CS2030 Programming Methodology
Semester 2 2018/2019

13 March – 15 March 2019
Tutorial 5 Suggested Guidance
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.

```
import java.util.Arrays;

public class Fraction {
    public static final Fraction ZERO = new Fraction(0, 1);
    public static final Fraction ONE = new Fraction (1, 1);
    private final int num;
    private final int den;

    private Fraction(int num, int den) {
        int sign = 1;

        if (num < 0) {
            num *= -1;
            sign *= -1;
        }
        if (den < 0) {
            den *= -1;
            sign *= -1;
        }

        int factor = gcd(num, den);
        this.num = sign * num / factor;
        this.den = den / factor;
    }

    public int numerator() {
        return this.num;
    }

    public int denominator() {
        return this.den;
    }

    public static Fraction valueOf(int num, int den) {
        if (den == 0) {
            throw new IllegalArgumentException("denominator is zero");
        }
        return new Fraction(num, den);
    }

    public Fraction add(Fraction other) {
        return new Fraction (
                this.num * other.den + other.num * this.den,
                this.den * other.den);
    }

    public Fraction subtract(Fraction other) {
        return add(other.negate());
    }
```

```
    public Fraction multiply(Fraction other) {
        return new Fraction(this.num * other.num, this.den * other.den);
    }

    public Fraction divide(Fraction other) {
        return new Fraction(this.num * other.den, this.den * other.num);
    }

    public Fraction simplify() {
        return new Fraction(this.num, this.den);
    }

    private Fraction negate() {
        return new Fraction(this.num * -1, this.den);
    }

    private int gcd(int a, int b) {
        while (b > 0) {
            int r = a % b;
            a = b;
            b = r;
        }
        return a;
    }

    @Override
    public int hashCode() {
        int[] array = new int[]{this.num, this.den};
        return Arrays.hashCode(array);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (!(obj instanceof Fraction)) {
            return false;
        }
        Fraction other = (Fraction) obj;
        return this.num == other.num && this.den == other.den;
    }

    @Override
    public String toString() {
        return this.num + "/" + this.den;
    }
}
```

```
/open Fraction.java
Fraction.valueOf(30, 40).simplify()
Fraction.valueOf(20, -40)
Fraction.valueOf(-20, 40)
Fraction.valueOf(-20, -40)
Fraction.valueOf(30, 40).add(Fraction.valueOf(10, 20))
Fraction.valueOf(30, 40).subtract(Fraction.valueOf(10, 20))
Fraction.valueOf(30, 40).multiply(Fraction.valueOf(10, 20))
Fraction.valueOf(30, 40).divide(Fraction.valueOf(10, 20))
Fraction.ZERO
Fraction.valueOf(3,4).add(Fraction.ONE)
Fraction.valueOf(3,4).multiply(Fraction.ZERO)
Fraction.valueOf(3,4).multiply(Fraction.valueOf(4,3)).equals(Fraction.ONE)
Fraction.valueOf(2, 0)
```

To run the script, save in a file (say `fraction.jsh`), and invoke:
`$ cat fraction.sh | jshell`

A word about overriding `equals` method. The following is extracted from the Java API of `Object`'s `equals` method:

> Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

This is particular important for he functionality of the `HashMap` and `HashSet` collections. Using a simple example,

```
jshell> Map<Fraction, String> map = new HashMap<>()
map ==> {}

jshell> map.put(Fraction.valueOf(0, 1), "first")
$4 ==> null

jshell> map.get(Fraction.valueOf(0, 1))
$5 ==> "first"
```

Without the `hashCode` method, `HashMap`'s `get` method will return `null` because the hashcodes of the fraction instances in `put` and `get` are different.

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.

```
class Burger {

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

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

    static Burger plainBurger(String bun) {
        return new Burger(bun, null, null);
    }

    static Burger herbiBurger(String bun, String vegetable) {
        return new Burger(bun, null, vegetable);
    }

    static Burger carniBurger(String bun, String patty) {
        return new Burger(bun, patty, null);
    }

    static Burger omniBurger(String bun, String patty, String vegetable) {
        return new Burger(bun, patty, vegetable);
    }

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

```
/open Burger.java
Burger plainCroissant = Burger.plainBurger("croissant")
Burger fishame = Burger.carniBurger("sesame", "fish")
Burger veggie = Burger.herbiBurger("croissant", "lettuce")
Burger hamburger = Burger.omniBurger("sesame", "beef", "lettuce")
/exit
```

Some issues that are addressed:

- Implementation A is not compilable as there are two constructors with the same signature, despite that their purpose is different.
- Multiple constructors in implementation A that varies only in their arguments. For example,

  ```
  Burger plainCroissant = new Burger(croissant)
  Burger fishame = new Burger(sesame, fish)
  Burger veggie = new Burger(croissant, lettuce)
  Burger hamburger = new Burger(sesame, beef, lettuce)
  ```

  Replace them with `static` factory methods having more meaningful names.
- Both implementations A and B allow for data to mutate.
- Implementation B's construction of, say hamburger, is cumbersome

  ```
  Burger hamburger = new Burger("sesame")
  hamburger.patty("beef");
  hamburger.vegetable("lettuce");
  ```

(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.

```
class Burger {

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

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

    static Burger create(String bun) {
        return new Burger(bun, null, null);
    }

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

    Burger vegetable(String vegetable) {
        return new Burger(this.bun, this.patty, vegetable);
    }
```

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

(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
```

```
import java.util.Optional;

class Burger {

    private final String bun;
    private final Optional<String> patty;
    private final Optional<String> vegetable;

    private Burger(String bun, Optional<String> patty,
            Optional<String> vegetable) {
        this.bun = bun;
        this.patty = patty;
        this.vegetable = vegetable;
    }

    static Burger create(String bun) {
        return new Burger(bun, Optional.empty(), Optional.empty());
    }

    Burger patty(String patty) {
        return new Burger(this.bun, Optional.of(patty), this.vegetable);
    }

    Burger vegetable(String vegetable) {
        return new Burger(this.bun, this.patty, Optional.of(vegetable));
    }
```

```
    @Override
    public String toString() {
        return patty.orElse("no patty") + ", " + vegetable.orElse("no vegetable") +
            ", on a " + bun + " bun";
    }
}
```