

Lecture 20: Rationals, Switch, Break, Continue

Last time:

1. Aliasing and Mutability
2. Floating Point calculations
3. Example class development: Rational Numbers

Today:

1. Continuation of Rational numbers
2. switch
3. break
4. Case continuation (or “fall through”)



Project #4 Is Assigned

- It is due **Friday, 3/16 at 11 pm**
- The project is **closed**
 - You must complete the project by yourself
 - Assistance can only be provided by teaching assistants (TAs) and instructors
 - You must not look at other students' code
- **Start now!**
 - Read entire assignment from beginning to end before starting to code
 - Check out assignment now from CVS
 - Follow the instructions *exactly*, as much of grading is automated

Rational Numbers (continued): Arithmetic Operations



- What you remember from middle / high school

- $p/q + s/t = (p*t + q*s) / (q*t)$

- $p/q * s/t = (p*s) / (q*t)$

We will focus on these two cases.

- $p/q - s/t = p/q + (-s/t)$

- $1/(p/q) = q/p$

- $(p/q) / (s/t) = p/q * t/s = p/q * (1/(s/t))$

Comparisons

- $p/q = s/t$ if
 - $p/q, s/t$ are in lowest terms, and
 - $p = s$ and $q = t$
- $p/q < s/t$ if $p \cdot t < q \cdot s$

We will focus on this case.

Using JUnit to test our Rational Class



```
public class RationalTest extends TestCase {

    private boolean check(Rational r, int n, int d) {
        return r.getNumerator() == n && r.getDenominator() == d;
    }

    public void testConstructorsEasy() {
        Rational r = new Rational(1, 3);
        assertTrue(check(r, 1, 3));
        r = new Rational(7);
        assertTrue(check(r, 7, 1));
    }

    public void testEquals() {
        assertTrue(new Rational(7, 5).equals(new Rational(70, 50)));
        assertFalse(new Rational(7, 5).equals(new Rational(7, 6)));
    }

    public void testToString() {
        assertEquals(new Rational(3,5).toString(), "3/5");
    }
}
```

Error Reporting

- When a Rational number is created with a 0 denominator, we want to report an error
 - In fact, for our very simple example, we will also exit (rather rudely) if there is a negative number, just to simplify things.
- Until now, we have been using text messages
- This is good, but does not halt execution
- To halt execution, use `System.exit`
`System.exit(1); // Halt with error`

Error Reporting Example

```
private boolean isValid() {
    return (num > 0 && den > 0);
}

public Rational(int num, int den) {
    this.num = num;
    this.den = den;
    if (!isValid()) {
        System.out.println("Error in constructor... Exiting");
        System.exit(1);
    }
    reduce();
}
```

Is Rational immutable?

YES! (So far.)

What would make it a mutable class? (A public setter!)

For example

```
public void addTo(Rational a) {  
    Rational answer = add(this, a);  
    num = answer.num;  
    den = answer.den;  
}
```

What is another way to write this `if-else-if` statement?



```
if (grade == 'A')
    System.out.println ("I'm very happy");
else if (grade == 'B')
    System.out.println ("I'm relatively happy");
else if (grade == 'C')
    System.out.println ("At least I get credit");
else
    System.out.println ("Check with the professor");
```

The switch Statement

- A way of doing **case analysis** based on a **control expression**

- Assume grade is a variable of type char
- Then following does a case analysis on value of grade

```
switch (grade) {  
    case 'A':  
        System.out.println ("I'm very happy");  
        break;  
    case 'B':  
        System.out.println ("I'm relatively happy");  
        break;  
    case 'C':  
        System.out.println ("At least I get credit");  
        break;  
    default: // All other cases  
        System.out.println ("Check with the professor");  
        break;  
}
```

- Case chosen depends on value of control expression
- Optional `default` case tells what to do if no other cases are applicable

The switch Statement: General Form



```
switch ( <control-expression> ) {  
  case <case-label-1> :  
    <statement-sequence-1>  
    break;  
  case <case-label-2> :  
    <statement-sequence-2>  
    break;  
  ...  
  case <case-label-n> :  
    <statement-sequence-n>  
    break;  
  default :  
    <default-statement-sequence>  
    break;  
}
```

The control-expression is one of the following types: char, int, short, byte

Our text says it cannot be a byte or short. This is wrong!

Each case label must be a value in type of control expression

You may have any number of statements, including if-else and loops

The "break" statement jumps out of the switch statement

The optional "default" case is executed if no other case matches

Control Expression and break

- The **control expression** can have one of the following types: char, int, short, byte
 - not float, double, boolean, long
 - not a String or other object
- The **break** statement jumps out of the switch statement
 - If omitted control flow continues into next case
 - Consider:

```
switch (grade) {
  case 'A':
    System.out.println ("I'm very happy");
  case 'B':
    System.out.println ("I'm relatively happy");
  case 'C':
    System.out.println ("At least I get credit");
  default: // All other cases
    System.out.println ("Check with the professor");
}
```
 - If grade has value 'B', then output is:

```
I'm relatively happy
At least I get credit
Check with the professor
```

Case Continuation

- Case continuation also called “cascading case behavior”, “falling through to the next case”, etc.
- It is occasionally handy for combining of cases

e.g. case-insensitivity

```
switch (grade) {  
    case 'a':  
    case 'A':  
        System.out.println ("I'm very happy");  
        break;  
    ...  
}
```

- **Be very careful** about using this cascading behavior!
 - Always insert `break` statements after every case
 - Then remove ones you do not want

The default Case

- `default` is optional
If omitted, and no case matches, then the switch statement does nothing
- However: you should **always include** a default case, even if you want nothing to be done if no case matches (you should never rely on implicit behavior!)
- Although cases are not required to be in order ... (following is legal):

```
switch ( option ) {  
    case 2:  
        ...  
    case 9:  
        ...  
    default:  
        ...  
    case 1:  
        ...  
}
```

- ... it is much better to list cases:
 - in increasing order
 - with `default` last

Why Use `switch`?

- `switch` can also be implemented using `if-else`
- `switch` also restricted in terms of data types in control statements
- Including `break` statements is a pain
- However
 - `switch` often more efficient (compiler generates better code)
 - Code can be more compact because of case-continuation behavior
 - Sometimes case analysis is clearer using `switch`