Advanced Java with Java 8 Labs

Lab 01: The lambda form

Objective: test your understanding of how to implement lambdas.

Create these four interfaces

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();

Then, 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.

Lab 02: Functional interfaces & method references

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

Refactor the code from lab 01:

1. Refactor 4 interfaces from lab 1 to @FunctionalInterfaces.
2. Refactor to use static method references where possible.
3. Refactor to use constructor references.

Lab 03: Default methods lab

Objective: test your understanding of how to use default implementations in interfaces:

Refactor the code from lab 01:

1. Refactor Interface1 & Interface2 and provide a **default** implementation for each.
2. Call the default implementation.
3. Refactor Interface3 & Interface4 and provide a **static** implementation for each.
4. Call the static implementation.

Lab 04: Standard functional interfaces

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

Refactor the code from lab 01 and use 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 & 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, .9).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 by 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 regular log lines to stdout and words with error to stderr.

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);
}
```

Write a movie database implementation using the functionalized collection methods of sets, lists and maps. Use the given ImperativeMovieDb.java class as a start to save time and convert.

Lab 07: Read/Write locks with conditions

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

Use the Queue class from the courseware (5th slide in Threading & concurrency module) and convert from notify/wait with synchronize blocks to read/write locks with signal.

- Use two threads: one to put in the queue and one to get.
- Make the getter thread slower to simulate latency in processing.
- Test the original implementation and the new to compare result.
- Use the SynchronizedQueue, QueueNotifyWait, and TestQueue from the lab to save time (optional).

Lab 08: Using the executor service to find prime numbers

Objective: test your understanding of the executor service.

Write an application that counts the number of prime numbers in ranges using the ExecutorService:

- Choose the appropriate ExecutorService implementation.
- Use submit, call and future.
- Each range is 1000 elements.
- Each range is calculated by different threads using in the executor service.
- Print the number of primes found for all ranges.
- Use this method to determine if a number is prime:

```
private boolean isPrime(int primeCandidate) {
  boolean isPrime = primeCandidate == 2;

  if (primeCandidate > 2) {
    isPrime = true;
    for (int testValue = 2; testValue <= Math.sqrt(primeCandidate); ++testValue) {
      if (primeCandidate % testValue == 0) {
        isPrime = false;
      break;
      }
    }
  }

  return isPrime;
}
```

Lab 09: Using promises to find prime numbers

Objective: test your understanding of promises.

1. Re-implement the solution of lab 8 using promises.
2. Add exception handling to the promise:
   - Modify the isPrimeMethod(int primeCandidate) to throw an exception if the number is negative.
   - 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.

Lab 10: Single-threaded stream networking

Objective: test your understanding of Java stream-based networking.

1. Write a simple chat server that will allow two peers to communicate via text to each other. Set up the communication this way:
   - ChatReceiver class sets up a server socket on port 8080.
   - Peer uses telnet to connect to port 8080.
   - Telnet side sends a user name to establish connection.
   - ChatReceiver waits for telnet to send name, then responds with its own name.
   - Telnet side then starts conversation.
   - Only one side at a time can speak while the other is waiting.
   - Chat is completed when either side types "Bye" or drops connection.
2. Write a chat connector that connects with ChatReceive and automates the handshaking.

Lab 11: NIO-based networking

Objective: test your understanding of Java NIO-based networking.

Write a chat server that will allow many users to chat to each other using chat rooms:

- User logs into chat room using telnet.
- Upon connection establishment, user is prompted for id and chat room to enter.
- User can then chat with all other user in that chat room. All messages will only be sent to users in the same chat room. If no other user is in the chat room, messages are sent nowhere.
- New users can log in to same or other chat room.
- User leaves the chat room when user enter "bye" or connection is broken.

Lab 12: Using streams

Objective: test your understanding and practice thinking in streams.

Use streams to implement these algorithms:

1. Iterate through all even numbers from 0 to 100:
   - Print out the 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.
2. Go back to lab 5 and change the implementation of the predicate composition using streams.
   - Keep the compositional portion intact - just change the imperative code to streams.
3. Implement a linux-style grep command using BufferedReader:
   - Count occurrences of a given search word.
   - Return line for each occurrence of word (regular grep).
4. Given a list of strings, print each string that is a palindrome:
   - Then modify your algorithm to return the original word (not inverted).
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.

Lab 13: Currying in Java

Objective: test your understanding of currying in Java.

Use currying to create a currying function that uses average, best or worse as a statistical method in calculating test scores. Use this type definitation 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.
- Worse, only the lowest score is used to determine the grade - all others are discarded.
- Use this enum definition:

```
private enum GradeCalcType
{
    AVERAGE,
    WORSE,
    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.WORSE).apply(scores));
}
```