Advanced Java Labs

Introduction

The following labs can be downloaded from:
https://github.com/ThoughtFlow/AdvancedJavaV2

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

Lab 01: The lambda form

Objective: test your understanding of how to implement lambdas.

Use these five 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 double getPi();
5. Interface5.java

   public boolean isEqualToTen(int a);

Implement these four lambdas:

1. Implement a lambda that squares itself and prints it.
2. Implement a lambda that returns the square of itself.
3. Implement a lambda that multiplies the two numbers.
4. Implement a lambda that returns 3.14.
5. Implement a lambda that returns true if the parameter is 10 – false otherwise.

Lab 02: Functional interfaces & method references

Objective: test your understanding of method references and the @functionalInterface annotation.

Refactor the code from lab 01 (code found in lab02.init):

1. Refactor the five interfaces to utilize the @FunctionalInterface annotation.
2. Refactor to use static, instance, or constructor method references where possible.

Lab 03: Default methods lab

Objective: test your understanding default and static implementations in interfaces:

Refactor the code from lab 01 (code found in lab03.init):

1. Refactor Interface1 to provide a default implementation named: prettyPrintSquareOfA(int a) that prints the square of A with a the text: “The square of “ + a + “ is “. Call the default implementation.
2. Refactor Interface2 to provide a default implementation that returns a stringified square of A. Call the default implementation.
3. Refactor Interface3 to provide a static implementation for get A times B. Call the static implementation.
4. Refactor Interface4 to provide a static implementation for get PI. Call the static implementation.
5. Refactor Interface5 to provide a default implementation for integers not equal to 10. Call the default implementation.

Lab 04: Standard functional interfaces

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

Refactor the code from lab 01 (code found in lab04.init) and replace the custom interfaces 1 to 5 with the standard functional interfaces.

Lab 05: Functional composition

Objective: test your understanding of how to aggregate behavior using functional composition.

1. 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();
```

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
3. Use Consumer composition to print all log lines to stdout and lines that contain the word "exception" to stderr (as well as stdout).
4. 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 06: 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 07: Read/Write locks with conditions

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

1. Implement locking for a queue class:
   - Use QueueLockCondition in the queuelock.init package.
   - Convert from notify/wait with synchronize blocks to read/write locks with signal.
   - Use QueueNotifyWait as a reference.
   - Test using pre-made TestQueue class. Errors will be flagged automatically.
2. 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 08: Using the executor service with callables and futures

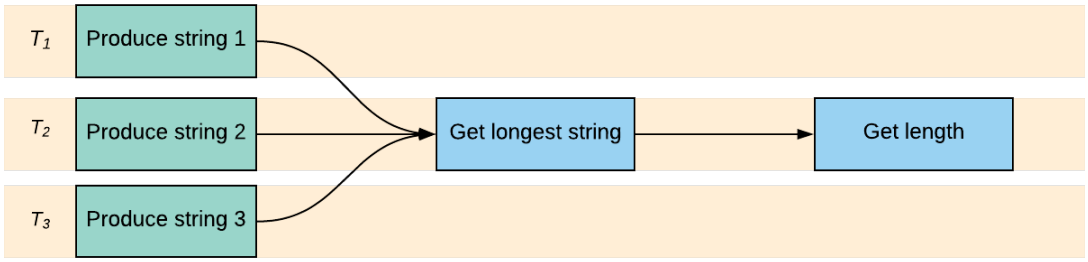
Objective: test your understanding of the executor service, callables and futures.

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 09: Using promises to find prime numbers

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 spliterators to find prime numbers

Objective: test your understanding of spliterators.

Re-implement the solution of lab 8 using spliterators (the divide and conquer strategy):

- Start with SpliteratorPrimeNumberFinder and implement these 2 functions:

```
// Input:  The list of integers to test for primeness
// Output: The list of spliterators of integers representing the range to process
Function<List<Integer>, List<Spliterator<Integer>>> spliteratorCreator;

// Input   The spliterator of integers to process
// Output: The number of primes found in the spliterator
Function<Spliterator<Integer>, Integer> primeCounter;
```

- Create four spliterators in spliteratorCreator.
- Count the number of primes in primeCounter.
- Each spliterator will be wrapped inside a callable and run by the executor service.

Lab 11: 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. Using PredicateCompositionWithStreams, implement the four predicates using streams.
   - Keep the compositional portion intact - just change the imperative code to streams.
   - Hint: Use *`Arrays.stream(anArray)`* to convert an array into a stream.
3. Implement a linux-style grep command using BufferedReader:
   - Count the occurrences of a given search word (grep -c).
   - Then, return a line for each occurrence of word (regular grep).
   - Hint: Use the method Util.getReader("a url").*`lines()`* to convert the reader into a stream.
4. Given a list of strings, print each string that is a palindrome:
   - Then, modify your algorithm to return the original word (unstripped).
5. 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.
6. Implement Conway's game of life using streams:
   - Run lab.extra.Life to see a graphical representation of the game in real-time.
   - Provide your own implementation by implementing this function:

     ```
     // Input: a 2D boolean array representing living (true) or dead (false)
     // Output: The next generation of the population.
     // Function<boolean[][], boolean[][]>
     ```

   - Call execute with your implementation.
   - You can parallelize the computation of the next generation.
   - This will stretch the limits of streams!

Lab 12: 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.

4. Revisit lab 11 and use currying to optimize the Fizz Buzz algorithm.
    - The map operation that uses *divisibleOrNot* duplicates code.
    - How can currying help?