JAVA编程中的须注意的细节

数值表达式

1. 奇偶判断

不要使用 i % 2 == 1 来判断是否是奇数，因为 i 为负奇数时不成立，请使用 i % 2 != 0 来判断是否是奇数，或使用高效式 (i & 1) != 0 来判断。

2. 小数精确计算

System.*out*.println(2.00 -1.10);//0.8999999999999999
上面的计算出的结果不是 0.9，而是一连串的小数。问题在于 1.1 这个数字不能被精确表示为一个 double，因此它被表示为最接近它的 double 值，该程序从 2 中减去的就是这个值，但这个计算的结果并不是最接近 0.9 的 double 值。
一般地说，问题在于并不是所有的小数都可以用二进制浮点数精确表示。
二进制浮点对于货币计算是非常不适合的，因为它不可能将 1.0 表示成 10 的其他任何负次幂。

解决问题的第一种方式是使用货币的最小单位（分）来表示：System.*out*.println(200-110);//90

第二种方式是使用BigDecimal，但一定要用BigDecimal(String)构造器，而千万不要用BigDecimal(double)来构造（也不能将float或double型转换成String再来使用BigDecimal(String)来构造，因为在将float或double转换成String时精度已丢失）。例如new BigDecimal(0.1)，它将返回一个BigDecimal，也即0.1000000000000000055511151231257827021181583404541015625，正确使用BigDecimal，程序就可以打印出我们所期望的结果0.9：
System.out.println(**new** BigDecimal("2.0").subtract(**new** BigDecimal("1.10")));// 0.9

另外，如果要比较两个浮点数的大小，要使用BigDecimal的compareTo方法。

3. int整数相乘溢出

我们计算一天中的微秒数：
long microsPerDay = 24 * 60 * 60 * 1000 * 1000;// 正确结果应为：86400000000
System.*out*.println(microsPerDay);// 实际上为：500654080

问题在于计算过程中溢出了。这个计算式完全是以 int 运算来执行的，并且只有在运算完成之后，其结果才被提升为 long，而此时已经太迟：计算已经溢出。

解决方法使计算表达式的第一个因子明确为 long 型，这样可以强制表达式中所有的后续计算都用 long 运算来完成，这样结果就不会溢出：

long microsPerDay = 24L * 60 * 60 * 1000 * 1000;

4. 负的十六进制与八进制字面常量

“数字字面常量”的类型都是 int 型，而不管他们是几进制，所以“2147483648”、“0x180000000（十六进制，共 33 位，所以超过了整数的取值范围）”字面常量是错误的，编译时会报超过 int 的取值范围了，所以要确定以 long 来表示“2147483648L”、“0x180000000L”。

十进制字面常量只有一个特性，即所有的十进制字面常量都是正数，如果想写一个负的十进制，则需要在正的十进制字面常量前加上“-”即可。

十六进制或八进制字面常量可就不一定是正数或负数，是正还是负，则要根据当前情况看：如果十六进制和八进制字面常量的最高位被设置成了 1，那么它们就是负数：

System.*out*.println(0x80);//128
//0x81 看作是 int 型，最高位(第 32 位)为 0，所以是正数
System.*out*.println(0x81);//129
System.*out*.println(0x8001);//32769
System.*out*.println(0x70000001);//1879048193
//字面量 0x80000001 为int型，最高位(第 32 位)为 1，所以是负数
System.*out*.println(0x80000001);//-2147483647
//字面量 0x80000001L 强制转为 long 型，最高位（第 64 位）为 0，所以是正数
System.*out*.println(0x80000001L);//2147483649
//最小int型
System.*out*.println(0x80000000);//-2147483648
//只要超过 32 位，就需要在字面常量后加 L 强转 long，否则编译时出错
System.*out*.println(0x8000000000000000L);//-9223372036854775808

从上面可以看出，十六进制的字面常量表示的是 int 型，如果超过 32 位，则需要在后面加“L”，否则编译过不过。如果为 32，则为负 int 正数，超过 32 位，则为 long 型，但需明确指定为 long。

System.*out*.println(Long.*toHexString*(0x100000000L + 0xcafebabe));// cafebabe
结果为什么不是 0x1cafebabe？该程序执行的加法是一个混合类型的计算：左操作数是 long 型，而右操作数是 int 类型。为了执行该计算，Java 将 int 类型的数值用拓宽原生类型转换提升为 long 类型，然后对两个 long 类型数值相加。因为 int 是有符号的整数类型，所以这个转换执行的是符号扩展。
这个加法的右操作数 0xcafebabe 为 32 位，将被提升为 long 类型的数值 0xffffffffcafebabeL，之后这个数值加上了左操作 0x100000000L。当视为 int 类型时，经过符号扩展之后的右操作数的高 32 位是-1，而左操作数的第 32 位是 1，两个数值相加得到了 0：
 0x 0xffffffffcafebabeL
+0x 0000000100000000L

0x 00000000cafebabeL

如果要得到正确的结果 0x1cafebabe，则需在第二个操作数组后加上“L”明确看作是正的 long 型即可，此时相加时拓展符号位就为 0：

System.*out*.println(Long.*toHexString*(0x100000000L + 0xcafebabeL));// 1cafebabe

5. 窄数字类型提升至宽类型时使用符号位扩展还是零扩展

*System.out.prin*tln((**int**)(**char**)(**byte**)-1);// 65535

结果为什么是 65535 而不是-1？

窄的整型转换成较宽的整型时符号扩展规则：如果最初的数值类型是有符号的，那么就执行符号扩展（即如果符号位为 1，则扩展为 1，如果为零，则扩展为 0）；如果它是 char，那么不管它将要被提升成什么类型，都执行零扩展。

了解上面的规则后，我们再来看看迷题：因为 byte 是有符号的类型，所以在将 byte 数值-1（二进制为：11111111）提升到 char 时，会发生符号位扩展，又符号位为 1，所以就补 8 个 1，最后为 16 个 1；然后从 char 到 int 的提升时，由于是 char 型提升到其他类型，所以采用零扩展而不是符号扩展，结果 int 数值就成了 65535。

如果将一个 char 数值 c 转型为一个宽度更宽的类型时，只是以零来扩展，但如果清晰表达以零扩展的意图，则可以考虑使用一个位掩码：

int i = c & 0xffff;//实质上等同于：int i = c ;

如果将一个 char 数值 c 转型为一个宽度更宽的整型，并且希望有符号扩展，那么就先将 char 转型为一个 short，它与 char 上个具有同样的宽度，但是它是有符号的：

int i = (short)c;

如果将一个 byte 数值 b 转型为一个 char，并且不希望有符号扩展，那么必须使用一个位掩码来限制它：

char c = (char)(b & 0xff);// char c = (char) b;为有符号扩展

6. ((byte)0x90 == 0x90)?

答案是不等的，尽管外表看起来是成立的，但是它却等于 false。为了比较 byte 数值(byte)0x90 和 int 数值 0x90，Java 通过拓宽原生类型将 byte 提升为 int，然后比较这两个 int 数值。因为 byte 是一个有符号类型，所以这个转换执行的是符号扩展，将负的 byte 数值提升为了在数字上相等的 int 值（10010000→11111111111111111111111 10010000）。在本例中，该转换将(byte)0x90 提升为 int 数值-112，它不等于 int 数值的 0x90，即+144。

解决办法：使用一个屏蔽码来消除符号扩展的影响，从而将 byte 转型为 int。

((byte)0x90 & 0xff)== 0x90

7. 三元表达式（?:）

```
char x = 'X';
int i = 0;
System.out.println(true ? x : 0);// X
System.out.println(false ? i : x);// 88
```

条件表达式结果类型的规则：

（1） 如果第二个和第三个操作数具有相同的类型，那么它就是条件表达式的类型。

（2） 如果一个操作的类型是 T，T 表示 byte、short 或 char，而另一个操作数是一个 int 类型的“字面常量”，并且它的值可以用类型 T 表示，那条件表达式的类型就是 T。

（3） 否则，将对操作数类型进行提升，而条件表达式的类型就是第二个和第三个操作被提升之后的类型。

现来使用以上规则解上面的迷题，第一个表达式符合第二条规则：一个操作数的类型是 char，另一个的类型是字面常量为 0 的 int 型，但 0 可以表示成 char，所以最终返回类型以 char 类型为准；第二个表达式符合第三条规则：因为 i 为 int 型变量，而 x 又为 char 型变量，所以会先将 x 提升至 int 型，所以最后的结果类型为 int 型，但如果将 i 定义成 final 时，则返回结果类型为 char，则此时符合第二条规则，因为 final 类型的变量在编译时就使用“字面常量 0”来替换三元表达式了：

```
final int i = 0;
System.out.println(false ? i : x);// X
```

在 JDK1.4 版本或之前，条件操作符 ?: 中，当第二个和延续三个操作数是引用类型时，条件操作符要求它们其中一个必须是另一个的子类型，那怕它们有同一个父类也不行：

```
public class T {
    public static void main(String[] args) {
        System.out.println(f());
    }
    public static T f() {
        // !!1.4 不能编译，但 1.5 可以
        // !!return true?new T1():new T2();
        return true ? (T) new T1() : new T2();// T1
    }
}

class T1 extends T {
    public String toString() {
        return "T1";
    }
}
```

```
class T2 extends T {
    public String toString() {
        return "T2";
    }
}
```

在 5.0 或以上版本中，条件操作符在延续二个和第三个操作数是引用类型时总是合法的。其结果类型是这两种类型的最小公共超类。公共超类总是存在的，因为 Object 是每一个对象类型的超类型，上面的最小公共超类是 T，所以能编译。

8. +=复合赋值问题

x+=i 与 x=x+i 等效吗，许多程序员都会认为第一个表达式 x+=i 只是第二个表达式 x=x+i 的简写方式，但这并不准确。

Java 语言规范中提到：复合赋值 E1 op= E2 等价于简单赋值 E1 = (T)((E1) op (E2))，其中 T 是 E1 的类型。

复合赋值表达式自动地将所执行计算的结果转型为其左侧变量的类型。如果结果的类型与该变量的类型相同，那么这个转型不会造成任何影响，然而，如果结果的类型比该变量的类型要宽，那么复合赋值操作符将悄悄地执行一个窄化原生类型转换，这样就会导致结果不正确：

```
short x=0;
int i = 123456;
x +=i;
System.out.println(x);//-7616
```

使用简单的赋值方式就不会有这样的问题了，因为宽类型不能自动转换成窄的类型，编译器会报错，这时我们就会注意到错误：x = x + i;//编译通不过

请不要将复合赋值操作符作用于 byte、short 或 char 类型的变量；在将复合赋值操作符作用于 int 类型的变量时，要确保表达式右侧不是 long、float 或 double 类型；在将复合赋值操作符作用于 float 类型的变量时，要确保表达式右侧不是 double 类型。其实一句：不要将让左侧的类型窄于右侧的数字类型。

总之，不要在 short、byte 或 char 类型的变量之上使用复合赋值操作符，因为这一过程会伴随着计算前类型的提升与计算后结果的截断，导致最后的计算结果不正确。

9. i =++i;与i=i++;的区别

```
int i = 0;
i = i++;
System.out.println(i);
```

上面的程序会输出什么？大部分会说是 1，是也，非也。运行时正确结果为 0。

i=++i;相当于以下二个语句（编译时出现警告，与 i=i;警告相同）：
i=i+1;
i=i;

i = i++;相当于以下三个语句：
int tmp = i;
i = i + 1;
i = tmp;

下面看看下面程序片段：
int i = 0, j = 0, y = 0;
i++;//相当于：i=i+1;
System.*out*.println("i=" + i);// i=1
++i;//相当于：i=i+1;
System.*out*.println("i=" + i);// i=2
i = i++;//相当于：int tmp=i;i=i+1;i=tmp;
System.*out*.println("i=" + i);// i=2
i = ++i;//编译时出现警告，与 i=i;警告相同。相当于：i=i+1;i=i;
System.*out*.println("i=" + i);// i=3
j = i++;//相当于：int tmp=i;i=i+1;j=tmp;
System.*out*.println("j=" + j);// j=3
System.*out*.println("i=" + i);// i=4
y = ++i;//相当于：i=i+1;y=i;
System.*out*.println("y=" + y);// y=5
System.*out*.println("i=" + i);// i=5

10.Integer.*MAX_VALUE* + 1=?

System.*out*.println(Integer.*MAX_VALUE* + 1);
上面的程序输出多少？2147483647+1=2147483648？答案为-2147483648。

查看源码 Integer.*MAX_VALUE* 为 MAX_VALUE = 0x7fffffff；所以加 1 后为 0x80000000，又 0x80000000 为整型字面常量，满了 32 位，且最位为 1，所以字面上等于 -0，但又由于 -0 就是等于 0，所以-0 这个编码就规定为最小的负数，32 位的最小负数就是-2147483648。

11.-1<<32=?、-1<<65=?

如果左操作数是 int（如果是 byte、short、char 型时会提升至 int 型再进行位操作）型，移位操作符只使用其右操作数的低 5 位作为移位长度（也就是将右操作数除以 32 取余）；如果左操作数是 long 型，移位操作符只使用其右操作数的低 6 位作为移位长度（也就是将右操作数除以 64 取余）；

再看看下面程序片段就会知道结果：

System.*out*.println(-1 << 31);// -2147483648 向左移 31%32=31 位
System.*out*.println(-1 << 32);// -1 向左移 32%32=0 位
System.*out*.println(-1 << 33);// -2 向左移 33%32=1 位
System.*out*.println(-1 << 1);// -2 向左移 1%32=1 位

System.*out*.println(-1L << 63);// -9223372036854775808 向左移 63%64=63 位
System.*out*.println(-1L << 64);// -1 向左移 64%64=0 位
System.*out*.println(-1L << 65);// -2 向左移 65%64=1 位
System.*out*.println(-1L << 1);// -2 向左移 1%64=1 位

byte b = -1;// byte 型在位操作前类型提升至 int
System.*out*.println(b << 31);// -2147483648 向左移 31%32=31 位
System.*out*.println(b << 63);// -2147483648 向左移 63%32=31 位

short s = -1;// short 型在位操作前类型提升至 int
System.*out*.println(s << 31);// -2147483648 向左移 31%32=31 位
System.*out*.println(s << 63);// -2147483648 向左移 63%32=31 位

char c = 1;// char 型在位操作前类型提升至 int
System.*out*.println(c << 31);// -2147483648 向左移 31%32=31 位
System.*out*.println(c << 63);// -2147483648 向左移 63%32=31 位

12.一个数永远不会等于它自己加 1 吗？i==i+1

一个数永远不会等于它自己加 1，对吗？如果数字是整型，则对；如果这个数字是无穷大或都是浮点型足够大（如 1.0e40），等式就可能成立了。

Java 强制要求使用 IEEE 754 浮点数算术运算，它可以让你用一个 double 或 float 来表示无穷大。

浮点型分为 double 型、float 型。

无穷分为正无穷与负无穷。

无穷大加 1 还是无穷大。

一个浮点数值越大，它和其后继数值之间的间隔就越大。

对一个足够大的浮点数加 1 不会改变它的值，因为 1 不足以“填补它与其后者之间的空隙”。

浮点数操作返回的是最接近其精确数学结果的浮点数值。

一旦毗邻的浮点数值之间的距离大于 2，那么对其中的一个浮点数值加 1 将不会产生任何效果，因为其结果没有达到两个数值之间的一半。对于 float 类型，加 1 不会产生任何效果的

最小数是 2^25，即 33554432；而对于 double 类型，最小数是 2^54，大约是 1.8*10^16。

33554432F 转二进制过程：
33554432 的二进制为：10000000000000000000000000，将该二进制化成规范的小数二进制，即小数从右向左移 25 位 1.00000000000000000000000000，化成浮点数二进制 0,25+127, 00000000000000000000000 00（丢弃最后两位），即 0, 10011000, 00000000000000000000000，最后的结果为 1.00000000000000000000000*2^25

毗邻的浮点数值之间的距离被称为一个 ulp，它是最小单位（unit in the last place）的首字母缩写。在 5.0 版本中，引入了 Math.ulp 方法来计算 float 或 double 数值的 ulp。

二进制浮点算术只是对实际算术的一种近似。

```
// 注，整型数不能被 0 除，即(int)XX/0 运行时抛异常
double i = 1.0 / 0.0;// 正无穷大
double j = -1.0 / 0.0;// 负无穷大
// Double.POSITIVE_INFINITY 定义为：POSITIVE_INFINITY = 1.0 / 0.0;
System.out.println(i + " " + (i == Double.POSITIVE_INFINITY));//Infinity true
// Double.NEGATIVE_INFINITY 定义为：NEGATIVE_INFINITY = -1.0 / 0.0;
System.out.println(j + " " + (j == Double.NEGATIVE_INFINITY));//-Infinity true
System.out.println(i == (i + 1));// true
System.out.println(0.1f == 0.1);// false
float f = 33554432;
System.out.println(f + " " + (f==(f+1)));//3.3554432E7 true
```

13. 自己不等于自己吗？ i!=i

NaN（Not a Number）不等于任何数，包括它自身在内。

double i = 0.0/0.0;可表示 NaN。

float 和 double 类型都有一个特殊的 NaN 值，Double.NaN、Float.NaN 表示 NaN。

如果一个表达式中产生了 NaN，则结果为 NaN。

```
System.out.println(0.0 / 0.0);// NaN
System.out.println(Double.NaN + " " + (Double.NaN == (0.0 / 0.0)));//NaN false
```

14. 自动拆箱

```
// 为了兼容以前版本，1.5 不會自動拆箱
System.out.println(new Integer(0) == new Integer(0));// false
```

```
// 1.4 编译非法，1.5 会自动拆箱
System.out.println(new Integer(0) == 0);// true
```

15. 为什么 -0x00000000==0x00000000 、 -0x80000000==0x80000000

为了取一个整数类型的负值，要对其每一位取反（如果是对某个十六进制形式整数求负，如：-0x00000000 则直接对这个十六进制数进行各位取反操作——但不包括前面的负号；如果是对某个十进制求负，如-0，则需先求其绝对值的十六进制的原码后，再各位取反），然后再加 1。
注：如果是对某个十进制数求负，如-1（0xffffffff），实质上按照平时求一个负数补码的方式来处理也是一样的，求某个负数的补码规则为：先求这个数绝对值的原码，然后从该二进制的右边开始向左找第一个为 1 的位置，最后将这个 1 前的各位取反（包括最高位符号位，即最高位 0 取反后为 1），其他位不变，最终所得的二进制就为这个负数的补码，也就是最终在内存中负数所表示的形式。不过在找这个第一个为 1 时可能找不到或在最高位，比如-0，其绝对值为 0（0x00000000）；也有可能最高位为 1，比如-2147483648，其绝对值为 2147483648（0x80000000），如果遇到绝对值的原码为 0x00000000 或 0x80000000 的情况下则不变，即为绝对值的原码本身。

-0x00000000 的运算过程：对 0x00000000 先取反得到 0xffffffff，再加 1，-0x00000000 的最后结果就为 0xffffffff+1，其最后的结果还是 0x00000000，所以-0x00000000 == 0x00000000。前面是对 0x00000000 求负的过程，如果是对 0 求负呢?先求 0 的十六进制形式 0x00000000，再按前面的过程来即可。或者根据前面规则对 0x00000000 求负不变，即最后结果还是 0x00000000。

-0x80000000 的运算过程：对 0x80000000 先取反得到 0x7fffffff，再加 1，-0x80000000 的最后结果就为 0x7fffffff+1，其最后的结果还是 0x80000000，即-0x80000000 == 0x80000000。前面是对 0x80000000 求负的过程，如果是对 2147483648 求负呢?先求 2147483648 的十六进制形式 0x80000000，再按前面的过程来即可。或者根据前面规则对 0x80000000 求负不变，即最后结果还是 0x80000000。

-0x00000001 的运算过程，实质上就是求-1 的补码过程，即对其绝对值的十六进制 0x00000001 求补码，即为 0xffffffff，即-1 的补码为 0xffffffff。

```
System.out.println(Integer.MIN_VALUE == -Integer.MIN_VALUE);// true
/*
 *  0x80000000 取反得 0x7fffffff，再加 1 得 0x80000000，因为负数是
 *  以补码形式存储于内存中的，所以推导出结果原码为：0x80000000，
 *  即为-0，又因为-0 是等于 0 的，所以不需要-0 这个编码位，那就多了
 *  一个 0x80000000 编码位了，所以最后就规定 0x80000000 为最小负数
 */
System.out.println(-0x80000000);// -2147483648
/*
```

```
 *  0x7fffffff 取反得 0x80000000，再加 1 得 0x80000001，因为负数是
 *  以补码形式存储于内存中的，所以推导出结果原码为：0xffffffff，
 *  第一位为符号位，所以最后的结果就为 -0x7fffffff = -2147483647
 */
System.out.println(-0x7fffffff);// -2147483647
```

另外，还发现有趣现象：最大整数加 1 后会等于最小整数：

```
// MAX_VALUE = 0x7fffffff; MIN_VALUE = 0x80000000;
System.out.println((Integer.MAX_VALUE + 1) == Integer.MIN_VALUE);// true
// MIN_VALUE = 0x8000000000000000L; MIN_VALUE = 0x8000000000000000L;
System.out.println((Long.MAX_VALUE + 1) == Long.MIN_VALUE);// true
```

当然，-Byte. *MIN_VALUE==Byte.MIN_VALUE*、-Short.*MIN_VALUE*== Short.*MIN_VALUE*、-Long.*MIN_VALUE*== Long.*MIN_VALUE*，也是成立的。

16.Math.abs结果一定为非负数吗?

```
System.out.println(Math.abs(Integer.MIN_VALUE));// -2147483648
```

上面的程序不会输出 2147483648，而是-2147483648，为什么？

其实我们看一下 Math.abs 源码就知道为什么了，源码：(a < 0) ? -a : a;，结合上面那个迷题，我们就发现-Integer.MIN_VALUE= Integer.MIN_VALUE，所以上面的答案就是最小整数自己。

另外我们也可以从 API 文档看到对 Math.abs() 方法的解释：如果参数等于 Integer.MIN_VALUE 的值（即能够表示的最小负 int 值），则结果与该值相同且为负。

所以 Math.abs 不能保证一定会返回非负结果。

当然，Long.*MIN_VALUE* 也是这样的。

17.不要使用基于减法的比较器

```
Comparator<Integer> c = new Comparator<Integer>() {
    public int compare(Integer i1, Integer i2) {
        return i1 - i2;//  升序
    }
};
List<Integer> l = new ArrayList<Integer>();
l.add(new Integer(-2000000000));
l.add(new Integer(2000000000));
Collections.sort(l, c);
```

System.*out*.println(l);// [2000000000, -2000000000]
上面程序的比较器是升序，结果却不是这样，比较时出现了什么问题？

先看看下面程序片断：

```
int x = -2000000000;
int y = 2000000000;
/*
 * -2000000000 即 -(01110111001101011001010000000000)
 * 的补码为：                    10001000110010100110110000000000
 *
 * 计算过程使用竖式表示：
 * 10001000110010100110110000000000
 * 10001000110010100110110000000000
 * --------------------------------
 * 00010001100101001101100000000000
 *
 * 计算结果溢出，结果为 294967296
 */
System.out.println(x - y);// 294967296
```

所以不要使用减法的比较器，除非能确保要比较的数值之间的距离永远不会大于 Intger. MAX_VALUE。

基于整型的比较器的实现一般使用如下的方式来比较：

```
public int compare(Integer i1, Integer i2) {
    return (i1 < i2 ? -1 : (i1 == i2 ? 0 : 1));
}
```

18. int i=-2147483648 与 int i=-(2147483648)?

int i=-(2147483648);
编译通不过！为什么

int 字面常量 2147483638 只能作为一元负操作符的操作数来使用。

类似的还有最大 long：
long i=–(9223372036854775808L);

字符串

19.char类型相加

System.*out*.println('a' + 'A');//162
上面的结果不是 aA ，而是 162。
当且仅当+操作符的操作数中至少有一个是 String 类型时，才会执行字符串连接操作；否则，执行加法。如果要连接的数值没有一个是字符串类型的，那么你可以有几种选择：预置一个空字符串（"" + 'a' + 'A'）；将第一个数值用 String.valueOf()显示地转换成一个字符串（String.*valueOf*('a') + 'A'）；使用一个字符串缓冲区（sb.append('a');sb.append('A');）；或者如果使用的是 JDK5.0，可以用 printf（System.out.printf("%c%c",'a','A')）；

20.程序中的Unicode转义字符

//\u0022 是双引号的Unicode编码表示
System.*out*.println("a\u0022.length() + \u0022b".length());// 2

Unicode 编码表示的字符是在编译期间就转换成了普通字符，它与普通转义字符（如：\"）是不一样的，它们是在程序被解析为各种符号之后才处理转义字符序列。

21.注释中的Unicode转义字符

如果程序中含有以下的注释：// d:\a\b\util ，程序将不能编译通过，原因是\u 后面跟的不是四个十六进制数字，但编译器在编译时却要把\u 开头的字符的字符看作是 Unicode 编码表示的字符。

所以要注意：注释中也是支持 Unicode 转义字符的。

另外一个问题是不能在注释行的中间含有 \u000A 表示换行的 Unicode 字符，因为这样在编译时读到 \u000A 时，表示行结束，那么后面的字符就会当作程序代码而不在是注释了。

22.Windows与Linux上的行结束标示符

```
String line = (String)System.getProperties().get("line.separator");
for(int i =0; i < line.length();i++){
      System.out.println((int)line.charAt(i));
}
```

在 Windows 上运行结果：

```
13
10
```

在Linux上运行的结果：
10

在 Windows 平台上，行分隔符是由回车（\r）和紧其后的换行（\n）组成，但在 Unix 平台上通常使用单独的换行（\n）表示。

23.输出 0-255 之间的ISO8859-1 符

```
byte bts[] = new byte[256];
for (int i = 0; i < 256; i++) {
    bts[i] = (byte) i;
}
// String str = new String(bts,"ISO8859-1");//正确的做法
String str = new String(bts);//使用操作系统默认编码方式编码（XP GBK）
for (int i = 0, n = str.length(); i < n; i++) {
    System.out.print((int) str.charAt(i) + " ");
}
```

上面不会输出 0-255 之间的数字串，正确的方式要使用 new String(bts," ISO8859-1") 方式来解码。

ISO8859-1 是唯一能够让该程序按顺序打印从 0 到 255 的整数的缺少字符集，这也是唯一在字符和字节之间一对一的映射字符集。

通过 java 获取操作系统的默认编码方式：
System.*getProperty*("file.encoding");//jdk1.4 或之前版本
java.nio.charset.Charset.*defaultCharset*();//jdk1.5 或之后版本

24.String的replace()与replaceAll()

```
System.out.println(".".replaceAll(".class", "\\$"));
```

上面程序将 . 替换成 \$，但运行时报异常，主要原 replaceAll 的第二参数有两个字符（\ $）是特殊字符，具有特殊意思（\用来转移 \ 与 $，$后面接数字表示反向引用）。另外，replaceAll 的第一参数是正则表达式，所以要注意特殊字符，正确的作法有以下三种：

```
System.out.println(".class".replaceAll("\\.", "\\\\\\$"));
System.out.println(".class".replaceAll("\\Q.\\E", "\\\\\\$"));
System.out.println(".class".replaceAll(Pattern.quote("."), Matcher.quoteReplacement("\\$")));
```

API 对\、\Q 与\E 的解释：
\　引用（转义）下一个字符
\Q 引用所有字符，直到 \E
\E 结束从 \Q 开始的引用

JDK5.0 新增了一些解决此问题的新方法：

java.util.regex.Pattern.quote(String s)：使用\Q 与\E 将参数引起来，这些被引用的字符串就是一般的字符，哪怕含有正则式特殊字符。
java.util.regex.Matcher.quoteReplacement(String s)：将\与$转换成能应用于 replaceAll 第二个参数的字符串，即可作为替换内容。

String 的 replace(**char** oldChar, **char** newChar)方法却不使用正则式，但它们只支持字符，而不是字符串，使用起来受限制：
System.*out*.println(".".replace('.','\\'));//能将 . 替换成 \
System.*out*.println(".".replace('.','$')); //能将 . 替换成 $

25.一段程序的三个Bug

```
Random rnd = new Random();
StringBuffer word = null;
switch (rnd.nextInt(2)) {
case 1:
    word = new StringBuffer('P');
case 2:
    word = new StringBuffer('G');
default:
    word = new StringBuffer('M');
}
word.append('a');
word.append('i');
word.append('n');
System.out.println(word);
```

上面的程序目的是等概率的打印 Pain、Gain、Main 三个单词，但多次运行程序却发现永远只会打印 ain，这是为什么？

第一个问题在于：rnd.nextInt(2)只会返回 0、1 两个数字，所以上面只会走 case 1: 的分支语句，case 2: 按理是永远不会走的。

第二个问题在于：如果 case 语句不以 break 结束时，则一直会往向运行，即直到执行到 break 的 case 语句止，所以上面的的语句每次都会执行 default 分支语句。

第三个问题在于：StringBuffer 的构造函数有两种可接受参数的，一个是 StringBuffer(int capacity)、另一个是 StringBuffer(String str)，上面用的是 StringBuffer(char)构造函数，实质上运行时将字符型转换成了 int 型，这样将字符当作 StringBuffer 的初始容量了，而不是字符本身。

以下是修改后的程序片段：

```
Random rnd = new Random();
StringBuffer word = null;
switch (rnd.nextInt(3)) {
```

```
case 1:
    word = new StringBuffer("P");
    break;
case 2:
    word = new StringBuffer("G");
    break;
default:
    word = new StringBuffer("M");
    break;// 可以不要

}
word.append('a');
word.append('i');
word.append('n');
System.out.println(word);
```

异常

26.finally与中断

```
//该方法返回 false
static boolean f() {
    try {
        return true;
    } finally {
        return false;
    }
}
```

不要用 return、break、continue 或 throw 来退出 finally 语句块，并且千万不要允许受检查的异常传播到 finally 语句块之外。也就是说不要在 finally 块内终止程序，而是执行完 finally 块后，要将控制权移交给 try 块，由 try 最终决定怎样结束方法的调用。

对于任何在 finally 语句块中可能抛出的受检查异常都要进行处理，而不是任其传播，下面流拷贝程序在关闭流时没有防止异常的传播，这会有问题：

```
static void copy(String src, String dest) throws IOException {
    InputStream in = null;
    OutputStream out = null;
    try {
        in = new FileInputStream(src);
        out = new FileOutputStream(dest);
        byte[] buf = new byte[1024];
        int n;
```

```
            while ((n = in.read(buf)) >= 0) {
                out.write(buf, 0, n);
            }
        } finally{
            //这里应该使用 try-catch 将每个 close 包装起来
            if(in != null){in.close();}
            if(in != null){out.close();}
        }
}
```

catch 块中的 return 语句是不会阻止 finally 块执行的，那么 catch 块中的 continue 和 break 能否阻止？答案是不会的，与 return 一样，finally 语句块是在循环被跳过（continue）和中断（break）之前被执行的：

```
int i = 0;
System.out.println("--continue--");
while (i++ <= 1) {
    try {
        System.out.println("i=" + i);
        continue;
    } catch (Exception e) {
    } finally {
        System.out.println("finally");
    }
}
System.out.println("--break--");
while (i++ <= 3) {
    try {
        System.out.println("i=" + i);
        break;
    } catch (Exception e) {
    } finally {
        System.out.println("finally");
    }
}
```

27.catch捕获异常规则

捕获 RuntimeException、Exception 或 Throwable 的 catch 语句是合法，不管 try 块里是否抛出了这三个异常。但如果 try 块没有抛出或不可能抛出检测性异常，则 catch 不能捕获这些异常，如 IOException 异常：

```
public class Test {
    public static void main(String[] args) {
        try{
            //...
```

```
        }catch (Exception e) {

        }catch (Throwable e) {

        }

        /* !! 编译出错
            try{
                //...
            }catch (IOException e) {

            }
         */
    }
}
```

28.重写时方法异常范围

重写或实现时不能扩大异常的范围，如果是多继承，则异常取所有父类方法异常的交集或不抛出异常：

```
interface I1 {
    void f() throws Exception;
}

interface I2 {
    void f() throws IOException;
}

interface I3 extends I1, I2 {}

class Imp implements I3 {
    // 不能编译通过，多继承时只能取父类方法异常交集，这样就不会扩大异常范围
    // !! void f () throws Exception;
    // void f();// 能编译通过
    // 能编译通过，Exception 与 IOException 的交集为 IOException
    public void f() throws IOException {
    }
}
```

29.静态与非静态final常量不能在catch块中初始化

静态与非静态块中如果抛出了异常，则一定要使用 try-catch 块来捕获。

```
public class Test {
    static final int i;
    static {
        try {
            i = f();
        } catch (RuntimeException e) {
            i = 1;
        }
    }

    static int f() {
        throw new RuntimeException();
    }
}
```

上面的程序编译不能通过。表面上是可以的，因为 i 第一次初始化时可能抛出异常，所以抛异常时可以在 catch 块中初始化，最终还是只初始化一次，这正是空 final 所要求的，但为什么编译器不知道这些呢？

要确定一个程序是否不止一次地对一个空 final 进行赋值是很困难的问题。语言规范在这一点上采用了保守的方式。

30. System.*exit*()与finally

```
try {
    System.out.println("Hello world");
    System.exit(0);
    // 或者使用 Runtime 退出系统
    // Runtime.getRuntime().exit(0);
} finally {
    System.out.println("Goodbyte world");
}
```

上面的程序会打印出"Goodbyte world"吗？不会。

System.exit 将立即停止所有的程序线程，它并不会使 finally 语句块得到调用，但是它在停止 VM 之前会执行关闭挂钩操作（这此挂钩操作是注册到 Runtime.addShutdownHook 上的线程），这对于释放 VM 之外的资源很有帮助。使用挂钩程序修改上面程序：

```
System.out.println("Hello world");
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() {
        System.out.println("Goodbyte world");
    }
});
System.exit(0);
```

对象回收时，使用 VM 调用对象的 finalize()方法有两种：
System.runFinalization()：该方法让虚拟机也只是尽最大努力去完成所有未执行的 finalize()终止方法，但不一定会执行。
System.runFinalizersOnExit(true)：该方法一定会回收，但不安全，已被废弃。因为它可能对正在使用的对象调用终结方法，而其他线程同时正在操作这些对象，从而导致不正确的行为或死锁。

为了加快垃圾回收，使用 System.gc()，但不一定马上执行加收动作，由虚拟机决定，实质上是调用 Runtime.getRuntime().gc()。

System 的很多方法都是调用 Runtime 类的相关方法来实现的。

31.递归构造

```
public class S    {
    private S instance = new S();
    public S() {}
}
```

如果在程序外面构造该类的实例，则会抛出 java.lang.StackOverflowError 错误。其原因是实例变量的初始化操作将先于构造器的程序体而运行。

32.构造器中的异常

如果父类构造器抛出了检测异常，则子类也只能抛出，而不能采用 try-catch 来捕获：

```
public class P {
    public P() throws Exception {}
}

class S extends P {
    public S() throws Exception {
        try {
            // 不能在 try 块中明确调用父类构造器，因为构造的
            // 明确调用只能放在第一行
            // !! super();
        //try-catch 不能捕获到父类构造器所抛出的异常，子类只能抛出
        } catch (Exception e) {
        }
    }
}
```

如果初使化实例属性时抛出了异常，则构造器只能抛出异常，在构造器中捕获不起作用：

```
public class A   {
    private String str = String.class.newInstance();
    public A()throws InstantiationException, IllegalAccessException {}
    public A(int i) throws Exception{
        try {

        } catch (Exception e) {

        }
    }
}
```

33.StackOverflowError

Java 虚拟机对栈的深度限制到了某个值，当超过这个值时，VM 就抛出 StackOverflowError。一般 VM 都将栈的深度限制为 1024，即当方法调用方法的层次超过 1024 时就会产生 StackOverflowError。

类

34.参数兼容的方法重载

```
public class Confusing {
    private Confusing(Object o) {
        System.out.println("Object");
    }
    private Confusing(double[] dArr) {
        System.out.println("double array");
    }
    public static void main(String[] args) {
        new Confusing(null);
    }
}
```

上面的程序打印的是“double array”，为什么？

null 可代表任何非基本类型对象。

Java 的重载解析过程是分两阶段运行的。第一阶段选取所有可获得并且可应用的方法或构造器。第二阶段在第一阶段选取的方法或构造器中选取最精确的一个。如果一个方法或构造器可以接受传递给另一个方法或构造器的任何参数，那么我们就说第一个方法比第二个方法缺乏精确性，调用时就会选取第二个方法。

使用上面的规则来解释该程序：构造器 Confusing(Object o)可以接受任何传递 Confusing(**double**[] dArr)的参数，因此 Confusing(Object o)相对缺乏精确性，所以 Confusing(**null**)会调用 Confusing(**double**[] dArr)构造器。

如果想强制要求编译器选择一个自己想要的重载版本，需要将实参强制转型为所需要的构造器或方法的参数类型：如这里要调用 Confusing(Object o)本版，则这样调用：Confusing((Object)null)。

如果你确实进行了重载，那么请确保所有的重载版本所接受的参数类型都互不兼容，这样，任何两个重载版本都不会同时是可应用的。

35.静态方法不具有多态特性

```
class A1 {
    public static void f() {
        System.out.println("A1.f()");
    }
}
class A2 extends A1 {
    public static void f() {
        System.out.println("A2.f()");
    }
}
class T {
    public static void main(String[] args) {
        A1 a1 = new A1();
        A1 a2 = new A2();
        // 静态方法不具有多态效果，它是根据引用声明类型来调用
        a1.f();// A1.f()
        a2.f();// A1.f()
    }
}
```

对静态方法的调用不存在任何动态的分派机制。当一个程序调用了一个静态方法时，要被调用的方法都是在编译时就被选定的，即调用哪个方法是根据该引用被声明的类型决定的。上面程序中 a1 与 a2 引用的类型都是 A1 类型，所以调用的是 A1 中的 f()方法。

36.属性只能被隐藏

```
class P {
    public String name = "P";
}
```

```
class S extends P {
    // 隐藏父类的 name 域，而不像方法属于重写
    private String name = "S";
}

public class Test {
    public static void main(String[] args) {
        // !! S.name is not visible
        // !! System.out.println(new S().name);
        // 属性不能被重写，只是被隐藏，所以不具有多态性为
        System.out.println(((P) new S()).name);// p
    }
}
```

属性的调用与静态方式的调用一样，只与前面引用类型相关，与具体的实例没有任何关系。

当你在声明一个域、一个静态方法或一个嵌套类型时，如果其名与基类中相对应的某个可访问的域、方法或类型相同时，就会发生隐藏。

37.属性对嵌套类的遮掩

```
class X {
    static class Y {
        static String Z = "Black";
    }
    static C Y = new C();
}
class C {
    String Z = "White";
}
public class T {
    public static void main(String[] args) {
        System.out.println(X.Y.Z);// White
        System.out.println(((X.Y) null).Z);// Black
    }
}
```

当一个变量和一个类型具有相同的名字，并且它们位于相同的作用域时，变量名具有优先权。变量名将遮掩类型名。相似地，变量名和类型名可以遮掩包名。

38.不能重写不同包中的defualt访问权限方法

```
package click;
```

```
public class P {
    public void f() {
        //因为子类没有重写该方法，所以调用的还是父类中的方法
        prt();
    }
    void prt() {
        System.out.println("P");
    }
}

package hack;
import click.P;
public class T {
    private static class S extends P {
        // 这里没有重写父类的方法，因为父类方法不可见
        void prt() {
            System.out.println("S");
        }
    }
    public static void main(String[] args) {
        new S().f();// P
    }
}
```

一个包内私有（default）的方法不能被位于另一个包中的某个方法直接重写。

39.重写、隐藏、重载、遮蔽、遮掩

重写：一个实例方法可以重写在其超类中可访问到的具有相同签名的所有实例方法，从而能动态分派，换句话说，VM 将基于实例的运行期类型来选择要调用的重写方法。重写是面向对象编程技术的基础。

```
public class P{
    public void f(){}
}
class S extends P{
    public void f(){}//重写
}
```

重写时异常要求：

- 如果父类方法抛出的是捕获型异常，则子类也只能抛出同类的捕获型异常或其子类，或不抛出。
- 父类抛出捕获型异常，子类却抛出运行时异常，这是可以，因为抛出运行时就相当于没有抛出任何异常。

- 如果父类抛出的是非捕获型异常，则子类可以抛出任意的非捕获型异常，没有扩大异常范围这一问题。
- 如果父类抛出的是非捕获异常，子类也可以不用抛出，这与父类为捕获型异常是一样的。
- 如果父类抛出的是非捕获异常，子类就不能抛出任何捕获型异常，因为这样会扩大异常的范围。

返回类型的协变：从Java SE5开始子类方法可以返回比它重写的基类方法更具体的类型，但是这在早先的Java版本是不允许——重写时子类的返回类型一定要与基类相同。但要注意的是：子类方法返回类型要是父类方法返回类型的子类，而不能反过来。

方法参数类型协变：如果父子类同名方法的参数类型为父子关系，则为参数类型协变，此时不属于重写，而是方法的重载，以前版本就是这样。

如果父类的方法为 private 时，子类同名的方法的方法名前可以使用任何修饰符来修饰。我们可以随意地添加一个新的私有成员（方法、域、类型），或都是修改和删除一个旧的私有成员，而不需要担心对该类的客户造成任何损害。换而言之，私有成员被包含它们的类完全封装了。

父与子类相同签名方法不能一静一动的，即父类的方法是静态的，而子类不是，或子类是静态的，而父类不是，编译时都不会通过。

父与子相同签名方法都是静态的方法时，方法名前的修饰符与非静态方法重写的规则一样，但不属于重写，因为静态方法根本就不具有多态性。

最后，属于成员也不具有多态特性，相同名的域属于隐藏，而不管域前面的修饰符为什么：

```
class P {
    public static final String str = "P";
}
class S extends P {
    //编译能通过。可以是 final，这里属于隐藏
    public static final String str = "S";
    public static void main(String[] args) {
        System.out.println(S.str);//s
    }
}
```

隐藏：一个域、静态方法或成员类型可以分别隐藏在其超类中可访问到的具有相同名字（对方法而言就是相同的方法签名）的所有域、静态方法或成员类型。隐藏一个成员将阻止其被继承。

```
public class P{
    public static void f(){}
}
class S extends P{
```

```
    //隐藏，不会继承 P.f()
    public static void f(){}
}
```

重载：在某个类中的方法可以重载另一个方法，只要它们具有相同的名字和不同的签名。由调用所指定的重载方法是在编译期选定的。

```
public class T{
    public static void f(int i){}
    public static void f(String str){}//重载
}
```

遮蔽：一个变量、方法或类型可以分别遮蔽在一个闭合的文本范围内的具有相同名字的所有变量、方法或类型。如果一个实体被遮蔽了，那么你用它的简单名是无法引用到它的；根据实体的不同，有时你根本就无法引用到它。

```
public class T {
    private static String str = "feild";
    public static void main(String[] args) {
        String str = "local";//  遮蔽
        System.out.println(str);// local
        //  可以通过适当的方式来指定
        System.out.println(T.str);// feild
    }
}

public class T {
    private final int size;
    //  参数属于方法局部范围类变量，遮蔽了同名成员变量
    public T(int size) {
        //使用适当的引用来指定
        this.size = size;
    }
}
```

遮掩：一个变量可以遮掩具有相同名字的一个类型，只要它们都在同一个范围内：如果这个名字被用于变量与类型都被许可的范围，那么它将引用到变量上。相似地，一个变量或一个类型可以遮掩一个包。遮掩是唯一一种两个名字位于不同的名字空间的名字重用形式，这些名字空间包括：变量、包、方法或类型。如果一个类型或一个包被遮掩了，那么你不能通过其简单名引用到它，除非是在这样一个上下文环境中，即语法只允许在其名字空间中出现一种名字：

```
public class T {
    static String System;
    public static void main(String[] args) {
        // !!不能编译，遮掩  java.lang.System
        // !! System.out.println("Hello");
```

```
        // 可明确指定
        java.lang.System.out.println("Hello");
    }
}
```

40.构造器中静态常量的引用问题

```
class T {
    // 先于静态常量 t 初始化，固可以在构造器中正常使用
    private static final int y = getY();
    /*
     * 严格按照静态常量声明的先后顺来初始化：即 t 初始
     * 化完后，才初始化后面的静态常量 j，所以构造器中
     * 引用后面的静态常量 j 时，会是 0，即内存清零时的值
     */
    public static final T t = new T();
    // 后于静态常量 t 初始化，不能在构造器中正常使用
    private static final int j = getJ();
    private final int i;

    static int getY() {
        return 2;
    }

    static int getJ() {
        return 2;
    }

    // 单例
    private T() {
        i = y - j - 1;
        //为什么 j 不是 2
        System.out.println("y=" + y + " j=" + j);// y=2 j=0
    }

    public int getI() {
        return i;
    }

    public static void main(String[] args) {
        System.out.println(T.t.getI());// 1
        System.out.println(T.j);// 2
    }
}
```

该程序所遇到的问题是由类初始化顺序中的循环而引起的：初始化 t 时需调用构造函数，而调用构造函数前需初始化所有静态成员，此时又包括对 t 的再次初始化。

T 类的初始化是由虚拟机对 main 方法的调用而触发的。首先，其静态域被设置缺省值，其中 y、j 被初始化为 0，而 t 被初始化为 null。接下来，静态域初始器按照其声明的顺序执行赋值动作。第一个静态域是 y，它的值是通过调用 getY 获取的，赋值操作完后结果为 2。第一个初始化完成后，再进行第二个静态域的赋值操作，第二个静态域为 t，它的值是通过调用 T()构造函数来完成的。这个构造器会用二个涉及静态域 y、j 来初始化非静态域 i。通常，读取一个静态域是会引起一个类被初始化，但是我们又已经在初始化 T 类。**JavaVM 规范对递归的初始化尝试会直接被忽略掉**（按理来说在创建出实例前需初始化完所有的静态域后再来创建实例），这样就导致在静态域被初始化之前就调用了构造器，后面的**静态域 j 将得不到正常的初始化前就在构造器中被使用了**，使用时的值为内存分配清零时的，即 0。
当 t 初始化完后，再初始化 j，此时 j 得到的值为 2，但此时对 i 的初始化过程来说已经晚了。

在 final 类型的静态域被初始化之前，存在着读取其值的可能，而此时该静态域包含的还只是其所属类型的缺省值。这是与直觉想违背的，因为我们通常会将 final 类型的域看作是常量，但 final 类型的域只有在其初始化表达式是字面常量表达式时才是真正的常量。

再看看另一程序：

```
class T {
    private final static int i = getJ();
    private final static int j;
    static {
        j = 2;
    }
    static int getJ() {
        return j;
    }
    public static void main(String[] args) {
        System.out.println(T.j);// 2
        /*
         * 因为上面的语句已经初使完 T 类，所以下面语句是
         * 不 会 再引起类的初始化，这里的结果用的是第一
         *  次（ 即上面语句）的初始化结果
         */
        System.out.println(T.i);// 0
    }
}
```

为什么第二个输出是 0 而不是 2 呢？这就是因为 VM 是严格按照你声明的顺序来初始化静态域的，所以前面的引用后面的静态域时，基本类型就是 0，引用类型就会是 null。

所以要记住：静态域，甚至是 final 类型的静态域，可能会在它们被初始化之前，被读走其缺省值。

另，类初始化规则请参考《惰性初始化》一节

41.instanceof与转型

```
System.out.println(null instanceof String);//false
System.out.println(new Object() instanceof String);//false
//编译能通过
System.out.println((Object) new Date() instanceof String);//false
//!!程序不具有实际意义，但编译时不能通过
//!!System.out.println(new Date() instanceof String);
//!!运行时抛 ClassCastException，这个程序没有任何意义，但可以编译
//!!System.out.println((Date) new Object());
```

null 可以表示任何引用类型，但是 instanceof 操作符被定义为在其左操作数为 null 时返回 false。

如果 instanceof 告诉你一个对象引用是某个特定类型的实例，那么你就可以将其转型为该类型，并调用该类型的方法，而不用担心会抛出 ClassCastException 或 NullPointerException 异常。

instanceof 操作符有这样的要求：左操作数要是一个对象的或引用，右操作数是一个引用类型，并且这两个操作数的类型是要父子关系（左是右的子类，或右是左的子类都行），否则编译时就会出错。

42.父类构造器调用已重写的方法

```
public class P {
    private int x, y;
    private String name;

    P(int x, int y) {
        this.x = x;
        this.y = y;
        // 这里实质上是调用子类被重写的方法
        name = makeName();
    }

    protected String makeName() {
        return "[" + x + "," + y + "]";
    }

    public String toString() {
```

```
        return name;
    }

}

class S extends P {
    private String color;

    S(int x, int y, String color) {
        super(x, y);
        this.color = color;
    }

    protected String makeName() {
        return super.makeName() + ":" + color;
    }

    public static void main(String[] args) {
        System.out.println(new S(1, 2, "red"));// [1,2]:null
    }
}
```

在一个构造器调用一个已经被其子类重写了的方法时，可能会出问题：如果子类重写的方法要访问的子类的域还未初始化，因为这种方式被调用的方法总是在实例初始化之前执行。要想避免这个问题，就千万不要在父类构造器中调用已重写的方法。

43.静态域与静态块的初始顺序

```
public class T {
    public static int i = prt();
    public static int y = 1;
    public static int prt() {
        return y;
    }

    public static void main(String[] args) {
        System.out.println(T.i);// 0
    }
}
```

上面的结果不是 1，而是 0，为什么?

类初始化是按照静态域或静态块在源码中出现的顺序去执行这些静态初始器的（即谁先定义，就先初始化谁），上现程序中由于 i 先于 y 声明，所以先初始化 i，但由于 i 初始化时需要由 y 来决定，此时 y 又未初始化，实为初始前的值 0，所以 i 的最后结果为 0。

44.请使用引用类型调用静态方法

```
public class Null {
    public static void greet() {
        System.out.println("Hello world!");
    }

    public static void main(String[] args) {
        ((Null) null).greet();
    }
}
```

上面程序运行时不会打印 NullPointerException 异常，而是输出"Hello world!"，关键原因是：调用静态方法时将忽略前面的调用对象或表达示，只与对象或表达式计算结果的类型有关。

在调用静态方法时，一定要使用类去调用，或是静态导入后直接使用。

45.循环中的不能声明局部变量

```
for (int i = 0; i < 1; i++)
    Object o ; //!!  编译不能通过

for (int i = 0; i < 1; i++)
    Object o = new Object(); //!!  编译不能通过
```

一个本地变量声明看起来像是一条语句，但是从技术上来说不是。

Java 语言规范不允许一个本地变量声明语句作为一条语句在 for、while 或 do 循环中重复执行。

一个本地变量声明作为一条语句只能直接出现在一个语句块中（一个语句块是由一对花 括号以及包含在这对花括号中的语句和声明构成的）：

```
for (int i = 0; i < 1; i++) {
    Object o = new Object(); //  编译 OK
}
```

46.内部类反射

```
public class Outer {
    public class Inner {
        public String toString() {
            return "Hello world";
        }
```

```
    }
    public void getInner() {
        try {
            // 普通方式创建内部类实例
            System.out.println(new Outer().new Inner());// Hello world
            //!! 反射创建内部类，抛异常：java.lang.InstantiationException:Outer$Inner
            System.out.println(Inner.class.newInstance());
        } catch (Exception e) {
        }
    }
    public static void main(String[] args) {
         new Outer().getInner();
    }
}
```

上面因为构造内部类时外部类实例不存在而抛异常。

一个非静态的嵌套类的构造器，在编译的时候会将一个隐藏的参数作为它的第一个参数，这个参数表示它的直接外围实例。如果使用反射创建内部类，则要传递个隐藏参数的唯一方法就是使用 java.lang.reflect.Constructor：

```
Constructor c = Inner.class.getConstructor(Outer.class);//获取带参数的内部类构造函数
System.out.println(c.newInstance(Outer.this));//反射时还需传进外围类
```

应用

47.不可变的引用类型

```
BigInteger total = BigInteger.ZERO;
total.add(new BigInteger("1"));
total.add(new BigInteger("10"));
System.out.println(total);//0
```

上面程序的结果为 11 吗？答案是 0。

BigInteger 实例是不可变的。String、BigDecimal 以及包装类型：Integer、Long、Short、Byte、Character、Boolean、Float 和 Double 也是如此。对这些类型的操作将返回新的实例。

不可变类型更容易设计、实现与作用；它们出错的可能性更小，并且更加安全。

本程序修改如下：

```
BigInteger total = BigInteger.ZERO;
total=total.add(new BigInteger("1"));
total=total.add(new BigInteger("10"));
System.out.println(total);//11
```

48.请同时重写equals()与hashCode()

```
class T {
    private String str;

    T(String str) {
        this.str = str;
    }

    public boolean equals(Object obj) {
        if(!(obj instanceof T)){
            return false;
        }
        T t = (T)obj;
        return t.equals(this.str);
    }

    public static void main(String[] args) {
        Set set = new HashSet();
        set.add(new T("str"));
        System.out.println(set.contains(new T("str")));//false
    }
}
```

上面的程序不会打印 true，而是 false，为什么？

hashCode 约定要求相等的对象要具有相同的散列码。

无论何时，只要你重写了 equals 方法，你就必须同时重写 hashCode 方法。

如果将自定的类型对象放入 HashSet、HashMap、Hashtable、LinkedHashSet、LinkedHashMap 这此散列集合时，一定需要重写 equals 与 hashCode 方法，这样在放入进去之后还能查找出来。如果放入其他非散列类型的集合时，其实只需要重写 equals 就可以了。

本程序解决办法重写 hashCode()方法：

```
public int hashCode() {
    return 37 * this.str.hashCode();
}
```

49.日期设置

```
Calendar c = Calendar.getInstance();
c.set(2010, 12, 31);// 月是从 0 开始的，11 其实表示 12 月
System.out.println(c.get(Calendar.YEAR) + " " + c.get(Calendar.MONTH));
```

```
c = Calendar.getInstance();
c.set(2010, 11, 31);
System.out.println(c.get(Calendar.YEAR) + " " + c.get(Calendar.MONTH));
```

本程序较简单，只需注意月是从 0 开始的就可以了，如果你设置月为 12，则会自动转换为下一年。

50.IdentityHashMap

```
class T {
    private String str;

    T(String str) {
        this.str = str;
    }

    public int hashCode() {
        return 37 * this.str.hashCode();
    }

    public boolean equals(Object obj) {
        return this.str.equals(((T) obj).str);
    }

    public static void put(Map m) {
        m.put("str", "1");
        /*
         * 由于上面程序将 "str" 放入了字符串常量池，
         * 所以 str 是同一个对象，不管是什么样类型的
         * Map，即使使用 IdentityHashMap 都只放入一次
         */
        m.put("str", "2");
        m.put(new T("str"), "3");
        m.put(new T("str"), "4");
    }

    public static void main(String[] args) {
        Map m = new HashMap();
        put(m);
        System.out.println(m.size());// 2
        //IdentityHashMap 比较时使用==替换 equals()方法
        m = new IdentityHashMap();
        put(m);
        System.out.println(m.size());// 3
    }
```

```
}
```

51.静态导入的优先权

```
import static java.util.Arrays.toString;
import java.util.Arrays;
public class T {
    public static void main(String[] args) {
        prt(1, 2, 3);
    }
    static void prt(Object... args) {
        // 自身继承至 Object 类的 toString 的优先级高于静态导入的方法
        //!! System.out.println(toString(args));//不能编译
        System.out.println(Arrays.toString(args));
    }
}
```

本身就属于某个范围的成员在该范围内与静态导入相比具有优先权。

52.PrintStream对输出结果的缓冲

```
public static void main(String[] args) {
    String str = "Hello World";
    for (int i = 0; i < str.length(); i++) {
        System.out.write(str.charAt(i));
    }
}
```

上面的程序没有输出结果。

这里的问题在于 System.out 是带有缓冲的。输出的结果被写入了 System.out 的缓冲区，但是缓冲区从来都没有被刷新。大多数人认为，当有输出产生的时候 System.out 和 System.err 会自动地进制刷新，但这并不完全正确，这两个流都属于 PrintStream 类型，请看 API DOC 描述：一个 PrintStream 被创建为自动刷新，这意味着当一个字节数组（byte[]）被写入、或者某个 println 方法被调用、或者一个换行字符或字节（'\n'）被写入之后，PrintStream 类型的 flush 方法就会被自动调用。

令人奇怪的是，如果这个程序用 print(char)去替代 write(int)，它就会刷新 System.out 并输出结果，这种行为与 print(char)的文档是矛盾的，因为其文档叙述道：“打印一个字符，这个字符将根据平台缺省的字符编码方式翻译成一个或多个字节，并且这些字节将完全按照 write(int)方法的方式输出。”，但这里没有换行符却也自动的刷新了。类似的，如果程序改用 print(String)，它也会对流进行刷新。所以调用 print 方法也是会自动刷新的。

请加入到原博客中：PrintStream 也可以对 OutputStream 进行包装并指定编码方式：

PrintStream(OutputStream out, boolean autoFlush, String encoding)，但实质上也是调用 OutputStreamWriter 来实现的。

System.err 在 eclipse 中输出时是红色的字体。

53.调用操作系统命令时被阻塞问题

```
public static void main(String[] args) throws IOException,
        InterruptedException {
    String command = "java ProcessTest exc";
    if (args.length != 0) {
        for (int i = 0; i < 200; i++) {
            System.out.println(command);
            System.err.println(command);
        }
    } else {
        Process process = Runtime.getRuntime().exec(command);
        int exitValue = process.waitFor();
        System.out.println("exit value = " + exitValue);
    }
}
```

执行 java ProcessTest 发现程序阻塞。

Process 文档描述：由于某些本地平台只提供有限大小的缓冲，所以如果不能迅速地读取子进程的输出流，就有可能会导致子进程的阻塞，甚至是死锁。这恰好就是这里所发生的事情：没有足够的缓冲空间来保存这些输出结果。为了结子进程（Process 线程），父进程（Main 线程）必须排空它的输出流（标准流与错误流都需要排空），即要去缓存中读取结果：

```
static void readResult(final InputStream is) {
    new Thread(new Runnable() {
        public void run() {
            try {
                // 排空缓存内容
                while (is.read() >= 0);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }).start();
}
```

然后在 process.waitFor()之前加上

```
readResult(process.getErrorStream());
readResult(process.getInputStream());
```

即可输出 exit value = 0。

另外，只能根据 process.waitFor 返回的结果来判断操作系统命令执行是否成功（成功：0，失败：1），我们不能根据错误流中是否有内容来判断是否执行成功。

54.实现Serializable的单例问题

```
class Dog implements Serializable{
    public static final Dog INSTANCE = new Dog();
    private Dog(){}
}
```

上面能控制只生成一个单实例吗？

如果对实现了 Serializable 的对象进行序列化后，再反序列化，内中会不只一个实例了，因为反序列化时会重新生成一个对象。

既然 INSTANCE 为静态域，那序列化时返回的对象如果也是 INSTANCE 就可以解决问题了，而打开 API 我们发现 Serializable 接口确实有这样两个特殊的方法描述：

- 将对象写入流时需要指定要使用的替代对象的可序列化类，应使用准确的签名来实现此特殊方法：

ANY-ACCESS-MODIFIER Object writeReplace() throws ObjectStreamException;

此 writeReplace 方法将由序列化调用，前提是如果此方法存在，而且它可以通过被序列化对象的类中定义的一个方法访问。因此，该方法可以拥有私有 (private)、受保护的 (protected) 和包私有 (package-private) 访问。子类对此方法的访问遵循 java 访问规则。

- 在从流中读取类的一个实例时需要指定替代的类应使用的准确签名来实现此特殊方法：

ANY-ACCESS-MODIFIER Object readResolve() throws ObjectStreamException;

此 readResolve 方法遵循与 writeReplace 相同的调用规则和访问规则。

上述两个方法的只要出现，就会履盖以下两个方法（这两个方法本质的意义就是用来替换序列与返序列的对象），虽然会执行它们，但最后得到的结果却是 writeReplace、readResolve 两个方法写入或读出的对象：

- private void writeObject(java.io.ObjectOutputStream out) throws IOException
- private void readObject(java.io.ObjectInputStream in)throws IOException, ClassNotFoundException;

另外，writeObject 与 readObject 需成对实现，而 writeReplace 与 readResolve 则不需要成对出现，一般单独使用。如果同时出现这四个方法，最后写入与读出的结果以 writeReplace 和 readResolve 方法的结果为准。

所以下要解决真真单实例问题，我们如下修正：

```
class Dog implements Serializable {
    public static final Dog INSTANCE = new Dog();
    private Dog() {}
    private Object readResolve() {
        return INSTANCE;
    }
```

```
}

public class SerialDog {
    public static void main(String[] args) throws IOException,
            ClassNotFoundException {
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        new ObjectOutputStream(bos).writeObject(Dog.INSTANCE);
        ByteArrayInputStream bin = new ByteArrayInputStream(bos.toByteArray());
        Dog dog = (Dog) new ObjectInputStream(bin).readObject();
        System.out.println(dog == Dog.INSTANCE);//true
    }
}
```

一个实现了 Serializable 的单例类，必须有一个 readResolve 方法，用以返回它的唯一实例。

55.thread. isInterrupted()与Thread.interrupted()

```
public class SelfInerruption {
    public static void main(String[] args) {
        Thread.currentThread().interrupt();
        if (Thread.interrupted()) {
            // Interruped:false
            System.out.println("Interruped:" + Thread.interrupted());
        } else {
            System.out.println("Not interruped:" + Thread.interrupted());
        }
    }
}
```

上面结果走的是第一个分支，但结果却不是 Interruped:true？

Thread.interrupted()为 Thread 的静态方法，调用它首先会返回当前线程的中断状态（如果当前线程上调用了 interrupt()方法，则返回 true，否则为 false），然后再清除当前线程的中断状态，即将中断状态设置为 false。换句话说，如果连续两次调用该方法，则第二次调用将返回 false。

而 isInterrupted()方法为实例方法，测试线程是否已经中断，并不会清除当前线程中断状态。

所以这里应该使用 isInterrupted()实例方法，就可以修复该问题。

56.惰性初始化

```
public class Lazy {
    private static boolean initial = false;
```

```
    static {
        Thread t = new Thread(new Runnable() {
            public void run() {
                System.out.println("befor...");//此句会输出
                /*
                 * 由于使用 Lazy.initial 静态成员，又因为 Lazy 还未 初
                 * 始化完成，所以该线程会在这里等待主线程初始化完成
                 */
                initial = true;
                System.out.println("after...");//此句不会输出
            }
        });
        t.start();
        try {
            t.join();// 主线程等待 t 线程结束
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        System.out.println(initial);
    }
}
```

看看上面变态的程序，一个静态变量的初始化由静态块里的线程来初始化，最后的结果怎样？

当一个线程访问一个类的某个成员的时候，它会去检查这个类是否已经被初始化，在这一过程中会有以下四种情况：

1、 这个类尚未被初始化
2、 这个类正在被当前线程初始化：这是对初始化的递归请求，会直接忽略掉（另，请参考《构造器中静态常量的引用问题》一节）
3、 这个类正在被其他线程而不是当前线程初始化：需等待其他线程初始化完成再使用类的 Class 对象，而不会两个线程都会去初始化一遍（如果这样，那不类会初始化两遍，这显示不合理）
4、 这个类已经被初始化

当主线程调用 Lazy.main，它会检查 Lazy 类是否已经被初始化。此时它并没有被初始化（情况 1），所以主线程会记录下当前正在进行的初始化，并开始对这个类进行初始化。这个过程是：主线程会将 initial 的值设为 false，然后在静态块中创建并启动一个初始化 initial 的线程 t，该线程的 run 方法会将 initial 设为 true，然后主线程会等待 t 线程执行完毕，此时，问题就来了。

由于 t 线程将 Lazy.initial 设为 true 之前，它也会去检查 Lazy 类是否已经被初始化。这时，这个类正在被另外一个线程（mian 线程）进行初始化（情况 3）。在这种情况下，当前线程，也就是 t 线程，会等待 Class 对象直到初始化完成，可惜的是，那个正在进行初始化工作的

main 线程，也正在等待 t 线程的运行结束。因为这两个线程现在正相互等待，形成了死锁。

修正这个程序的方法就是让主线程在等待线程前就完成初始化操作：

```
public class Lazy {
    private static boolean initial = false;
    static Thread t = new Thread(new Runnable() {
        public void run() {
            initial = true;
        }
    });
    static {
        t.start();
    }

    public static void main(String[] args) {
        // 让 Lazy 类初始化完成后再调用 join 方法
        try {
            t.join();// 主线程等待 t 线程结束
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(initial);
    }
}
```

虽然修正了该程序挂起问题，但如果还有另一线程要访问 Lazy 的 initial 时，则还是很有可能不等 initial 最后赋值就被使用了。

总之，在类的初始化期间等待某个线程很可能会造成死锁，要让类初始化的动作序列尽可能地简单。

57.继承内部类

一般地，要想实例化一个内部类，如类 Inner1，需要提供一个外围类的实例给构造器。一般情况下，它是隐式地传递给内部类的构造器，但是它也是可以以 expression.super(args) 的方式即通过调用超类的构造器显式的传递。

```
public class Outer {
    class Inner1 extends Outer{
        Inner1(){
            super();
        }
    }
    class Inner2 extends Inner1{
        Inner2(){
```

```
            Outer.this.super();
        }
        Inner2(Outer outer){
            outer.super();
        }
    }
}

class WithInner {
    class Inner {}
}
class InheritInner extends WithInner.Inner {
    // ! InheritInner() {} //  不能编译
    /*
     * 这里的 super 指 InheritInner 类的父类 WithInner.Inner 的默认构造函数，而不是
     * WithInner 的父类构造函数，这种特殊的语法只在继承一个非静态内部类时才用到，
     * 表示继承非静态内部类时，外围对象一定要存在，并且只能在 第一行调用，而且一
     * 定要调用一下。为什么不能直接使用 super()或不直接写出呢？最主要原因就是每个
     * 非静态的内部类都会与一个外围类实例对应，这个外围类实例是运行时传到内
     * 部类里去的，所以在内部类里可以直接使用那个对象（比如 Outer.this），但这里
     * 是在外部内外 ，使用时还是需要存在外围类实例对象，所以这里就显示的通过构造
     * 器传递进来，并且在外围对象上显示的调用一下内部类的构造器，这样就确保了在
     * 继承至一个类部类的情况下 ，外围对象一类会存在的约束。
     */
    InheritInner(WithInner wi) {
        wi.super();
    }

    public static void main(String[] args) {
        WithInner wi = new WithInner();
        InheritInner ii = new InheritInner(wi);
    }
}
```

58.Hash集合序列化问题

```
class Super implements Serializable{
    // HashSet 要放置在父类中会百分百机率出现
    //  放置到子类中就不一定会出现问题了
    final Set set = new HashSet();
}
class Sub extends Super {
    private int id;
    public Sub(int id) {
```

```
            this.id = id;
            set.add(this);
        }
        public int hashCode() {
            return id;
        }
        public boolean equals(Object o) {
            return (o instanceof Sub) && (id == ((Sub) o).id);
        }
}

public class SerialKiller {
        public static void main(String[] args) throws Exception {
            Sub sb = new Sub(888);
            System.out.println(sb.set.contains(sb));// true

            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            new ObjectOutputStream(bos).writeObject(sb);

            ByteArrayInputStream bin = new ByteArrayInputStream(bos.toByteArray());
            sb = (Sub) new ObjectInputStream(bin).readObject();

            System.out.println(sb.set.contains(sb));// false
        }
}
```

Hash 一类集合都实现了序列化的 writeObject()与 readObject()方法。这里错误原因是由 HashSet 的 readObject 方法引起的。在某些情况下，这个方法会间接地调用某个未初始化对象的被覆写的方法。为了组装正在反序列化的 HashSet，HashSet.readObject 调用了 HashMap.put 方法，而 put 方法会去调用键的 hashCode 方法。由于整个对象图正在被反序列化，并没有什么可以保证每个键在它的 hashCode 方法被调用时已经被完全初始化了，因为 HashSet 是在父类中定义的，而在序列化 HashSet 时子类还没有开始初始化（这里应该是序列化）子类，所以这就造成了在父类中调用还没有初始完成（此时 id 为 0）的被子类覆写的 hashCode 方法，导致该对象重新放入 hash 表格的位置与反序列化前不一样了。hashCode 返回了错误的值，相应的键值对条目将会放入错误的单元格中，当 id 被初始化为 888 时，一切都太迟了。

这个程序的说明，包含了 HashMap 的 readObject 方法的序列化系统总体上违背了不能从类的构造器或伪构造器（如序列化的 readObject）中调用可覆写方法的规则。

如果一个 HashSet、Hashtable 或 HashMap 被序列化，那么请确认它们的内容没有直接或间接地引用它们自身，即正在被序列化的对象。

另外，在 readObject 或 readResolve 方法中，请避免直接或间接地在正在进行反序列化的对

象上调用任何方法，因为正在反序列化的对象处于不稳定状态。

59.迷惑的内部类

```
public class Twisted {
    private final String name;
    Twisted(String name) {
        this.name = name;
    }
    // 私有的不能被继承，但能被内部类直接访问
    private String name() {
        return name;
    }
    private void reproduce() {
        new Twisted("reproduce") {
            void printName() {
                // name()为外部类的，因为没有被继承过来
                System.out.println(name());// main
            }
        }.printName();
    }

    public static void main(String[] args) {
        new Twisted("main").reproduce();
    }
}
```

在顶层的类型中，即本例中的 Twisted 类，所有的本地的、内部的、嵌套的长匿名的类都可以毫无限制地访问彼此的成员。

另一个原因是私有的不能被继承。

60.编译期常量表达式

第一个 PrintWords 代表客户端，第二个 Words 代表一个类库：

```
class PrintWords {
    public static void main(String[] args) {
        System.out//引用常量变量
                .println(Words.FIRST + " "
                        + Words.SECOND + " "
                        + Words.THIRD);
    }
}
```

```
class Words {
    // 常量变量
    public static final String FIRST = "the";
    // 非常量变量
    public static final String SECOND = null;
    // 常量变量
    public static final String THIRD = "set";
}
```

现在假设你像下面这样改变了那个库类并且重新编译了这个类，但并不重新编译客户端的程序 PrintWords：

```
class Words {
    public static final String FIRST = "physics";
    public static final String SECOND = "chemistry";
    public static final String THIRD = "biology";
}
```

此时，端的程序会打印出什么呢？结果是 the chemistry set，不是 the null set，也不是 physics chemistry biology，为什么？原因就是 null 不是一个编译期常量表达式，而其他两个都是。

对于常量变量（如上面 Words 类中的 FIRST、THIRD）的引用（如在 PrintWords 类中对 Words.FIRST、Words.THIRD 的引用）会在编译期被转换为它们所表示的常量的值（即 PrintWords 类中的 Words.FIRST、Words.THIRD 引用会替换成"the"与"set"）。

一个常量变量（如上面 Words 类中的 FIRST、THIRD）的定义是，一个在编译期被常量表达式（即编译期常量表达式）初始化的 final 的原生类型或 String 类型的变量。

那什么是“编译期常量表达式”？精确定义在[JLS 15.28]中可以找到，这样要说的是 null 不是一个编译期常量表达式。

由于常量变量会编译进客户端，API 的设计者在设计一个常量域之前应该仔细考虑一下是否应该定义成常量变量。

如果你使用了一个非常量的表达式去初始化一个域，甚至是一个 final 或，那么这个域就不是一个常量。下面你可以通过将一个常量表达式传给一个方法使用得它变成一个非常量：

```
class Words {
    // 以下都成非常量变量
    public static final String FIRST = ident("the");
    public static final String SECOND = ident(null);
    public static final String THIRD = ident("set");
    private static String ident(String s) {
        return s;
    }
}
```

总之，常量变量将会被编译进那些引用它们的类中。一个常量变量就是任何常量表达式初始

化的原生类型或字符串变量。且 null 不是一个常量表达式。

61.打乱数组

```
class Shuffle {
    private static Random rd = new Random();
    public static void shuffle(Object[] a) {
        for (int i = 0; i < a.length; i++) {
            swap(a, i, rd.nextInt(a.length));
        }
    }
    public static void swap(Object[] a, int i, int j) {
        Object tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }
    public static void main(String[] args) {
        Map map = new TreeMap();
        for (int i = 0; i < 9; i++) {
            map.put(i, 0);
        }

        // 测试数组上的每个位置放置的元素是否等概率
        for (int i = 0; i < 10000; i++) {
            Integer[] intArr = new Integer[] { 0, 1, 2, 3, 4, 5, 6, 7, 8 };
            shuffle(intArr);
            for (int j = 0; j < 9; j++) {
                map.put(j,(Integer)map.get(j)+intArr[j]);
            }
        }
        System.out.println(map);
        for (int i = 0; i < 9; i++) {
            map.put(i,(Integer) map.get(i)/10000f);
        }
        System.out.println(map);
    }
}
```

上面的算法不是很等概率的让某个元素打乱到其位置，程序运行了多次，大致的结果为：
{0=36031, 1=38094, 2=39347, 3=40264, 4=41374, 5=41648, 6=41780, 7=41188, 8=40274}
{0=3.6031, 1=3.8094, 2=3.9347, 3=4.0264, 4=4.1374, 5=4.1648, 6=4.178, 7=4.1188, 8=4.0274}

如果某个位置上等概率出现这 9 个值的话，则平均值会趋近于 4，但测试的结果表明：开始的时候比较低，然后增长超过了平均值，最后又降下来了。

如果改用下面算法：

```
public static void shuffle(Object[] a) {
    for (int i = 0; i < a.length; i++) {
        swap(a, i, i + rd.nextInt(a.length - i));
    }
}
```

多次测试的结果大致如下：

{0=40207, 1=40398, 2=40179, 3=39766, 4=39735, 5=39710, 6=40074, 7=39871, 8=40060}

{0=4.0207, 1=4.0398, 2=4.0179, 3=3.9766, 4=3.9735, 5=3.971, 6=4.0074, 7=3.9871, 8=4.006}

所以修改后的算法是合理的。

另一种打乱集合的方式是通过 Api 中的 Collections 工具类：

```
public static void shuffle(Object[] a) {
    Collections.shuffle(Arrays.asList(a));
}
```

其实算法与上面的基本相似，当然我们使用 API 中提供的会更好，会在效率上获得最大的受益。