02 Immutable Objects

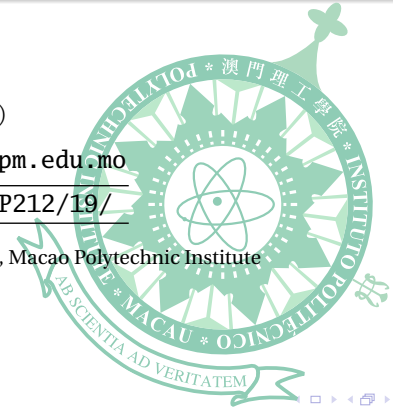
Instructor: Ke Wei（柯韋）

A319 Ext. 6452 wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP212/19/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

September 5(10), 2019



Outline

Immutable Objects and Classes

- If the contents of an object cannot be changed once the object is created, the object is called an *immutable* object, and its class is called an immutable class.
- The *Circle* class in the example is immutable because *x*, *y* and *radius* are all private, and there are no methods to change them.

```
1 class Circle {
2     private double x, y, radius;
3     public Circle(double x, double y, double radius) { ... }
4     public double getArea() { return Math.PI*radius*radius; }
5     public boolean contains(double x, double y) { ... }
6 }
```

- A method that can change a data field directly is called a *mutator*.
- A class with all private data fields and no mutators is "shallowly" immutable — the objects referred to by the data fields can still be changed.

Benefits of Immutability

Immutable classes, when used properly, can greatly simplify programming. For immutable objects,

- they can only be in one state,
- they are always consistent, if properly constructed,
- they are inherently *thread-safe*, and they can be shared freely.
- their references can be freely stored and cached.

Immutable classes are ideal for representing *values* of abstract data types, such as numbers or colors. Java's *String* class is a classic immutable class.

```
String s = "abc";
System.out.print(s.toUpperCase()); // prints "ABC", s is still "abc".
```

An Indirectly Mutable Class

```
1  public class Student {
2      private int id;
3      private BirthDate birthDate;
4      public Student(int id, int y, int m, int d) {
5          this.id = id;
6          birthDate = new BirthDate(y, m, d);
7      }
8      public int getId() {
9          return id;
10     }
11     public BirthDate getBirthDate() {
12         return birthDate;    // leaking a reference to the field
13     }
14 }
```

An Indirectly Mutable Class (2)

```
1  public class BirthDate {
2      private int year;
3      private int month;
4      private int day;
5      public BirthDate(int y, int m, int d) {
6          year = y;
7          month = m;
8          day = d;
9      }
10     public void setYear(int y) {
11         year = y;
12     }
13 }
```

- `BirthDate date = student.getBirthDate(); date.setYear(2000);`

Deep Immutables

For a class to be “deeply” immutable, it must

- mark all data fields private,
- provide no mutators, and
- provide no getters that would return a reference to a mutable data field object.

```
1  public class Student { ... // now a deeply immutable class
2      public BirthDate getBirthDate() {
3          return new BirthDate(birthDate); // returns a reference to a copy
4      }
5  }
6  public class BirthDate { ...
7      public BirthDate(BirthDate date) {
8          this(date.year, date.month, date.day);
9      }
10 }
```

An Immutable Rational Number Class

A rational number can be represented by a *fraction*, consisting of a pair of integers, one is the *numerator*, the other is the *denominator*:

$$\frac{\text{numerator}}{\text{denominator}}, \qquad \text{for example} \quad \frac{22}{7}.$$

We always set the denominator positive.

```
1 class Rational {
2     private long num;      // numerator
3     private long denom;    // denominator                    ...
```

We always return a *new* rational number object when a new value is required.

The Rational Number Class — Constructors

```
4    public Rational(long num, long denom) {
5        if ( denom == 0 && num == 0 ) {
6            this.num = 0; this.denom = 0;
7        } else if ( num == 0 ) {
8            this.num = 0; this.denom = 1;
9        } else if ( denom == 0 ) {
10           this.num = 1; this.denom = 0;
11       } else {
12           if ( denom < 0 ) { num = -num; denom = -denom; }
13           long d = gcd(num, denom);
14           this.num = num/d;
15           this.denom = denom/d;
16       }
17   }
18   public Rational() { this(0, 1); }                    //...
```

Rational Number Class — Operations

```
18      public Rational add(Rational y) {       // this + y
19          return new Rational(num*y.denom+y.num*denom, denom*y.denom);
20      }
21      public Rational neg() {                 // −this
22          return new Rational(-num, denom);
23      }
24      public Rational sub(Rational y) {       // this − y
25          return add(y.neg());
26      }
27      public Rational mul(Rational y) {       // this × y
28          return new Rational(num*y.num, denom*y.denom);
29      }                                                          //...
```

Rational Number Class — Operations (2)

```
30      public Rational recip() {                    // 1/this
31          return new Rational(denom, num);
32      }
33      public Rational div(Rational y) {            // this/y
34          return new Rational(num*y.denom, denom*y.num);
35      }
36      @Override
37      public String toString() {
38          return num+"/"+denom;
39      }
40  }
```

Practice: A Rational Number Class using Java's *BigInteger*

1. Create an empty Java project in Eclipse.
2. Create a class named *Rational*.
3. Copy and modify the code from the course slides, to implement the *immutable Rational* number class based on a pair of *immutable BigInteger* objects.
   - `import` *java.math.BigInteger*;
   - constructors, arithmetic operations, to-string operations, common instances.
4. Create a test class named *TestRational*, including the program entry point *main*.
5. Write some test code in *main* to make use of the *Rational* class.
6. Run the test code and capture the result.

The *BigInteger* Class — Selected Public Methods

- `static` *BigInteger valueOf*(`long` *y*) — returns a *BigInteger* whose value is equal to that of the specified `long`.
- `int` *compareTo*(*BigInteger y*) — compares this *BigInteger* with the specified *BigInteger* ($< 0, = 0, > 0$).
- *BigInteger negate*() — returns a *BigInteger* whose value is ($-$`this`).
- *BigInteger add*(*BigInteger y*) — returns a *BigInteger* whose value is (`this` $+ y$).
- *BigInteger multiply*(*BigInteger y*) — returns a *BigInteger* whose value is (`this` $\times y$).
- *BigInteger divide*(*BigInteger y*) — returns a *BigInteger* whose value is $\left(\frac{\texttt{this}}{y}\right)$.
- *BigInteger gcd*(*BigInteger y*) — returns a *BigInteger* whose value is the greatest common divisor of $|$`this`$|$ and $|y|$.
- *BigInteger pow*(`int` *y*) — returns a *BigInteger* whose value is ($\texttt{this}^y$).
- *String toString*() — returns the decimal String representation of `this` *BigInteger*.

Showing a Fraction as a Decimal Number

- We write a method to convert `this` rational number to a digit string with a specified number n of decimal digits.

  String toDecimalString(`int` n)

- To preserve n decimal digits by integer division, we multiply the numerator by 10^n.

$$22 \div 7 \cong 3.142857\,143, \quad (22 \times 100000) \div 7 \approx 3142857.143$$

- The integral quotient is now scaled up by 10^n, we need to put the decimal point n positions back.
- In the case there are no more than n digits, we need to prefix the result with additional zeros.
- We also need to consider negative results with a leading '—' sign.

Sample Test Cases

- A series with items canceling each other:

$$\frac{1}{1} = \left(-\frac{1}{99}\right) \times \left(-\frac{3}{97}\right) \times \left(-\frac{5}{95}\right) \times \cdots \times \left(-\frac{95}{5}\right) \times \left(-\frac{97}{3}\right) \times \left(-\frac{99}{1}\right).$$

```
Rational p = Rational.ONE;
for ( int i = 1; i <= 99; i += 2 ) {
    p = p.mul(new Rational(i, i-100));
    System.out.println(p);
}
```

- The Gregory-Leibniz series to approximate π:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots.$$

Van Wijngaarden Transformation

In order to accelerate convergence of an alternating series, Euler's transform can be computed as follows.

- Compute a row of partial sums,

$$s_{0,j} = \sum_{i=0}^{j} (-1)^i a_i.$$

- Form rows of averages between neighbors,

$$s_{k+1,j} = \frac{s_{k,j} + s_{k,j+1}}{2}.$$

- The first column $s_{k,0}$ then contains the partial sums of the Euler transform.

Adriaan van Wijngaarden pointed out that it is better not to carry this procedure through to the very end, but to stop two-thirds of the way. If $a_0, a_1, \ldots, a_{12}$ are available, then $s_{8,4}$ is almost always a better approximation than $s_{12,0}$.

Van Wijngaarden Transformation (2)

```
1.00000000 0.66666667 0.86666667 0.72380952 0.83492063 0.74401154 0.82093462 0.75426795 0.81309148 0.76045990 0.8
0.83333333 0.76666667 0.79523810 0.77936508 0.78946609 0.78247308 0.78760129 0.78367972 0.78677569 0.78426943 0.7
0.80000000 0.78095238 0.78730159 0.78441558 0.78596959 0.78503719 0.78564050 0.78522771 0.78552256 0.78530463 0.7
0.79047619 0.78412698 0.78585859 0.78519259 0.78550339 0.78533884 0.78543410 0.78537513 0.78541359 0.78538744
0.78730159 0.78499278 0.78552559 0.78534799 0.78542111 0.78538647 0.78540462 0.78539436 0.78540052
0.78614719 0.78525919 0.78543679 0.78538455 0.78540379 0.78539555 0.78539949 0.78539744
0.78570319 0.78534799 0.78541067 0.78539417 0.78539967 0.78539752 0.78539847
0.78552559 0.78537933 0.78540242 0.78539692 0.78539860 0.78539799
0.78545246 0.78539087 0.78539967 0.78539776 0.78539829
0.78542166 0.78539527 0.78539871 0.78539803
0.78540847 0.78539699 0.78539837
0.78540273 0.78539768
0.78540021
```

```
1 for ( int i = n-1; i > n/3; --i )
2     for ( int j = 0; j < i; ++j )
3         s[j] = s[j].add(s[j+1]).div(Rational.TWO);
```

Homework

- Use the *van Wijngaarden transformation* to improve the convergence of the Gregory-Leibniz series, modify the test class to implement this technique and obtain the result. You need to do some research yourself.
- To implement the transformation, you need an array of *Rational* objects. Try to recall how to create an array and initialize of the array elements.

```
Rational[] rs = new Rational[10];
Rational x = Rational.ZERO;
for ( int i = 0; i < rs.length; ++i ) {
    x = x.add(Rational.ONE); rs[i] = x;
}
```

- Zip your improved source files, including `TestRational.java`, `Rational.java` and a text file `output.txt` capturing the program output, in `Rational.zip` for future upload.