数据类型与运算符

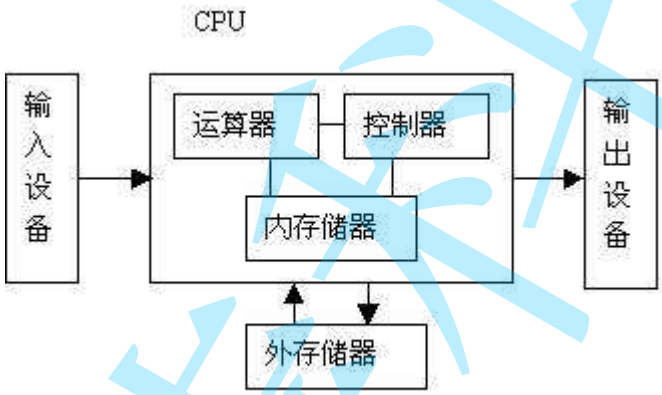
本节目标

1. 变量和类型

变量指的是程序运行时可变的量. 相当于开辟一块内存空间来保存一些数据.

类型则是对变量的种类进行了划分, 不同的类型的变量具有不同的特性.

我们所讨论的 "变量" 主要和我们的 "内存" 这样的硬件设备密切相关.



1.1 整型变量(重点)

基本语法格式

```
int 变量名 = 初始值;
```

代码示例:

```
int num = 10; // 定义一个整型变量
System.out.println(num) ;
```

注意事项:

1. int 表示变量的类型是一个整型
2. 变量名是变量的标识. 后续都是通过这个名字来使用变量
3. Java 中 `=` 表示赋值(和数学不一样), 意思是给变量设置一个初始值.
4. 初始化操作是可选的, 但是建议创建变量的时候都显式初始化.
5. 最后不要忘记分号, 否则会编译失败.
6. // 表示注释. 注释作为代码的解释说明部分, 不参与编译运行.

在 Java 中, 一个 int 变量占 4 个字节. 和操作系统没有直接关系.

> **什么是字节?**
>
> 字节是计算机中表示空间大小的基本单位.
>
> 计算机使用二进制表示数据. 我们认为 8 个二进制位(bit) 为一个字节(Byte).
>
> 我们平时的计算机为 8GB 内存, 意思是 8G 个字节.
>
> 其中 1KB = 1024 Byte, 1MB = 1024 KB, 1GB = 1024 MB.
>
> 所以 8GB 相当于 80 多亿个字节.

4 个字节表示的数据范围是 `-2^31 -> 2^31-1`, 也就大概是 -21亿 到 +21亿.

使用以下代码查看 Java 中的整型数据范围:

```
System.out.println(Integer.MAX_VALUE);  // int 的最大值
System.out.println(Integer.MIN_VALUE);  // int 的最小值
```

如果运算的结果超出了 int 的最大范围, 就会出现溢出的情况

```
int maxValue = Integer.MAX_VALUE;
System.out.println(maxValue+1);

int minValue = Integer.MIN_VALUE;
System.out.println(minValue-1);
```

21亿这样的数字对于当前的大数据时代来说, 是很容易超出的. 针对这种情况, 我们就需要使用更大范围的数据类型来表示了. Java 中提供了 long 类型.

1.2 长整型变量

基本语法格式:

```
long 变量名 = 初始值;
```

代码示例:

```
long num = 10L; // 定义一个长整型变量, 初始值写作 10l 也可以(小写的 L, 不是数字1).
System.out.println(num) ;
```

注意事项:

1. 基本语法格式和创建 int 变量基本一致, 只是把类型修改成 long
2. 初始化设定的值为 `10L`, 表示一个长整型的数字. `10l` 也可以.
3. 使用 `10` 初始化也可以, 10 的类型是 int, 10L 的类型是 long, 使用 10 L 或者 10 l 更好一些.

Java 中 long 类型占 8 个字节. 表示的数据范围 `-2^63 -> 2^63-1`

使用以下代码查看 Java 中的长整型数据范围:

```
System.out.println(Long.MAX_VALUE);
System.out.println(Long.MIN_VALUE)

// 运行结果
9223372036854775807
-9223372036854775808
```

这个数据范围远超过 int 的表示范围. 足够绝大部分的工程场景使用.

1.3 双精度浮点型变量(重点)

基本语法格式

```
double 变量名 = 初始值;
```

代码示例:

```
double num = 1.0;
System.out.println(num)
```

神奇的代码1:

```
int a = 1;
int b = 2;
System.out.println(a / b);

// 执行结果
0
```

在 Java 中, int 除以 int 的值仍然是 int(会直接舍弃小数部分).

如果想得到 0.5, 需要使用 double 类型计算.

```
double a = 1.0;
double b = 2.0;
System.out.println(a / b);

// 执行结果
0.5
```

神奇的代码2:

```
double num = 1.1;
System.out.println(num * num)

// 执行结果
1.2100000000000002
```

Java 中的 double 虽然也是 8 个字节, 但是浮点数的内存布局和整数差别很大, 不能单纯的用 2 ^ n 的形式表示数据范围.

Java 的 double 类型的内存布局遵守 IEEE 754 标准(和C语言一样), 尝试使用有限的内存空间表示可能无限的小数, 势必会存在一定的精度误差.

1.4 单精度浮点型变量

基本格式:

```
float 变量名 = 初始值;
```

代码示例:

```
float num = 1.0f;    // 写作 1.0F 也可以
System.out.println(num);
```

float 类型在 Java 中占四个字节, 同样遵守 IEEE 754 标准. 由于表示的数据精度范围较小, 一般在工程上用到浮点数都优先考虑 double, 不太推荐使用 float.

1.5 字符类型变量

基本格式:

```
char 变量名 = 初始值;
```

代码示例:

```
char ch = 'A';
```

注意事项:

1. Java 中使用 `单引号 + 单个字母` 的形式表示字符字面值.
2. 计算机中的字符本质上是一个整数. 在 C 语言中使用 ASCII 表示字符, 而 Java 中使用 Unicode 表示字符. 因此一个字符占用两个字节, 表示的字符种类更多, 包括中文.

使用一个字符表示一个汉字:

```
char ch = '呵';
System.out.println(ch);
```

执行 javac 的时候可能出现以下错误:

```
Test.java:3: 错误: 未结束的字符文字
        char ch = '鍛?';
                  ^
```

此时我们在执行 javac 时加上 -encoding UTF-8 选项即可

```
javac -encoding UTF-8 Test.java
```

关于字符编码方式的讨论, 参见

https://zhuanlan.zhihu.com/p/35172335

1.6 字节类型变量

基本语法格式:

```
byte 变量名 = 初始值;
```

代码示例:

```
byte value = 0;
System.out.println(value);
```

注意事项:

1. 字节类型表示的也是整数. 只占一个字节, 表示范围较小 (`-128 -> +127`)
2. 字节类型和字符类型互不相干.

1.7 短整型变量

基本语法格式:

```
short 变量名 = 初始值;
```

代码示例:

```
short value = 0;
System.out.println(value);
```

注意事项:

1. short 占用 2 个字节, 表示的数据范围是 `-32768 -> +32767`
2. 这个表示范围比较小, 一般不推荐使用.

1.8 布尔类型变量

基本语法格式:

```
boolean 变量名 = 初始值;
```

代码示例:

```
boolean value = true;
System.out.println(value);
```

注意事项:

1. boolean 类型的变量只有两种取值, true 表示真, false 表示假.
2. Java 的 boolean 类型和 int 不能相互转换, **不存在** 1 表示 true, 0 表示 false 这样的用法.
3. boolean 类型有些 JVM 的实现是占 1 个字节, 有些是占 1 个比特位, 这个没有明确规定.

```
boolean value = true;
System.out.println(value + 1);

// 代码编译会出现如下错误
Test.java:4: 错误: 二元运算符 '+' 的操作数类型错误
        System.out.println(value + 1);
                           ^
  第一个类型:  boolean
  第二个类型: int
1 个错误
```

1.9 字符串类型变量(重点)

把一些字符放到一起就构成了字符串

基本语法格式:

```
String 变量名 = "初始值";
```

代码示例:

```
String name = "zhangsan";
System.out.println(name);
```

注意事项:

1. Java 使用 `双引号 + 若干字符` 的方式表示字符串字面值.
2. 和上面的类型不同, String 不是基本类型, 而是**引用类型**(后面重点解释).
3. 字符串中的一些特定的不太方便直接表示的字符需要进行转义.

转义字符示例:

```
// 创建一个字符串 My name is "张三"
String name = "My name is \"张三\"";
```

转义字符有很多, 其中几个比较常见的如下:

转义字符	解释
\n	换行
\t	水平制表符
\'	单引号
\"	双引号
\\	反斜杠

字符串的 `+` 操作, 表示字符串拼接:

```
String a = "hello";
String b = "world";
String c = a + b;
System.out.println(c);
```

还可以用字符串和整数进行拼接:

```
String str = "result = ";
int a = 10;
int b = 20;
String result = str + a + b;
System.out.println(result);

// 执行结果
result = 1020
```

以上代码说明, 当一个 `+` 表达式中存在字符串的时候, 都是执行字符串拼接行为.

因此我们可以很方便的使用 `System.out.println` 同时打印多个字符串或数字

```
int a = 10;
int b = 20;
System.out.println("a = " + a + ",b = " + b)
```

1.10 变量的作用域

也就是该变量能生效的范围, 一般是变量定义所在的代码块 (大括号)

```
class Test {
    public static void main(String[] args) {
        {
            int x = 10;
            System.out.println(x);    // 编译通过;
        }
        System.out.println(x);       // 编译失败, 找不到变量 x.
    }
}
```

1.11 变量的命名规则

硬性指标:

1. 一个变量名只能包含数字, 字母, 下划线
2. 数字不能开头.
3. 变量名是大小写敏感的. 即 num 和 Num 是两个不同的变量.

注意: 虽然语法上也允许使用中文/美元符($)命名变量, 但是 **强烈** 不推荐这样做.

软性指标:

1. 变量命名要具有描述性, 见名知意.
2. 变量名不宜使用拼音(但是不绝对).
3. 变量名的词性推荐使用名词.
4. 变量命名推荐 **小驼峰命名法**, 当一个变量名由多个单词构成的时候, 除了第一个单词之外, 其他单词首字母都大写.

小驼峰命名示例:

```
int maxValue = 100;
String studentName = "张三";
```

1.12 常量

上面讨论的都是各种规则的变量, 每种类型的变量也对应着一种相同类型的常量.

常量指的是运行时类型不能发生改变.

常量主要有以下两种体现形式:

1. 字面值常量

```
10      // int 字面值常量(十进制)
010     // int 字面值常量(八进制) 由数字 0 开头. 010 也就是十进制的 8
0x10    // int 字面值常量(十六进制) 由数字 0x 开头. 0x10 也就是十进制的 16
10L     // long 字面值常量. 也可以写作 10l (小写的L)
1.0     // double 字面值常量. 也可以写作 1.0d 或者 1.0D
1.5e2   // double 字面值常量. 科学计数法表示. 相当于 1.5 * 10^2
1.0f    // float 字面值常量, 也可以写作 1.0F
true    // boolen 字面值常量, 同样的还有 false
'a'     // char 字面值常量, 单引号中只能有一个字符
"abc"   // String 字面值常量, 双引号中可以有多个字符.
```

2. final 关键字修饰的常量

```
final int a = 10;
a = 20;     // 编译出错. 提示 无法为最终变量a分配值
```

常量不能在**程序运行过程中**发生修改.

1.12 理解类型转换

Java 作为一个强类型编程语言, 当不同类型之间的变量相互赋值的时候, 会有教严格的校验.

先看以下几个代码场景:

int 和 long/double 相互赋值

```
int a = 10;
long b = 20;
a = b;          // 编译出错, 提示可能会损失精度.
b = a;          // 编译通过.

int a = 10;
double b = 1.0;
a = b;              // 编译出错, 提示可能会损失精度.
b = a;             // 编译通过.
```

long 表示的范围更大, 可以将 int 赋值给 long, 但是不能将 long 赋值给 int.

double 表示的范围更大, 可以将 int 赋值给 double, 但是不能将 double 赋值给 int.

结论: 不同数字类型的变量之间赋值, 表示范围更小的类型能隐式转换成范围较大的类型, 反之则不行.

int 和 boolean 相互赋值

```
int a = 10;
boolean b = true;
b = a;                  // 编译出错，提示不兼容的类型
a = b;                 // 编译出错，提示不兼容的类型
```

结论: int 和 boolean 是毫不相干的两种类型, 不能相互赋值.

int字面值常量 给 byte 赋值

```
byte a = 100;      // 编译通过
byte b = 256;      // 编译报错，提示 从int转换到byte可能会有损失
```

注意: byte 表示的数据范围是 -128 -> +127, 256 已经超过范围, 而 100 还在范围之内.

结论: 使用字面值常量赋值的时候, Java 会自动进行一些检查校验, 判定赋值是否合理.

使用强制类型转换

```
int a = 0;
double b = 10.5;
a = (int)b;

int a = 10;
boolean b = false;
b = (boolean)a;           // 编译出错，提示不兼容的类型.
```

结论: 使用 (类型) 的方式可以将 double 类型强制转成 int. 但是

1. 强制类型转换可能会导致精度丢失. 如刚才的例子中, 赋值之后, 10.5 就变成 10 了, 小数点后面的部分被忽略.
2. 强制类型转换不是一定能成功, 互不相干的类型之间无法强转.

类型转换小结

1. 不同数字类型的变量之间赋值, 表示范围更小的类型能隐式转换成范围较大的类型.
2. 如果需要把范围大的类型赋值给范围小的, 需要强制类型转换, 但是**可能精度丢失**.
3. 将一个字面值常量进行赋值的时候, Java 会自动针对数字范围进行检查.

1.13 理解数值提升

int 和 long 混合运算

```
int a = 10;
long b = 20;
int c = a + b;          // 编译出错，提示将 long 转成 int 会丢失精度
long d = a + b;         // 编译通过.
```

结论: 当 int 和 long 混合运算的时候, **int 会提升成 long**, 得到的结果仍然是 long 类型, 需要使用 long 类型的变量来接收结果. 如果非要用 int 来接收结果, 就需要使用强制类型转换.

byte 和 byte 的运算

```
byte a = 10;
byte b = 20;
byte c = a + b;
System.out.println(c);

// 编译报错
Test.java:5: 错误: 不兼容的类型: 从int转换到byte可能会有损失
                byte c = a + b;
                         ^
```

结论: byte 和 byte 都是相同类型, 但是出现编译报错. 原因是, 虽然 a 和 b 都是 byte, 但是计算 a + b 会先将 a 和 b 都提升成 int, 再进行计算, 得到的结果也是 int, 这是赋给 c, 就会出现上述错误.

由于计算机的 CPU 通常是按照 4 个字节为单位从内存中读写数据. 为了硬件上实现方便, 诸如 byte 和 short 这种低于 4 个字节的类型, 会先提升成 int, 再参与计算.

正确的写法:

```
byte a = 10;
byte b = 20;
byte c = (byte)(a + b);
System.out.println(c);
```

类型提升小结:

1. 不同类型的数据混合运算, 范围小的会提升成范围大的.
2. 对于 short, byte 这种比 4 个字节小的类型, 会先提升成 4 个字节的 int , 再运算.

1.14 int 和 String 之间的相互转换

int 转成 String

```
int num = 10;
// 方法1
String str1 = num + "";
// 方法2
String str2 = String.valueOf(num);
```

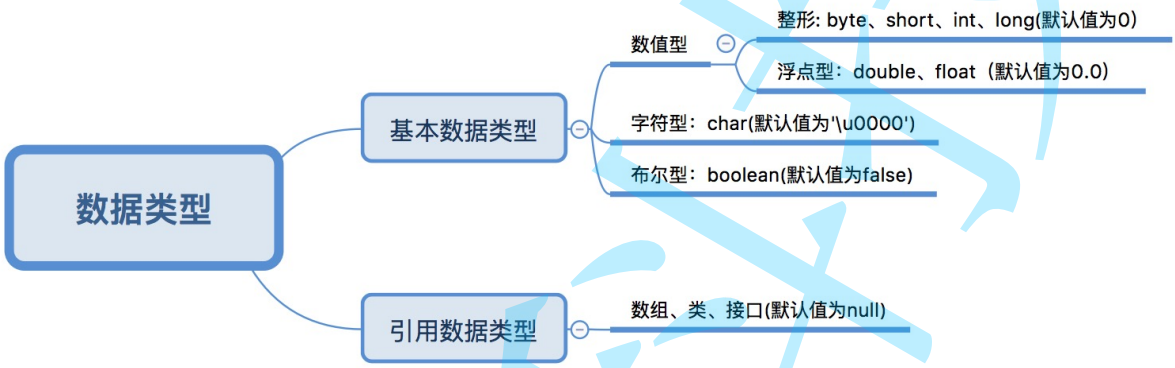
String 转成 int

```
String str = "100";

int num = Integer.parseInt(str);
```

1.15 小结

Java 类型汇总. 前面的内容重点介绍的是基本数据类型.



每种数据类型及其范围, 是需要我们掌握的重点.

隐式类型转换和类型提升, 是本节的难点. 但是一般我们更推荐在代码中避免不同类型混用的情况, 来规避类型转换和类型提升的问题.

2. 运算符

2.1 算术运算符

- 基本四则运算符 `+ - * / %`

规则比较简单, 值得注意的是除法:

a) **int / int 结果还是 int**, 需要使用 double 来计算.

```
int a = 1;
int b = 2;
System.out.println(a / b);

// 结果为 0
```

b) **0 不能作为除数**

```
int a = 1;
int b = 0;
System.out.println(a / b)

// 运行结果
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.main(Test.java:5)
```

c) **% 表示取余**, 不仅仅可以对 int 求模, 也能对 double 来求模

```
System.out.println(11.5 % 2.0);

// 运行结果
1.5
```

- 增量赋值运算符 `+= -= *= /= %=`

```
int a = 10;
a += 1;             // 等价于 a = a + 1
System.out.println(a);
```

- 自增/自减运算符 `++ --`

```
int a = 10;
int b = ++a;
System.out.println(b);
int c = a++;
System.out.println(c);
```

结论:

1. 如果不取自增运算的表达式的返回值, 则前置自增和后置自增没有区别.
2. 如果取表达式的返回值, 则前置自增的返回值是自增之后的值, 后置自增的返回值是自增之前的值.

2.2 关系运算符

关系运算符主要有六个:

`== != < > <= >=`

```
int a = 10;
int b = 20;
System.out.println(a == b);
System.out.println(a != b);
System.out.println(a < b);
System.out.println(a > b);
System.out.println(a <= b);
System.out.println(a >= b);
```

注意: 关系运算符的表达式返回值都是 boolean 类型.

2.3 逻辑运算符(重点)

逻辑运算符主要有三个:

`&& || !`

注意: 逻辑运算符的操作数(操作数往往是关系运算符的结果)和返回值都是 boolean .

逻辑与 &&

规则: 两个操作数都为 true, 结果为 true, 否则结果为 false.

```
int a = 10;
int b = 20;
int c = 30;
System.out.println(a < b && b < c);
```

逻辑或 ||

规则: 两个操作数都为 false, 结果为 false, 否则结果为 true

```
int a = 10;
int b = 20;
int c = 30;
System.out.println(a < b || b < c);
```

逻辑非 !

规则: 操作数为 true, 结果为 false; 操作数为 false, 结果为 true(这是个单目运算符, 只有一个操作数).

```
int a = 10;
int b = 20;
System.out.println(!a < b);
```

短路求值

&& 和 || 遵守短路求值的规则.

```
System.out.println(10 > 20 && 10 / 0 == 0);             // 打印 false
System.out.println(10 < 20 || 10 / 0 == 0);             // 打印 true
```

我们都知道, 计算 `10 / 0` 会导致程序抛出异常. 但是上面的代码却能正常运行, 说明 `10 / 0` 并没有真正被求值.

结论:

1. 对于 && , 如果左侧表达式值为 false, 则表达式的整体的值一定是 false, 无需计算右侧表达式.

2. 对于 ||, 如果左侧表达式值为 true, 则表达式的整体的值一定是 true, 无需计算右侧表达式.

& 和 | (不推荐使用)

& 和 | 如果操作数为 boolean 的时候, 也表示逻辑运算. 但是和 && 以及 || 相比, 它们不支持短路求值.

```
System.out.println(10 > 20 & 10 / 0 == 0);              // 程序抛出异常
System.out.println(10 < 20 | 10 / 0 == 0);              // 程序抛出异常
```

2.4 位运算符

Java 中对数据的操作的最小单位不是字节, 而是二进制位.

位运算符主要有四个:

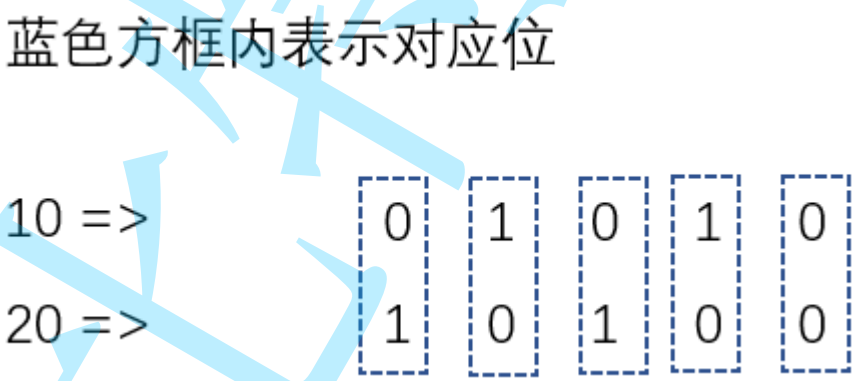
`& | ~ ^`

位操作表示 **按二进制位运算**. 计算机中都是使用二进制来表示数据的(01构成的序列), 按位运算就是在按照二进制位的每一位依次进行计算.

按位与 &: 如果两个二进制位都是 1, 则结果为 1, 否则结果为 0.

```
int a = 10;
int b = 20;
System.out.println(a & b);
```

进行按位运算, 需要先把 10 和 20 转成二进制, 分别为 1010 和 10100



按位或 |: 如果两个二进制位都是 0, 则结果为 0, 否则结果为 1.

```
int a = 10;
int b = 20;
System.out.println(a | b);
```

运算方式和按位于类似.

注意: 当 & 和 | 的操作数为整数(int, short, long, byte) 的时候, 表示按位运算, 当操作数为 boolean 的时候, 表示逻辑运算.

按位取反 ~: 如果该位为 0 则转为 1, 如果该位为 1 则转为 0

```
int a = 0xf;
System.out.printf("%x\n", ~a)
```

注意:

1. 0x 前缀的数字为 十六进制 数字. 十六进制可以看成是二进制的简化表示方式. 一个十六进制数字对应 4 个二进制位.
2. 0xf 表示 10 进制的 15, 也就是二进制的 1111
3. printf 能够格式化输出内容, %x 表示按照十六进制输出.
4. \n 表示换行符

按位异或 ^: 如果两个数字的二进制位相同, 则结果为 0, 相异则结果为 1.

```
int a = 0x1;
int b = 0x2;
System.out.printf("%x\n", a ^ b);
```

2.5 移位运算(了解)

移位运算符有三个:

`<< >> >>>`

都是按照二进制位来运算.

左移 <<: 最左侧位不要了, 最右侧补 0.

```
int a = 0x10;
System.out.printf("%x\n", a << 1);

// 运行结果(注意, 是按十六进制打印的)
20
```

右移 >>: 最右侧位不要了, 最左侧补符号位(正数补0, 负数补1)

```
int a = 0x10;
System.out.printf("%x\n", a >> 1);

// 运行结果(注意，是按十六进制打印的)
8

int b = 0xffff0000;
System.out.printf("%x\n", b >> 1);

// 运行结果(注意，是按十六进制打印的)
ffff8000
```

无符号右移 >>>: 最右侧位不要了, 最左侧补 0.

```
int a = 0xffffffff;
System.out.printf("%x\n", a >>> 1);

// 运行结果(注意，是按十六进制打印的)
7fffffff
```

注意:

1. 左移 1 位, 相当于原数字 * 2. 左移 N 位, 相当于原数字 * 2 的N次方.
2. 右移 1 位, 相当于原数字 / 2. 右移 N 位, 相当于原数字 / 2 的N次方.
3. 由于计算机计算移位效率高于计算乘除, 当某个代码正好乘除 2 的N次方的时候可以用移位运算代替.
4. 移动负数位或者移位位数过大都没有意义.

2.6 条件运算符

条件运算符只有一个:

`表达式1 ? 表达式2 : 表达式3`

当 `表达式1` 的值为 true 时, 整个表达式的值为 `表达式2` 的值; 当 `表达式1` 的值为 false 时, 整个表达式的值为 `表达式3` 的值.

也是 Java 中唯一的一个 **三目运算符**, 是条件判断语句的简化写法.

```
// 求两个整数的最大值
int a = 10;
int b = 20;
int max = a > b ? a : b;
```

2.7 运算符的优先级

先看一段代码

```
System.out.println(1 + 2 * 3);
```

结果为 7, 说明先计算了 `2*3`, 再计算 `1+`

另外一个例子

```
System.out.println(10 < 20 && 20 < 30);
```

此时明显是先计算的 10 < 20 和 20 < 30, 再计算 &&. 否则 20 && 20 这样的操作是语法上有误的(&& 的操作数只能是 boolean).

运算符之间是有**优先级**的. 具体的规则我们**不必记忆**. 在可能存在歧义的代码中加上括号即可.

2.8 小结

1. % 操作再 Java 中也能针对 double 来计算.
2. 需要区分清楚 前置自增 和 后置自增之间的区别.
3. 由于 Java 是强类型语言, 因此对于类型检查较严格, 因此像 && 之类的运算操作数必须是 boolean.
4. 要区分清楚 & 和 | 什么时候是表示按位运算, 什么时候表示逻辑运算.

整体来看, Java 的运算符的基本规则和 C 语言基本一致.

3. 注释

注释是为了让代码更容易被读懂而附加的描述信息. 不参与编译运行, 但是却非常重要.

时刻牢记! 代码写出来是为了给人看的, 更是为了给三个月后的你自己看的.

3.1 基本规则

Java中的注释主要分为以下三种

> - 单行注释：// 注释内容（用的最多）
> - 多行注释：/* 注释内容*/（不推荐）
> - 文档注释： /** 文档注释 */（常见于方法和类之上描述方法和类的作用），可用来自动生成文档

3.2 注释规范

1. 内容准确: 注释内容要和代码一致, 匹配, 并在代码修改时及时更新.
2. 篇幅合理: 注释既不应该太精简, 也不应该长篇大论.
3. 使用中文: 一般中国公司都要求使用中文写注释, 外企另当别论.
4. 积极向上: 注释中不要包含负能量(例如 领导 SB 等).

4. 关键字

关键字是 Java 中的一些具有特定含义的单词.

用于定义访问权限修饰符的关键字				
private	protected	public		
用于定义类，函数，变量修饰符的关键字				
abstract	final	static	synchronized	
用于定义类与类之间关系的关键字				
extends	implements			
用于定义建立实例及引用实例，判断实例的关键字				
new	this	super	instanceof	
用于异常处理的关键字				
try	catch	finally	throw	throws
用于包的关键字				
package	import			
其他修饰符关键字				
native	strictfp	transient	volatile	assert

随着课程内容的展开, 我们会逐渐学到上面内容.

另外, 定义的变量名不能和关键字冲突.

作业

1. 写代码实现: 给定两个 int 变量, 交换变量的值.
2. 写代码实现: 给定三个 int 变量, 求其中的最大值和最小值.
3. 写博客总结: 变量和运算符的基本知识点.
4. 写博客总结: 给定一个十进制整数, 如何转成二进制形式? 如何转成十六进制形式?
5. 课外阅读: 查找资料, 了解 "冯诺依曼体系结构".