Java 谜题 1——表达式谜题

谜题 1：奇数性

下面的方法意图确定它那唯一的参数是否是一个奇数。这个方法能够正确运转吗？

```
public static boolean isOdd(int i){
        return i % 2 == 1;
}
```

奇数可以被定义为被 2 整除余数为 1 的整数。表达式 i % 2 计算的是 i 整除 2 时所产生的余数，因此看起来这个程序应该能够正确运转。遗憾的是，它不能；它在四分之一的时间里返回的都是错误的答案。

为什么是四分之一？因为在所有的 int 数值中，有一半都是负数，而 isOdd 方法对于对所有负奇数的判断都会失败。在任何负整数上调用该方法都回返回 false ，不管该整数是偶数还是奇数。

这是 Java 对取余操作符（%）的定义所产生的后果。该操作符被定义为对于所有的 int 数值 a 和所有的非零 int 数值 b，都满足下面的恒等式：

```
(a / b) * b + (a % b) == a
```

换句话说，如果你用 b 整除 a，将商乘以 b，然后加上余数，那么你就得到了最初的值 a 。该恒等式具有正确的含义，但是当与 Java 的截尾整数整除操作符相结合时，它就意味着：当取余操作返回一个非零的结果时，它与左操作数具有相同的正负符号。

当 i 是一个负奇数时，i % 2 等于-1 而不是 1， 因此 isOdd 方法将错误地返回 false。为了防止这种意外，请测试你的方法在为每一个数值型参数传递负数、零和正数数值时，其行为是否正确。

这个问题很容易订正。只需将 i % 2 与 0 而不是与 1 比较，并且反转比较的含义即可：

```
public static boolean isOdd(int i){
        return i % 2 != 0;
}
```

如果你正在在一个性能临界（performance-critical）环境中使用 isOdd 方法，那么用位操作符 AND（&）来替代取余操作符会显得更好：

```
public static boolean isOdd(int i){
        return (i & 1) != 0;
}
```

总之，无论你何时使用到了取余操作符，都要考虑到操作数和结果的符号。该操作符的行为在其操作数非负时是一目了然的，但是当一个或两个操作数都是负数时，它的行为就不那么显而易见了。

谜题 2：找零时刻

请考虑下面这段话所描述的问题：

Tom 在一家汽车配件商店购买了一个价值$1.10 的火花塞，但是他钱包中都是两美元一张的钞票。如果他用一张两美元的钞票支付这个火花塞，那么应该找给他多少零钱呢？

下面是一个试图解决上述问题的程序，它会打印出什么呢？

```
public class Change{
        public static void main(String args[]){
                System.out.println(2.00 - 1.10);
        }
}
```

你可能会很天真地期望该程序能够打印出 0.90，但是它如何才能知道你想要打印小数点后两位小数呢？

如果你对在 Double.toString 文档中所设定的将 double 类型的值转换为字符串的规则有所了解，你就会知道该程序打印出来的小数，是足以将 double 类型的值与最靠近它的临近值区分出来的最短的小数，它在小数点之前和之后都至少有一位。因此，看起来，该程序应该打印 0.9 是合理的。

这么分析可能显得很合理，但是并不正确。如果你运行该程序，你就会发现它打印的是 0.8999999999999999。

问题在于 1.1 这个数字不能被精确表示成为一个 double，因此它被表示成为最接近它的 double 值。该程序从 2 中减去的就是这个值。遗憾的是，这个计算的结果并不是最接近 0.9 的 double 值。表示结果的 double 值的最短表示就是你所看到的打印出来的那个可恶的数字。

更一般地说，问题在于*并不是所有的小数都可以用二进制浮点数来精确表示的。*

如果你正在用的是 JDK 5.0 或更新的版本，那么你可能会受其诱惑，通过使用 printf 工具来设置输出精度的方订正该程序：

```
//拙劣的解决方案——仍旧是使用二进制浮点数
System.out.printf("%.2f%n",2.00 - 1.10);
```

这条语句打印的是正确的结果，但是这并不表示它就是对底层问题的通用解决方案：它使用的仍旧是二进制浮点数的 double 运算。浮点运算在一个范围很广的

值域上提供了很好的近似，但是它通常不能产生精确的结果。*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 0.1——或者 10 的其它任何次负幂——精确表示为一个长度有限的二进制小数

解决该问题的一种方式是使用某种整数类型，例如 int 或 long，并且以分为单位来执行计算。如果你采纳了此路线，请确保该整数类型大到足够表示在程序中你将要用到的所有值。对这里举例的谜题来说，int 就足够了。下面是我们用 int 类型来以分为单位表示货币值后重写的 println 语句。这个版本将打印出正确答案 90 分：

```
System.out.println((200 - 110) + "cents");
```

解决该问题的另一种方式是使用执行精确小数运算的 BigDecimal。它还可以通过 JDBC 与 SQL DECIMAL 类型进行互操作。这里要告诫你一点： *一定要用 BigDecimal(String)构造器，而千万不要用 BigDecimal(double)*。后一个构造器将用它的参数的“精确”值来创建一个实例：new BigDecimal(.1)将返回一个表示 0.1000000000000000055511151231257827021181583404541015625 的 BigDecimal。通过正确使用 BigDecimal，程序就可以打印出我们所期望的结果 0.90：

```
import java.math.BigDecimal;
public class Change1{
        public static void main(String args[]){
                System.out.println(new BigDecimal("2.00").
                subtract(new BigDecimal("1.10")));
        }
}
```

这个版本并不是十分地完美，因为 Java 并没有为 BigDecimal 提供任何语言上的支持。使用 BigDecimal 的计算很有可能比那些使用原始类型的计算要慢一些，对某些大量使用小数计算的程序来说，这可能会成为问题，而对大多数程序来说，这显得一点也不重要。

总之， *在需要精确答案的地方，要避免使用 float 和 double；对于货币计算，要使用 int、long 或 BigDecimal*。对于语言设计者来说，应该考虑对小数运算提供语言支持。一种方式是提供对操作符重载的有限支持，以使得运算符可以被塑造为能够对数值引用类型起作用，例如 BigDecimal。另一种方式是提供原始的小数类型，就像 COBOL 与 PL/I 所作的一样。

谜题 3：长整除

这个谜题之所以被称为长整除是因为它所涉及的程序是有关两个 long 型数值整除的。被除数表示的是一天里的微秒数；而除数表示的是一天里的毫秒数。这个程序会打印出什么呢？

```
public class LongDivision{
        public static void main(String args[]){
                final long MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
```

```
                final long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
                System.out.println(MICROS_PER_DAY/MILLIS_PER_DAY);
        }
}
```

这个谜题看起来相当直观。每天的毫秒数和每天的微秒数都是常量。为清楚起见，它们都被表示成积的形式。每天的微秒数是（24 小时/天*60 分钟/小时*60 秒/分钟*1000 毫秒/秒*1000 微秒/毫秒）。而每天的毫秒数的不同之处只是少了最后一个因子 1000。

当你用每天的毫秒数来整除每天的微秒数时，除数中所有的因子都被约掉了，只剩下 1000，这正是每毫秒包含的微秒数。

除数和被除数都是 long 类型的，long 类型大到了可以很容易地保存这两个乘积而不产生溢出。因此，看起来程序打印的必定是 1000。

遗憾的是，它打印的是 5。这里到底发生了什么呢？

问题在于常数 MICROS_PER_DAY 的计算“确实”溢出了。*尽管计算的结果适合放入 long 中，并且其空间还有富余，但是这个结果并不适合放入 int 中。这个计算完全是以 int 运算来执行的，并且只有在运算完成之后，其结果才被提升到 long，而此时已经太迟了：计算已经溢出了，它返回的是一个小了 200 倍的数值。从 int 提升到 long 是一种拓宽原始类型转换(widening primitive conversion)，它保留了（不正确的）数值。这个值之后被 MILLIS_PER_DAY 整除，而 MILLIS_PER_DAY 的计算是正确的，因为它适合 int 运算。这样整除的结果就得到了 5。*

那么为什么计算会是以 int 运算来执行的呢？因为所有乘在一起的因子都是 int 数值。当你将两个 int 数值相乘时，你将得到另一个 int 数值。*Java 不具有目标确定类型的特性，这是一种语言特性，其含义是指存储结果的变量的类型会影响到计算所使用的类型。*

通过使用 long 常量来替代 int 常量作为每一个乘积的第一个因子，我们就可以很容易地订正这个程序。这样做可以强制表达式中所有的后续计算都用 long 运作来完成。尽管这么做只在 MICROS_PER_DAY 表达式中是必需的，但是在两个乘积中都这么做是一种很好的方式。相似地，使用 long 作为乘积的“第一个”数值也并不总是必需的，但是这么做也是一种很好的形式。在两个计算中都以 long 数值开始可以很清楚地表明它们都不会溢出。下面的程序将打印出我们所期望的 1000：

```
public class LongDivision{
        public static void main(String args[ ]){
                final long MICROS_PER_DAY = 24L * 60 * 60 * 1000 * 1000;
                final long MILLIS_PER_DAY = 24L * 60 * 60 * 1000;
                System.out.println(MICROS_PER_DAY/MILLIS_PER_DAY);
        }
```

```
}
```
这个教训很简单：*当你在操作很大的数字时，千万要提防溢出——它可是一个缄默杀手*。即使用来保存结果的变量已显得足够大，也并不意味着要产生结果的计算具有正确的类型。当你拿不准时，就使用 long 运算来执行整个计算。

语言设计者从中可以吸取的教训是：也许降低缄默溢出产生的可能性确实是值得做的一件事。这可以通过对不会产生缄默溢出的运算提供支持来实现。程序可以抛出一个异常而不是直接溢出，就像 Ada 所作的那样，或者它们可以在需要的时候自动地切换到一个更大的内部表示上以防止溢出，就像 Lisp 所作的那样。这两种方式都可能会遭受与其相关的性能方面的损失。降低缄默溢出的另一种方式是支持目标确定类型，但是这么做会显著地增加类型系统的复杂度

谜题 4：初级问题

得啦，前面那个谜题是有点棘手，但它是有关整除的，每个人都知道整除是很麻烦的。那么下面的程序只涉及加法，它又会打印出什么呢？

```
public class Elementary{
        public static void main(String[] args){
                System.out.println(12345+54321);
        }
}
```
从表面上看，这像是一个很简单的谜题——简单到不需要纸和笔你就可以解决它。加号的左操作数的各个位是从 1 到 5 升序排列的，而右操作数是降序排列的。因此，相应各位的和仍然是常数，程序必定打印 66666。对于这样的分析，只有一个问题：当你运行该程序时，它打印出的是 17777。难道是 Java 对打印这样的非常数字抱有偏见吗？不知怎么的，这看起来并不像是一个合理的解释。

事物往往有别于它的表象。就以这个问题为例，它并没有打印出我们想要的输出。请仔细观察 + 操作符的两个操作数，我们是将一个 int 类型的 12345 加到了 long 类型的 54321 上。请注意左操作数开头的数字 1 和右操作数结尾的小写字母 l 之间的细微差异。数字 1 的水平笔划（称为“臂（arm）”）和垂直笔划（称为“茎（stem）”）之间是一个锐角，而与此相对照的是，小写字母 l 的臂和茎之间是一个直角。

在你大喊“恶心！”之前，你应该注意到这个问题确实已经引起了混乱，这里确实有一个教训：*在 long 型字面常量中，一定要用大写的 L，千万不要用小写的 l*。这样就可以完全掐断这个谜题所产生的混乱的源头。

```
System.out.println(12345+5432L);
```
相类似的，*要避免使用单独的一个 l 字母作为变量名*。例如，我们很难通过观察下面的代码段来判断它到底是打印出列表 l 还是数字 1。

```
//不良代码-使用了 l 作为变量名
List l = new ArrayList<String>();
l.add("Foo");
```

```
System.out.println(1);
```
总之，小写字母 l 和数字 1 在大多数打字机字体中都是几乎一样的。为避免你的程序的读者对二者产生混淆，千万不要使用小写的 l 来作为 long 型字面常量的结尾或是作为变量名。Java 从 C 编程语言中继承良多，包括 long 型字面常量的语法。也许当初允许用小写的 l 来编写 long 型字面常量本身就是一个错误。

谜题 5：十六进制的趣事

下面的程序是对两个十六进制（hex）字面常量进行相加，然后打印出十六进制的结果。这个程序会打印出什么呢？

```
public class JoyOfHex{
        public static void main(String[] args){
                System.out.println(
                        Long.toHexString(0x100000000L + 0xcafebabe));
        }
}
```

看起来很明显，该程序应该打印出 1cafebabe。毕竟，这确实就是十六进制数字 10000000016 与 cafebabe16 的和。该程序使用的是 long 型运算，它可以支持 16 位十六进制数，因此运算溢出是不可能的。

然而，如果你运行该程序，你就会发现它打印出来的是 cafebabe，并没有任何前导的 1。这个输出表示的是正确结果的低 32 位，但是不知何故，第 33 位丢失了。

看起来程序好像执行的是 int 型运算而不是 long 型运算，或者是忘了加第一个操作数。这里到底发生了什么呢？

十进制字面常量具有一个很好的属性，即所有的十进制字面常量都是正的，而十六进制或是八进制字面常量并不具备这个属性。要想书写一个负的十进制常量，可以使用一元取反操作符（-）连接一个十进制字面常量。以这种方式，你可以用十进制来书写任何 int 或 long 型的数值，不管它是正的还是负的，并且*负的十进制常数可以很明确地用一个减号符号来标识*。但是十六进制和八进制字面常量并不是这么回事，它们可以具有正的以及负的数值。*如果十六进制和八进制字面常量的最高位被置位了，那么它们就是负数*。在这个程序中，数字 0xcafebabe 是一个 int 常量，它的最高位被置位了，所以它是一个负数。它等于十进制数值 -889275714。

该程序执行的这个加法是一种“混合类型的计算（mixed-type computation）：左操作数是 long 类型的，而右操作数是 int 类型的。为了执行该计算，Java 将 int 类型的数值用拓宽原始类型转换提升为一个 long 类型，然后对两个 long 类型数值相加。因为 int 是一个有符号的整数类型，所以这个转换执行的是符合扩展：*它将负的 int 类型的数值提升为一个在数值上相等的 long 类型数值*。

这个加法的右操作数 0xcafebabe 被提升为了 long 类型的数值 0xffffffffcafebabeL。这个数值之后被加到了左操作数 0x100000000L 上。当作为 int 类型来被审视时，经过符号扩展之后的右操作数的高 32 位是-1，而左操作数的高 32 位是 1，将这两个数值相加就得到了 0，这也就解释了为什么在程序输出中前导 1 丢失了。下面所示是用手写的加法实现。（在加法上面的数字是进位。）

```
    1111111
  0xffffffffcafebabeL
+ 0x0000000100000000L
----------------------
  0x00000000cafebabeL
```

订正该程序非常简单，只需用一个 long 十六进制字面常量来表示右操作数即可。这就可以避免了具有破坏力的符号扩展，并且程序也就可以打印出我们所期望的结果 1cafebabe：

```
public class JoyOfHex{
        public static void main(String[] args){
                System.out.println(
                        Long.toHexString(0x100000000L + 0xcafebabeL));
        }
}
```

这个谜题给我们的教训是：*混合类型的计算可能会产生混淆，尤其是十六进制和八进制字面常量无需显式的减号符号就可以表示负的数值。为了避免这种窘境，通常最好是避免混合类型的计算*。对于语言的设计者们来说，应该考虑支持无符号的整数类型，从而根除符号扩展的可能性。可能会有这样的争辩：负的十六进制和八进制字面常量应该被禁用，但是这可能会挫伤程序员，他们经常使用十六进制字面常量来表示那些符号没有任何重要含义的数值。

谜题 6：多重转型

转型被用来将一个数值从一种类型转换到另一种类型。下面的程序连续使用了三个转型。那么它到底会打印出什么呢？

```
public class Multicast{
        public static void main (String[] args){
                System.out.println((int)(char)(byte) -1);
        }
}
```

无论你怎样分析这个程序，都会感到很迷惑。它以 int 数值-1 开始，然后从 int 转型为 byte，之后转型为 char，最后转型回 int。第一个转型将数值从 32 位窄化到了 8 位，第二个转型将数值从 8 位拓宽到了 16 位，最后一个转型又将数值从 16 位拓宽回了 32 位。这个数值最终是回到了起点吗？如果你运行该程序，你就会发现不是。它打印出来的是 65535，但是这是为什么呢？

该程序的行为紧密依赖于转型的符号扩展行为。Java 使用了基于 2 的补码的二进制运算，因此 int 类型的数值-1 的所有 32 位都是置位的。*从 int 到 byte 的转型是很简单的，它执行了一个窄化原始类型转化（narrowing primitive conversion），直接将除低 8 位之外的所有位全部砍掉。*这样做留下的是一个 8 位都被置位了的 byte，它仍旧表示-1。

从 byte 到 char 的转型稍微麻烦一点，因为 byte 是一个有符号类型，而 char 是一个无符号类型。在将一个整数类型转换成另一个宽度更宽的整数类型时，通常是可以保持其数值的，但是却不可能将一个负的 byte 数值表示成一个 char。因此，从 byte 到 char 的转换被认为不是一个拓宽原始类型的转换，而是一个拓宽并窄化原始类型的转换（widening and narrowing primitive conversion）：byte 被转换成了 int，而这个 int 又被转换成了 char。

所有这些听起来有点复杂，幸运的是，*有一条很简单的规则能够描述从较窄的整型转换成较宽的整型时的符号扩展行为：如果最初的数值类型是有符号的，那么就执行符号扩展；如果它是 char，那么不管它将要被转换成什么类型，都执行零扩展。*了解这条规则可以使我们很容易地解决这个谜题。

*因为 byte 是一个有符号的类型，所以在将 byte 数值-1 转换成 char 时，会发生符号扩展。作为结果的 char 数值的 16 个位就都被置位了，因此它等于 216-1，即 65535。从 char 到 int 的转型也是一个拓宽原始类型转换，所以这条规则告诉我们，它将执行零扩展而不是符号扩展。作为结果的 int 数值也就成了 65535，*这正是程序打印出的结果。

尽管这条简单的规则描述了在有符号和无符号整型之间进行拓宽原始类型时的符号扩展行为，你最好还是不要编写出依赖于它的程序。如果你正在执行一个转型到 char 或从 char 转型的拓宽原始类型转换，并且这个 char 是仅有的无符号整型，那么你最好将你的意图明确地表达出来。

如果你在将一个 char 数值 c 转型为一个宽度更宽的类型，并且你不希望有符号扩展，那么为清晰表达意图，可以考虑使用一个位掩码，即使它并不是必需的：

```
int i = c & 0xffff;
```

或者，书写一句注释来描述转换的行为：

```
int i = c; //不会执行符号扩展
```

如果你在将一个 char 数值 c 转型为一个宽度更宽的整型，并且你希望有符号扩展，那么就先将 char 转型为一个 short，它与 char 具有同样的宽度，但是它是有符号的。在给出了这种细微的代码之后，你应该也为它书写一句注释：

```
int i = (short) c; //转型将引起符号扩展
```

如果你在将一个 byte 数值 b 转型为一个 char，并且你不希望有符号扩展，那么你必须使用一个位掩码来限制它。这是一种通用做法，所以不需要任何注释：

```
char c = (char) (b & 0xff);
```

这个教训很简单：*如果你通过观察不能确定程序将要做什么，那么它做的就很有可能不是你想要的。*要为明白清晰地表达你的意图而努力。尽管有这么一条简单

的规则，描述了涉及有符号和无符号整型拓宽转换的符号扩展行为，但是大多数程序员都不知道它。如果你的程序依赖于它，那么你就应该把你的意图表达清楚。

谜题 7：互换内容

下面的程序使用了复合的异或赋值操作符，它所展示的技术是一种编程习俗。那么它会打印出什么呢？

```
public class CleverSwap{
        public static void main(String[] args){
                int x = 1984; // (0x7c0)
                int y = 2001; // (0x7d1)
                x^= y^= x^= y;
                System.out.println("x= " + x + "; y= " + y);
        }
}
```

就像其名称所暗示的，这个程序应该交换变量 x 和 y 的值。如果你运行它，就会发现很悲惨，它失败了，打印的是

```
x = 0; y = 1984。
```

交换两个变量的最显而易见的方式是使用一个临时变量：

```
int tmp = x;
x = y;
y = tmp;
```

很久以前，当中央处理器只有少数寄存器时，人们发现可以通过利用异或操作符（^）的属性(x ^ y ^ x) == y 来避免使用临时变量：

```
x = x ^ y;
y = y ^ x;
x = y ^ x;
```

这个惯用法曾经在 C 编程语言中被使用过，并进一步被构建到了 C++中，但是它并不保证在二者中都可以正确运行。但是有一点是肯定的，那就是它在 Java 中肯定是不能正确运行的。

Java 语言规范描述到：*操作符的操作数是从左向右求值的。*为了求表达式 x ^= expr 的值，x 的值是在计算 expr 之前被提取的，并且这两个值的异或结果被赋给变量 x。在 CleverSwap 程序中，变量 x 的值被提取了两次——每次在表达式中出现时都提取一次——但是两次提取都发生在所有的赋值操作之前。

下面的代码段详细地描述了将互换惯用法分解开之后的行为，并且解释了为什么产生的是我们所看到的输出：

```
// Java 中 x^= y^= x^= y 的实际行为
int tmp1 = x ; // x 在表达式中第一次出现
int tmp2 = y ; // y 的第一次出现
```

```
int tmp3 = x ^ y ; // 计算 x ^ y
x = tmp3 ; // 最后一个赋值：存储 x ^ y 到 x
y = tmp2 ^ tmp3 ; // 第二个赋值：存储最初的 x 值到 y 中
x = tmp1 ^ y ; // 第一个赋值：存储 0 到 x 中
```

在 C 和 C++中，并没有指定表达式的计算顺序。当编译表达式 x ^= expr 时，许多 C 和 C++编译器都是在计算 expr 之后才提取 x 的值的，这就使得上述的惯用法可以正常运转。尽管它可以正常运转，但是它仍然违背了 C/C++有关不能在两个连续的序列点之间重复修改变量的规则。因此，这个惯用法的行为在 C 和 C++中也没有明确定义。

为了看重其价值，我们还是可以写出不用临时变量就可以互换两个变量内容的 Java 表达式的。但是它同样是丑陋而无用的：

```
// 杀鸡用牛刀的做法，千万不要这么做！
y = (x^= (y^= x))^ y ;
```

这个教训很简单：*在单个的表达式中不要对相同的变量赋值两次。*表达式如果包含对相同变量的多次赋值，就会引起混乱，并且很少能够执行你希望的操作。即使对多个变量进行赋值也很容易出错。更一般地讲，要避免所谓聪明的编程技巧。它们都是易于产生 bug 的，很难以维护，并且运行速度经常是比它们所替代掉的简单直观的代码要慢。

语言设计者可能会考虑禁止在一个表达式中对相同的变量多次赋值，但是在一般的情况下，强制执行这条禁令会因为别名机制的存在而显得很不灵活。例如，请考虑表达式 x = a[i]++ - a[j]++，它是否递增了相同的变量两次呢？这取决于在表达式被计算时 i 和 j 的值，并且编译器通常是无法确定这一点。

谜题 8：Dos Equis

这个谜题将测试你对条件操作符的掌握程度，这个操作符有一个更广为人知的名字：问号冒号操作符。下面的程序将会打印出什么呢？

```
public class DosEquis{
        public static void main(String[] args){
                char x = 'X';
                int i = 0;
                System.out.println(true ? x : 0);
                System.out.println(false ? i : x);
        }
}
```

这个程序由两个变量声明和两个 print 语句构成。第一个 print 语句计算条件表达式(true ? x : 0)并打印出结果，这个结果是 char 类型变量 x 的值'X'。而第二个 print 语句计算表达式(false ? i : x)并打印出结果，这个结果还是依旧是'X'的 x，因此这个程序应该打印 XX。然而，如果你运行该程序，你就会发现它打印出来的是 X88。这种行为看起来挺怪的。第一个 print 语句打印的是 X，而第二个打印的却是 88。它们的不同行为说明了什么呢？

答案就在规范有关条件表达式部分的一个阴暗的角落里。请注意在这两个表达式中，每一个表达式的第二个和第三个操作数的类型都不相同：x 是 char 类型的，而 0 和 i 都是 int 类型的。就像在谜题 5 的解答中提到的，*混合类型的计算会引起混乱，而这一点比在条件表达式中比在其它任何地方都表现得更明显。*你可能考虑过，这个程序中两个条件表达式的结果类型是相同的，就像它们的操作数类型是相同的一样，尽管操作数的顺序颠倒了一下，但是实际情况并非如此。

确定条件表达式结果类型的规则过于冗长和复杂，很难完全记住它们，但是其核心就是一下三点：

- *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。换句话说，你可以通过绕过混合类型的计算来避免大麻烦。*
- *如果一个操作数的类型是 T，T 表示 byte、short 或 char，而另一个操作数是一个 int 类型的常量表达式，它的值是可以用类型 T 表示的，那么条件表达式的类型就是 T。*
- *否则，将对操作数类型运用二进制数字提升，而条件表达式的类型就是第二个和第三个操作数被提升之后的类型。*

2、3 两点对本谜题是关键。在程序的两个条件表达式中，一个操作数的类型是 char，另一个的类型是 int。在两个表达式中，int 操作数都是 0，它可以被表示成一个 char。然而，只有第一个表达式中的 int 操作数是常量（0），而第二个表达式中的 int 操作数是变量（i）。因此，*第 2 点被应用到了第一个表达式上，它返回的类型是 char，而第 3 点被应用到了第二个表达式上，其返回的类型是对 int 和 char 运用了二进制数字提升之后的类型，即 int。*

条件表达式的类型将确定哪一个重载的 print 方法将被调用。对第一个表达式来说，PrintStream.print(char)将被调用，而对第二个表达式来说，PrintStream.print(int)将被调用。前一个重载方法将变量 x 的值作为 Unicode 字符（X）来打印，而后一个重载方法将其作为一个十进制整数（88）来打印。至此，谜题被解开了。

总之，*通常最好是在条件表达式中使用类型相同的第二和第三操作数。*否则，你和你的程序的读者必须要彻底理解这些表达式行为的复杂规范。

对语言设计者来说，也许可以设计一个牺牲掉了部分灵活性，但是增加了简洁性的条件操作符。例如，要求第二和第三操作数必须就有相同的类型，这看起来就很合理。或者，条件操作符可以被定义为对常量没有任何特殊处理。为了让这些选择对程序员来说更加容易接受，可以提供用来表示所有原始类型字面常量的语法。这也许确实是一个好注意，因为它增加了语言的一致性和完备性，同时又减少了对转型的需求。

谜题 9：半斤

现在该轮到你来写些代码了，好消息是，你只需为这个谜题编写两行代码，并为下一个谜题也编写两行代码。这有什么难的呢？我们给出一个对变量 x 和 i 的声明即可，它肯定是一个合法的语句：

```
x += i;
```

但是，它并不是：

```
x = x + i;
```

许多程序员都会认为该迷题中的第一个表达式（x += i）只是第二个表达式（x = x + i）的简写方式。但是这并不十分准确。这两个表达式都被称为赋值表达式。第二条语句使用的是简单赋值操作符（=），而第一条语句使用的是复合赋值操作符。（复合赋值操作符包括 +=、-=、*=、/=、%=、<<=、>>=、>>>=、&=、^= 和|=）Java 语言规范中讲到，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，除非 E1 只被计算一次。

换句话说，*复合赋值表达式自动地将它们所执行的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始类型转换。因此，我们有很好的理由去解释为什么在尝试着执行等价的简单赋值可能会产生一个编译错误。

为了说得具体一些，并提供一个解决方案给这个谜题，假设我们在该谜题的两个赋值表达式之前有下面这些声明：

```
short x = 0;
int i = 123456;
```

复合赋值编译将不会产生任何错误：

```
x += i; // 包含了一个隐藏的转型！
```

你可能期望 x 的值在这条语句执行之后是 123,456，但是并非如此！它的值是 -7,616。int 类型的数值 123456 对于 short 来说太大了。自动产生的转型悄悄地把 int 数值的高两位给截掉了。这也许就不是你想要的了。

相对应的简单赋值是非法的，因为它试图将 int 数值赋值给 short 变量，它需要一个显式的转型：

```
x = x + i; // 不要编译——“可能会丢掉精度”
```

这应该是明显的，复合赋值表达式可能是很危险的。为了避免这种令人不快的突袭，*请不要将复合赋值操作符作用于 byte、short 或 char 类型的变量上*。在将复合赋值操作符作用于 int 类型的变量上时，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 类型的变量上时，要确保表达式右侧不是 double 类型。这些规则足以防止编译器产生危险的窄化转型。

总之，复合赋值操作符会悄悄地产生一个转型。如果计算结果的类型宽于变量的类型，那么所产生的转型就是一个危险的窄化转型。这样的转型可能会悄悄地丢弃掉精度或数量值。对语言设计者来说，也许让复合赋值操作符产生一个不可见的转型本身就是一个错误；对于在复合赋值中的变量类型比计算结果窄的情况，也许应该让其非法才对。

谜题 10：八两

与上面的例子相反，如果我们给出的关于变量 x 和 i 的声明是如下的合法语句：

```
x = x + i;
```

但是，它并不是：

```
x += i;
```

乍一看，这个谜题可能看起来与前面一个谜题相同。但是请放心，它们并不一样。这两个谜题在哪一条语句必是合法的，以及哪一条语句必是不合法的方面，正好相反。

就像前面的谜题一样，这个谜题也依赖于有关复合赋值操作符的规范中的细节。二者的相似之处就此打住。基于前面的谜题，你可能会想：符合赋值操作符比简单赋值操作符的限制要少一些。在一般情况下，这是对的，但是有这么一个领域，在其中简单赋值操作符会显得更宽松一些。

复合赋值操作符要求两个操作数都是原始类型的，例如 int，或包装了的原始类型，例如 Integer，但是有一个例外：如果在+=操作符左侧的操作数是 String 类型的，那么它允许右侧的操作数是任意类型，在这种情况下，该操作符执行的是字符串连接操作。简单赋值操作符（=）允许其左侧的是对象引用类型，这就显得要宽松许多了：你可以使用它们来表示任何你想要表示的内容，只要表达式的右侧与左侧的变量是赋值兼容的即可。

你可以利用这一差异来解决该谜题。要想用 += 操作符来执行字符串连接操作，你就必须将左侧的变量声明为 String 类型。通过使用直接赋值操作符，字符串连接的结果可以存放到一个 Object 类型的变量中。

为了说得具体一些，并提供一个解决方案给这个谜题，假设我们在该谜题的两个赋值表达式之前有下面这些声明：

```
Object x = "Buy ";
String i = "Effective Java!";
```

简单赋值是合法的，因为 x + i 是 String 类型的，而 String 类型又是与 Object 赋值兼容的：

```
x = x + i;
```

复合赋值是非法的，因为左侧是一个 Object 引用类型，而右侧是一个 String 类型：

```
x += i;
```

这个谜题对程序员来说几乎算不上什么教训。对语言设计者来说，加法的复合赋值操作符应该在右侧是 String 类型的情况下，允许左侧是 Object 类型。这项修改将根除这个谜题所展示的违背直觉的行为。

Java 谜题 2——字符谜题

谜题 11：最后的笑声

下面的程序将打印出什么呢？

```
public class LastLaugh{
    public static void main(String[] args){
        System.out.print("H"+"a");
        System.out.print('H'+'a');
    }
}
```

你可能会认为这个程序将打印 HaHa。该程序看起来好像是用两种方式连接了 H 和 a，但是你所见为虚。如果你运行这个程序，就会发现它打印的是 Ha169。那么，为什么它会产生这样的行为呢？

正如我们所期望的，第一个对 System.out.print 的调用打印的是 Ha：它的参数是表达式"H"+"a"，显然它执行的是一个字符串连接。而第二个对 System.out.print 的调用就是另外一回事了。*问题在于'H'和'a'是字符型字面常量，因为这两个操作数都不是字符串类型的，所以 + 操作符执行的是加法而不是字符串连接。*

编译器在计算常量表达式'H'+'a'时，是通过我们熟知的拓宽原始类型转换将两个具有字符型数值的操作数（'H'和'a'）提升为 int 数值而实现的。从 char 到 int 的拓宽原始类型转换是将 16 位的 char 数值零扩展到 32 位的 int。对于'H'，char 数值是 72，而对于'a'，char 数值是 97，因此表达式'H'+'a'等价于 int 常量 72 + 97，或 169。

站在语言的立场上，若干个 char 和字符串的相似之处是虚幻的。*语言所关心的是，char 是一个无符号 16 位原始类型整数——仅此而已。*对类库来说就不尽如此了，类库包含了许多可以接受 char 参数，并将其作为 Unicode 字符处理的方法。

那么你应该怎样将字符连接在一起呢？你可以使用这些类库。例如，你可以使用一个字符串缓冲区：

```
StringBuffer sb = new StringBuffer();
sb.append('H');
sb.append('a');
System.out.println(sb);
```

这么做可以正常运行，但是显得很丑陋。其实我们还是有办法去避免这种方式所产生的拖沓冗长的代码。 你可以通过确保至少有一个操作数为字符串类型，来强制 + 操作符去执行一个字符串连接操作，而不是一个加法操作。这种常见的惯用法用一个空字符串（""）作为一个连接序列的开始，如下所示：

```
System.out.println("" + 'H' + 'a');
```

这种惯用法可以确保子表达式都被转型为字符串。尽管这很有用，但是多少有一点难看，而且它自身可能会引发某些混淆。你能猜到下面的语句将会打印出什么吗？如果你不能确定，那么就试一下：

```
System.out.print("2 + 2 = " + 2+2);
```

如果使用的是 JDK 5.0，你还可以使用

```
System.out.printf("%c%c", 'H', 'a');
```

总之，使用字符串连接操作符使用格外小心。+ 操作符当且仅当它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，它执行的就是加法。如果要连接的没有一个数值是字符串类型的，那么你可以有几种选择：

- 预置一个空字符串；
- 将第一个数值用 String.valueOf 显式地转换成一个字符串；
- 使用一个字符串缓冲区；
- 或者如果你使用的 JDK 5.0，可以用 printf 方法。

这个谜题还包含了一个给语言设计者的教训。操作符重载，即使在 Java 中只在有限的范围内得到了支持，它仍然会引起混淆。为字符串连接而重载 + 操作符可能就是一个已铸成的错误。

谜题 12：ABC

这个谜题要问的是一个悦耳的问题，下面的程序将打印什么呢？

```
public class ABC{
    public static void main(String[] args){
        String letters = "ABC";
        char[] numbers = {'1', '2', '3'};
        System.out.println(letters + " easy as " + numbers);
    }
}
```

可能大家希望这个程序打印出 ABC easy as 123。遗憾的是，它没有。如果你运行它，就会发现它打印的是诸如 ABC easy as [C@16f0472 之类的东西。为什么这个输出会如此丑陋？

尽管 char 是一个整数类型，但是许多类库都对其进行了特殊处理，因为 char 数值通常表示的是字符而不是整数。例如，将一个 char 数值传递给 println 方法会打印出一个 Unicode 字符而不是它的数字代码。字符数组受到了相同的特殊处理：println 的 char[]重载版本会打印出数组所包含的所有字符，而 String.valueOf 和 StringBuffer.append 的 char[]重载版本的行为也是类似的。

然而，字符串连接操作符在这些方法中没有被定义。该操作符被定义为先对它的两个操作数执行字符串转换，然后将产生的两个字符串连接到一起。对包括数组在内的对象引用的字符串转换定义如下[JLS 15.18.1.1]：

如果引用为 null，它将被转换成字符串"null"。否则，该转换的执行就像是不用任何参数调用该引用对象的 toString 方法一样；但是如果调用 toString 方法的结果是 null，那么就用字符串"null"来代替。

那么，在一个非空 char 数组上面调用 toString 方法会产生什么样的行为呢？数组是从 Object 那里继承的 toString 方法[JLS 10.7]，规范中描述到：“返回一个字符串，它包含了该对象所属类的名字，'@'符号，以及表示对象散列码的一个无符号十六进制整数”[Java-API]。有关 Class.getName 的规范描述到：在 char[]类型的类对象上调用该方法的结果为字符串"[C"。将它们连接到一起就形成了在我们的程序中打印出来的那个丑陋的字符串。

有两种方法可以订正这个程序。你可以在调用字符串连接操作之前，显式地将一个数组转换成一个字符串：

```
System.out.println(letters + " easy as " +
                           String.valueOf(numbers));
```

或者，你可以将 System.out.println 调用分解为两个调用，以利用 println 的 char[]重载版本：

```
System.out.print(letters + " easy as ");
System.out.println(numbers);
```

请注意，这些订正只有在你调用了 valueOf 和 println 方法正确的重载版本的情况下，才能正常运行。换句话说，它们严格依赖于数组引用的编译期类型。

下面的程序说明了这种依赖性。看起来它像是所描述的第二种订正方式的具体实现，但是它产生的输出却与最初的程序所产生的输出一样丑陋，因为它调用的是 println 的 Object 重载版本，而不是 char[]重载版本。

```
class ABC2{
   public static void main(String[] args){
       String letters = "ABC";
       Object numbers = new char[] { '1', '2', '3' };
       System.out.print(letters + " easy as ");
       System.out.println(numbers);
   }
}
```

总之，char 数组不是字符串。要想将一个 char 数组转换成一个字符串，就要调用 String.valueOf(char[])方法。某些类库中的方法提供了对 char 数组的类似字符串的支持，通常是提供一个 Object 版本的重载方法和一个 char[]版本的重载方法，而之后后者才能产生我们想要的行为。

对语言设计者的教训是：char[]类型可能应该覆写 toString 方法，使其返回数组中包含的字符。更一般地讲，数组类型可能都应该覆写 toString 方法，使其返回数组内容的一个字符串表示。

谜题 13：畜牧场

George Orwell 的《畜牧场（Animal Farm）》一书的读者可能还记得老上校的宣言：“所有的动物都是平等的。”下面的 Java 程序试图要测试这项宣言。那么，它将打印出什么呢？

```
public class AnimalFarm{
    public static void main(String[] args){
        final String pig = "length: 10";
        final String dog = "length: " + pig.length();
        System.out. println("Animals are equal: "
                            + pig == dog);
    }
}
```

对该程序的表面分析可能会认为它应该打印出 Animal are equal: true。毕竟，pig 和 dog 都是 final 的 string 类型变量，它们都被初始化为字符序列“length: 10”。换句话说，被 pig 和 dog 引用的字符串是且永远是彼此相等的。然而，== 操作符测试的是这两个对象引用是否正好引用到了相同的对象上。在本例中，它们并非引用到了相同的对象上。

你可能知道 String 类型的编译期常量是内存限定的。换句话说，任何两个 String 类型的常量表达式，如果标明的是相同的字符序列，那么它们就用相同的对象引用来表示。如果用常量表达式来初始化 pig 和 dog，那么它们确实会指向相同的对象，但是 dog 并不是用常量表达式初始化的。既然语言已经对在常量表达式中允许出现的操作作出了限制，而方法调用又不在其中，那么，这个程序就应该打印 Animal are equal: false，对吗？

嗯，实际上不对。如果你运行该程序，你就会发现它打印的只是 false，并没有其它的任何东西。它没有打印 Animal are equal: 。它怎么会不打印这个字符串字面常量呢？毕竟打印它才是正确的呀！谜题 11 的解谜方案包含了一条暗示：+ 操作符，不论是用作加法还是字符串连接操作，它都比 == 操作符的优先级高。因此，println 方法的参数是按照下面的方式计算的：

```
System.out.println(("Animals are equal: " + pig) == dog);
```

这个布尔表达式的值当然是 false，它正是该程序的所打印的输出。

有一个肯定能够避免此类窘境的方法：在使用字符串连接操作符时，总是将非平凡的操作数用括号括起来。更一般地讲，当你不能确定你是否需要括号时，应该选择稳妥地做法，将它们括起来。如果你在 println 语句中像下面这样把比较部分括起来，它将产生所期望的输出 Animals are equal: false ：

```
System.out.println("Animals are equal: " + (pig == dog));
```

可以论证，该程序仍然有问题。

如果可以的话，你的代码不应该依赖于字符串常量的内存限定机制。内存限定机制只是设计用来减少虚拟机内存占有量的，它并不是作为程序员可以使用的一种工具而设计的。就像这个谜题所展示的，哪一个表达式会产生字符串常量并非总是很显而易见。

更糟的是，如果你的代码依赖于内存限定机制实现操作的正确性，那么你就必须仔细地了解哪些域和参数必定是内存限定的。编译器不会帮你去检查这些不变量，因为内存限定的和不限定的字符串使用相同的类型（String）来表示的。这些因在内存中限定字符串失败而导致的 bug 是非常难以探测到的。

在比较对象引用时，你应该优先使用 equals 方法而不是 == 操作符，除非你需要比较的是对象的标识而不是对象的值。通过把这个教训应用到我们的程序中，我们给出了下面的 println 语句，这才是它应该具有的模样。很明显，在用这种方式订正了该程序之后，它将打印出 true：

```
System.out.println("Animals are equal: " + pig.equals(dog));
```

这个谜题对语言设计者来说有两个教训。

- 字符串连接的优先级不应该和加法一样。这意味着重载 + 操作符来执行字符串连接是有问题的，就像在谜题 11 中提到的一样。
- 还有就是，对于不可修改的类型，例如 String，其引用的等价性比值的等价性更加让人感到迷惑。也许 == 操作符在被应用于不可修改的类型时应该执行值比较。要实现这一点，一种方法是将 == 操作符作为 equals 方法的简便写法，并提供一个单独的类似于 System.identityHashCode 的方法来执行引用标识的比较。

谜题 14：转义字符的溃败

下面的程序使用了两个 Unicode 的转义字符，它们是用其十六进制代码来表示 Unicode 字符。那么，这个程序会打印什么呢？

```
public class EscapeRout{
    public static void main(String[] args){
        // \u0022 是双引号的 Unicode 转义字符
        System.out.println("a\u0022.length()
+\u0022b".length());
    }
```

```
}
```

对该程序的一种很肤浅的分析会认为它应该打印出 26，因为在由两个双引号 "a\u0022.length()+\u0022b"标识的字符串之间总共有 26 个字符。

稍微深入一点的分析会认为该程序应该打印 16，因为两个 Unicode 转义字符每一个在源文件中都需要用 6 个字符来表示，但是它们只表示字符串中的一个字符。因此这个字符串应该比它的外表看其来要短 10 个字符。 如果你运行这个程序，就会发现事情远不是这么回事。它打印的既不是 26 也不是 16，而是 2。

理解这个谜题的关键是要知道：Java 对在字符串字面常量中的 Unicode 转义字符没有提供任何特殊处理。编译器在将程序解析成各种符号之前，先将 Unicode 转义字符转换成为它们所表示的字符[JLS 3.2]。因此，程序中的第一个 Unicode 转义字符将作为一个单字符字符串字面常量（"a"）的结束引号，而第二个 Unicode 转义字符将作为另一个单字符字符串字面常量（"b"）的开始引号。程序打印的是表达式"a".length()+"b".length()，即 2。

如果该程序的作者确实希望得到这种行为，那么下面的语句将要清楚得多：

```
System.out.println("a".length()+"b".length());
```

更有可能的情况是该作者希望将两个双引号字符置于字符串字面常量的内部。使用 Unicode 转义字符你是不能实现这一点的，但是你可以使用转义字符序列来实现[JLS 3.10.6]。表示一个双引号的转义字符序列是一个反斜杠后面紧跟着一个双引号（\”）。如果将最初的程序中的 Unicode 转义字符用转义字符序列来替换，那么它将打印出所期望的 16：

```
System.out.println("a\".length()+\"b".length());
```

许多字符都有相应的转义字符序列，包括单引号（\’）、换行（\n）、制表符（\t）和反斜线（\\）。你可以在字符字面常量和字符串字面常量中使用转义字符序列。

实际上，你可以通过使用被称为八进制转义字符的特殊类型的转义字符序列，将任何 ASCII 字符置于一个字符串字面常量或一个字符字面常量中，但是最好是尽可能地使用普通的转义字符序列。

普通的转义字符序列和八进制转义字符都比 Unicode 转义字符要好得多，因为与 Unicode 转义字符不同，转义字符序列是在程序被解析为各种符号之后被处理的。

ASCII 是字符集的最小公共特性集，它只有 128 个字符，但是 Unicode 有超过 65,000 个字符。一个 Unicode 转义字符可以被用来在只使用 ASCII 字符的程序中插入一个 Unicode 字符。一个 Unicode 转义字符精确地等价于它所表示的字符。

Unicode 转义字符被设计为用于在程序员需要插入一个不能用源文件字符集表示的字符的情况。它们主要用于将非 ASCII 字符置于标识符、字符串字面常量、字符字面常量以及注释中。偶尔地，Unicode 转义字符也被用来在看起来颇为相似的数个字符中明确地标识其中的某一个，从而增加程序的清晰度。

总之，在字符串和字符字面常量中要优先选择的是转义字符序列，而不是Unicode转义字符。Unicode转义字符可能会因为它们在编译序列中被处理得过早而引起混乱。不要使用Unicode转义字符来表示ASCII字符。在字符串和字符字面常量中，应该使用转义字符序列；对于除这些字面常量之外的情况，应该直接将ASCII字符插入到源文件中。

谜题15：令人晕头转向的Hello

下面的程序是对一个老生常谈的例子做出了稍许的变化之后的版本。那么，它会打印出什么呢？

```
/**
 * Generated by the IBM IDL-to-Java compiler, version 1.0
 * from F:\TestRoot\apps\a1\units\include\PolicyHome.idl
 * Wednesday, June 17, 1998 6:44:40 o' clock AM GMT+00:00
 */
public class Test{
    public static void main(String[] args){
        System.out.print("Hell");
        System.out.println("o world");
    }
}
```

这个谜题看起来相当简单。该程序包含了两条语句，第一条打印Hell，而第二条在同一行打印o world，从而将两个字符串有效地连接在了一起。因此，你可能期望该程序打印出Hello world。但是很可惜，你犯了错，实际上，它根本就通不过编译。

问题在于注释的第三行，它包含了字符\units。这些字符以反斜杠（\）以及紧跟着的字母u开头的，而它（\u）表示的是一个Unicode转义字符的开始。遗憾的是，这些字符后面没有紧跟四个十六进制的数字，因此，这个Unicode转义字符是病构的，而编译器则被要求拒绝该程序。Unicode转义字符必须是良构的，即使是出现在注释中也是如此。

在注释中插入一个良构的Unicode转义字符是合法的，但是我们几乎没有什么理由去这么做。程序员有时会在JavaDoc注释中使用Unicode转义字符来在文档中生成特殊的字符。

```
// Unicode转义字符在JavaDoc注释中有问题的用法
/**
 * This method calls itself recursively, causing a
 * StackOverflowError to be thrown.
 * The algorithm is due to Peter von der Ah\u00E9.
 */
```

这项技术表示了Unicode转义字符的一种没什么用处的用法。在Javadoc注释中，应该使用HTML实体转义字符来代替Unicode转义字符：

```
/**
 * This method calls itself recursively, causing a
 * StackOverflowError to be thrown.
 * The algorithm is due to Peter von der Ahé.
 */
```

前面的两个注释都应该是的在文档中出现的名字为“Peter der Ahé”，但是后一个注释在源文件中还是可理解的。

可能你会感到很诧异，在这个谜题中，问题出在注释这一信息来源自一个实际的 bug 报告。该程序是机器生成的，这使得我们很难追踪到问题的源头——IDL-to-Java 编译器。为了避免让其他程序员也陷入此境地，在没有将 Windows 文件名进行预先处理，以消除的其中的反斜杠的情况下，工具应该确保不将 Windows 文件名置于所生成的 Java 源文件的注释中。

总之，要确保字符\u 不出现在一个合法的 Unicode 转义字符上下文之外，即使是在注释中也是如此。在机器生成的代码中要特别注意此问题。

谜题 16：行打印程序

行分隔符（line separator）是为用来分隔文本行的字符或字符组合而起的名字，并且它在不同的平台上是存在差异的。在 Windows 平台上，它是 CR 字符（回车）和紧随其后的 LF 字符（换行）组成的，而在 UNIX 平台上，通常单独的 LF 字符被当作换行字符来引用。下面的程序将这个字符传递给了 println 方法，那么，它将打印出什么呢？它的行为是否是依赖于平台的呢？

```
public class LinePrinter{
    public static void main(String[] args){
    // Note: \u000A is Unicode representation of linefeed (LF)
        char c = 0x000A;
        System.out.println(c);
    }
}
```

这个程序的行为是平台无关的：它在任何平台上都不能通过编译。如果你尝试着去编译它，就会得到类似下面的出错信息：

```
LinePrinter.java:3: ';' expected
// Note: \u000A is Unicode representation of linefeed (LF)
^
1 error
```

如果你和大多数人一样，那么这条信息对界定问题是毫无用处的。

这个谜题的关键就是程序第三行的注释。与最好的注释一样，这条注释也是一种准确的表达，遗憾的是，它有一点准确得过头了。编译器不仅会在将程序解析成为符号之前把 Unicode 转义字符转换成它们所表示的字符（谜题 14），而且它是在丢弃注释和空格之前做这些事的[JLS 3.2]。

这个程序包含了一个 Unicode 转移字符（\u000A），它位于程序唯一的注释行中。就像注释所陈述的，这个转义字符表示换行符，编译器将在丢弃注释之前适时地转换它。遗憾的是，这个换行符是表示注释开始的两个斜杠符之后的第一个行终结符（line terminator），因此它将终结该注释[JLS 3.4]。所以，该转义字符之后的字（is Unicode representation of linefeed (LF)）就不是注释的一部分了，而它们在语法上也不是有效的。

订正该程序的最简单的方式就是在注释中移除 Unicode 转义字符，但是更好的方式是用一个转义字符序列而不是一个十六进制整型字面常量来初始化 c，从而消除使用注释的必要：

```
public class LinePrinter{
    public static void main(String[] args){
        char c = '\n';
        System.out.println(c);
    }
}
```

只要这么做了，程序就可以编译并运行，但是这仍然是一个有问题的程序：它是平台相关的，这正是本谜题所要表达的真正意图。在某些平台上，例如 UNIX，它将打印出两个完整的行分隔符；但是在其它一些平台上，例如 Windows，它就不会产生这样的行为。尽管这些输出用肉眼看起来是一样的，但是如果它们要被存储到文件中，或是输出到后续的其它处理程序中，那就很容易引发问题。

如果你想打印两行空行，你应该调用 println 两次。如果使用的是 JDK 5.0，那么你可以用带有格式化字符串"%n%n"的 printf 来代替 println。%n 的每一次出现都将导致 printf 打印一个恰当的、与平台相关的行分隔符。

我们希望，上面三个谜题已经使你信服：Unicode 转义字符绝对会产生混乱。教训很简单：除非确实是必需的，否则就不要使用 Unicode 转义字符。它们很少是必需的。

谜题 17：嗯？

下面的是一个合法的 Java 程序吗？如果是，它会打印出什么呢？

```
\u0070\u0075\u0062\u006c\u0069\u0063\u0020\u0020\u0020\u0020
\u0063\u006c\u0061\u0073\u0073\u0020\u0055\u0067\u006c\u0079
\u007b\u0070\u0075\u0062\u006c\u0069\u0063\u0020\u0020\u0020
\u0020\u0020\u0020\u0020\u0073\u0074\u0061\u0074\u0069\u0063
\u0076\u006f\u0069\u0064\u0020\u006d\u0061\u0069\u006e\u0028
\u0053\u0074\u0072\u0069\u006e\u0067\u005b\u005d\u0020\u0020
\u0020\u0020\u0020\u0020\u0061\u0072\u0067\u0073\u0029\u007b
\u0053\u0079\u0073\u0074\u0065\u006d\u002e\u006f\u0075\u0074
\u002e\u0070\u0072\u0069\u006e\u0074\u006c\u006e\u0028\u0020
\u0022\u0048\u0065\u006c\u006c\u006f\u0020\u0077\u0022\u002b
```

```
\u0022\u006f\u0072\u006c\u0064\u0022\u0029\u003b\u007d\u007d
```

这当然是一个合法的 Java 程序！这不是很显而易见吗？它会打印 Hello World。噢，可能是不那么明显。事实上，该程序根本让人无法理解。每当你没必要地使用了一个 Unicode 转义字符时，都会使你的程序的可理解性更缺失一点，而该程序将这种做法发挥到了极致。如果你很好奇，可以看看下面给出的该程序在 Unicode 转义字符都被转换为它们所表示的字符之后的样子：

```
public
class Ugly
{public
static
void main(
String[]
args){
System.out
.println(
 “Hello w” +
 “orld” );}}
```

下面给出了将其进行格式化整理之后的样子：

```
public class Ugly {
  public static void main(String[] args){
     System.out.println("Hello w"+"orld");
  }
}
```

这个谜题的教训是：仅仅是因为你可以不以应有的方式去进行表达。或者说，如果你这么做会造成损害，那么就请不要这么做！更严肃地讲，这个谜题是对前面三个教训的补充：Unicode 转义字符只有在你要向程序中插入用其他任何方式都无法表示的字符时才是必需的，除此之外的任何情况都不应该避免使用它们。Unicode 转义字符降低了程序的清晰度，并且增加了产生 bug 的可能性。

对语言的设计者来说，也许使用 Unicode 转义字符来表示 ASCII 字符应该被定义为是非法的。这样就可以使得在谜题 14、15 和 17（本谜题）中的程序非法，从而消除了大量的混乱。这个限制对程序员并不会造成任何困难。

谜题 18：字符串奶酪

下面的程序从一个字节序列创建了一个字符串，然后迭代遍历字符串中的字符，并将它们作为数字打印。请描述一下程序打印出来的数字序列：

```
public class StringCheese {
    public static void main(String[] args) {
        byte bytes[] = new byte[256];
        for (int i = 0; i < 256; i++)
             bytes[i] = (byte)i;
        String str = new String(bytes);
        for (int i = 0, n = str.length(); i < n; i++)
```

```
            System.out.println((int)str.charAt(i) + " ");
    }
}
```

首先，byte 数组用从 0 到 255 每一个可能的 byte 数值进行了初始化，然后这些 byte 数值通过 String 构造器被转换成了 char 数值。最后，char 数值被转型为 int 数值并被打印。打印出来的数值肯定是非负整数，因为 char 数值是无符号的，因此，你可能期望该程序将按顺序打印出 0 到 255 的整数。

如果你运行该程序，可能会看到这样的序列。但是在运行一次，可能看到的就不是这个序列了。我们在四台机器上运行它，会看到四个不同的序列，包括前面描述的那个序列。这个程序甚至都不能保证会正常终止，比打印其他任何特定字符串都要缺乏这种保证。它的行为完全是不确定的。

这里的罪魁祸首就是 String(byte[])构造器。有关它的规范描述道：“在通过解码使用平台缺省字符集的指定 byte 数组来构造一个新的 String 时，该新 String 的长度是字符集的一个函数，因此，它可能不等于 byte 数组的长度。当给定的所有字节在缺省字符集中并非全部有效时，这个构造器的行为是不确定的”[Java-API]。

到底什么是字符集？从技术角度上讲，它是“被编码的字符集合和字符编码模式的结合物”[Java-API]。换句话说，字符集是一个包，包含了字符、表示字符的数字编码以及在字符编码序列和字节序列之间来回转换的方式。转换模式在字符集之间存在着很大的区别：某些是在字符和字节之间做一对一的映射，但是大多数都不是这样。ISO-8859-1 是唯一能够让该程序按顺序打印从 0 到 255 的整数的缺省字符集，它更为大家所熟知的名字是 Latin-1[ISO-8859-1]。

J2SE 运行期环境（JRE）的缺省字符集依赖于底层的操作系统和语言。如果你想知道你的 JRE 的缺省字符集，并且你使用的是 5.0 或更新的版本，那么你可以通过调用 java.nio.charset.Charset.defaultCharset()来了解。如果你使用的是较早的版本，那么你可以通过阅读系统属性“file.encoding”来了解。

幸运的是，你没有被强制要求必须去容忍各种稀奇古怪的缺省字符集。当你在 char 序列和 byte 序列之间做转换时，你可以且通常是应该显式地指定字符集。除了接受 byte 数字之外，还可以接受一个字符集名称的 String 构造器就是专为此目的而设计的。如果你用下面的构造器去替换在最初的程序中的 String 构造器，那么不管缺省的字符集是什么，该程序都保证能够按照顺序打印从 0 到 255 的整数：

```
String str = new String(bytes, "ISO-8859-1");
```

这个构造器声明会抛出 UnsupportedEncodingException 异常，因此你必须捕获它，或者更适宜的方式是声明 main 方法将抛出它，要不然程序不能通过编译。尽管如此，该程序实际上不会抛出异常。Charset 的规范要求 Java 平台的每一种实现都要支持某些种类的字符集，ISO-8859-1 就位列其中。

这个谜题的教训是：每当你要将一个 byte 序列转换成一个 String 时，你都在使用某一个字符集，不管你是否显式地指定了它。如果你想让你的程序的行为是可预知的，那么就请你在每次使用字符集时都明确地指定。对 API 的设计者来说，提供这么一个依赖于缺省字符集的 String(byte[])构造器可能并非是一个好主意。

谜题 19：漂亮的火花

下面的程序用一个方法对字符进行了分类。这个程序会打印出什么呢？

```
public class Classifier {
    public static void main(String[] args) {
        System.out.println(
             classify('n') + classify('+') + classify('2'));
    }
    static String classify(char ch) {
        if ("0123456789".indexOf(ch) >= 0)
             return "NUMERAL ";
        if ("abcdefghijklmnopqrstuvwxyz".indexOf(ch) >= 0)
             return "LETTER ";
        /* (Operators not supported yet)
            if ("+-*/&|!=" >= 0)
                 return "OPERATOR ";
        */
        return "UNKNOWN";
    }
}
```

如果你猜想该程序将打印 LETTER UNKNOWN NUMERAL，那么你就掉进陷阱里面了。这个程序连编译都通不过。让我们再看一看相关的部分，这一次我们用粗体字突出注释部分：

```
if ("abcdefghijklmnopqrstuvwxyz".indexOf(ch) >= 0)
             return "LETTER ";
        /* (Operators not supported yet)
        if ("+-*/&|!=" >= 0)
                 return "OPERATOR ";
        */
        return "UNKNOWN";
    }
}
```

正如你之所见，注释在包含了字符*/的字符串内部就结束了，结果使得程序在语法上变成非法的了。我们将程序中的一部分注释出来的尝试之所以失败了，是因为字符串字面常量在注释中没有被特殊处理。

更一般地讲，注释内部的文本没有以任何方式进行特殊处理[JLS 3.7]。因此，块注释不能嵌套。请考虑下面的代码段：

```
/* Add the numbers from 1 to n */
int sum = 0;
for (int i = 1; I <= n; i++)
sum += i;
```
现在假设我们要将该代码段注释成为一个块注释，我们再次用粗体字突出整个注释：
```
/*
/* Add the numbers from 1 to n */
int sum = 0;
for (int i = 1; I <= n; i++)
sum += i;
*/
```
正如你之所见，我们没有能够将最初的代码段注释掉。好在所产生的代码包含了一个语法错误，因此编译器将会告诉我们代码存在着问题。

你可能偶尔看到过这样的代码段，它被一个布尔表达式为常量 false 的 if 语句禁用了：

```
//code commented out with an if statement - doesn't always work!
if (false) {
     /* Add the numbers from 1 to n */
     int sum = 0;
     for (int i = 1; i <= n; i++)
            sum += i;
}
```
语言规范建议将这种方式作为一种条件编译技术[JLS 14.21]，但是它不适合用来注释代码。除非要被禁用的代码是一个合法的语句序列，否则就不要使用这项技术。

注释掉一个代码段的最好的方式是使用单行的注释序列。大多数 IDE 工具都可以自动化这个过程：

```
//code commented out with an if statement - doesn't always work!
//     /* Add the numbers from 1 to n */
//     int sum = 0;
//     for (int i = 1; i <= n; i++)
//            sum += i;
```
总之，块注释不能可靠地注释掉代码段，应该用单行的注释序列来代替。对语言设计者来说，应该注意到可嵌套的块注释并不是一个好主意。他们强制编译器去解析块注释内部的文本，而由此引发的问题比它能够解决的问题还要多。

谜题 20：我的类是什么？

下面的程序被设计用来打印它的类文件的名称。如果你不熟悉类字面常量，那么我告诉你 Me.class.getName()将返回 Me 类完整的名称，即“com.javapuzzlers.Me”。那么，这个程序会打印出什么呢？

```
package com.javapuzzlers;
public class Me {
    public static void main(String[] args){
        System.out.println(
             Me.class.getName().
                replaceAll(".","/") + ".class");
    }
}
```

该程序看起来会获得它的类名（“com.javapuzzlers.Me”），然后用“/”替换掉所有出现的字符串“.”，并在末尾追加字符串“.class”。你可能会认为该程序将打印 com/javapuzzlers/Me.class，该程序正式从这个类文件中被加载的。如果你运行这个程序，就会发现它实际上打印的是 //////////////////.class。到底怎么回事？难道我们是斜杠的受害者吗？

问题在于 String.replaceAll 接受了一个正则表达式作为它的第一个参数，而并非接受了一个字符序列字面常量。（正则表达式已经被添加到了 Java 平台的 1.4 版本中。）正则表达式“.”可以匹配任何单个的字符，因此，类名中的每一个字符都被替换成了一个斜杠，进而产生了我们看到的输出。

要想只匹配句点符号，在正则表达式中的句点必须在其前面添加一个反斜杠（\）进行转义。因为反斜杠字符在字面含义的字符串中具有特殊的含义——它标识转义字符序列的开始——因此反斜杠自身必须用另一个反斜杠来转义，这样就可以产生一个转义字符序列，它可以在字面含义的字符串中生成一个反斜杠。把这些合在一起，就可以使下面的程序打印出我们所期望的 com/javapuzzlers/Me.class：

```
package com.javapuzzlers;
public class Me {
    public static void main(String[] args){
        System.out.println(
            Me.class.getName().replaceAll("\\.","/") + ".class");
    }
}
```

为了解决这类问题，5.0 版本提供了新的静态方法 java.util.regex.Pattern.quote。它接受一个字符串作为参数，并可以添加必需的转义字符，它将返回一个正则表达式字符串，该字符串将精确匹配输入的字符串。下面是使用该方法之后的程序：

```
package com.javapuzzlers;
import java.util.regex.Pattern;
public class Me {
    public static void main(String[] args){
        System.out.println(Me.class.getName().
```

```
                    replaceAll(Pattern.quote("."),"/") + ".class");
    }
}
```

该程序的另一个问题是：其正确的行为是与平台相关的。并不是所有的文件系统都使用斜杠符号来分隔层次结构的文件名组成部分的。要想获取一个你正在运行的平台上的有效文件名，你应该使用正确的平台相关的分隔符号来代替斜杠符号。这正是下一个谜题所要做的。

谜题 21：我的类是什么？ II

下面的程序所要做的事情正是前一个谜题所做的事情，但是它没有假设斜杠符号就是分隔文件名组成部分的符号。相反，该程序使用的是 java.io.File.separator，它被指定为一个公共的 String 域，包含了平台相关的文件名分隔符。那么，这个程序会打印出其正确的、平台相关的类文件名吗？

```
package com.javapuzzlers;
import java.io.File;
public class MeToo {
    public static void main(String[] args){
        System.out.println(MeToo.class.getName().
                    replaceAll("\\.", File.separator) + ".class");
    }
}
```

这个程序根据底层平台的不同会显示两种行为中的一种。如果文件分隔符是斜杠，就像在 UNIX 上一样，那么该程序将打印 com/javapuzzlers/MeToo.class，这是正确的。但是，如果文件分隔符是反斜杠，就像在 Windows 上一样，那么该程序将打印像下面这样的内容：

```
Exception in thread "main"
java.lang.StringIndexOutOfBoundsException: String index out of range: 1
        at java.lang.String.charAt(String.java:558)
        at java.util.regex.Matcher.appendReplacement(Mather.
java:696)
        at java.util.regex.Matcher.replaceAll(Mather.java:806)
        at java.lang.String.replaceAll(String.java:2000)
        at com.javapuzzlers.MeToo.main(MeToo.java:6)
```

尽管这种行为是平台相关的，但是它并非就是我们所期待的。在 Windows 上出了什么错呢？

事实证明，String.replaceAll 的第二个参数不是一个普通的字符串，而是一个替代字符串（replacement string），就像在 java.util.regex 规范中所定义的那样[Java-API]。在替代字符串中出现的反斜杠会把紧随其后的字符进行转义，从而导致其被按字面含义而处理了。

当你在 Windows 上运行该程序时，替代字符串是单独的一个反斜杠，它是无效的。不可否认，抛出的异常应该提供更多一些有用的信息。

那么你应该怎样解决此问题呢？5.0 版本提供了不是一个而是两个新的方法来解决它。第一个方法是 java.util.regex.Matcher.quoteReplacement，它将字符串转换成相应的替代字符串。下面展示了如何使用这个方法来订正该程序：

```
System.out.println(MeToo.class.getName().replaceAll("\\.",
        Matcher.quoteReplacement(File.separator)) + ".class");
```

引入到 5.0 版本中的第二个方法提供了一个更好的解决方案。该方法就是 String.replace(CharSequence, CharSequence)，它做的事情和 String.replaceAll 相同，但是它将模式和替代物都当作字面含义的字符串处理。下面展示了如何使用这个方法来订正该程序：

```
System.out.println(MeToo.class.getName().
        replace(".", File.separator) + ".class");
```

但是如果你使用的是较早版本的 Java 该怎么办？很遗憾，没有任何捷径能够生成替代字符串。完全不使用正则表达式，而使用 String.replace(char,char)也许要显得更容易一些：

```
System.out.println(MeToo.class.getName().
        replace('.', File.separatorChar) + ".class");
```

本谜题和前一个谜题的主要教训是：在使用不熟悉的类库方法时一定要格外小心。当你心存疑虑时，就要求助于 Javadoc。还有就是正则表达式是很棘手的：它所引发的问题趋向于在运行时刻而不是在编译时刻暴露出来。

对 API 的设计者来说，使用方法具名的模式来以明显的方式区分方法行为的差异是很重要的。Java 的 String 类就没有很好地遵从这一原则。对许多程序员来说，对于哪些字符串替代方法使用的是字面含义的字符串，以及哪些使用的是正则表达式或替代字符串，要记住这些都不是一件容易事。

谜题 22：URL 的愚弄

本谜题利用了 Java 编程语言中一个很少被人了解的特性。请考虑下面的程序将会做些什么？

```
public class BrowserTest {
    public static void main(String[] args) {
        System.out.print("iexplore:");
        http://www.google.com;
        System.out.println(":maximize");
    }
}
```

这是一个有点诡异的问题。该程序将不会做任何特殊的事情，而是直接打印 iexplore::maximize。在程序中间出现的 URL 是一个语句标号（statement label）[JLS 14.7]后面跟着一行行尾注释（end-of-line comment）[JLS 3.7]。在 Java 中很少需要标号，这多亏了 Java 没有 goto 语句。在本谜题中所引用的“Java 编程语言中很少被人了解的特性”实际上就是你可以在任何语句前面放置标号。这个程序标注了一个表达式语句，它是合法的，但是却没什么用处。

它的价值所在，就是提醒你，如果你真的想要使用标号，那么应该用一种更合理的方式来格式化程序：

```
public class BrowserTest {
    public static void main(String[] args) {
        System.out.print("iexplore:");
    http:      //www.google.com;
        System.out.println(":maximize");
    }
}
```

这就是说，我们没有任何可能的理由去使用与程序没有任何关系的标号和注释。

本谜题的教训是：令人误解的注释和无关的代码会引起混乱。要仔细地写注释，并让它们跟上时代；要切除那些已遭废弃的代码。还有就是如果某些东西看起来过于奇怪，以至于不像对的，那么它极有可能就是错的。

谜题 23：不劳无获

下面的程序将打印一个单词，其第一个字母是由一个随机数生成器来选择的。请描述该程序的行为：

```
import java.util.Random;
public class Rhymes {
   private static Random rnd = new Random();
   public static void main(String[] args) {
      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);
   }
}
```

乍一看，这个程序可能会在一次又一次的运行中，以相等的概率打印出 Pain，Gain 或 Main。看起来该程序会根据随机数生成器所选取的值来选择单词的第一个字母：0 选 M，1 选 P，2 选 G。谜题的题目也许已经给你提供了线索，它实际上既不会打印 Pain，也不会打印 Gain。也许更令人吃惊的是，它也不会打印 Main，并且它的行为不会在一次又一次的运行中发生变化，它总是在打印 ain。

有三个 bug 凑到一起引发了这种行为。你完全没有发现它们吗？第一个 bug 是所选取的随机数使得 switch 语句只能到达其三种情况中的两种。

Random.nextInt(int)的规范描述道：“返回一个伪随机的、均等地分布在从 0（包括）到指定的数值（不包括）之间的一个 int 数值”[Java-API]。这意味着表达式 rnd.nextInt(2)可能的取值只有 0 和 1，Switch 语句将永远也到不了 case 2 分支，这表示程序将永远不会打印 Gain。nextInt 的参数应该是 3 而不是 2。

这是一个相当常见的问题源，被熟知为“栅栏柱错误（fencepost error）”。这个名字来源于对下面这个问题最常见的但却是错误的答案，如果你要建造一个 100 英尺长的栅栏，其栅栏柱间隔为 10 英尺，那么你需要多少根栅栏柱呢？11 根或 9 根都是正确答案，这取决于是否要在栅栏的两端树立栅栏柱，但是 10 根却是错误的。要当心栅栏柱错误，每当你在处理长度、范围或模数的时候，都要仔细确定其端点是否应该被包括在内，并且要确保你的代码的行为要与其相对应。

第二个 bug 是在不同的情况（case）中没有任何 break 语句。不论 switch 表达式为何值，该程序都将执行其相对应的 case 以及所有后续的 case[JLS 14.11]。因此，尽管每一个 case 都对变量 word 赋了一个值，但是总是最后一个赋值胜出，覆盖了前面的赋值。最后一个赋值将总是最后一种情况（default），即 new StringBuffer{'M'}。这表明该程序将总是打印 Main，而从来不打印 Pain 或 Gain。

在 switch 的各种情况中缺少 break 语句是非常常见的错误。从 5.0 版本起，javac 提供了-Xlint:fallthrough 标志，当你忘记在一个 case 与下一个 case 之间添加 break 语句是，它可以生成警告信息。不要从一个非空的 case 向下进入了另一个 case。这是一种拙劣的风格，因为它并不常用，因此会误导读者。十次中有九次它都会包含错误。如果 Java 不是模仿 C 建模的，那么它倒是有可能不需要 break。对语言设计者的教训是：应该考虑提供一个结构化的 switch 语句。

最后一个，也是最微妙的一个 bug 是表达式 new StringBuffer('M')可能没有做哪些你希望它做的事情。你可能对 StringBuffer(char)构造器并不熟悉，这很容易解释：它压根就不存在。StringBuffer 有一个无参数的构造器，一个接受一个 String 作为字符串缓冲区初始内容的构造器，以及一个接受一个 int 作为缓冲区初始容量的构造器。在本例中，编译器会选择接受 int 的构造器，通过拓宽原始类型转换把字符数值'M'转换为一个 int 数值 77[JLS 5.1.2]。换句话说，new StringBuffer('M')返回的是一个具有初始容量 77 的空的字符串缓冲区。该程序余下的部分将字符 a、i 和 n 添加到了这个空字符串缓冲区中，并打印出该字符串缓冲区那总是 ain 的内容。

为了避免这类问题，不管在什么时候，都要尽可能使用熟悉的惯用法和 API。如果你必须使用不熟悉的 API，那么请仔细阅读其文档。在本例中，程序应该使用常用的接受一个 String 的 StringBuffer 构造器。

下面是该程序订正了这三个 bug 之后的正确版本，它将以均等的概率打印 Pain、Gain 和 Main：

```
import java.util.Random;
public class Rhymes1 {
```

```
    private static Random rnd = new Random();
    public static void main(String[] args) {
        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;
        }
        word.append('a');
        word.append('i');
        word.append('n');
        System.out.println(word);
    }
}
```

尽管这个程序订正了所有的 bug，它还是显得过于冗长了。下面是一个更优雅的版本：

```
import java.util.Random;
public class Rhymes2 {
    private static Random rnd = new Random();
    public static void main(String[] args) {
        System.out.println("PGM".charAt(rnd.nextInt(3)) + "ain");
    }
}
```

下面是一个更好的版本。尽管它稍微长了一点，但是它更加通用。它不依赖于所有可能的输出只是在它们的第一个字符上有所不同的这个事实：

```
import java.util.Random;
public class Rhymes3 {
    public static void main(String[] args) {
        String a[] = {"Main","Pain","Gain"};
        System.out.println(randomElement(a));
    }
    private static Random rnd = new Random();
    private static String randomElement(String[] a){
        return a[rnd.nextInt(a.length)];
    }
}
```

总结一下：首先，要当心栅栏柱错误。其次，牢记在 switch 语句的每一个 case 中都放置一条 break 语句。第三，要使用常用的惯用法和 API，并且当你在离

开老路子的时候，一定要参考相关的文档。第四，一个 char 不是一个 String，而是更像一个 int。最后，要提防各种诡异的谜题。

Java 谜题 3——循环谜题

谜题 24：尽情享受每一个字节

下面的程序循环遍历 byte 数值，以查找某个特定值。这个程序会打印出什么呢？

```
public class BigDelight {
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE; b < Byte.MAX_VALUE; b++) {
            if (b == 0x90)
                System.out.print("Joy!");
        }
    }
}
```

这个循环在除了 Byte.MAX_VALUE 之外所有的 byte 数值中进行迭代，以查找 0x90。这个数值适合用 byte 表示，并且不等于 Byte.MAX_VALUE，因此你可能会想这个循环在该迭代会找到它一次，并将打印出 Joy!。但是，所见为虚。如果你运行该程序，就会发现它没有打印任何东西。怎么回事？

简单地说，0x90 是一个 int 常量，它超出了 byte 数值的范围。这与直觉是相悖的，因为 0x90 是一个两位的十六进制字面常量，每一个十六进制位都占据 4 个比特的位置，所以整个数值也只占据 8 个比特，即 1 个 byte。问题在于 byte 是有符号类型。常量 0x90 是一个正的最高位被置位的 8 位 int 数值。合法的 byte 数值是从-128 到+127，但是 int 常量 0x90 等于+144。

拿一个 byte 与一个 int 进行的比较是一个混合类型比较（mixed-type comparison）。如果你把 byte 数值想象为苹果，把 int 数值想象成为桔子，那么该程序就是在拿苹果与桔子比较。请考虑表达式((byte)0x90 == 0x90)，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 通过拓宽原始类型转换将 byte 提升为一个 int[JLS 5.1.2]，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 数值。在本例中，该转换将(byte)0x90 提升为 int 数值-112，它不等于 int 数值 0x90，即+144。

由于系统总是强制地将一个操作数提升到与另一个操作数相匹配的类型，所以混合类型比较总是容易把人搞糊涂。这种转换是不可视的，而且可能不会产生你所期望的结果。有若干种方法可以避免混合类型比较。我们继续有关水果的比喻，

你可以选择拿苹果与苹果比较，或者是拿桔子与桔子比较。你可以将 int 转型为 byte，之后你就可以拿一个 byte 与另一个 byte 进行比较了：

```
if (b == (byte)0x90)
    System.out.println("Joy!");
```

或者，你可以用一个屏蔽码来消除符号扩展的影响，从而将 byte 转型为 int，之后你就可以拿一个 int 与另一个 int 进行比较了：

```
if ((b & 0xff) == 0x90)
    System.out.print("Joy!");
```

上面的两个解决方案都可以正常运行，但是避免这类问题的最佳方法还是将常量值移出到循环的外面，并将其在一个常量声明中定义它。下面是我们对此作出的第一个尝试：

```
public class BigDelight {
    private static final byte TARGET = 0x90;
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE; b <
             Byte.MAX_VALUE; b++) {
            if (b == TARGET)
                System.out.print("Joy!");
        }
    }
}
```

遗憾的是，它根本就通不过编译。常量声明有问题，编译器会告诉你问题所在：0x90 对于 byte 类型来说不是一个有效的数值。如果你想下面这样订正该声明，那么程序将运行得非常好：

```
private static final byte TARGET = (byte)0x90;
```

总之，要避免混合类型比较，因为它们内在地容易引起混乱（谜题 5）。为了帮助实现这个目标，请使用声明的常量替代“魔幻数字”。你已经了解了这确实是一个好主意：它说明了常量的含义，集中了常量的定义，并且根除了重复的定义。现在你知道它还可以强制你去为每一个常量赋予适合其用途的类型，从而消除了产生混合类型比较的一种根源。

对语言设计的教训是 byte 数值的符号扩展是产生 bug 和混乱的一种常见根源。而用来抵销符号扩展效果所需的屏蔽机制会使得程序显得混乱无序，从而降低了程序的可读性。因此，byte 类型应该是无符号的。还可以考虑为所有的原始类型提供定义字面常量的机制，这可以减少对易于产生错误的类型转换的需求（谜题 27）。

谜题 25：无情的增量操作

下面的程序对一个变量重复地进行增量操作，然后打印它的值。那么这个值是什么呢？

```
public class Increment {
    public static void main(String[] args) {
```

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

乍一看，这个程序可能会打印 100。毕竟，它对 j 做了 100 次增量操作。可能会令你感到有些震惊，它打印的不是 100 而是 0。所有的增量操作都无影无踪了，为什么？

就像本谜题的题目所暗示的，问题出在了执行增量操作的语句上：

```
j = j++;
```

大概该语句的作者是想让它执行对 j 的值加 1 的操作，也就是表达式 j++所做的操作。遗憾的是，作者大咧咧地将这个表达式的值有赋回给了 j。

当++操作符被置于一个变量值之后时，其作用就是一个后缀增量操作符（postfix increment operator）[JLS 15.14.2]：表达式 j++的值等于 j 在执行增量操作之前的初始值。因此，前面提到的赋值语句首先保存 j 的值，然后将 j 设置为其值加 1，最后将 j 复位到它的初始值。换句话说，这个赋值操作等价于下面的语句序列：

```
int tmp = j;
j = j + 1;
j = tmp?;
```

程序重复该过程 100 次，之后 j 的值还是等于它在循环开始之前的值，即 0。

订正该程序非常简单，只需从循环中移除无关的赋值操作，只留下：

```
for (int i = 0; i < 100; i++)
      j++;
```

经过这样的修改，程序就可以打印出我们所期望的 100 了。

这与谜题 7 中的教训相同：不要在单个的表达式中对相同的变量赋值超过一次。对相同的变量进行多次赋值的表达式会产生混淆，并且很少能够产生你希望的行为。

谜题 26：在循环中

下面的程序计算了一个循环的迭代次数，并且在该循环终止时将这个计数值打印了出来。那么，它打印的是什么呢？

```
public class InTheLoop {
    public static final int END = Integer.MAX_VALUE;
    public static final int START = END - 100;
```

```
    public static void main(String[] args) {
        int count = 0;
        for (int i = START; i <= END; i++)
            count++;
        System.out.println(count);
    }
}
```

如果你没有非常仔细地查看这个程序，你可能会认为它将打印 100，因为 END 比 START 大 100。如果你稍微仔细一点，你可能会发现该程序没有使用典型的循环惯用法。大多数的循环会在循环索引小于终止值时持续运行，而这个循环则是在循环索引小于或等于终止值时持续运行。所以它会打印 101，对吗？

嗯，根本不对。如果你运行该程序，就会发现它压根就什么都没有打印。更糟的是，它会持续运行直到你撤销它为止。它从来都没有机会去打印 count，因为在打印它的语句之前插入的是一个无限循环。

问题在于这个循环会在循环索引（i）小于或等于 Integer.MAX_VALUE 时持续运行，但是所有的 int 变量都是小于或等于 Integer.MAX_VALUE 的。因为它被定义为所有 int 数值中的最大值。当 i 达到 Integer.MAX_VALUE，并且再次被执行增量操作时，它就有绕回到了 Integer.MIN_VALUE。

如果你需要的循环会迭代到 int 数值的边界附近时，你最好是使用一个 long 变量作为循环索引。只需将循环索引的类型从 int 改变为 long 就可以解决该问题，从而使程序打印出我们所期望的 101：

```
for (long i = START; i <= END; i++)
```

更一般地讲，这里的教训就是 int 不能表示所有的整数。无论你在何时使用了一个整数类型，都要意识到其边界条件。如果其数值下溢或是上溢了，会怎么样呢？所以通常最好是使用一个取之范围更大的类型。（整数类型包括 byte、char、short、int 和 long。）

不使用 long 类型的循环索引变量也可以解决该问题，但是它看起来并不那么漂亮：

```
int i = START;
do {
    count++;
}while (i++ != END);
```

如果清晰性和简洁性占据了极其重要的地位，那么在这种情况下使用一个 long 类型的循环索引几乎总是最佳方案。

但是有一个例外：如果你在所有的（或者几乎所有的）int 数值上迭代，那么使用 int 类型的循环索引的速度大约可以提高一倍。下面是将 f 函数作用于所有 40 亿个 int 数值上的惯用法：

```
//Apply the function f to all four billion int values
int i = Integer.MIN_VALUE;
do {
    f(i);
}while (i++ != Integer.MAX_VALUE);
```

该谜题对语言设计者的教训与谜题 3 相同：可能真的值得去考虑，应该对那些不会在产生溢出时而不抛出异常的算术运算提供支持。同时，可能还值得去考虑，应该对那些在整数值范围之上进行迭代的循环进行特殊设计，就像许多其他语言所做的那样。

谜题 27：变幻莫测的 i 值

与谜题 26 中的程序一样，下面的程序也包含了一个记录在终止前有多少次迭代的循环。与那个程序不同的是，这个程序使用的是左移操作符（<<）。你的任务照旧是要指出这个程序将打印什么。当你阅读这个程序时，请记住 Java 使用的是基于 2 的补码的二进制算术运算，因此-1 在任何有符号的整数类型中（byte、short、int 或 long）的表示都是所有的位被置位：

```
public class Shifty {
    public static void main(String[] args) {
        int i = 0;
        while (-1 << i != 0)
            i++;
        System.out.println(i);
    }
}
```

常量-1 是所有 32 位都被置位的 int 数值（0xffffffff）。左移操作符将 0 移入到由移位所空出的右边的最低位，因此表达式（-1 << i）将 i 最右边的位设置为 0，并保持其余的 32 - i 位为 1。很明显，这个循环将完成 32 次迭代，因为 -1 << i 对任何小于 32 的 i 来说都不等于 0。你可能期望终止条件测试在 i 等于 32 时返回 false，从而使程序打印 32，但是它打印的并不是 32。实际上，它不会打印任何东西，而是进入了一个无限循环。

问题在于（-1 << 32）等于-1 而不是 0，因为移位操作符之使用其右操作数的低 5 位作为移位长度。或者是低 6 位，如果其左操作数是一个 long 类数值[JLS 15.19]。

这条规则作用于全部的三个移位操作符：<<、>>和>>>。移位长度总是介于 0 到 31 之间，如果左操作数是 long 类型的，则介于 0 到 63 之间。这个长度是对 32 取余的，如果左操作数是 long 类型的，则对 64 取余。如果试图对一个 int 数值移位 32 位，或者是对一个 long 数值移位 64 位，都只能返回这个数值自身的值。没有任何移位长度可以让一个 int 数值丢弃其所有的 32 位，或者是让一个 long 数值丢弃其所有的 64 位。

幸运的是，有一个非常容易的方式能够订正该问题。我们不是让-1 重复地移位不同的移位长度，而是将前一次移位操作的结果保存起来，并且让它在每一次迭代时都向左再移 1 位。下面这个版本的程序就可以打印出我们所期望的 32：

```
public class Shifty {
    public static void main(String[] args) {
        int distance = 0;
        for (int val = -1; val != 0; val <<= 1)
            distance++;
        System.out.println(distance);
    }
}
```

这个订正过的程序说明了一条普遍的原则：如果可能的话，移位长度应该是常量。如果移位长度紧盯着你不放，那么你让其值超过 31，或者如果左操作数是 long 类型的，让其值超过 63 的可能性就会大大降低。当然，你并不可能总是可以使用常量的移位长度。当你必须使用一个非常量的移位长度时，请确保你的程序可以应付这种容易产生问题的情况，或者压根就不会碰到这种情况。

前面提到的移位操作符的行为还有另外一个令人震惊的结果。很多程序员都希望具有负的移位长度的右移操作符可以起到左移操作符的作用，反之亦然。但是情况并非如此。右移操作符总是起到右移的作用，而左移操作符也总是起到左移的作用。负的移位长度通过只保留低 5 位而剔除其他位的方式被转换成了正的移位长度——如果左操作数是 long 类型的，则保留低 6 位。因此，如果要将一个 int 数值左移，其移位长度为-1，那么移位的效果是它被左移了 31 位。

总之，移位长度是对 32 取余的，或者如果左操作数是 long 类型的，则对 64 取余。因此，使用任何移位操作符和移位长度，都不可能将一个数值的所有位全部移走。同时，我们也不可能用右移操作符来执行左移操作，反之亦然。如果可能的话，请使用常量的移位长度，如果移位长度不能设为常量，那么就要千万当心。

语言设计者可能应该考虑将移位长度限制在从 0 到以位为单位的类型尺寸的范围内，并且修改移位长度为类型尺寸时的语义，让其返回 0。尽管这可以避免在本谜题中所展示的混乱情况，但是它可能会带来负面的执行结果，因为 Java 的移位操作符的语义正是许多处理器上的移位指令的语义。

谜题 28：循环者

下面的谜题以及随后的五个谜题对你来说是扭转了局面，它们不是向你展示某些代码，然后询问你这些代码将做些什么，它们要让你去写代码，但是数量会很少。这些谜题被称为“循环者（looper）”。你眼前会展示出一个循环，它看起来应该很快就终止的，而你的任务就是写一个变量声明，在将它作用于该循环之上时，使得该循环无限循环下去。例如，考虑下面的 for 循环：

```
for (int i = start; i <= start + 1; i++) {}
```

看起来它好像应该只迭代两次，但是通过利用在谜题 26 中所展示的溢出行为，可以使它无限循环下去。下面的的声明就采用了这项技巧：

```
int start = Integer.MAX_VALUE - 1;
```

现在该轮到你了。什么样的声明能够让下面的循环变成一个无限循环？

```
While (i == i + 1) {}
```

仔细查看这个 while 循环，它真的好像应该立即终止。一个数字永远不会等于它自己加 1，对吗？嗯，如果这个数字是无穷大的，又会怎样呢？Java 强制要求使用 IEEE 754 浮点数算术运算[IEEE 754]，它可以让你用一个 double 或 float 来表示无穷大。正如我们在学校里面学到的，无穷大加 1 还是无穷大。如果 i 在循环开始之前被初始化为无穷大，那么终止条件测试(i == i + 1)就会被计算为 true，从而使循环永远都不会终止。

你可以用任何被计算为无穷大的浮点算术表达式来初始化 i，例如：

```
double i = 1.0 / 0.0;
```

不过，你最好是能够利用标准类库为你提供的常量：

```
double i = Double.POSITIVE_INFINITY;
```

事实上，你不必将 i 初始化为无穷大以确保循环永远执行。任何足够大的浮点数都可以实现这一目的，例如：

```
double i = 1.0e40;
```

这样做之所以可以起作用，是因为一个浮点数值越大，它和其后继数值之间的间隔就越大。浮点数的这种分布是用固定数量的有效位来表示它们的必然结果。对一个足够大的浮点数加 1 不会改变它的值，因为 1 是不足以“填补它与其后继者之间的空隙”。

浮点数操作返回的是最接近其精确的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级数是 2^{25}，即 33,554,432；而对于 double 类型，最小级数是 2^{54}，大约是 1.8×10^{16}。

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）”的首字母缩写词。在 5.0 版中，引入了 Math.ulp 方法来计算 float 或 double 数值的 ulp。

总之，用一个 double 或一个 float 数值来表示无穷大是可以的。大多数人在第一次听到这句话时，多少都会有一点吃惊，可能是因为我们无法用任何整数类型来表示无穷大的原因。第二点，将一个很小的浮点数加到一个很大的浮点数上时，将不会改变大的浮点数的值。这过于违背直觉了，因为对实际的数字来说这是不成立的。我们应该记住二进制浮点算术只是对实际算术的一种近似。

谜题 29：循环者的新娘

请提供一个对 i 的声明，将下面的循环转变为一个无限循环：

```
while (i != i) {
}
```
这个循环可能比前一个还要使人感到困惑。不管在它前面作何种声明，它看起来确实应该立即终止。一个数字总是等于它自己，对吗？

对，但是 IEEE 754 浮点算术保留了一个特殊的值用来表示一个不是数字的数量[IEEE 754]。这个值就是 NaN（“不是一个数字（Not a Number）”的缩写），对于所有没有良好的数字定义的浮点计算，例如 0.0/0.0，其值都是它。规范中描述道，NaN 不等于任何浮点数值，包括它自身在内[JLS 15.21.1]。因此，如果 i 在循环开始之前被初始化为 NaN，那么终止条件测试(i != i)的计算结果就是 true，循环就永远不会终止。很奇怪但却是事实。

你可以用任何计算结果为 NaN 的浮点算术表达式来初始化 i，例如：

```
double i = 0.0 / 0.0;
```
同样，为了表达清晰，你可以使用标准类库提供的常量：
```
double i = Double.NaN;
```
NaN 还有其他的惊人之处。任何浮点操作，只要它的一个或多个操作数为 NaN，那么其结果为 NaN。这条规则是非常合理的，但是它却具有奇怪的结果。例如，下面的程序将打印 false：
```
class Test {
   public static void main(String[] args) {
       double i = 0.0 / 0.0;
       System.out.println(i - i == 0);
   }
}
```
这条计算 NaN 的规则所基于的原理是：一旦一个计算产生了 NaN，它就被损坏了，没有任何更进一步的计算可以修复这样的损坏。NaN 值意图使受损的计算继续执行下去，直到方便处理这种情况的地方为止。

总之，float 和 double 类型都有一个特殊的 NaN 值，用来表示不是数字的数量。对于涉及 NaN 值的计算，其规则很简单也很明智，但是这些规则的结果可能是违背直觉的。

谜题 30：循环者的爱子

请提供一个对 i 的声明，将下面的循环转变为一个无限循环：
```
while (i != i + 0) {
}
```
与前一个谜题不同，你必须在你的答案中不使用浮点数。换句话说，你不能把 i 声明为 double 或 float 类型的。

与前一个谜题一样，这个谜题初看起来是不可能实现的。毕竟，一个数字总是等于它自身加上 0，你被禁止使用浮点数，因此不能使用 NaN，而在整数类型中没有 NaN 的等价物。那么，你能给出什么呢？

我们必然可以得出这样的结论，即 i 的类型必须是非数值类型的，并且这其中存在着解谜方案。唯一的 + 操作符有定义的非数值类型就是 String。+ 操作符被重载了：对于 String 类型，它执行的不是加法而是字符串连接。如果在连接中的某个操作数具有非 String 的类型，那么这个操作书就会在连接之前转换成字符串[JLS 15.18.1]。

事实上，i 可以被初始化为任何值，只要它是 String 类型的即可，例如：

```
String i = "Buy seventeen copies of Effective Java";
```

int 类型的数值 0 被转换成 String 类型的数值”0”，并且被追加到了感叹号之后，所产生的字符串在用 equals 方法计算时就不等于最初的字符串了，这样它们在使用==操作符进行计算时，当然就不是相等的。因此，计算布尔表达式(i != i + 0)得到的值就是 true，循环也就永远不会被终止了。

总之，操作符重载是很容易令人误解的。在本谜题中的加号看起来是表示一个加法，但是通过为变量 i 选择合适的类型，即 String，我们让它执行了字符串连接操作。甚至是因为变量被命名为 i，都使得本谜题更加容易令人误解，因为 i 通常被当作整型变量名而被保留的。对于程序的可读性来说，好的变量名、方法名和类名至少与好的注释同等重要。

对语言设计者的教训与谜题 11 和 13 中的教训相同。操作符重载是很容易引起混乱的，也许 + 操作符就不应该被重载用来进行字符串连接操作。有充分的理由证明提供一个字符串连接操作符是多么必要，但是它不应该是 + 。

谜题 31：循环者的鬼魂

请提供一个对 i 的声明，将下面的循环转变为一个无限循环：

```
while (i != 0) {
    i >>>= 1;
}
```

回想一下，>>>=是对应于无符号右移操作符的赋值操作符。0 被从左移入到由移位操作而空出来的位上，即使被移位的负数也是如此。

这个循环比前面三个循环要稍微复杂一点，因为其循环体非空。在其循环题中，i 的值由它右移一位之后的值所替代。为了使移位合法，i 必须是一个整数类型（byte、char、short、int 或 long）。无符号右移操作符把 0 从左边移入，因此看起来这个循环执行迭代的次数与最大的整数类型所占据的位数相同，即 64 次。如果你在循环的前面放置如下的声明，那么这确实就是将要发生的事情：

```
long i = -1; // -1L has all 64 bits set
```

你怎样才能将它转变为一个无限循环呢？解决本谜题的关键在于>>>=是一个复合赋值操作符。（复合赋值操作符包括*=、/=、%=、+=、-=、<<=、>>=、>>>=、&=、^=和|=。）有关混合操作符的一个不幸的事实是，它们可能会自动地执行窄化原始类型转换[JLS 15.26.2]，这种转换把一种数字类型转换成了另一种更缺乏表示能力的类型。窄化原始类型转换可能会丢失级数的信息，或者是数值的精度[JLS 5.1.3]。

让我们更具体一些，假设你在循环的前面放置了下面的声明：

```
short i = -1;
```

因为 i 的初始值（(short)0xffff）是非 0 的，所以循环体会被执行。在执行移位操作时，第一步是将 i 提升为 int 类型。所有算数操作都会对 short、byte 和 char 类型的操作数执行这样的提升。这种提升是一个拓宽原始类型转换，因此没有任何信息会丢失。这种提升执行的是符号扩展，因此所产生的 int 数值是 0xffffffff。然后，这个数值右移 1 位，但不使用符号扩展，因此产生了 int 数值 0x7fffffff。最后，这个数值被存回到 i 中。为了将 int 数值存入 short 变量，Java 执行的是可怕的窄化原始类型转换，它直接将高 16 位截掉。这样就只剩下(short)oxffff 了，我们又回到了开始处。循环的第二次以及后续的迭代行为都是一样的，因此循环将永远不会终止。

如果你将 i 声明为一个 short 或 byte 变量，并且初始化为任何负数，那么这种行为也会发生。如果你声明 i 为一个 char，那么你将无法得到无限循环，因为 char 是无符号的，所以发生在移位之前的拓宽原始类型转换不会执行符号扩展。

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。因为这样的表达式执行的是混合类型算术运算，它容易造成混乱。更糟的是，它们执行将隐式地执行会丢失信息的窄化转型，其结果是灾难性的。

对语言设计者的教训是语言不应该自动地执行窄化转换。还有一点值得好好争论的是，Java 是否应该禁止在 short、byte 和 char 变量上使用复合赋值操作符。

谜题 32：循环者的诅咒

请提供一个对 i 的声明，将下面的循环转变为一个无限循环：

```
while (i <= j && j <= i && i != j) {
}
```

噢，不，不要再给我看起来不可能的循环了！如果 i <= j 并且 j <= i，i 不是肯定等于 j 吗？这一属性对实数肯定有效。事实上，它是如此地重要，以至于它有这样的定义：实数上的≤关系是反对称的。Java 的<=操作符在 5.0 版之前是反对称的，但是这从 5.0 版之后就不再是了。

直到 5.0 版之前，Java 的数字比较操作符（<、<=、>和>=）要求它们的两个操作数都是原始数字类型的（byte、char、short、int、long、float 和 double）[JLS 15.20.1]。但是在 5.0 版中，规范作出了修改，新规范描述道：每一个操

作数的类型必须可以转换成原始数字类型[JLS 15.20.1，5.1.8]。问题难就难在这里了。

在 5.0 版中，自动包装（autoboxing）和自动反包装（auto-unboxing）被添加到了 Java 语言中。如果你对它们并不了解，请查看：http://java.sun.com/j2se/5.0/docs/guide/language/autoboxing.html [Boxing]。<=操作符在原始数字类型集上仍然是反对称的，但是现在它还被应用到了被包装的数字类型上。（被包装的数字类型有：Byte、Character、Short、Integer、Long、Float 和 Double。）<=操作符在这些类型的操作数上不是反对称的，因为 Java 的判等操作符（==和!=）在作用于对象引用时，执行的是引用 ID 的比较，而不是值的比较。

让我们更具体一些，下面的声明赋予表达式(i <= j && j <= i && i != j)的值为 true，从而将这个循环变成了一个无限循环：

```
Integer i = new Integer(0);
Integer j = new Integer(0);
```

前两个子表达式（i <= j 和 j <= i）在 i 和 j 上执行解包转换[JLS 5.1.8]，并且在数字上比较所产生的 int 数值。i 和 j 都表示 0，所以这两个子表达式都被计算为 true。第三个子表达式（i != j）在对象引用 i 和 j 上执行标识比较，因为它们都初始化为一个新的 Integer 实例，因此，第三个子表达式同样也被计算为 true，循环也就永远地环绕下去了。

你可能会感到奇怪，为什么语言规范没有修改为：当判等操作符作用于被包装的数字类型时，它们执行的是值比较。答案很简单：兼容性。当一种语言被广泛使用之后，以违反现有规范的方式去改变现有程序的行为是让人无法接受的。下面的程序过去总是保证可以打印 false，因此它必须继续保持此特征：

```
public class ReferenceComparison {
    public static void main(String[] args) {
        System.out.println(
             new Integer(0) == new Integer(0));
    }
}
```

判等操作符在其两个操作数中只有一个是被包装的数字类型，而另一个是原始类型时，执行的确实是数值比较。因为这在 5.0 版之前是非法的，所有在这里没有任何兼容性的问题。让我们更具体一些，下面的程序在 1.4 版中是非法的，而在 5.0 版中将打印 true：

```
public class ValueComparison {
    public static void main(String[] args) {
        System.out.println(
             new Integer(0) == 0);
    }
}
```

总之，当两个操作数都是被包装的数字类型时，数值比较操作符和判等操作符的行为存在着根本的差异：数值比较操作符执行的是值比较，而判等操作符执行的是引用标识的比较。

对语言设计者来说，如果判等操作符一直执行的都是数值比较（谜题 13），那么生活可能就要简单得多、快乐得多。也许真正的教训应该是：语言设计者应该拥有高质量的水晶球，以预测语言的未来，并且做出相应的设计决策。严肃一点地讲，语言设计者应该考虑语言可能会如何演化，并且应该努力去最小化在演化之路上的各种制约影响。

谜题 33：循环者遇到了狼人

请提供一个对 i 的声明，将下面的循环转变为一个无限循环。这个循环不需要使用任何 5.0 版的特性：

```
while (i != 0 && i == -i) {
}
```

这仍然是一个循环。在布尔表达式(i != 0 && i == -i)中，一元减号操作符作用于 i，这意味着它的类型必须是数字型的：一元减号操作符作用于一个非数字型操作数是非法的。因此，我们要寻找一个非 0 的数字型数值，它等于它自己的负值。NaN 不能满足这个属性，因为它不等于任何数值，因此，i 必须表示一个实际的数字。肯定没有任何数字满足这样的属性吗？

嗯，没有任何实数具有这种属性，但是没有任何一种 Java 数值类型能够对实数进行完美建模。浮点数值是用一个符号位、一个被通俗地称为尾数（mantissa）的有效数字以及一个指数来表示的。除了 0 之外，没有任何浮点数等于其符号位反转之后的值，因此 i 的类型必然是整数型的。

有符号的整数类型使用的是 2 的补码算术运算：为了对一个数值取其负值，你要反转其每一位，然后加 1，从而得到结果[JLS 15.15.4]。2 的补码算术运算的一个很大的优势是，0 具有唯一的表示形式。如果你要对 int 数值 0 取负值，你将得到 0xffffffff+1，它仍然是 0。

但是，这也有一个相应的不利之处，总共存在偶数个 int 数值——准确地说有 2^{32} 个——其中一个用来表示 0，这样就剩些奇数个 int 数值来表示正整数和负整数，这意味着正的和负的 int 数值的数量必然不相等。这暗示着至少有一个 int 数值，其负值不能正确地表示成为一个 int 数值。

事实上，恰恰就有一个这样的 int 数值，它就是 Integer.MIN_VALUE，即-2^{31}。他的十六进制表示是 0x80000000。其符号位为 1，其余所有的位都是 0。如果我们对这个值取负值，那么我们将得到 0x7fffffff+1，也就是 0x80000000，即 Integer.MIN_VALUE！换句话说，Integer.MIN_VALUE 是它自己的负值，Long.MIN_VALUE 也是一样。对这两个值取负值将会产生溢出，但是 Java 在整数计算中忽略了溢出。其结果已经阐述清楚了，即使它们并不总是你所期望的。

下面的声明将使得布尔表达式(i != 0 && i == -i)的计算结果为 true，从而使循环无限环绕下去：

```
int i = Integer.MIN_VALUE;
```

下面这个也可以：

```
long i = Long.MIN_VALUE;
```

如果你对取模运算很熟悉，那么很有必要指出，这个谜题也可以用代数方法解决。Java 的 int 算术运算是实际的算术运算对 2^{32} 取模的运算，因此本谜题需要一个对这种线性全等的非 0 解决方案：

$$i \equiv -i \pmod{2^{32}}$$

将 i 加到恒等式的两边，我们可以得到：

$$2i \equiv 0 \pmod{32}$$

对这种全等的非 0 解决方案就是 $i = 2^{31}$。尽管这个值不能表示成为一个 int，但是它是和 -2^{31} 全等的，即与 Integer.MIN_VALUE 全等。

总之，Java 使用 2 的补码的算术运算，它是非对称的。对于每一种有符号的整数类型（int、long、byte 和 short），负的数值总是比正的数值多一个，这个多出来的值总是这种类型所能表示的最小数值。对 Integer.MIN_VALUE 取负值得到的还是它没有改变过的值，Long.MIN_VALUE 也是如此。对 Short.MIN_VALUE 取负值并将所产生的 int 数值转型回 short，返回的同样是最初的值（Short.MIN_VALUE）。对 Byte.MIN_VALUE 来说，也会产生相似的结果。更一般地讲，千万要当心溢出：就像狼人一样，它是个杀手。

对语言设计者的教训与谜题 26 中的教训一样。应该对某种溢出不会悄悄发生的整数算术运算形式提供语言级的支持。

谜题 34：被计数击倒了

与谜题 26 和 27 中的程序一样，下面的程序有一个单重的循环，它记录迭代的次数，并在循环终止时打印这个数。那么，这个程序会打印出什么呢？

```
public class Count {
    public static void main(String[] args) {
        final int START = 2000000000;
        int count = 0;
        for (float f = START; f < START + 50; f++)
            count++;
        System.out.println(count);
    }
}
```

表面的分析也许会认为这个程序将打印 50，毕竟，循环变量（f）被初始化为 2,000,000,000，而终止值比初始值大 50，并且这个循环具有传统的“半开”形式：它使用的是 < 操作符，这是的它包括初始值但是不包括终止值。

然而，这种分析遗漏了关键的一点：循环变量是 float 类型的，而非 int 类型的。回想一下谜题 28，很明显，增量操作（f++）不能正常工作。F 的初始值接近于 Integer.MAX_VALUE，因此它需要用 31 位来精确表示，而 float 类型只能提供 24 位的精度。对如此巨大的一个 float 数值进行增量操作将不会改变其值。因此，这个程序看起来应该无限地循环下去，因为 f 永远也不可能解决其终止值。但是，如果你运行该程序，就会发现它并没有无限循环下去，事实上，它立即就终止了，并打印出 0。怎么回事呢？

问题在于终止条件测试失败了，其方式与增量操作失败的方式非常相似。这个循环只有在循环索引 f 比(float)(START + 50)小的情况下才运行。在将一个 int 与一个 float 进行比较时，会自动执行从 int 到 float 的提升[JLS 15.20.1]。遗憾的是，这种提升是会导致精度丢失的三种拓宽原始类型转换的一种[JLS 5.1.2]。（另外两个是从 long 到 float 和从 long 到 double。）

f 的初始值太大了，以至于在对其加上 50，然后将结果转型为 float 时，所产生的数值等于直接将 f 转换成 float 的数值。换句话说，(float)2000000000 == 2000000050，因此表达式 f < START + 50 即使是在循环体第一次执行之前就是 false，所以，循环体也就永远的不到机会去运行。

订正这个程序非常简单，只需将循环变量的类型从 float 修改为 int 即可。这样就避免了所有与浮点数计算有关的不精确性：

```
for (int f = START; f < START + 50; f++)
        count++;
```

如果不使用计算机，你如何才能知道 2,000,000,050 与 2,000,000,000 有相同的 float 表示呢？关键是要观察到 2,000,000,000 有 10 个因子都是 2：它是一个 2 乘以 9 个 10，而每个 10 都是 5×2。这意味着 2,000,000,000 的二进制表示是以 10 个 0 结尾的。50 的二进制表示只需要 6 位，所以将 50 加到 2,000,000,000 上不会对右边 6 位之外的其他为产生影响。特别是，从右边数过来的第 7 位和第 8 位仍旧是 0。提升这个 31 位的 int 到具有 24 位精度的 float 会在第 7 位和第 8 位之间四舍五入，从而直接丢弃最右边的 7 位。而最右边的 6 位是 2,000,000,000 与 2,000,000,050 位以不同之处，因此它们的 float 表示是相同的。

这个谜题寓意很简单：不要使用浮点数作为循环索引，因为它会导致无法预测的行为。如果你在循环体内需要一个浮点数，那么请使用 int 或 long 循环索引，并将其转换为 float 或 double。在将一个 int 或 long 转换成一个 float 或 double 时，你可能会丢失精度，但是至少它不会影响到循环本身。当你使用浮点数时，要使用 double 而不是 float，除非你肯定 float 提供了足够的精度，并且存在强制性的性能需求迫使你使用 float。适合使用 float 而不是 double 的时刻是非常非常少的。

对语言设计者的教训，仍然是悄悄地丢失精度对程序员来说是非常令人迷惑的。请查看谜题 31 有关这一点的深入讨论。

谜题 35：一分钟又一分钟

下面的程序在模仿一个简单的时钟。它的循环变量表示一个毫秒计数器，其计数值从 0 开始直至一小时中包含的毫秒数。循环体以定期的时间间隔对一个分钟计数器执行增量操作。最后，该程序将打印分钟计数器。那么它会打印出什么呢？

```
public class Clock {
    public static void main(String[] args) {
        int minutes = 0;
        for (int ms = 0; ms < 60*60*1000; ms++)
            if (ms % 60*1000 == 0)
                minutes++;
        System.out.println(minutes);
    }
}
```

在这个程序中的循环是一个标准的惯用 for 循环。它步进毫秒计数器（ms），从 0 到一小时中的毫秒数，即 3,600,000，包括前者但是不包括后者。循环体看起来是在每当毫秒计数器的计数值是 60,000（一分钟内所包含毫秒数）的倍数时，对分钟计数器（minutes）执行增量操作。这在循环的生命周期内总共发生了 3,600,000/60,000 次，即 60 次，因此你可能期望程序打印出 60，毕竟，这就是一小时所包含的分钟数。但是，该程序的运行却会告诉你另外一番景象：它打印的是 60000。为什么它会如此频繁地对 minutes 执行了增量操作呢？

问题在于那个布尔表达式(ms % 60*1000 == 0)。你可能会认为这个表达式等价于(ms % 60000 == 0)，但是它们并不等价。取余和乘法操作符具有相同的优先级[JLS 15.17]，因此表达式 ms % 60*1000 等价于(ms % 60)*1000。如果(ms % 60)等于 0 的话，这个表达式就等于 0，因此循环每 60 次迭代就对 minutes 执行增量操作。这使得最终的结果相差 1000 倍。

订正该程序的最简单的方式就是在布尔表达式中插入一对括号，以强制规定计算的正确顺序：

```
if (ms % (60 * 1000) == 0)
    minutes++;
```

然而，有一个更好的方法可以订正该程序。用被恰当命名的常量来替代所有的魔幻数字：

```
public class Clock {
    private static final int MS_PER_HOUR = 60 * 60 * 1000;
    private static final int MS_PER_MINUTE = 60 * 1000;
    public static void main(String[] args) {
        int minutes = 0;
        for (int ms = 0; ms < MS_PER_HOUR; ms++)
            if (ms % MS_PER_MINUTE == 0)
                minutes++;
        System.out.println(minutes);
```

```
    }
}
```

之所以要在最初的程序中展现表达式 ms % 60*1000，是为了诱使你去认为乘法比取余有更高的优先级。然而，编译器是忽略空格的，所以千万不要使用空格来表示分组，要使用括号。空格是靠不住的，而括号是从来不说谎的。

Java 谜题 4——异常谜题

谜题 36：优柔寡断 | 谜题 37：极端不可思议 | 谜题 38：不受欢迎的宾客 | 谜题 39：您好，再见 | 谜题 40：不情愿的构造器

谜题 41：域和流 | 谜题 42：异常为循环而抛 | 谜题 43：异常地危险 | 谜题 44：切掉类 | 谜题 45：令人疲惫不堪的测验

谜题 36：优柔寡断

下面这个可怜的小程序并不能很好地做出其自己的决定。它的 decision 方法将返回 true，但是它还返回了 false。那么，它到底打印的是什么呢？甚至，它是合法的吗？

```
public class Indecisive {
    public static void main(String[] args) {
        System.out.println(decision());
    }
    static boolean decision() {
        try {
            return true;
        } finally {
            return false;
        }
    }
}
```

你可能会认为这个程序是不合法的。毕竟，decision 方法不能同时返回 true 和 false。如果你尝试一下，就会发现它编译时没有任何错误，并且它所打印的是 false。为什么呢？

原因就是在一个 try-finally 语句中，finally 语句块总是在控制权离开 try 语句块时执行的[JLS 14.20.2]。无论 try 语句块是正常结束的，还是意外结束的，情况都是如此。一条语句或一个语句块在它抛出了一个异常，或者对某个封闭型语句执行了一个 break 或 continue，或是象这个程序一样在方法中执行了一个 return 时，将发生意外结束。它们之所以被称为意外结束，是因为它们阻止程序去按顺序执行下面的语句。

当 try 语句块和 finally 语句块都意外结束时，在 try 语句块中引发意外结束的原因将被丢弃，而整个 try-finally 语句意外结束的原因将于 finally 语句块意外结束的原因相同。在这个程序中，在 try 语句块中的 return 语句所引发的意外结束将被丢弃，而 try-finally 语句意外结束是由 finally 语句块中的 return 造成的。简单地讲，程序尝试着（try）返回（return）true，但是它最终（finally）返回（return）的是 false。

丢弃意外结束的原因几乎永远都不是你想要的行为，因为意外结束的最初原因可能对程序的行为来说会显得更重要。对于那些在 try 语句块中执行 break、continue 或 return 语句，只是为了使其行为被 finally 语句块所否决掉的程序，要理解其行为是特别困难的。

总之，每一个 finally 语句块都应该正常结束，除非抛出的是不受检查的异常。千万不要用一个 return、break、continue 或 throw 来退出一个 finally 语句块，并且千万不要允许将一个受检查的异常传播到一个 finally 语句块之外去。

对于语言设计者，也许应该要求 finally 语句块在未出现不受检查的异常时必须正常结束。朝着这个目标，try-finally 结构将要求 finally 语句块可以正常结束[JLS 14.21]。return、break 或 continue 语句把控制权传递到 finally 语句块之外应该是被禁止的，任何可以引发将被检查异常传播到 finally 语句块之外的语句也同样应该是被禁止的。

谜题 37：极端不可思议

本谜题测试的是你对某些规则的掌握程度，这些规则用于声明从方法中抛出并被 catch 语句块所捕获的异常。下面的三个程序每一个都会打印些什么？不要假设它们都可以通过编译：

```
import java.io.IOException;
public class Arcane1 {
    public static void main(String[] args) {
        try {
            System.out.println("Hello world");
        } catch(IOException e) {
            System.out.println("I've never seen
               println fail!");
        }
    }
}

public class Arcane2 {
    public static void main(String[] args) {
        try {
            // If you have nothing nice to say, say nothing
        } catch(Exception e) {
```

```
                System.out.println("This can't
                    happen");
            }
        }
}

interface Type1 {
    void f() throws CloneNotSupportedException;
}

interface Type2 {
    void f() throws InterruptedException;
}

interface Type3 extends Type1, Type2 {
}

public class Arcane3 implements Type3 {
    public void f() {
        System.out.println("Hello world");
    }
    public static void main(String[] args) {
        Type3 t3 = new Arcane3();
        t3.f();
    }
}
```

第一个程序，Arcane1，展示了被检查异常的一个基本原则。它看起来应该是可以编译的：try 子句执行 I/O，并且 catch 子句捕获 IOException 异常。但是这个程序不能编译，因为 println 方法没有声明会抛出任何被检查异常，而 IOException 却正是一个被检查异常。语言规范中描述道：如果一个 catch 子句要捕获一个类型为 E 的被检查异常，而其相对应的 try 子句不能抛出 E 的某种子类型的异常，那么这就是一个编译期错误[JLS 11.2.3]。

基于同样的理由，第二个程序，Arcane2，看起来应该是不可以编译的，但是它却可以。它之所以可以编译，是因为它唯一的 catch 子句检查了 Exception。尽管 JLS 在这一点上十分含混不清，但是捕获 Exception 或 Throwble 的 catch 子句是合法的，不管与其相对应的 try 子句的内容为何。尽管 Arcane2 是一个合法的程序，但是 catch 子句的内容永远的不会被执行，这个程序什么都不会打印。

第三个程序，Arcane3，看起来它也不能编译。方法 f 在 Type1 接口中声明要抛出被检查异常 CloneNotSupportedException，并且在 Type2 接口中声明要抛出被检查异常 InterruptedException。Type3 接口继承了 Type1 和 Type2，因此，看起来在静态类型为 Type3 的对象上调用方法 f 时，有潜在可能会抛出这些异常。一个方法必须要么捕获其方法体可以抛出的所有被检查异常，要么声明它将抛出

这些异常。Arcane3 的 main 方法在静态类型为 Type3 的对象上调用了方法 f，但它对 CloneNotSupportedException 和 InterruptedExceptioin 并没有作这些处理。那么，为什么这个程序可以编译呢？

上述分析的缺陷在于对“Type3.f 可以抛出在 Type1.f 上声明的异常和在 Type2.f 上声明的异常”所做的假设。这并不正确，因为每一个接口都限制了方法 f 可以抛出的被检查异常集合。一个方法可以抛出的被检查异常集合是它所适用的所有类型声明要抛出的被检查异常集合的交集，而不是合集。因此，静态类型为 Type3 的对象上的 f 方法根本就不能抛出任何被检查异常。因此，Arcane3 可以毫无错误地通过编译，并且打印 Hello world。

总之，第一个程序说明了一项基本要求，即对于捕获被检查异常的 catch 子句，只有在相应的 try 子句可以抛出这些异常时才被允许。第二个程序说明了这项要求不会应用到的冷僻案例。第三个程序说明了多个继承而来的 throws 子句的交集，将减少而不是增加方法允许抛出的异常数量。本谜题所说明的行为一般不会引发难以捉摸的 bug，但是你第一次看到它们时，可能会有点吃惊。

谜题 38：不受欢迎的宾客

本谜题中的程序所建模的系统，将尝试着从其环境中读取一个用户 ID，如果这种尝试失败了，则缺省地认为它是一个来宾用户。该程序的作者将面对有一个静态域的初始化表达式可能会抛出异常的情况。因为 Java 不允许静态初始化操作抛出被检查异常，所以初始化必须包装在 try-finally 语句块中。那么，下面的程序会打印出什么呢？

```
public class UnwelcomeGuest {
    public static final long GUEST_USER_ID = -1;
    private static final long USER_ID;
    static {
        try {
            USER_ID = getUserIdFromEnvironment();
        } catch (IdUnavailableException e) {
            USER_ID = GUEST_USER_ID;
            System.out.println("Logging in as guest");
        }
    }

    private static long getUserIdFromEnvironment()
            throws IdUnavailableException {
        throw new IdUnavailableException();
    }

    public static void main(String[] args) {
        System.out.println("User ID: " + USER_ID);
    }
```

```
}

class IdUnavailableException extends Exception {
}
```

该程序看起来很直观。对 getUserIdFromEnvironment 的调用将抛出一个异常，从而使程序将 GUEST_USER_ID(-1L)赋值给 USER_ID，并打印 Loggin in as guest。然后 main 方法执行，使程序打印 User ID: -1。表象再次欺骗了我们，该程序并不能编译。如果你尝试着去编译它，你将看到和下面内容类似的一条错误信息：

```
UnwelcomeGuest.java:10:
variable USER_ID might already have been assigned
            USER_ID = GUEST_USER_ID;
            ^
```

问题出在哪里了？USER_ID 域是一个空 final（blank final），它是一个在声明中没有进行初始化操作的 final 域[JLS 4.12.4]。很明显，只有在对 USER_ID 赋值失败时，才会在 try 语句块中抛出异常，因此，在 catch 语句块中赋值是相当安全的。不管怎样执行静态初始化操作语句块，只会对 USER_ID 赋值一次，这正是空 final 所要求的。为什么编译器不知道这些呢？

要确定一个程序是否可以不止一次地对一个空 final 进行赋值是一个很困难的问题。事实上，这是不可能的。这等价于经典的停机问题，它通常被认为是不可能解决的[Turing 36]。为了能够编写出一个编译器，语言规范在这一点上采用了保守的方式。在程序中，一个空 final 域只有在它是明确未赋过值的地方才可以被赋值。规范长篇大论，对此术语提供了一个准确的但保守的定义[JLS 16]。因为它是保守的，所以编译器必须拒绝某些可以证明是安全的程序。这个谜题就展示了这样的一个程序。

幸运的是，你不必为了编写 Java 程序而去学习那些骇人的用于明确赋值的细节。通常明确赋值规则不会有任何妨碍。如果碰巧你编写了一个真的可能会对一个空 final 赋值超过一次的程序，编译器会帮你指出的。只有在极少的情况下，就像本谜题一样，你才会编写出一个安全的程序，但是它并不满足规范的形式化要求。编译器的抱怨就好像是你编写了一个不安全的程序一样，而且你必须修改你的程序以满足它。

解决这类问题的最好方式就是将这个烦人的域从空 final 类型改变为普通的 final 类型，用一个静态域的初始化操作替换掉静态的初始化语句块。实现这一点的最佳方式是重构静态语句块中的代码为一个助手方法：

```
public class UnwelcomeGuest {
    public static final long GUEST_USER_ID = -1;
    private static final long USER_ID = getUserIdOrGuest;
    private static long getUserIdOrGuest {
        try {
            return getUserIdFromEnvironment();
```

```
        } catch (IdUnavailableException e) {
            System.out.println("Logging in as guest");
            return GUEST_USER_ID;
        }
    }
    ...// The rest of the program is unchanged
}
```

程序的这个版本很显然是正确的，而且比最初的版本根据可读性，因为它为了域值的计算而增加了一个描述性的名字，而最初的版本只有一个匿名的静态初始化操作语句块。将这样的修改作用于程序，它就可以如我们的期望来运行了。

总之，大多数程序员都不需要学习明确赋值规则的细节。该规则的作为通常都是正确的。如果你必须重构一个程序，以消除由明确赋值规则所引发的错误，那么你应该考虑添加一个新方法。这样做除了可以解决明确赋值问题，还可以使程序的可读性提高。

谜题 39：您好，再见！

下面的程序在寻常的 Hello world 程序中添加了一段不寻常的曲折操作。那么，它将会打印出什么呢？

```
public class HelloGoodbye {
    public static void main(String[] args) {
        try {
            System.out.println("Hello world");
            System.exit(0);
        } finally {
            System.out.println("Goodbye world");
        }
    }
}
```

这个程序包含两个 println 语句：一个在 try 语句块中，另一个在相应的 finally 语句块中。try 语句块执行它的 println 语句，并且通过调用 System.exit 来提前结束执行。在此时，你可能希望控制权会转交给 finally 语句块。然而，如果你运行该程序，就会发现它永远不会说再见：它只打印了 Hello world。这是否违背了谜题 36 中所解释的原则呢？

不论 try 语句块的执行是正常地还是意外地结束，finally 语句块确实都会执行。然而在这个程序中，try 语句块根本就没有结束其执行过程。System.exit 方法将停止当前线程和所有其他当场死亡的线程。finally 子句的出现并不能给予线程继续去执行的特殊权限。

当 System.exit 被调用时，虚拟机在关闭前要执行两项清理工作。首先，它执行所有的关闭挂钩操作，这些挂钩已经注册到了 Runtime.addShutdownHook 上。这对于释放 VM 之外的资源将很有帮助。务必要为那些必须在 VM 退出之前发生的行为关闭挂钩。下面的程序版本示范了这种技术，它可以如我们所期望地打印出 Hello world 和 Goodbye world：

```
public class HelloGoodbye1 {
    public static void main(String[] args) {
       System.out.println("Hello world");
       Runtime.getRuntime().addShutdownHook(
                    new Thread() {
                    public void run() {
                       System.out.println("Goodbye world");
                }
            });
       System.exit(0);
    }
}
```

VM 执行在 System.exit 被调用时执行的第二个清理任务与终结器有关。如果 System.runFinalizerOnExit 或它的魔鬼双胞胎 Runtime.runFinalizersOnExit 被调用了，那么 VM 将在所有还未终结的对象上面调用终结器。这些方法很久以前就已经过时了，而且其原因也很合理。无论什么原因，永远不要调用 System.runFinalizersOnExit 和 Runtime.runFinalizersOnExit:它们属于 Java 类库中最危险的方法之一[ThreadStop]。调用这些方法导致的结果是，终结器会在那些其他线程正在并发操作的对象上面运行，从而导致不确定的行为或导致死锁。

总之，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。当 VM 被关闭时，请使用关闭挂钩来终止外部资源。通过调用 System.halt 可以在不执行关闭挂钩的情况下停止 VM，但是这个方法很少使用。

谜题 40：不情愿的构造器

尽管在一个方法声明中看到一个 throws 子句是很常见的，但是在构造器的声明中看到一个 throws 子句就很少见了。下面的程序就有这样的一个声明。那么，它将打印出什么呢？

```
public class Reluctant {
    private Reluctant internalInstance = new Reluctant();
    public Reluctant() throws Exception {
        throw new Exception("I'm not coming out");
    }
    public static void main(String[] args) {
        try {
```

```
            Reluctant b = new Reluctant();
            System.out.println("Surprise!");
        } catch (Exception ex) {
            System.out.println("I told you so");
        }
    }
}
```

main 方法调用了 Reluctant 构造器，它将抛出一个异常。你可能期望 catch 子句能够捕获这个异常，并且打印 I told you so。凑近仔细看看这个程序就会发现，Reluctant 实例还包含第二个内部实例，它的构造器也会抛出一个异常。无论抛出哪一个异常，看起来 main 中的 catch 子句都应该捕获它，因此预测该程序将打印 I told you 应该是一个安全的赌注。但是当你尝试着去运行它时，就会发现它压根没有去做这类的事情：它抛出了 StackOverflowError 异常，为什么呢？

与大多数抛出 StackOverflowError 异常的程序一样，本程序也包含了一个无限递归。当你调用一个构造器时，实例变量的初始化操作将先于构造器的程序体而运行[JLS 12.5]。在本谜题中， internalInstance 变量的初始化操作递归调用了构造器，而该构造器通过再次调用 Reluctant 构造器而初始化该变量自己的 internalInstance 域，如此无限递归下去。这些递归调用在构造器程序体获得执行机会之前就会抛出 StackOverflowError 异常，因为 StackOverflowError 是 Error 的子类型而不是 Exception 的子类型，所以 catch 子句无法捕获它。

对于一个对象包含与它自己类型相同的实例的情况，并不少见。例如，链接列表节点、树节点和图节点都属于这种情况。你必须非常小心地初始化这样的包含实例，以避免 StackOverflowError 异常。

至于本谜题名义上的题目：声明将抛出异常的构造器，你需要注意，构造器必须声明其实例初始化操作会抛出的所有被检查异常。下面这个展示了常见的“服务提供商”模式的程序，将不能编译，因为它违反了这条规则：

```
public class Car {
    private static Class engineClass = ...;
    private Engine engine =
            (Engine)enginClass.newInstance();
    public Car(){ }
}
```

尽管其构造器没有任何程序体，但是它将抛出两个被检查异常，InstantiationException 和 IllegalAccessException。它们是 Class.Instance 抛出的，该方法是在初始化 engine 域的时候被调用的。订正该程序的最好方式是创建一个私有的、静态的助手方法，它负责计算域的初始值，并恰当地处理异常。在本案中，我们假设选择 engineClass 所引用的 Class 对象，保证它是可访问的并且是可实例化的。

下面的 Car 版本将可以毫无错误地通过编译：

```
//Fixed - instance initializers don't throw checked exceptions
public class Car {
     private static Class engineClass = ...;
     private Engine engine = newEngine;
     private static Engine newEngine() {
                try {
                        return (Engine)engineClass.newInstance();
          } catch (IllegalAccessException e) {
                throw new AssertionError(e);
          } catch (InstantiationException e) {
                throw new AssertionError(e);
          }
     }
     public Car(){ }
}
```

总之，实例初始化操作是先于构造器的程序体而运行的。实例初始化操作抛出的任何异常都会传播给构造器。如果初始化操作抛出的是被检查异常，那么构造器必须声明也会抛出这些异常，但是应该避免这样做，因为它会造成混乱。最后，对于我们所设计的类，如果其实例包含同样属于这个类的其他实例，那么对这种无限递归要格外当心。

谜题 41：域和流

下面的方法将一个文件拷贝到另一个文件，并且被设计为要关闭它所创建的每一个流，即使它碰到 I/O 错误也要如此。遗憾的是，它并非总是能够做到这一点。为什么不能呢，你如何才能订正它呢？

```
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 {
            if (in != null) in.close();
            if (out != null) out.close();
        }
}
```

这个程序看起来已经面面俱到了。其流域（in 和 out）被初始化为 null，并且新的流一旦被创建，它们马上就被设置为这些流域的新值。对于这些域所引用的

流，如果不为空，则 finally 语句块会将其关闭。即便在拷贝操作引发了一个 IOException 的情况下，finally 语句块也会在方法返回之前执行。出什么错了呢？

问题在 finally 语句块自身中。close 方法也可能会抛出 IOException 异常。如果这正好发生在 in.close 被调用之时，那么这个异常就会阻止 out.close 被调用，从而使输出流仍保持在开放状态。

请注意，该程序违反了谜题 36 的建议：对 close 的调用可能会导致 finally 语句块意外结束。遗憾的是，编译器并不能帮助你发现此问题，因为 close 方法抛出的异常与 read 和 write 抛出的异常类型相同，而其外围方法（copy）声明将传播该异常。

解决方式是将每一个 close 都包装在一个嵌套的 try 语句块中。下面的 finally 语句块的版本可以保证在两个流上都会调用 close：

```
} finally {
     if (in != null) {
          try {
              in.close();
          } catch (IOException ex) {
              // There is nothing we can do if close fails
          }
     if (out != null)
          try {
              out.close();
          } catch (IOException ex) {
              // There is nothing we can do if close fails
          }
    }
}
```

从 5.0 版本开始，你可以对代码进行重构，以利用 Closeable 接口：

```
} finally {
     closeIgnoringException(in);
     closeIgnoringEcception(out);
}
private static void closeIgnoringException(Closeable c) {
     if (c != null) {
           try {
             c.close();
           } catch (IOException ex) {
             // There is nothing we can do if close  fails
           }
     }
}
```

总之，当你在 finally 语句块中调用 close 方法时，要用一个嵌套的 try-catch 语句来保护它，以防止 IOException 的传播。更一般地讲，对于任何在 finally 语句块中可能会抛出的被检查异常都要进行处理，而不是任其传播。这是谜题 36 中的教训的一种特例，而对语言设计着的教训情况也相同。

谜题 42：异常为循环而抛

下面的程序循环遍历了一个 int 类型的数组序列，并且记录了满足某个特定属性的数组个数。那么，该程序会打印出什么呢？

```
public class Loop {
    public static void main(String[] args) {
        int[][] tests = { { 6, 5, 4, 3, 2, 1 }, { 1, 2 },
                          { 1, 2, 3 }, { 1, 2, 3, 4 }, { 1 } };
        int successCount = 0;
        try {
            int i = 0;
            while (true) {
                if (thirdElementIsThree(tests[i++]))
                    successCount ++;
            }
        } catch(ArrayIndexOutOfBoundsException e) {
            // No more tests to process
        }
        System.out.println(successCount);
    }

    private static boolean thirdElementIsThree(int[] a) {
        return a.length >= 3 & a[2] == 3;
    }
}
```

该程序用 thirdElementIsThree 方法测试了 tests 数组中的每一个元素。遍历这个数组的循环显然是非传统的循环：它不是在循环变量等于数组长度的时候终止，而是在它试图访问一个并不在数组中的元素时终止。尽管它是非传统的，但是这个循环应该可以工作。如果传递给 thirdElementIsThree 的参数具有 3 个或更多的元素，并且其第三个元素等于 3，那么该方法将返回 true。对于 tests 中的 5 个元素来说，有 2 个将返回 true，因此看起来该程序应该打印 2。如果你运行它，就会发现它打印的时 0。肯定是哪里出了问题，你能确定吗？

事实上，这个程序犯了两个错误。第一个错误是该程序使用了一种可怕的循环惯用法，该惯用法依赖的是对数组的访问会抛出异常。这种惯用法不仅难以阅读，而且运行速度还非常地慢。不要使用异常来进行循环控制；应该只为异常条件而

使用异常[EJ Item 39]。为了纠正这个错误，可以将整个 try-finally 语句块替换为循环遍历数组的标准惯用法：

```
for (int i = 0; i < test.length; i++)
        if (thirdElementIsThree(tests[i]))
                successCount++;
```

如果你使用的是 5.0 或者是更新的版本，那么你可以用 for 循环结构来代替：

```
for (int[] test : tests)
        if(thirdElementIsThree(test))
                successCount++;
```

就第一个错误的糟糕情况来说，只有它自己还不足以产生我们所观察到的行为。然而，订正该错误可以帮助我们找到真正的 bug，它更加深奥：

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 2
        at Loop1.thirdElementIsThree(Loop1.java:19)
        at Loop1.main(Loop1.java:13)
```

很明显，在 thirdElementIsThree 方法中有一个 bug：它抛出了一个 ArrayIndexOutOfBoundsException 异常。这个异常先前伪装成了那个可怕的基于异常的循环的终止条件。

如果传递给 thirdElementIsThree 的参数具有 3 个或更多的元素，并且其第三个元素等于 3，那么该方法将返回 true。问题是在这些条件不满足时它会做些什么呢。如果你仔细观察其值将会被返回的那个布尔表达式，你就会发现它与大多数布尔 AND 操作有一点不一样。这个表达式是 a.length >= 3 & a[2] == 3。通常，你在这种情况下看到的是 && 操作符，而这个表达式使用的是 & 操作符。那是一个位 AND 操作符吗？

事实证明 & 操作符有其他的含义。除了常见的被当作整型操作数的位 AND 操作符之外，当被用于布尔操作数时，它的功能被重载为逻辑 AND 操作符[JLS 15.22.2]。这个操作符与更经常被使用的条件 AND 操作符有所不同，& 操作符总是要计算它的两个操作数，而 && 操作符在其左边的操作数被计算为 false 时，就不再计算右边的操作数了[JLS 15.23]。因此，thirdElementIsThree 方法总是要试图访问其数组参数的第三个元素，即使该数组参数的元素不足 3 个也是如此。订正这个方法只需将 & 操作符替换为 && 操作符即可。通过这样的修改，这个程序就可以打印出我们所期望的 2 了：

```
private static boolean thirdElementIsThree(int[] a) {
        return a.length >= 3 && a[2] == 3;
    }
```

正像有一个逻辑 AND 操作符伴随着更经常被使用的条件 AND 操作符一样，还有一个逻辑 OR 操作符(|)也伴随着条件 OR 操作符(||)[JLS 15.22.2，15.24]。| 操作符总是要计算它的两个操作数，而 || 操作符在其左边的操作数被计算为 true 时，就不再计算右边的操作数了。我们一不注意，就很容易使用了逻辑操作符而不是条件操作符。遗憾的是，编译器并不能帮助你发现这种错误。有意识地使用逻辑操作符的情形非常少见，少到了我们对所有这样使用的程序都应该持

怀疑态度的地步。如果你真的想使用这样的操作符，为了是你的意图清楚起见，请加上注释。

总之，不要去用那些可怕的使用异常而不是使用显式的终止测试的循环惯用法，因为这种惯用法非常不清晰，而且会掩盖 bug。要意识到逻辑 AND 和 OR 操作符的存在，并且不要因无意识的误用而受害。对语言设计者来说，这又是一个操作符重载会导致混乱的明证。对于在条件 AND 和 OR 操作符之外还要提供逻辑 AND 和 OR 操作符这一点，并没有很明显的理由。如果这些操作符确实要得到支持的话，它们应该与其相对应的条件操作符存在着视觉上的明显差异。

谜题 43：异常地危险

在 JDK1.2 中，Thread.stop、Thread.suspend 以及其他许多线程相关的方法都因为它们不安全而不推荐使用了[ThreadStop]。下面的方法展示了你用 Thread.stop 可以实现的可怕事情之一。

```
// Don’t do this - circumvents exception checking!
public static void sneakyThrow(Throwable t) {
        Thread.currentThread().stop(t); // Deprecated!!
}
```

这个讨厌的小方法所做的事情正是 throw 语句要做的事情，但是它绕过了编译器的所有异常检查操作。你可以（卑鄙地）在你的代码的任意一点上抛出任何受检查的或不受检查的异常，而编译器对此连眉头都不会皱一下。

不使用任何不推荐的方法，你也可以编写出在功能上等价于 sneakyThrow 的方法。事实上，至少有两种方式可以这么实现这一点，其中一种只能在 5.0 或更新的版本中运行。你能够编写出这样的方法吗？它必须是用 Java 而不是用 JVM 字节码编写的，你不能在其客户对它编译完之后再去修改它。你的方法不必是完美无瑕的：如果它不能抛出一两个 Exception 的子类，也是可以接受的。

本谜题的一种解决之道是利用 Class.newInstance 方法中的设计缺陷，该方法通过反射来对一个类进行实例化。引用有关该方法的文档中的话[Java-API]：“请注意，该方法将传播从空的[换句话说，就是无参数的]构造器所抛出的任何异常，包括受检查的异常。使用这个方法可以有效地绕开在其他情况下都会执行的编译期异常检查。”一旦你了解了这一点，编写一个 sneakyThrow 的等价方法就不是太难了。

```
public class Thrower {
    private static Throwable t;
    private Thrower() throws Throwable {
        throw t;
    }

public static synchronized void sneakyThrow(Throwable t) {
        Thrower.t = t;
```

```
            try {
                Thrower.class.newInstance();
            } catch (InstantiationException e) {
                throw new IllegalArgumentException();
            } catch (IllegalAccessException e) {
                throw new IllegalArgumentException();
            } finally {
                Thrower.t = null; // Avoid memory leak
            }
        }
}
```

在这个解决方案中将会发生许多微妙的事情。我们想要在构造器执行期间所抛出的异常不能作为一个参数传递给该构造器，因为 Class.newInstance 调用的是一个类的无参数构造器。因此，sneakyThrow 方法将这个异常藏匿于一个静态变量中。为了使该方法是线程安全的，它必须被同步，这使得对其的并发调用将顺序地使用静态域 t。

要注意的是，t 这个域在从 finally 语句块中出来时是被赋为空的：这只是因为该方法虽然是卑鄙的，但这并不意味着它还应该是内存泄漏的。如果这个域不是被赋为空出来的，那么它阻止该异常被垃圾回收。最后，请注意，如果你让该方法抛出一个 InstantiationException 或是一个 IllegalAccessException 异常，它将以抛出一个 IllegalArgumentException 而失败。这是这项技术的一个内在限制。

Class.newInstance 的文档继续描述道“Constructor.newInstance 方法通过将构造器抛出的任何异常都包装在一个（受检查的）InvocationTargetException 异常中而避免了这个问题。”很明显，Class.newInstance 应该是做了相同的处理，但是纠正这个缺陷已经为时过晚，因为这么做将引入源代码级别的不兼容性，这将使许多依赖于 Class.newInstance 的程序崩溃。而弃用这个方法也不切实际，因为它太常用了。当你在使用它时，一定要意识到 Class.newInstance 可以抛出它没有声明过的受检查异常。

被添加到 5.0 版本中的“通用类型（generics）”可以为本谜题提供一个完全不同的解决方案。为了实现最大的兼容性，通用类型是通过类型擦除（type erasure）来实现的：通用类型信息是在编译期而非运行期检查的[JLS 4.7]。

下面的解决方案就利用了这项技术：

```
// Don't do this either - circumvents exception checking!
class TigerThrower<T extends Throwable> {
    public static void sneakyThrow(Throwable t) {
        new TigerThrower<Error>().sneakyThrow2(t);
    }
    private void sneakyThrow2(Throwable t) throws T {
        throw (T) t;
```

```
    }
}
```

这个程序在编译时将产生一条警告信息：

```
TigerThrower.java:7:warning: [unchecked] unchecked cast
found    :  java.lang.Throwable, required: T
          throw (T) t;
                    ^
```

警告信息是编译器所采用的一种手段，用来告诉你：你可能正在搬起石头砸自己的脚，而且事实也正是如此。“不受检查的转型”警告告诉你这个有问题的转型将不会在运行时刻受到检查。当你获得了一个不受检查的转型警告时，你应该修改你的程序以消除它，或者你可以确信这个转型不会失败。如果你不这么做，那么某个其他的转型可能会在未来不确定的某个时刻失败，而你也就很难跟踪此错误到其源头了。对于本谜题所示的情况，其情况更糟糕：在运行期抛出的异常可能与方法的签名不一致。sneakyThrow2 方法正是利用了这一点。

对平台设计者来说，有好几条教训。在设计诸如反射类库之类在语言之外实现的类库时， 要保留语言所作的所有承诺。当从头设计一个支持通用类型的平台时，要考虑强制要求其在运行期的正确性。Java 通用类型工具的设计者可没有这么做，因为他们受制于通用类库必须能够与现有客户进行互操作的要求。对于违反方法签名的异常，为了消除其产生的可能性，应该考虑强制在运行期进行异常检查。

总之，Java 的异常检查机制并不是虚拟机强制执行的。它只是一个编译期工具，被设计用来帮助我们更加容易地编写正确的程序，但是在运行期可以绕过它。要想减少你因为这类问题而被曝光的次数，就不要忽视编译器给出的警告信息。

谜题 44：切掉类

请考虑下面的两个类：

```
public class Strange1 {
    public static void main(String[] args) {
        try {
            Missing m = new Missing();
        } catch (java.lang.NoClassDefFoundError ex) {
            System.out.println("Got it!");
        }
    }
}

public class Strange2 {
    public static void main(String[] args) {
        Missing m;
```

```
        try {
            m = new Missing();
        } catch (java.lang.NoClassDefFoundError ex) {
            System.out.println("Got it!");
        }
    }
}
```

Strange1 和 Strange2 都用到了下面这个类：

```
class Missing {
    Missing() { }
}
```

如果你编译所有这三个类，然后在运行 Strange1 和 Strange2 之前删除 Missing.class 文件，你就会发现这两个程序的行为有所不同。其中一个抛出了一个未被捕获的 NoClassDefFoundError 异常，而另一个却打印出了 Got it! 到底哪一个程序具有哪一种行为，你又如何去解释这种行为上的差异呢？

程序 Strange1 只在其 try 语句块中提及 Missing 类型，因此你可能会认为它捕获 NoClassDefFoundError 异常，并打印 Got it! 另一方面，程序 Strange2 在 try 语句块之外声明了一个 Missing 类型的变量，因此你可能会认为所产生的 NoClassDefFoundError 异常不会被捕获。如果你试着运行这些程序，就会看到它们的行为正好相反：Strange1 抛出了未被捕获的 NoClassDefFoundError 异常，而 Strange2 却打印出了 Got it! 怎样才能解释这些奇怪的行为呢？

如果你去查看 Java 规范以找出应该抛出 NoClassDefFoundError 异常的地方，那么你不会得到很多的指导信息。该规范描述道，这个错误可以“在（直接或间接）使用某个类的程序中的任何地方”抛出[JLS 12.2.1]。当 VM 调用 Strange1 和 Strange2 的 main 方法时，这些程序都间接使用了 Missing 类，因此，它们都在其权利范围内于这一点上抛出了该错误。

于是，本谜题的答案就是这两个程序可以依据其实现而展示出各自不同的行为。但是这并不能解释为什么这些程序在所有我们所知的 Java 实现上的实际行为，与你所认为的必然行为都正好相反。要查明为什么会是这样，我们需要研究一下由编译器生成的这些程序的字节码。

如果你去比较 Strange1 和 Strange2 的字节码，就会发现几乎是一样的。除了类名之外，唯一的差异就是 catch 语句块所捕获的参数 ex 与 VM 本地变量之间的映射关系不同。尽管哪一个程序变量被指派给了哪一个 VM 变量的具体细节会因编译器的不同而有所差异，但是对于和上述程序一样简单的程序来说，这些细节不太可能会差异很大。下面是通过执行 javap -c Strange1 命令而显示的 Strange1.main 的字节码：

```
0: new
3: dup
4: invokespecial    #3; //Method Missing."<init>":()V
7: astore_1
```

```
8: goto 20
11: astore_1
12: getstatic       #5; // Field System.out:Ljava/io/PrintStream;
15: ldc             #6; // String "Got it!"
17: invokevirtual   #7;//Method PrintStream.println: (String); V
20: return
Exception table:
from to target type
  0   8    11    Class java/lang/NoClassDefFoundError
```

Strange2.main 相对应的字节码与其只有一条指令不同：

```
11: astore_2
```

这是一条将 catch 语句块中的捕获异常存储到捕获参数 ex 中的指令。在 Strange1 中，这个参数是存储在 VM 变量 1 中的，而在 Strange2 中，它是存储在 VM 变量 2 中的。这就是两个类之间唯一的差异，但是它所造成的程序行为上的差异是多么地大呀！

为了运行一个程序，VM 要加载和初始化包含 main 方法的类。在加载和初始化之间，VM 必须链接（link）类[JLS 12.3]。链接的第一阶段是校验，校验要确保一个类是良构的，并且遵循语言的语法要求。校验非常关键，它维护着可以将像 Java 这样的安全语言与像 C 或 C++这样的不安全语言区分开的各种承诺。

在 Strange1 和 Strange2 这两个类中，本地变量 m 碰巧都被存储在 VM 变量 1 中。两个版本的 main 都有一个连接点，从两个不同位置而来的控制流汇聚于此。该连接点就是指令 20，即从 main 返回的指令。在正常结束 try 语句块的情况下，我们执行到指令 8，即 goto 20，从而可以到达指令 20；而对于在 catch 语句块中结束的情况，我们将执行指令 17，并按顺序执行下去，到达指令 20。

连接点的存在使得在校验 Strange1 类时产生异常，而在校验 Strange2 类时并不会产生异常。当校验去执行对 Strange1.main 的流分析（flow analysis）[JLS 12.3.1]时，由于指令 20 可以通过两条不同的路径到达，因此校验器必须合并在变量 1 中的类型。两种类型是通过计算它们的首个公共超类（first common superclass）[JVMS 4.9.2]而合并的。两个类的首个公共超类是它们所共有的最详细而精确的超类。

在 Strange1.main 方法中，当从指令 8 到达指令 20 时，VM 变量 1 的状态包含了一个 Missing 类的实例。当从指令 17 到达时，它包含了一个 NoClassDefFoundError 类的实例。为了计算首个公共超类，校验器必须加载 Missing 类以确定其超类。因为 Missing.class 文件已经被删除了，所以校验器不能加载它，因而抛出了一个 NoClassDefFoundError 异常。请注意，这个异常是在校验期间、在类被初始化之前，并且在 main 方法开始执行之前很早就抛出的。这就解释了为什么没有打印出任何关于这个未被捕获异常的跟踪栈信息。

要想编写一个能够探测出某个类是否丢失的程序，请使用反射来引用类而不要使用通常的语言结构[EJ Item35]。

下面展示了用这种技巧重写的程序：

```
public class Strange {
    public static void main(String[] args) throws
    Exception{
        try {
            Object m = Class.forName("Missing").
                       newInstance();
        } catch (ClassNotFoundException ex) {
            System.err.println("Got it!");
        }
    }
}
```

总之，不要对捕获 NoClassDefFoundError 形成依赖。语言规范非常仔细地描述了类初始化是在何时发生的[JLS 12.4.1]，但是类被加载的时机却显得更加不可预测。更一般地讲，捕获 Error 及其子类型几乎是完全不恰当的。这些异常是为那些不能被恢复的错误而保留的。

谜题 45：令人疲惫不堪的测验

本谜题将测试你对递归的了解程度。下面的程序将做些什么呢？

```
public class Workout {
    public static void main(String[] args) {
        workHard();
        System.out.println("It's nap time.");
    }
    private static void workHard() {
        try {
            workHard();
        } finally {
            workHard();
        }
    }
}
```

要不是有 try-finally 语句，该程序的行为将非常明显：workHard 方法递归地调用它自身，直到程序抛出 StackOverflowError，在此刻它以这个未捕获的异常而终止。但是，try-finally 语句把事情搞得复杂了。当它试图抛出 StackOverflowError 时，程序将会在 finally 语句块的 workHard 方法中终止，这样，它就递归调用了自己。这看起来确实就像是一个无限循环的秘方，但是这个程序真的会无限循环下去吗？如果你运行它，它似乎确实是这么做的，但是要想确认的唯一方式就是分析它的行为。

Java 虚拟机对栈的深度限制到了某个预设的水平。当超过这个水平时，VM 就抛出 StackOverflowError。为了让我们能够更方便地考虑程序的行为，我们假设栈的深度为 3，这比它实际的深度要小得多。现在让我们来跟踪其执行过程。

main 方法调用 workHard，而它又从其 try 语句块中递归地调用了自己，然后它再一次从其 try 语句块中调用了自己。在此时，栈的深度是 3。当 workHard 方法试图从其 try 语句块中再次调用自己时，该调用立即就会以 StackOverflowError 而失败。这个错误是在最内部的 finally 语句块中被捕获的，在此处栈的深度已经达到了 3。在那里，workHard 方法试图递归地调用它自己，但是该调用却以 StackOverflowError 而失败。这个错误将在上一级的 finally 语句块中被捕获，在此处站的深度是 2。该 finally 中的调用将与相对应的 try 语句块具有相同的行为：最终都会产生一个 StackOverflowError。这似乎形成了一种模式，而事实也确实如此。

WorkOut 的运行过程如左面的图所示。在这张图中，对 workHard 的调用用箭头表示，workHard 的执行用圆圈表示。所有的调用除了一个之外，都是递归的。会立即产生 StackOverflowError 异常的调用用由灰色圆圈前导的箭头表示，try 语句块中的调用用向左边的向下箭头表示，finally 语句块中的调用用向右边的向下箭头表示。箭头上的数字描述了调用的顺序。

这张图展示了一个深度为 0 的调用（即 main 中的调用），两个深度为 1 的调用，四个深度为 2 的调用，和八个深度为 3 的调用，总共是 15 个调用。那八个深度为 3 的调用每一个都会立即产生 StackOverflowError。至少在把栈的深度限制为 3 的 VM 上，该程序不会是一个无限循环：它在 15 个调用和 8 个异常之后就会终止。但是对于真实的 VM 又会怎样呢？它仍然不会是一个无限循环。其调用图与前面的图相似，只不过要大得多得多而已。

那么，究竟大到什么程度呢？有一个快速的试验表明许多 VM 都将栈的深度限制为 1024，因此，调用的数量就是 1+2+4+8…+21,024=21,025-1，而抛出的异常的数量是 21,024。假设我们的机器可以在每秒钟内执行 1010 个调用，并产生 1010 个异常，按照当前的标准，这个假设的数量已经相当高了。在这样的假设条件下，程序将在大约 1.7×10291 年后终止。为了让你对这个时间有直观的概念，我告诉你，我们的太阳的生命周期大约是 1010 年，所以我们可以很确定，我们中没有任何人能够看到这个程序终止的时刻。尽管它不是一个无限循环，但是它也就算是一个无限循环吧。

从技术角度讲，调用图是一棵完全二叉树，它的深度就是 VM 的栈深度的上限。WorkOut 程序的执行过程等于是在先序遍历这棵树。在先序遍历中，程序先访问一个节点，然后递归地访问它的左子树和右子树。对于树中的每一条边，都会产生一个调用，而对于树中的每一个节点，都会抛出一个异常。

本谜题没有很多关于教训方面的东西。它证明了指数算法对于除了最小输入之外的所有情况都是不可行的，它还表明了你甚至可以不费什么劲就可以编写出一个指数算法。

Java 谜题 5——类谜题

谜题 46：令人混淆的构造器案例

本谜题呈现给你了两个容易令人混淆的构造器。main 方法调用了一个构造器，但是它调用的到底是哪一个呢？该程序的输出取决于这个问题的答案。那么它到底会打印出什么呢？甚至它是否是合法的呢？

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

传递给构造器的参数是一个空的对象引用，因此，初看起来，该程序好像应该调用参数类型为 Object 的重载版本，并且将打印出 Object。另一方面，数组也是引用类型，因此 null 也可以应用于类型为 double[ ]的重载版本。你由此可能会得出结论：这个调用是模棱两可的，该程序应该不能编译。如果你试着去运行该程序，就会发现这些直观感觉都是不对的：该程序打印的是 double array。这种行为可能显得有悖常理，但是有一个很好的理由可以解释它。

Java 的重载解析过程是以两阶段运行的。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[JLS 15.12.2.5]。

在我们的程序中，两个构造器都是可获得并且可应用的。构造器Confusing(Object)可以接受任何传递给Confusing(double[ ])的参数，因此Confusing(Object)相对缺乏精确性。（每一个double数组都是一个Object，但是每一个Object并不一定是一个double数组。）因此，最精确的构造器就是Confusing(double[ ])，这也就解释了为什么程序会产生这样的输出。

如果你传递的是一个double[ ]类型的值，那么这种行为是有意义的；但是如果你传递的是null，这种行为就有违直觉了。理解本谜题的关键在于在测试哪一个方法或构造器最精确时，这些测试没有使用实际的参数：即出现在调用中的参数。这些参数只是被用来确定哪一个重载版本是可应用的。一旦编译器确定了哪些重载版本是可获得且可应用的，它就会选择最精确的一个重载版本，而此时使用的仅仅是形式参数：即出现在声明中的参数。

要想用一个null参数来调用Confusing(Object)构造器，你需要这样写代码：new Confusing((Object)null)。这可以确保只有Confusing(Object)是可应用的。更一般地讲，要想强制要求编译器选择一个精确的重载版本，需要将实际的参数转型为形式参数所声明的类型。

以这种方式来在多个重载版本中进行选择是相当令人不快的。在你的API中，应该确保不会让客户端走这种极端。理想状态下，你应该避免使用重载：为不同的方法取不同的名称。当然，有时候这无法实现，例如，构造器就没有名称，因而也就无法被赋予不同的名称。然而，你可以通过将构造器设置为私有的并提供公有的静态工厂，以此来缓解这个问题[EJ Item 1]。如果构造器有许多参数，你可以用Builder模式[Gamma95]来减少对重载版本的需求量。

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。如果做不到这一点，那么就请确保所有可应用的重载版本都具有相同的行为[EJ Item 26]。

总之，重载版本的解析可能会产生混淆。应该尽可能地避免重载，如果你必须进行重载，那么你必须遵守上述方针，以最小化这种混淆。如果一个设计糟糕的API强制你在不同的重载版本之间进行选择，那么请将实际的参数转型为你希望调用的重载版本的形式参数所具有的类型。

谜题47：啊呀！我的猫变成狗了

下面的程序使用了一个Counter类来跟踪每一种家庭宠物叫唤的次数。那么该程序会打印出什么呢？

```
class Counter {
    private static int count = 0;
    public static final synchronized void increment() {
        count++;
    }
    public static final synchronized int getCount() {
```

```
        return count;
    }
}

class Dog extends Counter {
    public Dog() { }
    public void woof() { increment(); }
}

class Cat extends Counter {
    public Cat() { }
    public void meow() { increment(); }
}

public class Ruckus {
    public static void main(String[] args) {
        Dog dogs[] = { new Dog(), new Dog() };
        for (int i = 0; i < dogs.length; i++)
            dogs[i].woof();
        Cat cats[] = { new Cat(), new Cat(), new Cat() };
        for (int i = 0; i < cats.length; i++)
            cats[i].meow();
        System.out.print(Dog.getCount() + " woofs and ");
        System.out.println(Cat.getCount() + " meows");
    }
}
```

我们听到两声狗叫和三声猫叫——肯定是好一阵喧闹——因此，程序应该打印 2 woofs and 3 meows，不是吗？不：它打印的是 5 woofs and 5 meows。所有这些多出来的吵闹声是从哪里来的？我们做些什么才能够阻止它？

该程序打印出的犬吠声和猫叫声的数量之和是 10，它是实际总数的两倍。问题在于 Dog 和 Cat 都从其共同的超类那里继承了 count 域，而 count 又是一个静态域。每一个静态域在声明它的类及其所有子类中共享一份单一的拷贝，因此 Dog 和 Cat 使用的是相同的 count 域。每一个对 woof 或 meow 的调用都在递增这个域，因此它被递增了 5 次。该程序分别通过调用 Dog.getCount 和 Cat.getCount 读取了这个域两次，在每一次读取时，都返回并打印了 5。

在设计一个类的时候，如果该类构建于另一个类的行为之上，那么你有两种选择：一种是继承，即一个类扩展另一个类；另一种是组合，即在一个类中包含另一个类的一个实例。选择的依据是，一个类的每一个实例都是另一个类的一个实例，还是都有另一个类的一个实例。在第一种情况应该使用继承，而第二种情况应该使用组合。当你拿不准时，优选组合而不是继承[EJ Item 14]。

一条狗或是一只猫都不是一种计数器，因此使用继承是错误的。Dog 和 Cat 不应该扩展 Counter，而是应该都包含一个计数器域。每一种宠物都需要有一个计数

器，但并非每一只宠物都需要有一个计数器，因此，这些计数器域应该是静态的。我们不必为 Counter 类而感到烦恼；一个 int 域就足够了。

下面是我们重新设计过的程序，它会打印出我们所期望的 2 woofs, 3 meows：

```
class Dog {
    private static int woofCounter;
    public Dog() { }
    public static int woofCount() { return woofCounter; };
    public void woof() { woofCounter++; }
}

class Cat {
    private static int meowCounter;
    public Cat() { }
    public static int meowCount() { return meowCounter; };
    public void meow() { meowCounter++; }
}
```

Ruckus 类除了两行语句之外没有其它的变化，这两行语句被修改为使用新的方法名来访问计数器：

```
System.out.print(Dog.woofCount() + " woofs and ");
System.out.println(Cat.meowCount() + " meows");
```

总之，静态域由声明它的类及其所有子类所共享。如果你需要让每一个子类都具有某个域的单独拷贝，那么你必须在每一个子类中声明一个单独的静态域。如果每一个实例都需要一个单独的拷贝，那么你可以在基类中声明一个非静态域。还有就是，要优选组合而不是继承，除非导出类真的需要被当作是某一种基类来看待。

谜题 48：我所得到的都是静态的

下面的程序对巴辛吉小鬣狗和其它狗之间的行为差异进行了建模。如果你不知道什么是巴辛吉小鬣狗，那么我告诉你，这是一种产自非洲的小型卷尾狗，它们从来都不叫唤。那么，这个程序将打印出什么呢？

```
class Dog {
    public static void bark() {
        System.out.print("woof ");
    }
}

class Basenji extends Dog {
    public static void bark() { }
}

public class Bark {
```

```
    public static void main(String args[]) {
        Dog woofer = new Dog();
        Dog nipper = new Basenji();
        woofer.bark();
        nipper.bark();
    }
}
```

随意地看一看，好像该程序应该只打印一个 woof。毕竟，Basenji 扩展自 Dog，并且它的 bark 方法定义为什么也不做。main 方法调用了 bark 方法，第一次是在 Dog 类型的 woofer 上调用，第二次是在 Basenji 类型的 nipper 上调用。巴辛吉小鬣狗并不会叫唤，但是很显然，这一只会。如果你运行该程序，就会发现它打印的是 woof woof。这只可怜的小家伙到底出什么问题了？

问题在于 bark 是一个静态方法，而对静态方法的调用不存在任何动态的分派机制[JLS 15.12.4.4]。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刻被选定的，而这种选定是基于修饰符的编译期类型而做出的，修饰符的编译期类型就是我们给出的方法调用表达式中圆点左边部分的名字。在本案中，两个方法调用的修饰符分别是变量 woofer 和 nipper，它们都被声明为 Dog 类型。因为它们具有相同的编译期类型，所以编译器使得它们调用的是相同的方法：Dog.bark。这也就解释了为什么程序打印出 woof woof。尽管 nipper 的运行期类型是 Basenji，但是编译器只会考虑其编译器类型。

要订正这个程序，直接从两个 bark 方法定义中移除掉 static 修饰符即可。这样，Basenji 中的 bark 方法将覆写而不是隐藏 Dog 中的 bark 方法，而该程序也将会打印出 woof，而不是 woof woof。通过覆写，你可以获得动态的分派；而通过隐藏，你却得不到这种特性。

当你调用了一个静态方法时，通常都是用一个类而不是表达式来标识它：例如，Dog.bark 或 Basenji.bark。当你在阅读一个 Java 程序时，你会期望类被用作为静态方法的修饰符，这些静态方法都是被静态分派的，而表达式被用作为实例方法的修饰符，这些实例方法都是被动态分派的。通过耦合类和变量的不同的命名规范，我们可以提供一个很强的可视化线索，用来表明一个给定的方法调用是动态的还是静态的。本谜题的程序使用了一个表达式作为静态方法调用的修饰符，这就误导了我们。千万不要用一个表达式来标识一个静态方法调用。

覆写的使用与上述的混乱局面搅到了一起。Basenji 中的 bark 方法与 Dog 中的 bark 方法具有相同的方法签名，这正是覆写的惯用方式，预示着要进行动态的分派。然而在本案中，该方法被声明为是 static 的，而静态方法是不能被覆写的；它们只能被隐藏，而这仅仅是因为你没有表达出你应该表达的意思。为了避免这样的混乱，千万不要隐藏静态方法。即便在子类中重用了超类中的静态方法的名称，也不会给你带来任何新的东西，但是却会丧失很多东西。

对语言设计者的教训是：对类和实例方法的调用彼此之间看起来应该具有明显的差异。第一种实现此目标的方式是不允许使用表达式作为静态方法的修饰符；第二种区分静态方法和实例方法调用的方式是使用不同的操作符，就像 C++那样；

第三种方式是通过完全抛弃静态方法这一概念来解决此问题，就像 Smalltalk 那样。

总之，要用类名来修饰静态方法的调用，或者当你在静态方法所属的类中去调用它们时，压根不去修饰这些方法，但是千万不要用一个表达式去修饰它们。还有就是要避免隐藏静态方法。所有这些原则合起来就可以帮助我们去消除那些容易令人误解的覆写，这些覆写需要对静态方法进行动态分派。

谜题 49：比生命更大

假如小报是可信的，那么摇滚之王“猫王”就会直到今天仍然在世。下面的程序用来估算猫王当前的腰带尺寸，方法是根据在公开演出中所观察到的他的体态发展趋势来进行投射。该程序中使用了 Calendar.getInstance().get(Calendar.YEAR)这个惯用法，它返回当前的日历年份。那么，该程序会打印出什么呢？

```
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private final int beltSize;
    private static final int CURRENT_YEAR =
        Calendar.getInstance().get(Calendar.YEAR);
    private Elvis() {
        beltSize = CURRENT_YEAR - 1930;
    }
    public int beltSize() {
        return beltSize;
    }
    public static void main(String[] args) {
        System.out.println("Elvis wears a size " +
                           INSTANCE.beltSize() + " belt.");
    }
}
```

第一眼看去，这个程序是在计算当前的年份减去 1930 的值。如果它是正确的，那么在 2006 年，该程序将打印出 Elvis wears a size 76 belt。如果你尝试着去运行该程序，你就会了解到小报是错误的，这证明你不能相信在报纸到读到的任何东西。该程序将打印出 Elvis wears a size -1930 belt。也许猫王已经在反物质的宇宙中定居了。

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[JLS 12.4]。让我们来看看其细节。Elvis 类的初始化是由虚拟机对其 main 方法的调用而触发的。首先，其静态域被设置为缺省值[JLS 4.12.5]，其中 INSTANCE 域被设置为 null，CURRENT_YEAR 被设置为 0。接下来，静态域初始器按照其出现的顺序执行。第一个静态域是 INSTANCE，它的值是通过调用 Elvis()构造器而计算出来的。

这个构造器会用一个涉及静态域 CURRENT_YEAR 的表达式来初始化 beltSize。通常，读取一个静态域是会引起一个类被初始化的事件之一，但是我们已经在初始化 Elvis 类了。递归的初始化尝试会直接被忽略掉[JLS 12.4.2, 第 3 步]。因此，CURRENT_YEAR 的值仍旧是其缺省值 0。这就是为什么 Elvis 的腰带尺寸变成了 -1930 的原因。

最后，从构造器返回以完成 Elvis 类的初始化，假设我们是在 2006 年运行该程序，那么我们就将静态域 CURRENT_YEAR 初始化成了 2006。遗憾的是，这个域现在所具有的正确值对于向 Elvis.INSTANCE.beltSize 的计算施加影响来说已经太晚了，beltSize 的值已经是-1930 了。这正是后续所有对 Elvis.INSTANCE.beltSize()的调用将返回的值。

该程序表明，在 final 类型的静态域被初始化之前，存在着读取它的值的可能，而此时该静态域包含的还只是其所属类型的缺省值。这是与直觉相违背的，因为我们通常会将 final 类型的域看作是常量。final 类型的域只有在其初始化表达式是常量表达式时才是常量[JLS 15.28]。

由类初始化中的循环所引发的问题是难以诊断的，但是一旦被诊断到，通常是很容易订正的。要想订正一个类初始化循环，需要重新对静态域的初始器进行排序，使得每一个初始器都出现在任何依赖于它的初始器之前。在这个程序中，CURRENT_YEAR 的声明属于在 INSTANCE 声明之前的情况，因为 Elvis 实例的创建需要 CURRENT_YEAR 被初始化。一旦 CURRENT_YEAR 的声明被移走，Elvis 就真的比生命更大了。

某些通用的设计模式本质上就是初始化循环的，特别是本谜题所展示的单例模式（Singleton）[Gamma95]和服务提供者框架（Service Provider Framework）[EJ Item 1]。类型安全的枚举模式（Typesafe Enum pattern）[EJ Item 21]也会引起类初始化的循环。5.0 版本添加了对这种使用枚举类型的模式的语言级支持。为了减少问题发生的可能性，对枚举类型的静态初始器做了一些限制[JLS 16.5, 8.9]。

总之，要当心类初始化循环。最简单的循环只涉及到一个单一的类，但是它们也可能涉及多个类。类初始化循环也并非总是坏事，但是它们可能会导致在静态域被初始化之前就调用构造器。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。

谜题 50：不是你的类型

本谜题要测试你对 Java 的两个最经典的操作符：instanceof 和转型的理解程度。下面的三个程序每一个都会做些什么呢？

```
public class Type1 {
    public static void main(String[] args) {
        String s = null;
        System.out.println(s instanceof String);
```

```
    }
}

public class Type2 {
    public static void main(String[] args) {
        System.out.println(new Type2() instanceof String);
    }
}

public class Type3 {
    public static void main(String args[]) {
        Type3 t3 = (Type3) new Object();
    }
}
```

第一个程序,Type1,展示了 instanceof 操作符应用于一个空对象引用时的行为。尽管 null 对于每一个引用类型来说都是其子类型,但是 instanceof 操作符被定义为在其左操作数为 null 时返回 false。因此，Type1 将打印 false。这被证明是实践中非常有用的行为。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。

第二个程序，Type2，展示了 instanceof 操作符在测试一个类的实例，以查看它是否是某个不相关的类的实例时所表现出来的行为。你可能会期望该程序打印出 false。毕竟，Type2 的实例不是 String 的实例，因此该测试应该失败，对吗？不，instanceof 测试在编译时刻就失败了，我们只能得到下面这样的出错消息：

```
Type2.java:3: inconvertible types
found   : Type2, required: java.lang.String
        System.out.println(new Type2() instanceof String);
                           ^
```

该程序编译失败是因为 instanceof 操作符有这样的要求：如果两个操作数的类型都是类，其中一个必须是另一个的子类型[JLS 15.20.2, 15.16, 5.5]。Type2 和 String 彼此都不是对方的子类型，所以 instanceof 测试将导致编译期错误。这个错误有助于让你警惕 instanceof 测试，它们可能并没有去做你希望它们做的事情。

第三个程序，Type3，展示了当要被转型的表达式的静态类型是转型类型的超类时，转型操作符的行为。与 instanceof 操作相同，如果在一个转型操作中的两种类型都是类，那么其中一个必须是另一个的子类型。尽管对我们来说，这个转型很显然会失败，但是类型系统还没有强大到能够洞悉表达式 new Object()的运行期类型不可能是 Type3 的一个子类型。因此，该程序将在运行期抛出 ClassCastException 异常。这有一点违背直觉：第二个程序完全具有实际意义，但是却不能编译；而这个程序没有任何实际意义，但是却可以编译。

总之，第一个程序展示了 instanceof 运行期行为的一个很有用的冷僻案例。第二个程序展示了其编译期行为的一个很有用的冷僻案例。第三个程序展示了转型操作符的行为的一个冷僻案例，在此案例中，编译器并不能将你从你所做荒唐的事中搭救出来，只能靠 VM 在运行期来帮你绷紧这根弦。

谜题 51：那个点是什么？

下面这个程序有两个不可变的值类（value class），值类即其实例表示值的类。第一个类用整数坐标来表示平面上的一个点，第二个类在此基础上添加了一点颜色。主程序将创建和打印第二个类的一个实例。那么，下面的程序将打印出什么呢？

```
class Point {
    protected final int x, y;
    private final String name; // Cached at construction time
    Point(int x, int y) {
        this.x = x;
        this.y = y;
        name = makeName();
    }

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

public class ColorPoint extends Point {
    private final String color;
    ColorPoint(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 ColorPoint(4, 2, "purple"));
    }
}
```

main 方法创建并打印了一个 ColorPoint 实例。println 方法调用了该 ColorPoint 实例的 toString 方法，这个方法是在 Point 中定义的。toString

方法将直接返回 name 域的值，这个值是通过调用 makeName 方法在 Point 的构造器中被初始化的。对于一个 Point 实例来说，makeName 方法将返回[x,y]形式的字符串。对于一个 ColorPoint 实例来说，makeName 方法被覆写为返回[x,y]:color 形式的字符串。在本例中，x 是 4，y 是 2，color 的 purple，因此程序将打印[4,2]:purple，对吗？不，如果你运行该程序，就会发现它打印的是[4,2]:null。这个程序出什么问题了呢？

这个程序遭遇了实例初始化顺序这一问题。要理解该程序，我们就需要详细跟踪该程序的执行过程。下面是该程序注释过的版本的列表，用来引导我们了解其执行顺序：

```
class Point {
    protected final int x, y;
    private final String name; // Cached at construction time
    Point(int x, int y) {
        this.x = x;
        this.y = y;
        name = makeName(); // 3. Invoke subclass method
    }

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

public class ColorPoint extends Point {
    private final String color;
    ColorPoint(int x, int y, String color) {
        super(x, y);          // 2. Chain to Point constructor
        this.color = color; // 5. Initialize blank final-Too late
    }
    protected String makeName() {
       // 4. Executes before subclass constructor body!
       return super.makeName() + ":" + color;
    }
    public static void main(String[] args) {
        // 1. Invoke subclass constructor
        System.out.println(new ColorPoint(4, 2, "purple"));
    }
}
```

在下面的解释中，括号中的数字引用的就是在上述注释版本的列表中的注释标号。首先，程序通过调用 ColorPoint 构造器创建了一个 ColorPoint 实例（1）。这个构造器以链接调用其超类构造器开始，就像所有构造器所做的那样（2）。超类构造器在构造过程中对该对象的 x 域赋值为 4，对 y 域赋值为 2。然后该超类构造器调用 makeName，该方法被子类覆写了（3）。

ColorPoint 中的 makeName 方法（4）是在 ColorPoint 构造器的程序体之前执行的，这就是问题的核心所在。makeName 方法首先调用 super.makeName，它将返回我们所期望的[4,2]，然后该方法在此基础上追加字符串“：”和由 color 域的值所转换成的字符串。但是此刻 color 域的值是什么呢？由于它仍处于待初始化状态，所以它的值仍旧是缺省值 null。因此，makeName 方法返回的是字符串“[4,2]:null”。超类构造器将这个值赋给 name 域（3），然后将控制流返回给子类的构造器。

这之后子类构造器才将“purple”赋予 color 域（5），但是此刻已经为时过晚了。color 域已经在超类中被用来初始化 name 域了，并且产生了不正确的值。之后，子类构造器返回，新创建的 ColorPoint 实例被传递给 println 方法，它适时地调用了该实例的 toString 方法，这个方法返回的是该实例的 name 域的内容，即“[4,2]:null”，这也就成为了程序要打印的东西。

本谜题说明：在一个 final 类型的实例域被赋值之前，存在着取用其值的可能，而此时它包含的仍旧是其所属类型的缺省值。在某种意义上，本谜题是谜题 49 在实例方面的相似物，谜题 49 是在 final 类型的静态域被赋值之前，取用了它的值。在这两种情况中，谜题都是因初始化的循环而产生的，在谜题 49 中，是类的初始化；而在本谜题中，是实例初始化。两种情况都存在着产生极大的混乱的可能性，但是它们之间有一个重要的差别：循环的类初始化是无法避免的灾难，但是循环的实例初始化总是可以且总是应该避免的。

无论何时，只要一个构造器调用了一个已经被其子类覆写了的方法，那么该问题就会出现，因为以这种方式被调用的方法总是在实例被初始化之前执行。要想避免这个问题，就千万不要在构造器中调用可覆写的方法，直接调用或间接调用都不行[EJ Item 15]。这项禁令应该扩展至实例初始器和伪构造器（pseudoconstructors）readObject 与 clone。（这些方法之所以被称为伪构造器，是因为它们可以在不调用构造器的情况下创建对象。）

你可以通过惰性初始化 name 域来订正该问题，即当它第一次被使用时初始化，以此取代积极初始化，即当 Point 实例被创建时初始化。

通过这种修改，该程序就可以打印出我们期望的[4,2]:purple。

```
class Point {
    protected final int x, y;
    private String name; // Lazily initialized
    Point(int x, int y) {
        this.x = x;
```

```
        this.y = y;
        // name initialization removed
    }

    protected String makeName() {
        return "[" + x + "," + y + "]";
    }
    // Lazily computers and caches name on first use
    public final synchronized String toString() {
        if (name == null)
            name = makeName();
        return name;
    }
}
```

尽管惰性加载可以订正这个问题，但是对于让一个值类去扩展另一个值类，并且在其中添加一个会对 euqals 比较方法产生影响的域的这种做法仍旧不是一个好主意。你无法在超类和子类上都提供一个基于值的 equals 方法，而同时又不违反 Object.equals 方法的通用约定，或者是不消除在超类和子类之间进行有实际意义的比较操作的可能性[EJ Item 7]。

循环实例初始化问题对语言设计者来说是问题成堆的地方。C++是通过在构造阶段将对象的类型从超类类型改变为子类类型来解决这个问题的。如果采用这种解决方法，本谜题中最开始的程序将打印[4,2]。我们发现没有任何一种流行的语言能够令人满意地解决这个问题。也许，我们值得去考虑，当超类构造器调用子类方法时，通过抛出一个不受检查的异常使循环实例初始化非法。

总之，在任何情况下，你都务必要记住：不要在构造器中调用可覆 写的方法。在实例初始化中产生的循环将是致命的。该问题的解决方案就是惰性初始化[EJ Items 13,48]。

谜题 52：合计数的玩笑

下面的程序在一个类中计算并缓存了一个合计数，并且在另一个类中打印了这个合计数。那么，这个程序将打印出什么呢？这里给一点提示：你可能已经回忆起来了，在代数学中我们曾经学到过，从 1 到 n 的整数总和是 n(n+1)/2。

```
class Cache {
    static {
        initializeIfNecessary();
    }
    private static int sum;
    public static int getSum() {
        initializeIfNecessary();
        return sum;
    }
```

```
    private static boolean initialized = false;
    private static synchronized void initializeIfNecessary() {
        if (!initialized) {
            for (int i = 0; i < 100; i++)
                sum += i;
            initialized = true;
        }
    }
}
public class Client {
    public static void main(String[] args) {
        System.out.println(Cache.getSum());
    }
}
```

草草地看一遍，你可能会认为这个程序从 1 加到了 100，但实际上它并没有这么做。再稍微仔细地看一看那个循环，它是一个典型的半开循环，因此它将从 0 循环到 99。有了这个印象之后，你可能会认为这个程序打印的是从 0 到 99 的整数总和。用前面提示中给出的公式，我们知道这个总和是 99×100/2，即 4,950。但是，这个程序可不这么想，它打印的是 9900，是我们所预期值的整整两倍。是什么导致它如此热情地翻倍计算了这个总和呢？

该程序的作者显然在确保 sum 在被使用前就已经在初始化这个问题上，经历了众多的麻烦。该程序结合了惰性初始化和积极初始化，甚至还用上了同步，以确保缓存在多线程环境下也能工作。看起来这个程序已经把所有的问题都考虑到了，但是它仍然不能正常工作。它到底出了什么问题呢？

与谜题 49 中的程序一样，该程序受到了类初始化顺序问题的影响。为了理解其行为，我们来跟踪其执行过程。在可以调用 Client.main 之前，VM 必须初始化 Client 类。这项初始化工作异常简单，我们就不多说什么了。Client.main 方法调用了 Cache.getsum 方法，在 getsum 方法可以被执行之前，VM 必须初始化 Cache 类。

回想一下，类初始化是按照静态初始器在源代码中出现的顺序去执行这些初始器的。Cache 类有两个静态初始器：在类顶端的一个 static 语句块，以及静态域 initialized 的初始化。静态语句块是先出现的，它调用了方法 initializeIfNecessary，该方法将测试 initialized 域。因为该域还没有被赋予任何值，所以它具有缺省的布尔值 false。与此类似，sum 具有缺省的 int 值 0。因此，initializeIfNecessary 方法执行的正是你所期望的行为，将 4,950 添加到了 sum 上，并将 initialized 设置为 true。

在静态语句块执行之后，initialized 域的静态初始器将其设置回 false，从而完成 Cache 的类初始化。遗憾的是，sum 现在包含的是正确的缓存值，但是 initialized 包含的却是 false：Cache 类的两个关键状态并未同步。

此后，Client 类的 main 方法调用 Cache.getSum 方法，它将再次调用 initializeIfNecessary 方法。因为 initialized 标志是 false，所以 initializeIfNecessary 方法将进入其循环，该循环将把另一个 4,950 添加到 sum 上，从而使其值增加到了 9,900。getSum 方法返回的就是这个值，而程序打印的也是它。

很明显，该程序的作者认为 Cache 类的初始化不会以这种顺序发生。由于不能在惰性初始化和积极初始化之间作出抉择，所以作者同时运用这二者，结果产生了大麻烦。要么使用积极初始化，要么使用惰性初始化，但是千万不要同时使用二者。

如果初始化一个域的时间和空间代价比较低，或者该域在程序的每一次执行中都需要用到时，那么使用积极初始化是恰当的。如果其代价比较高，或者该域在某些执行中并不会被用到，那么惰性初始化可能是更好的选择[EJ Item 48]。另外，惰性初始化对于打破类或实例初始化中的循环也可能是必需的（谜题 51）。

通过重排静态初始化的顺序，使得 initialized 域在 sum 被初始化之后不被复位到 false，或者通过移除 initialized 域的显式静态初始化操作，Cache 类就可以得到修复。尽管这样所产生的程序可以工作，但是它们仍旧是混乱的和病构的。Cache 类应该被重写为使用积极初始化，这样产生的版本很明显是正确的，而且比最初的版本更加简单。

使用这个版本的 Cache 类，程序就可以打印出我们所期望的 4950：

```
class Cache {
    private static final int sum = computeSum();
    private static int computeSum() {
        int result = 0;
        for (int i = 0; i < 100; i++)
            result += i;
        return result;
    }
    public static int getSum() {
        return sum;
    }
}
```

请注意，我们使用了一个助手方法来初始化 sum。助手方法通常都优于静态语句块，因为它让你可以对计算命名。只有在极少的情况下，你才必须使用一个静态语句块来初始化一个静态域，此时请将该语句块紧随该域声明之后放置。这提高了程序的清晰度，并且消除了像最初的程序中出现的静态初始化与静态语句块互相竞争的可能性。

总之，请考虑类初始化的顺序，特别是当初始化显得很重要时更是如此。请你执行测试，以确保类初始化序列的简洁。请使用积极初始化，除非你有某种很好的理由要使用惰性初始化，例如性能方面的因素，或者需要打破初始化循环。

谜题 53：按你的意愿行事

现在该轮到你写一些代码了。假设你有一个称为 Thing 的库类，它唯一的构造器将接受一个 int 参数：

```
public class Thing {
    public Thing(int i) { ... }
    ...
}
```

Thing 实例没有提供任何可以获取其构造器参数的值的途径。因为 Thing 是一个库类，所以你不具有访问其内部的权限，因此你不能修改它。

假设你想编写一个称为 MyThing 的子类，其构造器将通过调用 SomeOtherClass.func()方法来计算超类构造器的参数。这个方法返回的值被一个个的调用以不可预知的方式所修改。最后，假设你想将这个曾经传递给超类构造器的值存储到子类的一个 final 实例域中，以供将来使用。那么下面就是你自然会写出的代码：

```
public class MyThing extends Thing {
    private final int arg;
    public MyThing() {
        super(arg = SomeOtherClass.func());
        ...
    }
}
```

遗憾的是，这个程序是非法的。如果你尝试着去编译它，那么你将得到一条像下面这样的错误消息：

```
MyThing.java:
can't reference arg before supertype constructor has been called
        super(arg = SomeOtherClass.func());
                    ^
```

你怎样才能重写 MyThing 以实现想要的效果呢？MyThing()构造器必须是线程安全的：多个线程可能会并发地调用它。

这个解决方案内在地就是线程安全的和优雅的，它涉及对 MyThing 中第二个私有的构造器的运用：

```
public class MyThing extends Thing {
    private final int arg;

    public MyThing() {
        this(SomeOtherClass.func());
    }
    private MyThing(int i) {
        super(i);
```

```
            arg = i;
        }
}
```

这个解决方案使用了交替构造器调用机制(alternate constructor invocation)[JLS 8.8.7.1]。这个特征允许一个类中的某个构造器链接调用同一个类中的另一个构造器。在本例中，MyThing()链接调用了私有构造器 MyThing(int)，它执行了所需的实例初始化。在这个私有构造器中，表达式 SomeOtherClass.func()的值已经被捕获到了变量 i 中，并且它可以在超类构造器返回之后存储到 final 类型的域 param 中。

通过本谜题所展示的私有构造器捕获（Private Constructor Capture）惯用法是一种非常有用的模式，你应该把它添加到你的技巧库中。我们已经看到了某些真的是很丑陋的代码，它们本来是可以通过使用本模式而避免如此丑陋的。

谜题 54：Null 与 Void

下面仍然是经典的 Hello World 程序的另一个变种。那么，这个变种将打印什么呢？

```
public class Null {
    public static void greet() {
        System.out.println("Hello world!");
    }
    public static void main(String[] args) {
        ((Null) null).greet();
    }
}
```

这个程序看起来似乎应该抛出 NullPointerExceptioin 异常，因为其 main 方法是在常量 null 上调用 greet 方法，而你是不可以在 null 上调用方法的，对吗？嗯，某些时候是可以的。如果你运次该程序，就会发现它打印出了“Hello World!”

理解本谜题的关键是 Null.greet 是一个静态方法。正如你在谜题 48 中所看到的，在静态方法的调用中，使用表达式作为其限定符并非是一个好主意，而这也正是问题之所在。不仅表达式的值所引用的对象的运行期类型在确定哪一个方法将被调用时并不起任何作用，而且如果对象有标识的话，其标识也不起任何作用。在本例中，没有任何对象，但是这并不会造成任何区别。静态方法调用的限定表达式是可以计算的，但是它的值将被忽略。没有任何要求其值为非空的限制。

要想消除该程序中的混乱，你可以用它的类作为限定符来调用 greet 方法：

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

然而更好的方式是完全消除限定符：

```
public static void main(String[] args) {
     greet();
}
```
总之，本谜题的教训与谜题 48 的完全相同：要么用某种类型来限定静态方法调用，要么就压根不要限定它们。对语言设计者来说，应该不允许用表达式来污染静态方法调用的可能性存在，因为它们只会产生混乱。

谜题 55：特创论

某些时候，对于一个类来说，跟踪其创建出来的实例个数会非常用有，其典型实现是通过让它的构造器递增一个私有静态域来完成的。在下面的程序中，Creature 类展示了这种技巧，而 Creator 类对其进行了操练，将打印出已经创建的 Creature 实例的数量。那么，这个程序会打印出什么呢？

```
public class Creator {
    public static void main(String[] args) {
        for (int i = 0; i < 100; i++)
            Creature creature = new Creature();
        System.out.println(Creature.numCreated());
    }
}

class Creature {
    private static long numCreated = 0;
    public Creature() {
        numCreated++;
    }
    public static long numCreated() {
        return numCreated;
    }
}
```
这是一个捉弄人的问题。该程序看起来似乎应该打印 100，但是它没有打印任何东西，因为它根本就不能编译。如果你尝试着去编译它，你就会发现编译器的诊断信息基本没什么用处。下面就是 javac 打印的东西：

```
Creator.java:4: not a statement
            Creature creature = new Creature();
            ^

Creator.java:4: ';' expected
            Creature creature = new Creature();
                              ^
```
一个本地变量声明看起来像是一条语句，但是从技术上说，它不是；它应该是一个本地变量声明语句（local variable declaration statement）[JLS 14.4]。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[JLS 14.12-14]。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展中的语句和声明构成的。）

有两种方式可以订正这个问题。最显而易见的方式是将这个声明至于一个语句块中：

```
for (int i = 0; i < 100; i++) {
     Creature creature = new Creature();
}
```

然而，请注意，该程序没有使用本地变量 creature。因此，将该声明用一个无任何修饰的构造器调用来替代将更具实际意义，这样可以强调对新创建对象的引用正在被丢弃：

```
for (int i = 0; i < 100; i++)
     new Creature();
```

无论我们做出了上面的哪种修改，该程序都将打印出我们所期望的 100。

请注意，用于跟踪 Creature 实例个数的变量（numCreated）是 long 类型而不是 int 类型的。我们很容易想象到，一个程序创建出的某个类的实例可能会多余 int 数值的最大值，但是它不会多于 long 数值的最大值。

int 数值的最大值是 231-1，即大约 2.1×109，而 long 数值的最大值是 263-1，即大约 9.2×1018。当前，每秒钟创建 108 个对象是可能的，这意味着一个程序在 long 类型的对象计数器溢出之前，不得不运行大约三千年。即使是面对硬件速度的提升，long 类型的对象计数器也应该足以应付可预见的未来。

还要注意的是，本谜题中的创建计数策略并不是线程安全的。如果多个线程可以并行地创建对象，那么递增计数器的代码和读取计数器的代码都应该被同步：

```
// Thread-safe creation counter
class Creature {
    private static long numCreated;
    public Creature() {
        synchronized (Creature.class) {
            numCreated++;
        }
    }
    public static synchronized long numCreated() {
        return numCreated;
    }
}
```

或者，如果你使用的是 5.0 或更新的版本，你可以使用一个 AtomicLong 实例，它在面临并发时可以绕过对同步的需求。

```
// Thread-safe creation counter using AtomicLong;
import java.util.concurrent.atomic.AtomicLong;
class Creature {
```

```
    private static AtomicLong numCreated = new AtomicLong();
    public Creature() {
        numCreated.incrementAndGet();
    }
    public static long numCreated() {
        return numCreated.get();
    }
}
```

请注意，把 numCreated 声明为瞬时的是不足以解决问题的，因为 volatile 修饰符可以保证其他线程将看到最近赋予该域的值，但是它不能进行原子性的递增操作。

总之，一个本地变量声明不能被用作 for、while 或 do 循环中的重复执行语句，它作为一条语句只能出现在一个语句块中。另外，在使用一个变量来对实例的创建进行计数时，要使用 long 类型而不是 int 类型的变量，以防止溢出。最后，如果你打算在多线程中创建实例，要么将对实例计数器的访问进行同步，要么使用一个 AtomicLong 类型的计数器。

Java 谜题 6——库谜题

谜题 56：大问题 | 谜题 57：名字里有什么？ | 谜题 58：产生它的散列码 | 谜题 59：什么是差？ | 谜题 60：一行的方法 | 谜题 61：日期游戏 | 谜题 62：名字游戏 | 谜题 63：更多同样的问题 | 谜题 64：按余数编组 | 谜题 65：一种疑似排序的惊人传奇

谜题 56：大问题

作为一项热身活动，我们来测试一下你对 BigInteger 的了解程度。下面这个程序将打印出什么呢？

```
import java.math.BigInteger;
public class BigProblem {
    public static void main(String[ ] args) {
        BigInteger fiveThousand  = new BigInteger("5000");
        BigInteger fiftyThousand = new BigInteger("50000");
        BigInteger fiveHundredThousand = new BigInteger("500000");
        BigInteger total = BigInteger.ZERO;
        total.add(fiveThousand);
        total.add(fiftyThousand);
        total.add(fiveHundredThousand);
        System.out.println(total);
    }
}
```

你可能会认为这个程序会打印出 555000。毕竟，它将 total 设置为用 BigInteger 表示的 0，然后将 5,000、50,000 和 500,000 加到了这个变量上。如果你运行该程序，你就会发现它打印的不是 555000，而是 0。很明显，所有这些加法对 total 没有产生任何影响。

对此有一个很好理由可以解释：BigInteger 实例是不可变的。String、BigDecimal 以及包装器类型：Integer、Long、Short、Byte、Character、Boolean、Float 和 Double 也是如此，你不能修改它们的值。我们不能修改现有实例的值，对这些类型的操作将返回新的实例。起先，不可变类型看起来可能很不自然，但

是它们具有很多胜过与其向对应的可变类型的优势。不可变类型更容易设计、实现和使用；它们出错的可能性更小，并且更加安全[EJ Item 13]。

为了在一个包含对不可变对象引用的变量上执行计算，我们需要将计算的结果赋值给该变量。这样做就会产生下面的程序，它将打印出我们所期望的 555000：

```
import java.math.BigInteger;
public class BigProblem {
    public static void main(String[] args) {
        BigInteger fiveThousand  = new BigInteger("5000");
        BigInteger fiftyThousand = new BigInteger("50000");
        BigInteger fiveHundredThousand = new BigInteger("500000");
        BigInteger total = BigInteger.ZERO;
        total = total.add(fiveThousand);
        total = total.add(fiftyThousand);
        total = total.add(fiveHundredThousand);
        System.out.println(total);
    }
}
```

本谜题的教训是：不要被误导，认为不可变类型是可变的。这是一个在刚入门的 Java 程序员中很常见的错误。公正地说，Java 不可变类型的某些方法名促使我们走上了歧途。像 add、subtract 和 negate 之类的名字似乎是在暗示这些方法将修改它们所调用的实例。也许 plus、minus 和 negation 才是更好的名字。

对 API 设计来说，其教训是：在命名不可变类型的方法时，应该优选介词和名词，而不是动词。介词适用于带有参数的方法，而名词适用于不带参数的方法。对语言设计者而言，其教训与谜题 2 相同，那就是应该考虑对操作符重载提供有限的支持，这样算数操作符就可以作用于诸如 BigInteger 这样的数值型的引用类型。由此，即使是初学者也不会认为计算表达式 total + fiveThousand 将会对 total 的值产生任何影响。

谜题 57：名字里有什么？

下面的程序包含了一个简单的不可变类，它表示一个名字，其 main 方法将一个名字置于一个集合中，并检查该集合是否确实包含了该名字。那么，这个程序到底会打印出什么呢？

```
import java.util.*;
public class Name {
    private String first, last;
    public Name(String first, String last) {
        this.first = first;
        this.last = last;
    }
```

```
    public boolean equals(Object o) {
        if (!(o instanceof Name))
            return false;
        Name n = (Name)o;
        return n.first.equals(first) && n.last.equals(last);
    }
    public static void main(String[] args) {
        Set s = new HashSet();
        s.add(new Name("Mickey", "Mouse"));
        System.out.println(
            s.contains(new Name("Mickey", "Mouse")));
    }
}
```

一个 Name 实例由一个姓和一个名构成。两个 Name 实例在通过 equals 方法进行计算时，如果它们的姓相等且名也相等，则这两个 Name 实例相等。姓和名是用在 String 中定义的 equals 方法来比较的，两个字符串如果以相同的顺序包含相同的若干个字符，那么它们就相等。因此，两个 Name 实例如果表示相同的名字，那么它们就相等。例如，下面的方法调用将返回 true：

```
new Name("Mickey", "Mouse").equals(new Name("Mickey", "Mouse"))
```

该程序的 main 方法创建了两个 Name 实例，它们都表示 Mickey Mouse。该程序将第一个实例放置到了一个散列集合中，然后检查该集合是否包含第二个实例。这两个 Name 实例是相等的，因此看起来该程序似乎应该打印 true。如果你运行它，几乎可以肯定它将打印 false。那么这个程序出了什么问题呢？

这里的 bug 在于 Name 违反了 hashCode 约定。这看起来有点奇怪，因为 Name 连 hashCode 都没有，但是这确实是问题所在。Name 类覆写了 equals 方法，而 hashCode 约定要求相等的对象要具有相同的散列码。为了遵守这项约定，无论何时，只要你覆写了 equals 方法，你就必须同时覆写 hashCode 方法[EJ Item 8]。

因为 Name 类没有覆写 hashCode 方法，所以它从 Object 那里继承了其 hashCode 实现。这个实现返回的是基于标识的散列码。换句话说，不同的对象几乎总是产生不相等的散列值，即使它们是相等的也是如此。所以说 Name 没有遵守 hashCode 的约定，因此包含 Name 元素的散列集合的行为是不确定的。

当程序将第一个 Name 实例放置到散列集合中时，该集合就会在某个散列位置上放置这个实例对应的项。该集合是基于实例的散列值来选择散列位置的，这个散列值是通过实例的 hashCode 方法计算出来的。

当该程序在检查第二个 Name 实例是否包含在散列集合中时，它基于第二个实例的散列值来选择要搜索的散列位置。因为第二个实例有别于第一个实例，因此它极有可能产生不同的散列值。如果这两个散列值映射到了不同的位置，那么

contains 方法将返回 false：我们所喜爱的啮齿动物米老鼠就在这个散列集合中，但是该集合却找不到他。

假设两个 Name 实例映射到了相同的位置，那又会怎样呢？我们所了解的所有的 HashSet 实现都进行了一种优化，即每一项在存储元素本身之外，还存储了元素的散列值。在搜索某个元素时，这种实现通过遍历集合中的项，去拿存储在每一项中的散列值与我们想要查找的元素的散列值进行比较，从而选取适当的散列位置。只有在两个元素的散列值相等的情况下，这种实现才会认为这两个元素相等。这种优化是有实际意义的，因为比较散列码相对于比较元素来说，其代价要小得多。

对散列集合来说，这项优化并不足以使其能够搜索到正确的位置；两个 Name 实例必须具有相同的散列值才能让散列集合能够将它们识别为是相等的。该程序偶尔也会打印出 true，这是因为被连续创建的两个对象偶尔也会具有相同的标识散列码。一个粗略的实验表明，这种偶然性出现的概率大约是 25,000,000 分之一。这个实验的结果可能会因所使用的 Java 实现的不同而有所变化，但是在任何我们所知的 JRE 上，你基本上是不可能看到该程序打印出 true 的。

要想订正该程序，只需在 Name 类中添加一个恰当的 hashCode 方法即可。尽管任何其返回值仅有姓和名来确定的方法都可以满足 hashCode 的约定，但是高质量的散列函数应该尝试着对不同的名字返回不同的散列值。下面的方法就能够很好地实现这一点[EJ Item 8]。只要我们把该方法添加到了程序中，那么该程序就可以打印出我们所期望的 true：

```
public int hashCode() {
    return 37 * first.hashCode() + last.hashCode();
}
```

总之，当你覆写 equals 方法时，一定要记着覆写 hashCode 方法。更一般地讲，当你在覆写一个方法时，如果它具有一个通用的约定，那么你一定要遵守它。对于大多数在 Object 中声明的非 final 的方法，都需要注意这一点[EJ Chapter 3]。不采用这项建议就会导致任意的、不确定的行为。

谜题 58：产生它的散列码

本谜题试图从前一个谜题中吸取教训。下面的程序还是由一个 Name 类和一个 main 方法构成，这个 main 方法还是将一个名字放置到一个散列集合中，然后检查该集合是否包含了这个名字。然而，这一次 Name 类已经覆写了 hashCode 方法。那么下面的程序将打印出什么呢？

```
import java.util.*;
public class Name {
    private String first, last;
    public Name(String first, String last) {
        this.first = first; this.last = last;
```

```
    }
    public boolean equals(Name n) {
        return n.first.equals(first) && n.last.equals(last);
    }
    public int hashCode() {
        return 31 * first.hashCode() + last.hashCode();
    }
    public static void main(String[ ] args) {
        Set s = new HashSet();
        s.add(new Name("Donald", "Duck"));
        System.out.println(
            s.contains(new Name("Donald", "Duck")));
    }
}
```

与谜题 57 一样，该程序的 main 方法创建了两个 Name 实例，它们表示的是相同的名字。这一次使用的名字是 Donald Duck 而不是 Mickey Mouse，但是它们不应该有很大的区别。main 方法同样还是将第一个实例置于一个散列集合中，然后检查该集合中是否包含了第二个实例。这一次 hashCode 方法明显是正确的，因此看起来该程序应该打印 true。但是，表象再次欺骗了我们：它总是打印出 false。这一次又是哪里出错了呢？

这个程序的缺陷与谜题 57 中的缺陷很相似，在谜题 57 中，Name 覆写了 equals 方法，但是没有覆写 hashCode 方法；而在本谜题中，Name 覆写了 hashCode 方法，但是没有覆写 equals 方法。这并不是说 Name 没有声明一个 equals 方法，它确实声明了，但是那是个错误的声明。Name 类声明了一个参数类型是 Name 而不是 Object 的 equals 方法。这个类的作者可能想要覆写 equals 方法，但是却错误地重载了它[JLS 8.4.8.1, 8.4.9]。

HashSet 类是使用 equals(Object)方法来测试元素的相等性的；Name 类中声明一个 equals(Name)方法对 HashSet 不造成任何影响。那么 Name 是从哪里得到了它的 equals(Object)方法的呢？它是从 Object 哪里继承而来的。这个方法只有在它的参数与在其上调用该方法的对象完全相同时才返回 true。我们的程序中的 main 方法将一个 Name 实例插入到了散列集合中，并且测试另一个实例是否存在于该散列集合中，由此可知该测试一定是返回 false 的。对我们而言，两个实例可以代表那令人惊奇的水禽唐老鸭，但是对散列映射表而言，它们只是两个不相等的对象。

订正该程序只需用可以在谜题 57 中找到的覆写的 equals 方法来替换重载的 equals 方法即可。通过使用这个 equals 方法，该程序就可以打印出我们所期望的 true：

```
public boolean equals(Object o) {
    if (!(o instanceof Name))
        return false;
```

```
    Name n = (Name)o;
    return n.first.equals(first) && n.last.equals(last);
}
```

要让该程序可以正常工作，你只需增加一个覆写的 equals 方法即可。你不必剔除那个重载的版本，但是你最好是删掉它。重载为错误和混乱提供了机会[EJ Item 26]。如果兼容性要求强制你必须保留一个自身类型的 equals 方法，那么你应该用自身类型的重载去实现 Object 的重载，以此来确保它们具有相同的行为：

```
public boolean equals(Object o) { return o instanceof Name && equals((Name) o); }
```

本谜题的教训是：当你想要进行覆写时，千万不要进行重载。为了避免无意识地重载，你应该机械地对你想要覆写的每一个超类方法都拷贝其声明，或者更好的方式是让你的 IDE 帮你去做这些事。这样做除了可以保护你免受无意识的重载之害，而且还可以保护你免受拼错方法名之害。如果你使用的 5.0 或者更新的版本，那么对于那些意在覆写超类方法的方法，你可以将@Override 注释应用于每一个这样的方法的声明上：

```
@Override public Boolean equals(Object o) { ... }
```

在使用这个注释时，除非被注释的方法确实覆写了一个超类方法，否则它将不能编译。对语言设计者来说，值得去考虑在每一个覆写超类方法的方法声明上都添加一个强制性的修饰符。

谜题 59：什么是差？

下面的程序在计算一个 int 数组中的元素两两之间的差，将这些差置于一个集合中，然后打印该集合的尺寸大小。那么，这个程序将打印出什么呢？

```
import java.util.*;
public class Differences {
    public static void main(String[ ] args) {
        int vals[ ] = { 789, 678, 567, 456, 345, 234, 123, 012 };
        Set diffs = new HashSet();
        for (int i = 0; i < vals.length; i++)
            for (int j = i; j < vals.length; j++)
                diffs.add(vals[i] - vals[j]);
        System.out.println(diffs.size());
    }
}
```

外层循环迭代数组中的每一个元素，而内层循环从外层循环当前迭代到的元素开始迭代到数组中的最后一个元素。因此，这个嵌套的循环将遍历数组中每一种可

能的两两组合。（元素可以与其自身组成一对。）这个嵌套循环中的每一次迭代都计算了一对元素之间的差（总是正的），并将这个差存储到了集合中，集合是可以消除重复元素的。因此，本谜题就带来了一个问题，在由 vals 数组中的元素结成的对中，有多少唯一的正的差存在呢？

当你仔细观察程序中的数组时，会发现其构成模式非常明显：连续两个元素之间的差总是 111。因此，两个元素之间的差是它们在数组之间的偏移量之差的函数。如果两个元素是相同的，那么它们的差就是 0；如果两个元素是相邻的，那么它们的差就是 111；如果两个元素被另一个元素分割开了，那么它们的差就是 222；以此类推。看起来不同的差的数量与元素间不同的距离的数量是相等的，也就是等于数组的尺寸，即 8。如果你运行该程序，就会发现它打印的是 14。怎么回事呢？

上面的分析有一个小的漏洞。要想了解清楚这个缺陷，我们可以通过将 println 语句中的.size()这几个字符移除掉，来打印出集合中的内容。这么做会产生下面的输出：

```
[111, 222, 446, 557, 668, 113, 335, 444, 779, 224, 0, 333, 555, 666]
```

这些数字并非都是 111 的倍数。在 vals 数组中肯定有两个毗邻的元素的差是 113。如果你观察该数组的声明，不可能很清楚地发现原因所在：

```
int vals[ ] = { 789, 678, 567, 456, 345, 234, 123, 012 };
```

但是如果你打印数组的内容，你就会看见下面的内容：

```
[789, 678, 567, 456, 345, 234, 123, 10]
```

为什么数组中的最后一个元素是 10 而不是 12 呢？因为以 0 开头的整数类型字面常量将被解释成为八进制数值[JLS 3.10.1]。这个隐晦的结构是从 C 编程语言那里遗留下来东西，C 语言产生于 1970 年代，那时八进制比现在要通用得多。

一旦你知道了 012 == 10，就会很清楚为什么该程序打印出了 14：有 6 个不涉及最后一个元素的唯一的非 0 差，有 7 个涉及最后一个元素的非 0 差，还有 0，加在一起正好是 14 个唯一的差。订正该程序的方法更加明显：将八进制整型字面常量 012 替换为十进制整型字面常量 12。如果你这么做了，该程序将打印出我们所期望的 8。

本谜题的教训很简单：千万不要在一个整型字面常量的前面加上一个 0；这会使它变成一个八进制字面常量。有意识地使用八进制整型字面常量的情况相当少见，你应该对所有的这种特殊用法增加注释。对语言设计者来说，在决定应该包含什么特性时，应该考虑到其限制条件。当有所迟疑时，应该将它剔除在外。

谜题 60：一行的方法

现在该轮到你写一些代码了。下面的谜题每一个都可以用一个方法来解决，这些方法的方法体都只包含一行代码。各就各位，预备，编码！

- A. 编写一个方法，它接受一个包含元素的 List，并返回一个新的 List，它以相同的顺序包含相同的元素，只不过它把第二次以及后续出现的重复元素都剔除了。例如，如果你传递了一个包含”spam”,”sausage”,”spam”,”spam”,”bacon”,”spam”,”tomato”和”spam”的列表，那么你将得到一个包含”spam”,”sausage”,”bacon”,”tomato”的新列表。
- B. 编写一个方法，它接受一个由 0 个或多个由逗号分隔的标志所组成的字符串，并返回一个表示这些标志的字符串数组，数组中的元素的顺序与这些标志在输入字符串中出现的顺序相同。每一个逗号后面都可能会跟随 0 个或多个空格字符，这个方法忽略它们。例如，如果你传递的字符串是”fear, surprise, ruthless efficiency, an almost fanatical devotion to the Pope, nice red uniforms”，那么你得到的将是一个包含 5 个元素的字符串数组，这些元素是”fear”，”surprise”，”ruthless efficiency”，”an almost fanatical devotion to the Pope” 和 “nice red uniform”。
- C. 假设你有一个多维数组，出于调试的目的，你想打印它。你不知道这个数组有多少级，以及在数组的每一级中所存储的对象的类型。编写一个方法，它可以向你显示出在每一级上的所有元素。
- D. 编写一个方法，它接受两个 int 数值，并在第一个数值与第二个数值以二进制补码形式进行比较，具有更多的位被置位时，返回 true。

A. 众所周知，你可以通过把集合（collection）中的元素置于一个 Set 中将集合中的所有重复元素都消除掉。在本谜题中，你还被要求要保持最初的集合中的元素顺序。幸运的是，有一种 Set 的实现可以维护其元素被插入的顺序，它提供的导入性能接近 HashMap。它就是 LinkedHashSet，它是在 1.4 版本的 JDK 中被添加到 Java 平台中的。在内部，它是用一个链接列表来处理的，从而被实现为一个散列表。它还有一个映射表版本可供你使用，以定制缓存。一旦你了解了 LinkedHashSet，本谜题就很容易解决了。剩下唯一的关键就是你被要求要返回一个 List，因此你必须用 LinkedHashSet 的内容来初始化一个 List。把它们放到一块，就形成了下面的解决方案：

```
static <E> List<E> withoutDuplicates(List<E> original) {
    return new ArrayList<E>(new LinkedHashSet<E>(original));
}
```

B. 在将字符串解析成标志时，许多程序员都立刻想到了使用 StringTokenizer。这是最不幸的事情，自 1.4 版本开始，由于正则表达式被添加到了 Java 平台中（java.util.regex），StringTokenizer 开始变得过时了。如果你试图通过 StringTokenizer 来解决本谜题，那么你很快就会意识到它不是非常适合。通过使用正则表达式，它就是小菜一碟。为了在一行代码中解决本谜题，我们要使用很方便的方法 String.split，它接受一个描述标志分界符的正则表达式作为参数。如果你以前从来没有使用过正则表达式，那么它们看起来会显得有一点神秘，但是它们惊人地强大，值得我们好好学习一下：

```
static String[ ] parse(String string) {
     return string.split(",\\S*");
}
```

C. 这是一个讲究技巧的问题。你甚至不必去编写一个方法。这个方法在 5.0 或之后的版本中已经提供了，它就是 Arrays.deepToString。如果你传递给它一个对象引用的数组，它将返回一个精密的字符串表示。它可以处理嵌套数组，甚至可以处理循环引用，即一个数组元素直接或间接地引用了其嵌套外层的数组。事实上，5.0 版本中的 Arrays 类提供了一整套的 toString、equals 和 hashCode 方法，使你能够打印、比较或散列任何原始类型数组或对象引用数组的内容。

D. 为了在一行代码中解决该谜题，你需要了解在 5.0 版本中添加到 Java 平台中的一整套位操作方法。整数类型的包装器类（Integer、Long、Short、Byte 和 Char）现在支持通用的位处理操作，包括 highestOneBit、lowestOneBit、numberOfLeadingZeros、numberOfTrailingZeros、bitCount、rotateLeft、rotateRight、reverse、signum 和 reverseBytes。在本例中，你需要的是 Integer.bitCount，它返回的是一个 int 数值中被置位的位数：

```
static Boolean hasMoreBitsSet(int i, int j) {
     return (Integer.bitCount(i) > Integer.bitCount(j));
}
```

总之，Java 平台的每一个主版本都在其类库中隐藏了一些宝藏。本谜题的所有 4 个部分都依赖于这样的宝藏。每当该平台发布一个新版本时，你都应该研究就一下新特性和提高（new features and enhancements）页面，这样你就不会遗漏掉新版本提供的任何惊喜[Features-1.4, Features-5.0]。了解类库中有些什么可以节省你大量的时间和精力，并且可以提高你的程序的速度和质量。

谜题 61：日期游戏

下面的程序演练了 Date 和 Calendar 类的某些基本特性，它会打印出什么呢？

```
import java.util.*;
public class DatingGame {
    public static void main(String[ ] args) {
        Calendar cal = Calendar.getInstance();
        cal.set(1999, 12, 31); // Year, Month, Day
        System.out.print(cal.get(Calendar.YEAR) + " ");
        Date d = cal.getTime();
        System.out.println(d.getDay());
    }
}
```

该程序创建了一个 Calendar 实例，它应该表示的是 1999 年的除夕夜，然后该程序打印年份和日。看起来该程序应该打印 1999 31，但是它没有；它打印的是 2000 1。难道这是致命的 Y2K(千年虫)问题吗？

不，事情比我们想象的要糟糕得多：这是致命的 Date/Calendar 问题。在 Java 平台首次发布时，它唯一支持日历计算类的就是 Date 类。这个类在能力方面是受限的，特别是当需要支持国际化时，它就暴露出了一个基本的设计缺陷：Date 实例是易变的。在 1.1 版中，Calendar 类被添加到了 Java 平台中，以矫正 Date 的缺点，由此大部分的 Date 方法就都被弃用了。遗憾的是，这么做只能使情况更糟。我们的程序说明 Date 和 Calendar API 有许多问题。

该程序的第一个 bug 就位于方法调用 cal.set(1999,12,31)中。当月份以数字来表示时，习惯上我们将第一个月被赋值为 1。遗憾的是，Date 将一月表示为 0，而 Calendar 延续了这个错误。因此，这个方法调用将日历设置到了 1999 年第 13 个月的第 31 天。但是标准的（西历）日历只有 12 个月，该方法调用肯定应该抛出一个 IllegalArgumentException 异常，对吗？它是应该这么做，但是它并没有这么做。Calendar 类直接将其替换为下一年，在本例中即 2000 年的第一个月。这也就解释了我们的程序为什么打印出的第一个数字是 2000。

有两种方法可以订正这个问题。你可以将 cal.set 调用的第二个参数由 12 改为 11，但是这么做容易引起混淆，因为数字 11 会让读者误以为是 11 月。更好的方式是使用 Calendar 专为此目的而定义的常量，即 Calendar.DECEMBER。

该程序打印出的第二个数字又是怎么回事呢？cal.set 调用很明显是要把日历设置到这个月的第 31 天，Date 实例 d 表示的是与 Calendar 相同的时间点，因此它的 getDay 方法应该返回 31，但是程序打印的却是 1，这是怎么搞得呢？

为了找出原因，你必须先阅读一下文档，它叙述道 Date.getDay 返回的是 Date 实例所表示的星期日期，而不是月份日期。这个返回值是基于 0 的，从星期天开始计算。因此程序所打印的 1 表示 2000 年 1 月 31 日是星期一。请注意，相应的 Calendar 方法 get(Calendar.DAY_OF_WEEK) 不知为什么返回的是基于 1 的星期日期值，而不是像 Date 的对应方法那样返回基于 0 的星期日期值。

有两种方法可以订正这个问题。你可以调用 Date.date 这一名字极易让人混淆的方法，它返回的是月份日期。然而，与大多数 Date 方法一样，它已经被弃用了，因此你最好是将 Date 彻底抛弃，直接调用 Calendar 的 get(Calendar.DAY_OF_MONTH)方法。用这两种方法，该程序都可以打印出我们想要的 1999 31：

```
public class DatingGame {
    public static void main(String[] args) {
        Calendar cal = Calendar.getInstance();
        cal.set(1999, Calendar.DECEMBER, 31);
        System.out.print(cal.get(Calendar.YEAR) + " ");
        System.out.println(cal.get(Calendar.DAY_OF_MONTH));
```

```
    }
}
```

本谜题只是掀开了 Calendar 和 Date 缺陷的冰山一角。这些 API 简直就是雷区。Calendar 其他的严重问题包括弱类型（几乎每样事物都是一个 int）、过于复杂的状态空间、拙劣的结构、不一致的命名以及不一致的雨衣等。在使用 Calendar 和 Date 的时候一定要当心，千万要记着查阅 API 文档。

对 API 设计者来说，其教训是：如果你不能在第一次设计时就使它正确，那么至少应该在第二次设计时应该使它正确，绝对不能留到第三次设计时去处理。如果你对某个 API 的首次尝试出现了严重问题，那么你的客户可能会原谅你，并且会再给你一次机会。如果你第二次尝试又有问题，你可能会永远坚持这些错误了。

谜题 62：名字游戏

下面的程序将两个映射关系放置到了一个映射表中，然后打印它们的尺寸。那么，它会打印出什么呢？

```
import java.util.*;
public class NameGame {
    public static void main(String args[ ]) {
        Map<String, String> m =
                new IdentityHashMap<String, String>();
        m.put("Mickey", "Mouse");
        m.put("Mickey", "Mantle");
        System.out.println(m.size());
    }
}
```

对该程序的一种幼稚的分析认为，它应该打印 1。该程序虽然将两个映射关系放置到了映射表中，但是它们具有相同的键（Mickey）。这是一个映射表，不是一个多重映射表，所以棒球传奇人物（Mickey Mantle）应该覆盖了啮齿类动画明星（Mickey Mouse），从而只留下一个映射关系在映射表中。

更透彻一些的分析会对这个预测产生质疑。IdentityHashMap 的文档中叙述道：“这个类用一个散列表实现了 Map 接口，它在比较键时，使用的是引用等价性而不是值等价性”[Java-API]。换句话说，如果第二次出现的字符串字面常量“Mickey”被计算出来是与第一次出现的“Mickey”字符串不同的 String 实例的话，那么该程序应该打印 2 而不是 1。如此说来，该程序到底是打印 1，还是打印 2，抑或是其行为会根据不同的实现而有所变化？

如果你试着运行该程序，你就会发现，尽管我们那个幼稚的分析是有缺陷的，但是该程序正如这种分析所指出的一样，打印出来的是 1。这是为什么呢？语言规范保证了字符串是内存限定的，换句话说，相等的字符串常量同时也是相同的[JLS 15.28]。这可以确保在我们的程序中第二次出现的字符串字面常量

“Mickey”引用到了与第一次相同的 String 实例上，因此尽管我们使用了一个 IdentityHashMap 来代替诸如 HashMap 这样的通用目的的 Map 实现，但是对程序的行为却不会产生任何影响。我们那个幼稚的分析忽略了两个细节，但是这些细节造成的影响却彼此有效地抵消了。

本谜题的一个重要教训是：不要使用 IdentityHashMap，除非你需要其基于标识的语义；它不是一个通用目的的 Map 实现。这些语义对于实现保持拓扑结构的对象图转换（topology-preserving object graph transformations）非常有用，例如序列化和深层复制。我们得到的次要教训是字符串常量是内存限定的。正如在谜题 13 中所述，在任何时候，程序都应该尽量不依赖于这种行为去保证它们的操作正确。

谜题 63：更多同样的问题

下面的程序除了是面向对象的这一点之外，与前一个非常相似。因为从前一个程序中已经吸取了教训，这个程序使用了一个通用目的的 Map 实现，即一个 HashMap，来替代前一个程序的 IdentityHashMap。那么，这个程序会打印出什么呢？

```
import java.util.*;
public class MoreNames {
    private Map<String,String> m = new HashMap<String,String>();
    public void MoreNames() {
        m.put("Mickey", "Mouse");
        m.put("Mickey", "Mantle");
    }
    public int size() {
        return m.size();
    }
    public static void main(String args[ ]) {
        MoreNames moreNames = new MoreNames();
        System.out.println(moreNames.size());
    }
}
```

这个程序看起来很直观，其 main 方法通过调用无参数的构造器创建了一个 MoreNames 实例。这个 MoreNames 实例包含一个私有的 Map 域（m），它被初始化成一个空的 HashMap。该无参数的构造器似乎将两个映射关系放置到了映射表 m 中，这两个映射关系都具有相同的键（Mickey）。我们从前一个谜题已知，棒球手（Mickey Mantle）应该覆盖啮齿明星（Mickey Mouse），从而只留下一个映射关系。main 方法之后在 MoreNames 实例上调用了 size 方法，它会调用映射表 m 上的 size 方法，并返回结果，我们假设其为 1。这种分析还剩下一个问题：该程序打印的是 0 而不是 1。这种分析出了什么错呢？

问题在于 MoreNames 没有任何程序员声明的构造器。它拥有的只是一个返回值为 void 的实例方法，即 MoreNames，作者可能是想让它作为构造器的。遗憾的是，返回类型（void）的出现将想要的构造器声明变成了一个方法声明，而且该方法永远都不会被调用。因为 MoreNames 没有任何程序员声明的构造器，所以编译器会帮助（真的是在帮忙吗？）生成一个公共的无参数构造器，它除了初始化它所创建的域实例之外，不做任何事情。就像前面提到的，m 被初始化成了一个空的 HashMap。当在这个 HashMap 上调用 size 方法时，它将返回 0，这正是该程序打印出来的内容。

订正该程序很简单，只需将 void 返回类型从 MoreNames 声明中移除即可，这将使它从一个实例方法声明变成一个构造器声明。通过这种修改，该程序就可以打印出我们所期望的 1。

本谜题的教训是：不要因为偶然地添加了一个返回类型，而将一个构造器声明变成了一个方法声明。尽管一个方法的名字与声明它的类的名字相同是合法的，但是你千万不要这么做。更一般地讲，要遵守标准的命名习惯，它强制要求方法名必须以小写字母开头，而类名应该以大写字母开头。

对语言设计者来说，在没有任何程序员声明的构造器的情况下，自动生成一个缺省的构造器这种做法并非是一个很好的主意。如果确实生成了这样的构造器，也许应该让它们是私有的。有好几种其他的方法可以消除这个陷阱。一种方法是禁止方法名与类名相同，就像 C#所作的那样，另一种是彻底消灭所有的构造器，就像 Smalltalk 所作的那样。

谜题 64：按余数编组

下面的程序将生成整数对 3 取余的柱状图，那么，它将打印出什么呢？

```
public class Mod {
    public static void main(String[ ] args) {
        final int MODULUS = 3;
        int[] histogram = new int[MODULUS];
        // Iterate over all ints (Idiom from Puzzle 26)
        int i = Integer.MIN_VALUE;
        do {
            histogram[Math.abs(i) % MODULUS]++;
        } while (i++ != Integer.MAX_VALUE);
        for (int j = 0; j < MODULUS; j++)
            System.out.println(histogram[j] + " ");
    }
}
```

该程序首先初始化 int 数组 histogram，其每一个位置都为对 3 取余的一个数值而准备（0、1 和 2），所有这三个位置都被初始化为 0。然后，该程序在所有 232 个 int 数值上遍历变量 i，使用的是在谜题 26 中介绍的惯用法。因为整数取余

操作（%）在第一个操作数是负数时，可以返回一个负值，就像在谜题 1 中所描述的那样，所以该程序在计算 i 被 3 整除的余数之前，先取 i 的绝对值。然后用这个余数来递增数组位置的索引。在循环完成之后，该程序将打印 histogram 数组中的内容，它的元素表示对 3 取余得到 0、1 和 2 的 int 数值的个数。

该程序所打印的三个数字应该彼此大致相等，它们加起来应该等于 232。如果你想知道怎样计算出它们的精确值，那么你需要有一点数学气质，并仔细阅读下面两段话。否则，你可以跳过这两段话。

该程序打印的三个数字不可能精确地相等，因为它们必须加起来等于 232，这个数字不能被 3 除尽。如果你仔细观察 2 的连续幂级数对 3 取余的值，就会发现，它们在 1 和 2 之间交替变化：20 对 3 取余是 1，21 对 3 取余是 2，22 对 3 取余是 1，23 对 3 取余是 2，以此类推。每一个 2 的偶次幂对 3 取余的值都是 1，每一个 2 的奇次幂对 3 取余的值都是 2。因为 232 对 3 取余是 1，所以该程序所打印的三个数字中有一个将比另外两个大 1，但是它是哪一个呢？

该循环依次递增三个数组元素的数值，因此该循环最后递增的那个数值必然是最大的数值，它就是表示 Integer.MAX_VALUE 或(232-1)对 3 取余的数值。因为 231 是 2 的奇次幂，所以它对 3 取余应该得到 2，因此(232-1)对 3 取余将得到 1。该程序打印的三个数字中的第二个表示的就是对 3 取余得到 1 的 int 数值的个数，因此，我们期望这个值比第一个和最后一个数值大 1。

由此，该程序应该在运行了相当长的时间之后，打印(232/3)的较小值 (232/3)的较大值 (232/3)的较小值，即 1431655765 1431655766 1431655765。但是它真的是这么做的吗？不，它几乎立刻就抛出了下面的异常：

```
Exception in thread "main" ArrayIndexOutOfBoundsException: -2
        at Mod.main(Mod.java:9)
```

问题出在哪了呢？

问题在于该程序对 Math.abs 方法的使用上，它会导致错误的对 3 取余的数值。考虑一下当 i 为 -2 时所发生的事情，该程序计算 Math.abs(-2) % 3 的数值，得到 2，但是-2 对 3 取余应该得到 1。这可以解释为什么产生了不正确的统计结果，但是还有一个问题留待解决，为什么程序抛出了 ArrayIndexOutOfBoundsException 异常呢？这个异常表明该程序使用了一个负的数组索引，但是这肯定是不可能的：数组索引是通过的接受 i 的绝对值并计算这个绝对值被 3 整除时的余数而计算出来的。在计算一个非负的 int 数值整除一个正的 int 数值的余数时，可以保证将产生一个非负的结果[JLS 15.17.3]。我们又要问了，这里又出了什么问题呢？

要回答这个问题，我们必须要去看看 Math.abs 的文档。这个方法的名字有一点带有欺骗性，它几乎总是返回它的参数的绝对值，但是在有一种情况下，它做不到这一点。文档中叙述道：“如果其参数等于 Integer.MIN_VALUE，那么产生的结果与该参数相同，它是一个负数。”通过对这条知识的掌握，就可以很清楚地

知道为什么该程序立即抛出了 ArrayIndexOutOfBoundsException 异常。循环索引 i 的初始值是 Integer.MIN_VALUE，由 Math.abs(Integer.MIN_VALUE) % 3 所产生的数组索引等于 Integer.MIN_VALUE % 3，即 -2。

为了订正这个程序，我们必须用一个真正的取余操作来替代伪取余计算 (Math.abs(i) % MODULUS)。如果我们将这个表达式替换为对下面这个方法的调用，那么该程序就可以产生我们做期望的输出 1431655765 1431655766 1431655765：

```
private static int mod(int i, int modulus) {
    int result = i % modulus;
    return result < 0 ? result + modulus : result;
}
```

本谜题的教训是：Math.abs 不能保证一定会返回非负的结果。如果它的参数是 Integer.MIN_VALUE，或者对于 long 版本的实现传递的是 Long.MIN_VALUE，那么它将返回它的参数。这个方法在一般情况下是不会这么做的，上述这种行为的根源在于 2 的补码算数具有不对称性，这在谜题 33 中已经很详细的讨论过了。简单地讲，没有任何 int 数值可以表示 Integer.MIN_VALUE 的负值，也没有任何 long 数值可以表示 Long.MIN_VALUE 的负值。对类库的设计者来说，也许在将 Integer.MIN_VALUE 和 Long.MIN_VALUE 传递给 Math.abs 时，抛出 IllegalArgumentException 会显得更合理。然而，有人可能会争辩道，该方法的实际行为应该与 Java 内置的整数算术操作相一致，它们在溢出时并不会抛出异常。

谜题 65：一种疑似排序的惊人传奇

下面的程序使用定制的比较器，对一个由随机挑选的 Integer 实例组成的数组进行排序，然后打印了一个描述了数组顺序的单词。回忆一下，Comparator 接口只有一个方法，即 compare，它在第一个参数小于第二个参数时返回一个负数，在两个参数相等时返回 0，在第一个参数大于第二个参数时返回一个整数。这个程序是展示 5.0 版特性的一个样例程序。它使用了自动包装和解包、泛型和枚举类型。那么，它会打印出什么呢？

```
import java.util.*;
public class SuspiciousSort {
    public static void main(String[ ] args) {
        Random rnd = new Random();
        Integer[ ] arr = new Integer[100];
        for (int i = 0; i < arr.length; i++)
            arr[i] = rnd.nextInt();
        Comparator<Integer> cmp = new Comparator<Integer>() {
            public int compare(Integer i1, Integer i2) {
                return i2 - i1;
            }
```

```
        };
        Arrays.sort(arr, cmp);
        System.out.println(order(arr));
    }

    enum Order { ASCENDING, DESCENDING, CONSTANT, UNORDERED };

    static Order order(Integer[ ] a) {
        boolean ascending  = false;
        boolean descending = false;
        for (int i = 1; i < a.length; i++) {
            ascending  |= a[i] > a[i-1];
            descending |= a[i] < a[i-1];
        }
        if (ascending  && !descending)
            return Order.ASCENDING;
        if (descending && !ascending)
            return Order.DESCENDING;
        if (!ascending)
            return Order.CONSTANT;    // All elements equal
        return Order.UNORDERED;       // Array is not sorted
    }
}
```

该程序的 main 方法创建了一个 Integer 实例的数组，并用随机数对其进行了初始化，然后用比较器 cmp 对该数组进行排序。这个比较器的 compare 方法将返回它的第二个参数减去第一个参数的值，如果第二个参数表示的是比第一个参数大的数值，其返回值就是正的；如果这两个参数相等，其返回值为 0；如果第二个参数表示的是比第一个参数小的数值，其返回值就是负的。这种行为正好与 compare 方法通常的做法相反，因此，该比较器应该施加的是降序排列。

在对数组排序之后，main 方法将该数组传递给了静态方法 order，然后打印由这个方法返回的结果。该方法在数组中所有的元素都表示相同的数值时，返回 CONSTANT；在数组中每一对毗邻的元素中第二个元素都大于等于第一个元素时，返回 ASCENDING；在数组中每一对毗邻的元素中第二个元素都小于等于第一个元素时，返回 DESCENDING；在这些条件都不满足时，返回 UNORDERED。尽管理论上说，数组中的 100 个随机数有可能彼此都相等，但是这种奇特现象发生的非常小：232×99 分之一，即大约 5×10953 分之一。因此，该程序看起来应该打印 DESCENDING。如果你运行该程序，几乎可以肯定你将看到它打印的是 UNORDERED。为什么它会产生如此的行为呢？

order 方法很直观，它并不会说谎。Arrays.sort 方法已经存在许多年了，它工作得非常好。现在只有一个地方能够发现 bug 了：比较器。乍一看，这个比较器似乎不可能出错。毕竟，它使用的是标准的惯用法：如果你有两个数字，你想得到一个数值，其符号表示它们的顺序，那么你可以计算它们的差。这个惯用法至

少从 1970 年代早期就一直存在了，它在早期的 UNIX 里面被广泛地应用。遗憾的是，这种惯用法从来都没有正确地工作过。本谜题也许应该称为“白痴一般的惯用法的案例”。这种惯用法的问题在于定长的整数没有大到可以保存任意两个同等长度的整数之差的程度。当你在做两个 int 或 long 数值的减法时，其结果可能会溢出，在这种情况下我们就会得到错误的符号。

例如，请考虑下面的程序：

```
public class Overflow {
    public static void main(String[] args){
        int x = -2000000000;
        int z = 2000000000;
        System.out.println(x - z);
    }
}
```

很明显，x 比 z 小，但是程序打印的是 294967296，它是一个正数。既然这种比较的惯用法是有问题的，那么为什么它会被如此广泛地应用呢？因为它在大多数时间里可以正常工作的。它只在用来来进行比较的两个数字的差大于 Integer.MAX_VALUE 的时候才会出问题。这意味着对于许多应用而言，在实际使用中是不会看到这种错误的。更糟的是，它们被观察到的次数少之又少，以至于这个 bug 永远都不会被发现和订正。

那么这对于我们的程序的行为意味着什么呢？如果你查阅一下 Comparator 的文档，你就会看到它所实现的排序关系必须是可传递的（transitive），换句话说，(compare(x,y) > 0)&&(compare(y,z) > 0)蕴含着 compare(x,z) > 0。如果我们取 Overflow 例子中的 x 和 z，并取 y 为 0，那么我们的比较器在这些数值上就违反了可传递性。事实上，在所有随机选取的 int 数值对中，有四分之一该比较器都会返回错误的值。用这样的比较器来执行一个搜索或排序，或者用它去排序一个有序的集合，都会产生不确定的行为，就像我们在运行本谜题的程序时所看到的那样。出于数学上的倾向性，Comparator.compare 方法的一般约定要求比较器要产生一个全序（total order），但是这个比较器在数个计算上都未能做到这一点。

我们可以通过替换遵守上述一般约定的 Comparator 实现来订正我们的程序。因为我们只是想要反转自然排序的顺序，所以我们甚至可以不必编写我们自己的比较器。Collection 类提供了一个可以产生这种顺序的比较器。如果你用 Arrays.sort(arr,Collections.reverseOrder())来替代最初的 Arrays.sort 调用，该程序就可以打印出我们所期望的 DESCENDING。

或者，你可以编写你自己的比较器。下面的代码并不“聪明”，但是它可以工作，从而使该程序可以打印出我们所期望的 DESCENDING：

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

```
}
```

本谜题有数个教训，最具体的是：不要使用基于减法的比较器，除非你能够确保要比较的数值之间的差永远不会大于 Integer.MAX_VALUE [EJ Item 11]。更一般地讲，要意识到 int 的溢出，就像谜题 3、26 和 33 所讨论的那样。另一个教训是你应该避免“聪明”的代码。应该努力去编写清晰正确的代码，不要对它作任何优化，除非该优化被证明是必需的[EJ Item 37]。

对语言设计者来说，得到的教训与谜题 3、26 和 33 相同：也许真的值得去考虑支持某种形式整数算数运算，它不会在溢出时不抛出异常。还有就是可能应该在语言中提供一个三值的比较器操作符，就像 Perl 所作的那样（<=>操作符）。

Java 谜题 7——更多的类谜题

谜题 66：一件私事 | 谜题 67：对字符串上瘾 | 谜题 68：灰色的阴影 | 谜题 69：黑色的渐隐 | 谜题 70：一揽子交易 | 谜题 71：进口税 | 谜题 72：终极危难 | 谜题 73：你的隐私正在公开 | 谜题 74：同一性的危机 | 谜题 75：头还是尾 | 名字重用的术语表

谜题 66：一件私事

在下面的程序中，子类的一个域具有与超类的一个域相同的名字。那么，这个程序会打印出什么呢？

```
class Base {
    public String className = "Base";
}

class Derived extends Base {
    private String className = "Derived";
}

public class PrivateMatter {
    public static void main(String[ ] args) {
        System.out.println(new Derived().className);
    }
}
```

对该程序的表面分析可能会认为它应该打印 Derived，因为这正是存储在每一个 Derived 实例的 className 域中的内容。

更深入一点的分析会认为 Derived 类不能编译，因为 Derived 中的 className 变量具有比 Base 中的 className 变量更具限制性的访问权限。

如果你尝试着编译该程序，就会发现这种分析也不正确。该程序确实不能编译，但是错误却出在 PrivateMatter 中。

如果 className 是一个实例方法，而不是一个实例域，那么 Derived.className() 将覆写 Base.className()，而这样的程序是非法的。一个覆写方法的访问修饰符所提供的访问权限与被覆写方法的访问修饰符所提供的访问权限相比，至少要一样多[JLS 8.4.8.3]。

因为 className 是一个域，所以 Derived.className 隐藏（hide）了 Base.className，而不是覆盖了它[JLS 8.3]。对一个域来说，当它要隐藏另一个域时，如果隐藏域的访问修饰符提供的访问权限比被隐藏域的少，尽管这么做不可取的，但是它确实是合法的。事实上，对于隐藏域来说，如果它具有与被隐藏域完全无关的类型，也是合法的：即使 Derived.className 是 GregorianCalendar 类型的，Derived 类也是合法的。

在我们的程序中的编译错误出现在 PrivateMatter 类试图访问 Derived.className 的时候。尽管 Base 有一个公共域 className，但是这个域没有被继承到 Derived 类中，因为它被 Derived.className 隐藏了。在 Derived 类内部，域名 className 引用的是私有域 Derived.className。因为这个域被声明为是 private 的，所以它对于 PrivateMatter 来说是不可访问的。因此，编译器产生了类似下面这样的一条错误信息：

```
PrivateMatter.java:11: className has private access in Derived
        System.out.println(new Derived().className);
                                         ^
```

请注意，尽管在 Derived 实例中的公共域 Base.className 被隐藏了，但是我们还是可以通过将 Derived 实例转型为 Base 来访问到它。下面版本的 PrivateMatter 就可以打印出 Base：

```
public class PrivateMatter {
    public static void main(String[] args) {
        System.out.println(((Base)new Derived()).className);
    }
}
```

这说明了覆写与隐藏之间的一个非常大的区别。一旦一个方法在子类中被覆写，你就不能在子类的实例上调用它了（除了在子类内部，通过使用 super 关键字来方法）。然而，你可以通过将子类实例转型为某个超类类型来访问到被隐藏的域，在这个超类中该域未被隐藏。

如果你想让这个程序打印 Derived，也就是说，你想展示覆写行为，那么你可以用公共方法来替代公共域。在任何情况下，这都是一个好主意，因为它提供了更好的封装[EJ Item 19]。下面的程序版本就使用了这项技术，并且能够打印出我们所期望的 Derived：

```
class Base {
    public String getClassName() {
        return "Base";
    }
}

class Derived extends Base {
    public String getClassName() {
        return "Derived";
    }
}

public class PrivateMatter {
    public static void main(String[] args) {
        System.out.println(new Derived().getClassName());
    }
}
```

请注意，我们将 Derived 类中的 getClassName 方法声明成了 public 的，尽管在最初的程序中与其相对应的域是私有的。就像前面提到的那样，覆写方法的访问修饰符与它要覆写的方法的访问修饰符相比，所具有的限制性不能有任何降低。

本谜题的教训是隐藏通常都不是一个好主意。Java 语言允许你去隐藏变量、嵌套类型，甚至是静态方法（就像在谜题 48 所展示的那样），但是你不能认为你就应该去隐藏。隐藏的问题在于它将导致读者头脑的混乱。你正在使用一个被隐藏实体，或者是正在使用一个执行了隐藏的实体吗？要避免这类混乱，只需避免隐藏。

如果一个类要隐藏一个域，而用来隐藏该域的域具有的可访问性比被隐藏域更具限制性，就像我们最初的程序那样，那么这就违反了包容性（subsumption）原则，即大家所熟知的 Liskov 置换原则（Liskov Substitution Principle）[Liskov87]。这项原则叙述道，你能够对基类所作的任何事，都同样能够作用于其子类。包容性是面向对象编程的自然心理模型的一个不可分割的部分。无论何时，只要违反了这项原则，就会对程序的理解造成困难。还有其它数种用另一个域来隐藏某个域的方法也会违反包容性：例如，两个域具有不同的类型；一个域是静态的而另一个域不是；一个域是 final 的而另一个域不是；一个域是常量而另一个域不是；以及两个域都是常量但是它们具有不同的值。

对于语言设计者而言，应该考虑消除隐藏的可能性：例如，使所有的域都隐含地是私有的。如果这样做显得过于严苛，那么至少应该考虑对隐藏进行限制，以使其遵守包容性原则。

总之，当你在声明一个域、一个静态方法或一个嵌套类型时，如果其名字与基类中相对应的某个可访问的域、方法或类型相同，就会发生隐藏。隐藏是容易产生混乱的：违反包容性的隐藏域在某种意义上是特别有害的。更一般地讲，除了覆写之外，要避免名字重用。

谜题 67：对字符串上瘾

一个名字可以被用来引用位于不同包内的多个类。下面的程序就是在探究当你重用了一个平台类的名字时，会发生什么。你认为它会做些什么呢？尽管这个程序属于那种让你通常一看到就会感到尴尬的程序，但是你还是应该继续下去，把门锁上，把百叶窗拉上，然后试试看：

```
public class StrungOut {
    public static void main(String[] args) {
        String s = new String("Hello world");
        System.out.println(s);
    }
}

class String {
    private final java.lang.String s;
    public String(java.lang.String s) {
        this.s = s;
    }
    public java.lang.String toString() {
        return s;
    }
}
```

如果说这个程序有点让人讨厌的话，它看起来还是相当简单的。在未命名包中的 String 类就是一个 java.lang.String 实例的包装器，看起来该程序应该打印 Hello world。如果你尝试着运行该程序，你会发现你运行不了它，VM 将弹出了一个像下面这样的错误消息：

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

但是它肯定是一个 main 方法的：它就白纸黑字地写在那里。为什么 VM 找不到它呢？

VM 不能找到 main 方法是因为它并不在那里。尽管 StrungOut 有一个被命名为 main 的方法，但是它却具有错误的签名。一个 main 方法必须接受一个单一的字符串数组参数[JVMS 5.2]。VM 努力要告诉我们的是 StrungOut.main 接受的是由

我们的 String 类所构成的数组，它无论如何都与 java.lang.String 没有任何关系。

如果你确实需要编写自己的字符串类，看在老天爷的份上，千万不要称其为 String。要避免重用平台类的名字，并且千万不要重用 java.lang 中的类名，因为这些名字会被各处的程序自动加载。程序员习惯于看到这些名字以无限定的形式出现，并且会很自然地认为这些名字引用的是我们所熟知的 java.lang 中的类。如果你重用了这些名字的某一个，那么当这个名字在其自己的包内被使用时，该名字的无限定形式将会引用到新的定义上。

要订正该程序，只需为这个非标准的字符串类挑选一个合理的名字即可。该程序下面的这个版本很明显是正确的，而且它比最初的版本要更易于理解。它将打印出如你所期望的 Hello World:

```
public class StrungOut {
    public static void main(String[ ] args) {
        MyString s = new MyString("Hello world");
        System.out.println(s);
    }
}

class MyString {
    private final java.lang.String s;
    public MyString(java.lang.String s) { this.s = s;}
    public java.lang.String toString() { return s;}
}
```

宽泛地讲，本谜题的教训就是要避免重用类名，尤其是 Java 平台类的类名。千万不要重用 java.lang 包内的类名，相同的教训也适用于类库的设计者。Java 平台的设计者已经在这个问题上栽过数次了，著名的例子有 java.sql.Date，它与 java.util.Date 和 org.omg.CORBA.Object 相冲突。与在本章中的许多其他谜题一样，这个教训是有关你在除了覆写之外的其他情况应该避免名字重用这一原则的一个具体实例。对平台实现者来说，其教训是诊断信息应该清晰地解释失败的原因。VM 应该可以很容易地将没有任何具有正确签名的 main 方法的情况与根本就没有任何 main 方法的情况区分开。

谜题 68：灰色的阴影

下面的程序在相同的范围内具有两个名字相同的声明，并且没有任何明显的方式可以在它们二者之间做选择。这个程序会打印 Black 吗？它会打印 White 吗？甚至，它是合法的吗？

```
public class ShadesOfGray {
    public static void main(String[] args){
        System.out.println(X.Y.Z);
    }
```

```
}

class X {
    static class Y {
        static String Z = "Black";
    }
    static C Y = new C();
}

class C {
    String Z = "White";
}
```

没有任何显而易见的方法可以确定该程序应该打印 Black 还是 White。编译器通常会拒绝模棱两可的程序，而这个程序看起来肯定是模棱两可的。因此，它似乎应该是非法的。如果你试着运行它，就会发现它是合法的，并且会打印出 White。你怎样才能事先了解这一切呢？

可以证明，在这样的上下文环境中，有一条规则决定着程序的行为，即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[JLS 6.5.2]。变量名将遮掩（obscure）类型名[JLS 6.3.2]。相似地，变量名和类型名可以遮掩包名。这条规则真的是相当地晦涩，任何依赖于它的程序都极有可能使它的读者晕头转向。

幸运的是，遵守标准的 Java 命名习惯的程序继续从来都不会遇上这个问题。类应该以一个大写字母开头，以 MixedCase 的形式书写；变量应该以一个小写字母开头，以 mixedCase 的形式书写；而常量应该以一个大写字母开头，以 ALL_CAPS 的方式书写。单个的大写字母只能用于类型参数，就像在泛型接口 Map<K,V>中那样。包名应该以 lower.case 的方式命名[JLS 6.8]。

为了避免常量名与类名的冲突，在类名中应该将首字母缩拼词当作普通的词处理[EJ Item 38]。例如，一个表示全局唯一标识符的类应该被命名为 Uuid，而不是 UUID，尽管其首字母缩拼词通常被写为 UUID。（Java 平台库就违反了这项建议，因为它具有 UUID、URL 和 URI 这样的类名。）为了避免变量名与包名的冲突，请不要使用顶层的包名或领域名作为变量的名字，特别是不要将一个变量命名为 com、org、net、edu、java 或 javax。

要想移除 ShadesOfGray 这个程序中的所有不明确性，只需以遵守命名习惯的方式对其重写即可。很明显，下面的程序将打印 Black。作为一种附加的好处，当你大声朗读这个程序时，听起来还最初的那个程序是完全一样的。

```
public class ShadesOfGray {
    public static void main(String[ ] args){
        System.out.println(Ex.Why.Z);
    }
```

```
}

class Ex {
    static class Why {
        static String Z = "Black";
    }
    static See y = new See();
}

class See {
    String Z = "White";
}
```

总之，应该遵守标准的命名习惯以避免不同的命名空间之间的冲突，还有一个原因就是如果你违反这些习惯，那么你的程序将让人难以辨认。同样，为了避免变量名与通用的顶层包名相冲突，请使用 MixedCase 风格的类名，即使其名字是首字母缩拼词也应如此。通过遵守这些规则，你就可以确保你的程序永远不会遮掩类名或包名。再次说明一下，这里列举的仍然是你应该在覆写之外的情况中避免名字重用的一个实例。对语言设计者来说，应该考虑去消除遮掩的可能性。C# 是通过将域和嵌套类置于相同的命名空间来实现这一点的。

谜题 69：黑色的渐隐

假设你不能修改前一个谜题（谜题 68）中的 X 和 C 这两个类。你能否编写一个类，其 main 方法将读取 X.Y 类中的 Z 域的值，然后打印它。注意，不能使用反射。

本谜题初看起来是不可能实现的。毕竟，X.Y 类被具有相同名字的一个域给遮掩了，因此对其命名的尝试将引用到该域上。

事实上，我们是可以引用到一个被遮掩的类型名的，其技巧就是在某一种特殊的语法上下文环境中使用该名字，在该语法上下文环境中允许出现一个类型但是不允许出现一个变量。在转型表达式的括号中间的部分就是这样一种上下文环境。下面的程序通过使用这种技术解决了这个谜题，并且将打印出我们所期望的 Black：

```
public class FadeToBlack {
    public static void main(String[] args){
        System.out.println(((X.Y)null).Z);
    }
}
```

请注意，我们是用一个具有 X.Y 类型的表达式来访问 X.Y 类的 Z 域的。就像我们在谜题 48 和 54 中所看到的，用一个表达式而不是类型名来访问一个静态成员是合法的，但却是一种有问题的用法。

不借助这种有问题的用法，而是通过在一个类声明的 extends 子句中使用一个被遮掩的类这种方式，你也可以解决本谜题。因为基类总是一种类型，出现在 extends 子句中的名字从来都不会被解析为变量名。下面的程序就展示了这项技术，它也会打印出 Black：

```
public class FadeToBlack {
    static class Xy extends X.Y{ }
    public static void main(String[ ] args){
        System.out.println(Xy.Z);
    }
}
```

如果你使用的 5.0 或更新的版本，那么通过在一个类型变量声明的 extends 子句中使用 X.Y 这种方式，你也可以解决本谜题：

```
public class FadeToBlack {
    public static <T extends X.Y> void main(String[] args){
        System.out.println(T.Z);
    }
}
```

总之，要解决由类型被变量遮掩而引发的问题，需要按照标准的命名习惯来重命名类型和变量，就像在谜题 68 中所讨论的那样。如果做不到这一点，那么你应该在只允许类型名的上下文环境中使用被遮掩的类型名。幸运的话，你将永远不需要凭借这种对程序的变形来解决问题，因为大多数的类库作者都很明智，他们都避免了必需使用这种变形的有问题的用法。然而，如果你确实发现自己身处这种境地，那么你最好是要了解这个问题需要解决。

谜题 70：一揽子交易

下面这个程序设计在不同的包中的两个类的交互，main 方法位于 hack.TypeIt 中。那么，这个程序会打印什么呢？

```
package hack;
import click.CodeTalk;
public class TypeIt {
    private static class ClickIt extends CodeTalk {
        void printMessage() {
            System.out.println("Hack");
        }
    }

    public static void main(String[ ] args) {
        ClickIt clickit = new ClickIt();
        clickit.doIt();
```

```
    }
}

package click;
public class CodeTalk {
    public void doIt() {
        printMessage();
    }

    void printMessage() {
        System.out.println("Click");
    }
}
```

本谜题看起来很直观。Hack.TypeIt 的 main 方法对 TypeIt.ClickIt 类实例化，然后调用其 doIt 方法，该方法是从 CodeTalk 继承而来。接着，该方法调用 printMessage 方法，它在 TypeIt.ClickIt 中被声明为打印 Hack。然而，如果你运行该程序，它打印的将是 Click。怎么会这样呢？

上面的分析做出了一个不正确的假设，即 Hack.TypeIt.ClickIt.printMessage 方法覆写了 click.CodeTalk.printMessage 方法。一个包内私有的方法不能被位于另一个包中的某个方法直接覆写[JLS 8.4.8]。在程序中的这两个 twoMessage 方法是无关的，它们仅仅是具有相同的名字而已。当程序在 hack 包内调用 printMessage 方法时，运行的是 hack.TypeIt.ClickIt.printMessage 方法。这个方法将打印 Click，这也就解释了我们所观察到的行为。

如果你想让 hack.TypeIt.ClickIt 中的 printMessage 方法覆写在 Click.CodeTalk 中的该方法，那么你必须在 Click.CodeTalk 中的该方法声明之前添加 protected 或 public 修饰符。要使该程序能够编译，你还必须在 hack.TypeIt.ClickIt 的覆写声明的前面添加一个修饰符，该修饰符与你在 Click.CodeTalk 的 printMessage 方法上放置的修饰符相比，所具备的限制性不能更多[JLS 8.4.8.3]。换句话说，两个 printMessage 方法可以都被声明为是 public 的，也可以都被声明为是 protected 的，或者，超类中的方法被声明为是 protected，而子类中的方法被声明为是 public 的。无论你执行了上述三种修改中的任何一种，该程序都将打印 Hack，从而表明确实发生了覆写。

总之，包内私有的方法不能直接被包外的方法声明所覆写。尽管包内私有的访问权限和覆写结合到一起会导致某种混乱，但是 Java 当前的行为是允许使用包的，以支持比单个的类更大的抽象封装。包内私有的方法是它们所属包的实现细节，在包外重用它们的名字是不会对包内产生任何影响的。

谜题 71：进口税

在 5.0 版中，Java 平台引入了大量的可以使操作数组变得更加容易的工具。下面这个谜题使

用了变量参数、自动包装、静态导入（请查看 http://java.sun.com/j2se/5.0/docs/guide/language [Java-5.0]）以及便捷方法 Arrays.toString（请查看谜题 60）。那么，这个程序会打印什么呢？

```
import static java.util.Arrays.toString;
class ImportDuty {
    public static void main(String[ ] args) {
        printArgs(1, 2, 3, 4, 5);
    }
    static void printArgs(Object... args) {
        System.out.println(toString(args));
    }
}
```

你可能会期望该程序打印[1,2,3,4,5]，实际上它确实会这么做，只要它能编译。令人沮丧的是，看起来编译器找不到恰当的 toString 方法：

```
ImportDuty.java:9:Object.toString()can't be applied to(Object[])
        System.out.println(toString(args));
                           ^
```

是不是编译器的理解力太差了？为什么它会尝试着去应用 Object.toString()呢？它与调用参数列表并不匹配，而 Arrays.toString(Object[ ])却可以完全匹配。

编译器在选择在运行期将被调用的方法时，所作的第一件事就是在肯定能找到该方法的范围内挑选[JLS 15.12.1]。编译器将在包含了具有恰当名字的方法的最小闭合范围内进行挑选，在我们的程序中，这个范围就是 ImportDuty 类，它包含了从 Object 继承而来的 toString 方法。在这个范围中没有任何可以应用于 toString(args)调用的方法，因此编译器必须拒绝该程序。

换句话说，我们想要的 toString 方法没有在调用点所处的范围内。导入的 toString 方法被 ImportDuty 从 Object 那里继承而来的具有相同名字的方法所遮蔽（shade）了[JLS 6.3.1]。遮蔽与遮掩（谜题 68）非常相像，二者的关键区别是一个声明只能遮蔽类型相同的另一个声明：一个类型声明可以遮蔽另一个类型声明，一个变量声明可以遮蔽另一个变量声明，一个方法声明可以遮蔽另一个方法声明。与其形成对照的是，变量声明可以遮掩类型和包声明，而类型声明也可以遮掩包声明。

当一个声明遮蔽了另一个声明时，简单名将引用到遮蔽声明中的实体。在本例中，toString 引用的是从 Object 继承而来的 toString 方法。简单地说，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。这导致的后果之一就是与 Object 的方法具有相同名字的静态方法不能通过静态导入工具而得到使用。

既然你不能对 Arrays.toString 使用静态导入，那么你就应该用一个普通的导入声明来代替。下面就是 Arrays.toString 应该被正确使用的方式：

```
import java.util.Arrays;
class ImportDuty {
    static void printArgs(Object... args) {
        System.out.println(Arrays.toString(args));
    }
}
```

如果你特别强烈地想避免显式地限定 Arrays.toString 调用，那么你可以编写你自己的私有静态转发方法：

```
private  static String toString(Object[] a) {
    return Arrays.toString(a);
}
```

静态导入工具所专门针对的情况是：程序中会重复地使用另一个类的静态元素，而每一次用到的时候都进行限定又会使程序变得乱成一锅粥。在这类情况中，静态导入工具可以显著地提高可读性。这比通过实现接口来继承其常量要安全得多，而实现接口这种做法是你从来都不应该采用的 [EJ Item 17]。然而，滥用静态导入工具也会损害可读性，因为这会使得静态成员的类在何处被使用显得非常不清晰。应该有节制地使用静态导入，只有在非常需要的情况下才应该使用它们。

对 API 设计者来说，要意识到当某个方法的名字已经出现在某个作用域内时，静态导入工具并不能被有效地作用于该方法上。这意味着静态导入不能用于那些与通用接口中的方法共享方法名的静态方法，而且也从来不能用于那些与 Object 中的方法共享方法名的静态方法。再次说明一下，本谜题所要说明的仍然是你在覆写之外的情况中使用名字重用通常都会产生混乱。我们通过重载、隐藏和遮掩看清楚了这一点，现在我们又通过遮蔽看到了同样的问题。

谜题 72：终极危难

本谜题旨在检验当你试图隐藏一个 final 域时将要发生的事情。下面的程序将做些什么呢？

```
class Jeopardy {
    public static final String PRIZE = "$64,000";
}

public class DoubleJeopardy extends Jeopardy {
    public static final String PRIZE = "2 cents";
    public static void main(String[ ] args) {
        System.out.println(DoubleJeopardy.PRIZE);
    }
}
```

因为在 Jeopardy 中的 PRIZE 域被声明为是 public 和 final 的，你可能会认为 Java 语言将阻止你在子类中重用该域名。毕竟，final 类型的方法不能被覆写或隐藏。如果你尝试着运行该程序，就会发现它可以毫无问题地通过编译，并且将打印 2 cents。出什么错了呢？

可以证明，final 修饰符对方法和域而言，意味着某些完全不同的事情。对于方法，final 意味着该方法不能被覆写（对实例方法而言）或者隐藏（对静态方法而言）[JLS 8.4.3.3]。对于域，final 意味着该域不能被赋值超过一次[JLS 8.3.1.2]。关键字相同，但是其行为却完全不相关。

在该程序中，final 域 DoubleJeopardy.PRIZE 隐藏了 final 域 Jeopardy.PRIZE，其净损失达到了$63,999.98。尽管我们可以隐藏一个域，但是通常这都是一个不好的念头。就像我们在谜题 66 中所讨论的，隐藏域可能会违反包容性，并且会混淆我们对类型与其成员之间的关系所产生的直觉。

如果你想保证在 Jeopardy 类中的奖金可以保留到子类中，那么你应该用一个 final 方法来代替 final 域：

```
class Jeopardy {
    private static final String PRIZE = "$64,000";
    public static final String prize() {
         return PRIZE;
   }
}
```

对语言设计者来说，其教训是应该避免在不相关的概念之间重用关键字。一个关键字应该只在密切相关的概念之间重用，这样可以帮助程序员构建关于易混淆的语言特性之间的关系的印象。在 Java 的 final 关键字这一案例中，重用就导致了混乱。应该注意的是，作为一种有年头的语言来说，在无关的概念之间重用关键字是它的一种自然趋势，这样做可以避免引入新的关键字，而引入新的关键字会对语言的稳定性造成极大的损害。当语言设计者在考虑该怎么做时，总是在两害相权取其轻。

总之，要避免在无关的变量或无关的概念之间重用名字。对无关的概念使用有区别的名字有助于让读者和程序员区分这些概念。

谜题 73：你的隐私正在公开

私有成员，即私有方法、域和类型这些概念的幕后思想是它们只是实现细节：一个类的实现者可以随意地添加一个新的私有成员，或者修改和移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。换句话说，私有成员被包含它们的类完全封装了。

遗憾的是，在这种严密的盔甲保护中仍然存在细小的裂缝。例如，序列化就可以打破这种封装。如果使一个类成为可序列化的，并且接受缺省的序列化形式，那

么该类的私有实例域将成为其导出 API 的一部分[EJ Item 54,55]。当客户正在使用现有的被序列化对象时，对私有表示的修改将会导致异常或者是错误的行为。

但是编译期的错误又会怎么样呢？你能否写出一个 final 的“库”类和“客户”类，这两者都可以毫无问题地通过编译，然后在库类中添加一个私有成员，使得库类仍然能够编译，而客户类却再也不能编译了？

如果你的解谜方案是要对库类添加一个私有构造器，以抑制通过缺省的公共构造器而创建实例的行为，那么你只是一知半解。本谜题要求你添加一个私有成员，严格地讲，构造器不是成员[JLS 6.4.3]。

本谜题有数个解谜方案，其中一个是使用遮蔽：

```
package library;
public final class Api {
     // private static class String{ }
     public static String newString() {
          return new String();
     }
}

package client;
import library.Api;
public class Client {
    String s = Api.newString();
}
```

如上编写，该程序就可以毫无问题地通过编译。如果我们不注释掉 library.Api 中的局部类 String 的私有声明，那么 Api.newString 方法就再也不会返回 java.lang.String 类型了，因此变量 Client.s 的初始化将不能通过编译：

```
client/Client.java:4: incompatible types
found: library.Api.String, required: java.lang.String
     String s = Api.newString();
                            ^
```

尽管我们所做的文本修改仅仅是添加了一个私有类声明，但是我们间接地修改了一个现有公共方法的返回类型，而这是一个不兼容的 API 修改，因为我们修改了一个被导出 API 所使用的名字的含义。

这种解谜方案的数个变种也都可以实现这个目的。被遮蔽类型也可以来自一个外围类而不是来自 java.lang；你可以遮蔽一个变量而不是一个类型，而被遮蔽变量可以来自一个 static import 声明或者是来自一个外围类。

不修改类库的某个被导出成员的类型也可以解决本谜题。下面就是这样的一个解谜方案，它使用的是隐藏而不是遮蔽：

```
package library;
class ApiBase {
     public static final int ANSWER = 42;
}

public final class Api extends ApiBase() {
     // private static final int ANSWER = 6 * 9;
}

package client;
import library.Api;
public class Client {
    int answer = Api.ANSWER;
}
```

如上编写，该程序就可以毫无问题地通过编译。如果我们不注释掉 library.Api 中的私有声明，那么客户类将不能通过编译：

```
client/Client.java:4: ANSWER has private access in library.Api
int answer = Api.ANSWER;
                 ^
```

这个新的私有域 Api.ANSWER 隐藏了公共域 ApiBase.ANSWER，而这个域本来是应该被继承到 Api 中的。因为新的域被声明为是 private 的，所以它不能被 Client 访问。这种解谜方案的数个变种也都可以实现这个目的。你可以用隐藏一个实例域去替代隐藏一个静态域，或者用隐藏一个类型去替代隐藏一个域。

你还可以用遮掩来解决本谜题。所有的解谜方案都是通过重用某个名字来破坏客户类。重用名字是危险的；应该避免隐藏、遮蔽和遮掩。是不是对此已经耳熟能详了？很好！

谜题 74：同一性的危机

下面的程序是不完整的，它缺乏对 Enigma 的声明，这个类扩展自 java.lang.Object。请为 Enigma 提供一个声明，它可以使该程序打印 false：

```
public class Conundrum {
    public static void main(String[] args) {
        Enigma e = new Enigma();
        System.out.println(e.equals(e));
    }
}
```

噢，还有一件事：你不能覆写 equals 方法。

乍一看，这似乎不可能实现。因为 Object.equals 方法将测试对象的同一性，通过 Enigma 传递给 equals 方法的对象肯定是与其自身相同的。如果你不能覆写 Object.equals 方法，那么 main 方法必然打印 true，对吗？

别那么快下结论，伙计。尽管本谜题禁止你覆写（override）Object.equals 方法，但是你是可以重载（overload）它的，这也就引出了下面的解谜方案：

```
final class Enigma {
    // Don't do this!
    public Boolean equals(Enigma other){
        return false;
    }
}
```

尽管这个声明能够解决本谜题，但是它的做法确实非常不好的。它违反了谜题 58 的建议：如果同一个方法的两个重载版本都可以应用于某些参数，那么它们应该具有相同的行为。在本例中，e.equals(e)和 e.equals((Object)e)将返回不同的结果，其潜在的混乱是显而易见的。

然而，有一种解谜方案是不会违反这项建议的：

```
final class Enigma {
    public Enigma() {
        System.out.println(false);
        System.exit(0);
    }
}
```

可能会有些争论，这个解谜方案似乎违背了本谜题的精神：能够产生我们想要的输出的 println 调用出现在了构造器中，而不是在 main 方法中。然而，它确实解决了这个谜题，你不得不承认它很伶俐。

这里的教训，可以参阅前面的 8 个谜题和谜题 58。如果你重载了一个方法，那么一定要确保所有的重载版本行为一致。

谜题 75：头还是尾？

这个程序的行为在 1.4 版和 5.0 版的 Java 平台上会有些变化。这个程序在这些版本上会分别做些什么呢？（如果你只能访问 5.0 版本的平台，那么你可以在编译的时候使用-source 1.4 标记，以此来模拟 1.4 版的行为。）

```
import java.util.Random;
public class CoinSide {
```

```
private static Random rnd = new Random();
    public static CoinSide flip() {
        return rnd.nextBoolean() ?
            Heads.INSTANCE : Tails.INSTANCE;
    }
    public static void main(String[ ] args) {
        System.out.println(flip());
    }
}

class Heads extends CoinSide {
    private Heads() { }
    public static final Heads INSTANCE = new Heads();
    public String toString() {
        return "heads";
    }
}

class Tails extends CoinSide {
    private Tails() { }
    public static final Tails INSTANCE = new Tails();
    public String toString() {
        return "tails";
    }
}
```

该程序看起来根本没有使用 5.0 版的任何新特性，因此很难看出来为什么它们在行为上应该有差异。事实上，该程序在 1.4 或更早版本的平台上是不能编译的：

```
CoinSide.java:7:
incompatible types for ?: neither is a subtype of the other
second operand: Heads
third operand : Tails
              return rnd.nextBoolean() ?
                                       ^
```

条件操作符（? :）的行为在 5.0 版本之前是非常受限的[JLS2 15.25]。当第二个和第三个操作数是引用类型时，条件操作符要求它们其中的一个必须是另一个的子类型。Heads 和 Tails 彼此都不是对方的子类型，所以这里就产生了一个错误。为了让这段代码能够编译，你可以将其中一个操作数转型为二者的公共超类：

```
return rnd.nextBooleam() ?
(CoinSide)Heads.INSTANCE : Tails.INSTANCE;
```

在 5.0 或更新的版本中，Java 语言显得更加宽大了，条件操作符在第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。在实际使用中，这种变化的主要结果就是条件操作符做正确的事情的情况更多了，而给出编译期错误的情况更少了。对于我们当中的语言菜鸟来说，作用于引用类型的条件操作符的结果所具备的编译期类型与在第二个和第三个操作数上调用下面的方法的结果相同：

```
T choose(T a,T b) { }
```

本谜题所展示的问题在 1.4 和更早的版本中发生得相当频繁，迫使你必须插入只是为了遮掩你的代码的真实目的而进行的转型。这就是说，该谜题本身是人为制造的。在 5.0 版本之前，使用类型安全的枚举模式来编写 CoinSide 对程序员来说会显得更自然一些[EJ Item 21]：

```
import java.util.Random;
public class CoinSide {
    public static final CoinSide HEADS = new CoinSide("heads");
    public static final CoinSide TAILS = new CoinSide("tails");
    private final String name;

    private CoinSide(String name) {
        this.name = name;
    }

    public String toString() {
        return name;
    }

    private static Random rnd = new Random();

    public static CoinSide flip() {
        return rnd.nextBoolean() ? HEADS : TAILS;
    }

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

在 5.0 或更新的版本中，自然会将 CoinSide 当作是一个枚举类型来编写：

```
public enum CoinSide {
    HEADS, TAILS;
    public String toString() {
```

```
        return name().toLowerCase();
    }
    // flip 和 main 与上面的 1.4 版上的实现一样
}
```

本谜题的教训是：应该升级到最新的 Java 平台版本上。较新的版本都包含许多让程序员更轻松的改进，你并不需要费力去学习怎样利用所有的新特性，有些新特性不需要你付出任何努力就可以给你带来实惠。对语言和类库的设计者来说，得到的教训是：不要让程序员去做那些语言或类库本可以帮他们做的事。

名字重用的术语表

覆写（override）

一个实例方法可以覆写（override）在其超类中可访问到的具有相同签名的所有实例方法[JLS 8.4.8.1]，从而使能了动态分派（dynamic dispatch）；换句话说，VM 将基于实例的运行期类型来选择要调用的覆写方法[JLS 15.12.4.4]。覆写是面向对象编程技术的基础，并且是唯一没有被普遍劝阻的名字重用形式：

```
class Base {
    public void f() { }
}

class Derived extends Base {
    public void f() { } // overrides Base.f()
}
```

隐藏（hide）

一个域、静态方法或成员类型可以分别隐藏（hide）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[JLS 8.3, 8.4.8.2, 8.5]：

```
class Base {
    public static void f() { }
}

class Derived extends Base {
    private static void f() { } // hides Base.f()
}
```

重载（overload）

在某个类中的方法可以重载（overload）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[JLS 8.4.9, 15.12.2]：

```
class CircuitBreaker {
     public void f(int i)      { } // int overloading
     public void f(String s) { } // String overloading
}
```

遮蔽（shadow）

一个变量、方法或类型可以分别遮蔽（shadow）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[JLS 6.3.1]：

```
class WhoKnows {
     static String sentence = "I don't know.";
     public static void main(String[ ] args) {
          String sentence = "I know!";   // shadows static field
          System.out.println(sentence);  // prints local variable
     }
}
```

尽管遮蔽通常是被劝阻的，但是有一种通用的惯用法确实涉及遮蔽。构造器经常将来自其所在类的某个域名重用为一个参数，以传递这个命名域的值。这种惯用法并不是没有风险，但是大多数 Java 程序员都认为这种风格带来的实惠要超过其风险：

```
class Belt {
     private final int size;
     public Belt(int size) { // Parameter shadows Belt.size
          this.size = size;
     }
}
```

遮掩（obscure）

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。遵守命名习惯就可以极大地消除产生遮掩的可能性[JLS 6.3.2, 6.5]：

```
public class Obscure {
     static String System;  // Obscures type java.lang.System
     public static void main(String[ ] args) {
          // Next line won't compile: System refers to static field
          System.out.println("hello, obscure world!");
     }
}
```

Java 谜题 8——更多的库谜题

谜题 76：乒乓 | 谜题 77：搞乱锁的妖怪 | 谜题 78：反射的污染 | 谜题 79：这是狗的生活 | 谜题 80：更深层的反射 | 谜题 81：烧焦（字符化）到无法识别 | 谜题 82：啤酒爆炸 | 谜题 83：诵读困难者的一神论 | 谜题 84：被粗暴地中断 | 谜题 85：惰性初始化

谜题 76：乒乓

下面的程序全部是由同步化（synchronized）的静态方法组成的。那么它会打印出什么呢？在你每次运行这段程序的时候，它都能保证会打印出相同的内容吗？

```
public class PingPong{
  public static synchronized void main(String[] a){
    Thread t = new Thread(){
      public void run(){ pong(); }
    };
    t.run();
    System.out.print( "Ping" );
  }
  static synchronized void pong(){
    System.out.print( "Pong" );
  }
}
```

在多线程程序中，通常正确的观点是程序每次运行的结果都有可能发生变化，但是上面这段程序总是打印出相同的内容。在一个同步化的静态方法执行之前，它会获取与它的 Class 对象相关联的一个管程（monitor）锁[JLS 8.4.3.6]。所以在上面的程序中，主线程会在创建第二个线程之前获得与 PingPong.class 相关联的那个锁。只要主线程占有着这个锁，第二个线程就不可能执行同步化的静态方法。具体地讲，在 main 方法打印了 Ping 并且执行结束之后，第二个线程才能执行 pong 方法。只有当主线程放弃那个锁的时候，第二个线程才被允许获得这个锁并且打印 Pong 。根据以上的分析，我们似乎可以确信这个程序应该总是打

印 PingPong。但是这里有一个小问题：当你尝试着运行这个程序的时候，你会发现它总是会打印 PongPing。到底发生了什么呢？

正如它看起来的那样奇怪，这段程序并不是一个多线程程序。不是一个多线程程序？怎么可能呢？它肯定会生成第二个线程啊。喔，对的，它确实是创建了第二个线程，但是它从未启动这个线程。相反地，主线程会调用那个新的线程实例的 run 方法，这个 run 方法会在主线程中同步地运行。由于一个线程可以重复地获得某个相同的锁［JLS 17.1］，所以当 run 方法调用 pong 方法的时候，主线程就被允许再次获得与 PingPong.class 相关联的锁。pong 方法打印了 Pong 并且返回到了 run 方法，而 run 方法又返回到 main 方法。最后，main 方法打印了 Ping，这就解释了我们看到的输出结果是怎么来的。

要订正这个程序很简单，只需将 t.run 改写成 t.start。这么做之后，这个程序就会如你所愿的总是打印出 PingPong 了。

这个教训很简单：当你想调用一个线程的 start 方法时要多加小心，别弄错成调用这个线程的 run 方法了。遗憾的是，这个错误实在是太普遍了，而且它可能很难被发现。或许这个谜题的教训应该是针对 API 的设计者的：如果一个线程没有一个公共的 run 方法，那么程序员就不可能意外地调用到它。Thread 类之所以有一个公共的 run 方法，是因为它实现了 Runnable 接口，但是这种方式并不是必须的。另外一种可选的设计方案是：使用组合（composition）来替代接口继承(interface inheritance)，让每个 Thread 实例都封装一个 Runnable。正如谜题 47 中所讨论的，组合通常比继承更可取。这个谜题说明了上述的原则甚至对于接口继承也是适用的。

谜题 77：搞乱锁的妖怪

下面的这段程序模拟了一个小车间。程序首先启动了一个工人线程，该线程在停止时间到来之前会一直工作（至少是假装在工作），然后程序安排了一个定时器任务（timer task）用来模拟一个恶毒的老板，他会试图阻止停止时间的到来。最后，主线程作为一个善良的老板会告诉工人停止时间到了，并且等待工人停止工作。那么这个程序会打印什么呢？

```
import java.util.*;
public class Worker extends Thread {
    private volatile boolean quittingTime = false;
    public void run() {
        while (!quittingTime)
            pretendToWork();
        System.out.println("Beer is good");
    }

    private void pretendToWork() {
        try {
            Thread.sleep(300); // Sleeping on the job?
```

```
            } catch (InterruptedException ex) { }
        }
        // It's quitting time, wait for worker - Called by good boss
        synchronized void quit() throws InterruptedException {
            quittingTime = true;
            join();
        }
        // Rescind quitting time - Called by evil boss
        synchronized void keepWorking() {
            quittingTime = false;
        }

        public static void main(String[] args)
                throws InterruptedException {
            final Worker worker = new Worker();
            worker.start();
            Timer t = new Timer(true); // Daemon thread
            t.schedule(new TimerTask() {
                public void run() { worker.keepWorking(); }
            }, 500);
            Thread.sleep(400);
            worker.quit();
        }
}
```

想要探究这个程序到底做了什么的最好方法就是手动地模拟一下它的执行过程。下面是一个近似的时间轴，这些时间点的数值是相对于程序的开始时刻进行计算的：

- 300 ms：工人线程去检查易变的 quittingTime 域，看看停止时间是否已经到了。这个时候并没有到停止时间，所以工人线程会回去继续“工作”。
- 400ms：作为善良的老板的主线程会去调用工人线程的 quit 方法。主线程会获得工人线程实例上的锁（因为 quit 是一个同步化的方法），将 quittingTime 的值设为 true，并且调用工人线程上的 join 方法。这个对 join 方法的调用并不会马上返回，而是会等待工人线程执行完毕。
- 500m：作为恶毒的老板定时器任务开始执行。它将试图调用工人线程的 keepWorking 方法，但是这个调用将会被阻塞，因为 keepWorking 是一个同步化的方法，而主线程当时正在执行工人线程上的另一个同步化方法（quit 方法）。
- 600ms：工人线程会再次检查停止时间是否已经到来。由于 quittingTime 域是易变的，那么工人线程肯定会看到新的值 true，所以它会打印 Beer is good 并结束运行。这会让主线程对 join 方法的调用执行返回，随后主线程也结束了运行。而定时器线程是后台的，所以它也会随之结束运行，整个程序也就结束了。

所以，我们会认为程序将运行不到 1 秒钟，打印 Beer is good ，然后正常的结束。但是当你尝试运行这个程序的时候，你会发现它没有打印任何东西，而是一直处于挂起状态（没有结束）。我们的分析哪里出错了呢？

其实，并没有什么可以保证上述几个交叉的事件会按照上面的时间轴发生。无论是 Timer 类还是 Thread.sleep 方法，都不能保证具有实时（real-time）性。这就是说，由于这里计时的粒度太粗，所以上述几个事件很有可能会在时间轴上互有重叠地交替发生。100 毫秒对于计算机来说是一段很长的时间。此外，这个程序被重复地挂起；看起来好像有什么其他的东西在工作着，事实上，确实是有这种东西。

我们的分析存在着一个基本的错误。在 500ms 时，当作为恶毒老板的定时器任务运行时，根据时间轴的显示，它对 keepWorking 方法的调用会被阻塞，因为 keepWorking 是一个同步化的方法并且主线程正在同一个对象上执行着同步化方法 quit(在 Thread.join 中等待着)。这些都是对的，keepWorking 确实是一个同步化的方法，并且主线程确实正在同一个对象上执行着同步化的 quit 方法。即使如此，定时器线程仍然可以获得这个对象上的锁，并且执行 keepWorking 方法。这是如何发生的呢？

问题的答案涉及到了 Thread.join 的实现。这部分内容在关于该方法的文档中（JDK 文档）是找不到的，至少在迄今为止发布的文档中如此，也包括 5.0 版。在内部，Thread.join 方法在表示正在被连接（join）的那个 Thread 实例上调用 Object.wait 方法。这样就在等待期间释放了该对象上的锁。在我们的程序中，这就使得作为恶毒老板的定时器线程能够堂而皇之的将 quittingTime 重新设置成 false，尽管此时主线程正在执行同步化的 quit 方法。这样的结果是，工人线程永远不会看到停止时间的到来，它会永远运行下去。作为善良的老板的主线程也就永远不会从 join 方法中返回了。

使这个程序产生了预料之外的行为的根本原因就是 WorkerThread 类的作者使用了实例上的锁来确保 quit 方法和 keepWorking 方法的互斥，但是这种用法与超类（Thread）内部对该锁的用法发生了冲突。这里的教训是：除非有关于某个类的详细说明作为保证，否则千万不要假设库中的这个类对它的实例或类上的锁会做（或者不会做）某些事情。对于库的任何调用都可能会产生对 wait、notify、notifyAll 方法或者某个同步化方法的调用。所有这些，都可能对应用级的代码产生影响。

如果你需要获得某个锁的完全控制权，那么就要确定没有任何其他人能够访问到它。如果你的类扩展了库中的某个类，而这个库中的类可能使用了它的锁，或者如果某些不可信的人可能会获得对你的类的实例的访问权，那么请不要使用与这个类或它的实例自动关联的那些锁。取而代之的，你应该在一个私有的域中创建一个单独的锁对象。在 5.0 版本发布之前，用于这种锁对象的正确类型只有 Object 或者它的某个普通的子类。从 5.0 版本开始，
java.util.concurrent.locks 提供了 2 种可选方案：ReentrantLock 和 ReentrantReadWriteLock。相对于 Object 类，这 2 个类提供了更好的机动性，但是它们使用起来也要更麻烦一点。它们不能被用在同步化的语句块

（synchronized block）中，而且必须辅以 try-finally 语句对其进行显式的获取和释放。

订正这个程序最直接的方法是添加一个 Object 类型的私有域作为锁，并且在 quit 和 keepWorking 方法中对这个锁对象进行同步。通过上述修改之后，该程序就会打印出我们所期望的 Beer is good。可以看出，该程序能够产生正确行为并不依赖于它必须遵从我们前面分析的时间轴：

```
private final Object lock = new Object();
// It's quitting time, wait for worker - Called by good boss
void quit() throws InterruptedException{
    synchronized (lock){
        quittingTime = true;
        join();
    }
}
// Rescind quitting time - Called by evil boss
void keepWorking(){
    synchronized(lock){
        quittingTime = false;
    }
}
```

另外一种可以修复这个程序的方法是让 Worker 类实现 Runnable 而不是扩展 Thread，然后在创建每个工人线程的时候都使用 Thread(Runnable)构造器。这样可以将每个 Worker 实例上的锁与其线程上的锁进行解耦。这是一个规模稍大一些的重构。

正如库类对锁的使用会干扰应用程序一样，应用程序中对锁的使用也会干扰库类。例如，在迄今为止发布的所有版本的 JDK（包括 5.0 版本）中，为了创建一个新的 Thread 实例，系统都会去获取 Thread 类上的锁。而执行下面的代码就可以阻止任何新线程的创建：

```
synchronized(Thread.class){
    Thread.sleep(Long.MAX_VALUE);
}
```

总之，永远不要假设库类会（或者不会）对它的锁做某些事情。为了隔离你自己的程序与库类对锁的使用，除了那些专门设计用来被继承的库类之外，请避免继承其它库类 [EJ Item 15]。为了确保你的锁不会遭受外部的干扰，可以将它们设为私有以阻止其他人对它们的访问。

对于语言设计者来说，需要考虑的是为每个对象都关联一个锁是否是合适的。如果你决定这么做了，就需要考虑限制对这些锁的访问。在 Java 中，锁实际上是对象的公共属性，或许它们变为私有的会更有意义。同时请记住在 Java 语言中，

一个对象实际上就是一个锁：你在对象本身之上进行同步。如果每个对象都有一个锁，而且你可以通过调用一个访问器方法来获得它，这样或许会更有意义。

谜题 78：反射的污染

这个谜题举例说明了一个关于反射的简单应用。这个程序会打印出什么呢？

```
import java.util.*;
import java.lang.reflect.*;
public class Reflector {
    public static void main(String[] args) throws Exception {
        Set<String> s = new HashSet<String>();
        s.add("foo");
        Iterator it = s.iterator();
        Method m = it.getClass().getMethod("hasNext");
        System.out.println(m.invoke(it));
    }
}
```

这个程序首先创建了一个只包含单个元素的集合(set)，获得了该集合上的迭代器，然后利用反射调用了迭代器的 hasNext 方法，最后打印出此该方法调用的结果。由于该迭代器尚未返回该集合中那个唯一的元素，hasNext 方法应该返回 true。然而，运行这个程序却得到了截然不同的结果：

```
Exception in thread "main" java.lang.IllegalAccessException:
   Class Reflector can not access a member of class HashMap$HashIterator
with modifiers "public"
        at Reflection.ensureMemberAccess(Reflection.java:65)
        at Method.invoke(Method.java:578)
        at Reflector.main(Reflector.java:11)
```

这是怎么发生的呢？正如这个异常所显示的，hasNext 方法当然是公共的，所以它在任何地方都是可以被访问的。那么为什么这个基于反射的方法调用是非法的呢？这里的问题并不在于该方法的访问级别（access level），而在于该方法所在的类型的访问级别。这个类型所扮演的角色和一个普通方法调用中的限定类型（qualifying type）是相同的[JLS 13.1]。在这个程序中，该方法是从某个类中选择出来的，而这个类型是由从 it.getClass 方法返回的 Class 对象表示的。这是迭代器的动态类型(dynamic type)，它恰好是私有的嵌套类(nested class) java.util.HashMap.KeyIterator。出现 IllegalAccessException 异常的原因就是这个类不是公共的，它来自另外一个包：访问位于其他包中的非公共类型的成员是不合法的[JLS 6.6.1]。无论是一般的访问还是通过反射的访问，上述的禁律都是有效的。下面这段没有使用反射的程序也违反了这条规则。

```
package library;
public class Api{
```

```
    static class PackagePrivate{}
    public static PackagePrivate member = new PackagePrivate();
}

package client;
import library.Api;
class Client{
    public static void main(String[] args){
        System.out.println(Api.member.hashCode());
    }
}
```

尝试编译这段程序会得到如下的错误：

```
Client.java:5: Object.hashCode() isn't defined in a public
class or interface; can't be accessed from outside package
      System.out.println(Api.member.hashCode());
                                            ^
```

这个错误与前面那个由含有反射的程序所产生的运行期错误具有相同的意义。Object 类型和 hashCode 方法都是公共的。问题在于 hashCode 方法是通过一个限定类型调用的，但用户访问不到这个类型。该方法调用的限定类型是 library.Api.PackagePrivate，这是一个位于其他包的非公共类型。

这并不意味着 Client 就不能调用 Api.member 的 hashCode 方法。要做到这一点，只需要使用一个可访问的限定类型即可，在这里可以将 Api.member 转型成 Object。经过这样的修改之后，Client 类就可以顺利地编译和运行了：

```
System.out.println(((Object)Api.member).hashCode());
```

实际上，这个问题并不会在普通的非反射的访问中出现，因为 API 的编写者在他们的公共 API 中只会使用公共的类型。即使这个问题有可能发生，它也会以编译期错误的形式显现出来，所以比较容易修改。而使用反射的访问就不同了，object.getClass().getMethod(“methodName”) 这种惯用法虽然很常见，但是却有问题的，它不应该被使用。就像我们在前面的程序中看到的那样，这种用法很容易在运行期产生一个 IllegalAccessException。

在使用反射访问某个类型时，请使用表示某种可访问类型的 Class 对象。回到我们前面的那个程序，hasNext 方法是声明在一个公共类型 java.util.Iterator 中的，所以它的类对象应该被用来进行反射访问。经过这样的修改后，这个 Reflector 程序就会打印出 true：

```
Method m = Iterator.class.getMethod("hasNext");
```

你完全可以避免这一类的问题，你应该只有在实例化时才使用反射，而方法调用都通过使用接口进行[EJ Item 35]。这种使用反射的用法，可以将那些调用方法的类与那些实现这些方法的类隔离开，并且提供了更高程度的类型安全。这种用法在“服务提供者框架”（Service Provider Frameworks）中很常见。这种模式并不能解决反射访问中的所有问题，但是如果它可以解决你所遇到的问题，请务必使用它。

总之，访问其他包中的非公共类型的成员是不合法的，即使这个成员同时也被声明为某个公共类型的公共成员也是如此。不论这个成员是否是通过反射被访问的，上述规则都是成立的。这个问题很有可能只在反射访问中才会出现。对于平台的设计者来说，这里的教训与谜题 67 中的一样，应该让错误症状尽可能清晰地显示出来。对于运行期的异常和编译期的提示都还有些东西需要改进。

谜题 79：这是狗的生活

下面的这个类模拟了一个家庭宠物的生活。main 方法创建了一个 Pet 实例，用它来表示一只名叫 Fido 的狗，然后让它运行。虽然绝大部分的狗都在后院里奔跑（run），这只狗却是在后台运行（run）。那么，这个程序会打印出什么呢？

```
public class Pet{
    public final String name;
    public final String food;
    public final String sound;
    public Pet(String name, String food, String sound){
        this.name = name;
        this.food = food;
        this.sound = sound;
    }

    public void eat(){
        System.out.println(name + ": Mmmmm, " + food );
    }
    public void play(){
        System.out.println(name + ": " + sound + " " + sound);
    }
    public void sleep(){
        System.out.println(name + ": Zzzzzzz...");
    }
    public void live(){
        new Thread(){
            public void run(){
                while(true){
                    eat();
                    play();
                    sleep();
```

```
                }
            }
        }.start();
    }

    public static void main(String[] args){
        new Pet("Fido", "beef", "Woof").live();
    }
}
```

main 方法创建了一个用来表示 Fido 的 Pet 实例，并且调用了它的 live 方法。然后，live 方法创建并且启动了一个线程，该线程反复的调用其外围（enclosing）的 Pet 实例的 eat、play 和 sleep 方法，就这么一直进行下去。这些方法都会打印单独的一行，所以你会想到这个程序会反复的打印以下的 3 行：

```
Fido: Mmmmm, beef
Fido: Woof Woof
Fido: Zzzzzzz…
```

但是如果你尝试运行这个程序，你会发现它甚至不能通过编译。而产生的编译错误信息没有什么用处：

```
Pet.java:28: cannot find symbol
symbol: method sleep()
                        sleep();
                        ^
```

为什么编译器找不到那个符号呢？这个符号确实是白纸黑字地写在那里。与谜题 74 一样，这个问题的源自重载解析过程的细节。编译器会在包含有正确名称的方法的最内层范围内查找需要调用的方法[JLS 15.12.1]。在我们的程序中，对于对 sleep 方法的调用，这个最内层的范围就是包含有该调用的匿名类（anonymous class），这个类继承了 Thread.sleep(long)方法和 Thread.sleep(long,int)方法，它们是该范围内唯一的名称为 sleep 的方法，但是由于它们都带有参数，所以都不适用于这里的调用。由于该方法调用的 2 个候选方法都不适用，所以编译器就打印出了错误信息。

从 Thread 那里继承到匿名类中的 2 个 sleep 方法遮蔽（shadow）[JLS 6.3.1]了我们想要调用的 sleep 方法。正如你在谜题 71 和谜题 73 中所看到的那样，你应该避免遮蔽。在这个谜题中的遮蔽是间接地无意识地发生的，这使得它更加“阴险”。

订正这个程序的一个比较显而易见的方法，就是把 Pet 中的 sleep 方法的名字改成 snooze，doze 或者 nap。订正该程序的另一个方法，是在方法调用的时候使

用受限的(qualified) this 结构来显式地为该类命名。此时的调用就变成了 Pet.this.sleep() 。

订正该程序的第三个方法，也是可以被证明是最好的方法，就是采纳谜题 77 的建议，使用 Thread(Runnable)构造器来替代对 Thread 的继承。如果你这么做了，原有的问题将会消失，因为那个匿名类不会再继承 Thread.sleep 方法。

程序经过少许的修改，就可以产生我们想要的输出了，当然这里的输出可能有点无聊：

```
public void live(){
  new Thread(new Runnable(){
      public void run(){
           while(true){
                eat();
                play();
                sleep();
           }
      }
  }).start();
}
```

总之，要小心无意间产生的遮蔽，并且要学会识别表明存在这种情况的编译器错误信息。对于编译器的编写者来说，你应该尽力去产生那些对程序员来说有意义的错误消息。例如在我们的程序中，编译器应该可以警告程序员，存在着适用于方法调用但却被遮蔽掉的方法。

谜题 80：更深层的反射

下面这个程序通过打印一个由反射创建的对象来产生输出。那么它会打印出什么呢？

```
public class Outer{
    public static void main(String[] args) throws Exception{
        new Outer().greetWorld();
    }

    private void greetWorld()throws Exception {
        System.out.println( Inner.class.newInstance() );
    }

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

```
    }
}
```

这个程序看起来是最普通的 Hello World 程序的又一个特殊的变体。Outer 中的 main 方法创建了一个 Outer 实例，并且调用了它的 greetWorld 方法，该方法以字符串形式打印了通过反射创建的一个新的 Inner 实例。Inner 的 toString 方法总是返回标准的问候语，所以程序的输出应该与往常一样，是 Hello World。如果你尝试运行这个程序，你会发现实际的输出比较长，而且更加令人迷惑：

```
Exception in thread "main" InstantiationException: Outer$Inner
        at java.lang.Class.newInstance0(Class.java:335)
        at java.lang.Class.newInstance(Class.java:303)
        at Outer.greetWorld(Outer.java:7)
        at Outer.main(Outer.java:3)
```

为什么会抛出这个异常呢？从 5.0 版本开始，关于 Class.newInstance 的文档叙述道：如果那个 Class 对象“代表了一个抽象类（abstract class），一个接口（interface），一个数组类（array class），一个原始类型（primitive type），或者是空（void）；或者这个类没有任何空的[也就是无参数的]构造器；或者实例化由于某些其他原因而失败，那么它就会抛出异常”[JAVA-API]。这里出现的问题满足上面的哪些条件呢？遗憾的是，异常信息没有提供任何提示。在这些条件中，只有后 2 个有可能会满足：要么是 Outer.Inner 没有空的构造器，要么是实例化由于“某些其它原因”而失败了。正如 Outer.Inner 这种情况，当一个类没有任何显式的构造器时，Java 会自动地提供一个不带参数的公共的缺省构造器[JLS 8.8.9]，所以它应该是有一个空构造器的。不过，newInstance 方法调用失败的原因还是因为 Outer.Inner 没有空构造器！

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，这个参数表示了它的直接外围实例（immediately enclosing instance）[JLS 13.1]。当你在代码中任何可以让编译器找到合适的外围实例的地方去调用构造器的时候，这个参数就会被隐式地传递进去。但是，上述的过程只适用于普通的构造器调用，也就是不使用反射的情况。当你使用反射调用构造器时，这个隐藏的参数就需要被显式地传递，这对于 Class.newInstance 方法是不可能做到的。要传递这个隐藏参数的唯一办法就是使用 java.lang.reflect.Constructor。当对程序进行了这样的修改后，它就可以正常的打印出 Hello World 了：

```
private void greetWorld() throws Exception{
  Constructor c = Inner.class.getConstructor(Outer.class);
  System.out.println(c.newInstance(Outer.this));
}
```

作为其他的选择，你可能观察到了，Inner 实例并不需要一个外围的 Outer 实例，所以可以将 Inner 类型声明为静态的（static）。除非你确实是需要一个外围实

例，否则你应该优先使用静态成员类（static member class）而不是非静态成员类[EJ Item 18]。下面这个简单的修改就可以订正这个程序：

```
public static class Inner{...}
```

Java 程序的反射模型和它的语言模型是不同的。反射操作处于虚拟机层次，暴露了很多从 Java 程序到 class 文件的翻译细节。这些细节当中的一部分由 Java 的语言规范来管理，但是其余的部分可能会随着不同的具体实现而有所不同。在 Java 语言的早期版本中，从 Java 程序到 class 文件的映射是很直接的，但是随着一些不能被虚拟机直接支持的高级语言特性的加入，如嵌套类（nested class）、协变返回类型（covariant return types）、泛型（generics）和枚举类型（enums），使得这种映射变得越来越复杂了。

考虑到从 Java 程序到 class 文件的映射的复杂度，请避免使用反射来实例化内部类。更一般地讲，当我们在用高级语言特性定义的程序元素之上使用反射的时候，一定要小心，从反射的视角观察程序可能不同与从代码的视角去观察它。请避免依赖那些没有被语言规范所管理的翻译细节。对于平台的实现者来说，这里的教训就是要再次重申，请提供清晰准确的诊断信息。

谜题 81：烧焦到无法识别

下面这个程序看起来是在用一种特殊的方法做一件普通的事。那么，它会打印出什么呢？

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

尽管这个程序有点奇怪，但是我们没有理由怀疑它会产生不正确的行为。它将“Hello World”写入了 System.out，每次写一个字符。你可能会意识到 write 方法只会使用其输入参数的低位字节（lower-order byte）。所以当“Hello World”含有任何外来字符的时候，可能会造成一些麻烦，但这里不会：因为“Hello World”完全是由 ASCII 字符组成的。无论你是每次打印一个字符，还是一次全部打印，结果都应该是一样的：这个程序应该打印 Hello World。然而，如果你运行该程序，就会发现它不会打印任何东西。那句问候语到哪里去了？难道是程序认为它并不令人愉快？

这里的问题在于 System.out 是带有缓冲的。Hello World 中的字符被写入了 System.out 的缓冲区，但是缓冲区从来都没有被刷新（flush）。大多数的程序员认为，当有输出产生的时候 System.out 和 System.err 会自动地进行刷新，这

并不完全正确。这 2 个流都属于 PrintStream 类型，在 5.0 版[Java-API]中，有关这个类型的文档叙述道：

一个 PrintStream 可以被创建为自动刷新的；这意味着当一个字节数组(byte array)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 方法就会被自动地调用。

System.out 和 System.err 所引用的流确实是 PrintStream 的能够自动刷新的变体，但是上面的文档中并没有提及 write(int)方法。有关 write(int)方法的文档叙述道：将指定的 byte 写入流。如果这个 byte 是一个换行字符，并且流可以自动刷新，那么 flush 方法将被调用[Java-API]。实际上，write(int)是唯一一个在自动刷新(automatic flushing)功能开启的情况下不刷新 PrintStream 的输出方法(output method)。

令人好奇的是，如果这个程序改用 print(char)去替代 write(int)，它就会刷新 System.out 并打印出 Hello World。这种行为与 print(char)的文档是矛盾的，因为其文档叙述道[Java-API]：

打印一个字符：这个字符将根据平台缺省的字符编码方式被翻译成为一个或多个字节，并且这些字节将完全按照 write(int)方法的方式被写出。

类似地，如果程序改用 print(String)，它也会对流进行刷新，虽然文档中是禁止这么做的。相应的文档确实应该被修改为描述该方法的实际行为，而修改方法的行为则会破坏稳定性。

修改这个程序最简单的方法就是在循环之后加上一个对 System.out.flush 方法的调用。经过这样的修改之后，程序就会正常地打印出 Hello World。当然，更好的办法是重写这个程序，使用我们更熟悉的 System.out.println 方法在控制台上产生输出。

这个谜题的教训与谜题 23 一样：尽可能使用熟悉的惯用法；如果你不得不使用陌生的 API，请一定要参考相关的文档。这里有 3 条教训给 API 的设计者们：请让你们的方法的行为能够清晰的反映在方法名上；请清楚而详细地给出这些行为的文档；请正确地实现这些行为。

谜题 82：啤酒爆炸

这一章的许多谜题都涉及到了多线程，而这个谜题涉及到了多进程。如果你用一行命令行带上参数 slave 去运行这个程序，它会打印什么呢？如果你使用的命令行不带任何参数，它又会打印什么呢？

```
public class BeerBlast{
    static final String COMMAND = "java BeerBlast slave";
    public static void main(String[] args) throws Exception{
        if(args.length == 1 && args[0].equals("slave")) {
```

```
            for(int i = 99; i > 0; i--){
                System.out.println( i +
                        " bottles of beer on the wall" );
                System.out.println(i + " bottles of beer");
                System.out.println(
                    "You take on down, pass it around,");
                System.out.println( (i-1) +
                    " bottles of beer on the wall");
                System.out.println();
            }
        }else{
            // Master
            Process process = Runtime.getRuntime().exec(COMMAND);
            int exitValue = process.waitFor();
            System.out.println("exit value = " + exitValue);
        }
    }
}
```

如果你使用参数 slave 来运行该程序，它就会打印出那首激动人心的名为”99 Bottles of Beer on the Wall”的童谣的歌词，这没有什么神秘的。如果你不使用该参数来运行这个程序，它会启动一个 slave 进程来打印这首歌谣，但是你看不到 slave 进程的输出。主进程会等待 slave 进程结束，然后打印出 slave 进程的退出值(exit value)。根据惯例，0 值表示正常结束，所以 0 就是你可能期望该程序打印的东西。如果你运行了程序，你可能会发现该程序只会悬挂在那里，不会打印任何东西，看起来 slave 进程好像永远都在运行着。所以你可能会觉得你应该一直都能听到”99 Bottles of Beer on the Wall”这首童谣，即使是这首歌被唱走调了也是如此，但是这首歌只有 99 句，而且，电脑是很快的，你假设的情况应该是不存在的，那么这个程序出了什么问题呢？

这个秘密的线索可以在 Process 类的文档中找到，它叙述道：“由于某些本地平台只提供有限大小的缓冲，所以如果未能迅速地读取子进程(subprocess)的输出流，就有可能会导致子进程的阻塞，甚至是死锁” [Java-API]。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首冗长的歌谣。为了确保 slave 进程能够结束，父进程必须排空(drain)它的输出流，而这个输出流从 master 线程的角度来看是输入流。下面的这个工具方法会在后台线程中完成这项工作：

```
    static void drainInBackground(final InputStream is) {
        new Thread(new Runnable(){
            public void run(){
                try{
                    while( is.read() >= 0 );
                } catch(IOException e){
                    // return on IOException
                }
```

```
            }
        }).start();
    }
```

如果我们修改原有的程序，在等待 slave 进程之前调用这个方法，程序就会打印出 0：

```
    }else{ // Master
        Process process = Runtime.getRuntime().exec(COMMAND);
        drainInBackground(process.getInputStream());
        int exitValue = process.waitFor();
        System.out.println("exit value = " + exitValue);
    }
```

这里的教训是：为了确保子进程能够结束，你必须排空它的输出流；对于错误流（error stream）也是一样，而且它可能会更麻烦，因为你无法预测进程什么时候会倾倒（dump）一些输出到这个流中。在 5.0 版本中，加入了一个名为 ProcessBuilder 的类用于排空这些流。它的 redirectErrorStream 方法将各个流合并起来，所以你只需要排空这一个流。如果你决定不合并输出流和错误流，你必须并行地（concurrently）排空它们。试图顺序化地（sequentially）排空它们会导致子进程被挂起。

多年以来，很多程序员都被这个缺陷所刺痛。这里对于 API 设计者们的教训是，Process 类应该避免这个错误，也许应该自动地排空输出流和错误流，除非用户表示要读取它们。更一般的讲，API 应该设计得更容易做出正确的事，而很难或不可能做出错误的事。

谜题 83：诵读困难者的一神论

从前有一个人，他认为世上只有一只不寻常的狗，所以他写出了如下的类，将它作为一个单件（singleton）[Gamma95]：

```
public class Dog extends Exception {
  public static final Dog  INSTANCE = new Dog();
  private Dog() {}
  public String toString(){
    return "Woof";
  }
}
```

结果证明这个人的做法是错误的。你能够在这个类的外部不使用反射来创建出第 2 个 Dog 实例吗？

这个类可能看起来像一个单件，但它并不是。问题在于，Dog 扩展了 Exception，而 Exception 实现了 java.io.Serializable。这就意味着 Dog 是可序列化的

（serializable），并且解序列（deserialization）会创建一个隐藏的构造器。正如下面的这段程序所演示的，如果你序列化了 Dog.INSTANCE，然后对得到的字节序列（byte sequence）进行解序列，最后你就会得到另外一个 Dog。该程序打印的是 false，表示新的 Dog 实例和原来的那个实例是不同的，并且它还打印了 Woof，说明新的 Dog 实例也具有相应的功能：

```
import java.io.*;
public class CopyDog{ // Not to be confused with copycat
    public static void main(String[] args){
         Dog newDog = (Dog) deepCopy(Dog.INSTANCE);
         System.out.println(newDog == Dog.INSTANCE);
         System.out.println(newDog);
    }

    // This method is very slow and generally a bad idea!
    static public Object deepCopy(Object obj){
        try{
            ByteArrayOutputStream bos =
                new ByteArrayOutputStream();
           new ObjectOutputStream(bos).writeObject(obj);
            ByteArrayInputStream bin =
                new ByteArrayInputStream(bos.toByteArray());
            return new ObjectInputStream(bin).readObject();
        } catch(Exception e) {
            throw new IllegalArgumentException(e);
        }
    }
}
```

要订正这个问题，可在 Dog 中添加一个 readResolve 方法，它可以将那个隐藏的构造器转变为一个隐藏的静态工厂（static factory），以返回原来那个的 Dog [EJ Items 2,57]。在 Dog 中添加了这个方法之后，CopyDog 将打印 true 而不是 false，表示那个“复本”实际上就是原来的那个实例：

```
    private Object readResolve(){
        // Accept no substitues!
        return INSTANCE;
    }
```

这个谜题的主要教训就是一个实现了 Serializable 的单件类，必须有一个 readResolve 方法，用以返回它的唯一的实例。一个次要的教训就是，有可能由于对一个实现了 Serializable 的类进行了扩展，或者由于实现了一个扩展自 Serializable 的接口，使得我们在无意中实现了 Serializable。给平台设计者的教训是，隐藏的构造器，例如序列化中产生的那个，会让读者对程序行为的产生错觉。

谜题 84：被粗暴地中断

在下面这个程序中，一个线程试图中断自己，然后检查中断是否成功。它会打印什么呢？

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

虽然一个线程中断自己不是很常见，但这也不是没有听说过的。当一个方法捕捉到了一个 InterruptedException 异常，而且没有做好处理这个异常的准备时，那么这个方法通常会将该异常重新抛出（rethrow）。但是由于这是一个“被检查的异常”，所以只有在方法声明允许的情况下该方法才能够将异常重新抛出。如果不能重新抛出，该方法可以通过中断当前线程对异常“再构建”(reraise)。这种方式工作得很好，所以这个程序中的线程中断自己应该是没有任何问题的。所以，该程序应该进入 if 语句的第一个分支，打印出 Interrupted: true。如果你运行该程序，你会发现并不是这样。但是它也没有打印 Not interrupted: false，它打印的是 Interrupted: false。

看起来该程序好像不能确定线程是否被中断了。当然，这种看法是毫无意义的。实际上发生的事情是，Thread.interrupted 方法第一次被调用的时候返回了 true，并且清除了线程的中断状态，所以在 if-then-else 语句的分支中第 2 次调用该方法的时候，返回的就是 false。调用 Thread.interrupted 方法总是会清除当前线程的中断状态。方法的名称没有为这种行为提供任何线索，而对于 5.0 版本，在相应的文档中有一句话概要地也同样具有误导性地叙述道：“测试当前的线程是否中断”[Java-API]。所以，可以理解为什么很多程序员都没有意识到 Thread.interrupted 方法会对线程的中断状态造成影响。

Thread 类有 2 个方法可以查询一个线程的中断状态。另外一个方法是一个名为 isInterrupted 的实例方法，而它不会清除线程的中断状态。如果使用这个方法重写程序，它就会打印出我们想要的结果 true：

```
public class SelfInterruption {
    public static void main(String[ ] args) {
        Thread.currentThread().interrupt();
        if(Thread.currentThread().isInterrupted()) {
```

```
            System.out.println("Interrupted: " +
                    Thread.currentThread().isInterrupted());
        }else{
            System.out.println("Not interrupted: " +
                    Thread.currentThread().isInterrupted());
        }
    }
}
```

这个谜题的教训是：不要使用 Thread.interrupted 方法，除非你想要清除当前线程的中断状态。如果你只是想查询中断状态，请使用 isInterrupted 方法。这里给 API 设计者们的教训是方法的名称应该用来描述它们主要功能。根据 Thread.interrupted 方法的行为，它的名称应该是 clearInterruptStatus，因为相对于它对中断状态的改变，它的返回值是次要的。特别是当一个方法的名称并不完美的时候，文档是否能清楚地描述它的行为就显得非常重要了。

谜题 85：惰性初始化

下面这个可怜的小类实在是太懒了，甚至于都不愿意用通常的方法进行初始化，所以它求助于后台线程。这个程序会打印什么呢？每次你运行它的时候都会打印出相同的东西吗？

```
public class Lazy {
    private static boolean initialized = false;
    static {
        Thread t = new Thread(new Runnable() {
                public void run() {
                    initialized = true;
                }
            });
        t.start();
        try{
            t.join();
        }catch (InterruptedException e){
            throw new AssertionError(e);
        }
    }

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

虽然有点奇怪，但是这个程序看起来很直观的。静态域 initialized 初始时被设为 false。然后主线程创建了一个后台线程，该线程的 run 方法将 initialized

的值设为 true。主线程启动了后台线程之后，就调用了 join 方法等待它的结束。当后台线程完成运行的时候，毫无疑问 initialized 的值已经被设为了 true。当且仅当这个时候，调用了 main 方法的主线程会打印出 initialized 的值。如果是这样的话，程序肯定会打印出 true 吗？如果你运行该程序，你会发现它不会打印任何东西，它只是被挂起了。

为了理解这个程序的行为，我们需要模拟它初始化的细节。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。在忽略严重错误的情况下，有 4 种可能的情况[JLS 12.4.2]：

- 这个类尚未被初始化。
- 这个类正在被当前线程初始化：这是对初始化的递归请求。
- 这个类正在被其他线程而不是当前线程初始化。
- 这个类已经被初始化。

当主线程调用 Lazy.main 方法时，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初始化，并开始对这个类进行初始化。按照我们前面的分析，主线程会将 initialized 的值设为 false，创建并启动一个后台线程，该线程的 run 方法会将 initialized 设为 true，然后主线程会等待后台线程执行完毕。此时，有趣的事情开始了。

那个后台线程调用了它的 run 方法。在该线程将 Lazy.initialized 设为 true 之前，它也会去检查 Lazy 类是否已经被初始化。这个时候，这个类正在被另外一个线程进行初始化（情况 3）。在这种情况下，当前线程，也就是那个后台线程，会等待 Class 对象直到初始化完成。遗憾的是，那个正在进行初始化工作的线程，也就是主线程，正在等待着后台线程运行结束。因为这 2 个线程现在正相互等待着，该程序就死锁了(deadlock)。这就是所有的一切，真是遗憾。有 2 种方法可以订正这个程序。到目前为止，最好的方法就是不要在类进行初始化的时候启动任何后台线程：有些时候，2 个线程并不比 1 个线程好。更一般的讲，要让类的初始化尽可能地简单。订正这个程序的第 2 种方法就是让主线程在等待后台线程之前就完成类的初始化：

```
// Bad way to eliminate the deadlock. Complex and error prone
public class Lazy {
    private static boolean initialized = false;
    private static Thread t = new Thread(new Runnable() {
                    public void run() {
                            initialized = true;
                    }
            });
    static {
        t.start();
    }

    public static void main(String[] args){
```

```
            try{
                t.join();
            }catch (InterruptedException e){
                throw new AssertionError(e);
            }
            System.out.println(initialized);
        }
}
```

虽然这么做确实消除了死锁，但是它却是一个非常不好的想法。主线程需要等待后台线程完成工作，但是其他的线程不需要这么做。一旦主线程完成了对 Lazy 类的初始化，其他线程就可以使用这个类了。这使得在 initialized 的值还是 false 的时候，其他线程就可以观察到它。

总之，在类的初始化期间等待某个后台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。类的自动初始化被公认为是语言设计上的难题，Java 的设计者们在这个方面做得很不错。如果你写了一些复杂的类初始化代码，很多种情况下，你这是在搬起石头砸自己的脚。

Java 谜题 9——高级谜题

谜题 86：有毒的括号垃圾 | 谜题 87：紧张的关系 | 谜题 88：原生类型的处理 | 谜题 89：泛型迷药 | 谜题 90：荒谬痛苦的超类 | 谜题 91：序列杀手 | 谜题 92：双绞线 | 谜题 93：类的战争 | 谜题 94：迷失在混乱中 | 谜题 95：只是些甜点

谜题 86：有毒的括号垃圾

你能否举出这样一个合法的 Java 表达式，只要对它的某个子表达式加上括号就可以使其成为不合法的表达式，而添加的括号只是为了注解未加括号时赋值的顺序？

插入一对用来注解现有赋值顺序的括号对程序的合法性似乎是应该没有任何影响的。事实上，绝大多数情况下确实是没有影响的。但是，在两种情况下，插入一对看上去没有影响的括号可能会令合法的 Java 程序变得不合法。这种奇怪的情况是由于数值的二进制补码的不对称性引起的，就像在谜题 33 和谜题 64 中所讨论的那样。 你可能会联想到，最小的 int 型负数其绝对值比最大的 int 型正数大 1：Integer.MIN_VALUE 是-2^{31}，即-2,147,483,648，而 Integer.MAX_VALUE 是$2^{31}-1$，即 2,147,483,647。

Java 不支持负的十进制字面常量；int 和 long 类型的负数常量都是由正数十进制字面常量前加一元负操作符（-）构成。这种构成方式是由一条特殊的语言规则所决定的：在 int 类型的十进制字面常量中，最大的是 2147483648。而从 0 到 2147483647 的所有十进制字面常量都可以在任何能够使用 int 类型字面常量的地方出现，但是字面常量 2147483648 只能作为一元负操作符的操作数来使用[JLS 3.10.1]。

一旦你知道了这个规则，这个谜题就很容易了。符号-2147483648 构成了一个合法的 Java 表达式，它由一元负操作符加上一个 int 型字面常量 2147483648 组成。通过添加一对括号来注解（很不重要的）赋值顺序，即写成-（2147483648），就会破坏这条规则。信不信由你，下面这个程序肯定会出现一个编译期错误，如果去掉了括号，那么错误也就没有了：

```
public class PoisonParen {
    int i = -(2147483648);
}
```

类似地，上述情况也适用于 long 型字面常量。下面这个程序也会产生一个编译期错误，并且如果你去掉括号错误也会消失：

```
public class PoisonParen {
long j = -(9223372036854774808L);
}
```

谜题 87：紧张的关系

在数学中，等号（＝）定义了一种真实的数之间的等价关系（equivalence relation）。这种等价关系将一个集合分成许多等价类（equivalence class），每个等价类由所有相互相等的值组成。其他的等价关系包括有所有三角形集合上的“全等”关系和所有书的集合上的“有相同页数”的关系等。事实上，关系 ～ 是一种等价关系，当且仅当它是自反的、传递的和对称的。这些性质定义如下：

- 自反性：对于所有 x，x ～ x。也就是说，每个值与其自身存在关系 ～ 。
- 传递性：如果 x ～ y 并且 y ～ z，那么 x ～ z。也就是说，如果第一个值与第二个值存在关系 ～，并且第二个值与第三个值存在关系 ～ ，那么第一个值与第三个值也存在关系 ～ 。
- 对称性：如果 x ～ y，那么 y ～ x。也就是说，如果第一个值和第二个值存在关系 ～ ，那么第二个值与第一个值也存在关系 ～ 。

如果你看了谜题 29，便可以知道操作符 == 不是自反的，因为表达式（Double.NaN == Double.NaN ）值为 false，表达式（ Float.NaN == Float.NaN ）也是如此。但是操作符 == 是否还违反了对称性和传递性呢？事实上它并不违反对称性：对于所有 x 和 y 的值，（ x == y ）意味着（ y == x ）。 传递性则完全是另一回事。 谜题 35 为操作符 == 作用于原始类型的数值时不符合传递性

的原因提供了线索。当比较两个原始类型数值时，操作符 == 首先进行二进制数据类型提升（binary numeric promotion）[JLS 5.6.2]。这会导致这两个数值中有一个会进行拓宽原始类型转换（widening primitive conversion）。大部分拓宽原始类型转换是不会有问题的，但有三个值得注意的异常情况：将 int 或 long 值转换成 float 值，或 long 值转换成 double 值时，均会导致精度丢失。这种精度丢失可以证明 == 操作符的不可传递性。

实现这种不可传递性的窍门就是利用上述三种数值比较中的两种去丢失精度，然后就可以得到与事实相反的结果。可以这样构造例子：选择两个较大的但不相同的 long 型数值赋给 x 和 z，将一个与前面两个 long 型数值相近的 double 型数值赋给 y。下面的程序就是其代码，它打印的结果是 true true false，这显然证明了操作符 == 作用于原始类型时具有不可传递性。

```
public class Transitive {
    public static void main(String[] args) throws Exception {
        long x = Long.MAX_VALUE;
        double y = (double) Long.MAX_VALUE;
        long z = Long.MAX_VALUE - 1;
        System.out.print((x == y) + " "); // Imprecise!
        System.out.print((y == z) + " "); // Imprecise!
        System.out.println(x == z);       // Precise!
    }
}
```

本谜题的教训是：要警惕到 float 和 double 类型的拓宽原始类型转换所造成的损失。它们是悄无声息的，但却是致命的。它们会违反你的直觉，并且可以造成非常微妙的错误（见谜题 34）。更一般地说，要警惕那些混合类型的运算（谜题 5、8、24 和 31）。本谜题给语言设计者的教训和谜题 34 一样：悄无声息的精度损失把程序员们搞糊涂了。

谜题 88：原生类型的处理

下面的程序由一个单一的类构成，该类表示一对类型相似的对象。它大量使用了 5.0 版的特性，包括泛型、自动包装、变长参数（varargs）和 for-each 循环。关于这些特性的介绍，请查看 http://java.sun.com/j2se/5.0/docs/guide/language[Java-5.0]。这个程序的 main 方法只是执行这个类。那么它会打印什么呢？

```
import java.util.*;
public class Pair<T> {
    private final T first;
    private final T second;

    public Pair(T first, T second) {
        this.first = first;
```

```
        this.second = second;
    }

    public T first() {
        return first;
    }
    public T second() {
        return second;
    }
    public List<String> stringList() {
        return Arrays.asList(String.valueOf(first),
                             String.valueOf(second));
    }

    public static void main(String[] args) {
        Pair p = new Pair<Object> (23, "skidoo");
        System.out.println(p.first() + " " + p.second());
        for (String s : p.stringList())
            System.out.print(s + " ");
    }
}
```

这段程序看上去似乎相当简单。它创建了一个对象对，其中第一个元素是一个表示 23 的 Integer 对象，第二个元素是一个字符串“skidoo”，然后这段程序将打印这个对象对的第一个和第二个元素，并用一个空格隔开。最后它循环迭代这些元素的 string 表示，并且再次打印它们，所以这段程序应该打印 23 skidoo 两次。然而可惜的是，它根本不能通过编译。更糟的是，编译器的错误消息更是另人困惑：

```
Pair.java:26: incompatible types;
found: Object, required: String
            for (String s : p.stringList())
                             ^
```

如果 Pair.stringList 是声明返回 List<Object>的话，那么这个错误消息还是可以明白的，但是事实是它返回的是 List<String>。究竟是怎么回事呢？

这个十分奇怪的现象是因为程序使用了原生类型（raw type）而引起的。一个原生类型就是一个没有任何类型参数的泛型类或泛型接口的名字。例如，List<E>是一个泛型接口，List<String> 是一个参数化的类型，而 List 就是一个原生类型。在我们的程序中，唯一用到原生类型的地方就是在 main 方法中对局部变量 p 的声明：

```
Pair p = new Pair<Object> (23, "skidoo");
```

一个原生类型很像其对应的参数化类型，但是它的所有实例成员都要被替换掉，而替换物就是这些实例成员被擦除掉对应部分之后剩下的东西。具体地说，在一个实例方法声明中出现的每个参数化的类型都要被其对应的原生部分所取代[JLS 4.8]。我们程序中的变量 p 是属于原生类型 Pair 的，所以它的所有实例方法都要执行这种擦除。这也包括声明返回 List<String>的方法 stringList。编译器会将这个方法解释为返回原生类型 List。

当 List<String>实现了参数化类型 Iterable<String>时，List 也实现了原生类型 Iterable。Iterable<String>有一个 iterator 方法返回参数化类型 Iterator<String>，相应地，Iterable 也有一个 iterator 方法返回原生类型 Iterator。当 Iterator<String>的 next 方法返回 String 时，Iterator 的 next 方法返回 Object。因此，循环迭代 p.stringList()需要一个 Object 类型的循环变量，这就解释了编译器的那个奇怪的错误消息的由来。这种现象令人想不通的原因在于参数化类型 List<String>虽然是方法 stringList 的返回类型，但它与 Pair 的类型参数没有关系，事实上最后它被擦除了。

你可以尝试通过将循环变量类型从 String 改成 Object 这一做法来解决这个问题：

```
// Don't do this; it doesn't really fix the problem!
    for (Object s : p.stringList())
        System.out.print(s + " ");
```

这样确实令程序输出了满意的结果，但是它并没有真正解决这个问题。你会失去泛型带来的所有优点，并且如果该循环在 s 上调用了任何 String 方法，那么程序甚至不能通过编译。正确解决这个问题的方法是为局部变量 p 提供一个合适的参数化的声明：

```
    Pair<Object> p = new Pair<Object>(23, "skidoo");
```

以下是要点强调：原生类型 List 和参数化类型 List<Object>是不一样的。如果使用了原生类型，编译器不会知道在 list 允许接受的元素类型上是否有任何限制，它会允许你添加任何类型的元素到 list 中。这不是类型安全的：如果你添加了一个错误类型的对象，那么在程序接下来的执行中的某个时刻，你会得到一个 ClassCastException 异常。如果使用了参数化类型 List<Object>，编译器便会明白这个 list 可以包含任何类型的元素，所以你添加任何对象都是安全的。

还有第三种与以上两种类型密切相关的类型：List<?>是一种特殊的参数化类型，被称为通配符类型（wildcard type）。像原生类型 List 一样，编译器也不会知道它接受哪种类型的元素，但是因为 List<?>是一个参数化类型，从语言上来说需要更强的类型检查。为了避免出现 ClassCastException 异常，编译器不允许你添加除 null 以外的任何元素到一个类型为 List<?>的 list 中。

原生类型是为兼容 5.0 版以前的已有代码而设计的，因为它们不能使用泛型。5.0 版中的许多核心库类，如 collections，已经利用泛型做了改变，但是使用这些

类的已有程序的行为仍然与在以前的版本上运行一样。这些原生类型及其成员的行为被设计成可以镜像映射到 5.0 之前的 Java 语言上，从而保持了兼容性。

这个 Pair 程序的真正问题在于编程者没有决定究竟使用哪种 Java 版本。尽管程序中大部分使用了泛型，而变量 p 却被声明成原生类型。为了避免被编译错误所迷惑，请避免在打算用 5.0 或更新的版本来运行的代码中编写原生类型。如果一个已有的库方法返回了一个原生类型，那么请将它的结果存储在一个恰当的参数化类型的变量中。然而，最好的办法还是尽量将该库升级到使用泛型的版本上。虽然 Java 提供了原生类型和参数化类型间的良好的互用性，但是原生类型的局限性会妨碍泛型的使用。

实际上，这种问题在用 getAnnotation 方法在运行期读取 Class 的注解（annotations）的情况下也会发生，该方法是在 5.0 版中新添加到 Class 类中的。每次调用 getAnnotation 方法时都会涉及到两个 Class 对象：一个是在其上调用该方法的对象，另一个是作为传递参数指出需要哪个类的注解的对象。在一个典型的调用中，前者是通过反射获得的，而后者是一个类名称字面常量，如下例所示：

```
Author a = Class.forName(name).getAnnotation(Author.class);
```

你不必把 getAnnotation 的返回值转型为 Author。以下两种机制保证了这种做法可以正常工作：（1）getAnnotation 方法是泛型的。它是通过它的参数类型来确定返回类型的。具体地说，它接受一个 Class<T>类型的参数，返回一个 T 类型的值。（2）类名称字面常量提供了泛型信息。例如，Author.class 的类型是 Class<Author>。类名称字面常量可以传递运行时和编译时的类型信息。以这种方式使用的类名称字面常量被称作类型符号（type token）[Bracha04]。

与类名称字面常量不同的是，通过反射获得的 Class 对象不能提供完整的泛型类型信息:Class.forName 的返回类型是通配类型 Class<?>。在调用 getAnnotation 方法的表达式中，使用的是通配类型而不是原生类型 Class，这一点很重要。如果你采用了原生类型，返回的注解具有的就是编译期的 Annotation 类型而不是通过类名称字面常量指示的类型了。下面的程序片断错误地使用了原生类型，和本谜题中最初的程序一样不能通过编译，其原因也一样：

```
Class c = Class.forName(name);              // Raw type!
Author a = c.getAnnotation(Author.class);    // Type mismatch
```

总之，原生类型的成员被擦掉，是为了模拟泛型被添加到语言中之前的那些类型的行为。如果你将原生类型和参数化类型混合使用，那么便无法获得使用泛型的所有好处，而且有可能产生让你困惑的编译错误。另外，原生类型和以 Object 为类型参数的参数化类型也不相同。最后，如果你想重构现有的代码以利用泛型的优点，那么最好的方法是一次只重构一个 API，并且保证新的代码中绝不使用原生类型。

谜题 89：泛型迷药

和前一个谜题一样，本谜题也大量使用了泛型。我们从前面的错误中吸取教训，这次不再使用原生类型了。这个程序实现了一个简单的链表数据结构。main 程序构建了一个包含 2 个元素的 list，然后输出它的内容。那么，这个程序会打印出什么呢？

```
public class LinkedList<E> {
    private Node<E> head = null;

    private class Node<E> {
        E value;
        Node<E> next;

        // Node constructor links the node as a new head
        Node(E value) {
            this.value = value;
            this.next = head;
            head = this;
        }
    }

    public void add(E e) {
        new Node<E>(e);
        // Link node as new head
    }

    public void dump() {
        for (Node<E> n = head; n != null; n = n.next)
            System.out.println(n.value + " ");
    }

    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<String>();
        list.add("world");
        list.add("Hello");
        list.dump();
    }
}
```

又是一个看上去相当简单的程序。新元素被添加到链表的表头，而 dump 方法也是从表头开始打印 list。因此，元素的打印顺序正好和它们被添加到链表中的顺序相反。在本例中，程序先添加了“world”然后添加了“Hello”，所以总体

来看它似乎就是一个复杂化的 Hello World 程序。遗憾的是，如果你尝试着编译它，就会发现它不能通过编译。编译器的错误消息是令人完全无法理解的：

```
LinkedList.java:11: incompatible types
found : LinkedList<E>.Node<E>
required: LinkedList<E>.Node<E>
              this.next = head;
                          ^
LinkedList.java:12: incompatible types
found : LinkedList<E>.Node<E>
required: LinkedList<E>.Node<E>
              head = this;
                     ^
```

编译器试图告诉我们，这个程序太过复杂了。一个泛型类的内部类可以访问到它的外围类的类型参数。而编程者的意图很明显，即一个 Node 的类型参数应该和它外围的 LinkedList 类的类型参数一样，所以 Node 完全不需要有自己的类型参数。要订正这个程序，只需要去掉内部类的类型参数即可：

```
// 修复后的代码，可以继续修改得更好
public class LinkedList<E> {
    private Node head = null;

    private class Node {
        E value;
        Node next;

        //Node 的构造器，将 node 链接到链表上作为新的表头
        Node(E value) {
            this.value = value;
            this.next = head;
            head = this;
        }
    }

    public void add(E e) {
        new Node(e);
        //将 node 链接到链表上作为新的表头
    }

    public void dump() {
        for (Node n = head; n != null; n = n.next)
            System.out.print(n.value + " ");
    }
}
```

以上是解决问题的最简单的修改方案，但不是最优的。最初的程序所使用的内部类并不是必需的。正如谜题 80 中提到的，你应该优先使用静态成员类而不是非静态成员类[EJ Item 18]。LinkedList.Node 的一个实例不仅含有 value 和 next 域，还有一个隐藏的域，它包含了对外围的 LinkedList 实例的引用。虽然外部类的实例在构造阶段会被用来读取和修改 head，但是一旦构造完成，它就变成了一个甩不掉的包袱。更糟的是，这样使得构造器中被置入了修改 head 的负作用，从而使程序变得难以读懂。应该只在一个类自己的方法中修改该类的实例域。

因此，一个更好的修改方案是将最初的那个程序中对 head 的操作移到 LinkedList.add 方法中，这将会使 Node 成为一个静态嵌套类而不是真正的内部类。静态嵌套类不能访问它的外围类的类型参数，所以现在 Node 就必须有自己的类型参数了。修改后的程序既简单清楚又正确无误：

```
class LinkedList<E> {
    private Node<E> head = null;
    private static class Node<T> {
        T value; Node<T> next;
        Node(T value, Node<T> next) {
            this.value = value;
            this.next = next;
        }
    }
    public void add(E e) {
        head = new Node<E>(e, head);
    }
    public void dump() {
        for (Node<E> n = head; n != null; n = n.next)
            System.out.print(n.value + " ");
    }
}
```

总之，泛型类的内部类可以访问到其外围类的类型参数，这可能会使得程序模糊难懂。本谜题所阐述的误解对于初学泛型的程序员来说是普遍存在的。在一个泛型类中设置一个内部类并不是必错的，但是很少用到这种情况，而且你应该考虑重构你的代码来避免这种情况。当你在一个泛型类中嵌套另一个泛型类时，最好为它们的类型参数设置不同的名字，即使那个嵌套类是静态的也应如此。对于语言设计者来说，或许应该考虑禁止类型参数的遮蔽机制，同样的，局部变量的遮蔽机制也应该被禁止。这样的规则就可以捕获到本谜题中的错误了。

谜题 90：荒谬痛苦的超类

下面的程序实际上不会做任何事情。更糟的是，它连编译也通不过。为什么呢？又怎么来订正它呢？

```
public class Outer {
```

```
    class Inner1 extends Outer{}
    class Inner2 extends Inner1{}
}
```

这个程序看上去简单得不可能有错误，但是如果你尝试编译它，就会得到下面这个有用的错误消息：

```
Outer.java:3: cannot reference this before supertype constructor has been
called
    class Inner2 extends Inner1{}
    ^
```

好吧，可能这个消息不那么有用，但是我们还是从此入手。问题在于编译器产生的缺省的 Inner2 的构造器为它的 super 调用找不到合适的外部类实例。让我们来看看显式地包含了构造器的该程序：

```
public class Outer {
    public Outer() {}
    class Inner1 extends Outer{
        public Inner1() {
            super();    // 调用 Object()构造器
        }
    }
    class Inner2 extends Inner1{
        public Inner2() {
            super();    // 调用 Inner1()构造器
        }
    }
}
```

现在错误消息就会显示出多一点的信息了：

```
Outer.java:12: cannot reference this before
                supertype constructor has been called
        super();    // 调用 Inner1()构造器
        ^
```

因为 Inner2 的超类本身也是一个内部类，一个晦涩的语言规则登场了。正如大家知道的，要想实例化一个内部类，如类 Inner1，需要提供一个外部类的实例给构造器。一般情况下，它是隐式地传递给构造器的，但是它也可以以 expression.super(args)的方式通过超类构造器调用(superclass constructor invovation)显式地传递[JLS 8.8.7]。如果外部类实例是隐式传递的，编译器会自动产生表达式：它使用 this 来指代最内部的其超类是一个成员变量的外部类。这确实有点绕口，但是这就是编译器所作的事情。在本例中，那个超类就是 Inner1。因为当前类 Inner2 间接扩展了 Outer 类，Inner1 便是它的一个继承而

来的成员。因此，超类构造器的限定表达式直接就是 this。编译器提供外部类实例，将 super 重写成 this.super。 解释到这里，编译错误所含的意思可扩展为：

```
Outer.java:12: cannot reference this before
                        supertype constructor has been called
        this.super();
        ^
```

现在问题就清楚了：缺省的 Inner2 的构造器试图在超类构造器被调用前访问 this，这是一个非法的操作[JLS 8.8.7.1]。解决这个问题的蛮力方法是显式地传递合理的外部类实例：

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

这样可以通过编译，但是它太复杂了。这里有一个更好的解决方案：无论何时你写了一个成员类，都要问问你自己，是否这个成员类真的需要使用它的外部类实例？如果答案是否定的，那么应该把它设为静态成员类。内部类有时是非常有用的，但是它们很容易增加程序的复杂性，从而使程序难以被理解。它们和泛型（谜题 89）、反射（谜题 80）以及继承（本谜题）都有着复杂的交互方式。在本例中，如果你将 Inner1 设为静态的便可以解决问题了。如果你将 Inner2 也设为静态的，你就会真正明白这个程序做了什么：确实是一个相当好的意外收获。

总之，这种一个类既是外部类又是其他类的超类的方式是很不合理的。更一般地讲，扩展一个内部类的方式是很不恰当的；如果必须这样做的话，你也要好好考虑其外部类实例的问题。另外，尽量用静态嵌套类而少用非静态的[EJ Item 18]。大部分成员类可以并且应该被声明为静态的。

谜题 91：序列杀手

这个程序创建了一个对象并且检查它是否遵从某个类的不变规则(invariant)。然后该程序序列化这个对象，之后将其反序列化，然后再次检查反序列化得到的副本是否也遵从这个规则。它会遵从这个规则吗？如果不是的话，又是为什么呢？

```
import java.util.*;
import java.io.*;
```

```
public class SerialKiller {
    public static void main(String[] args) {
        Sub sub = new Sub(666);
        sub.checkInvariant();

        Sub copy = (Sub) deepCopy(sub);
        copy.checkInvariant();
    }

    // Copies its argument via serialization (See Puzzle 80)
    static public Object deepCopy(Object obj) {
        try {
            ByteArrayOutputStream bos =
                new ByteArrayOutputStream();
            new ObjectOutputStream(bos).writeObject(obj);
            ByteArrayInputStream bin =
                new ByteArrayInputStream(bos.toByteArray());
            return new ObjectInputStream(bin).readObject();
        } catch(Exception e) {
            throw new IllegalArgumentException(e);
        }
    }
}

class Super implements Serializable {
    final Set<Super> set = new HashSet<Super>();
}

final class Sub extends Super {
    private int id;
    public Sub(int id) {
        this.id = id;
        set.add(this); // Establish invariant
    }

    public void checkInvariant() {
        if (!set.contains(this))
            throw new AssertionError("invariant violated");
    }

    public int hashCode() {
        return id;
    }
```

```
    public boolean equals(Object o) {
        return (o instanceof Sub) && (id == ((Sub)o).id);
    }
}
```

程序中除了使用了序列化之外，看起来是很简单的。子类 Sub 覆写了 hashCode 方法和 equals 方法。这些覆写过的方法符合了相关的一般规约[EJ Item 7,8]。Sub 的构造器建立了这个类的不变规则，而在它这么做的时候没有调用到可覆写的方法（谜题 51）。Super 类有一个单独的 Set<Super>类型的域，Sub 类添加了另外一个 int 类型的域。Super 和 Sub 都不需要定制的序列化形式。那么什么东西会出错呢？

其实有很多。对于 5.0 版本，运行该程序会得到如下的“堆轨迹”（stack trace）：

```
Exception in thread “main” AssertionError
    at Sub.checkInvariant(SerialKiller.java:41)
    at SerialKiller.main(SerialKiller.java:10)
```

序列化和反序列化一个 Sub 实例会产生一个被破坏的副本。为什么呢？阅读程序并不会帮助你找出原因，因为真正引起问题的代码在其他地方。错误是由 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(populate)正在被反序列化的散列集合，HashSet.readObject 调用了 HashMap.put 方法，而它会去调用每个键(key)的 hashCode 方法。由于整个对象图（object graph）正在被反序列化，并没有什么可以保证每个键在它的 hashCode 方法被调用的时候已经被完全初始化了。实际上，这很少会成为一个问题，但是有时候它会造成绝对的混乱。这个缺陷会在正在被反序列化的对象图的某些循环中出现。

为了更具体一些，让我们看看程序中在反序列化 Sub 实例的时候发生了什么。首先，序列化系统会反序列化 Sub 实例中 Super 的域。唯一的这样的域就是 set，它包含了一个对 HashSet 的引用。在内部，每个 HashSet 实例包含一个对 HashMap 的引用，HashMap 的键是该散列集合的元素。HashSet 类有一个 readObject 方法，它创建一个空的 HashMap，并且使用 HashMap 的 put 方法，针对集合中的每个元素在 HashMap 中插入一个键-值对。put 方法会调用键的 hashCode 方法以确定它所在的单元格（bucket）。在我们的程序中，散列映射表中唯一的键就是 Sub 的实例，而它的 set 域正在被反序列化。这个实例的子类域（subclass field），即 id，尚未被初始化，所以它的值为 0，即所有 int 域的缺省初始值。不幸的是，Sub 的 hashCode 方法将返回这个值，而不是最后保存在这个域中的值 666。因为 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 域被初始化为 666 时，一切都太迟了。当 Sub 实例在 HashMap 中的时候，改变这个域的值就会破坏这个域，进而破坏 HashSet，破坏 Sub 实例。程序检测到了这个情况，就报告出了相应的错误。

这个程序说明，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（pseudoconstructor）中调用其可覆写方法的规

则[EJ Item 15]。Super 类的（缺省的）readObject 方法调用了 HashSet 的(显式的)readObject 方法，该方法进而调用了它内部的 HashMap 的 put 方法，put 方法又调用了 Sub 实例的 hashCode 方法，而该实例正处在创建的过程中。现在我们遇到大麻烦了：Super 类中，从 Object 类继承而来的 hashCode 方法在 Sub 中被覆写了，但是这个被覆写的方法在 Sub 的域被初始化之前就被调用了，而该方法需要依赖于 Sub 的域。

这个问题和谜题 51 中的那个本质上几乎是完全相同的。唯一真正不同的是在这个谜题中，readObject 伪构造器错误地替代了构造器。HashMap 和 Hashtable 的 readObject 方法受到的影响是类似的。

对于平台的实现者来说，也许可以通过牺牲一点性能来订正 HashSet、HashMap 和 HashTable 中的这个问题。当针对 HashSet 时，订正的策略可以是重写 readObject 方法使其在反序列化期间，将集合的元素保存到一个数组中，而不是将它们放入散列集合中。这样，当被反序列化的散列集合的公共方法首次被调用的时候，数组中的元素将在方法执行之前被插入到集合中。

这种方法的代价是它需要在与散列集合的每个公共方法相对应的条目上检查是否要组装散列集合。由于 HashSet、HashMap 以及 HashTable 都是性能临界（performance-critical）的，所以这个方法看起来是不可取的。更不幸的是，所有的用户都要付出这种代价，甚至当他们不对这些集合（collection）进行序列化时也是如此。这就违背了这样一个原则：你绝不应该为你不使用的功能而付出代价。

另外一个可能的方法是让 HashSet.readObject 方法调用 ObjectInputStream.registerValidation 方法，用以将散列集合的组装延迟到 validateObject 方法回调时再进行。这个方法看起来更吸引人，因为它仅仅增加了反序列化的开销，但是它会破坏任何在“包含流”（containing stream）的反序列化过程中试图使用 HashSet 实例的代码。

上述的 2 个方法是否可行还有待研究。但是现在，我们必须接受这些类的这种行为。幸运的是，有一个工作区（workaround）：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到它们自身。这里的内容（content），指的是元素、键和值。

这里也有一个教训送给那些使用可序列化类型的开发者们：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。如果你必须在某个类型 C 的 readObject 或 readResolve 方法中违背这条建议，请确定没有 C 的实例会出现在正在被反序列化的对象图的某个循环内。不幸的是，这不是一个本地的属性：一般说来，你需要考虑到整个系统来验证这一点。

总之，Java 的序列化系统是很脆弱的。为了正确而且高效地序列化大量的类，你必须编写 readObject 或 readResolve 方法[EJ Items 55-57]。这个谜题说明了，为了避免破坏反序列化的实例，你必须小心翼翼地编写这些方法。HashSet、

HashMap 和 Hashtable 的 readObject 方法很容易产生这种错误。对于平台设计者来说，如果你决定提供序列化系统，请不要提供如此脆弱的东西。健壮的序列化系统是很难设计的。

谜题 92：双绞线

下面这个程序使用一个匿名类执行了一个并不自然的动作。它会打印出什么呢？

```
public class Twisted {
    private final String name;
    Twisted(String name) {
        this.name = name;
    }
    private String name() {
        return name;
    }
    private void reproduce() {
        new Twisted("reproduce") {
            void printName() {
                System.out.println(name());
            }
        }.printName();
    }
    public static void main(String[] args) {
        new Twisted("main").reproduce();
    }
}
```

根据一个肤浅的分析会判断该程序不能通过编译。reproduce 方法中的匿名类试图调用 Twisted 类中的私有方法 name。一个类不能调用另一个类的私有方法，是吗？如果你试图编译这个程序，你会发现它可以成功地通过编译。在顶层的类型（top-level type）中，即本例中的 Twisted 类，所有的本地的、内部的、嵌套的和匿名的类都可以毫无限制地访问彼此的成员[JLS 6.6.1]。这是一个欢乐的大家庭。

在了解了这些之后，你可能会希望程序打印出 reproduce，因为它在 new Twisted(“reproduce”)实例上调用了 printName 方法，这个实例将字符串”reproduce”传给其超类的构造器使其存储到它的 name 域中。printName 方法调用 name 方法，name 方法返回了 name 域的内容。但是如果你运行这个程序，你会发现它打印的是 main。现在的问题是它为什么会做出这样的事情呢？

这种行为背后的原因是私有成员不会被继承[JLS 8.2]。在这个程序中，name 方法并没有被继承到 reproduce 方法中的匿名类中。所以，匿名类中对于 printName 方法的调用必须关联到外围(“main”)实例而不是当前(“reproduce”)实例。

这就是含有正确名称的方法的最小外围范围（enclosing scope）（谜题 71 和 79）。

这个程序违反了谜题 90 中的建议：在”reproduce”中的匿名类即是 Twisted 类的内部类又扩展了它。单独这一点就足以使程序难以阅读。再加上调用超类的私有方法的复杂度，这个程序就成了纯粹的冗长的废话。这个谜题可以用来强调谜题 6 中的教训：如果你不能通过阅读代码来分辨程序会做什么，那么它很可能不会做你想让它做的事。要尽量争取程序的清晰。

谜题 93：类的战争

下面这个谜题测试了你关于二进制兼容性（binary compatibility）的知识：当你改变了某个类所依赖的另外一个类时，第一个类的行为会发生什么改变呢？更特殊的是，假设你编译的是如下的 2 个类。第一个作为一个客户端，第二个作为一个库类，会怎么样呢：

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

public class Words {
    private Words() { };  // Uninstantiable

    public static final String FIRST  = "the";
    public static final String SECOND = null;
    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编译了这个类，但并不重编译客户端的程序：

```
public class Words {
    private Words() { };  // Uninstantiable

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

此时，客户端的程序会打印出什么呢？

简单地看看程序，你会觉得它应该打印 physics chemistry biology；毕竟 Java 是在运行期对类进行装载的，所以它总是会访问到最新版本的类。但是更深入一点的分析会得出不同的结论。对于常量域的引用会在编译期被转化为它们所表示的常量的值[JLS 13.1]。这样的域从技术上讲，被称作常量变量（constant variables），这可能在修辞上显得有点矛盾。一个常量变量的定义是：一个在编译期被常量表达式初始化的 final 的原始类型或 String 类型的变量[JLS 4.12.4]。在知道了这些知识之后，我们有理由认为客户端程序会将初始值 Words.FIRST, Words.SECOND, Words.THIRD 编译进 class 文件，然后无论 Words 类是否被改变，客户端都会打印 the null set。

这种分析可能是有道理的，但是却是不对的。如果你运行了程序，你会发现它打印的是 the chemistry set。这看起来确实太奇怪的了。它为什么会做出这种事情呢？答案可以在编译期常量表达式(compile-time constant expression)[JLS 15.28]的精确定义中找到。它的定义太长了，就不在这里写出来了，但是理解这个程序的行为的关键是 null 不是一个编译期常量表达式。

由于常量域将会编译进客户端，API 的设计者在设计一个常量域之前应该深思熟虑。如果一个域表示的是一个真实的常量，例如 π 或者一周之内的天数，那么将这个域设为常量域没有任何坏处。但是如果你想让客户端程序感知并适应这个域的变化，那么就不能让这个域成为一个常量。有一个简单的方法可以做到这一点：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 域，那么这个域就不是一个常量。你可以通过将一个常量表达式传给一个方法使得它变成一个非常量，该方法将直接返回其输入参数。

如果我们使用这种方法来修改 Word 类，在 Words 类被重新修改和编译之后，PrintWords 类将打印出 physics chemistry biology：

```
public class Words {
private Words() {};  // Uninstantiable

    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;
    }
}
```

在 5.0 版本中引入的枚举常量（enum constants），虽然有这样一个名字，但是它们并不是常量变量。你可以在枚举类型中加入枚举常量，对它们重新排序，甚至可以移除没有用的枚举常量，而且并不需要重新编译客户端。

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被常量表达式初始化的原始类型或字符串变量。令人惊讶的是，null 不是一个常量表达式。

对于语言设计者来说，在一个动态链接的语言中，将常量表达式编译进客户端可能并不是一个好主意。这让很多程序员大吃一惊，并且很容易产生一些难以查出的缺陷：当缺陷被侦测出来的时候，那些定义常量的源代码可能已经不存在了。另外一方面，将常量表达式编译进客户端使得我们可以使用 if 语句来模拟条件编译（conditional compilation）[JLS 14.21]。为了正当目的可以不择手段的做法是需要每个人自己来判断的。

谜题 94：迷失在混乱中

下面的 shuffle 方法声称它将公平的打乱它的输入数组的次序。换句话说，假设其使用的伪随机数发生器是公正的，它将会以均等的概率产生各种排列的数组。它真的兑现了它的诺言吗？如果没有，你将如何订正它呢？

```
import java.util.Random;
public class Shuffle {
    private static Random rnd = new Random();
    public static void shuffle(Object[] a) {
        for(int i = 0; i < a.length; i++)
            swap(a, i, rnd.nextInt(a.length));
    }
    private static void swap(Object[] a, int i, int j) {
        Object tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }
}
```

看看这个 shuffle 方法，它并没有什么明显的错误。它遍历了整个数组，将随机抽取的元素互换位置。这会公平地将数组打乱，对吗？不对。“它没有明显的错误”和“它明显没有错误”，这 2 种说法是很不同的。在这里，有很严重的错误，但是它并不明显，除非你专门研究算法。

如果你使用一个长度为 n 的数组作为参数去调用 shuffle 方法，这个循环体会执行 n 次。在每次执行中，这个方法会选取从 0 到 n-1 这 n 个整数中的一个。所以，该方法就有 nn 种不同的执行动作。我们假设随机数发生器是公平的，那么每一种执行动作出现的概率是相等的。每一种执行动作都产生数组的一种排列。但是，这里就有一个小问题：对于一个长度为 n 的数组来说，只有 n!种不同的排列。（在 n 之后的感叹号表示了阶乘（factorial）操作：n 的阶乘定义为 n×(n-1)×(n-2) ×…×1。）问题在于，对于任何大于 2 的 n，nn 都无法被 n！整除，因为 n!包含了从 2 到 n 的所有质数因子，而 nn 只包含了 n 所包含的质数因子。这就毫无疑问的证明了 shuffle 方法将会更多地产生某些排列。

为了使这个问题更具体一些，让我们来考虑一个包含了字符串”a”，”b”，”c”的长度为 3 的数组。此时 shuffle 方法就有 33 = 27 种执行动作。这些动作出现机率相同，并且都会产生某个排列。数组有 3! = 6 种不同的排列：{“a”,”b”,”c”}，{“a”,”c”,”b”}，{“b”,”a”,”c”}，{“b”,”c”,”a”}，{“c”,”a”,”b”}和{“c”,”b”,”a”}。由于 27 不能被 6 整除，比起其他的排列，某些排列肯定会被更多的执行动作所产生，所以 shuffle 方法并不是公平的。

这里的一个问题就是，上述的证明只是证明了 shuffle 方法确实存在偏差，而并没有提供任何这种偏差的感性材料。有时候深入了解的最好办法就是动手实验。我们让该方法操作“恒等数组”(identity array，即满足 a[i]=i 的数组 a)，然后测试程序将计算每个位置上的元素的期望值(expected value)。宽松的说，这个期望值，就是在重复运行 shuffle 方法的时候，你在数组的某个位置上看到的所有数值的平均值。如果 shuffle 方法是公平的，那么每个位置的元素的期望值应该是相等的：((n-1)/2)。图 10.1 显示了在一个长度为 9 的数组中各个元素的期望值。请注意这张图特殊的形状：开始的时候比较低，然后增长超过了公平值（4），然后在最后一个元素下降到公平值。

为什么这张图会有这种形状呢？我们不知道具体的细节，但是我们会有一些直觉上的认识。让我们把注意力集中到数组的第一个元素上。当循环体第一次执行之后，它会有正确的期望值(n-1)/2。然而在第 2 次执行中，有 n 分之 1 的可能性，随机数发生器会返回 0 且数组第一个元素的值会被设为 1 或 0。也就是说，第 2 次执行系统地减少了第一个元素的期望值。在第 3 次执行中，也会有 n 分之 1 的可能性，第一个元素的值会被设为 2、1 或者 0，然后就这么继续下去。在循环的前 n/2 次执行中，第一个元素的期望值是减少的。在后 n/2 次执行中，它的期望值是增加的，但是再也达不到它的公平值了。请注意，数组的最后一个元素肯定会有正确的期望值，因为在方法执行的最后一步，就是在数组的所有元素中为其选择一个值。

好了，我们的 shuffle 方法是坏掉了。我们怎么修复它呢？使用类库中提供的 shuffle 方法：

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

如果库中有可以满足你需要的方法，请务必使用它[EJ Item 30]。一般来说，库提供了高效的解决方案，并且可以让你付出最小的努力。

另外，在你忍受了所有这些数学的东西之后，如果不告诉你如何修复这个坏掉的 shuffle 方法是不公平的。修复方法是非常直接的。在循环体中，将当前的元素和某个在当前元素与数组末尾元素之间的所有元素中随机选择出来的元素进行互换。不要去碰那些你已经进行过值互换的元素。这本质上也就是库中的方法所使用的算法：

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

使用归纳法很容易证明这个方法是公平的。最基础情况，让我们观察长度为 0 的数组，这显然是公平的。根据归纳法的步骤，如果你将这个方法用在一个长度 n>0 的数组上，它会为这个数组的 0 位置上的元素随机选择一个值。然后，它会遍历数组剩下的元素：在每个位置上，它会在“子数组”中随机选择一个元素，这个子数组从当前位置开始到原数组的末尾。对于从位置 1 到原数组末尾的这个长度为 n-1 的子数组来说，如果将该方法作用在这个子数组上，它实际上也是在做上述的事。这就完成了证明。它同时也提供了 shuffle 方法的递归形式，它的细节就留给读者作为练习了。

你可能会认为到此为止就是故事的全部内容了，但却还有一部分内容。你设想过这个经过修复的 shuffle 方法会等概率的产生一个表示 52 张牌的 52 个元素的数组的所有排列吗？毕竟我们只是证明了它是公平的。在这里你可能不会很惊讶地发现答案很显然是“不”。这里的问题是，在谜题的开始，我们做出了“使用的伪随机数发生器是公平的”这一假设。但是它不是。

这个随机数发生器，java.util.Random，使用的是一个 64 位的种子，而它产生的随机数完全是由这个种子决定的。52 张牌有 52!种排列，而种子却只有 264 个。它能够覆盖的排列占所有排列的多少呢？你相信是百分之 2.3×10-47 吗？这只是委婉地表示了“实际上就没怎么覆盖”。如果你使用 java.security.SecureRandom 代替 java.util.Random，你会得到一个 160 位的种子，但是它给你带来的东西少得惊人：对于元素个数大于 40 的数组，这个 shuffle 方法仍然不能返回它的某些排列(因为 40!>2160) 。对于一个 52 个元素的数组，你只能获得所有可能的排列的百分之 1.8×10-18 。

这难道意味着你在洗牌的时候不能相信这些伪随机数发生器吗？这要看情况。它们确实只能产生所有可能排列的微不足道的一部分，但是它们没有我们前面所看到的那种系统性的偏差。公平地讲，这些发生器在非正式的场景中已经足够好用了。如果你需要一个尖端的随机数发生器，那你就需要到别的什么地方去寻找了。总之，像很多算法一样，打乱一个数组是需要慎重对待的。这么做很容易犯错并且很难发现错误。在其他条件相似的情况下，你应该优先使用类库而不是手写的代码。如果你想学习更多的关于本谜题的论题的内容，请参见[Knuth98 3.4.2]。

谜题 95：只是些甜点

本章的大多数谜题都是颇具挑战性的。但是这个不是。下面这个程序会打印出什么呢？如果你相信的话，前 2 个程序被报告为系统的缺陷[Bug 4157460 4763901]：

```
public class ApplePie {
    public static void main(String[] args) {
```

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

import java.util.*;
public class BananaBread {
    public static void main(String[] args) {
        Integer[] array = { 3, 1, 4, 1, 5, 9 };
        Arrays.sort(array, new Comparator<Integer>() {
            public int compare(Integer i1, Integer i2) {
                return i1 < i2 ? -1 : (i2 > i1 ? 1 : 0);
            }
        });
       System.out.println(Arrays.toString(array));
  }
}

public class ChocolateCake {
    public static void main(String[] args) {
         System.out.println(true?false:true == true?false:true);
    }
}
```

如果你受够这些东西了，那么你不需要知道这些愚蠢谜题的详细解释，所以让我们把它们变得又短又甜：

- 这个程序会打印出 1。这是由多余的标号造成的。（分号的恶习？）
- 这个程序在我们所知道的所有平台实现上都会打印出[3, 1, 4, 1, 5, 9]。从技术上说，程序的输出是未被定义的。它的比较器（comparator）承受着“是头我赢，是尾你输”的综合症。
- 这个程序会打印出 false。它书写的布局和它的操作符的优先级并不匹配。加一些括号可以解决问题。

这个谜题的教训，也是整本书的教训，就是：不要像我的兄弟那样编码。