Modern Java Labs

Introduction

The following labs can be downloaded from:
https://github.com/ThoughtFlow/ModernJava

Each lab is modeled as a Java package with *init* and *fin* as sub-packages. The *init* package contains the initial code from which to start and the *fin* package contains the final solution. You may simply modify the code directly in *init* or copy the code in another package then modify it. Modular labs (e.g. lab12, lab13 and lab15) are under the modules directory.

Resist the urge to look at the answers in the *fin* package!

Lab 1: The lambda form

Objective: test your understanding of how to implement lambdas.

Use these six pre-created interfaces in lab01.init:

1. Interface1.java
   public void printSquareOfA(int a);
2. Interface2.java

   public int getSquareOfA(int a);
3. Interface3.java

   public int getAxB(int a, int b);
4. Interface4.java

   public void consume(int a);
5. Interface5.java

   public double getPi();
6. Interface6.java

   public Double create(double x);

Implement these lambdas:

1. Implement a lambda squares the parameter value and prints the result conforming to Interface1.
2. Implement a lambda that will return the square of the parameter value using interface 2.
3. Implement a lambda that will return the multiplication of the two parameter values using Interface3.
4. Implement a lambda that will return the value of pi as a double using Interface5.
5. Implement a lambda that returns the square of the parameter value itself conforming to Interface2. Provide a default method that pretty prints a message around getSquareOfA().
6. Implement a static method in Interface3 that multiplies the two numbers. Call the method and print the result.
7. Implement a static method reference for interface4 and use calculateAndConsume with Interface2 to square the value of the parameter and print the results.
8. Implement a lambda that uses the static method Double.valueOf() to implement interface6.

Lab 2: Standard functional interfaces

Objective: test your understanding of how to use the standard functional interfaces.

1. Replace functional interfaces 1 through 5 in lab01.init with their standard functional interface equivalent. Test it with LambdaTest.
2. Use Functions to create a series of functions that:
    a. Double, square, cube then negate a number using andThen
    b. Double, square, cube then negate a number using compose
    c. Use FunctionalComposition in lab01.init as a starting point.
3. Use functional composition to implement lambda that will determine if a student has passed a course based on an array of Double representing test scores. A pass is calculated with these rules:
    a. All test scores must be > 60%
    b. Average test score must yield a B average (>= 80%)
    c. If A and/or B are false, a pass is given if last exam was perfect
    d. Must have taken all exams
    e. Use this test data:

```
// True: Passed all
Double[] scores = (Double[]) Arrays.asList(.65, .90, .90, .90, .90, .90).toArray();
// False: Not all passed
scores = (Double[]) Arrays.asList(.59, .90, .90, .90, .90, .90).toArray();
// False: C average - fail
scores = (Double[]) Arrays.asList(.70, .70, .70, .70, .70, .70).toArray();
// True: C average but aced last
scores = (Double[]) Arrays.asList(.70, .70, .70, .70, .70, 1d).toArray();
// True: Failed first but scored perfect on last
scores = (Double[]) Arrays.asList(.59, .90, .90, .90, .90, 1d).toArray();
// False: same as previous but missed a test
scores = (Double[]) Arrays.asList(.59, .90, .90, .90, 0d, 1d).toArray();
```

4. Use Consumer composition to print all log lines to stdout and lines that contain the word "exception" to stderr (as well as stdout).
5. MessageComposition.java defines five methods: Encrypt/decrypt, encode/decode, and obfuscate. Encrypt and decrypt work together to encrypt and decrypt data while encode and decode convert to and from base64. Obfuscate hides credit card number. Use these predefined functions to create two super functions:
    a. writingFunction obfuscates, encodes and encrypt data
    b. readingFunction decrypts then decode data
    c. Only Implement:
        i. Supplier<Function<List<String>, List<String>>> writingFunction;
        ii. Supplier<Function<List<String>, List<String>>> readingFunction;

Lab 3: Using functionalized collections

Objective: test your understanding of how the newly functionalized collections library in Java 8.

Using this interface:

```
public interface MovieDb {
        /**
         * Adds a movie to the database with the given categories, name and year
         * released.
         *
         * @param categories The set of categories for the new movie.
         * @param name The name of the movie.
         * @param yearReleased The year of release
         */
        void add(Set<Category> categories, String name, Integer yearReleased);

        /**
         * Adds a movie to the database with the given category, name and year
         * released.
         *
         * @param category The category for the new movie
         * @param name The name of the movie.
         * @param yearReleased The year of release
         */
        void add(Category category, String name, Integer yearReleased);

        /**
         * Searches for the given movie title and returns as a Movie record.
         *
         * @param name The name of the movie to search.
         * @return The found movie or null if not found.
         */
        Movie findByName(String name);

        /**
         * Searches by category and returns the list of movies for the given category.
         *
         * @param category The category name to search.
         * @return The list of movies matching the category or an empty list.
         */
        List<String> findByCategory(Category category);

        /**
         * Deletes the movie with the given name.
         *
         * @param name The name of the movie to delete.
         * @return True if found and deleted - false otherwise.
         */
        boolean delete(String name);
}
```

- Convert each method in FunctionalMovieDb using the functionalized collection methods of sets, lists and maps.
- Fill in the methods where indicated by “implement this”.

- Refer to ImperativeMovieDb for an implementation using imperative method.
- Test your implementation with TestMovieDb.

Lab 4: Currying in Java

Objective: test your understanding of currying in Java.

1. Implement this function used for currying and partial application:
   - `Function<String, Function<String, Function<String, String>>> func`
   - Then use partial application to concatenate strings:
     `func.apply("Currying").apply(" is").apply(" great!");`
2. Define a function that multiplies three numbers using currying and partial application.
   - Use the Function functional interface to define *a function that takes a integer and returns a function that takes an integer that returns another function that takes and returns an integer.*
3. Use the currying and partial application techniques to create a function that uses average, best or worst as a statistical method in calculating test scores. Use this type definition as the currying function:

   ```
   Function<GradeCalcType, Function<List<Double>, Double>>
   curryingFunction;
   ```

   The statistical methods are:
   - Average: the average of the test scores is used to determine the grade.
   - Best: only the highest score is used to determine the grade - all others are discarded.
   - Worst, only the lowest score is used to determine the grade - all others are discarded.
   - Use this enum definition:

     ```
     private enum GradeCalcType
     {
         AVERAGE,
         WORST,
         BEST
     }
     ```

   - Use this to test:

     ```
     public static void main(String... args)
     {
       List<Double> scores = Arrays.asList(.65, .75, .85);

       System.out.println(curryingFunction.apply(GradeCalcType.AVERAGE).apply(scores));
       System.out.println(curryingFunction.apply(GradeCalcType.BEST).apply(scores));
     ```

```
  System.out.println(curryingFunction.apply(GradeCalcType.WORST).apply(scores));
}
```

- Use the class CurriedGrading as a starting point.

Lab 5: Garbage collection

Objective: test your understanding of garbage collection settings and profiling.

Use the class Infinite.java in lab05.fin as the application to profile.

1. Configure the GC thusly:
   a. Use the G1 garbage collector
   b. Turn on debug heap profiling and output to file /tmp/GC.log
   c. Set the minimum heap to 100Mb and the max heap to 200Mb

   How many eden (young), survivor and old GC cycles occurred?

   What was the size of each region?
2. Repeat the same exercise with the Epsilon GC.

Lab 6: Using the executor service with callables and futures

Objective: test your understanding of the executor service, callables and futures.

1. Write an application that counts the number of prime numbers in ranges using the ExecutorService:
    - Start with the single-threaded implementation ThreadedPrimeNumberFinder. Re-implement the method countPrimes.
    - Choose the appropriate ExecutorService implementation and mind the pool size.
    - Use submit/invoke, call and future, then shutdown the pool.
    - Each range is 1000 elements.
    - Each range is calculated by different threads using the executor service.
    - Print the number of primes found for all ranges.
    - Use the method Util.isPrime (lab.util package) to determine if a number is prime.
    - There are 78,498 prime numbers between 1 and 1,000,000.
2. Write an application that scrapes the HREF URLs from a given set of html pages, catalogs them inside a map and tallies the occurrence of each distinct URL.
    - Start with LinkScaper and use pre-made methods Util.scrapeHrefs(), Util.catalog() and Util.merge() to scrape the urls, catalog them into a map and merge into one map. The map will be keyed by the stringedUrl. The value will be the occurrences of that url.
    - The method invoke implements a single-threaded version. Use that code to convert to multi-threads
    - Choose the appropriate ExecutorService implementation and mind the pool size.
    - Use submit/invoke, call and future, then shutdown the pool.
    - The list of URLs to scrape is given.

Lab 7: Using Read/Write locks with conditions

Objective: test your understanding of Java's Read/Write locks

Refactor the movie database from lab06 to make it thread-safe:

- Use the class ThreadSafeMovieDb as a starting point.
- Use read/write locks to access the database class attribute in a thread-safe way.
- Look for markers in the code for help in locating areas to protect.
- Use the pre-made TestMovieDb to test your implementation.

Lab 8: Using the Fork/Join framework

Objective: test your understanding of the fork/join framework.

1. The class in lab08.init.ThreadedPrimeNumberCollector creates an array of one million booleans where each element indicates whether or not the number at its index is prime. It then lists the numbers that are prime. It already uses the fork/join framework and extends the RecursiveAction class but does not divide and conquer the processing. Refactor the compute method to:
   - Divide the array in chunks of 1000 elements using fork()
   - Populate each array element with true if its index is prime – false otherwise
   - Rejoin the processing with join().
2. The class in lab08.init.ThreadedPrimeNumberFinder counts the number of primes as a recursive task of type Integer. It already uses the fork/join framework and extends the RecursiveTask class but does not divide and conquer the processing. Refactor the compute method to:
   - Divide the array in chunks of 1000 elements using fork()
   - Count the number of prime numbers in the range
   - Rejoin the processing with join() and aggregate the results.

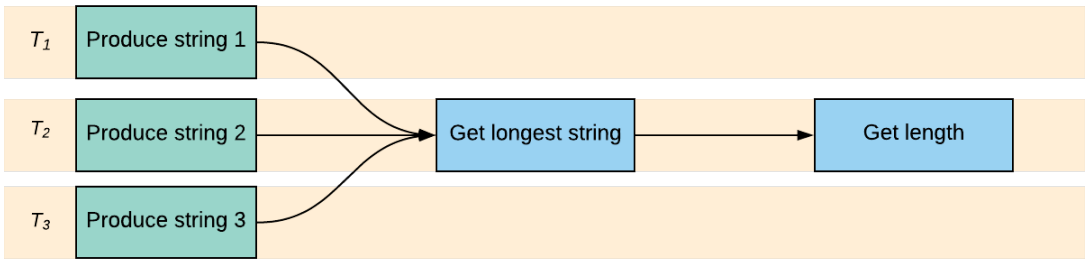
As a bonus, refactor the LinkScraper in lab06 using the Fork/Join framework. Consider how you would:

- Break down the work into smaller chunks
- Break down different kinds of tasks (e.g. scraping urls, cataloging, counting).

Lab 9: Using promises

Objective: test your understanding of promises.

1. Implement a chained set of promises that:
    - Call produceString1, produceString2 and produceString3 asynchronously. Each return a string.
    - Get the longest string of the three.
    - Return the length of that string as an integer.
    - Use PromiseCombiner as a starting point and implement weavePromise().
    - Should return: 10.



2. Re-implement the prime number finder from lab 8 using promises:
    - Start with PromisesPrimeNumberFinder and implement the method createPromise();
3. Add exception handling to the promise:
    - Add exception handling in the promise to handle exceptions. This handler should simply return 0 and continue with the next range.
    - Print an error message but continue anyway.
    - Test with a negative range.
    - Start with PromisesPrimeNumberFinder and implement the method createPromiseWithException();
4. Re-implement the URL scraper using promises:
    - Use Util.scrapeHrefs, Util.catalog and Util.merge within chained promises.
    - Mind the merge being done in parallel (hint: use a concurrentHashMap)

Lab 10: Using streams

Objective: test your understanding and practice thinking in streams.

Use streams to implement these algorithms:

1. Iterate through numbers from 0 to 100:
   - Print out all the even numbers.
   - Then, modify your algorithm to add only odd numbers 0, 100.
   - Then, modify your algorithm to add only odd numbers 0, 100 but remove prime numbers.
   - Then, modify your algorithm to find the smallest int whose factorial is >= 1,000,000
2. Given a list of strings, print each string that is a palindrome:
   - Then, modify your algorithm to return the original word (unstripped).
3. Implement the Fizz Buzz algorithm:
   - Iterate from 1 to 100.
   - Print "Fizz" for every number divisible by 3 and "Buzz" for every number divisible by 5.

Reimplement the Fizz Buzz algorithm in parallel from 1 to 10 million.

Lab 11: Reactive programming

Objective: test your understanding of reactive programming in Java.

ReactiveProcessor is a queue-based program in which a publisher pushes events to a Subscriber using reactive principles. That is, it is aware of its subscriber and can track any dropped messages. The subscriber can also apply back-pressure to the publisher if it publishes too fast.

Using the ReactiveProcessor in lab11.init as a starting point, finish the implementation by supplying a subscriber and publisher:

- The subscriber is a Flow.Subscriber subclass whose parameterized type is an Integer (it will receive integers). The subscriber will call the simulateWork(Integer) method that already exists in the ReactiveProcessor.
- The event publisher is a function that takes a subscriber and returns the list of messages dropped by the subscriber.
- The event publisher then offers integers to the subscriber via SubmissionPublisher.offer().
- It publishes messages from 1 to 1000 and gives the subscriber up to 50ms to process the message. Otherwise, messages are dropped and recorded into a list.
- The SubmissionPublisher is configured with a queue size of 2.
- Ensure that you request integers in the subscriber (or nothing will happen).

Repeat with a publisher whose range is -1 to 10 to simulate an error.

Lab 12: Creating JPMS modules in the IDE

Objective: test your understanding of creating JPMS modules in the IDE.

In the directory modules/lab12.fin, there are two JPMS modules:

- com.lab12.movement
- com.lab12.axle

Use those to create a modular IDE project.

- Create two IDE modules each with its own hierarchy and module-info class. Use the pre-defined module-info class.
- com.lab12.movement is the root module with no dependencies
- com.lab12.axle depends upon movement.

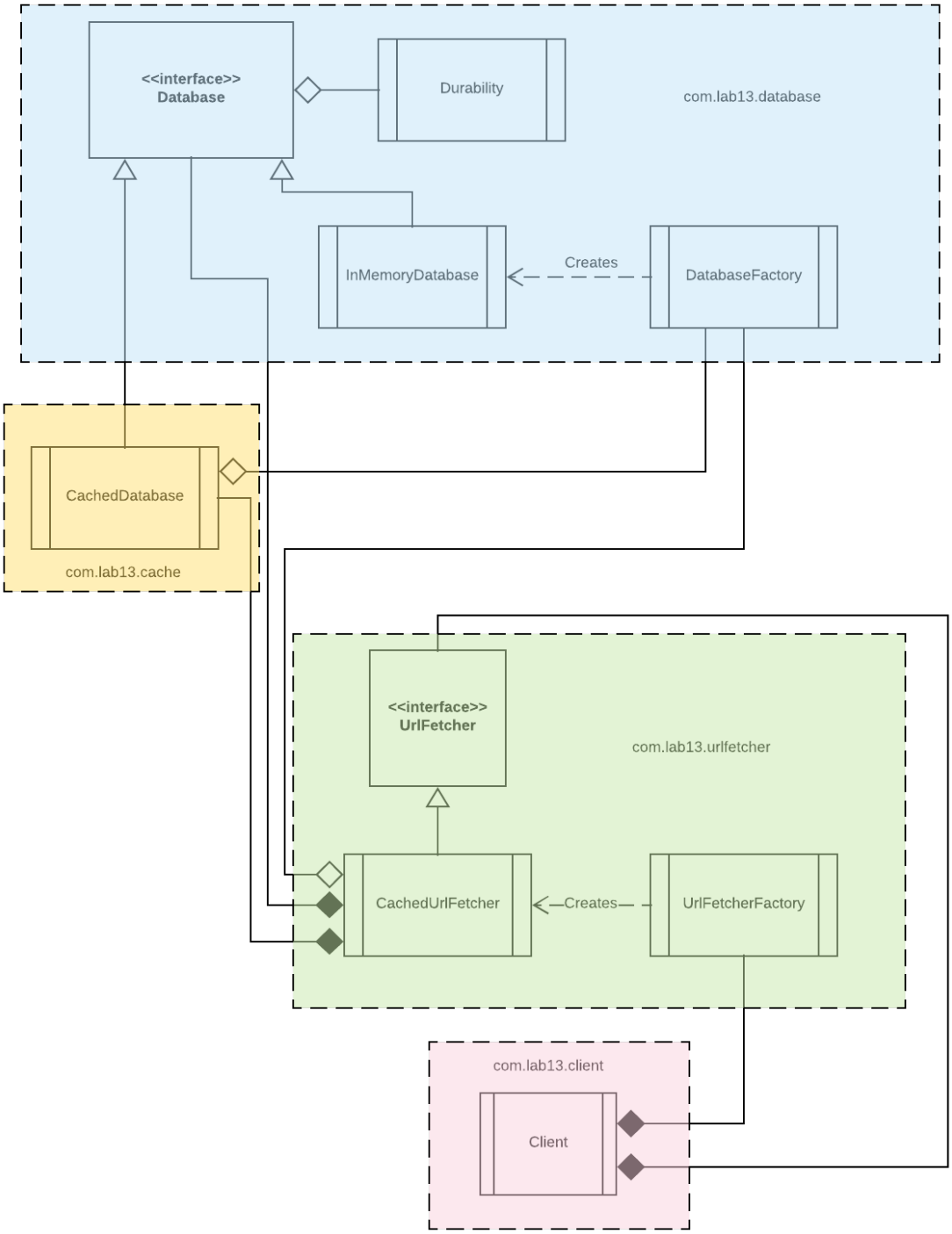
Define the project and run the class com.lab12.axle.api.TestAxle.

What happens if you try to instantiate com.lab12.movement.Rim from Axle?

Lab 13: Modules

Objective: test your understanding of modules.

Consider this class diagram:



Note that:

- com.lab13.database has no dependencies
- com.lab13.cache depends upon com.lab13.database
- com.lab13.urlfetcher depends upon com.lab13.database and com.lab13.cache. It depends upon Durability defined in com.lab13.database, which is exposed in its UlrFetcher.
- com.lab13.client depends upon com.lab13.urlfetcher.

Using the pre-existing code in modules/lab13/init, modularize this application such that:

- Each box is in its own Java module (use the names shown in the diagram) – Create IDE modules.
- Create a module-info class that follows the class diagram dependencies. Expose the least number of classes to outside packages (strong encapsulation).
- Make com.lab13.cache an optional module and UrlFetcher must work with or without it.
- Repackage any class as necessary to meet the stated objectives.

Run the class com.lab13.client.Client to test your configuration:

- Without caching
- With caching.

Lab 14: Create jmod files and jlink distributions

Objective: test your understanding of the jmod and jlink tools

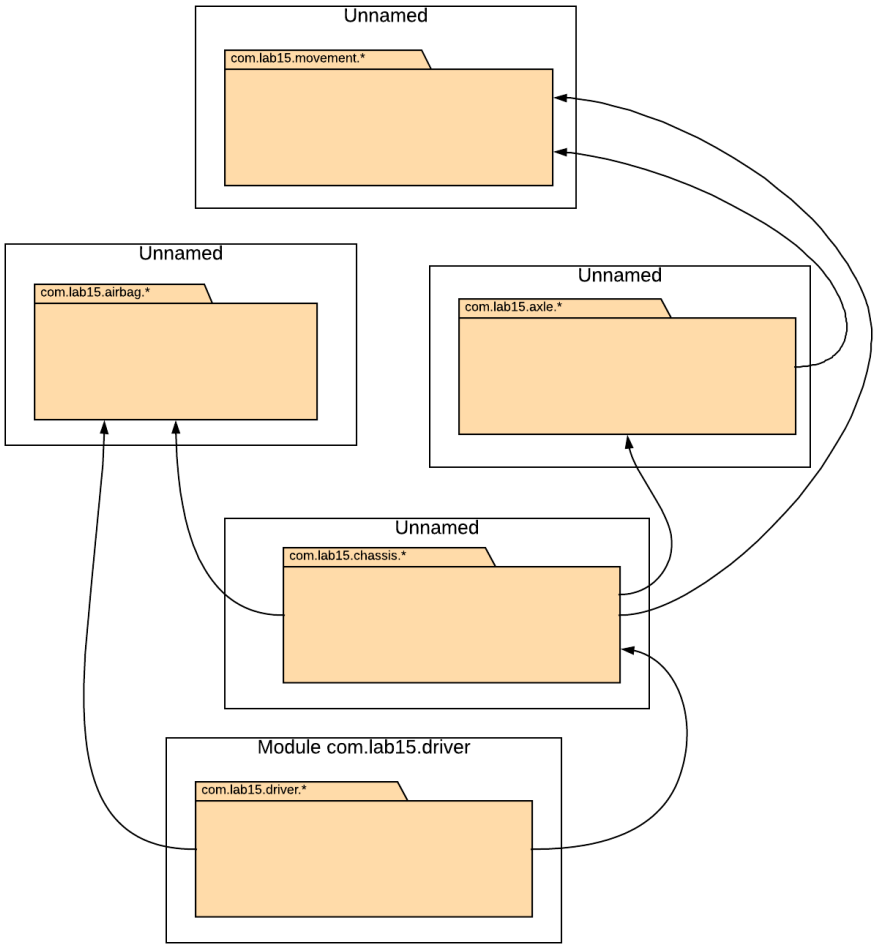
Use the pre-created application in lab14 init, which is a copy of lab13, to create a self-contained image file that requires no JVM installation to run:

- Start with lab14/init/build script. It contains the javac command.
- Create a jmod file for all four modules
- Describe the contents of each jmod file
- Create a jlink distribution with an auto-launch of com.lab14.client/com.lab14.client.Client
- Run the generated jlink auto-run script.
- Package this under lab14.jmods (for the jmod files) and lab14.jlink for the distribution.
- Complete the build and run script.

Lab 15: Modular Jars

Objective: test your understanding of modular jars.

Consider this dependency graph where each package is contained in a jar:



The package com.lab15.driver has already been modularized but will no longer work with the rest of the non-modular application. Budget has been allocated to convert two packages into modules: which two packages would you pick?

Tasks:

- Create the module-info class for module com.lab15.driver
- Create the module-info class for two more packages only
- Use automatic modules for any remaining dependency.
- Use build and run scripts to both build and run the application. (No need to configure the module in the IDE.)

Lab 16: HTTP 2

Objective: test your understanding of the HTTP 2 client library in Java.

Using lab16.init.HttpFunctions as a starting point, complete the following methods using the pre-made templates:

1. Perform an HTTP GET using a String-based body as a response in getBodyAsString.
2. Perform an HTTP GET using a stream-based body as a response in getBodyAsStream
3. Perform an HTTP POST writing the response to a file using postResponseToFile.

Follow the instructions in each method by implementing the Function and Consumer as specified.

Lab 17: Process API

Objective: test your understanding of the process API.

Write a REPL shell that performs commands on processes running on your host. Start with the class ProcessHandlerRepl in lab16.init and fill in the missing methods:

- `start` *`program output`*: Starts a linux program sending output to output.
- `find` *`program`*: Prints the PID of the given program if found
- `kill` *`pid`*: Kills the process with the given PID.
- `stats` *`pid`*: Prints the user under which the given PID is owned and the CPU duration in seconds consumed so far.
- `all`: Prints the PID of all processes running on the host.

The remainder of the program is already written.

Lab 18: JShell

Objective: test your understanding of the JShell REPL engine.

Start JShell and perform this series of commands:

1. Copy the find and stats method from Lab16.fin.ProcessHandleRepl.
2. List the contents of the methods.
3. Import the missing java.time.Duration class.
4. Invoke the method find("jshell") and note the PID.
5. Define the variable PID as a long and initialize it with the PID from the previous step.
6. Edit the stats method and remove the parameter. Hard code the method to always use the PID constant.
7. Save all declared methods and constants to file context.txt
8. Save the command history to file history.txt
9. Reset jshell
10. Reload the saved methods and constants
11. Call stats again
12. Exit out of jshell and list the history.