CS2030 Programming Methodology
Semester 2 2018/2019

13 March – 15 March 2019
Tutorial 5
Testability of Objects and Methods

1. In an earlier course on introductory programming methodology, you would probably have been tasked to write a program to deal with fraction operations. In this question, your task is to define a `Fraction` class with similar functionality, while paying particular attention on the ease of testability of the class.

   Design a `Fraction` class that supports the operations prescribed by the following methods:

   - `public Fraction add(Fraction other)`: adds a fraction `other` to itself, resulting in a fraction in the simplest form
   - `public Fraction subtract(Fraction other)`: subtracts a fraction `other` from itself, resulting in a fraction in the simplest form
   - `public Fraction multiply(Fraction other)`: multiplies a fraction `other` to itself, resulting in a fraction in the simplest form
   - `public Fraction divideBy(Fraction other)`: divides itself by a fraction `other`, resulting in a fraction in the simplest form
   - `public Fraction simplify()`: returns the fraction in its simplest form
   - `public int numerator()`: returns the numerator of the fraction
   - `public int denominator()`: returns the denominator of the fraction
   - `public boolean equals(Object other)`: compares the `other` fraction with itself for equality
   - `public String toString()`: outputs the fraction

   Just like the Java's classes `String`, `Integer`, `Double`, etc., our `Fraction` class should be implemented as an immutable class. Here are some issues to consider:

   - Methods should not be provided to alter the state of the object
   - All instance fields should be made `private`
   - All instance fields should be made `final`

   Take note that the class should not allow the creation of a fraction whose denominator is zero. Decide how you would like to implement this. You also need to provide the static constants `ZERO` and `ONE`.

   Finally, write suitable tests to test each method.

2. Study the following two implementations of the `Burger` class.

- Implementation A

```
class Burger {

    private String bun;
    private String patty;
    private String vegetable;

    Burger(String bun) {
        this.bun = bun;
    }

    Burger(String bun,
            String patty) {
        this.bun = bun;
        this.patty = patty;
    }

    Burger(String bun,
            String vegetable) {
        this.bun = bun;
        this.vegetable = vegetable;
    }

    Burger(String bun, String patty,
            String vegetable) {
        this.bun = bun;
        this.patty = patty;
        this.vegetable = vegetable;
    }

    @Override
    public String toString() {
        return patty + ", " +
            vegetable + " on a " +
            bun + " bun";
    }
}
```

- Implementation B

```
class Burger {

    private String bun;
    private String patty;
    private String vegetable;

    Burger(String bun) {
        this.bun = bun;
    }

    void vegetable(String vegetable) {
        this.vegetable = vegetable;
    }

    void patty(String patty) {
        this.patty = patty;
    }

    @Override
    public String toString() {
        return patty + ", " + vegetable +
            " on a " + bun + " bun";
    }
}
```

Now there are four types of burgers offered on the menu:

- Plain-burger: bun only
- Herbi-burger: vegetable on bun
- Carni-burger: patty on bun
- Omni-burger: patty and vegetable on bun

Suppose the following that buns, patties, and vegetable are represented using `String`

(a) Identify the shortcomings of the above implementations and design a new `Burger` class. Instantiate and output the following burgers:

- croissant only
- fish in sesame seed bun
- lettuce in croissant bun
- beef and lettuce on sesame seed bun

You may ignore the `null` values in the output.

(b) Now suppose we would like to employ method chaining when creating burgers. Specifically, we create a burger with a bun first (using the `create`) and thereafter, include patty and/or vegetable. For example,

```
Burger.create(sesame).patty(beef).vegetable(lettuce)
```

Design a `Burger` class to support the above.

(c) Replace all occurrences of the `null` value by making use of Java's `Optional<T>`, so that the following output is generated.

```
jshell> Burger.create("croissant")
$3 ==> no patty, no vegetable, on a croissant bun

jshell> Burger.create("sesame").patty("fish")
$4 ==> fish, no vegetable, on a sesame bun

jshell> Burger.create("croissant").vegetable("lettuce")
$5 ==> no patty, lettuce, on a croissant bun

jshell> Burger.create("sesame").patty("beef").vegetable("lettuce")
$6 ==> beef, lettuce, on a sesame bun
```