常量（constant）

A. 字面常量

- 字符串常量（双引号括起来的内容）
- 字符常量（单引号括起来的单个字符、数字或符号，空格也算，不能为空）
- 整型常量（自然数例如12345等等）
- 浮点型常量（带有小数点的数字）
- 布尔常量（true真和false假）
- 空常量（null）

```
class ConstantTest{
    public static void main(String[] args){
        System.out.println("abcd");// 字符串常量
        System.out.println('a');// 字符常量
        System.out.println(1234);//整型常量
        System.out.println(1.2);//浮点型常量
        System.out.println(true);// 布尔常量
        System.out.println(null);//空常量
    }
}
```

进制（scale）

A. 理解

> 1个开关有开和关两种状态表示0，1。计算机规定8个开关（0/1）为基本单位，称8bit（位）。

- 1byte（字节） = 8 bit（位）
- 1kbyte（千字节） = 1024 byte（字节）
- 1mbyte（兆字节） = 1024 kbyte
- 1gbyte（千兆字节） = 1024 mbyte

B. 各种进制

- 二进制：0和1，逢二进一：1+1=10（二进制）=2（十进制）
- 八进制：0~7，逢八进一：1+7=10（八进制）=8（十进制）
- 十进制：0~9，逢十进一：1+9=10（十进制）
- 十六进制：0~9A~F，逢十六进一：1+F=10（十六进制）=16（十进制）

```
class ScaleTest{
    public static void main(String[] args){
        /* java7之后 */
        System.out.println(0b10); // 前面加0b，二进制写法，输出2
        System.out.println(010); // 前面加0，八进制写法，输出8
        System.out.println(10);// 普通十进制
        System.out.println(0x10); //前面加0x，十六进制写法，输出16
    }
}
```

C. 进制转换

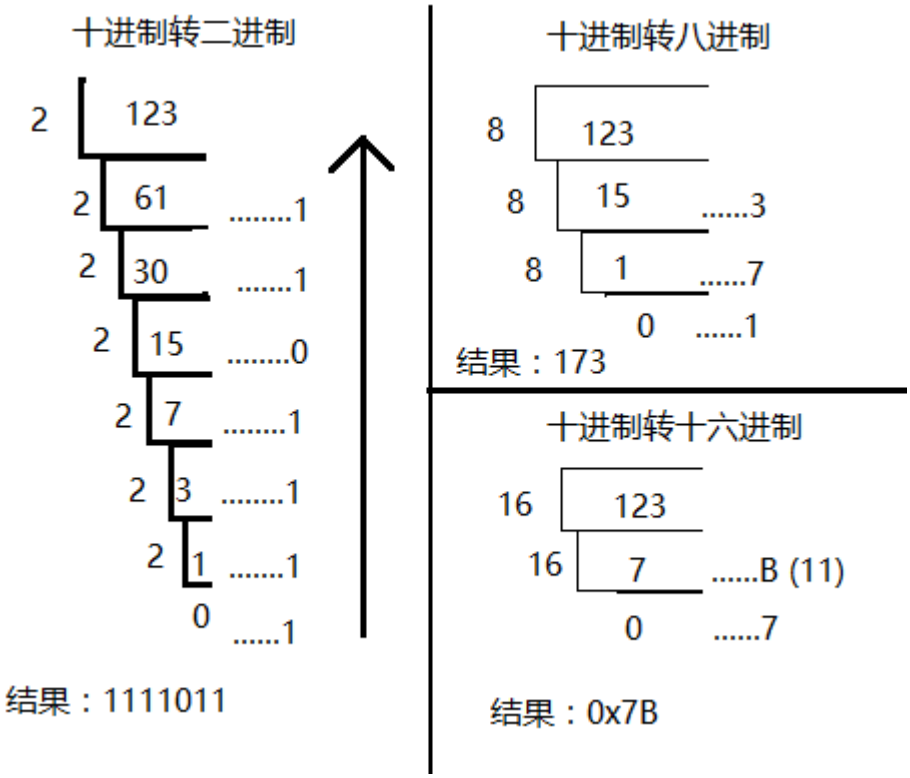
1. **X** 进制转换为十进制

- 系数：每位数（e.g. 123中的1、2、3）
- 基数：X进制，X为基数
- 权：从右数开始数，0开始编号（e.g 12345,权为4）
- 结果：SUM（系数 * 基数^权）

```
  二进制转十进制
  0b10010 = (1 * 2^4) + (0 * 2^3) + (0 * 2^2) + (1 * 2^1) + (0 * 2^0)
= 18
  八进制转十进制
  01212 = (1 * 8^3) + (2 * 8^2) + (1 * 8^1) + (2 * 8^0) = 512+128+8 =
648
  十六进制转十进制
  0x2b = (2 * 16^1) + (11 * 16^0) = 43
```

2. 十进制转换为 **X** 进制

- 公式： 十进制数值除以 X 进制直至商为0，再从下到上取其余数



3. 任意进制转换为任意进制

> 8421 快速转换法：第3、2、1、0位对应的8、4、2和1

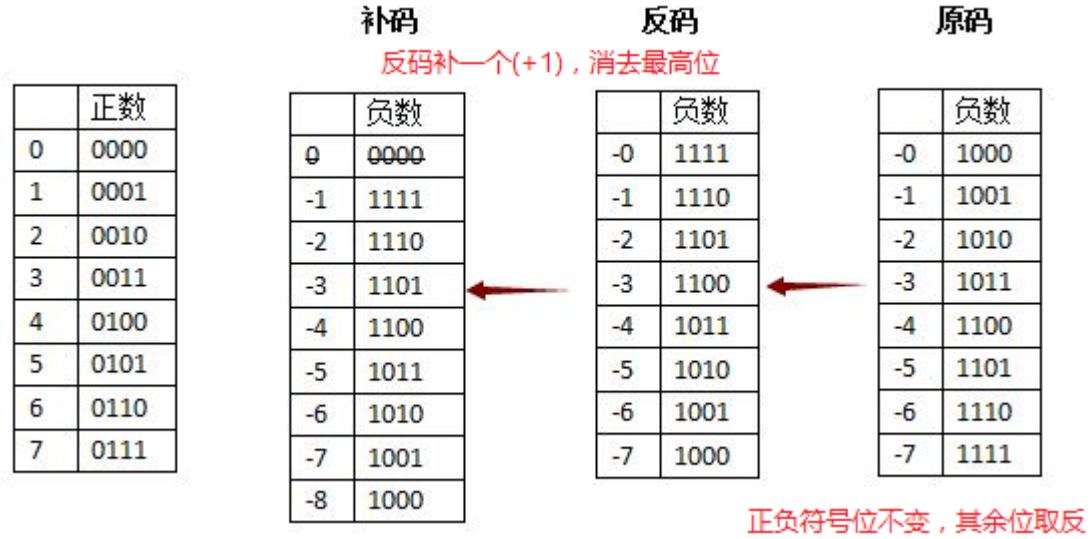
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**
1*2^7	1*2^6	1*2^5	1*2^4	1*2^3	1*2^2	1*1^7	1*0^7
128	64	32	16	8	4	2	1

- 十进制快速转换为二进制：
  - 123比第7位的128小故取0
  - 123比第6位的64大故取1，且123-64=59
  - 59比第5位的32大故取1，且59-32=27
  - 27比第4位的16大故取1，且27-16=11
  - 11比第3位的8大故取1，且11-8=3
  - 3比第2位的4小故取0
  - 3比第1位的2大故取1，且3-2=1
  - 1和第0位的1相等故取1，且1-1=0
  - 结果：**123 = 01111011**
- 二进制快速转换为八进制：
  - 1111011每3位为1单位：1、111和011
  - 第一个单位1对照表第0位的状态也是1，故取其值1
  - 第二单位111分别对应表的第0、1和2位，因其状态都为1，故取其值4+2+1=7
  - 第三单位的011分别对应表的第0、1和2位，因011的第2位为0，故不取表对应的第2位的值，所以剩下的1+2=3
  - 结果： **173**
- 二进制快速转换为十六进制：
  - 1111011每4位为1单位：111和1011

    - 第一单位1111对应表中的第0、1和2位，因为都为1，故取其值4+2+1=7
    - 第二单位1011同上，但有0的存在，所以对应表中相应第2位的值不取，剩下的8+2+1=11，即B
    - 结果： **0x7B**
- 十进制要想快速转换为其他进制，就要先转换为二进制。

D. 机器正负运算（原码、反码和补码）

- 原码：最高位存放正负，正为0，负为1
    - e.g. 1的正二进制为0001，负二进制为1001；同样3的正二进制为0011，负二进制为1011.
    - 方便人类快速识别二进制正负，但阻碍了计算机运算。
    - e.g. （+1）+（-1）应等0，但计算机算出0001+1001=1010。如果按照原码的方式看则为-2.明显不对。
    - 另一个问题是存在（**+0**）和（**-0**）两个**0**
- 反码：专门用于处理负数，符号位置不变，其余位取反
    - e.g. 原码中负数1010（-2）转换为反码1101（-2）
    - 原码转换为反码，可以解决“正负相加等于0”
    - 但还剩下（**+0**）和（**-0**）的问题
- 补码：在原来反码的基础上，补充或加上一个新的码（+1）
    - e.g. 反码中的负数1111（-0）加上（+1）转换为补码0000（0），1110（反码-1）转为1111（补码-1），其他依次类推。
    - 当反码1111（-0）补1之后，变成10000，丢掉最高位就是0000，干好是正数的0。
    - 正负数相加等于**0**的问题同样满足
    - e.g. 3和（-3）相加，0011 + 1101 = 10000，去掉最高位，就是0000，全面解决问题！



	正数
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

	负数
~~0~~	~~0000~~
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

	负数
-0	1111
-1	1110
-2	1101
-3	1100
-4	1011
-5	1010
-6	1001
-7	1000

	负数
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

- 正负运算：
    - 1减去3
    - = 1+(-3)
    - = 0001 + 1011 (原码)
    - = 0001 + 1100 (反码)

    - = 0001 + 1101 (补码)
    - = 1110 (补码)，将其再转回原码：
    - = 1101 (反码)
    - = 1010 (原码，最高位1为负，010为2，故为-2)

- 已知补码，求原码：

    - 如果该补码的符号为正，则其原码就是这个补码
    - 如果该补码的符号为负，则对该补码再次求补码就是要求的原码。

- 负数的补码：

    - 对负数的绝对值二进制全部取反(包括符号位)再加1,则为该负数的补码
    - e.g. -35的补码
    - -35的的绝对值35的二进制为00100011
    - 全部取反为11011100
    - 再加1为11011101,这就是-35的补码

- A减去B = A加(-B)的补码（A、B>0）

- 反正运算时都得将参与运算的数转换为补码运算，最后结果转回原码，取正负和尾数就是答案。

数据类型

> Java设定具体的数据类型一个重要原因是可以更好地利用内存资源。

A. 数据类型分类

1. 基本数据类型（**4**类**8**种）

- 整形
    - **byte**：占一个字节**(8bit)** （**-128**到**127**）
    - short：占两个字节 （-2^15 ~ 2^15 - 1）
    - int：占四个字节 （-2^31 ~ 2^31 - 1）
    - long：占八个字节 （-2^63 ~ 2^63 - 1)
- 浮点型
    - float：占四个字节 （-3.403E38 ~ 3.403E38）单精度
    - double：占八个字节 （-1.798E308 ~ 1.798E308） 双精度
- 字符型
    - char：占两个字节 （0 ~ 65535）只有正的
- 布尔型
    - **boolean**：理论上占八分之一个字节**(1bit**位**),**但**Java**没有明确指定布尔型的空间大小。

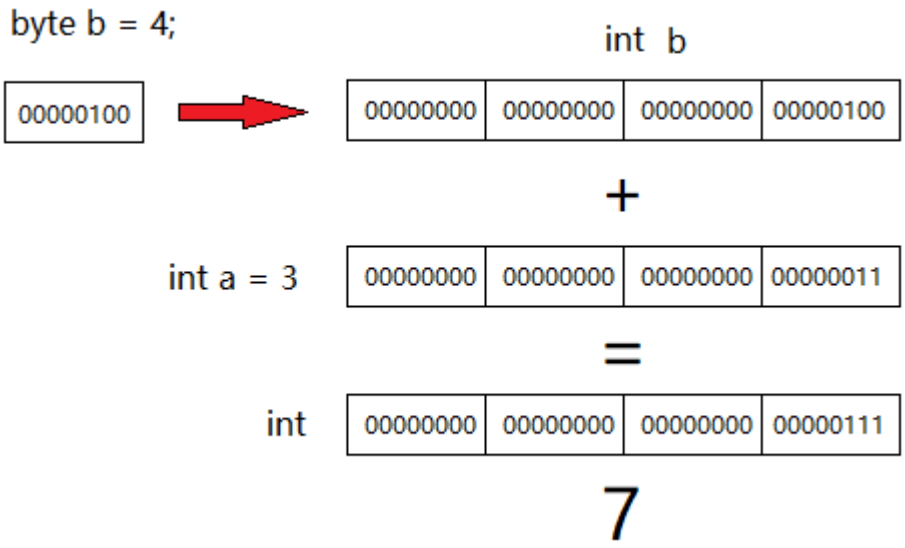
2. 引用数据类型

> 占位

3. 数据类型转换_隐式转换

> 当空间大的数据类型和较之空间小的数据类型进行运算时，会先将空间小的数据类型转换为空间大的数据类型再进行运算。

```
class DataTypeConversionTest{
    public static void main(String args[]){
        int a = 3;
        byte b = 5;
        byte c = a + b;
        System.out.println(c);
        // 发生编译错误：不兼容的类型： 从int转换到byte可能会有损失
    }
}
```



4. 数据类型转换_强制转换(高级向低级转换)

> 高级类型数据和低级类型数据运算结果为高级类型，如果将结果赋给低级类型声明的值，则需要在前面加(低级类型)来将高级类型数据强制转换为低级类型数据。

```
class DataTypeConversionForce{
    /*强制转换*/
    DataTypeConversionForce(){
        int i = 3;
        byte b = 4;
        // 前面加上 (byte) 强制转换为byte类型
        byte result = (byte)(i + b);
        System.out.println(result);//输出7,因为7在-128到127之间，所以编译器允许赋值给byte类型。

        int i2 = 128;
        byte b2 = 10;
        byte result2 = (byte)(i2 + b2);
        // 输出是-118,因为i2+b2=138,已经超出byte的取值范围-128～127了，损失精度(砍掉前面三个字节)。

        byte b3 = (byte)300;
        //输出44
```

```
    }
}
```

> 上面例子，强制转换会将int类型(占4个byte)的result去掉前面的3个byte，只留下一个byte，就转变成byte类型了(占1个byte)。这样的话，如果result的值超过byte类型的空间大小(-128～127),即超过一字节，就会损失精度。

> 例如result2的结果竟是-118,这是因为机器运算得出的结果是10001010(138),这是-118的补码，将其补码转回原码就是-118了。

> b3的输出竟然是44,300的二进制是100101100，9位，因为int转byte会砍掉前面3个字节，所以最后只剩下 00101100，因为最高位为0即正数，所以求它的原码就是它的补码，00101100转为十进制为44.

面试题_变量相加和常量相加对于类型转换的区别:

```
void DataTypeConversionForce_2(){
        /*数据类型强制转换中的变量相加和常量相加*/
        byte b1 = 3;
        byte b2 = 4;
        /*报错，1、因为3和4是int类型，运算结果也是int。将int数据赋值给byte类型会损失精度
        * 2、 b1和b2是两个变量，变量储存的值是变化得，编译时无法判断里面具体得值，相加可能会超出byte范围*/
        byte b3 = b1 + b2;

        /*常量相加*/
        byte b4 = 3 + 4;
        /*不报错，因为java编译器有常量优化机制
        * 编译时，已知常量3和4,且不再改变，运算得出的结果如果判断超过byte
        * 范围，且未进行强制转换，则报错
        * 否则赋值*/
    }
```

5. 不同数据类型的变量定义

```
class DataTypeDefineTest{
    public static void main(String[] args){
        /* 加上后缀只是告诉JVM这是什么类型的值，并没有强制转换的发生 */
        /* 整型 */
        byte b = 1; // 占一个字节 -128~127
        short s = 12; // 占两个字节  -2*10^15~2*10^15-1
        int i = 123; //所有整数值默认为int
        long l = 12344L; //因为所有整数值默认为int，最好加L后缀告诉JVM这是long，防止出错。不加也不影响，因为JVM会自动转换为long，前提是不超过取值范围

        /* 浮点型 */
        float f = 1.33F; //因为所有浮点型数值默认为double型，不加F标识的话以为着将double类型的值赋给float，报错
        double d = 2.333; //浮点型数值默认类型
```

```
        /* 字符型 */
        char c = 'c';

        /* 布尔型 */
        boolean tf = true;//或者false
    }
}
```

6. 字符和字符串参与运算

- 字符(char)类型数据与int类型数据相加，会先将字符根据ASCII码表转换为int类型数字，再和int类型数据运算。

```
char c = 'c'; // 'c'字符在ASCII码表中是99
int i = 1;

System.out.println(c+i);// 输出100，99+1=100
System.out.println((char)(c+i));// 输出'd'字符，100对应
System.out.println('a'+'c');//输出196,char一旦参与数学运算都将转换为int
```

```
char c1 = 97;
char c2 = 98;
System.out.println(c1);//输出a
System.out.println(c2);//输出b
/*为什么输出ab？因为char的取值范围是0～65535，97和98不超出
* 范围，所以编译器将其当作ASCII码处理*/
```

> java中的char可以储存一个中文字符么？为什么？

可以，因为java采用Unicode编码。Unicode编码中的每个字符占2个字节，中文单个汉字也占2个字节，所以char可以储存一个中文汉字.

- 任何类型数据用 + 与字符串相连接都会产生新的字符串。按照运算优先级产生的内容也不相同。

```
/*任何类型数据用 + 和字符串连接(强调连接而不是运算)都产生新的字符串*/
System.out.println("hello"+2); // 输出 hello2
System.out.println("hello" + 'c'); //输出 helloc
System.out.println("hello"+true); // 输出 hellotrue
System.out.println(2 + 'c' + "hello");// 输出 101hello
System.out.println("hello" + 2 + 'c');// 输出 hello2c
System.out.println("5 + 5 = " + 5+5);// 输出 5 + 5 = 55
System.out.println("5 + 5 = " + (5+5));// 输出 5 + 5 = 10,优先级
```

> 注：因为是强调“连接”，而不是运算，所以字符char不会转换为int，这就是为啥输出helloc。

> 一般按照从左到右执行，没有加括号的话。

运算符(operator)

> 对常量和变量进行操作的符号。

- 算术运算符：+、-、*、/、%、++、--
- 赋值运算符：=、+=、-=
- 比较(关系/条件)运算符：>、<、>=、<=、!=
- 逻辑运算符：&&、||
- 位运算符：
- 三目(元)运算符

> 注意：+号三种作用，正、加法、字符串连接符

A.算术运算符

1. %(模，取余)：

```
void Demo(){
    /* % 模运算符，取余*/
    /* % 运算结果的正负根据左边值正负*/

    /*当左边值的绝对值小于右边绝对值时，结果为左边*/
    System.out.println(-3 % 5);//输出 -3
    System.out.println(-3 % -5);//输出 -3
    System.out.println(-12 % 5);//输出 -2

    /*当左边的绝对值等于右边或右边的倍数时，结果是0*/
    System.out.println(-9 % 3);// 输出 0
    System.out.println(-9 % -3);// 输出 0
    System.out.println(-3 % -3);// 输出 0

    /*当左边绝对值大于右边绝对值时，结果为余数*/
    System.out.println(-4 % 3);// 输出 -1
    System.out.println(4 % 3);// 输出 1
    System.out.println(4 % -3);// 输出 1

    /*奇数 %2 为1,偶数 %2 为0，,可用作切换条件*/
    System.out.println(4 % 2);// 输出 0
    System.out.println(5 % 2);// 输出 1
    System.out.println(6 % 2);// 输出 0
}
```

2. ++a先自增,a++后自增(减):

> 在未参与运算(或赋值)时，一般a++和++a的效果是一样的

```
int i = 1;
i++; // 输出2

int j = 2;
++j; // 输出3
```

> 但一旦参与运算或赋值，a++就会先将a赋给目标，然后才自增；而++a则先自增后再赋给目标。

```
int a = 1;
int b = a++; // 输出b为1，a为2

int c = 1;
int d = ++c; // 输出d为2，c为2

int x = 1;
int y = (x++)+(++x)+(x*10);
/*  开始x参与运算，先将x初始值取出(1)，然后自增，此时x已经变成2。然后到第二步(++x),这里先自增x变成了3。第三步x*10则为30。所以最终结果为 1+3+3x10=34 */
```

面试题_下面那一句会报错：

```
byte b = 10;
b++;// 这等价于 b = (byte)(b + 1) JVM自动强制转换，所以这句不报错.
b = b + 1;// JVM会先将b(byte)变量提升为int然后再参与运算，结果为int，无法赋值给byte类型，需要手动强制转换
```

B. 逻辑运算符

1.异或(^)

> 真假异或

```
符号左右两边同为true或false时，则结果为false。
否则两边分别是true和false时，则结果为true。
总结：两者同则false，异则true
```

> 异或运算

```
4 ^ 6:
= 100 ^ 110 (二进制)
= 010 (同则0,异则1)
= 2 (十进制)
```

2.与(&/&&)、或(|/||):

> 与&：两者同时成立则true，否则false

> 与&&：和&相同效果，但有短路功能=>A&&B，如果A判定为false，则不需要再判断B了，因为这足可以判定整个条件不成立了。

> 或|：两者之一成立则true，否则false

> 或 ||：同上，A||B,如果A为true，则不需判断B了，因为已经有一个成立了，足以判断条件是否成立了。

短路功能可以节省计算资源。

C.位运算符

> 位与运算：

```
4 & 6:
= 100 & 110
= 100 (同1则1，否则0)和逻辑与一样
= 4
```

> 位或运算：

```
例子:
4 | 6:
= 100 | 110
= 110 (有1则1,否则0)
= 6

// 简单模式
4 | -7:
= 100 | 001(-7的补码)
= 101 (结果也是补码，要转为原码)
= 110 (尾数取反，符号位不变)
= 111
= -3 (原码)
等价于:
// 具体模式
= 00000000 00000000 00000000 00000100 | 11111111 11111111 11111111 11111001
= 11111111 11111111 11111111 11111101
= 10000000 00000000 00000000 00000010
= 10000000 00000000 00000000 00000011
= -3
```

> 位取反运算：

```
~4:
= 00000000 00000000 00000000 00000100
= 11111111 11111111 11111111 11111011 (取反运算:全体取反), //这里是补码，需要转换为原码来看
= 10000000 00000000 00000000 00000100 (尾数位取反)
= 10000000 00000000 00000000 00000101 (补码)
= -5 (原码)
// 为什么这么多01,因为4是int类型，所以最高位是第31位即符号位
```

面试题_实现两个变量的交换

```
/* 异或运算的特点:
*  一个数A对另一个数B异或运算两次，还是这个数A,变回自己 */
// 推荐方案
int a = 10;
int b = 3;
a = a ^ b; // 10 ^ 3
b = a ^ b; // 10 ^ 3 ^ 3 = 10
a = a ^ b; // 10 ^ 3 ^ 10 = 3
// a = 3, b = 10 交换成功

// 另一个方案
a = a + b; // 10+3=13
b = a - b; // 13-3=10
a = a - b; // 13-10=3
// a=3, b=10 交换成功，但是缺陷是如果a和b运算结果超出int取值范围，就会报错

// 又一个方案
int temp = a; // temp=10
a = b; // a=3
b = temp; // b=10
// 开发推荐，但面试可能会限制这个中间条件
```

D.位运算符（<<、>>）

1.左移<<

二进制位向左移 N 位，等价于该数乘2的 N 次方，左边移出去N位，右边补N个0。

```
12 << 1:
00000000 00000000 00000000 00001100 =
00000000 00000000 00000000 00011000 = 24
```

2.右移>>和无符号右移>>>

```
右移 N 位，等价于该数除以2的 N 次方，左边最高位若是1则补1,否则补0.
```

```
12 >> 3:
00000000 00000000 00000000 00001100 =
00000000 00000000 00000000 00000001 = 1

2147483656 >>> 1:无符号右移
10000000 00000000 00000000 00001000 =
01000000 00000000 00000000 00000100 = 1073741828
```

键盘输入

- 导包：`import java.util.Scanner;`
- 创建扫描对象：`Scanner sc = new Scanner(System.in);`
- 通过该对象获取数据：`int x = sc.nextInt();//整型获取`

流程控制结构

- 顺序结构： 普通从上到下依次执行
- 选择结构： 也叫分支结构，if/else和switch语句
- 循环结构： for、while、do/while和foreach语句

A.选择结构(分支结构)

- 格式1：

```
int age = 19;
if (age >= 18){
    // 语句
}
```

- 格式2：

```
// 不加大括号，则只控制离它最近的一句话，是一句话！！！
if (age < 11)
    int a = 13; // 报错，因为该语句共两句，int a和a = 13
```