编程基础 I

刘 钦

Outline

- 代码是用来读的（实践经验）
- 降低复杂度（方法学）
- 编程=数据结构+算法（理论逻辑）
- 算法建模（理论逻辑）
- 数据建模（理论逻辑）
- 编程范式（表现形式）
- Java基础语法（具体实现）
- 有代码就得有测试（实践经验）

Outline

- 代码是用来读的
- 降低复杂度
- 编程=数据结构+算法
- 算法建模
- 数据建模
- 编程范式
- Java基础语法
- 有代码就得有测试

代码是用来读的！

Write once,
Read many times

- “When you program, you have to think about how someone will read your code, not just how a computer will interpret it.”
- — kent beck

代码可读

- 团队的需要
- 维护的需要

糟糕的变量名字

- x = x - xx;
- xxx = aretha + SalesTax(aretha);
- x = x + LateFee(x1, x) + xxx;
- x = x + Interest(x1, x);

良好的变量名字

- balance = balance - lastPayment;
- monthlyTotal = NewPurchases + SalesTax(newPurchases);
- balance = balance + LateFee(customerID, balance) + monthlyTotal;
- balance = balance + Interest(customerID, balance);

代码是用来读的！

Outline

- 代码是用来读的
- 降低复杂度
- 编程=数据结构+算法
- 算法建模
- 数据建模
- 编程范式
- Java基础语法
- 有代码就得有测试

为什么要降低复杂度?

世界是复杂的?

程序员是干什么的?

将复杂的问题转化为代码

创建一个简单的消费计算程序

有什么降低复杂度的方法?

降低复杂度的方法 —

分解（组合）

*

//打印1颗*

System.out.print(“*”);

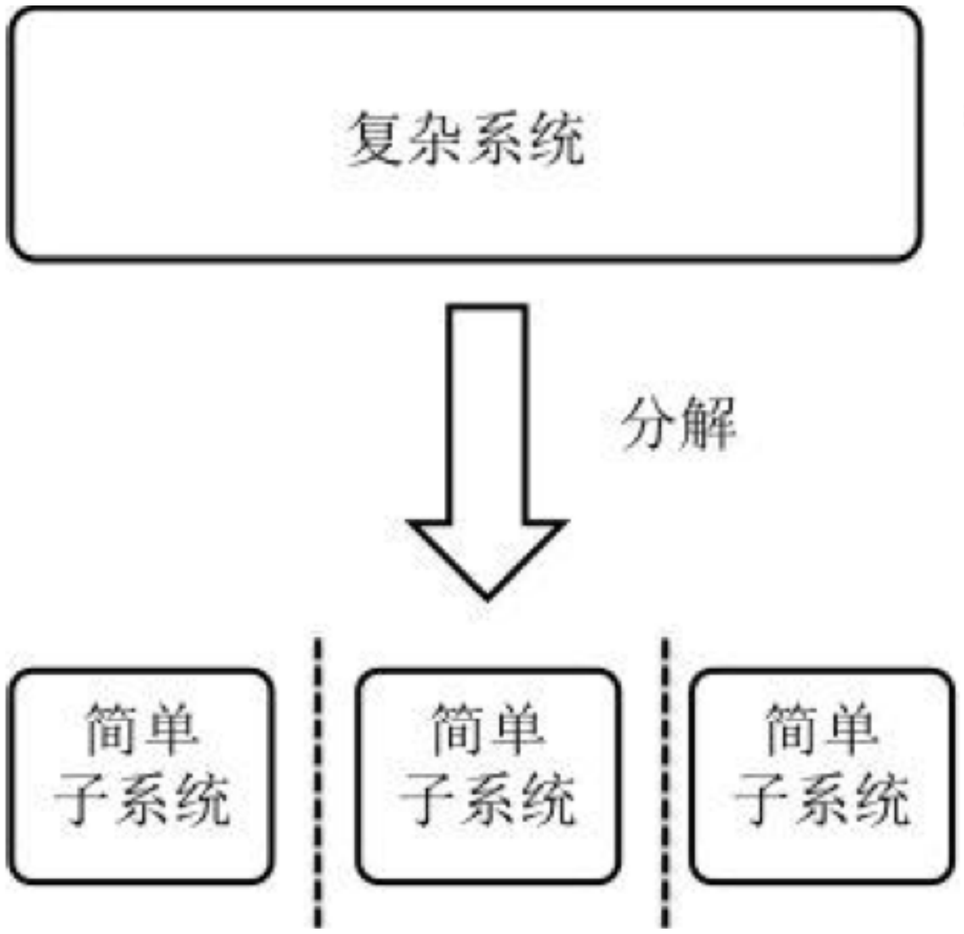
#

//打印1颗 #

System.out.print(“#”);

```
* #
//  打印＊ #
//=打印一颗* + 打印一颗#
```

System.out.print(“*”);
System.out.print(“ # ”);



分解（组合）

分解（组合）的关键点

- 分解之后，每一部分复杂度要变小，
- 相互之间关联要小，相对独立。

FIGURE 7.1 The design of a computer.

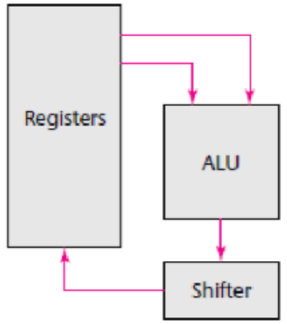


FIGURE 7.2 The computer of Figure 7.1 fabricated on three chips.

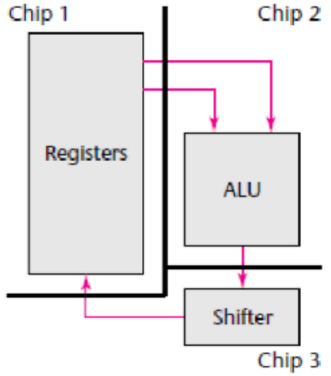
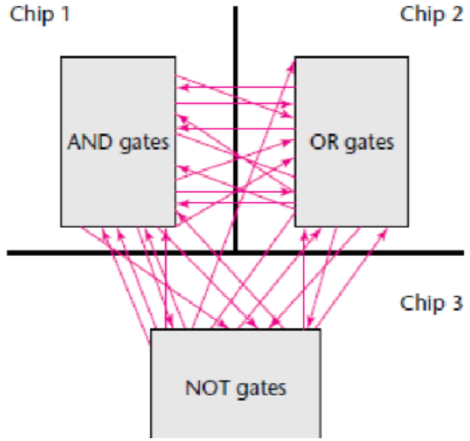


FIGURE 7.3 The computer of Figure 7.1 fabricated on three other chips.



好的分解和坏的分解

##

//打印2对 * #

```
System.out.print(“*”);
System.out.print(“ # ”);
System.out.print(“*”);
System.out.print(“ # ”);
```

##*#...

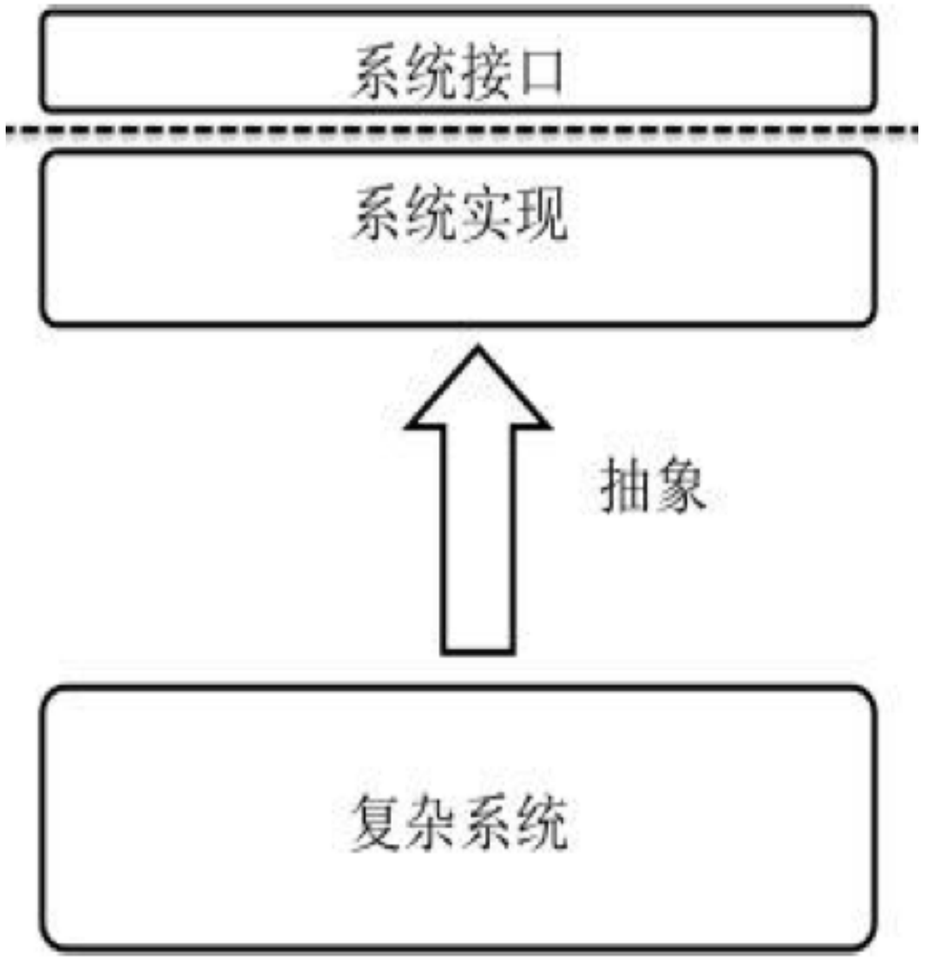
//打印100对 * #

我们愿意输入200遍么？

System.out.print(“*”);
System.out.print(“ # ”);
…
System.out.print(“*”);
System.out.print(“ # ”);

降低复杂度的方法 二

抽象

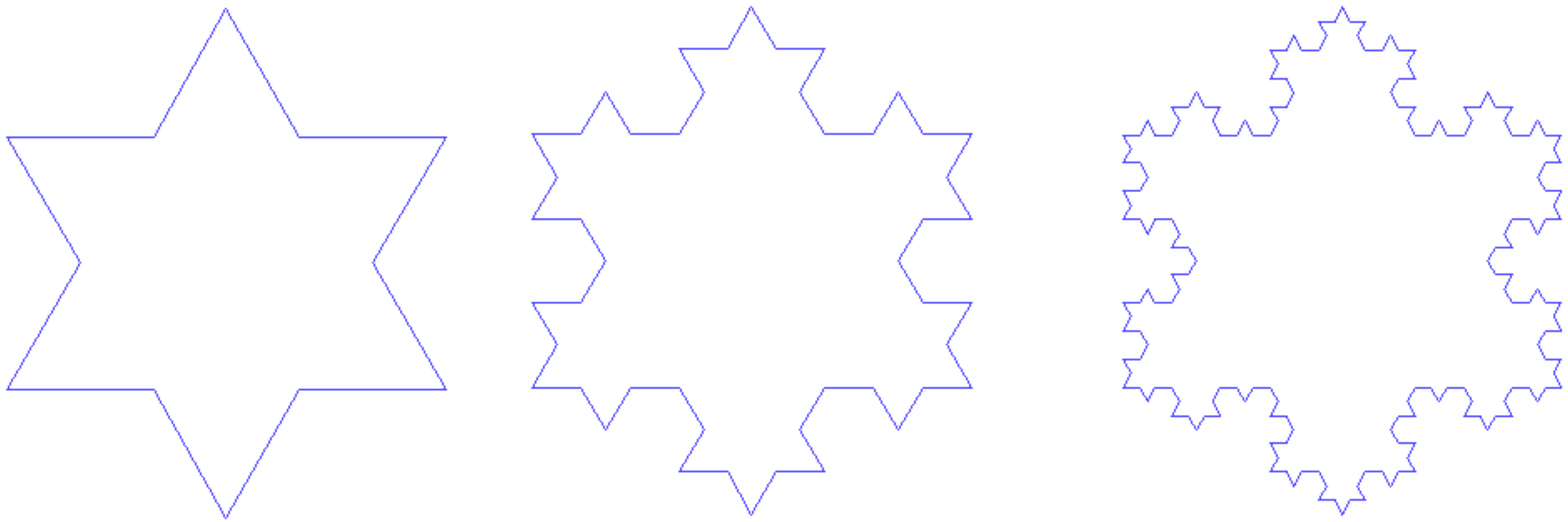


抽象

```
void printSingleStarSharp(){
    System.out.print(“*”);
    System.out.print(“ # ”);
}

void printStars(){
     printSingleStarSharp();
     printSingleStarSharp();
     …//100遍
}
```

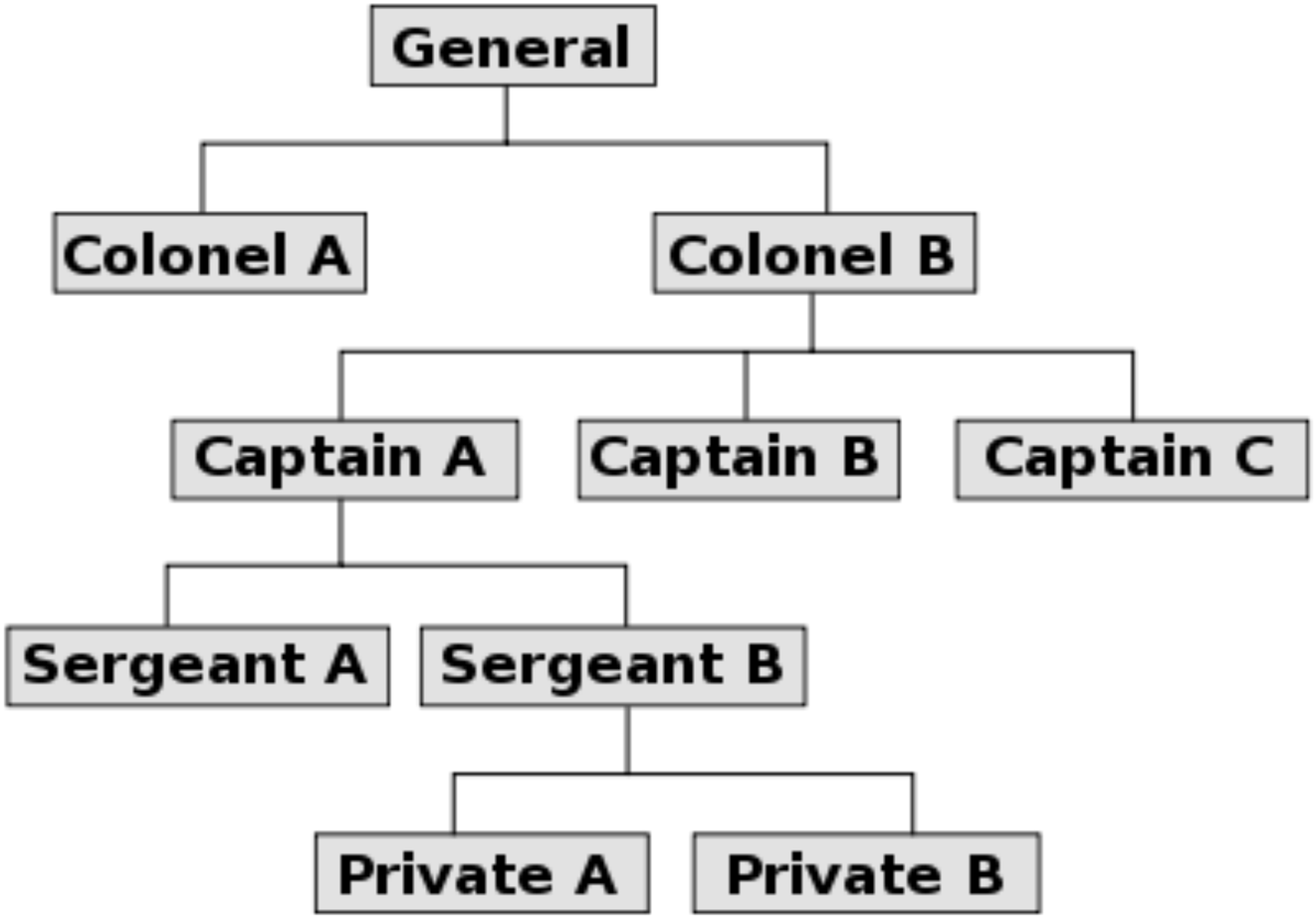
抽象的关键点

- 抽象之后，接口的复杂度变小，
- 接口和实现之间达成一种契约。

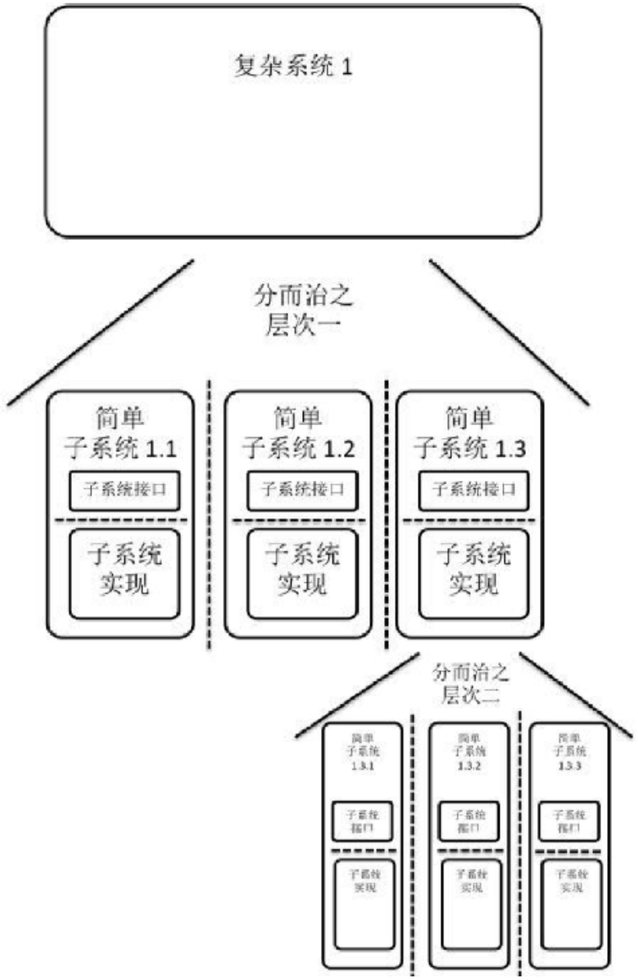


分形 — 科赫雪花

套娃


General
Colonel A
Colonel B
Captain A
Captain B
Captain C
Sergeant A
Sergeant B
Private A
Private B

组织结构



分解与抽象的并用和层次性

Outline

- 代码是用来读的
- 降低复杂度
- 编程=数据结构+算法
- 算法建模
- 数据建模
- 编程范式
- Java基础语法
- 有代码就得有测试

Program = Algorithm + Data Structure - 70s ~ 80s

Program

- Algorithms + Data Structures = Programs is a 1976 book written by Niklaus Wirth covering some of the fundamental topics of computer programming, particularly that algorithms and data structures are inherently related.

- For example, if one has a sorted list one will use a search algorithm optimal for sorted lists.

Outline

- 降低复杂度
- 编程=数据结构+算法
- 算法建模
  - 基本、分解（组合）与抽象的应用
  - 迭代与递归
- 数据建模
- 编程范式
- Java基础语法

每种语言都会提供的三种机制

- 基本表达式
- 分解（组合）的方法
- 抽象的方法

基本表达式

基本表达式

- 数字
  - 数学运算
  - 3 * 5
- 逻辑
  - 逻辑运算
  - True & False

分解（组合）

树形表示法

- （* （＋２（* ４６））（＋３５７））

抽象

复合

- 定义平方
  - (define (square x) (* x x))
- 定义平方和
  - (define (sum-of-squares x y)
  - (+ (square x) (square y)))

... //打印n颗星

参数与变量

- void printSingleStar(){
- 	System.out.print(“*”);
- }

- void printStars(int n){
- 	for(int i=0;i<n;i++)
- 		printSingleStar();
- }

参数 n

局部变量 i

线性的递归和迭代

- 计算n的阶乘
  - 递归
  - 迭代

递归

- (define (factorial n)
  - (if (= n 1)
    - 1
    - (* n (facorial (- n 1)))))

计算6！的线性递归过程

- (factorial 6)
- (* 6 (factorial 5))
- (* 6 (* 5 (factorial 4)))
- (* 6 (* 5 (* 4 (factorial 3))))
- (* 6 (* 5 (* 4 (* 3 (factorial 2)))))
- (* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
- (* 6 (* 5 (* 4 (* 3 (* 2 1)))))
- (* 6 (* 5 (* 4 (* 3 2))))
- (* 6 (* 5 (* 4 6)))
- (* 6 (* 5 24))
- (* 6 120)
- 720

迭代

- (define (factorial n)
  - (fact-iter 1 1 n)
- (define (fact-iter product counter max-count)
  - (if (> counter max-count)
    - product
    - (fact-iter (* counter product)
      - (+ counter 1)
      - max-count)))

计算6！的线性迭代过程

- (factorial 6)
- (fact-iter 1 1 6)
- (fact-iter 1 2 6)
- (fact-iter 2 3 6)
- (fact-iter 6 4 6)
- (fact-iter 24 5 6)
- (fact-iter 120 6 6)
- (fact-iter 720 7 6)
- 720

案例 - 利用牛顿法求平方根

- 做什么
  - y = 根号 x
  - x 的平方 = y
- 怎么做
  - 牛顿的逐步逼近方法

手动计算2的平方根

猜测	商	平均值
1	2/1=2	(2+1)/2=1.5
1.5	2/1.5=1.3333	(1.3333+1.5)/2=1.4167
1.4167	2/1.4167=1.4118	(1.4167+1.4118)/2=1.4142
1.4142	…	…

步骤

- sqrt
  - sqrt-iter
    - good-enough
      - square
      - abs
    - improve
      - average

Scheme语言版

- (define (sqrt-iter guess x)
  - (if (good-enough ? guess x)
    - guess
      - (squrt-iter (improve guess x)
      - x)))
- (define (improve guess x)
  - (average guess(/ x guess)))
- (define (average x y)
  - (/ (+ x y) 2))
- (define (good-enough ? guess x)
  - (< (abs (- (square guess) x)) 0.001))
- (define (squrt x)
  - (sqrt-iter 1.0 x)

C语言版

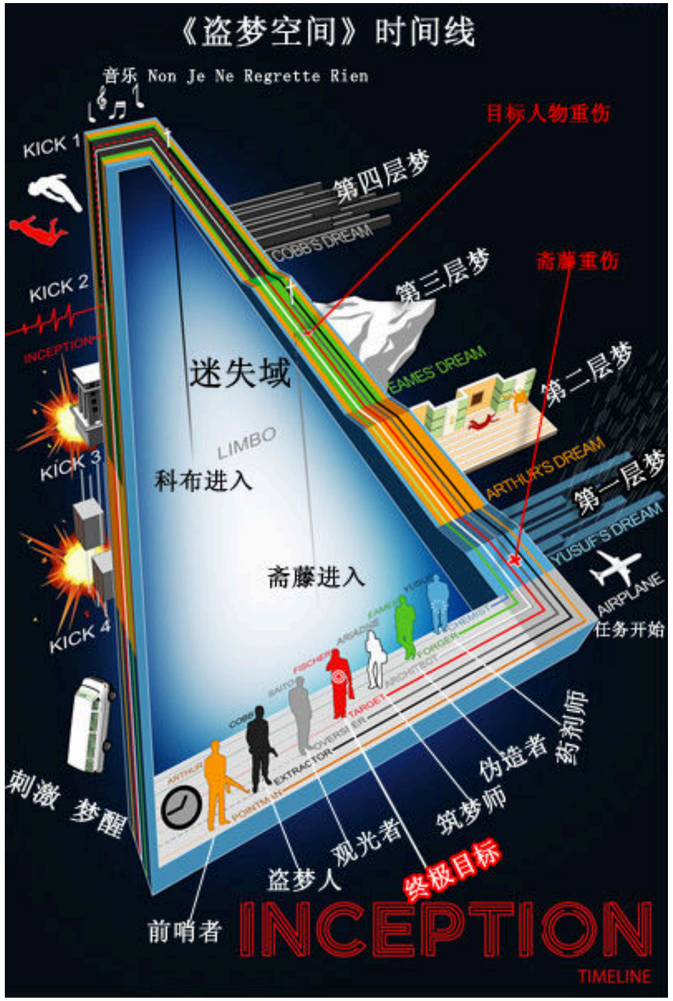
```
1 #define ABS(VAL) (((VAL)>0)?(VAL):(-(VAL)))
2 //用牛顿迭代法求浮点数的平方根
3 double mysqrt(float x) {
4     double g0,g1;
5     if(x==0)
6         return 0;
7     g0=x/2;
8     g1=(g0+x/g0)/2;
9     while(ABS(g1-g0)>0.01)
10    {
11        g0=g1;
12        g1=(g0+(x/g0))/2;
13    }
14    return g1;
15 }
```

对比

- (factorial 6)
- (* 6 (factorial 5))
- (* 6 (* 5 (factorial 4)))
- (* 6 (* 5 (* 4 (factorial 3))))
- (* 6 (* 5 (* 4 (* 3 (factorial 2)))))
- (* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
- (* 6 (* 5 (* 4 (* 3 (* 2 1)))))
- (* 6 (* 5 (* 4 (* 3 2))))
- (* 6 (* 5 (* 4 6)))
- (* 6 (* 5 24))
- (* 6 120)
- 720

- (factorial 6)
- (fact-iter 1 1 6)
- (fact-iter 1 2 6)
- (fact-iter 2 3 6)
- (fact-iter 6 4 6)
- (fact-iter 24 5 6)
- (fact-iter 120 6 6)
- (fact-iter 720 7 6)
- 720

对比


《盗梦空间》时间线
音乐 Non Je Ne Regrette Rien
目标人物重伤
KICK 1
第四层梦
CORB'S DREAM
KICK 2
第三层梦
斋藤重伤
INCEPTION
迷失域
EAMES' DREAM
第二层梦
LIMBO
KICK 3
科布进入
ARTHUR'S DREAM
第一层梦
YUSUF'S DREAM
斋藤进入
AIRPLANE
KICK 4
任务开始
刺激
梦醒
盗梦人
观光者
筑梦师
伪造者
药剂师
前哨者
终极目标
INCEPTION
TIMELINE


BLU-RAY™ + DVD + DIGITAL HD
ULTRAVIOLET
LIVE
VIVRE
"A SENSATIONAL RIDE."
DIE
MOURIR
"CRAZY, MIND-BLOWING AND LOADS OF FUN!"
REPEAT
RECOMMENCER
CRUISE/BLUNT/EDGEOFTOMORROW
UN JOUR SANS LENDEMAIN

Outline

- 降低复杂度
- 编程=数据结构+算法
- 算法建模
- 数据建模
- 编程范式
- Java基础语法

数据的组合

- 数组
  - 同一类数据的组合
- 结构体 struct
  - 不同数据的组合
- 对象
  - 数据和行为的组合

数据的抽象

- 有序对(Ordered Pair)

有序对的定义

- Wiener's definition

$$(a,b) := \{\{\{a\}, \emptyset\}, \{\{b\}\}\}.$$

- Haudorff's definition

$$(a,b) := \{\{a,1\},\{b,2\}\}$$

- Kuratowski definition

$$(a,\ b)_K\ :=\ \{\{a\},\ \{a,\ b\}\}.$$

有序对性质

Let (a_1, b_1) and (a_2, b_2) be ordered pairs. Then the characteristic (or *defining*) property of the ordered pair is:

$$(a_1, b_1) = (a_2, b_2) \quad \text{if and only if} \quad a_1 = a_2 \text{ and } b_1 = b_2.$$

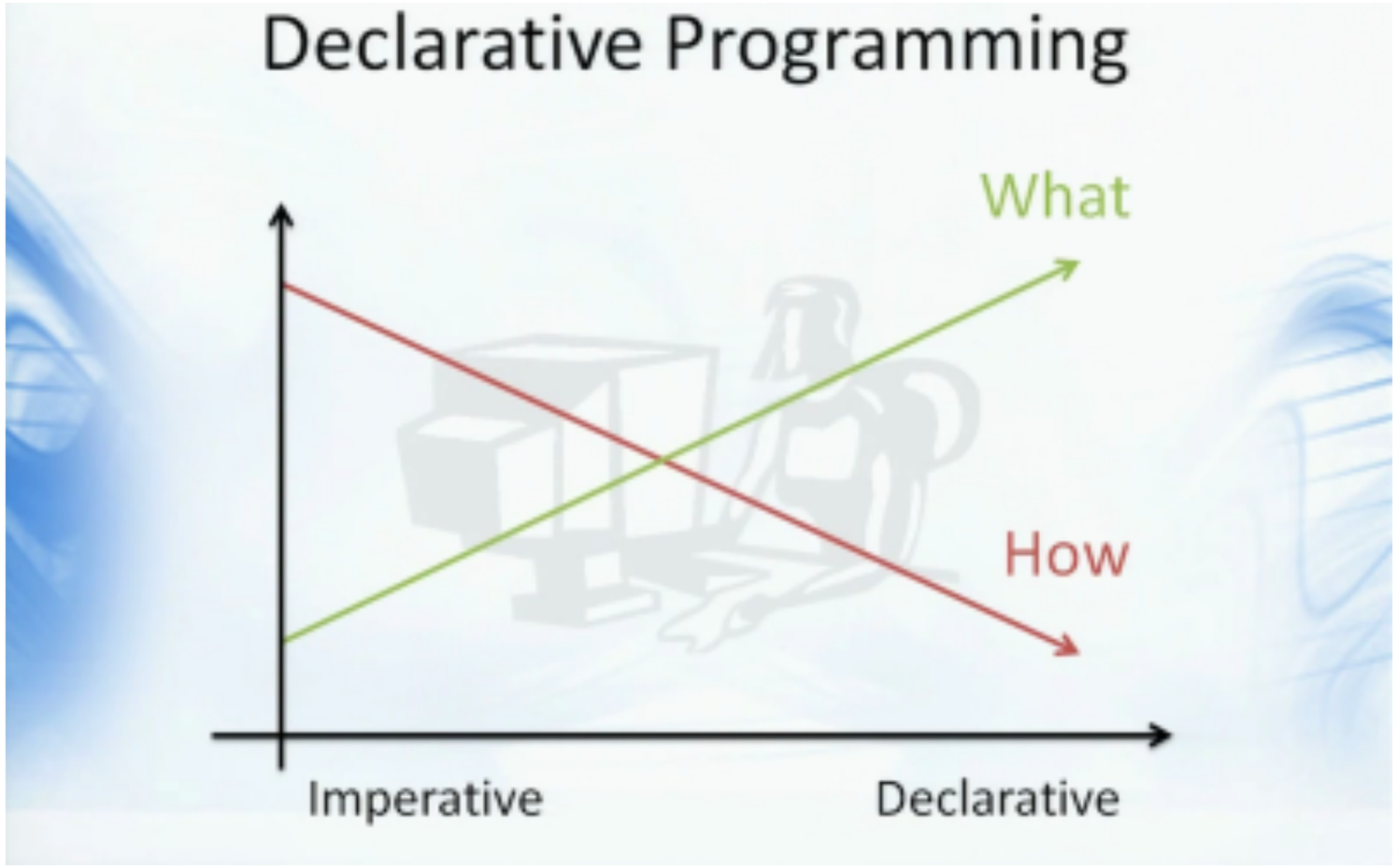
证明

- If a = b:
  - (a, b)K = {{a}, {a, b}} = {{a}, {a, a}} = {{a}}.
  - (c, d)K = {{c}, {c, d}} = {{a}}.
  - Thus {c} = {c, d} = {a}, which implies a = c and a = d. By hypothesis, a = b. Hence b = d.
- If a ≠ b, then (a, b)K = (c, d)K implies {{a}, {a, b}} = {{c}, {c, d}}.
  - Suppose {c, d} = {a}. Then c = d = a, and so {{c}, {c, d}} = {{a}, {a, a}} = {{a}, {a}} = {{a}}. But then {{a}, {a, b}} would also equal {{a}}, so that b = a which contradicts a ≠ b.
  - Suppose {c} = {a, b}. Then a = b = c, which also contradicts a ≠ b.
  - Therefore {c} = {a}, so that c = a and {c, d} = {a, b}.
  - If d = a were true, then {c, d} = {a, a} = {a} ≠ {a, b}, a contradiction. Thus d = b is the case, so that a = c and b = d.

Outline

- 降低复杂度
- 编程=数据结构+算法
- 算法建模
- 数据建模
- 编程范式
  - 命令式
  - 声明式
- Java基础语法

编程范式



Declarative vs Imperative

举个例子

- 展示单价超过20的Product对象，而是要查看每个分类中究竟有多少个单价超过20的对象，然后根据数量进行排序。

```
Dictionary<string, Grouping> groups = new Dictionary<string, Grouping>();
foreach (Product p in products)
{
    if (p.UnitPrice >= 20)
    {
        if (!groups.ContainsKey(p.CategoryName))
        {
            Grouping r = new Grouping();
            r.CategoryName = p.CategoryName;
            r.ProductCount = 0;
            groups[p.CategoryName] = r;
        }
        groups[p.CategoryName].ProductCount++;
    }
}

List<Grouping> result = new List<Grouping>(groups.Values);
result.Sort(delegate(Grouping x, Grouping y)
{
    return
        x.ProductCount > y.ProductCount ? -1 :
        x.ProductCount < y.ProductCount ? 1 :
        0;
});
```

Imperative

```
var result = products
    .Where(p => p.UnitPrice >= 20)
    .GroupBy(p => p.CategoryName)
    .OrderByDescending(g => g.Count())
    .Select(g => new { CategoryName = g.Key, ProductCount = g.Count() })
```

Declarative

Outline

- 代码是用来读的
- 降低复杂度
- 编程=数据结构+算法
- 算法建模
- 数据建模
- 编程范式
- Java基础语法
  - Java变量
  - 操作符
  - Java表达式、语句、块
  - 控制台输入
- 有代码就得有测试

变量

变量

- Primitive variable 基础数据类型
- Reference variable 引用变量

示例

```
// Compute the area of a circle。

class Area {

    public static void main（String args[]） {

        double pi，r，a;

       r = 10.8; // radius of circle

       pi = 3.1416; // pi，approximately

       a = pi * r * r; // compute area

       System.out.println(“Area of circle is " + a);

    }

}
```

基本数据类型

- 不用其他类型来定义的数据类型被称为基本数据类型。
- 几乎所有程序设计语言都提供一组基本数据类型。
  - 某些基本数据类型仅仅是硬件的反映，例如整数类型。而另一些基本类型只是在实现上需要一点非硬件的支持。

示例

```
// Compute the area of a circle。

class Area {

    public static void main（String args[]） {

        double pi，r，a;  //声明

       r = 10.8; // radius of circle //赋数值给变量

       pi = 3.1416; // pi，approximately

       a = pi * r * r; // compute area//赋变量的值给另一个变量

       System.out.println(“Area of circle is " + a);

    }

}
```

Java基本数据类型

- 整数：该组包括字节型（byte），短整型（short），整型（int），长整型（long），它们是有符号整数。
- 浮点型数：该组包括浮点型（float），双精度型（double），它们代表有小数精度要求的数字。
- 字符：这个组包括字符型（char），它代表字符集的符号，例如字母和数字。
- 布尔型：这个组包括布尔型（boolean），它是一种特殊的类型，表示真/假值。

- 基础数据类型是其他类型数据的基础。
- Java是完全面向对象的，但简单数据类型不是。
- 因为Java可移植性的要求，所有的数据类型都有一个严格的定义的范围。

整数类型

名称	长度	数的范围
长整型(long)	64	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
整型(int)	32	–2,147,483,648 ~ 2,147,483,647
短整型(short)	16	–32,768 ~ 32,767
字节型(byte)	8	–128~127

浮点类型

名称	位数	数的范围
double	64	1.7E–308～1.7E+308
float	32	3.4E–038～3.4E+038

命名

- 在JAVA中，标识符可以有不同的长度，但是它必须由字母、下划线（_），或者美元符（$）开头，而其余部分可以是除了JAVA运算符（比如+、-或者*）之外的任何字符，但是通常最好只使用字母、数字和下划线字符。
- Java是区分大小写
- 在名字中间不能包括空格或者制表符

常量

- 类常量的声明，应该全部大写，单词间用下划线隔开。例如
- static final int MIN_WIDTH = 4;
- static final int MAX_WIDTH = 999;
- static final int GET_THE_CPU = 1;

操作符

操作符

- Assignment
- Arithmetic Operators
- Unary Operators
- Equality and Relational Operators
- Conditional Operators
- Bitwise and Bit Shift Operators

简单赋值操作符

- int cadence = 0;
- int speed = 0;
- int gear = 1;

The Arithmetic Operators

- + additive operator (also used for String concatenation)
- - subtraction operator
- * multiplication operator
- / division operator
- % remainder operator

The Unary Operators

- + Unary plus operator; indicates positive value (numbers are positive without this, however)
- - Unary minus operator; negates an expression
- ++ Increment operator; increments a value by 1
- -- Decrement operator; decrements a value by 1
- ! Logical complement operator; inverts the value of a boolean

Equality and relational operators

- == equal to
- != not equal to
- > greater than
- >= greater than or equal to
- < less than
- <= less than or equal to

The Conditional Operators

- && Conditional-AND
- || Conditional-OR

Bitwise and bit shift operators

- The unary bitwise "~" inverts a bit pattern; it can be applied to any of the integral types, making every "0" a "1" and every "1" a "0
- The signed left shift operator "<<" shifts a bit pattern to the left,
- The signed right shift operator ">>" shifts a bit pattern to the right.
- The unsigned right shift operator ">>>" shifts a zero into the leftmost position, while the leftmost position after ">>" depends on sign extension.

- The bitwise & performs a bitwise AND operation.
- The bitwise ^ performs a bitwise exclusive OR operation.
- The bitwise | performs a bitwise inclusive OR operation.

表达式、语句、块

表达式（Expressions）

- An expression is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value.

语句（Statements）

- Expression statement
  - Statements are roughly equivalent to sentences in natural languages. A statement forms a complete unit of execution.
  - The following types of expressions can be made into a statement by terminating the expression with a semicolon (;).
    - Assignment expressions
    - Any use of ++ or --
    - Method invocations
    - Object creation expressions
- Declaration statement
- Control flow statement

块（Blocks）

- A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed.

控制台输入输出

```
public static void twoIntAdd(){
    int one=0;
    int two=0;
    String temp=null;
    System.out.println("please enter the first integer:");
    try {
      BufferedReader br1=new BufferedReader(new
 InputStreamReader(System.in));
      temp=br1.readLine();
      one=Integer.parseInt(temp);
      System.out.println("please enter the second
integer:");
      BufferedReader br2=new BufferedReader(new
InputStreamReader(System.in));
      temp=br2.readLine();
      two=Integer.parseInt(temp);
    } catch (IOException e) {
      e.printStackTrace();
    }
     System.out.println(one+"+"+two+"="+(one+two));
}
```

Outline

- 代码是用来读的（实践经验）
- 降低复杂度（方法学）
- 编程=数据结构+算法（理论逻辑）
- 算法建模（理论逻辑）
- 数据建模（理论逻辑）
- 编程范式（表现形式）
- Java基础语法（具体实现）
- 有代码就得有测试（实践经验）

源代码

- public String kindofTriangle (double side1, double side2, double side3)
- …
- return …;
- }

有代码就得有测试！

黑盒测试

黑盒测试

- 不了解程序的内部情况
- 只知道程序的输入、输出和系统的功能

黑盒测试一般流程

- 0. 初始化测试
- 1. 调用被测方法得到实际结果result
- 2. 与期望的结果相比较
- 3. 输出测试的结果

测试代码

- public void testRightTriangle（）{
- //0.初始化测试
- String testResult = “fail”;

- //1. 调用被测方法得到实际结果result
- String result = kindofTriangle(3.0, 4.0, 5.0);
- if(result.equals(“Right Triangle”)){ //2. 与期望的结果（Right Triangle）相比较
- testResult = “pass”; // 改变测试的结果
- }

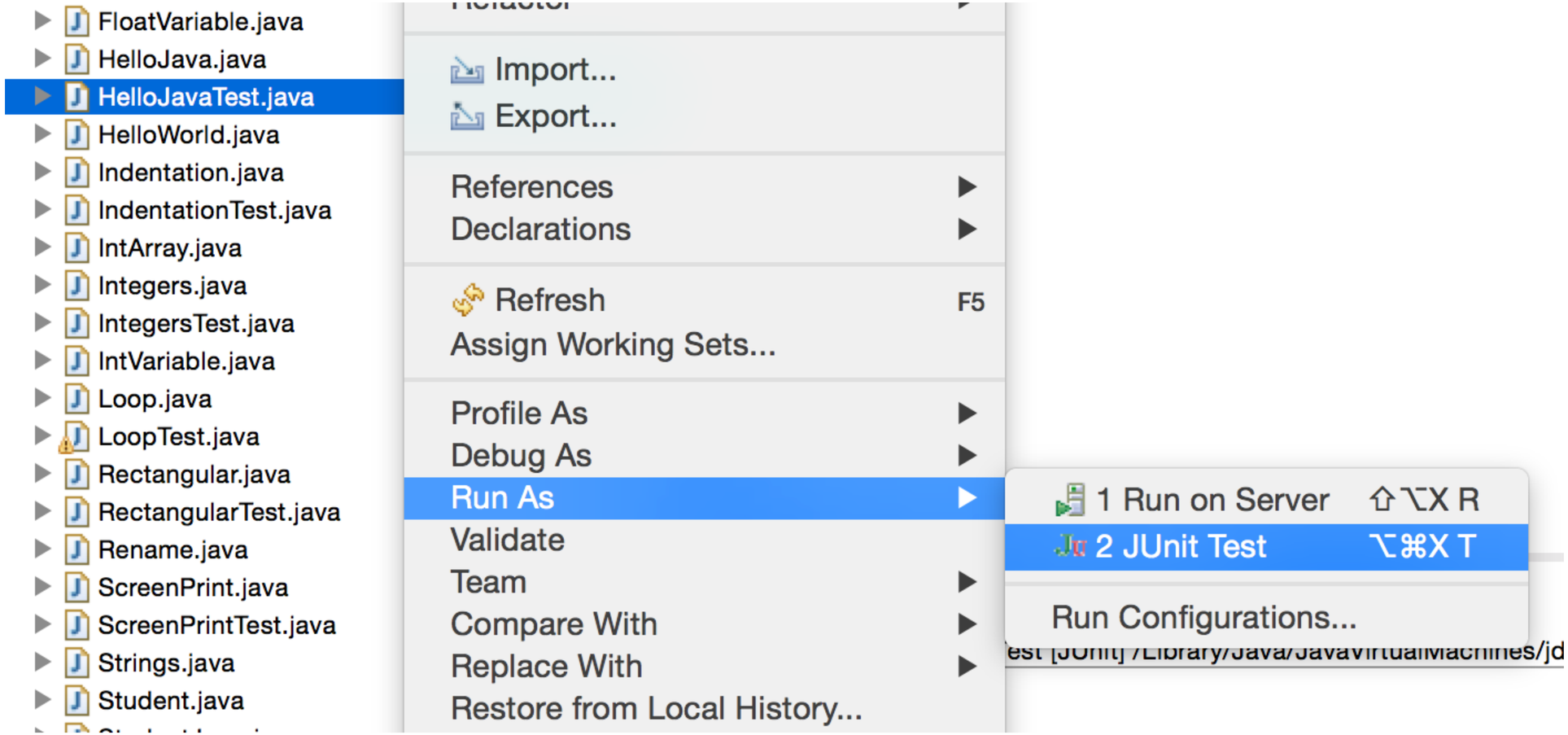
- //3. 输出测试的结果
- System.out.println(testResult);
- }

```java
public class HelloJava {

    public String hellojava() {
        //Submit the code without any revision
        return "Hello Java!";
    }

}
```

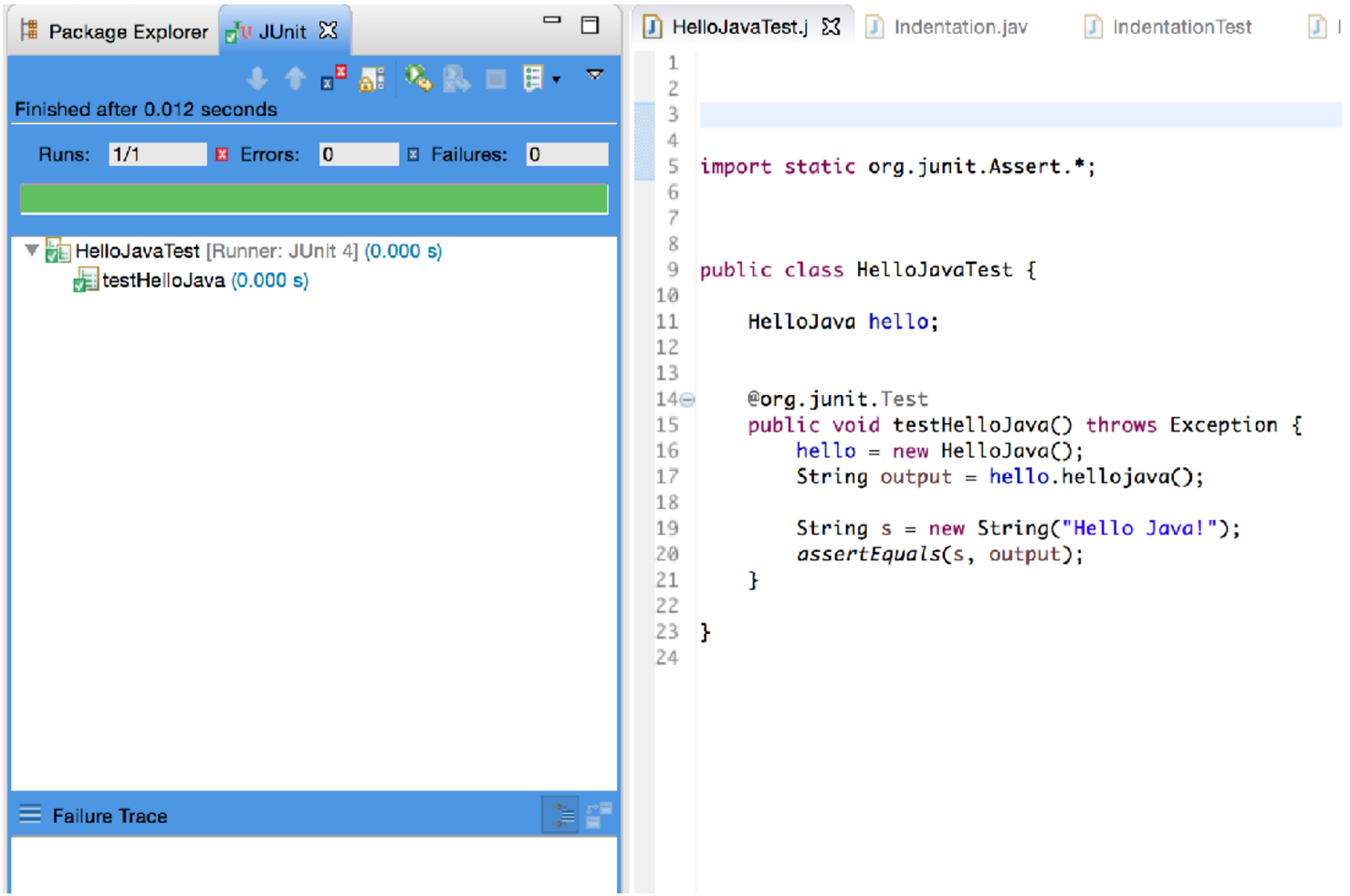
源程序

```java
import java.io.ByteArrayOutputStream;



public class HelloJavaTest {

    HelloJava hello;


    @org.junit.Test
    public void testHelloJava() throws Exception {
        hello = new HelloJava();
        String output = hello.hellojava();

        String s = new String("Hello Java!");
        assertEquals(s, output);
    }

}
```

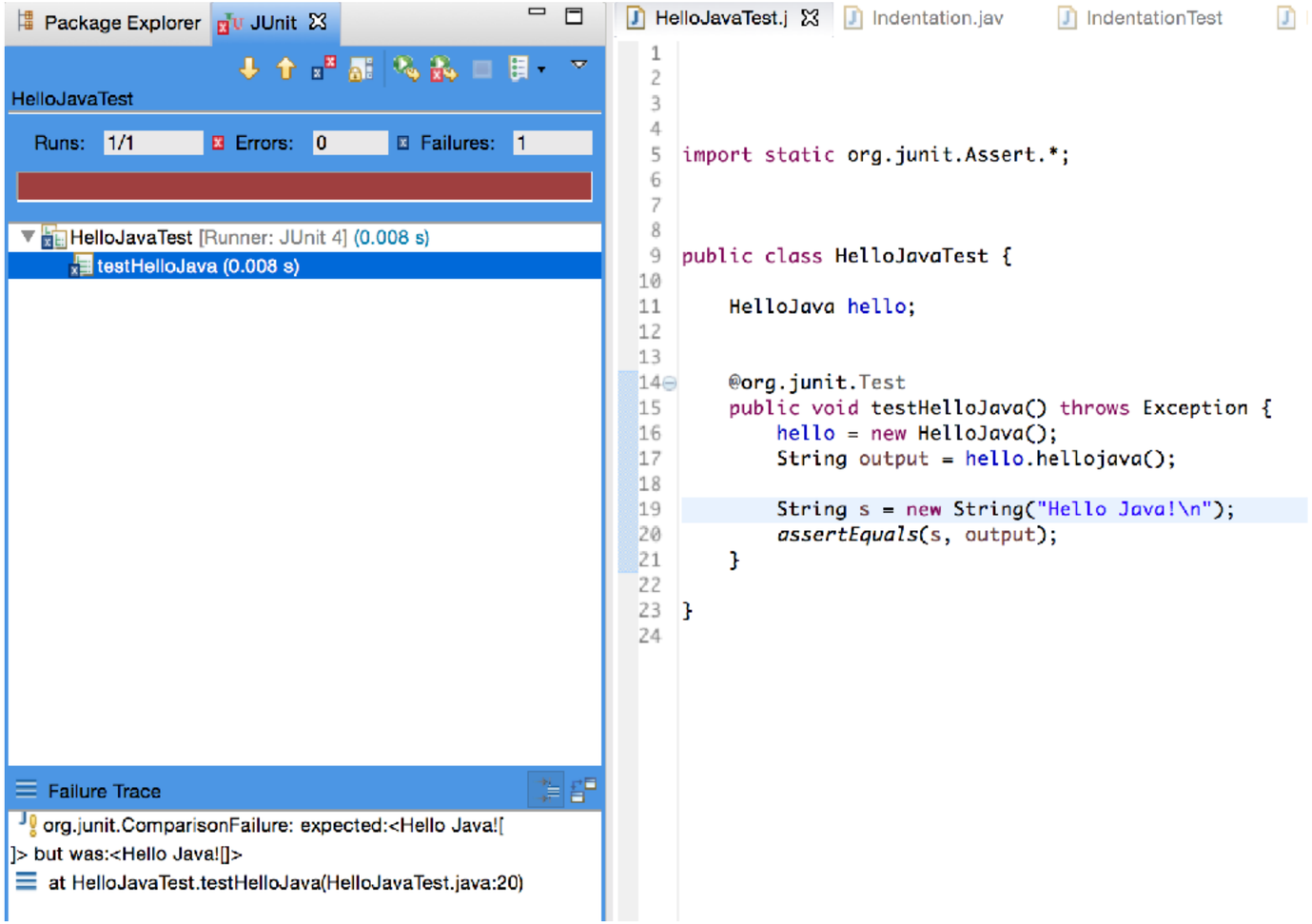
测试程序


FloatVariable.java
HelloJava.java
HelloJavaTest.java
HelloWorld.java
Indentation.java
IndentationTest.java
IntArray.java
Integers.java
IntegersTest.java
IntVariable.java
Loop.java
LoopTest.java
Rectangular.java
RectangularTest.java
Rename.java
ScreenPrint.java
ScreenPrintTest.java
Strings.java
Student.java
Import...
Export...
References
Declarations
Refresh
F5
Assign Working Sets...
Profile As
Debug As
Run As
Validate
Team
Compare With
Replace With
Restore from Local History...
1 Run on Server
2 JUnit Test
Run Configurations...

运行测试用例


Package Explorer
JUnit
Finished after 0.012 seconds
Runs: 1/1
Errors: 0
Failures: 0
HelloJavaTest [Runner: JUnit 4] (0.000 s)
testHelloJava (0.000 s)
Failure Trace
HelloJavaTest.j
Indentation.jav
IndentationTest
import static org.junit.Assert.*;
public class HelloJavaTest {
HelloJava hello;
@org.junit.Test
public void testHelloJava() throws Exception {
hello = new HelloJava();
String output = hello.hellojava();
String s = new String("Hello Java!");
assertEquals(s, output);
}
}

结果


Package Explorer
JUnit
HelloJavaTest
Runs: 1/1
Errors: 0
Failures: 1
HelloJavaTest [Runner: JUnit 4] (0.008 s)
testHelloJava (0.008 s)
Failure Trace
org.junit.ComparisonFailure: expected:<Hello Java![
]> but was:<Hello Java!>
at HelloJavaTest.testHelloJava(HelloJavaTest.java:20)
HelloJavaTest.j
Indentation.jav
IndentationTest
1
2
3
4
5 import static org.junit.Assert.*;
6
7
8
9 public class HelloJavaTest {
10
11 HelloJava hello;
12
13
14 @org.junit.Test
15 public void testHelloJava() throws Exception {
16 hello = new HelloJava();
17 String output = hello.hellojava();
18
19 String s = new String("Hello Java!\n");
20 assertEquals(s, output);
21 }
22
23 }
24

错误的结果