## Lecture 16

We have been commissioned to write a class definition from which clients can construct objects that represent *fractions*. We are to provide simple fraction arithmetic methods.

We use the example to illustrate the *process* of producing near-professional-quality software (as professional as our current knowledge of Java allows).



### Program errors

- Correctness (being free from error; mapping inputs to outputs in accord with the program spec.) is one of the most important aspects of software quality
- Three types of error
  - Compile-time errors
    - syntax errors
    - certain semantic errors
  - Run-time errors
  - Logic errors
- The dangers of releasing buggy software include loss of money, livelihood, life, etc. and damage to health, environment, etc.



### The student approach to software development

```
Write some code at home on paper (optional)

Type in the whole program all in one go (from
paper if applicable, but mostly from off the
top of your head)

Repeat
   Compile the program and then hack at the parts
   that seem to cause compiler-errors
until the program compiles

For each example given by lecturer in the exercise

   Run a test with this example as data

   Repeat

      Hack at the parts that seem to cause run-time
      or logic errors

      Repeat
         Compile the program and then hack at the
         parts that seem to cause compiler-errors
      until the program compiles

   until the output for this test run kind-of matches
   that given in the exercise

Add as many comments as time allows ("because the
lecturer likes them")

Hand-up the program
```



### A better approach

What to do when you are ready to write a class definition:

- Determine the class interface
  - try to offer as complete a set of services as possible
- Create an initial class definition from a template
- Take each item in the interface in turn
  - write a header and a header comment (including preconditions) first
  - then write black-box and random test cases into a test-driver



### A better approach, continued

- Now decide on a representation (instance variables)
- Take each item in the interface in turn
  - write a pseudocode method body and include this as comments
  - write real code beneath these comments
  - add any glass-box test cases to the test-driver
  - inspect what has been written
  - compile, run on the test cases and debug



### The interface of the `Fraction` class

- Public constructors:

- Public instance methods:

### Header's and documentation for `Fraction`'s constructor

```
/* ====================================
      CONSTRUCTORS
   ====================================
*/
```



### Testing: proper attitude

- Your goal switches to being a destructive one: you must hope to break the software/to find bugs
- You should assume that you will never eliminate all bugs: assume that there are always more bugs that you haven't found yet
- *Question: What is a successful test?*
  *Answer:*



### Test-drivers

- Testing should be as automated as possible
- In particular, it makes *regression testing* easier (re-testing after program changes)
- So we write *test-drivers*: methods that push test data through other parts of our code
- In Java, *every* class definition can have a `main` method
- The `main` method can be the test-driver (e.g. the `main` method in `Fraction`)
- (Some programmers prefer to put test-drivers into separate class definitions (even separate packages) to avoid clutter)



### Automating testing

- Within test-drivers, even the comparison of *actual values* with *required values* should be automated
  - calculate required values by hand, embed into test-driver
  - check some mathematical or other property of the actual values
  - include code that computes the required value by another (presumably inferior) approach



### Test-suite design: black-box & random tests

- Rarely is it practical to test exhaustively
  - instead your test-suite must have good *coverage*: favour quality over quantity
  - use *equivalence partitioning*: choose test cases that are representative of collections of data
- Black-box tests: based on the purpose of the code
  - normal, boundary and exceptional values
- Random tests: avoids subconsciously under-testing parts you're not sure of



### Testing `Fraction`'s constructor

Automation is made easier in this case if we also assume we have written the `toString` method

```
/* ====================================
      TEST DRIVER
   ====================================
*/
   public static void main(String[] args)
   {
```