Programming languages Java

Code organization

Kozsik Tamás

Signaling errors by throwing an exception

```
public class Time {
  int hour;                          // 0 <= hour < 24
  int min;                           // 0 <= min  < 60

  ...

  public void setHour(int hour) {
    if (0 <= hour && hour <= 23) {
      this.hour = hour;
    } else {
      throw new IllegalArgumentException("Invalid hour!");
    }
  }
}
```

The assert statement

```
public class Time {
  int hour;                          // 0 <= hour < 24
  int min;                           // 0 <= min  < 60

  ...

  public void setHour(int hour) {
    assert 0 <= hour && hour <= 23;
    this.hour = hour;
  }
```

The assert statement

TestTime.java

```java
Time time = new Time(6,30);
time.setHour(30);
```

Running the program

```
$ java TestTime
$ java -enableassertions TestTime
Exception in thread "main" java.lang.AssertionError
    at Time.setHour(Time.java:7)
    at TestTime.main(TestTime.java:5)
$
```

Options to signal errors

Good solutions

- `IllegalArgumentException`: at module boundaries
- `assert`: inside a module
- Doc comment

Bad solutions

- Silently not perform the requested operation
- Do not check correct behaviour, let the program go wrong

Checked vs unchecked exceptions

Checked exceptions

```
public Time readTime(String fname) throws java.io.IOException
    // this code may throw an IOException
}
```

- The method's source code must declare that it propagates such an exception
- The compiler checks consistency
- E.g. `java.sql.SQLException`, `java.security.KeyException`

Checked vs unchecked exceptions

Checked exceptions

```
public Time readTime(String fname) throws java.io.IOException
    // this code may throw an IOException
}
```

- The method's source code must declare that it propagates such an exception
- The compiler checks consistency
- E.g. `java.sql.SQLException`, `java.security.KeyException`

Unchecked exceptions

- E.g. `NullPointerException`, `ArrayIndexOutOfBoundsException`
- Violation of a dynamic semantic rule of the language
- May occur (practically) everywhere
- Methods don't declare propagation

Propagation detection: compilation error

```
import java.io.IOException;
public class TestTime {
  public Time readTime(String fname) throws IOException {
    ... new java.io.FileReader(fname) ...
  }

  public static void main(String[] args) {
    TestTime tt = new TestTime();
    Time wakeUp = tt.readTime("wakeup.txt");
    wakeUp.aMinutePassed();
  }
}
```

Propagation detection: compilation error eliminated

```
import java.io.IOException;
public class TestTime {
  public Time readTime(String fname) throws IOException {
    ... new java.io.FileReader(fname) ...
  }

  public static void main(String[] args) throws IOException {
    TestTime tt = new TestTime();
    Time wakeUp = tt.readTime("wakeup.txt");
    wakeUp.aMinutePassed();
  }
}
```

Exception handling (catch an exception)

```
import java.io.IOException;
public class TestTime {
  public Time readTime(String fname) throws IOException {
    ... new java.io.FileReader(fname) ...
  }
  public static void main(String[] args) {
    TestTime tt = new TestTime();
    try {
      Time wakeUp = tt.readTime("wakeup.txt");
      wakeUp.aMinutePassed();
    } catch(IOException e) {
      System.err.println("Could not read wake-up time.");
    }
  }
}
```

The program can continue in spite of the problem

```
public class Receptionist {
   ...
  public Time[] readWakeupTimes(String[] fnames) {
    Time[] times = new Time[fnames.length];
    for(int i = 0; i < fnames.length; ++i) {
      try {
        times[i] = readTime(fnames[i]);
      } catch(java.io.IOException e) {
        times[i] = null;   // no-op
        System.err.println("Could not read " + fnames[i]);
      }
    }
    return times; // maybe sort times before returning?
  }
}
```

The try-catch statement

```
<try-catch-statement> ::= try <block-statement>
                           <catch-list>
                           <optional-finally-part>
<catch-list> ::= ""
               | <catch-part> <catch-list>

<catch-part> ::= catch (<exceptions> <identifier>)
                      <block-statement>

<exceptions> ::= <identifier>
               | <identifier> | <exceptions>

<optional-finally-part> ::= ""
                          | finally <block-statement>
```

Multiple catch-clauses

```
public static Time parse(String str) {
  String errorMessage;
  try {   String[] parts = str.split(":");
          int hour = Integer.parseInt(parts[0]);
          int minute = Integer.parseInt(parts[1]);
          return new Time(hour,minute);
  } catch(NullPointerException e) {
      errorMessage = "Null parameter is not allowed!";
  } catch(ArrayIndexOutOfBoundsException e) {
      errorMessage = "String must contain \":\"!";
  } catch(NumberFormatException e) {
      errorMessage = "String must contain two numbers!";
  }
  throw new IllegalArgumentException(errorMessage);
}
```

Multiple exceptions in a single catch-clause

```java
public static Time parse(String str) {
  try {
    String[] parts = str.split(":");
    int hour = Integer.parseInt(parts[0]);
    int minute = Integer.parseInt(parts[1]);
    return new Time(hour,minute);
  } catch(NullPointerException
         | ArrayIndexOutOfBoundsException
         | NumberFormatException e) {
    throw new IllegalArgumentException("Can't parse time!");
  }
}
```

The try-finally statement

```
public static Time readTime(String fname) throws IOException {
  var in = new BufferedReader(new FileReader(fname));
  Time time;
  try {
    String line = in.readLine();
    time = parse(line);
  } finally {
    in.close();
  }
  return time;
}
```

The finally clause gets executed in any case

```
public static Time readTime(String fname) throws IOException {
  var in = new BufferedReader(new FileReader(fname));
  try {
    String line = in.readLine();
    return parse(line);
  } finally {
    in.close();
  }
}
```

The try-catch-finally statement

```
public static Time readTime(String fname) throws IOException {
  var in = new BufferedReader(new FileReader(fname));
  try {
    String line = in.readLine();
    return parse(line);
  } catch (IllegalArgumentException e) {
    System.err.println(e);
    System.err.println("Using default value!");
    return new Time(0,0);
  } finally {
    in.close();
  }
}
```

Nesting try-statements

```
public static Time readTimeOrUseDefault(String fn) {
  try {
    var in = new BufferedReader(new FileReader(fn));
    try {
       String line = in.readLine();
       return parse(line);
    } finally {
       in.close();
    }
  } catch(IOException | IllegalArgumentException e) {
    System.err.println(e);
    System.err.println("Using default value!");
    return new Time(0,0);
  }
}
```

The *try-with-resources* statement

```
public static Time readTimeOrUseDefault(String fn) {
  try (
    var in = new BufferedReader(new FileReader(fn))
  ) {
    String line = in.readLine();
    return parse(line);
  } catch(IOException | IllegalArgumentException e) {
    System.err.println(e);
    System.err.println("Using default value!");
    return new Time(0,0);
  }
}
```

These are practically equivalent

try-finally

```
BufferedReader in = ...;
try {
  String line = in.readLine();
  return parse(line);
} finally {
  in.close();
}
```

try-with-resources

```
try (
  BufferedReader in = ...
) {
  String line = in.readLine();
  return parse(line);
}
```

A more sophisticated case: copying a file

```
static void copy(String in, String out) throws IOException {
    try (
        FileInputStream infile = new FileInputStream(in);
        FileOutputStream outfile = new FileOutputStream(out)
   ) {
        int b;
        while((b = infile.read()) != -1) {    // idiom!
            outfile.write(b);
        }
    }
}
```

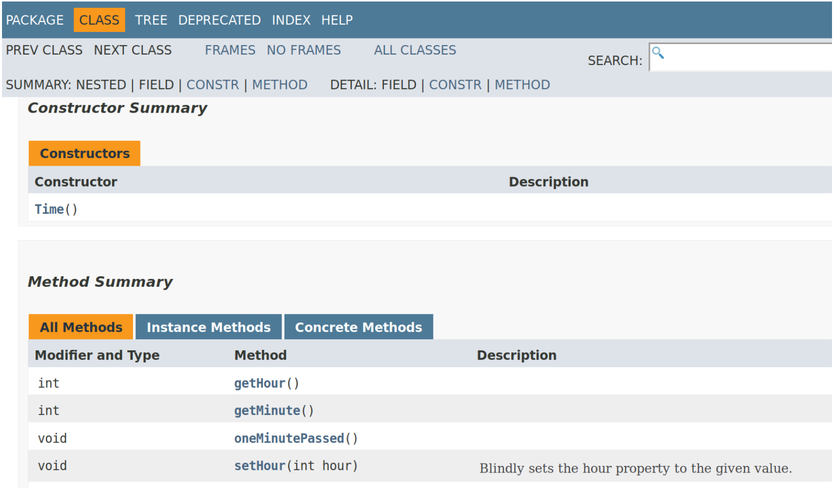
Documentation comment

```java
/** May throw AssertionError. */
public void setHour(int hour) {
    assert 0 <= hour && hour <= 23;
    this.hour = hour;
}
```

Documenting potentially erroneous use

```
/**
  Blindly sets the hour property to the given value.
  Use it with care: only pass {@code hour} satisfying
  {@code 0 <= hour && hour <= 23}.
*/
public void setHour(int hour) {
    this.hour = hour;
}
```

javadoc Time.java



PACKAGE CLASS TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SEARCH:

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Constructor Summary

Constructors

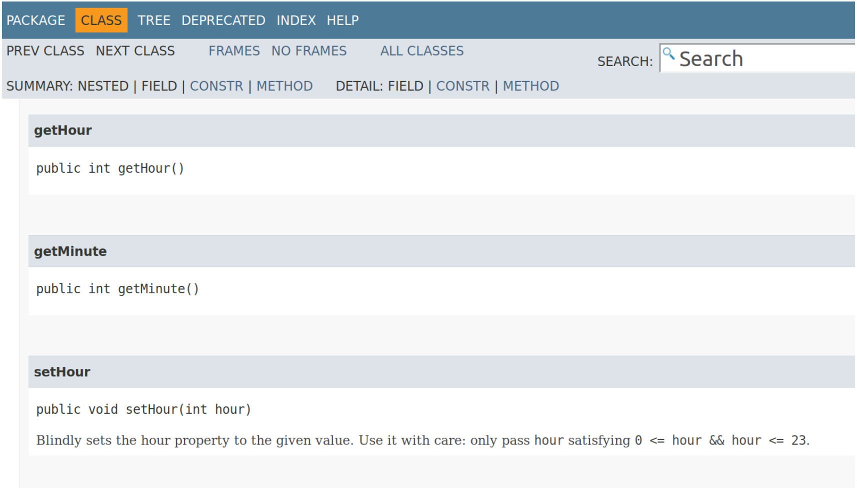
Constructor	Description
`Time()`	

Method Summary

All Methods **Instance Methods** **Concrete Methods**

Modifier and Type	Method	Description
`int`	`getHour()`	
`int`	`getMinute()`	
`void`	`oneMinutePassed()`	
`void`	`setHour(int hour)`	Blindly sets the hour property to the given value.

javadoc Time.java


PACKAGE CLASS TREE DEPRECATED INDEX HELP
PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES
SEARCH: Search
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

getHour

```
public int getHour()
```

getMinute

```
public int getMinute()
```

setHour

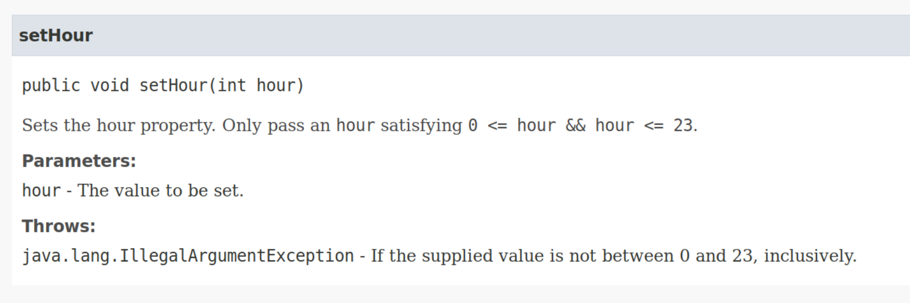
```
public void setHour(int hour)
```

Blindly sets the hour property to the given value. Use it with care: only pass `hour` satisfying `0 <= hour && hour <= 23`.

A typical (and stupidly verbose) doc comment

```
/**
 * Sets the hour property. Only pass an {@code hour}
 * satisfying {@code 0 <= hour && hour <= 23}.
 * @param hour The value to be set.
 * @throws IllegalArgumentException
 *    If the supplied value is not between 0 and 23,
 *    inclusively.
 */
public void setHour(int hour) {
    if (0 <= hour && hour <= 23) {
        this.hour = hour;
    } else {
        throw new IllegalArgumentException("Invalid hour!");
    }
}
```

javadoc Time.java



setHour

```
public void setHour(int hour)
```

Sets the hour property. Only pass an hour satisfying `0 <= hour && hour <= 23`.

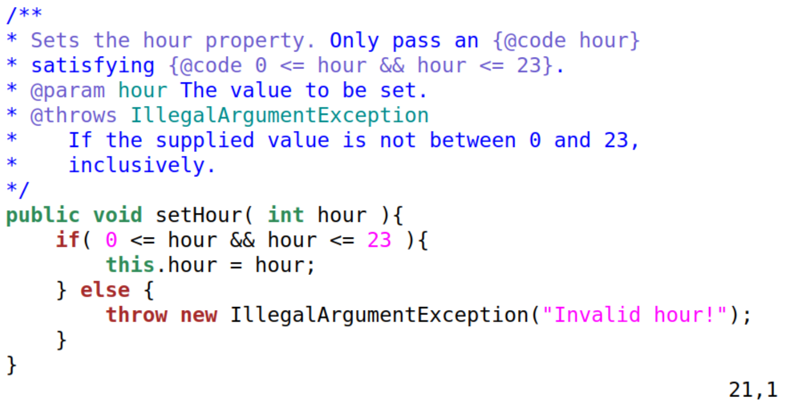
Parameters:

`hour` - The value to be set.

Throws:

`java.lang.IllegalArgumentException` - If the supplied value is not between 0 and 23, inclusively.

Syntax highlighting



```
/**
* Sets the hour property. Only pass an {@code hour}
* satisfying {@code 0 <= hour && hour <= 23}.
* @param hour The value to be set.
* @throws IllegalArgumentException
*    If the supplied value is not between 0 and 23,
*    inclusively.
*/
public void setHour( int hour ){
    if( 0 <= hour && hour <= 23 ){
        this.hour = hour;
    } else {
        throw new IllegalArgumentException("Invalid hour!");
    }
}
                                                        21,1
```

Implementing rational numbers

```java
package numbers;
public class Rational {
    private int numerator, denominator;
    /* class invariant: denominator > 0 */

    public Rational(int numerator, int denominator) {
        if (denominator <= 0) throw new IllegalArgumentException();
        this.numerator = numerator;
        this.denominator = denominator;
    }

}
```

Getter-setter

```
package numbers;
public class Rational {
    private int numerator, denominator;

    public Rational(int numerator, int denominator) { ... }

    public void setDenominator(int denominator) {
        if (denominator <= 0) throw new IllegalArgumentExcepti
        this.denominator = denominator;
    }

    public int getDenominator() { return denominator; }

    ...
}
```

How to use this class

```
import numbers.Rational;
public class Main {
    public static void main(String[] args) {
        Rational p = new Rational(1,3);
        Rational q = new Rational(1,2);
        p.multiplyWith(q);
        println(p);             // 1/6
        println(q);             // 1/2
    }
    private static void println(Rational r) {
        System.out.println(r.getNumerator()+"/"+r.getDenominator())
    }
}
```

Arithmetics

```java
package numbers;
public class Rational {
    private int numerator, denominator;
    public Rational(int numerator, int denominator) { ... }
    public int getNumerator() { return numerator; }
    public int getDenominator() { return denominator; }
    public void setNumerator(int numerator) { ... }
    public void setDenominator(int denominator) { ... }

    public void multiplyWith(Rational that) {
        this.numerator *= that.numerator;
        this.denominator *= that.denominator;
    }
    ...
}
```

Doc comment

```
package numbers;
public class Rational {
    ...
    /**
    *  Set {@code this} to {@code this} * {@code that}.
    *  @param that Non-null reference to a rational number,
    *              it will not be changed in the method.
    *  @throws NullPointerException When {@code that} is null
    */
    public void multiplyWith(Rational that) {
        this.numerator *= that.numerator;
        this.denominator *= that.denominator;
    }
    ...
}
```

Sequencing operations

```
package numbers;
public class Rational {
    ...
    public Rational multiplyWith(Rational that) {
        this.numerator *= that.numerator;
        this.denominator *= that.denominator;
        return this;
    }
    ...
}
```

```
Rational p = new Rational(1,3);
Rational q = new Rational(1,2);
p.multiplyWith(q).multiplyWith(q).divideBy(q);
println(p);
```

Class-wide method (function)

```
public class Rational {
    private final int numerator, denominator;
    public Rational(int numerator, int denominator) { ... }
    public int numerator(){ return numerator; }
    public int denominator(){ return denominator; }

    public static Rational times(Rational left, Rational righ
        return new Rational(left.numerator * right.numerator,
                            left.denominator * right.denominat
    }
}
```

```
Rational p = new Rational(1,3), q = new Rational(1,2);
Rational r = Rational.times(p,q);
```

Class-wide method (procedure)

```
public class Rational {
    private int numerator, denominator;
    ...
    public static void multiplyInPlace(Rational left,
                                       Rational right) {
        left.numerator *= right.numerator;
        left.denominator *= right.denominator;
    }
}
```

```
Rational p = new Rational(1,3), q = new Rational(1,2);
Rational.multiplyLeftWithRight(p,q);
```

A different approach

```
package numbers;
public class Rational {
    ...
    public void multiplyWith(Rational that) { ... }
    public Rational times(Rational that) { ... }
}
```

```
Rational p = new Rational(1,3);
Rational q = new Rational(1,2);
p.multiplyWith(q);
println(p);                 // 1/6
Rational r = p.times(q);
println(r);                 // 1/12
println(p);                 // 1/6
```

Implementation

```
package numbers;
public class Rational {
    private int numerator, denominator;
    public Rational(int numerator, int denominator) { ... }
    ...
    public Rational times(Rational that) {
        return new Rational(this.numerator * that.numerator,
                            this.denominator * that.denominator);
    }
    public void multiplyWith(Rational that) {
        this.numerator *= that.numerator;
        this.denominator *= that.denominator;
    }
}
```

Implementation

```
package numbers;
public class Rational {
    private int numerator, denominator;
    public Rational(int numerator, int denominator) { ... }
    ...
    public Rational times(Rational that) {
        return new Rational(this.numerator * that.numerator,
                            this.denominator * that.denominator);
    }
    public Rational multiplyWith(Rational that) {
        this.numerator *= that.numerator;
        this.denominator *= that.denominator;
        return this;
    }
}
```

There is no operator overloading in Java!

```
package numbers;
public class Rational {
    private int numerator, denominator;
    public Rational(int numerator, int denominator) { ... }
    ...
    public Rational operator*(Rational that) { // compilation error
        return new Rational(this.numerator * that.numerator,
                            this.denominator * that.denominator);
    }
    public Rational operator*=(Rational that) { // compilation erro
        this.numerator *= that.numerator;
        this.denominator *= that.denominator;
        return this;
    }
}
```

Object state never modified

```java
package numbers;
public class Rational {
    private int numerator, denominator;
    public Rational(int numerator, int denominator) {
        if (denominator <= 0) throw new IllegalArgumentException();
        this.numerator = numerator;
        this.denominator = denominator;
    }
    public int getNumerator() { return numerator; }
    public int getDenominator() { return denominator; }
    public Rational times(Rational that) { ... }
    public Rational  plus(Rational that) { ... }
    ...
}
```

Using unmodifiable fields

```
package numbers;
public class Rational {
    private final int numerator, denominator;
    public Rational(int numerator, int denominator) {
        if (denominator <= 0) throw new IllegalArgumentException();
        this.numerator = numerator;
        this.denominator = denominator;
    }
    public int getNumerator() { return numerator; }
    public int getDenominator() { return denominator; }
    public Rational times(Rational that) { ... }
    public Rational  plus(Rational that) { ... }
    ...
}
```

Multiple methods with the same name

```
public class Rational {
    ...
    public void multiplyWith(Rational that) {
        this.numerator *= that.numerator;
        this.denominator *= that.denominator;
    }

    public void multiplyWith(int that) {
        this.numerator *= that;
    }
}
```

```
Rational p = new Rational(1,3), q = new Rational(1,2);
p.multiplyWith(q);
p.multiplyWith(2);
```

Tricky rules: “matching more precisely”

```
static void m(long n) { ... }
static void m(float n) { ... }
public static void main(String[] args) {
    m(3);
}
```

Matching “equally precisely”

```
static void m(long n, float m) { ... }
static void m(float m, long n) { ... }
public static void main(String[] args) {
    m(4,2);
}
```

```
Foo.java:5: error: reference to m is ambiguous
        m(4,2);
        ^
  both method m(long,float) in Foo
   and method m(float,long) in Foo match
1 error
```

Multiple constructors in a class

```
public class Rational {
    ...
    public Rational(int numerator, int denominator) {
        if (denominator <= 0) throw new IllegalArgumentException();
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public Rational(int value) {
        numerator = value;
        denominator = 1;
    }
}
```

```
Rational p = new Rational(1,3), q = new Rational(3);
```

Overloading

- Class has multiple constructors or methods with the same name

Overloading

- Class has multiple constructors or methods with the same name
- Formal parameter list must be different
    - ◇ Number of parameters
    - ◇ Declared type of parameters

Overloading

- Class has multiple constructors or methods with the same name
- Formal parameter list must be different
    - ◇ Number of parameters
    - ◇ Declared type of parameters
- The compiler decides which method/constructor to call based on
    - ◇ number of actual parameters
    - ◇ declared (static) type of actual parameters

Overloading

- Class has multiple constructors or methods with the same name
- Formal parameter list must be different
  - ◇ Number of parameters
  - ◇ Declared type of parameters
- The compiler decides which method/constructor to call based on
  - ◇ number of actual parameters
  - ◇ declared (static) type of actual parameters
- Compilation error:
  - ◇ If no overloaded variant matches the call
  - ◇ If multiple overloaded variant equally matches the call

Overloading

- Class has multiple constructors or methods with the same name
- Formal parameter list must be different
  - ◇ Number of parameters
  - ◇ Declared type of parameters
- The compiler decides which method/constructor to call based on
  - ◇ number of actual parameters
  - ◇ declared (static) type of actual parameters
- Compilation error:
  - ◇ If no overloaded variant matches the call
  - ◇ If multiple overloaded variant equally matches the call
- Note: over**rid**ing is different from over**load**ing

Is this correct?

```
public class Rational {
    ...

    public void multiplyWith(Rational that) {
        this.numerator *= that.numerator;
        this.denominator *= that.denominator;
    }

    public Rational multiplyWith(Rational that) {
        this.numerator *= that.numerator;
        this.denominator *= that.denominator;
        return this;
    }
    ...
}
```

Meaningful overloading

```
public class Rational {
    ...
    public void set(int numerator, int denominator) {
        if (denominator <= 0) throw new IllegalArgumentException();
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public void set(Rational that) {
        if (that == null) throw new IllegalArgumentException();
        this.numerator = that.numerator;
        this.denominator = that.denominator;
    }
    ...
}
```

Default value for parameters?

```
public class Rational {
    ...
    public void set(int numerator, int denominator) {
        if (denominator <= 0) throw new IllegalArgumentException();
        this.numerator = numerator;
        this.denominator = denominator;
    }
    public void set(int value) {
        set(value,1);
    }
    public void set() {
        set(0);
    }
    ...
}
```

Default parameter values – Java does not have this

```
public class Rational {
    ...
    public Rational(int numerator = 0, int denominator = 1) {
        if (denominator <= 0) throw new IllegalArgumentException();
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public void set(int numerator = 0, int denominator = 1) {
        if (denominator <= 0) throw new IllegalArgumentException();
        this.numerator = numerator;
        this.denominator = denominator;
    }
    ...
}
```

Constructors may call each other

```
public class Rational {
    ...
    public Rational(int numerator, int denominator) {
        if (denominator <= 0) throw new IllegalArgumentException();
        this.numerator = numerator;
        this.denominator = denominator;
    }
    public Rational(int value) {
        this(value,1);              // this must be the first statem
    }
    public Rational() {
        this(0);
    }
    ...
}
```

Replacing constructors with factory methods

e.g. `Rational.zero()` instead of `new Rational(0)`

```
public class Rational {
    ...
    private Rational(int numerator, int denominator) {
        this.numerator = numerator;
        this.denominator = denominator;
    }
    public static Rational make(int numerator, int denominator) {
        return new Rational(numerator,denominator);
    }
    public static Rational valueOf(int val) {return make(val,1);}
    public static Rational oneOver(int den) {return make(1,den);}
    public static Rational zero() { return make(0,1); }
}
```

Parameter passing techniques

- Textual substitution
- Call-by-value
- Call-by-value-result
- Call-by-result
- Call-by-reference
- Call-by-sharing
- Call-by-name
- Call-by-need

Parameter passing in Java

Call-by-value

parameters of primitive types

```
public void setNumerator( int numerator ){
    this.numerator = numerator;
}
```

Call-by-sharing

parameters of reference types (the reference is passed by value)

```
public static void multiplyLeftWithRight( Rational left,
                                          Rational right ){
    left.numerator *= right.numerator;
    left.denominator *= right.denominator;
}
```

Call-by-value

```
public void setNumerator( int numerator ){
    this.numerator = numerator;
    numerator = 0;
}
```

```
Rational p = new Rational(1,3);
int two = 2;
p.setNumerator(two);
println(p);
System.out.println(two);
```

Call-by-sharing

```
public static void multiplyLeftWithRight( Rational left,
                                          Rational right ){
    left.numerator *= right.numerator;
    left.denominator *= right.denominator;
    left = new Rational(9,7);
}
```

```
Rational p = new Rational(1,3), q = new Rational(1,2);
Rational.multiplyLeftWithRight(p,q);
println(p);
```

Variable number of arguments

```
static int sum( int[] nums ){
    int s = 0;
    for( int n: nums ){ s += n; }
    return s;
}
                                 sum( new int[]{1,2,3,4,5,6} )
```

Variable number of arguments

```
static int sum( int[] nums ){
    int s = 0;
    for( int n: nums ){ s += n; }
    return s;
}
                          sum( new int[]{1,2,3,4,5,6} )
```

```
static int sum( int... nums ){
    int s = 0;
    for( int n: nums ){ s += n; }
    return s;
}
                                        sum(1,2,3,4,5,6)
```

Global constants

```
public static final int WIDTH = 80;
```

- static (class-wide) field
- bit similar to `#define` in C
- similar to `const` in C (not completely the same)
- convention: `ALL_CAPS` identifier

Final field

- E.g. global constant `WIDTH`
- Or instance fields of `Rational`
- Once assigned, never re-assigned
- Must be assigned during object initialisation
  - ◇ "blank final" is allowed

```
public class Rational {
    private final int numerator, denominator;
    public Rational(int numerator, int denominator) {
        this.numerator = numerator;
        this.denominator = denominator;
    }
...
```

Final local variable

```
public class Rational {
    ...
    public void simplify() {
        final int gcd = gcd(numerator, denominator);
        numerator /= gcd;
        denominator /= gcd;
    }
    ...
}
```

Final formal parameter

Erroneous

```
static java.math.BigInteger factorial(final int n) {
    assert n > 0;
    java.math.BigInteger result = java.math.BigInteger.ONE;
    while (n > 1) {
        result = result.multiply(java.math.BigInteger.valueOf(
        --n;
    }
    return result;
}
```

Final formal parameter

Correct

```
static java.math.BigInteger factorial(final int n) {
    assert n > 0;
    java.math.BigInteger result = java.math.BigInteger.ONE;
    for (int i=n; i>1; --i) {
        result = result.multiply(java.math.BigInteger.valueOf(
    }
    return result;
}
```

Mutable versus Immutable

Mutable object state

```
public class Rational {
    private int numerator, denominator;
    public Rational(int numerator, int denominator) { ... }
    public int getNumerator() { return numerator; }      ...
    public void setNumerator(int numerator) { ... }   ...
    public void multiplyWith(Rational that) { ... }
```

Immutable object state

```
public class Rational {
    private final int numerator, denominator;
    public Rational(int numerator, int denominator) { ... }
    public int getNumerator() { return numerator; }
    public int getDenominator() { return denominator; }
    public Rational times(Rational that) { ... }
```

An alternative naming convention

```java
public class Rational {
    private final int numerator, denominator;
    public Rational(int numerator, int denominator) { ... }
    public int numerator() { return numerator; }
    public int denominator() { return denominator; }
    public Rational times(Rational that) { ... }
}
```

```java
System.out.println(p.numerator() + "/" + p.denominator());
```

An alternative naming convention + mutable + overloading

```
public class Rational {

    private int numerator, denominator;

    public int numerator() { return numerator; }
    public void numerator(int numerator) {
        this.numerator = numerator;
    }

    ...
}
```

```
p.numerator(3);
System.out.println(p.numerator());
```

Public immutable object state

```
public class Rational {
    public final int numerator, denominator;
    public Rational(int numerator, int denominator) { ... }
    public Rational times(Rational that) { ... }
    ...
}
```

Hard to change representation!

Changing the representation

```
public class Rational {
    private final int[] data;
    public Rational(int numerator, int denominator) {
        if(denominator <= 0) throw new IllegalArgumentExceptio
        data = new int[]{ numerator, denominator };
    }
    public int numerator() { return data[0]; }
    public int denominator() { return data[1]; }
    public Rational times(Rational that) { ... }
}
```

Side note

```
int[] t = new int[3];
t = new int[4];

int[] s = {1,2,3};
s = {1,2,3,4};  // compilation error
s = new int[]{1,2,3,4};
```

final reference

```java
final Rational p = new Rational(1,2);
p.setNumerator(3);
p = new Rational(1,4); // compilation error
```

final reference

```
final Rational p = new Rational(1,2);
p.setNumerator(3);
p = new Rational(1,4); // compilation error

final int[] data = new int[2];
data[0] = 3;
data[1] = 4;
data = new int[3]; // compilation error
```

ELTE
IK

Representing character sequences

- `java.lang.String`: immutable

```
String num42 = "42";
String num42 = num42.reverse();
String num4224 = num42 + fourtytwo;
```

Representing character sequences

- java.lang.String: immutable

```
String num42 = "42";
String num42 = num42.reverse();
String num4224 = num42 + fourtytwo;
```

- java.lang.StringBuilder and java.lang.StringBuffer: mutable

```
StringBuilder sb = new StringBuilder("");
for (char c = 'a'; c <= 'z'; ++c) {
   sb.append(c).append(',');
}
sb.deleteCharAt(sb.length()-1); // cut last comma
String letters = sb.toString();
```

Representing character sequences

- java.lang.String: immutable

```
String num42 = "42";
String num42 = num42.reverse();
String num4224 = num42 + fourtytwo;
```

- java.lang.StringBuilder and java.lang.StringBuffer: mutable

```
StringBuilder sb = new StringBuilder("");
for (char c = 'a'; c <= 'z'; ++c) {
   sb.append(c).append(',');
}
sb.deleteCharAt(sb.length()-1); // cut last comma
String letters = sb.toString();
```

- char[]: mutable

Performance?

```
StringBuilder sb = new StringBuilder("");
for (char c = 'a'; c <= 'z'; ++c) {
  sb.append(c).append(',');
}
sb.deleteCharAt(sb.length()-1);
String letters = sb.toString();
```

```
String letters = "";
for (char c = 'a'; c <= 'z'; ++c) {
  letters += (c + ",");
}
letters = letters.substring(0,letters.length()-1);
```

A class definition that looks good, but···

```
package numbers;
public class Rational {
    ...
    public void multiplyWith(Rational that) {
        this.numerator *= that.numerator;
        this.denominator *= that.denominator;
    }
    public void divideBy(Rational that) {
        if (that.numerator == 0)
            throw new ArithmeticException("Division by zero!")
        this.numerator *= that.denominator;
        this.denominator *= that.numerator;
    }
}
```

What about not completely disjoint parameters?

```
package numbers;
public class Rational {
    ...
    public void divideBy(Rational that) {
        if (that.numerator == 0)
            throw new ArithmeticException("Division by zero!"
        this.numerator *= that.denominator;
        this.denominator *= that.numerator;
    }
}
```

```
Rational p = new Rational(1,2);
p.divideBy(p);
```

Escaping of object state

```
public class Rational {
    private int[] data;
    ...
    public int getNumerator() { return data[0]; }
    public int getDenominator() { return data[1]; }
    public void set(int[] data) {
        if (data == null || data.length != 2 || data[1] <= 0)
            throw new IllegalArgumentException();
        this.data = data;
    }
}
```

```
int[] cheat = {3,4};
Rational p = new Rational(1,2);  p.set(cheat);
cheat[1] = 0;       // p.getDenominator() == 0   :-(
```

Escaping object state because of clumsy construction

```
public class Rational {
    private final int[] data;
    public int getNumerator() { return data[0]; }
    public int getDenominator() { return data[1]; }
    public Rational(int[] data) {
        if (data == null || data.length != 2 || data[1] <= 0)
            throw new IllegalArgumentException();
        this.data = data;
    }
}
```

```
int[] cheat = {3,4};
Rational p = new Rational(cheat);
cheat[1] = 0;        // p.getDenominator() == 0   :-(
```

Escaping object state because of clumsy *getter*

```
public class Rational {
    private final int[] data;
    ...
    public int getNumerator() { return data[0]; }
    public int getDenominator() { return data[1]; }
    public int[] get() { return data; }
}
```

```
Rational p = new Rational(1,2);
int[] cheat = p.get();
cheat[1] = 0;         // p.getDenominator() == 0   :-(
```

Defensive copy

```
public class Rational {
    private final int[] data;
    public Rational(int[] data) {
        if (data == null || data.length != 2 || data[1] <= 0)
            throw new IllegalArgumentException();
        this.data = new int[]{ data[0], data[1] };
    }
    public void set(int[] data) { /* similarly */ }
    public int[] get() {
        return new int[]{ data[0], data[1] };
    }
}
```

Immutable objects need not be copied

```java
public class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        if (name == null || name.trim().isEmpty() || age < 0)
            throw new IllegalArgumentException();
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() { return age; }
    public void setName(String name) { ... this.name = name; }
    public void setAge(int age) { ... this.age = age; }
}
```

Aliasing in an array

```
Rational rats[2]; // compilation error

Rational rats[] = new Rational[2]; // = {null,null};

Rational[] rats = new Rational[2]; // preferred
rats[0] = new Rational(1,2);
rats[1] = rats[0];
rats[1].setDenominator(3);
System.out.println(rats[0].getDenominator());
```

- mutable versus immutable

Array may contain the same object multiple times

```
/**
    ...
    PRE: rats != null
    ...
*/
public static void increaseAllByOne(Rational[] rats) {
    for (Rational r: rats) {
        r.setNumerator(r.getNumerator() + r.getDenominator());
    }
}
```

Doc comment

```
/**
    ...
    PRE: rats != null and (i!=j => rats[i] != rats[j])
    ...
*/
public static void increaseAllByOne(Rational[] rats) {
    for (Rational r: rats) {
        r.setNumerator(r.getNumerator() + r.getDenominator());
    }
}
```

Arrays of arrays

- Java does not support multi-dimensional arrays (row- or column-first)
- Array of arrays (array of references)

```
int[][] matrix = {{1,0,0},{0,1,0},{0,0,1}};

int[][] matrix = new int[3][3];
for (int i=0; i<matrix.length; ++i) matrix[i][i] = 1;

int[][] matrix = new int[5][];
for (int i=0; i<matrix.length; ++i) matrix[i] = new int[i];
```

Aliasing again – seems like a bug

```
Rational[][] matrix =
    { {new Rational(1,2), new Rational(1,2)},
      {new Rational(1,2), new Rational(1,2)},
      {new Rational(1,2), new Rational(1,2)} };
```

```
Rational half = new Rational(1,2);
Rational[] halves = {half, half};
Rational[][] matrix = {halves, halves, halves};
```