NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
PRACTICAL ASSESSMENT 2 FOR
Semester 2 AY2023/2024

CS2030S Programming Methodology II

April 2024 Time Allowed 120 minutes

INSTRUCTIONS TO CANDIDATES

1. The total mark is 45. We may deduct up to 4 marks for violating of style.
2. This is a CLOSED-BOOK assessment. You are only allowed to refer to one double-sided A4-size paper.
3. You should see the following files/directories in your home directory.
   - `Named.java`, `Perishable.java`, `Fruit.java`, `FruitStall.java` and `Q4.java` - `Q8.java` are for you to edit and solve the given task.
   - `pristine`, that contains a clean copy of the files above for your reference.
   - `Test1.java`, `Test2.java`, and `Test3.java`, for testing your code.
   - `Test1.jsh`, `Test2.jsh`, `Test3.jsh`, and `Fruits.jsh` are code snippets to help with interactive testing using `jshell`.
   - `checkstyle.sh`, `checkstyle.jar`, and `pe2_checks.xml`, for style checks.
   - `.vimrc`, `.vim`, and `.backup`, for `vim` configuration and backup files.
   - `StreamAPI.md` and `ListAPI.md`, for references to Java's Stream and List API.
4. You can leave the files in your home directory and log off after the assessment is over. There is no separate step to submit your code.
5. Only the files directly under your home directory will be graded. Do not put your code under a subdirectory.
6. Write your student number on top of EVERY FILE you created or edited as part of the `@author` tag. Do not write your name.
7. To compile your code, run `javac -Xlint:unchecked -Xlint:rawtypes *.java`. You can also compile the files individually if you wish.
8. To check the style of your code, run `./checkstyle.sh`. You can also check the style of individual file with `java -jar checkstyle.jar -c pe2_style.xml <FILENAME>`.

IMPORTANT: Any code you wrote that cannot be compiled will result in 0 marks will be given for the question or questions that depend on it. Make sure your program compiles by running

```
javac -Xlint:unchecked -Xlint:rawtypes *.java
```

before submission.

1. (4 points) Write a Java functional interface `Perishable` with a method `getDaysToExpiry` that returns the number of days to expiry of a perishable item. The method should return an integer.

   Write another Java functional interface `Named` with a method `getName` that returns the name of an item. The method should return a string.

2. (6 points) Write a Java class `Fruit` that has two fields: `nameFunc` of type `Named` and `expiryFunc` of type `Perishable`.

   The class should have (i) a constructor that takes two parameters: `nameFunc` and `expiryFunc`, and (ii) three methods: `getName` that returns the name of the fruit, `getDaysToExpiry` that returns the number of days to expiry of the fruit, and `toString` that returns a string representation of the fruit.

   Write your class so that it behaves as follows:

```
jshell> /open Named.java
jshell> /open Perishable.java
jshell> /open Fruit.java

jshell> new Fruit(() -> "Apple", () -> 5)
$.. ==> Apple (expires in 5 days)

jshell> new Fruit(() -> "Apple", () -> 5).getName()
$.. ==> "Apple"

jshell> new Fruit(() -> "Apple", () -> 5).getDaysToExpiry()
$.. ==> 5
```

   Make sure that your `toString` method returns *exactly* the same output as above.

IMPORTANT: For the rest of this paper, all methods written should follow the following constraints:

- The method body must consist of a single return statement.
- The method body must not contain any loop or conditional statement.
- The method body must not use any lambda expression with blocks (e.g., `() -> { .. }`.)

Useful APIs are provided in the files `ListAPI.md` and `StreamAPI.md`.

Three classes representing apple, banana, and devil's fruit, have been created in the file `Fruits.jsh` for your convenience. The jshell code given below uses these three classes.

---

3. (6 points) Write a generic class `FruitStall<T>`, where `T` is a subtype of `Fruit`. The class has a field of type `List<T>` that keeps track of the list of fruits sold at the stall.

   The class should have two constructors: (i) one that takes a list of fruits and stores it in the `fruits` field, and (ii) one that takes in no arguments and initializes the list of fruits to an empty list.

   The class should have a method `getFruits` that returns the list of fruits sold at the stall.

   The class also provides two methods: `findFruitsByName` that takes a string `name` and returns a list of fruits with the given name, and `toString` that returns the string representation of the list of fruits sold at the stall.

   Write your class so that it behaves as follows:

```
jshell> /open Named.java
jshell> /open Perishable.java
jshell> /open Fruit.java
jshell> /open FruitStall.java

jshell> /open Fruits.jsh

jshell> new FruitStall<String>()
|  Error:
|  type argument java.lang.String is not within bounds of type-variable T
|  new FruitStall<String>(); // Compilation error
|                 ^----^

jshell> new FruitStall<Fruit>(List.of(
   ...>       new Fruit(() -> "Apple", () -> 4),
   ...>       new Fruit(() -> "Banana", () -> 1)))
$.. ==>
- Apple (expires in 4 days)
- Banana (expires in 1 days)


jshell> List<Apple> apples = List.of(new Apple(2), new Apple(5))
apples ==> [Apple (expires in 2 days), Apple (expires in 5 days)]

jshell> new FruitStall<Apple>(apples)
$.. ==>
- Apple (expires in 2 days)
- Apple (expires in 5 days)


jshell> new FruitStall<Fruit>(apples)
$.. ==>
- Apple (expires in 2 days)
- Apple (expires in 5 days)

jshell> List<Fruit> list = new FruitStall<Fruit>(apples).getFruits()
list ==> [Apple (expires in 2 days), Apple (expires in 5 days)]

jshell> List<Fruit> list = List.of(new Mango(), new Apple(5), new Mango())
list ==> [Mango (expires in 4 days), Apple (expires in 5 days), Mango (expires in 4 days)]

jshell> new FruitStall<>(list).findFruitsByName("Apple")
$.. ==> [Apple (expires in 5 days)]

jshell> new FruitStall<>(list).findFruitsByName("Banana")
$18 ==> []

jshell> new FruitStall<>(list).findFruitsByName("Mango")
$19 ==> [Mango (expires in 4 days), Mango (expires in 4 days)]
```

Make sure that your `toString` method returns *exactly* the same output as above.

4. (6 points) Write a static generic method `mergeStalls` in the class `Q4` that takes two fruit stalls as arguments and returns a new fruit stall that contains all the fruits from both stalls. The fruits in the new stall are concatenated in the order they appear in the input stalls.

   Write your code so that it behaves as follows:

```
jshell> /open Named.java
jshell> /open Perishable.java
jshell> /open Fruit.java
jshell> /open FruitStall.java
jshell> /open Q4.java

jshell> /open Fruits.jsh

jshell> FruitStall<Apple> fs1 = new FruitStall<>(List.of(new Apple(1), new Apple(9)))
fs1 ==>
- Apple (expires in 1 days)
- Apple (expires in 9 days)

jshell> FruitStall<Fruit> fs2 = new FruitStall<>(List.of(new Apple(2)))
fs2 ==>
- Apple (expires in 2 days)

jshell> Q4.mergeStalls(fs1, fs2)
$.. ==>
- Apple (expires in 1 days)
- Apple (expires in 9 days)
- Apple (expires in 2 days)

jshell> Q4.mergeStalls(fs1, new FruitStall<>())
$.. ==>
- Apple (expires in 1 days)
- Apple (expires in 9 days)

jshell> FruitStall<Fruit> fs = Q4.mergeStalls(fs1, fs2)
fs ==>
- Apple (expires in 1 days)
- Apple (expires in 9 days)
- Apple (expires in 2 days)

jshell> FruitStall<Apple> fs = Q4.mergeStalls(fs1, fs2)
|  Error:
|  incompatible types: inference variable U has incompatible bounds
|      equality constraints: Apple
|      lower bounds: Fruit,Apple
|  FruitStall<Apple> fs = Q4.mergeStalls(fs1, fs2);  // error
|                         ^----------------------^
```

5. (6 points) Write a static method `findUniqueFruitTypes` in a class `Q5` that takes a list of fruit stalls and returns a list of unique fruit names sold at these stalls, sorted in alphabetical order.

   Write your code so that it behaves as follows (continuing from the examples in Question 4):

```
jshell> /open Q5.java

jshell> FruitStall<Fruit> fs3 = new FruitStall<>(List.of(new Apple(8), new Mango()))
fs3 ==>
- Apple (expires in 8 days)
- Mango (expires in 4 days)

jshell> FruitStall<Mango> fs4 = new FruitStall<>(List.of(new Mango()))
fs4 ==>
- Mango (expires in 4 days)
```

```
jshell> Q5.findUniqueFruitTypes(List.of(fs1, fs2, fs3, fs4))
$.. ==> [Apple, Mango]

jshell> Q5.findUniqueFruitTypes(List.of(fs4, fs3, fs2, fs1))
$.. ==> [Apple, Mango]

jshell> Q5.findUniqueFruitTypes(List.of(fs4))
$.. ==> [Mango]
```

6. (6 points) Write a static method `consolidateStallsbyType` in a class `Q6` that takes a list of fruit stalls and returns a list of new consolidated fruit stalls, where each of the return fruit stalls contains the fruits of a particular type. The order of the fruit stalls in the returned list should be the same as the order of the unique fruit types sorted by their names. The order of fruits in each stall should be the same as the order in the original stall. The input list can contain fruit stalls for different types of fruits, but the return type must be `List<FruitStall<Fruit>>`.

   Write your code so that it behaves as follows (continuing from the examples in Question 5):

```
jshell> /open Q6.java

jshell> FruitStall<Devil> fs5 = new FruitStall<>(List.of(new Devil(), new Devil()))
fs5 ==>
- Devil (expires in 100000 days)
- Devil (expires in 100000 days)

jshell> Q6.consolidateStallsbyType(List.of(fs3, fs4, fs5))
$.. ==> [
- Apple (expires in 8 days)
,
- Devil (expires in 100000 days)
- Devil (expires in 100000 days)
,
- Mango (expires in 4 days)
- Mango (expires in 4 days)
]
```

7. (6 points) Write a generic class `Q7` that implements the `Comparator` interface that allows comparison of two fruit stalls, to see which fruit stalls has the shortest day to expiry among all their fruits.

   Recall that the `Comparable` interface is defined as follows:

```
interface Comparator<T> {
  int compare(T o1, T o2);
}
```

   where `compare` returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

   Write your code so that it behaves as follows (continuing from the examples in Question 6):

```
jshell> /open Q7.java

jshell> new Q7<>().compare(fs3, fs4) == 0
$.. ==> true
jshell> new Q7<>().compare(fs4, fs5) < 0
$.. ==> true
```

```
jshell> new Q7<>().compare(fs5, fs4) > 0
$.. ==> true
```

Note: The value infinity and negative infinity are represented by `Integer.MAX_VALUE` and `Integer.MIN_VALUE` in Java respectively.

8. (5 points) The rating of a fruit stall is proportional to the soonest expiry dates among all the fruits sold at the stall.

   Write a static method `sortByShortestExpiry` in a class `Q8` that takes a list of fruit stalls and returns a list of fruit stalls sorted in ascending order of the ratings. If two stalls have the same rating, the order of the stalls in the returned list should be the same as the order of the stalls in the input list.

   Write your code so that it behaves as follows (continuing from the examples in Question 7):

```
jshell> /open Q8.java
jshell> Q8.sortByShortestExpiry(List.of(fs1, fs2, fs3, fs4, fs5))
$.. ==> [
- Apple (expires in 1 days)
- Apple (expires in 9 days)
,
- Apple (expires in 2 days)
,
- Apple (expires in 8 days)
- Mango (expires in 4 days)
,
- Mango (expires in 4 days)
,
- Devil (expires in 100000 days)
- Devil (expires in 100000 days)
]
```

END OF PAPER

This page is intentionally left blank.

This page is intentionally left blank.