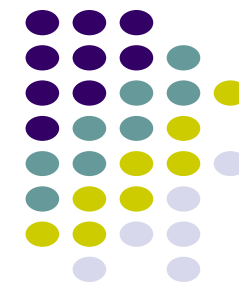


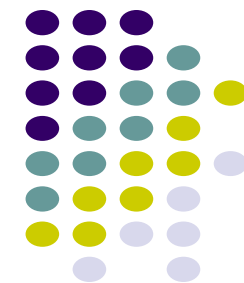
第二章 Java结构化程序设计



- **标识符、关键字**
- **数据类型、常量和变量**
- **运算符**
- **控制结构**
- **编程规范**

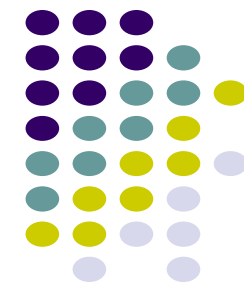


标识符 (Identifier)



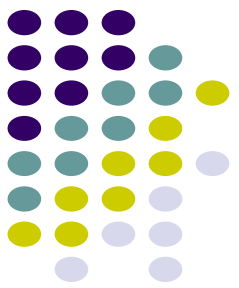
- **标识符**：是对变量、类、方法、标号和其它各种用户自定义对象的命名。
- **标识符规则**
 - 是一个由字符、数字、下划线（_）和美元符号（\$）构成的字符串。
 - 必须以字符、下划线（_）或美元符号（\$）开头，不能用数字开头。
 - 不能是保留字。
 - 不能是true、false和null（这三个不是保留字）。
 - 可以有任意的长度。
- **正确的标识符**
 - google、h1n1、_start_time、\$china、University
- **不正确的标识符**
 - 360safe、public、true、bill@gmail.com
- **汉字也可以作为标识符（但书写不便，不推荐）：**
 - `int 年龄 = 20;`

Java对标识符的规范



- 规范标识符的重要性：
 - 没有规范的命名不是一个科班出身的软件开发者应有的行为
- Java标识符规范：
 - 类名称：Mammal
 - 函数名：getAge
 - 常量：MAX_HEIGHT
- 标识符起名应该尽量做到“望名知义”。

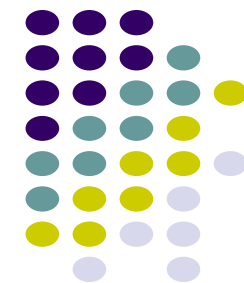
关键字 (reserved word、keyword)



- 对编译器具有特殊意义，在程序中不能用作其他目的的字，如：`class`、`public`、`static`、`void` 等。
- Java关键字列表(共50个):

<code>abstract</code>	<code>continue</code>	<code>for</code>	<code>new</code>	<code>switch</code>
<code>assert</code>	<code>default</code>	<code>goto</code>	<code>package</code>	<code>synchronized</code>
<code>boolean</code>	<code>do</code>	<code>if</code>	<code>private</code>	<code>this</code>
<code>break</code>	<code>double</code>	<code>implements</code>	<code>protected</code>	<code>throw</code>
<code>byte</code>	<code>else</code>	<code>import</code>	<code>public</code>	<code>throws</code>
<code>case</code>	<code>enum</code>	<code>instanceof</code>	<code>return</code>	<code>transient</code>
<code>catch</code>	<code>extends</code>	<code>int</code>	<code>short</code>	<code>try</code>
<code>char</code>	<code>final</code>	<code>interface</code>	<code>static</code>	<code>void</code>
<code>class</code>	<code>finally</code>	<code>long</code>	<code>strictfp</code>	<code>volatile</code>
<code>const</code>	<code>float</code>	<code>native</code>	<code>super</code>	<code>while</code>

第二章 Java结构化程序设计



- 标识符、关键字
- 数据类型、常量和变量
- 运算符
- 控制结构
- 编程规范



数据类型 (data types)



■ 基本数据类型 (primitive type)

- 布尔型——boolean
- 字符型——char
- 整型——byte、short、int、long
- 浮点型——float、double

■ 引用数据类型 (reference type)

- 数组——array
- 类——class
- 接口——interface
- 枚举——enum

常量 (constant)



- 在程序运行过程中，不许改变的变量，称为常量
- 利用关键字**final**声明常量，对于全局的常量（即在整个项目中都可用），通常按以下模式声明：**public static final** int MAX_VALUE=512;
- 如果某常量只在本类使用，则应将其定义为**private**的。
- 常量名字**通常采用大写字母，可以使用下划线**

```
public final double PI = 3.14159;  
public final double CM_PER_INCH = 2.54;  
public final int NOTEBOOK_WEIGHT;  
.....  
NOTEBOOK_WEIGHT = 2000;      // 第一次赋值  
NOTEBOOK_WEIGHT = 3000;      // 第二次赋值，错误！  
CM_PER_INCH = 3.00;          // 第二次赋值，错误！
```

- 注：在 C/C++ 语言中，使用 **const** 来定义常量。Java 中 **const** 也是保留字，但是没有用。

常量的数据类型



- Java常量包括基本数据类型常量、字符串(String)常量和null
 - 布尔(boolean)常量: true, false
 - char常量: 'c', '\u0061', '\u0051', '\u005a'
 - int常量: $(10)_{10} = (0xA)_{16} = (012)_8$, 34
 - long常量: 34L
 - double常量: 1.5, 45.6, 76.4E8, -32.0
 - float常量: 1.5F, 45.6f, 76.4E8F, -32.0F
 - String常量: "Hello World!"
 - 引用数据类型常量: null
- 独立于平台
 - Java中, 整型的范围与运行机器无关, 解决了移植时的问题
 - 在C/C++中, int表示与目标机器相关的整数类型

常量的数据类型



	类型	范围	存储空间大小
整型	byte	-2^7 (-128) ~ 2^7-1 (127)	8-bit signed
	short	-2^{15} (-32768) ~ $2^{15}-1$ (32767)	16-bit signed
	int	-2^{31} (-2147483648) ~ $2^{31}-1$ (2147483647)	32-bit signed
	long	-2^{63} (-9223372036854775808) ~ $2^{63}-1$ (9223372036854775807)	64-bit signed
浮点型	float	负数范围: $-3.40282347 \times 10^{38}$ ~ -1.4×10^{-45} 正数范围: 1.4×10^{-45} ~ $3.40282347 \times 10^{38}$	32-bit IEEE 754
	double	负数范围: $-1.7976931348623157 \times 10^{308}$ ~ -4.9×10^{-324} 正数范围: 4.9×10^{-324} ~ $1.7976931348623157 \times 10^{308}$	64-bit IEEE 754

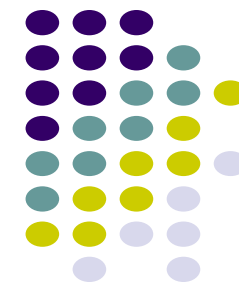
byte:

-128: 1000,0000 负数, 补码方式: $128=1000,0000 \Rightarrow$ (按位取反) $\Rightarrow 0111,1111 \Rightarrow$ (加1) $\Rightarrow 1000,0000$

-1: 1111,1111 负数, 补码方式: $1=0000,0001 \Rightarrow$ (按位取反) $\Rightarrow 1111,1110 \Rightarrow$ (加1) $\Rightarrow 1111,1111$

127: 0111,1111

常量的数据类型

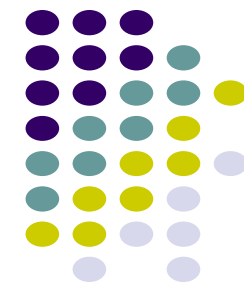


■ Java1.7的新特性

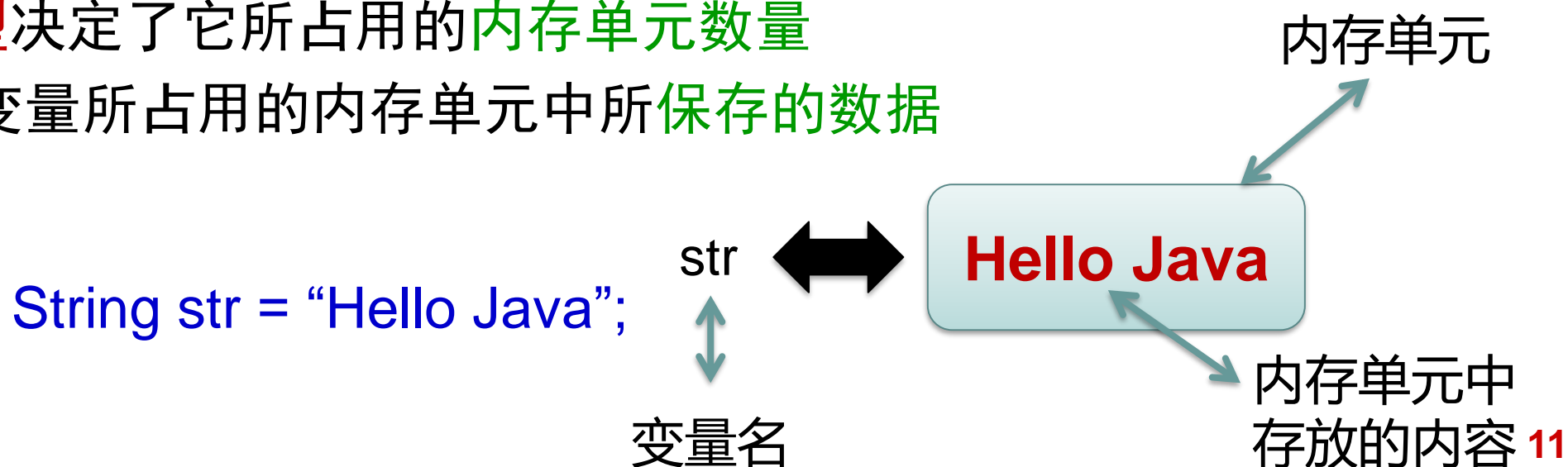
- 为了方便阅读大的整数，Java 7中现在允许使用下划线分隔多个数位：

```
int number1=1_000_000;  
int number2=1000000;  
  
System.out.println(number1==number2); //true  
//使用当前区域语言特性格式化数字  
  
NumberFormat format=NumberFormat.getInstance();  
  
System.out.println(format.format(number1)); //1,000,000
```

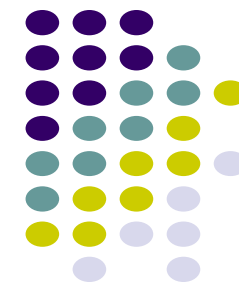
变量 (Variable)



- 变量表示Java程序中的基本存储单元，总是具有某种数据类型：
 - 基本数据类型
 - 引用数据类型
- 变量总是具有与其数据类型相对应的值
- 每个变量均具有：**名字、类型、一定大小的存储单元以及值**
 - 变量就是内存单元的**名字**，即对应内存的**位置**
 - **数据类型**决定了它所占用的**内存单元数量**
 - **值** 表示变量所占用的内存单元中所**保存的数据**



变量



■ 变量的读写

- 当新值被赋给变量时，老值将被取代，仅从内存中读数据不会破坏数据

■ 变量的使用

- 变量在使用前应保证它有确切的值。
- 同名变量的屏蔽原则。
- 在实际开发中，一般使用变量来存储用户在程序运行时输入的数据。

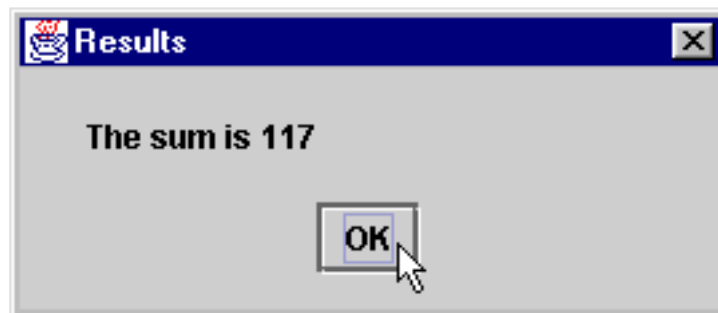
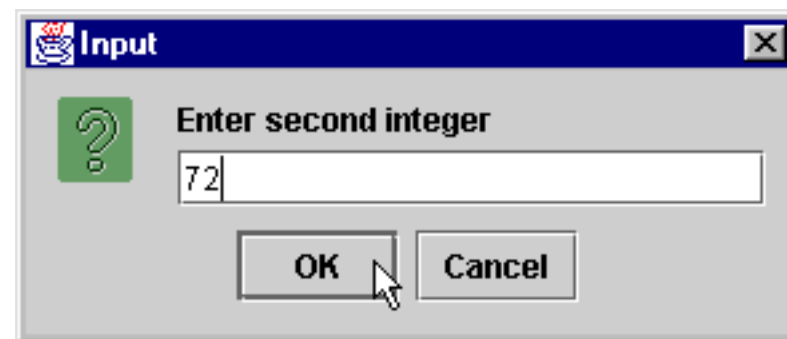
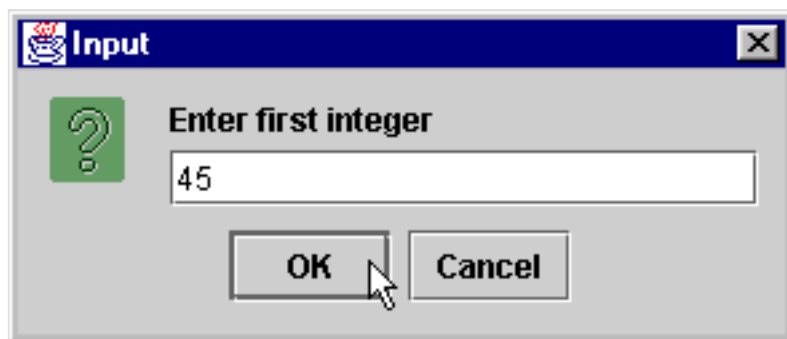
■ 举例：在Java应用程序中如何读入数据。

Java应用程序读入数据

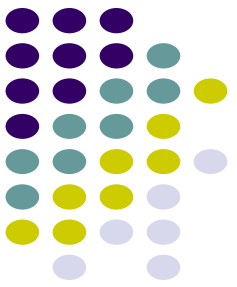


- 在运行时读取用户输入
 - 方法一：使用JOptionPane类的showInputDialog方法。

```
String firstNumber =  
    JOptionPane.showInputDialog("Enter :" );
```



Java应用程序读入数据

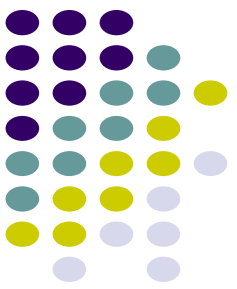


- 方法二：使用Scanner类（JDK1.5以上）

```
Imports java.util.*;  
Scanner in=new Scanner(System.in);  
System.out.print("What is your name?");  
String name=in.nextLine();
```

- Scanner类有nextInt, nextDouble等方法。
- 实例：InputTest.java

浮点数据的精度



■ 浮点数的精度？

- 包含浮点数的计算是近似的，因为这些数没有以完全的准确度存储。例如：

- `System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);`

显示的是0.50000000000000001而不是0.5

- `System.out.println(1.0 - 0.9);`

显示的是0.09999999999999998而不是0.1

- 整数可以精确地存储。因此，整数计算得到的是精确的运算结果

■ 注意：浮点数的比较问题 Demo:Test.java

```
double n1 = 0.1;
double n2 = 1.0 - 0.9;
if (n1 == n2) {
    System.out.println("equal!");
}
else {
    System.out.println("not equal!");
}
```



```
double n1 = 0.1;
double n2 = 1.0 - 0.9;
if (Math.abs(n1-n2) < 0.00000001) {
    System.out.println("equal!");
}
else {
    System.out.println("not equal!");
}
```

浮点数据的精度



■ Demo: `TestDouble.java`

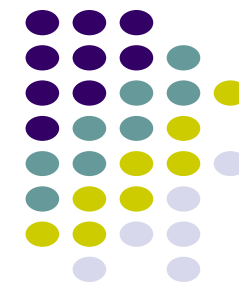
`0.05 + 0.01 = 0.060000000000000000000005`

`1.0 - 0.42 = 0.58000000000000000001`

`4.015 * 100 = 401.499999999999994`

`123.3 / 100 = 1.232999999999999999`

浮点数据的精度



■ 处理精度损失 (*)

- 解决方法——使用BigDecimal类
- Demo: [TestBigDecimal.java](#)

- 注意：在构建BigDecimal对象时应使用字符串而不是double数值，否则，仍有可能引发计算精度问题。

```
Problems @ Javadoc Declaration Console X
<terminated> TestBigDecimal [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (2013-5-13 下午3:19:56)
下面使用String作为BigDecimal构造器参数的计算结果：
0.05 + 0.01 = 0.06
0.05 - 0.01 = 0.04
0.05 * 0.01 = 0.0005
0.05 / 0.01 = 5
下面使用double作为BigDecimal构造器参数的计算结果：
0.05 + 0.01 = 0.060000000000000000277555756156289135105907917022705078125
0.05 - 0.01 = 0.040000000000000000277555756156289135105907917022705078125
0.05 * 0.01 = 0.00050000000000000000277555756156289135105907917022705078125
0.05 / 0.01 = 5.0000000000000000277555756156289135105907917022705078125
```

字符类型



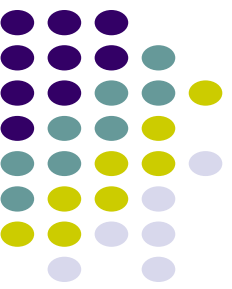
■ 字符类型使用单引号括起来

```
char letter = 'A';           // ASCII
char number = '4';           // ASCII
char unicodeLetter = '\u0041'; // Unicode
char unicodeNumber = '\u0034'; // Unicode
char cnChar1 = '中';
char cnChar2 = '国';
```

- 注意，双引号括起来的是字符串。'A'和"A"是不同的
- char类型用来表示在Unicode编码表中的字符
- 增量和减量运算也可以用于char型变量，得到后一个或前一个Unicode字符，如：

```
char letter = 'A';           // ASCII
System.out.println(++letter);
```

字符类型



■ Unicode

- 由Unicode Consortium建立的一种16位编码方案，支持世界不同语言的文本交换、处理和显示。
- 一个Unicode码占两个字节，书写上用以\u开头的4位十六进制数表示，范围从 '\u0000' 到 '\uFFFF'，共65536个
- 目前大约使用了35000个
- 例子：

Demo: UnicodeDisplay

```
public class UnicodeDisplay
{
    /** Main method */
    public static void main(String[] args)
    {
        // 显示: 欢迎αβγ
        System.out.println( "\u6B22\u8FCE\u03b1\u03b2\u03b3" );
    }
}
```

字符类型



■ 特殊字符的转义序列

运算符	含义	Unicode值
<code>\b</code>	退格	<code>\u0008</code>
<code>\t</code>	Tab	<code>\u0009</code>
<code>\n</code>	换行	<code>\u000A</code>
<code>\r</code>	回车	<code>\u000D</code>
<code>\\</code>	反斜杠	<code>\u005C</code>
<code>\'</code>	单引号	<code>\u0027</code>
<code>\"</code>	双引号	<code>\u0022</code>

```
char letter1 = '\'';           // 单引号
char letter2 = '\\"';          // 双引号
char letter3 = '\\';           // 反斜杠
```



■ boolean类型

- 基本数据类型（同int、double一样）
- 只有两个取值：true（真）、false（假）
- 整型数据和boolean型不能相互转换
 - C/C++中是可以相互转换的。
 - 在C/C++中，数字、指针都可以充当boolean值。但是这种灵活性带来了潜在的灾难，无数程序的bug也出现在这里。

字符串类型 (String)



- char类型只表示一个字符，要表示一个字符构成的串，就应使用字符串 (String)，如：
 - String message = "Hello, world";
 - 注意String的S要大写
 - 字符串由双引号括起来
- String是Java的一个预定义类，String不是基本类型，称为引用类型。
- 任何Java类都可以作为变量的引用类型
- 目前，只需要知道如何声明一个String型变量，如何将字符串赋值给变量以及如何连接字符串。

字符串类型



■ 字符串连接

- 使用 “+” 号连接

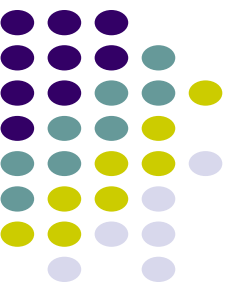
```
String welcome = "Hello, " + "world";
```

```
String age = "Age is " + 20; // 字符串和整数可以直接连接起来!
```

```
String multipleAdd = "My " + "name " + "is " + "Julie Mao"; // 可以连加
```

- 将String 和其它数据类型相加，结果是一个新的String

枚举数据类型



■ 定义：

- `enum Size{SMALL, MEDIUM, LARGE}`

■ 使用：

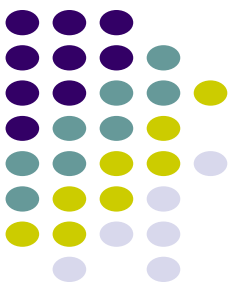
```
Size s=Size.SMALL;
```

```
//从字符串转换为枚举
```

```
Size t=Size.valueOf("SMALL") ;
```

- 注意： 枚举类型适用于JDK 5.0及更新的版本
- 枚举类型是引用类型，枚举不属于原始数据类型，它的每个值都引用一个不同的对象。
 - 两个枚举型对象比较
 - **Demo: EnumTest.java**

数值类型转换



■ 隐式转换——自动转换是安全的

- `double d = 3;` (拓宽类型 `type widening`)

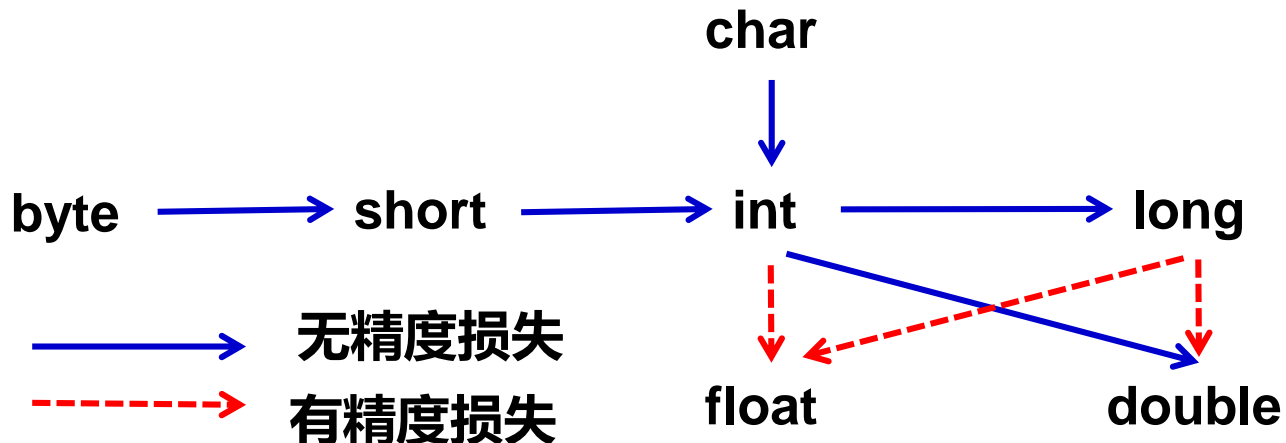
■ 显式转换——强制转换可能引起信息丢失

- `int i = (int)3.0;` (缩窄类型 `type narrowing`)
- `int i = (int)3.9;` (截掉小数部分)

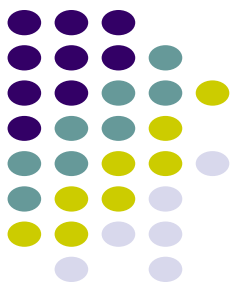
■ 拓宽类型不需要明确指出，可以自动执行；缩窄类型必须明确地指明

■ 错误的例子：

```
int x = 5 / 2.0;
```



举例



■ 保留小数点后面两位（四舍五入）

- 假设营业额为1197.64，税率为6%，应缴税款为何（保留小数点后面两位，四舍五入）？ Demo: [TaxCalculator.java](#)

```
public class TaxCaculator
{
    /** Main method */
    public static void main(String[] args)
    {
        double income = 1197.64;
        double taxRate = 0.06;
        double tax = income * taxRate; // tax = 71.8584
        double trueTax = ((int)((tax + 0.005) * 100)) / 100.0;
        System.out.println("tax = " + trueTax); // tax = 71.86
    }
}
```

注：1、由于要保留小数点后面两位，因此先乘以100，取整（截去小数部分），再除以100
2、由于要四舍五入，因此加上0.005

另一种数据类型转换方法



■ 字符串转换为数值

- 123 和 "123" 是不一样的
- 字符串 → int: 用 **Integer** 类中的 **parseInt** 方法
int → 字符串: 用 **String** 类中的 **valueOf** 方法

```
int intValue = Integer.parseInt("123");    // String → int  
String stringValue = String.valueOf(123);  // int → String
```

- 字符串 → double: 用 **Double** 类中的 **parseDouble** 方法
double → 字符串: 用 **String** 类中的 **valueOf** 方法

```
double doubleValue = Double.parseDouble("123.0"); // String → double  
String doubleValue = String.valueOf(123.0);       // double → String
```

注意: int/double 是基本数据类型; Integer/Double是引用类型

另一种数据类型转换方法



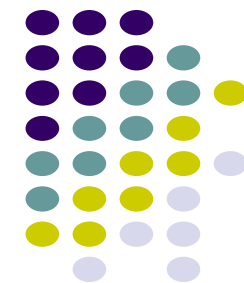
- 上例：类Integer 属于包 java.lang，它“封装”了一个int类型的整数，因此，它是原始数据类型int的“包装类”。
- 除了使用C的强制类型转换方式，我们还可以通过**原始类型**的**包装类**完成类型转换：

```
double doubleValue = 156.5d;  
Double doubleObj = new Double(doubleValue);  
byte myByteValue = doubleObj.byteValue();  
int myIntValue = doubleObj.intValue();  
float myFloatValue = doubleObj.floatValue();  
String myString = doubleObj.toString();
```

适用场景：

- 同一个数据需要转换为多种类型，并且这一数据需要比较长期的使用。
- 多数情况下，推荐直接使用强制类型转换的方式。

简单数据类型的包装类



■ java.lang包中有类：

- Boolean
- Character
- Byte
- Double
- Float
- Integer
- Long
- Short

Java两种类型的变量



■ Java中有两种类型的变量

- 引用类型变量（Reference variables）

- 引用一个对象（变量本身用于存放对象在内存中的位置，
可以看成是一个指针），故又称为“对象变量”

- 原始数据类型变量，变量中仅包含数据。

Java两种类型的变量

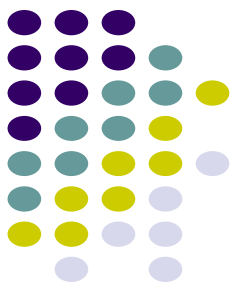


■ 区分引用类型和原始类型的变量

```
String firstNumber;           // first string entered by user
String secondNumber;          // second string entered by user
double number1;                // first number to add
double number2;                // second number to add
```

- 如果变量数据类型是一个类的名字，就是引用类型变量，它将引用一个对象。
 - **String** 是个**类**（注意类名首字母大写）
 - **firstNumber, secondNumber**将分别引用两个字符串对象。
- 如果数据类型是一个原始类型（其名称由小写字母组成），这种类型的变量将直接保存一个原始数据类型的值。
 - **double** 是一个**原始数据类型**
 - **number1, number2**变量将保存两个双精度数

Java两种类型的变量



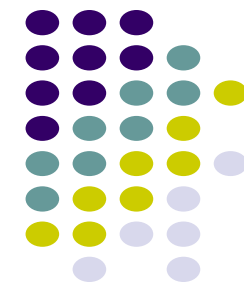
■ 类型转换——字符串转为浮点数

```
number1 = Double.parseDouble( firstNumber );  
number2 = Double.parseDouble( secondNumber );
```

■ **Double.parseDouble**是一个Double类所定义的静态方法

- 将 String 数据转为double类型的
 - 返回 double类型的数值
 - 记住静态方法调用语法：类名.静态方法名(参数)
 - Double是原始数据类型double的“包装类”，属于引用类型
- 实例——**RandomStr.java**：使用类型转换生成6位验证字符串，示例程序每次运行时，都会生成不同的字符串。

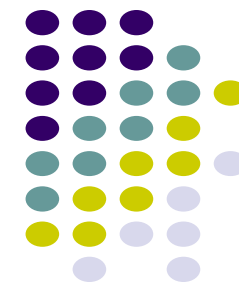
第二章 Java结构化程序设计



- 标识符、关键字
- 数据类型、常量和变量
- **运算符**
- 控制结构
- 编程规范



运算符(Operator)



- 算术运算符 (arithmetic operator)
 - 基本与C/C++一致

运算符	含义	例子	值
+	加	20 + 30	50
-	减	5.0 - 1.0	4.0
*	乘	30 * 30	900
/	除/整除	1.0/2.0	0.5
%	取模	20 % 7 5.0 % 2.0 10 % 5	6 1.0 0

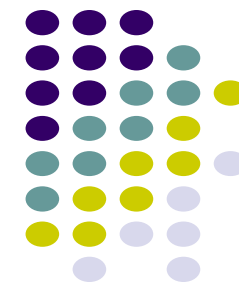
运算符



■ 整数和浮点数混合运算

例子	值	含义
<code>int x = 5/2;</code>	2	整数除法
<code>double y = 5/2;</code>	2.0	先做整数除法，然后再将结果转换为double
<code>double z = 5.0/2;</code> <code>double u = 5/2.0;</code>	2.5 2.5	由于分子、分母至少有一个是浮点数，因此做浮点除法
<code>int w = (int) (5.0/2)</code>	2	先做浮点除法，然后取整。 取整的算法就是，直接截去小数部分 (注意：不是四舍五入)

运算符

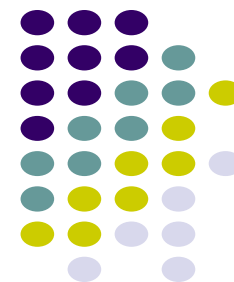


■ 复合赋值运算符

运算符	例子	值
<code>+=</code>	<code>i += 8</code>	<code>i = i + 8</code>
<code>-=</code>	<code>j -= 15.0</code>	<code>j = j - 15.0</code>
<code>*=</code>	<code>k *= 10</code>	<code>k = k * 10</code>
<code>/=</code>	<code>m /= 3</code>	<code>m = m / 3</code>
<code>%=</code>	<code>n %= 7</code>	<code>n = n % 7</code>

注意：`+-*/%` 同 `=` 之间没有空格

运算符



■ 自增自减运算符

运算符	名称	说明
<code>++var</code>	前置增量运算符	表达式 (<code>++var</code>) 使变量 <code>var</code> 的值加1, 并且该表达式的值取 <code>var</code> 增加以后的新值
<code>var++</code>	后置增量运算符	表达式 (<code>var++</code>) 的值取变量 <code>var</code> 原来的值, 并使 <code>var</code> 的值加1
<code>--var</code>	前置减量运算符	表达式 (<code>--var</code>) 使变量 <code>var</code> 的值减1, 并且该表达式的值取 <code>var</code> 减少以后的新值
<code>var--</code>	后置减量运算符	表达式 (<code>var--</code>) 的值取变量 <code>var</code> 原来的值, 并使 <code>var</code> 的值减1

```
int number = 10;  
int newNumber = 10 * number++;
```

相当于

```
int number = 10;  
int newNumber = 10 * number;  
number = number + 1;
```

结果: `number = 11, newNumber = 100`

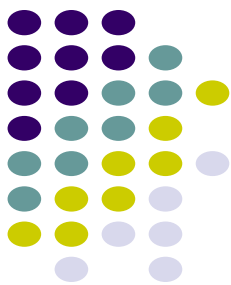
```
int number = 10;  
int newNumber = 10 * (++number);
```

相当于

```
int number = 10;  
number = number + 1;  
int newNumber = 10 * number;
```

结果: `number = 11, newNumber = 110`

举例



■ 从秒数计算小时和分钟 Demo. ConvertSeconds.java

```
public class ConvertSeconds
{
    /** Main method */
    public static void main(String[] args)
    {
        int totalSeconds = 10000;

        int totalMinutes = totalSeconds / 60;
        int hours = totalMinutes / 60;
        int minutes = totalMinutes % 60;
        int seconds = totalSeconds % 60;

        // 输出方法一:
        System.out.println(totalSeconds + " = "
            + hours + ":" + minutes + ":" + seconds);

        // 输出方法二:
        System.out.printf("%d = %d:%d:%d\n",
            totalSeconds, hours, minutes, seconds);
    }
}
```

运算符



■ 关系运算符 (relational operator)

- 用于两个值的比较

关系运算符	含义
<	小于
<=	小于等于
>	大于
>=	大于等于
==	等于
!=	不等于

注：

- 相等的比较运算符是两个等号 (==)
- 字符也可以进行比较。实际上是对字符的 Unicode 进行比较。如：('a' > 'A') = true

运算符

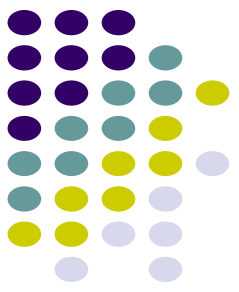


■ 逻辑运算符（logical operator）

- 对boolean值运算，得到新的boolean值

比较运算符	名称	含义
!	非（NOT）	取反。true → false, false → true
&&	与（AND）	两个运算符都为true时，结果才为true
	或（OR）	当两个运算符至少有一个为true时，结果才为true
^	异或（XOR）	当两个运算符不一样时，结果才为true

BOOLEAN HAIR LOGIC



A



B



AND



OR



XOR



■ 与、或的短路运算法则

- **与（&&）运算**：一旦有一个操作数为false，则直接得到结果false，后面的表达式不再计算。
 - 如：(4<3) && (1/0 ==1) 不会产生“被零除”错误。因为(4<3)为false，最终的结果就是false，程序不会再计算(1/0 ==1)了
- **或（||）运算**：一旦有一个操作数为true，则直接得到结果true，后面的表达式不再计算。
 - 如：(4>3) || (1/0 ==1) 不会产生“被零除”错误。因为(4>3)为true，最终的结果就是true，程序不会再计算(1/0 ==1)了

逻辑运算符



■ 例：判断是不是闰年

- 闰年的定义：该年可以被4整除而不能被100整除，或者可以被400整除，那就是闰年

```
import javax.swing.JOptionPane;
public class LeapYearVerifier
{
    /** Main method */
    public static void main(String[] args)
    {
        // 用户输入年份
        String yearString = JOptionPane.showInputDialog("Enter a year");
        int year = Integer.parseInt(yearString);
        // 是否为闰年
        boolean isLeapYear = ( (year % 4 == 0)
                                && (year % 100 != 0) ) || (year % 400 == 0);

        // 显示
        String display = "Year " + year + " isLeapYear = " + isLeapYear;
        JOptionPane.showMessageDialog(null, display);
    }
}
```

例：简单的数学学习工具



- 程序随机产生两个一位整数number1和number2，显示给学生如 “What is 7 + 9 ?”，学生在输入对话框中敲入答案之后，程序显示一个消息对话框，判定答案是true还是false。 **Demo: AdditionTest.java**

```
import javax.swing.JOptionPane;
public class AdditionTest
{
    /** Main method */
    public static void main(String[] args)
    {
        // 产生两个随机数 (int类型)
        int n1 = (int) (System.currentTimeMillis() % 100);
        int n2 = (int) (System.currentTimeMillis() * 7 % 10);
        // 提示用户输入结果
        String answerString = JOptionPane.showInputDialog(
            "What is " + n1 + " + " + n2 + " ?");
        int answer = Integer.parseInt(answerString);
        // 计算结果
        String result = n1 + " + " + n2 + " = " + answer + " is "
            + (n1 + n2 == answer);
        JOptionPane.showMessageDialog(null, result);
    }
}
```

运算符



- 位运算符 (bit operator)
- 当操作整型数据时，可以使用位运算符（即：按位运算）
 - 二进制级别的运算，当用于boolean运算时，& 和 | 会产生boolean值，结果同 && 和 || 相同，但是不会进行短路运算。

位运算符	名称	含义
~	非（NOT）	整数按位取反， $1 \rightarrow 0$ ， $0 \rightarrow 1$
&	与（AND）	两个整数，按位进行“与”操作
	或（OR）	两个整数，按位进行“或”操作
^	异或（XOR）	两个整数，按位进行“异或”操作

运算符



- 位运算符
- &运算符的其他用途
 - 测试一个数的某位是否为 1
 - $x \& 4 = 0$, 说明 x 的第 2 位为 0
 - $x \& 8 \neq 0$, 说明 x 的第 3 位为 1
 - 截取一个数的低4位
 - $0x7B \& 0x0F \Rightarrow 0x0B$
 - 截取一个数的高4位
 - $0x7B \& 0xF0 \Rightarrow 0x70$

运算符

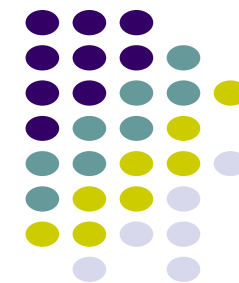


■ 移位运算符

位运算符	名称	举例
<<	左移位	<code>0x01 << 2 ==> 4</code>
>>	带符号的右移位	<code>0x3F >> 2 ==> 15</code> <code>-1 >> 2 ==> -1 (-1 = 0xFFFFFFFF)</code> <code>-8 >> 2 ==> -2 (-8 = 0xFFFFFFFF8)</code> <code>(-2 = 0xFFFFFFF8)</code>
>>>	无符号的右移位	<code>-1 >>> 2 ==> 1073741823 (0x3FFFFFFF)</code>

- 移位操作是：先将整数写成二进制形式，然后按位操作，最后产生一个新的数
- 注意：只用于整数

运算符



■ 条件运算符

- 条件运算符 “?:”的表达式形式为 “op1 ? op2 : op3”

- op1:布尔表达式

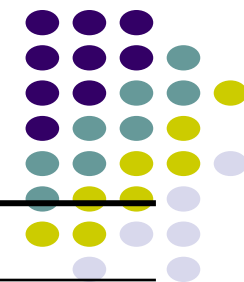
- 例如:

```
int k= ( i>=0) ? 1 : -1;
```

■ 其他运算符

- 其他运算符包括: (类型)、.、[]、()、instanceof和new

运算符



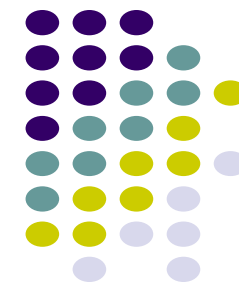
■ 运算符优先级

高

低

运算符	含义
var++, var--	后置自增/自减运算符
+var, -var, ++var, --var ~ !	正/负数标识符 前置自增/自减运算符 一元运算符
*, /, %	乘、除、求余运算符
+, -	加减运算符
<< >> >>>	移位运算符
<, <=, >, >= instanceof	关系运算符
==, !=	相等/不等比较运算符
&	按位与运算符
^	按位异或运算符
	按位或运算符
&&	逻辑与运算符
	逻辑或运算符
=, +=, -=, *=, /=, %= &=, ^=, =, <<=, >>=, >>>=	赋值运算符

运算符



■ 运算符结合性 (Precedence of operators)

- 计算没有括号的表达式时，运算符依照优先级规则和结合方向进行运算。
- 除了赋值运算符 `=`，所有的双目运算符都是左结合的 (left-associative)，**从左到右**。
- 例如：`x = y = z` 相当于 `x = (y = z)`

■ 计算表达式的规则

- 规则1：可能的情况下，从左向右依次计算所有的子表达式。
- 规则2：根据运算符的优先级进行运算。
- 规则3：对优先级相同的相邻运算符，根据结合方向进行运算。

运算符



- 应用这些规则，表达式 $3+4*4>5*(4+3)-1$ 的计算如下：

$3 + \underline{4 * 4} > 5 * (4 + 3) - 1$

$\underline{3 + 16} > 5 * (4 + 3) - 1$

$19 > 5 * \underline{(4 + 3)} - 1$

$19 > \underline{5 * 7} - 1$

$19 > \underline{35} - 1$

$\underline{19} > 34$

false

(1) 从左边开始， $4*4$ 是第一个可计算的表达式

(2) 接下来计算 $3+16$

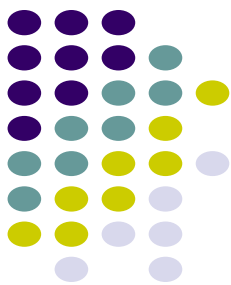
(3) 接下来计算 $4+3$

(4) 接下来计算 $5*7$

(5) 接下来计算 $35-1$

(6) 比较运算 $19>34$

(7) 得到最终结果



- 太复杂了！我根本记不住！！！！



- 除了以下一些专门考人的场合：

- Java课程考试
- 找工作时，面试/笔试
- 你像我一样，给别人教 Java
- 你要设计一门新的编程语言

- 否则你永远都不要写 $3+4*4>5*(4+3)-1$ 这样的表达式！

- 怎么办？多用几个括号就搞定：

$(3 + 4 * 4) > (5 * (4 + 3) - 1)$







第一次作业

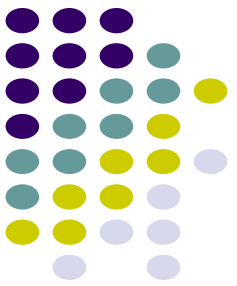


- 编写一个Java应用程序，输入代表华氏温度的整数值，计算出相应的摄氏温度的小数。利用如下公式进行换算：
 - $C = 5(F - 32) / 9$ 。
- 编写一个程序，用户输入两个数，求出其加减乘除，并用消息框显示计算结果。



■ 作业提示

Message dialog type	Icon	Description
<code>JOptionPane.ERROR_MESSAGE</code>		Displays a dialog that indicates an error to the user.
<code>JOptionPane.INFORMATION_MESSAGE</code>		Displays a dialog with an informational message to the user. The user can simply dismiss the dialog.
<code>JOptionPane.WARNING_MESSAGE</code>		Displays a dialog that warns the user of a potential problem.
<code>JOptionPane.QUESTION_MESSAGE</code>		Displays a dialog that poses a question to the user. This dialog normally requires a response, such as clicking on a Yes or a No button.
<code>JOptionPane.PLAIN_MESSAGE</code>	no icon	Displays a dialog that simply contains a message, with no icon.



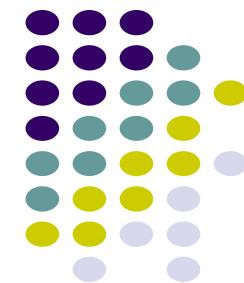
■ 字符串转为浮点数/整数

```
String firstNumber;           // first string entered by user
String secondNumber;          // second string entered by user
double number1;               // first number to add
double number2;               // second number to add
```

```
number1 = Double.parseDouble( firstNumber );

number2 = Double.parseDouble( secondNumber );
```

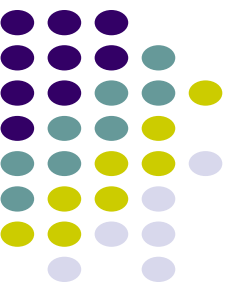
第二章 Java结构化程序设计



- 标识符、关键字
- 数据类型、常量和变量
- 运算符
- **控制结构**
- 编程规范

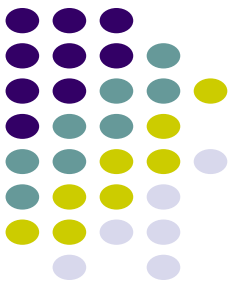


控制结构



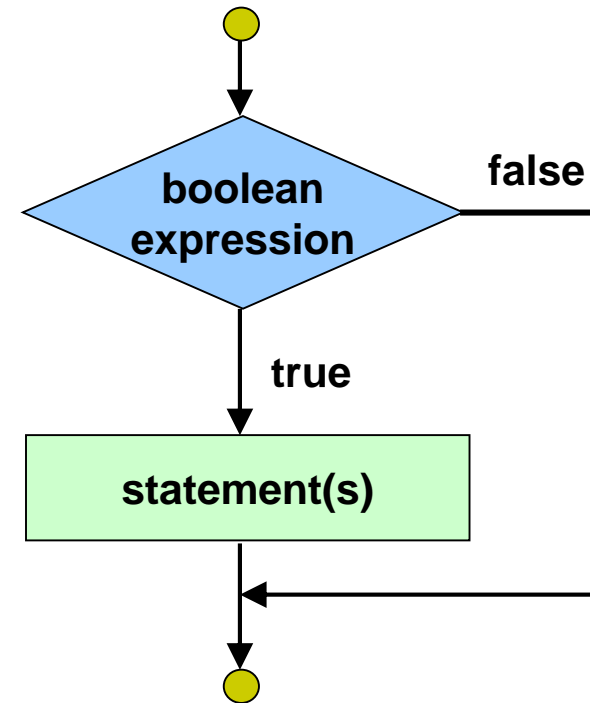
- 程序执行的顺序
 - 三种程序基本结构
 - Sequence structure（顺序结构）
 - Selection structure（选择结构）
 - Repetition structure（循环结构）
 - 上述三种程序基本结构是“**完备的**”，再复杂的程序执行流程，都可以分解为上述三种结构的组合。
 - “**流程图（Flowchart）**”常用于描述程序执行顺序。

选择结构



■ 简单的if语句

```
if (booleanExpression)
{
    statement(s) ;
}
```



```
if ( (i>0) && (i<10) )
{
    System.out.println(
        "i is between 0 to 10");
}
```

boolean expression

true statement

选择结构



■ 注意：

1、外括号必不可少

```
if ( (i>0) && (i<10) )  
{  
    System.out.println(  
        "i is between 0 to 10");  
}
```

2、当块里只包含一条语句，可以不用花括号。

等效于
→

```
if ( (i>0) && (i<10) )  
    System.out.println(  
        "i is between 0 to 10");
```

3、if 子句末尾不能加分号！

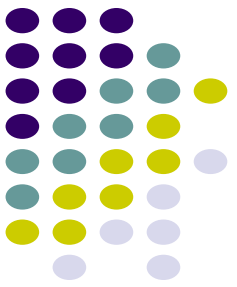
```
if ( (i>0) && (i<10) );  
{  
    System.out.println(  
        "i is between 0 to 10");  
}
```

这个错误很难被发现，因为它不是一个编译错误或运行错误，而是一个逻辑错误。

使用次行块风格时经常发生这样的错误，使用行尾块风格可以避免。

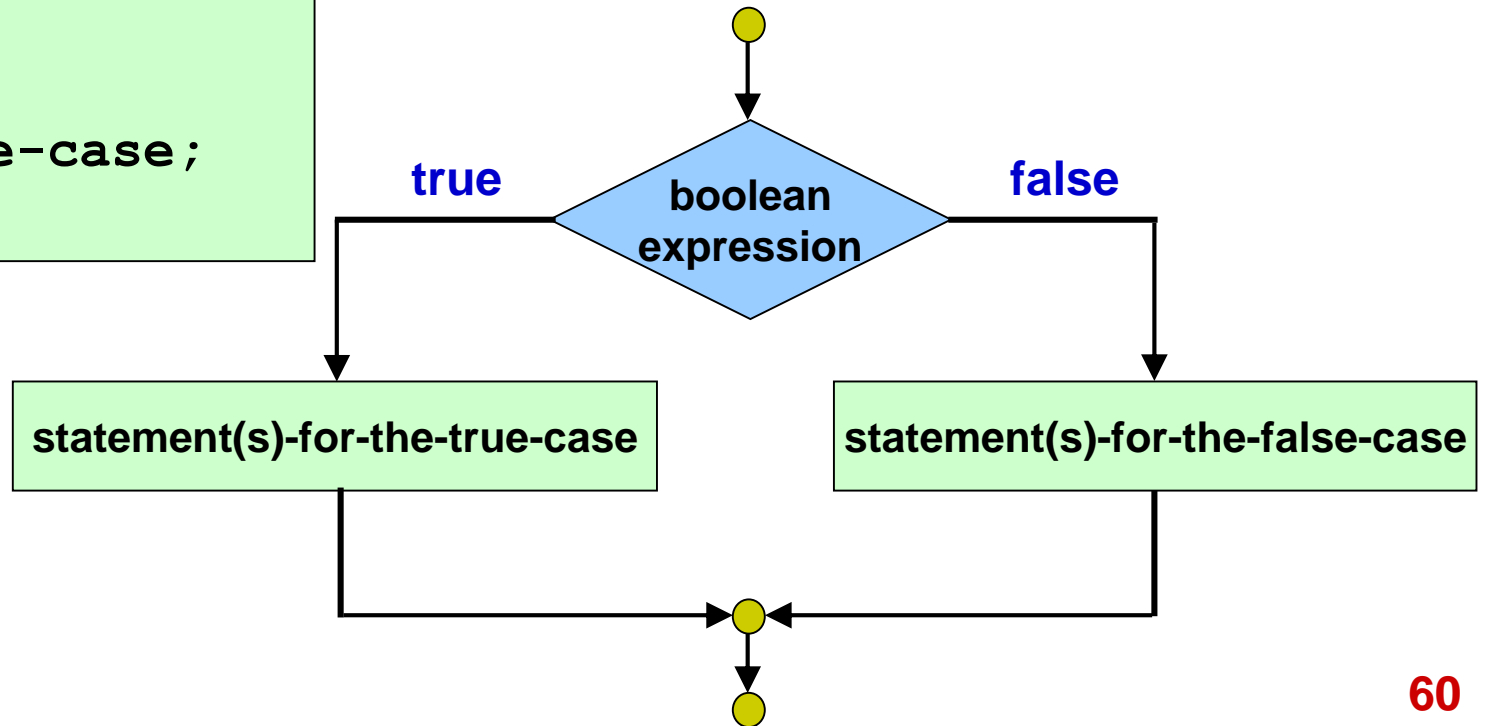
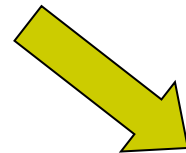
该语句会被无条件执行！

选择结构



■ if ... else 语句

```
if (booleanExpression)
{
    statement(s) -for-the-true-case;
}
else
{
    statement(s) -for-the-false-case;
}
```



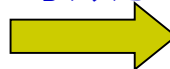
选择结构



■ if 语句的嵌套

```
if (score >= 90.0)
    grade = 'A';
else
    if (score >= 80.0)
        grade = 'B';
    else
        if (score >= 70.0)
            grade = 'C';
        else
            if (score >= 60.0)
                grade = 'D';
            else
                grade = 'F';
```

等效于



```
if (score >= 90.0)
    grade = 'A';
else if (score >= 80.0)
    grade = 'B';
else if (score >= 70.0)
    grade = 'C';
else if (score >= 60.0)
    grade = 'D';
else
    grade = 'F';
```

建议采用这种书写风格：

- 避免了深层缩进；
- 程序可读性好

选择结构



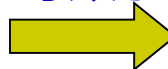
■ else 和 if 的匹配

- else子句与同一块中离得最近的if子句相匹配

```
int i = 1;
int j = 2;
int k = 3;

if (i > j)
    if (i > k)
        System.out.println('A');
else
    System.out.println('B');
```

等效于



```
int i = 1;
int j = 2;
int k = 3;

if (i > j)
    if (i > k)
        System.out.println('A');
else
    System.out.println('B');
```

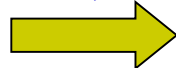
选择结构



■ 提示1:

```
if (number % 2 == 0)
    even = true;
else
    even = false;
```

等效于



```
boolean even = (number % 2 == 0);
```

■ 提示2:

```
if (even == true)
{
    System.out.println("...");
}
```

等效于



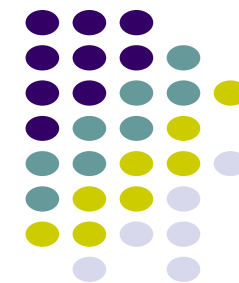
```
if (even)
{
    System.out.println("...");
}
```

建议使用这种形式，因为如果不小心写为：

```
if (even = true)
{
    System.out.println("...");
}
```

将很难被发现。

选择结构



■ Switch 语句

表达式 `switch-expression` 必须能计算出一个 `char`、`byte`、`short` 或 `int` 型值，并且必须用括号括住它

```
switch (switch-expression)
{
    case value1: statement(s)1;
                break;
    case value2: statement(s)2;
                break;
    ... ..
    case valueN: statement(s)N;
                break;
    default:    default-statement(s);
}
```

关键字 `break` 是可选的。

`break` 语句可以立即终止整个 `switch` 语句。如果 `break` 语句没有出现，那么执行下一条 `case` 语句

默认情况 (`default`) 是可选的，它用来执行指定情况与 `switch-expression` 都不匹配时的操作

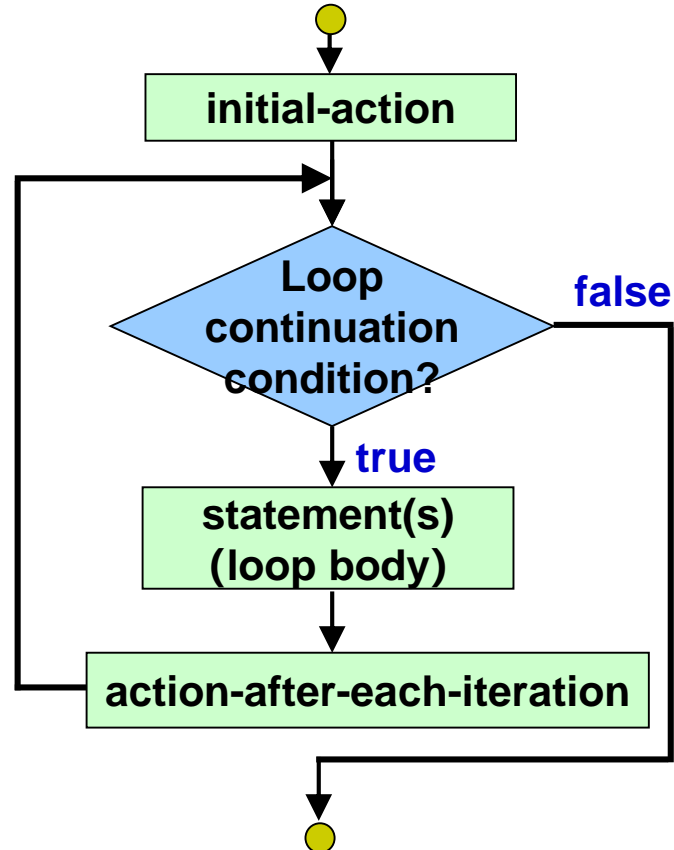
`case` 语句是顺序检测的，这些 `case` 的顺序（包括默认情况）是无所谓的。

但是，将所有情况按照逻辑顺序排列并把默认情况放在最后是良好的编程风格。

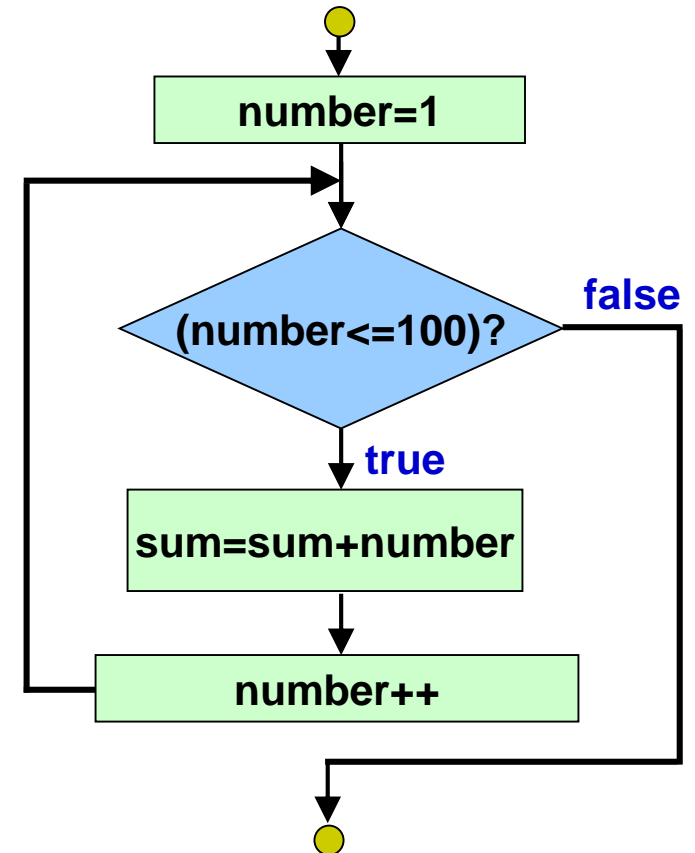
循环结构

■ for 循环

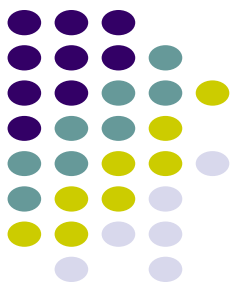
```
for (initial-action;  
    loop-continuation-condition;  
    action-after-each-iteration)  
{  
    // loop body  
    statement(s);  
}
```



```
// 1 加到 100  
int sum = 0;  
int number;  
for (number=1; number<=100; number++)  
{  
    sum = sum + number;  
}
```



循环结构



■ 注意:

- for 循环的初始条件可以是 0 个或多个逗号分隔开的表达式。
- for 循环中迭代之后的动作也可以是 0 个或多个逗号分隔开的语句。
- 因此，下面的两个for 循环是正确的，尽管实际中很少用到它们。

```
for (int i = 1; i < 100; System.out.println(i++));
```

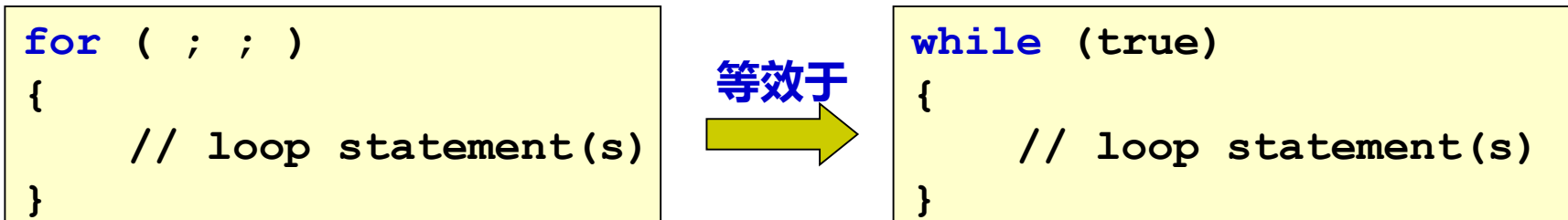
```
for (int i = 0, j = 0; (i + j < 10); i++, j++)  
{  
    // Do something  
}
```

循环结构



■ 注意：

- 如果for 循环中的继续循环条件（loop-continuation-condition）被忽略，那么它是隐式为true 的



这是一个无限循环，建议使用这种形式。

举例：嵌套循环



■ 编写九九乘法表

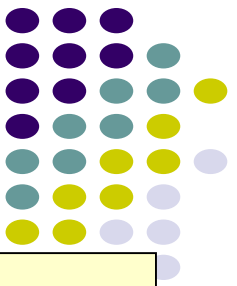
```
import javax.swing.JOptionPane;

public class MultiplicationTable
{
    public static void main(String[] args)
    {
        for (int i=1; i<=9; i++)
        {
            for (int j=1; j<=i; j++)
            {
                System.out.printf("%d*%d=%2d ", j, i, i*j);
            }
            System.out.println();
        }
    }
}
```

输出结果：

```
1*1= 1
1*2= 2 2*2= 4
1*3= 3 2*3= 6 3*3= 9
1*4= 4 2*4= 8 3*4=12 4*4=16
1*5= 5 2*5=10 3*5=15 4*5=20 5*5=25
1*6= 6 2*6=12 3*6=18 4*6=24 5*6=30 6*6=36
1*7= 7 2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49
1*8= 8 2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64
1*9= 9 2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81
```

循环结构



```
while (loop-continuation-  
condition)  
{  
    // loop body  
    statement(s);  
}
```

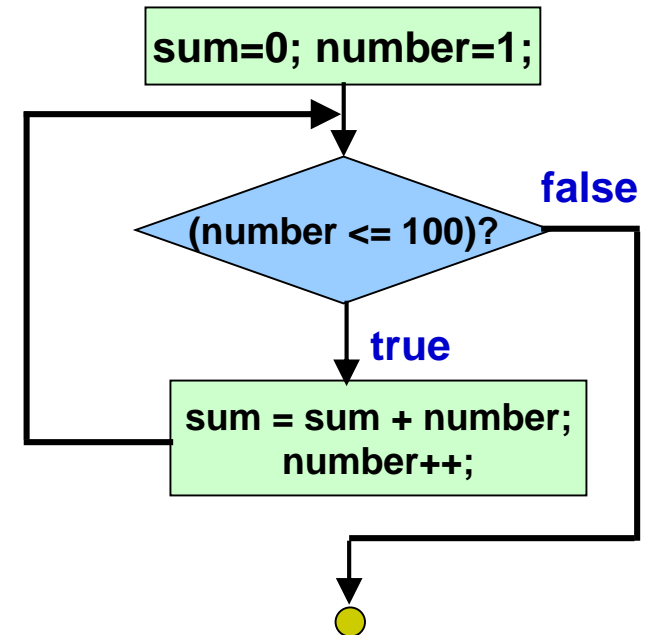
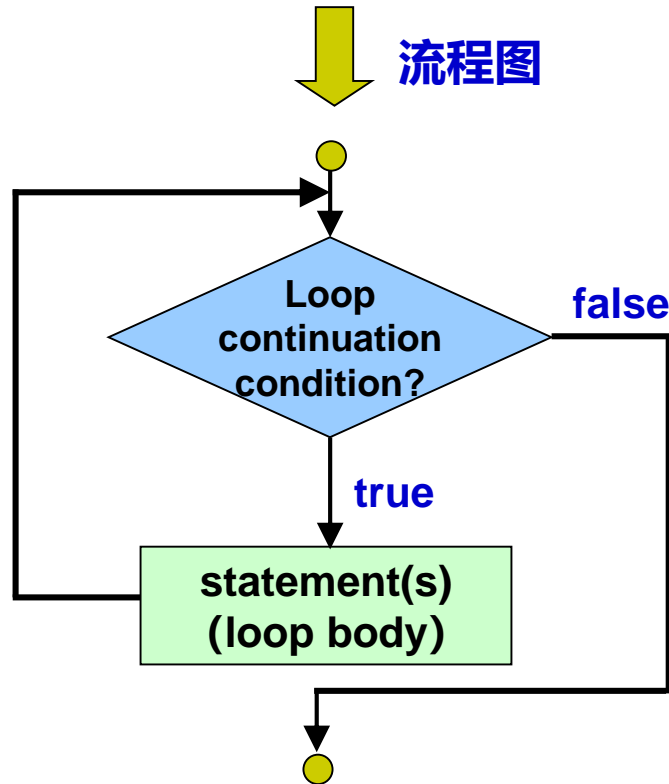
例子



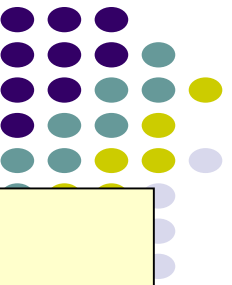
```
// 1 加到 100  
int sum = 0;  
int number = 1;  
while (number <= 100)  
{  
    sum = sum + number;  
    number++;  
}
```

■ while循环

流程图



循环结构



```
import javax.swing.JOptionPane;

public class SentinelValue
{
    public static void main(String[] args)
    {
        String dataString = JOptionPane.showInputDialog("Enter value(0 to exit)");
        int data = Integer.parseInt(dataString);
        int sum = 0;
        while (data != 0)
        {
            sum = sum + data;
            dataString = JOptionPane.showInputDialog(
                "Enter value ( 0 to exit)");
            data = Integer.parseInt(dataString);
        }
        // Display result
        JOptionPane.showMessageDialog(null, "Sum = " + sum);
    }
}
```

循环结构



■ 注意：

- 在循环控制中**不要比较浮点数相等**。因为浮点数是近似的，使用它们可能导致不精确的循环次数和不准确的结果。

```
double data = Math.pow(Math.sqrt(2), 2) - 2;

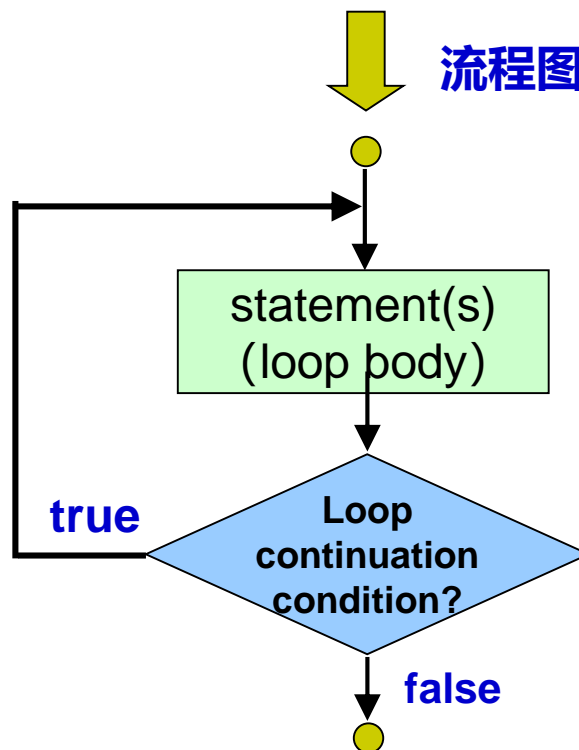
if (data == 0)
{
    System.out.println("data is zero");
}
else
{
    System.out.println("data is not zero");
}
```

循环结构



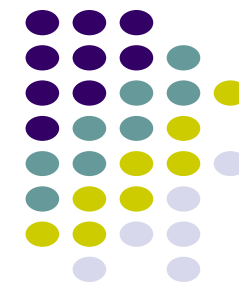
■ do-while 循环

```
do
{
    // loop body
    statement(s);
}
while (loop-continuation-condition);
```



同 while 循环相比，在do-while循环中，循环体至少会被执行一次。

采用哪种循环？



- 三种形式的循环语句 `while`, `do-while` 和 `for` 在表达上都是等价的。这也就是说，可以用三种循环的任一种。

```
while (loop-continuation-condition)
{
    // loop statement(s)
}
```

等效于

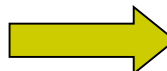


```
for ( ; loop-continuation-condition; )
{
    // loop statement(s)
}
```

while 循环转换为 for 循环

```
for (initial-action;
     loop-continuation-condition;
     action-after-each-iteration)
{
    // loop statement(s)
}
```

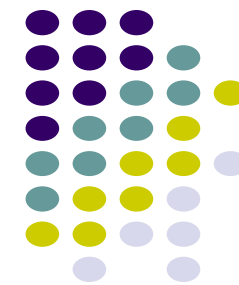
等效于



```
initial-action;
while (loop-continuation-condition)
{
    // loop statement(s)
    action-after-each-iteration;
}
```

for 循环转换为 while 循环

采用哪种循环？



- 建议使用自己觉得最自然、最舒服的一种。
 - 通常，如果重复次数已经知道，就采用for循环。
 - 例如将一条信息打印100次。
 - 如果不知道重复次数，就采用while循环。
 - 例如读入一些数值直到读入0为止。
 - 如果在检验循环条件前需要执行循环体，就用do-while循环替代while循环。

采用哪种循环？



- 在for/while循环子句之后循环体之前多写分号是常见的错误，如下所示：

```
for (int i=0; i<100; i++);
{
    System.out.println(i);
}
```

```
int i=0;
while (i<100);
{
    System.out.println(i);
    i++;
}
```

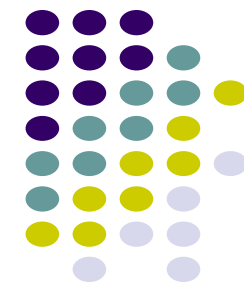
错误!!!

- 但是在 do-while 循环中，while 子句必须要以分号结尾：

```
int i=0;
do
{
    System.out.println(i);
    i++;
}
while (i<100);
```

正确!!!

第二章 Java结构化程序设计



- 标识符、关键字
- 数据类型、常量和变量
- 运算符
- 控制结构
- 编程规范



Java语言规范



■ 可以从以下网址获取最新的Java语言规范:

- <http://docs.oracle.com/javase/specs/>

ORACLE

[Oracle Technology Network](#) > [Java SE](#) > [Java SE Documentation](#) > Java SE Specifications

Java Language and Virtual Machine Specifications

Java SE 7



The Java Language Specification, Java SE 7 Edition

- [View HTML](#)
- [Download PDF](#)
- [Purchase book at Amazon](#)



The Java Virtual Machine Specification, Java SE 7 Edition

- [View HTML](#)
- [Download PDF](#)
- [Purchase book at Amazon](#)

Java SE 5.0 / SE 6



The Java Language Specification, Third Edition

- [View HTML](#)
- [Download PDF](#)



The Java Virtual Machine Specification, Second Edition (Updated)

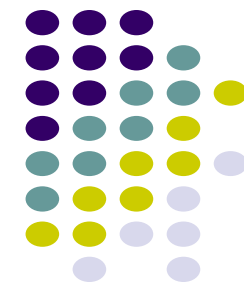
- [View HTML \(Second Edition\)](#)
- [Download HTML \(Second Edition\)](#)
- View updates to the Second Edition: [For SE 5.0](#) [For SE 6](#)

The Java® Language Specification *Java SE 7 Edition*

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley

2013-02-28

附录A：编程风格和文档



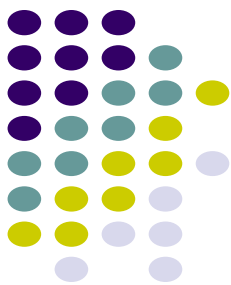
- 适当的注释
- 命名习惯
- 适当的缩进和空白
- 块的对齐方式
- 详见：《**Java** 编码规范》

编程风格和文档



■ 适当的注释

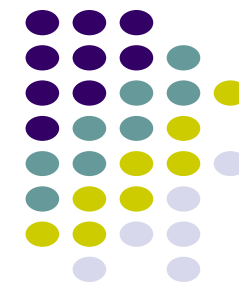
- 应该在程序开头写一个摘要，说明程序的目的和主要特点是什么，解释所用的重要数据结构和独特技术，还要包括作者的名字、类块、数据等。
- 在重要的方法前面，也要有相应的注释，说明该方法的作用、采用的算法等。
- 在程序的关键点上，也需要做一定的注释。
- 注释很重要，但是程序的可读性同样重要！读一个好的程序，就像读一篇美妙的文章。



```
import javax.swing.JOptionPane;
/**
 * <br> 计算并显示当前的时分秒
 * <br> 通过 System.currentTimeMillis() 得到从 Unix Epoch Time 经过的毫秒数,
 * <br> 然后计算出时分秒
 *
 * @author julie
 * @version 1.0
 */
public class CurrentTime
{
    /**
     * Main 方法
     * @param args 命令行参数, 字符串数组类型
     * @return 无
     */
    public static void main(String[] args)
    {
        // ... ..
    }
}
```

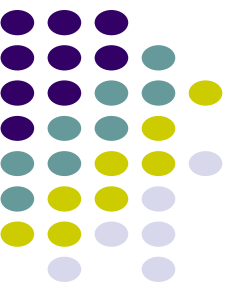
在 Eclipse 中, 完成上面的注释。

1. 将鼠标悬停到“CurrentTime”和“main”上面, 看看会发生什么事情。
2. 在 /** 中, 删掉一个 *, 然后再悬停鼠标, 看看会发生什么事情。



■ 命名习惯

- 选择有意义的描述性名字。一个恰到好处的名字，胜过长篇大论的注释
- 类名
 - 类名的每个单词的首字母大写，如 `TaxCalculator`
 - 类名一般是名词形式
- 常量
 - 所有的字母都大写，两个单词间要用下划线连接
 - 如： `PI`、`MAX_INT_VALUE` 等



- 变量和方法名
 - 常用小写，如： `area`, `radius`
 - 如果名字包括几个词，则把它们连成一个，第一个词的字母小写而后面的每个单词的首字母大写，例如：
 - 变量： `currentTime`、`annualInterestRate`
 - 方法： `showInputDialog()`、`currentTimeMillis()`
 - 这种书写方式又称为**骆驼 (camel)** 式
 - 变量名一般是**名词**形式，如 `name`、`age`、`amount` 等
 - 方法名一般是**动词**形式，如 `getName()`，`calculateAmount()` 等

编程风格和文档



■ 缩进

- 建议缩进 4 个空格

■ 空白

- 使用空行分隔代码段

■ 块的对齐方式

下行风格

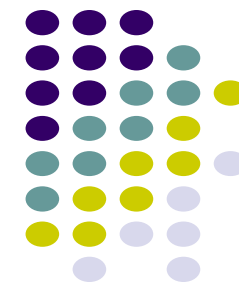
```
public class Demo
{
    public static void main(String[] args)
    {
        System.out.println("This is a demo!");
    }
}
```

行尾风格

```
public class Demo {
    public static void main(String[] args) {
        System.out.println("This is a demo!");
    }
}
```

这两种风格的拥护者争论已久，各有所长。推荐使用“下行风格”

附录B：编程错误



■ 语法错误

- 编译器检测

■ 运行错误

- 导致程序终止

■ 逻辑错误

- 产生错误的结果

编程错误

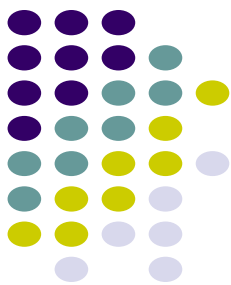


■ 语法错误

- 语法错误是由代码结构中的问题引起的，例如：拼错关键字，丢掉必要的标点，或者左花括弧没有对应的右花括弧等。这些错误通常容易查出，因为编译器会指出它们在哪儿，原因是什么。
- 下面的程序有三个错误：

```
public class SyntaxErrorDemo
{
    /** Main method */
    public static void main(string[] args)
    {
        float n1 = 13.0;
        System.out.println("n1=" + n1) _____
    }
}
```

编程错误



■ 运行错误

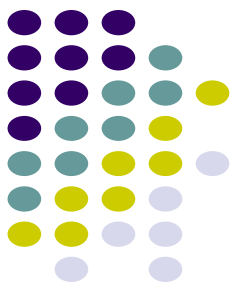
- 运行应用程序时，当环境检测到一个不可能执行的操作时就会出现运行错误。
- Java的运行时错误都是 **RuntimeException**
- 典型的运行时错误，如：被 0 除、数组越界、类型转换错误等。

```
public class RuntimeExceptionDemo
{
    /** Main method */
    public static void main(String[] args)
    {
        int average = 20/0;
    }
}
```



■ 逻辑错误

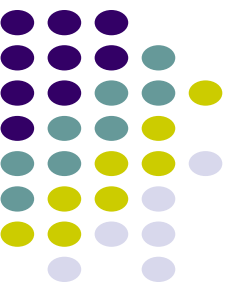
- 逻辑错误会产生错误的结果，但是编译器无法检测到。
- 一般通过一些调试手段来定位错误，并修复错误
- 逻辑错误也称为bug，查找和改正错误的过程称为调试（debugging）
- 调试的一般途径是采用各种方法逐步缩小程序中错误所在的范围。
 - 可以手工跟踪（hand trace）程序（即通过读程序找错误）
 - 或者插入输出语句，显示变量的值或程序的执行流程。这种方法适用于短小、简单的程序。
 - 对于庞大复杂的程序，最有效的调试方法是使用调试工具。



■ 调试器

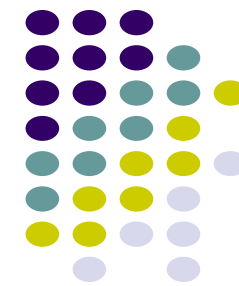
- 调试器是用来方便调试的程序，可以使用调试器
 - 一次执行一条语句
 - 进入或跳过方法
 - 设置断点
 - 显示变量
 - 显示调用栈
 - 修改变量
- Eclipse、NetBeans 都内置了功能强大的调试器

第二次作业



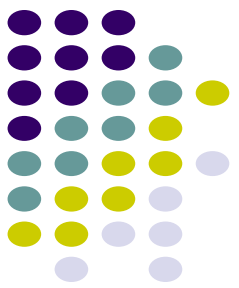
- 编写程序，计算个人所得税。
- 计算方法：
 - 起征点：3500元
 - 收入-起征点，差值部分：
 - 1、不超过1500元的部分，税率3%
 - 2、超过1500元至4500元的部分，税率10%
 - 3、超过4500元至9000元的部分，税率20%
 - 4、超过9000元至35000元的部分，税率25%
 - 5、超过35000元至55000元的部分，税率30%
 - 6、超过55000元至80000元的部分，税率35%
 - 7、超过80000元的部分，税率45%
- 要求：取小数点后2位，并输出
- 如某人的收入为10000元，需要缴纳的个人所得税为：
 - $10000 - 3500 = 6500$
 - $1500 * 3\% + (4500 - 1500) * 10\% + (6500 - 4500) * 20\%$
 $= 45 + 300 + 400$
 $= 745 \text{ 元}$

第二次作业



- 编写程序，提示用户输入两个正整数，并求出它们的最大公约数。
- 提示
 - 设输入的两个整数为 $n1$ 和 $n2$ 。已知1是一个公约数，但可能不是最大公约数。可以检测 k ($k=2, 3, 4, \dots$) 是否为 $n1$ 和 $n2$ 的最大公约数，直到 k 大于 $n1$ 或 $n2$ 。

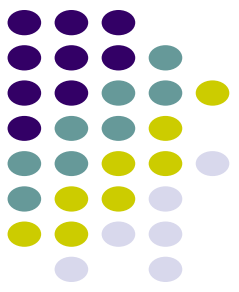
第二次作业



编写程序，打印下面的图形。

*

**



■ 要求：

- 提交作业至：HHU_java@163.com
- 提交的作业文件名为：“学号+姓名+Java第一次作业”
- 谢谢同学们的配合！