indicate an error.) For example:

```
if (args.length < 1) {
    System.out.println("Usage: java FileScanInteractive <file to
search>");
    System.exit(-1);
}
```

4. Save the first argument passed into the application as a `String`.

```
String file = args[0];
```

5. Create an instance of the `FileScanInteractive` class. You will need this instance to call the `countTokens` method.

```
FileScanInteractive scan = new FileScanInteractive ();
```

6. Open the system console for input using a buffered reader.
   a. Use a `try`-with-resources to open a `BufferedReader` chained to the system console input. (Recall that `System.in` is an input stream connected to the system console.)
   b. Be sure to add a catch statement to the try block. Any exception returned will be an `IOException` type. For example:

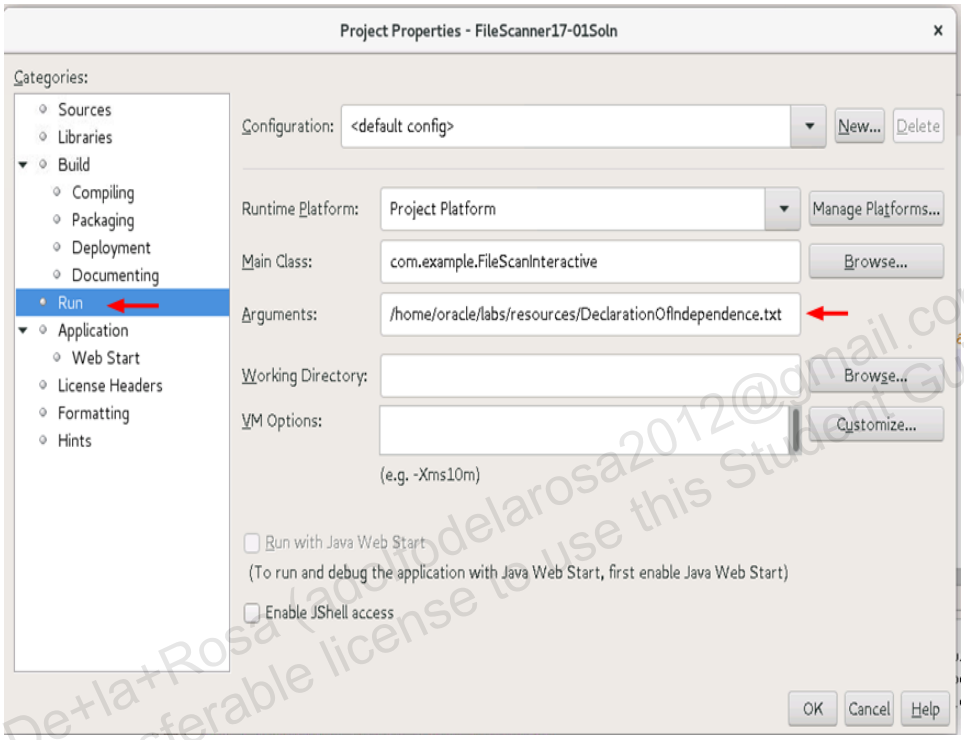
```
try (BufferedReader in =
    new BufferedReader(new InputStreamReader(System.in))) {

} catch (IOException e) { // Catch any IO exceptions.
    System.out.println("Exception: " + e);
    System.exit(-1);
}
```

   c. In the try block that you created, add a while loop. The while loop should run until a break statement. Inside the while loop, read from the system console into a string until the string "q" is entered on the console by itself.
   Note: You can use `equalsIgnoreCase` to allow your users to enter an upper- or lowercase "Q." Also the `trim()` method is a good choice to remove any whitespace characters from the input.
   d. If the string read from the console is not the terminate character, call the `countTokens` method, passing in the file name and the search string.
   e. Print a string indicating how many times the search token appeared in the file.
   f. Your code inside the try block should look something like this:

```
String search = "";
System.out.println ("Searching through the file: " + file);
while (true) {
    System.out.print("Enter the search string or q to exit: ");
    search = in.readLine().trim();
    if (search.equalsIgnoreCase("q")){
     break;
    }
    int count = scan.countTokens(file, search);
    System.out.println("The word \"" + search + "\" appears "
                       + count + " times in the file.");
}
```

   g. Add any missing import statements.
7. Save the `FileScanInteractive` class.

8. If you have no compilation errors, you can test your application by using a file from the resources directory.
   a. Right-click the project and select Properties.
   b. Select Run from the Categories column.
   c. Enter the name of a file to open in the Arguments text box, for example: `/home/oracle/labs/resources/DeclarationOfIndependence.txt`
   d. Click OK.



   e. Run the application and try searching for some words like `when`, `rights`, and `free`. Your output should look something like this:

```
Searching through the file:
/home/oracle/labs/resources/DeclarationOfIndependence.txt
Enter the search string or q to exit: when
The word "when" appears 3 times in the file.
Enter the search string or q to exit: rights
The word "rights" appears 3 times in the file.
Enter the search string or q to exit: free
The word "free" appears 4 times in the file.
Enter the search string or q to exit: q
BUILD SUCCESSFUL (total time: 16 seconds)
```

Practice 17-2: Working with Files

Overview

In this practice, read text files and the `lines` method.

Assumptions

You have completed the lecture portion of this lesson and the previous practice. A text excerpt from the play Hamlet has been provided you as a test file in the root directory of the project. The contents of the files are as follows.

```
                Enter Rosencrantz and Guildenstern.

  Pol. Fare you well, my lord.
  Ham. These tedious old fools!
  Pol. You go to seek the Lord Hamlet. There he is.
  Ros. [to Polonius] God save you, sir!
                                                Exit [Polonius].
  Guil. My honour'd lord!
  Ros. My most dear lord!
  Ham. My excellent good friends! How dost thou, Guildenstern? Ah,
    Rosencrantz! Good lads, how do ye both?
  Ros. As the indifferent children of the earth.
  Guil. Happy in that we are not over-happy.
    On Fortune's cap we are not the very button.
  Ham. Nor the soles of her shoe?
  Ros. Neither, my lord.
  Ham. Then you live about her waist, or in the middle of her
    favours?
  Guil. Faith, her privates we.
  Ham. In the secret parts of Fortune? O! most true! she is a
    strumpet. What news ?
  Ros. None, my lord, but that the world's grown honest.
  Ham. Then is doomsday near! But your news is not true. Let me
    question more in particular. What have you, my good friends,
    deserved at the hands of Fortune that she sends you to prison
    hither?
  Guil. Prison, my lord?
  Ham. Denmark's a prison.
  Ros. Then is the world one.
  Ham. A goodly one; in which there are many confines, wards, and
    dungeons, Denmark being one o' th' worst.
  Ros. We think not so, my lord.
  Ham. Why, then 'tis none to you; for there is nothing either good
    or bad but thinking makes it so. To me it is a prison.
  Ros. Why, then your ambition makes it one. 'Tis too narrow for
your
    mind.
  Ham. O God, I could be bounded in a nutshell and count myself a
    king of infinite space, were it not that I have bad dreams.
  Guil. Which dreams indeed are ambition; for the very substance of
    the ambitious is merely the shadow of a dream.
  Ham. A dream itself is but a shadow.
```

Tasks

1. Open the LambdaFiles17-02Prac project.
   - Select File > Open Project.
   - Browse to `/home/oracle/labs/17-IO_Fundamentals/practices/practice2`.
   - Select `LambdaFiles17-02Prac` and click Open Project.
2. Edit the `P01BufferedReader` class to perform the steps that follow.
3. Using a `BufferedReader` and a stream, read in and print out the `hamlet.txt` file.
4. The output should look like the original text provided above.
5. Edit the `P02NioRead` class to perform the steps that follow.
6. Using the `Path`, `File`, and `Files` classes and a stream to read and print the contents of the `hamlet.txt` file.
7. The output should look like the original text provided above.
8. Edit the `P03NioReadAll` class to perform the steps that follow.
9. Using the NIO features and streams, read the contents of the `hamlet.txt` file into an `ArrayList`.
10. Filter and print out the lines for Rosencrantz for example: `String.contains("Ros.")`. The output should look similar to the following:

```
=== Rosencrantz ===
  Ros. [to Polonius] God save you, sir!
  Ros. My most dear lord!
  Ros. As the indifferent children of the earth.
  Ros. Neither, my lord.
  Ros. None, my lord, but that the world's grown honest.
  Ros. Then is the world one.
  Ros. We think not so, my lord.
  Ros. Why, then your ambition makes it one. 'Tis too narrow for
your
```

11. Filter and print out the lines for Guildenstern `("Guil.")`. The output should look similar to the following:

```
=== Guildenstern ===
  Guil. My honour'd lord!
  Guil. Happy in that we are not over-happy.
  Guil. Faith, her privates we.
  Guil. Prison, my lord?
  Guil. Which dreams indeed are ambition; for the very substance of
```

12. Edit the `P04NioReadAll` class to perform the steps that follow.
13. Using the NIO features and streams, read the contents of the `hamlet.txt` file into an `ArrayList`.

14. Filter and print out the word "lord". Print a count of the number of times the word occurs. The output should look similar to the following:

```
=== Lord Count ===
lord.
lord!
lord!
lord.
lord,
lord?
lord.
Word count: 7
```

15. Filter and print out the word "prison". Print a count of the number of times the word occurs. The output should look similar to the following:

```
=== Prison Count ===
prison
prison.
prison.
Word count: 3
```

Practices for Lesson 19: Building Database Applications with JDBC

Practices for Lesson 19: Overview

Practices Overview

In these practices, you will work with the JavaDB (Derby) database, creating, reading, updating, and deleting data from a SQL database by using the Java JDBC API.

Practice 19-1: Working with the Derby Database and JDBC

Overview

In this practice, you will start the JavaDB (Derby) database, load some sample data using a script, and write an application to read the contents of an employee database table and print the results to the console.

Tasks

1. Create the Employee Database by using the SQL script provided in the resource directory.
   Perform the following steps in NetBeans:
   a. Click Services tab.
   b. Expand the Databases folder.
   c. Right-click JavaDB and select Start Server.
   d. Right-click JavaDB again and select Create Database.
   e. Enter the following information:

Window/Page Description	Choices or Values
Database Name	EmployeeDB
User Name	tiger
Password	scott
Confirm Password	scott

   f. Click OK.
   g. Right-click the connection that you created: `jdbc:derby://localhost:1527/EmployeeDB[tiger on TIGER]` and select Connect.
   h. Select File > Open File.
   i. Browse to /`home/oracle/labs/resources` and open the `EmployeeTable.sql` script. The file will open in a SQL Execute window.
   j. Select the connection that you created from the drop-down list and click the Run-SQL icon  or press Ctrl-Shift-E to run the script.
   k. Expand the `EmployeeDB` connection. You will see that the `TIGER` schema is now created. Expand the `TIGER` Schema, expand Tables, and then expand the table `Employee`.
   l. Right-click the connection again and select Execute Command to open another SQL window. Enter the command:
   ```
   select * from Employee
   ```
   and click the Run-SQL icon to see the contents of the `Employee` table.

2. Open the `SimpleJDBC19-01Prac` Project and run it.
   a. Select File > Open Project.
   b. Select /`home/oracle/labs/19-JDBC/practices/practice1/SimpleJDBC19-01Prac`.
   c. Click Open Project.
   d. Expand the Source Packages and look at the `SimpleJDBCExample.java`
   e. Run the project: Right-click the project and select Run, or click the Run icon, or press F6.
   f. You should see all the records from the Employee table displayed.

Note: In case you get a broken reference link to Java DB driver error, perform the following steps:

1) Right-click on the project and select properties.
2) In the categories column select Libraries.
3) Click Add Library and select Java DB Driver from the Available libraries.
4) Click Add Library.
5) Click OK.

3. Add a SQL command to add a new Employee record.
   a. Modify the `SimpleJDBCExample` class to add a new Employee record to the database.
   b. The syntax for adding a row in a SQL database is:
      `INSERT INTO <table name> VALUES (<column 1 value>, <column 2 value>, ...)`
   c. Use the Statement `executeUpdate` method to execute the query. What is the return type for this method? What value should the return type be? Test to make sure that the value of the return is correct.
   d. Your code may look like this:

```
query = "INSERT INTO Employee VALUES (400, 'Bill',
'Murray','1950-09-21', 150000)";
if (stmt.executeUpdate(query) != 1) {
    System.out.println ("Failed to add a new employee record");
}
```

Note: If you run the application again, it will throw an exception, because this key already exists in the database.

Practices for Lesson 20: Localization

Practices for Lesson 20: Overview

Practices Overview

In these practices, you create a date application that is similar to the example used in the lesson. For each practice, a NetBeans project is provided for you. Complete the project as indicated in the instructions.

Practice 20-1: Creating a Localized Date Application

Overview

In this practice, you create a text-based application that displays dates and times in a number of different ways. Create the resource bundles to localize the application for French, Simplified Chinese, and Russian.

Assumptions

You have attended the lecture for this lesson. You have access to the JDK 11 API documentation.

Summary

Create a simple text-based date application that displays the following date information for today:

- Full date
- Long date
- Short date
- Medium date/time
- Medium time

Localize the application so that it displays this information in Simplified Chinese and Russian. The user should be able to switch between languages.

The application output in English is shown here.

```
=== Date App ===
Full Date is: Tuesday, June 17, 2014
Long Date is: June 17, 2014
Short Date is: 6/17/14
Medium Date and Time is: Jun 17, 2014 10:51:09 AM
Medium Time is: 10:51:09 AM

--- Choose Language Option ---
1. Set to English
2. Set to French
3. Set to Chinese
4. Set to Russian
q. Enter q to quit
Enter a command:
```

Tasks

1. Open the `Localized20-01Prac` project in NetBeans.
   a. Select File > Open Project.
   b. Browse to `/home/oracle/labs/20-Localization/practices/practice1`.
   c. Select `Localized20-01Prac` and click Open Project.
2. Expand the project directories.

3. Edit the `DateApplication.java` file.
4. Open the `MessagesText.txt` file found in the `practices` directory for this practice in a text editor.
5. Create a message bundle file for Russian text named `MessagesBundle_ru_RU.properties`.
   - Right-click the project and select New > Other > Other > Properties File.
   - Click Next.
   - Enter `MessagesBundle_ru_RU` in the File Name field.
   - Click Browse.
   - Select the `src` directory.
   - Click Select Folder.
   - Click Finish.
   - Paste the localized Russian text into the file and save it.
6. Create a message bundle file for Simplified Chinese text named `MessagesBundle_zh_CN.properties`.
   - Right-click the project and select New > Other > Other > Properties File.
   - Click Next.
   - Enter `MessagesBundle_zh_CN` in the File Name field.
   - Click Finish.
   - Paste the localized Simplified Chinese text into the file and save it.
7. Update the code that sets the locale based on user input.

```
    public void setEnglish(){
        currentLocale = Locale.US;
        messages = ResourceBundle.getBundle("MessagesBundle",
currentLocale);
    }

    public void setFrench(){
        currentLocale = Locale.FRANCE;
        messages = ResourceBundle.getBundle("MessagesBundle",
currentLocale);
    }

    public void setChinese(){
        currentLocale = Locale.SIMPLIFIED_CHINESE;
        messages = ResourceBundle.getBundle("MessagesBundle",
currentLocale);
    }

    public void setRussian(){
```

```
        currentLocale = ruLocale;
        this.messages =
ResourceBundle.getBundle("MessagesBundle", currentLocale);
    }
```

8. Add the code that displays the date information to the `printMenu` method.

```
    public void printMenu(){
      pw.println("=== Date App ===");

      // Full Date
      df =
DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL).withLocale(c
urrentLocale);
      pw.println(messages.getString("date1") + " " +
today.format(df));

      // Long Date
      df =
DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG).withLocale(c
urrentLocale);
      pw.println(messages.getString("date2") + " " +
today.format(df));

      // Short Date
      df =
DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT).withLocale(
currentLocale);
      pw.println(messages.getString("date3") + " " +
today.format(df));

       // Medium Date/Time
      df =
DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM).withLo
cale(currentLocale);
      pw.println(messages.getString("date4") + " " +
today.format(df));

       // Medium Time
      df =
DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM).withLocale
(currentLocale);
      pw.println(messages.getString("date5") + " " +
today.format(df));

        pw.println("\n--- Choose Language Option ---");
        pw.println("1. " + messages.getString("menu1"));
```

```
        pw.println("2. " + messages.getString("menu2"));
        pw.println("3. " + messages.getString("menu3"));
        pw.println("4. " + messages.getString("menu4"));
        pw.println("q. " + messages.getString("menuq"));
        System.out.print(messages.getString("menucommand") + "
");
}
```

9. Run the `DateApplication.java` file and verify that it operates as described.