Lecture 17

We continue to illustrate the *process* of producing near-professional-quality software, using the `Fraction` class as our example.

First, you decide the interface. Then, for each item in the interface, you write a header and header documentation (including preconditions). Then, you create a test-driver and include black-box and random test cases.

Once that has been done, you decide on a representation.

Then, you tackle method bodies: write them in pseudocode, write the code, write glass-box test cases, inspect the code, and then compile, run & debug the code.

`Fraction`'s `public` interface

```
/**
 * Returns a new fraction that represents
 * the sum of this fraction and the
 * specified fraction.
 * @param aFraction the fraction we are
 * adding; must not be null.
 * @return a fraction that represents the
 * sum of this fraction and the parameter.
 */
public Fraction plus(Fraction aFraction)
```

(`minus` is similar)


```
/**
 * Determines whether the specified
 * fraction is equal to this fraction.
 * The result is true if and only if the
 * specified fraction is non-null and
 * denotes the same fraction as this
 * fraction.
 *
 * @param aFraction the fraction to
 * compare this fraction against.
 * @return true if the fractions are
 * mathematically equal; false otherwise.
 */
public boolean isEqualTo(Fraction aFraction)

/**
 * Returns a string representation of this
 * fraction in reduced form, e.g. it
 * returns "2/3", never "4/6".
 *
 * @return a string representation.
 */
public String toString()
```


Testing `plus` and `minus`

As well as doing calculations by hand. . .

```
f1 = new Fraction(1, 2);
f2 = new Fraction(1, 3);
f3 = new Fraction(5, 6);
if (! (f1.plus(f2).isEqualTo(f3)))
{  System.out.println("Error");
}
```

. . . we can also exploit mathematical properties, e.g.:

$$f_1 + f_2 - f_1 = f_2$$

```
if (! (f1.plus(f2).minus(f1).isEqualTo(f2)))
{  System.out.println("Error");
}
```


Testing `plus` and `minus`, continued

As well as black-box tests, this is a case where random testing makes good sense, e.g.:

```
for (int i = 0; i < 1000; i++)
{  int r1 = (int)
       (Math.random() * 101 - 50);
   int r2 = (int)
       (Math.random() * 101 - 50);
   int r3 = (int)
       (Math.random() * 101 - 50);
   int r4 = (int)
       (Math.random() * 101 - 50);
   if (r2 != 0 && r4 != 0)
   {  f1 = new Fraction(r1, r2);
      f2 = new Fraction(r3, r4);
      if (! (f1.plus(f2).minus(f1).
            isEqualTo(f2)))
      {  System.out.println("Error");
      }
   }
}
```


`Fraction`'s representation

- Option 1: Store as a floating-point number

  ```
  private double decimalValue;
  ```

- Option 2: Store the numerator & denominator in instance variables

  ```
  private int num;
  private int denom;
  ```

  – Issue a: should we ‘reduce’ the values by dividing by their greatest common divisor?

  – Issue b: should we simplify negative values?

- Crucially, whichever option is chosen, the representation is hidden — so it can easily be changed later without affecting clients

`Fraction`**'s instance variables**

```
/* ====================================
   INSTANCE VARIABLES & CLASS VARIABLES
   ====================================
*/

   /**
    * The fraction's numerator.
    * Fraction's are stored in reduced
    * form, e.g. 4/6 is stored as 2/3;
    * 0/7 is stored as 0/1.
    * The numerator will always carry the
    * sign.
    */
   private int num;

   /**
    * The fraction's denominator.
    * Since the numerator carries the
    * sign, the denominator will always
    * be positive.
    */
   private int denom;
```


The body of `Fraction`**'s constructor**

```
public Fraction(int aNum, int aDenom)
{  /* If fraction represents zero, the
      denominator is always reduced to 1
    */
   if


   else
   {  int n = aNum; // the new numerator.
      int d = aDenom; // the new denominator
      /* Reduce n and d using their gcd.
         The gcd method requires +ve args,
         so pass in absolute values.
       */
```



```
      /* If denom is negative, move its
         sign to the numerator.
       */


      /* Now move n and d to the instance
         variables.
       */


   }
}
```


Helper method needed by the constructor

```
/* ====================================
   HELPER METHODS
   ====================================
*/
   /**
    * Returns the greatest common divisor
    * of two positive integers.
    *
    * @param a - any int; must be > 0.
    * @param b - any int; must be > 0.
    * @return the gcd of a and b.
    */
   private static int gcd(int a, int b)
   {  // Euclid's algorithm (recursive)
      if (b != 0)
      {  return gcd(b, a % b);
      }
      else
      {  return a;
      }
   }
```


Glass-box tests, inspection, compile/run/debug

- Glass-box tests: based on the code itself (method bodies)
  - all-paths/all-statements
- Manual inspection of the source code is a highly successful, low-cost methd of discovering bugs
- Only now do we compile, run & debug!
- Note the three kinds of comments.
  - HTML documentation can be automatically produced from the `javadoc` comments (`/** ...*/`) using the `javacdoc` utility program

The bodies of the methods

```
public Fraction plus(Fraction aFraction)
{  return new Fraction(
      this.num * aFraction.denom +
         aFraction.num * this.denom,
      denom * aFraction.denom);
}
public boolean isEqualTo(
   Fraction aFraction)
{  if (aFraction == null)
   {  return false;
   }
   else
   {  return (this.num == aFraction.num
      && this.denom == aFraction.denom);
   }
}
public String toString()
{  return (num + "/" + denom);
}
```

(`minus` is similar to `plus`)