myCareerTEC
Technology Education Center

Agile Development Using Java 8 Features: Unit Testing, Design Patterns, and RESTful Services

Exercise Manual

ROITraining
Maximize Your Training Investment

Fidelity INVESTMENTS

This page intentionally left blank.

Table of Contents

This page intentionally left blank.

Exercise 1.1: Working with Lambda Expressions

Overview

In this short exercise, you will gain experience with writing simple lambda expressions.

1. In your development environment, create a new Java project named `intro_lambdas`.
2. In this project, add a new Java class named `Driver`.
3. In the class `Driver`, add a `main` method as the entry point of execution for your program.
4. In the `main` method, declare an array of strings named `currencies` and initialize it with the following values:

   "USD", "JPY", "EUR", "HKD", "INR", "AUD"

5. Now, using internal iteration and a lambda expression, print all the values of the `currencies` to the console.
6. Sort the `currencies` in ascending order, using a lambda expression, and print the sorted values to the console.
7. Now sort the `currencies` in descending order, using a lambda expression, and print the sorted values to the console.

Additional Exercise (If Time Permits)

8. In the `main` method above, in a separate thread of execution, print out the numbers 1 to 10 to the console. You should use a lambda expression for the thread implementation.

This page intentionally left blank.

Exercise 1.2: Working with Functional Interfaces

Overview

In this exercise, you will gain further experience of writing lambda expressions and using functional interfaces. You will use a predefined set of domain classes that represent simplified orders.

1. Import the project `functional_interfaces` into your development environment.
2. Examine the domain classes in the package domain. Make sure you are familiar with these and ask your instructor if you are not.
3. Now open the class `Driver` in the package `main`. This is where you will perform the required tasks. Note the List of orders that has been created for you.
4. In the class `Driver`, write a static method named `printMatchedOrders` that receives two parameters:
    a. A list of orders: `List<Order>`
    b. A predicate: `Predicate<Order>`
5. The method `printMatchedOrders` should iterate over the list of orders and print to the consoles those that pass the predicate.
6. In the main method, call the `printMatchedOrders` method passing in the list of orders and also a predicate that selects only buy side orders.
7. Repeat Step 6, but for sell side orders only.
8. The Functional interface `Function<T,R>` represents a unary function. It performs a function on a single argument of type T and returns a result of type R.
9. In your class `Driver`, define an implementation of the functional interface in a variable named `averageOrder` that receives a list of orders and returns the average amount of the orders.
10. In main, call the a`verageOrder` implementation and print the result to the console.

Additional Exercise (If Time Permits)

11. Print all the orders to the console in ascending amount order. Make sure you use functional interfaces.
12. Now print all the orders to the console in descending amount order. Make sure you use functional interfaces.

Exercise 2.1: Working with Streams

Overview

In this exercise, you will gain experience of working with streams. You will use the same project as the last exercise and the same domain classes.

1. In the main method, write a single statement that will create a stream from the list of orders. Using internal iteration, print to the console only those that are buy side orders.
2. Now repeat the above statement, but sort the orders by amount in ascending order before printing them to the console.
3. Write a single statement that, using the `map()` stream operation, will print just the amount of each order in the stream.
4. Write a statement that will calculate the total amount of all orders using a stream. Then print the total amount to the console.
5. Write a statement that will count the number of orders in the orders stream. Print the count to the console.
6. From the stream of orders, create a new list of orders that contains just buy side orders.

Additional Exercise (If Time Permits)

7. Declare an integer stream in the range 1 to 10 and print its values to the console. Investigate the difference between `range()` and `rangeClosed()` for generating the range.
8. Write some code that will generate a stream of `Stream<String>` from a text file as source. Use the stream and internal iteration to print the file contents line by line to the console.

This page intentionally left blank.

Exercise 2.2: Further Working with Streams

Overview

In this exercise, you will gain further experience of working with streams.

1. In your development environment, open the Java project named `further_streams`.
2. In this project, open the Java class named `Driver`.
3. You will now experiment with the order of intermediate operations, in particular `map()` and `filter()`.
4. In the `main` method, declare a stream of strings containing lowercase "a" through to "e". Add two intermediate operations, `map()` and `filter()`, in this order. Map should translate the string to uppercase and filter should only return uppercase "A"s. Add a terminal operation `forEach()` that prints each string in the resultant stream to the console prefixed by `forEach()`. In the `map()` and `filter()` operations, print the words "map" and "filter" with the strings they are processing respectively. Your code may be similar to that below.

```
Stream.of("a", "b","c","d","e")
      .map(s-> {
            System.out.println("map : " + s);
            return s.toUpperCase();
      })
      .filter(s-> {
            System.out.println("filter: " + s);
            return s.startsWith("A");
      })
      .forEach(s-> System.out.println("forEach: " +s));
```

5. Predict the output of the above program. Now execute your program and verify your prediction was correct.
6. Now repeat the above code, but swap the order of the `map()` and `filter()` operations. You will need to change the filter to filter on lowercase "a" now.
7. Predict the output of the program now. Run the program and verify your prediction is correct. If you are not sure why the output is as it shows, ask your instructor.

8. It is important when working with streams and intermediate operations to chain them in the correct order. For large streams, chaining them in the wrong order can add a very high overhead to the amount of processing that is performed unnecessarily.
9. In the class `Driver`, look at the `createOrders()` method provided. Familiarize yourself with the classes used as this method provides the objects that you will now process in the rest of this exercise.
10. In `main`, make a call to `createOrders` so that you have a List<Order> available for processing.
11. Now write the code that will create a stream of `Order` objects and group the stream of orders by their `Side` property (Side is either a buy or sell) storing the result in a `Map`. The key of the map will be the Side and the value a List<Order>.
12. Print the orders to the console grouped by side.
13. Now using the List<Order> and streams and lambdas, calculate and print out the total amount of all orders per currency.
14. Using the List<Order> and streams and lambdas, calculate and print out the average value of all orders.

Additional Exercise (If Time Permits)

15. Using the List<Order> and streams and lambdas, calculate and print out the summary statistics of the amount of the orders. The summary statistics includes maximum value, average, and minimum values.
16. Using the List<Order> and streams and lambdas, calculate ad print out the maximum value of an order per individual currency.

Exercise 6.1: Java Design Patterns

Objectives

To introduce working with design patterns in Java.

Overview

In this exercise, you will create a new Java project in Eclipse that has Robot classes and will use the Observer and Composite design pattern.

1. In Eclipse, create a new Java project named `robots`.
2. In your project, create a new interface named `Robot` that has one method void `move(int x, int y)`.
3. In your project, define a new class named `SimpleRobot` that implements the Robot interface. Have a constructor that takes a String that represents the robot's name. In the move method, just output the message "`Robot xxxxx is moving x = aa and y=bb`", where xxxxx is the robot name, aa = the x distance, and bb the y distance being moved.
4. Create a new class named `Driver` with a main method. In main, create a `SimpleRobot` and ask it to move. Verify your code works as expected.
5. You will now use the Observer design pattern. Your robots will become Observable objects that will notify any registered Observers that they have moved. Make your SimpleRobot observable according to the Observer design pattern. Use `the java.util.Observable` class.
6. Write a new class `RobotObserver` that implements an Observer interface compatible with your observable robot from Step 5. Use the `java.util.Observe`r interface. The robot observer should output a message every time a robot it is observing moves. It should also keep track of the total distance in x and y directions any observed robot moves.
7. In Driver, create a `RobotObserver` and register it with the SimpleRobot. Move the SimpleRobot a few times and make sure the RobotObserver is working correctly. Is the observer keeping track of the total x and y distances moved correctly?

Bonus

8. Define a composite robot that can be built from many individual robots. When the composite is asked to move, all the individual robots internally should move.
9. In the class `Driver` create three SimpleRobots and then a composite robot that is built from the three SimpleRobots. Ask the composite to move and make sure that it works as expected.
10. Make your composite robot observable and register your robot observer on the composite. Move the composite and make sure your observer is notified of the movement correctly.

Exercise 6.2: Implementing the Proxy Design Pattern

Objectives

To implement the proxy design pattern using Java's dynamic proxy mechanism.

Overview

In this exercise, you will build a factory class that creates a robot. You will also define an annotation that when added to a robot will trigger that robot to have its move method profiled. This will be achieved by a proxy that the factory creates.

1. In the provided `robot_proxy` project, define an annotation named `Profile` that is available at runtime and can be applied to classes and interfaces. Place your annotation in the package named `profile`.
2. In the package `profile`, define a class named `Profiler`. This class is going to be the implementation of a method profiler that is used by a proxy. The method should implement the `InvocationHandler` interface. In the invoke method, the code should take a snapshot of the system time call invoke on the target method. When the target method returns, then the handler should take another snapshot of the system time and record to the console the method that has been called and the time it took to execute.
3. Open the file `RobotFactory.java` in the package `robots.factory`. In the `createRobot()` method, modify it so that once the SimpleRobot is created, using reflection, determine if the class has the `@Profile annotation`. If it does, create a dynamic proxy that uses the invocation handler defined in Step 3 above. Return the proxy from the factory.
4. Run the main method in the class `Driver` in package main and verify the program executes. Now add the `@Profile` annotation to the `SimpleRobot` class and re-run the program. You should see your profiler output on the console.
5. The main method did not need to change to use the proxy.

Bonus

6. Modify your `@Profile` annotation to accept a parameter named `timeUnits`. This parameter should have one of two values MILLISECS or SECONDS. The idea is that the user can specify what time unit they would like the profile output to be logged in.
7. Modify your code in `Profiler.java` so that it will output the time to the console in the user-selected units.
8. Make any other changes to the code base you think are required.
9. Run the program again and make sure your output is as expected.