Mutation and immutability

Mutation

- **mutation**: A modification to the state of an object.

- Mutation must be done with care.
  - Can the object's state be damaged?
  - Is the old state important? Is it okay to "lose" it?
  - Do any other clients depend on this object?
    - Do they expect that its state will not change?

Immutable classes

- **immutable**: Unable to be changed (mutated).
  - Basic idea: A class with no "set" methods (*mutators*).

- In Java, Strings are immutable.
  - Many methods appear to "modify" a string.
  - But actually, they create and return a new string (*producers*).

"Modifying" strings

- What is the output of this code?

```
String name = "lil bow wow";
name.toUpperCase();
System.out.println(name);
```

- The code outputs `lil bow wow` in lowercase.
- To capitalize it, we must reassign the string:

```
name = name.toUpperCase();
```

  - The `toUpperCase` method is a producer, not a mutator.

If Strings were mutable...

- What could go wrong if strings were mutable?

```
public Employee(String name, ...) {
    this.name = name;
    ...
}

public String getName() {
    return name;
}
```

  - A client could accidentally damage the Employee's name.

```
String s = myEmployee.getName();
s.substring(0, s.indexOf(" "));  // first name
s.toUpperCase();
```

Making a class immutable

- 1. Don't provide any methods that modify the object's state.
- 2. Ensure that the class cannot be extended. (later)
- 3. Make all fields `final`.
- 4. Make all fields `private`. (ensure encapsulation)
- 5. Ensure exclusive access to any mutable object fields.
  - Don't let a client get a reference to a field that is a mutable object.

    (Don't allow any mutable representation exposure.)

Mutable Fraction class

```
public class Fraction {
    private int numerator, denominator;

    public Fraction(int n)
    public Fraction(int n, int d)
    public int getNumerator(),  getDenominator()

  public void add(Fraction other) {
    numerator = numerator * other.denominator
              + other.numerator * denominator;
    denominator = denominator * other.denominator;
}

}
```

Immutable methods

```
// mutable version
public void add(Fraction other) {
    numerator = numerator * other.denominator
              + other.numerator * denominator;
    denominator = denominator * other.denominator;
}

// immutable version
public Fraction add(Fraction other) {
    int n = numerator  * other.denominator
          + other.numerator * denominator;
    int d = denominator * other.denominator;
    return new Fraction(n, d);
}
```

- former mutators become *producers*
  - create/return a new immutable object rather than modifying this one

Pros/cons of immutability

- Immutable objects are **simple**.
  - You know what state they're in (the state in which they were born).
- Immutable objects can be **freely shared** among code.
  - You can pass them, return them, etc. without fear of damage.
- Con: Immutable objects can consume more **memory**.
  - Need a unique instance for each unique abstract value used.

END