Advanced Java with Java 8 Labs

Lab 01: The lambda form

Objective: test your understanding of how to implement lambdas.

Create these five 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();
5. Interface5.java

   public boolean isEqualToTen(int a);

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.
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 how to use the functional interfaces.

Refactor the code from lab 01:

1. Refactor 5 interfaces from lab 1 to @FunctionalInterfaces.
2. Refactor to use static, instance, or constructor method references where possible.

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 to provide a default implementation that pretty prints the square of A. 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 and use the standard functional interfaces for all five 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).

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. Implement the methods in FunctionalMovieDb and test with TestMovieDb.

Lab 07: Read/Write locks with conditions

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

Use the Queue class from the courseware and convert from notify/wait with synchronize blocks to read/write locks with signal.

- Implement the missing methods in QueueLockCondition using read/write locks.
- Test using pre-made TestQueue class. Errors will be flagged automatically.

Lab 08: Using the executor service to find prime numbers

Objective: test your understanding of the executor service.

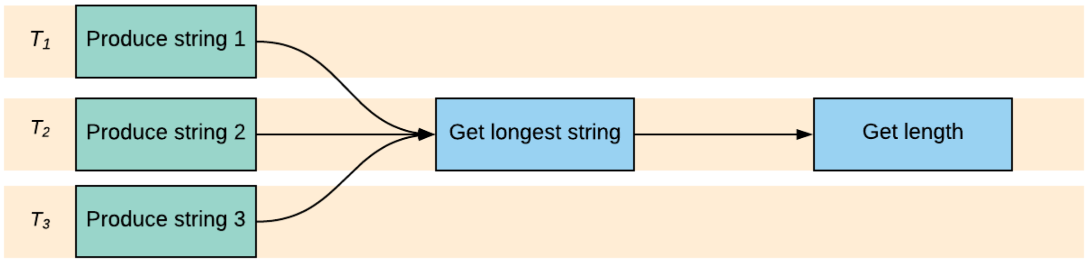
Write an application that counts the number of prime numbers in ranges using the ExecutorService:

- Choose the appropriate ExecutorService implementation.
- Use submit, call and future.
- Each range is 1000 elements.
- Each range is calculated by different threads using 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.

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 solution of lab 8 using promises.
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.

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 the methods:
  - getSpliterators() to create the spliterators
  - call() to search for prime numbers in spliterator
  - Print the number of elements that each thread processed.
- Divide the list in 4 *equal* pieces.
- Mind the spliterators that don't split.
- 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.

Lab 12: Currying in Java

Objective: test your understanding of currying in Java.

1. Use the currying and partial application techniques to implement a function that concatenates these strings together:
    - “Currying“, “ is”, “ great!”
    - Use the Function functional interface to define *a function that takes a string and returns a function that takes a string that returns another function that takes and returns a string.*
2. 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.