幸运的是，面向对象的语言例如Java，就提供了其他更好的方法来定义能表示多种风格对象的单个数据类型：子类型化（subtyping）。标签类正是类层次的一种简单的仿效。

为了将标签类转变成类层次，首先要为标签类中的每个方法都定义一个包含抽象方法的抽象类，这每个方法的行为都依赖于标签值。在Figure类中，只有一个这样的方法：area。这个抽象类是类层次的根（root）。如果还有其他的方法其行为不依赖于标签的值，就把这样的方法放在这个类中。同样地，如果所有的方法都用到了某些数据域，就应该把它们放在这个类中。在Figure类中，不存在这种类型独立的方法或者数据域。

接下来，为每种原始标签类都定义根类的具体子类。在前面的例子中，这样的类型有两个：圆形（circle）和矩形（rectangle）。在每个子类中都包含特定于该类型的数据域。在我们的示例中，radius是特定于圆形的，length和width是特定于矩形的。同时在每个子类中还包括针对根类中每个抽象方法的相应实现。以下是与原始的Figure类相对应的类层次：

```
// Class hierarchy replacement for a tagged class
abstract class Figure {
    abstract double area();
}

class Circle extends Figure {
    final double radius;

    Circle(double radius) { this.radius = radius; }

    double area() { return Math.PI * (radius * radius); }
}
class Rectangle extends Figure {
    final double length;
    final double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width  = width;
    }
    double area() { return length * width; }
}
```

这个类层次纠正了前面提到过的标签类的所有缺点。这段代码简单且清楚，没有包含在原来的版本中所见到的所有样板代码。每个类型的实现都配有自己的类，这些类都没有受到不相关的数据域的拖累。所有的域都是final的。编译器确保每个类的构造器都初始化它的数据域，对于根类中声明的每个抽象方法，都确保有一个实现。这样就杜绝了由于遗漏switch case而导致运行时失败的可能性。多个程序员可以独立地扩展层次结构，并且不用访问根类的源代码就能相互操作。每种类型都有一种相关的独立的数据类型，允许程序员指明变量的类型，限制变量，并将参数输入到特殊的类型。

类层次的另一种好处在于，它们可以用来反映类型之间本质上的层次关系，有助于增强灵活性，并进行更好的编译时类型检查。假设上述例子中的标签类也允许表达正方形。类层次

可以反映出正方形是一种特殊的矩形这一事实（假设两者都是不可变的）：

```
class Square extends Rectangle {
    Square(double side) {
        super(side, side);
    }
}
```

注意，上述层次中的域是被直接访问，而不是通过访问方法。这是为了简洁起见才这么做的，如果层次结构是公有的（见第14条），则不允许这样做。

简而言之，标签类很少有适用的时候。当你想要编写一个包含显式标签域的类时，应该考虑一下，这个标签是否可以被取消，这个类是否可以用类层次来代替。当你遇到一个包含标签域的现有类时，就要考虑将它重构到一个层次结构中去。

第21条：用函数对象表示策略

有些语言支持函数指针（**function pointer**）、代理（**delegate**）、**lambda**表达式（**lambda expression**），或者支持类似的机制，允许程序把“调用特殊函数的能力”存储起来并传递这种能力。这种机制通常用于允许函数的调用者通过传入第二个函数，来指定自己的行为。例如，C语言标准库中的qsort函数要求用一个指向**comparator**（比较器）函数的指针作为参数，它用这个函数来比较待排序的元素。比较器函数有两个参数，都是指向元素的指针。如果第一个参数所指的元素小于第二个参数所指的元素，则返回一个负整数；如果两个元素相等则返回零；如果第一个参数所指的元素大于第二个参数所指的元素，则返回一个正整数。通过传递不同的比较器函数，就可以获得各种不同的排列顺序。这正是策略（Strategy）模式[Gamma95, p.315]的一个例子。比较器函数代表一种为元素排序的策略。

Java没有提供函数指针，但是可以用对象引用实现同样的功能。调用对象上的方法通常是执行该对象（**that object**）上的某项操作。然而，我们也可能定义这样一种对象，它的方法执行其他对象（**other objects**）（这些对象被显式传递给这些方法）上的操作。如果一个类仅仅导出这样的一个方法，它的实例实际上就等同于一个指向该方法的指针。这样的实例被称为函数对象（**function object**）。例如，考虑下面的类：

```
class StringLengthComparator {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

这个类导出一个带两个字符串参数的方法，如果第一个字符串的长度比第二个的短，则返回一个负整数；如果两个字符串的长度相等，则返回零；如果第一个字符串比第二个的长，则返回一个正整数。这个方法是一个比较器，它根据长度来给字符串排序，而不是根据更常用的字典顺序。指向StringLengthComparator对象的引用可以被当作是一个指向该比较器的“函数指针（function pointer）”，可以在任意一对字符串上被调用。换句话说，StringLength Comparator实例是用于字符串比较操作的具体策略（**concrete strategy**）。

作为典型的具体策略类，StringLengthComparator类是无状态的（**stateless**）：它没有域，所以，这个类的所有实例在功能上都是相互等价的。因此，它作为一个Singleton是非常合适的，可以节省不必要的对象创建开销（见第3条和第5条）：

```
class StringLengthComparator {
    private StringLengthComparator() { }
    public static final StringLengthComparator
        INSTANCE = new StringLengthComparator();
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
```

```
    }
}
```

为了把StringLengthComparator实例传递给方法，需要适当的参数类型。使用StringLengthComparator并不好，因为客户端将无法传递任何其他的比较策略。相反，我们需要定义一个Comparator接口，并修改StringLengthComparator来实现这个接口。换句话说，我们在设计具体的策略类时，还需要定义一个策略接口（**strategy interface**），如下所示：

```
// Strategy interface
public interface Comparator<T> {
    public int compare(T t1, T t2);
}
```

Comparator接口的这个定义碰巧也出现在java.util包中，但是这并不神奇；你自己也完全可以定义它。Comparator接口是泛型（见第26条）的，因此它适合作为除字符串之外的其他对象的比较器。它的compare方法的两个参数类型为T（它正常的类型参数），而不是String。只要声明前面所示的StringLengthComparator类要这么做，就可以用它实现Comparator<String>接口：

```
class StringLengthComparator implements Comparator<String> {
    ... // class body is identical to the one shown above
}
```

具体的策略类往往使用匿名类声明（见第22条）。下面的语句根据长度对一个字符串数组进行排序：

```
Arrays.sort(stringArray, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
});
```

但是注意，以这种方式使用匿名类时，将会在每次执行调用的时候创建一个新的实例。如果它被重复执行，考虑将函数对象存储到一个私有的静态final域里，并重用它。这样做的另一种好处是，可以为这个函数对象取一个有意义的域名称。

因为策略接口被用做所有具体策略实例的类型，所以我们并不需要为了导出具体策略，而把具体策略类做成公有的。相反，“宿主类（host class）”还可以导出公有的静态域（或者静态工厂方法），其类型为策略接口，具体的策略类可以是宿主类的私有嵌套类。下面的例子使用静态成员类，而不是匿名类，以便允许具体的策略类实现第二个接口Serializable：

```
// Exporting a concrete strategy
class Host {
    private static class StrLenCmp
            implements Comparator<String>, Serializable {
```

```
        public int compare(String s1, String s2) {
            return s1.length() - s2.length();
        }
    }

    // Returned comparator is serializable
    public static final Comparator<String>
        STRING_LENGTH_COMPARATOR = new StrLenCmp();

    ... // Bulk of class omitted
}
```

String类利用这种模式，通过它的CASE_INSENSITIVE_ORDER域，导出一个不区分大小写的字符串比较器。

简而言之，函数指针的主要用途就是实现策略（Strategy）模式。为了在Java中实现这种模式，要声明一个接口来表示该策略，并且为每个具体策略声明一个实现了该接口的类。当一个具体策略只被使用一次时，通常使用匿名类来声明和实例化这个具体策略类。当一个具体策略是设计用来重复使用的时候，它的类通常就要被实现为私有的静态成员类，并通过公有的静态final域被导出，其类型为该策略接口。

第22条：优先考虑静态成员类

嵌套类（**nested class**）是指被定义在另一个类的内部的类。嵌套类存在的目的应该只是为它的外围类（enclosing class）提供服务。如果嵌套类将来可能会用于其他的某个环境中，它就应该是顶层类（top-level class）。嵌套类有四种：*静态成员类*（**static member class**）、*非静态成员类*（**nonstatic member class**）、*匿名类*（**anonymous class**）和*局部类*（**local class**）。除了第一种之外，其他三种都被称为*内部类*（**inner class**）。本条目将告诉你什么时候应该使用哪种嵌套类，以及这样做的原因。

静态成员类是最简单的一种嵌套类。最好把它看作是普通的类，只是碰巧被声明在另一个类的内部而已，它可以访问外围类的所有成员，包括那些声明为私有的成员。静态成员类是外围类的一个静态成员，与其他的静态成员一样，也遵守同样的可访问性规则。如果它被声明为私有的，它就只能在外围类的内部才可以被访问，等等。

静态成员类的一种常见用法是作为公有的辅助类，仅当与它的外部类一起使用时才有意义。例如，考虑一个枚举，它描述了计算器支持的各种操作（见第30条）。Operation枚举应该是Calculator类的公有静态成员类，然后，Calculator类的客户端就可以用诸如Calculator.Operation.PLUS和Calculator.Operation.MINUS这样的名称来引用这些操作。

从语法上讲，静态成员类和非静态成员类之间唯一的区别是，静态成员类的声明中包含修饰符static。尽管它们的语法非常相似，但是这两种嵌套类有很大的不同。非静态成员类的每个实例都隐含着与外围类的一个外围实例（**enclosing instance**）相关联。在非静态成员类的实例方法内部，可以调用外围实例上的方法，或者利用修饰过的this构造获得外围实例的引用[JLS, 15.8.4]。如果嵌套类的实例可以在它外围类的实例之外独立存在，这个嵌套类就必须是静态成员类：在没有外围实例的情况下，要想创建非静态成员类的实例是不可能的。

当非静态成员类的实例被创建的时候，它和外围实例之间的关联关系也随之被建立起来；而且，这种关联关系以后不能被修改。通常情况下，当在外围类的某个实例方法的内部调用非静态成员类的构造器时，这种关联关系被自动建立起来。使用表达式enclosingInstance.new MemberClass(args)来手工建立这种关联关系也是有可能的，但是很少使用。正如你所预料的那样，这种关联关系需要消耗非静态成员类实例的空间，并且增加了构造的时间开销。

非静态成员类的一种常见用法是定义一个Adapter[Gamma95, p.139]，它允许外部类的实例被看作是另一个不相关的类的实例。例如，Map接口的实现往往使用非静态成员类来实现它们的*集合视图*（**collection view**），这些集合视图是由Map的keySet、entrySet和Values方法返回的。同样地，诸如Set和List这种集合接口的实现往往也使用非静态成员类来实现它们的迭代器（iterator）：

```
// Typical use of a nonstatic member class
public class MySet<E> extends AbstractSet<E> {
    ... // Bulk of the class omitted

    public Iterator<E> iterator() {
        return new MyIterator();
    }

    private class MyIterator implements Iterator<E> {
        ...
    }
}
```

如果声明成员类不要求访问外围实例，就要始终把**static**修饰符放在它的声明中，使它成为静态成员类，而不是非静态成员类。如果省略了static修饰符，则每个实例都将包含一个额外的指向外围对象的引用。保存这份引用要消耗时间和空间，并且会导致外围实例在符合垃圾回收（见第6条）时却仍然得以保留。如果在没有外围实例的情况下，也需要分配实例，就不能使用非静态成员类，因为非静态成员类的实例必须要有一个外围实例。

私有静态成员类的一种常见用法是用来代表外围类所代表的对象的组件。例如，考虑一个Map实例，它把键（key）和值（value）关联起来。许多Map实现的内部都有一个Entry对象，对应于Map中的每个键-值对。虽然每个entry都与一个Map关联，但是entry上的方法（getKey、getValue和setValue）并不需要访问该Map。因此，使用非静态成员来表示entry是很浪费的：私有的静态成员类是最佳的选择。如果不小心漏掉了entry声明中的static修饰符，该Map仍然可以工作，但是每个entry中将会包含一个指向该Map的引用，这样就浪费了空间和时间。

如果相关的类是导出类的公有的或受保护的成员，毫无疑问，在静态和非静态成员类之间做出正确的选择是非常重要的。在这种情况下，该成员类就是导出的API元素，在后续的发行版本中，如果不违反二进制兼容性，就不能从非静态成员类变为静态成员类。

匿名类不同于Java程序设计语言中的其他任何语法单元。正如你所想像的，匿名类没有名字。它不是外围类的一个成员。它并不与其他的成员一起被声明，而是在使用的同时被声明和实例化。匿名类可以出现在代码中任何允许存在表达式的地方。当且仅当匿名类出现在非静态的环境中时，它才有外围实例。但是即使它们出现在静态的环境中，也不可能拥有任何静态成员。

匿名类的适用性受到诸多的限制。除了在它们被声明的时候之外，是无法将它们实例化的。你不能执行instanceof测试，或者做任何需要命名类的其他事情。你无法声明一个匿名类来实现多个接口，或者扩展一个类，并同时扩展类和实现接口。匿名类的客户端无法调用任何成员，除了从它的超类型中继承得到之外。由于匿名类出现在表达式当中，它们必须保持简短——大约10行或者更少些——否则会影响程序的可读性。

匿名类的一种常见用法是动态地创建函数对象（**function object**，见第21条）。例如，第

92页中的sort方法调用，利用匿名的Comparator实例，根据一组字符串的长度对它们进行排序。匿名类的另一种常见用法是创建过程对象（**process object**），比如Runnable、Thread或者TimerTask实例。第三种常见的用法是在静态工厂方法的内部（参见第18条中的intArrayAsList方法）。

局部类是四种嵌套类中用得最少的类。在任何“可以声明局部变量”的地方，都可以声明局部类，并且局部类也遵守同样的作用域规则。局部类与其他三种嵌套类中的每一种都有一些共同的属性。与成员类一样，局部类有名字，可以被重复地使用。与匿名类一样，只有当局部类是在非静态环境中定义的时候，才有外围实例，它们也不能包含静态成员。与匿名类一样，它们必须非常简短，以便不会影响到可读性。

简而言之，共有四种不同的嵌套类，每一种都有自己的用途。如果一个嵌套类需要在单个方法之外仍然是可见的，或者它太长了，不适合于放在方法内部，就应该使用成员类。如果成员类的每个实例都需要一个指向其外围实例的引用，就要把成员类做成非静态的；否则，就做成静态的。假设这个嵌套类属于一个方法的内部，如果你只需要在一个地方创建实例，并且已经有了一个预置的类型可以说明这个类的特征，就要把它做成匿名类；否则，就做成局部类。

第5章

泛 型

Java 1.5发行版本中增加了泛型（**Generic**）。在没有泛型之前，从集合中读取到的每一个对象都必须进行转换。如果有人不小心插入了类型错误的对象，在运行时的转换处理就会出错。有了泛型之后，可以告诉编译器每个集合中接受哪些对象类型。编译器自动地为你的插入进行转化，并在编译时告知是否插入了类型错误的对象。这样可以使程序既更加安全，也更加清楚，但是要享有这些优势有一定的难度。本章就是教你如何最大限度地享有这些优势，又能使整个过程尽可能地简单化。有关这部分内容的详情，请参见Langer的教程[Langer08]，或者Naftalin和Wadler合著的书[Naftalin07]。

第23条：请不要在新代码中使用原生态类型

先来介绍一些术语。声明中具有一个或者多个*类型参数*（**type parameter**）的类或者接口，就是*泛型*（**generic**）类或者接口[JLS，8.1.2，9.1.2]。例如，从Java 1.5发行版本起，List接口就只有单个类型参数E，表示列表的元素类型。从技术的角度来看，这个接口的名称应该是指现在的List<E>（读作“E的列表”），但是人们经常把它简称为List。泛型类和接口统称为*泛型*（**generic type**）。

每种泛型定义一组*参数化的类型*（**parameterized type**），构成格式为：先是类或者接口的名称，接着用尖括号（< >）把对应于泛型形式类型参数的实际类型参数列表[JLS，4.4，4.5]括起来。例如，List<String>（读作“字符串列表”）是一个参数化的类型，表示元素类型为String的列表。（String是与形式类型参数E相对应的实际类型参数。）

最后一点，每个泛型都定义一个*原生态类型*（**raw type**），即不带任何实际类型参数的泛型名称[JLS，4.8]。例如，与List<E>相对应的原生态类型是List。原生态类型就像从类型声明中删除了所有泛型信息一样。实际上，原生态类型List与Java平台没有泛型之前的接口类型List完全一样。

在Java 1.5版本发行之前，以下集合声明是值得参考的：

```
// Now a raw collection type - don't do this!

/**
 * My stamp collection. Contains only Stamp instances.
 */
private final Collection stamps = ... ;
```

如果不小心将一个coin放进了stamp集合中，这一错误的插入照样得以编译和运行并且不会出现任何错误提示：

```
// Erroneous insertion of coin into stamp collection
stamps.add(new Coin( ... ));
```

直到从stamp集合中获取coin时才会收到错误提示：

```
// Now a raw iterator type - don't do this!
for (Iterator i = stamps.iterator(); i.hasNext(); ) {
    Stamp s = (Stamp) i.next(); // Throws ClassCastException
    ... // Do something with the stamp
}
```

就如本书中经常提到的，出错之后应该尽快发现，最好是编译时就发现。本例中，直到运行时才发现错误，已经出错很久了，而且你在代码中所处的位置距离包含错误的这部分代码已经很远了。一旦发现ClassCastException，就必须搜索代码，查找将coin放进stamp集合的方法调用。此时编译器帮不上忙，因为它无法理解这种注释："Contains only Stamp instances（只包含Stamp实例）"。

有了泛型，就可以利用改进后的类型声明来代替集合中的这种注释，告诉编译器之前的注释中所隐含的信息：

```
// Parameterized collection type - typesafe
private final Collection<Stamp> stamps = ... ;
```

通过这条声明，编译器知道stamps应该只包含Stamp实例，并给予保证，假设整个代码是利用Java 1.5及其之后版本的编译器进行编译的，所有代码在编译过程中都没有发出（或者禁止，请见第24条）任何警告。当stamps利用一个参数化的类型进行声明时，错误的插入会产生一条编译时的错误消息，准确地告诉你哪里出错了：

```
Test.java:9: add(Stamp) in Collection<Stamp> cannot be applied
to (Coin)
    stamps.add(new Coin());
               ^
```

还有一个好处是，从集合中删除元素时不再需要进行手工转换了。编译器会替你插入隐式的转换，并确保它们不会失败（依然假设所有代码都是通过支持泛型的编译器进行编译的，

并且没有产生或者禁止任何警告）。无论你是否使用for-each循环（见第46条），上述功能都适用：

```
// for-each loop over a parameterized collection - typesafe
for (Stamp s : stamps) { // No cast
    ... // Do something with the stamp
}
```

或者无论是否使用传统的for循环也一样：

```
// for loop with parameterized iterator declaration - typesafe
for (Iterator<Stamp> i = stamps.iterator(); i.hasNext(); ) {
    Stamp s = i.next(); // No cast necessary
    ... // Do something with the stamp
}
```

虽然假设不小心将coin插入到stamp集合中可能显得有点牵强，但这类问题却是真实的。例如，很容易想像有人会不小心将一个java.util.Date实例放进一个原本只包含java.sql.Date实例的集合中。

如上所述，如果不提供类型参数，使用集合类型和其他泛型也仍然是合法的，但是不应该这么做。*如果使用原生态类型，就失掉了泛型在安全性和表述性方面的所有优势。*既然不应该使用原生态类型，为什么Java的设计者还要允许使用它们呢？这是为了提供兼容性。因为泛型出现的时候，Java平台即将进入它的第二个10年，已经存在大量没有使用泛型的Java代码。人们认为让所有这些代码保持合法，并且能够与使用泛型的新代码互用，这一点很重要。它必须合法，才能将参数化类型的实例传递给那些被设计成使用普通类型的方法，反之亦然。这种需求被称作*移植兼容性*（Migration Compatibility），促成了支持原生态类型的决定。

虽然不应该在新代码中使用像List这样的原生态类型，使用参数化的类型以允许插入任意对象，如List<Object>，这还是可以的。原生态类型List和参数化的类型List<Object>之间到底有什么区别呢？不严格地说，前者逃避了泛型检查，后者则明确告知编译器，它能够持有任意类型的对象。虽然你可以将List<String>传递给类型List的参数，但是不能将它传给类型List<Object>的参数。泛型有子类型化（subtyping）的规则，List<String>是原生态类型List的一个子类型，而不是参数化类型List<Object>的子类型（见第25条）。因此，*如果使用像**List**这样的原生态类型，就会失掉类型安全性，但是如果使用像**List<Object>**这样的参数化类型，则不会。*

为了更具体地进行说明，请参考下面的程序：

```
// Uses raw type (List) - fails at runtime!
public static void main(String[] args) {
    List<String> strings = new ArrayList<String>();
    unsafeAdd(strings, new Integer(42));
    String s = strings.get(0); // Compiler-generated cast
```

```
}

private static void unsafeAdd(List list, Object o) {
    list.add(o);
}
```

这个程序可以进行编译，但是因为它使用了原生态类型List，你会收到一条警告：

```
Test.java:10: warning: unchecked call to add(E) in raw type List
    list.add(o);
             ^
```

实际上，如果运行这段程序，在程序试图将strings.get(0)的调用结果转换成一个String时，会收到一个ClassCastException异常。这是一个编译器生成的转换，因此一般保证会成功，但是我们在这个例子中忽略了一条编译器警告，就会为此而付出代价。

如果在unsafeAdd声明中用参数化类型List<Object>代替原生态类型List，并试着重新编译这段程序，会发现它无法再进行编译了。以下是它的错误消息：

```
Test.java:5: unsafeAdd(List<Object>,Object) cannot be applied
to (List<String>,Integer)
    unsafeAdd(strings, new Integer(42));
    ^
```

在不确定或者不在乎集合中的元素类型的情况下，你也许会使用原生态类型。例如，假设想要编写一个方法，它有两个集合（set），并从中返回它们共有的元素的数量。如果你对泛型还不熟悉的话，可以参考以下方式来编写这种方法：

```
// Use of raw type for unknown element type - don't do this!
static int numElementsInCommon(Set s1, Set s2) {
    int result = 0;
    for (Object o1 : s1)
        if (s2.contains(o1))
            result++;
    return result;
}
```

这个方法倒是可以，但它使用了原生态类型，这是很危险的。从Java 1.5发行版本开始，Java就提供了一种安全的替代方法，称作*无限制的通配符类型*（unbounded wildcard type）。如果要使用泛型，但不确定或者不关心实际的类型参数，就可以使用一个问号代替。例如，泛型Set<E>的无限制通配符类型为Set<?>（读作“某个类型的集合”）。这是最普通的参数化Set类型，可以持有任何集合。下面是numElementsInCommon方法使用了无限制通配符类型时的情形：

```
// Unbounded wildcard type - typesafe and flexible
static int numElementsInCommon(Set<?> s1, Set<?> s2) {
    int result = 0;
    for (Object o1 : s1)
```

```
            if (s2.contains(o1))
                result++;
        return result;
    }
```

在无限制通配类型Set<?>和原生态类型Set之间有什么区别呢？这个问号真正起到作用了吗？这一点不需要赘述，但通配符类型是安全的，原生态类型则不安全。由于可以将任何元素放进使用原生态类型的集合中，因此很容易破坏该集合的类型约束条件（如第100页的例子中所示的unsafeAdd方法）；但不能将任何元素（除了**null**之外）放到**Collection<?>**中。如果尝试这么做的话，将会产生一条像这样的编译时错误消息：

```
WildCard.java:13: cannot find symbol
symbol  : method add(String)
location: interface Collection<capture#825 of ?>
        c.add("verboten");
          ^
```

这样的错误消息显然还无法令人满意，但是编译器已经尽到了它的职责，防止你破坏集合的类型约束条件。你不仅无法将任何元素（除了null之外）放进Collection<?>中，而且根本无法猜测你会得到哪种类型的对象。要是无法接受这些限制，就可以使用泛型方法（generic method，见第27条）或者有限制的通配符类型（bounded wildcard type，见第28条）。

不要在新代码中使用原生态类型，这条规则有两个小小的例外，两者都源于“泛型信息可以在运行时被擦除”（见第25条）这一事实。**在类文字（class literal）中必须使用原生态类型。**规范不允许使用参数化类型（虽然允许数组类型和基本类型）[JLS，15.8.2]。换句话说，List.class，String[].class和int.class都合法，但是List<String.class和List<?>.class则不合法。

这条规则的第二个例外与instanceof操作符有关。由于泛型信息可以在运行时被擦除，因此在参数化类型而非无限制通配符类型上使用instanceof操作符是非法的。用无限制通配符类型代替原生态类型，对instanceof操作符的行为不会产生任何影响。在这种情况下，尖括号（<>）和问号（？）就显得多余了。**下面是利用泛型来使用instanceof操作符的首选方法：**

```
// Legitimate use of raw type - instanceof operator
if (o instanceof Set) {       // Raw type
    Set<?> m = (Set<?>) o;    // Wildcard type
    ...
}
```

注意，一旦确定这个o是个Set，就必须将它转换成通配符类型Set<?>，而不是转换成原生态类型Set。这是个受检的（checked）转换，因此不会导致编译时警告。

总之，使用原生态类型会在运行时导致异常，因此不要在新代码中使用。原生态类型只是为了与引入泛型之前的遗留代码进行兼容和互用而提供的。让我们做个快速的回顾：Set<Object>是个参数化类型，表示可以包含任何对象类型的一个集合；Set<?>则是一个通配

符类型，表示只能包含某种未知对象类型的一个集合；Set则是个原生态类型，它脱离了泛型系统。前两种是安全的，最后一种不安全。

为便于参考，表5-1概括了本条目中所介绍的术语（及本章其他条目中介绍的一些术语）：

表5-1 本章条目中所介绍的术语

术 语	示 例	所在条目
参数化的类型	List<String>	第23条
实际类型参数	String	第23条
泛型	List<E>	第23，26条
形式类型参数	E	第23条
无限制通配符类型	List<?>	第23条
原生态类型	List	第23条
有限制类型参数	<E extends Number>	第26条
递归类型限制	<T extends Comparable<T>>	第27条
有限制通配符类型	List<? extends Number>	第28条
泛型方法	static <E> List<E> asList(E[] a)	第27条
类型令牌	String.class	第29条

第24条：消除非受检警告

用泛型编程时，会遇到许多编译器警告：非受检强制转化警告（unchecked cast warnings）、非受检方法调用警告、非受检普通数组创建警告，以及非受检转换警告（unchecked conversion warnings）。当你越来越熟悉泛型之后，遇到的警告也会越来越少，但是不要期待从一开始用泛型编写代码就可以正确地进行编译。

有许多非受检警告很容易消除。例如，假设意外地编写了这样一个声明：

```
Set<Lark> exaltation = new HashSet();
```

编译器会细致地提醒你哪里出错了：

```
Venery.java:4: warning: [unchecked] unchecked conversion
found   : HashSet, required: Set<Lark>
    Set<Lark> exaltation = new HashSet();
                           ^
```

你就可以纠正所显示的错误，消除警告：

```
Set<Lark> exaltation = new HashSet<Lark>();
```

有些警告比较难以消除。本章主要介绍这种警告的示例。当你遇到需要进行一番思考的警告时，要坚持住！要尽可能地消除每一个非受检警告。如果消除了所有警告，就可以确保代码是类型安全的，这是一件很好的事情。这意味着不会在运行时出现ClassCastException异常，你会更加自信自己的程序可以实现预期的功能。

如果无法消除警告，同时可以证明引起警告的代码是类型安全的，（只有在这种情况下才）可以用一个**@SuppressWarnings ("unchecked")**注解来禁止这条警告。如果在禁止警告之前没有先证实代码是类型安全的，那就只是给你自己一种错误的安全感而已。代码在编译的时候可能没有出现任何警告，但它在运行时仍然会抛出ClassCastException异常。但是如果忽略（而不是禁止）明知道是安全的非受检警告，那么当新出现一条真正有问题的警告时，你也不会注意到。新出现的警告就会淹没在所有的错误警告当中。

SuppressWarnings注解可以用在任何粒度的级别中，从单独的局部变量声明到整个类都可以。应该始终在尽可能小的范围中使用**SuppressWarnings**注解。它通常是个变量声明，或是非常简短的方法或者构造器。永远不要在整个类上使用**SuppressWarnings**，这么做可能会掩盖了重要的警告。

如果你发现自己在长度不止一行的方法或者构造器中使用了**SuppressWarnings**注解，可以将它移到一个局部变量的声明中。虽然你必须声明一个新的局部变量，不过这么做还是值

得的。例如，考虑**ArrayList**类当中的**toArray**方法：

```
public <T> T[] toArray(T[] a) {
    if (a.length < size)
        return (T[]) Arrays.copyOf(elements, size, a.getClass());
    System.arraycopy(elements, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

如果编译**ArrayList**，该方法就会产生成这条警告：

```
ArrayList.java:305: warning: [unchecked] unchecked cast
found   : Object[], required: T[]
    return (T[]) Arrays.copyOf(elements, size, a.getClass());
                               ^
```

将**SuppressWarnings**注解放在**return**语句中是非法的，因为它不是一个声明[JLS,9.7]。你可以试着将注解放在整个方法上，但是在实践中千万不要这么做，而是应该声明一个局部变量来保存返回值，并注解其声明，像这样：

```
// Adding local variable to reduce scope of @SuppressWarnings
public <T> T[] toArray(T[] a) {
    if (a.length < size) {
        // This cast is correct because the array we're creating
        // is of the same type as the one passed in, which is T[].
        @SuppressWarnings("unchecked") T[] result =
            (T[]) Arrays.copyOf(elements, size, a.getClass());
        return result;
    }
    System.arraycopy(elements, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

这个方法可以正确地编译，禁止非受检警告的范围也减到了最小。

每当使用**SuppressWarnings ("unchecked")**注解时，都要添加一条注释，说明为什么这么做是安全的。这样可以帮助其他人理解代码，更重要的是，可以尽量减少其他人修改代码后导致计算不安全的概率。如果你觉得这种注释很难编写，就要多加思考。最终你会发现非受检操作是非常不安全的。

总而言之，非受检警告很重要，不要忽略它们。每一条警告都表示可能在运行时抛出**ClassCastException**异常。要尽最大的努力消除这些警告。如果无法消除非受检警告，同时可以证明引起警告的代码是类型安全的，就可以在尽可能小的范围中，用**@SuppressWarnings ("unchecked")**注解禁止该警告。要用注释把禁止该警告的原因记录下来。

第25条：列表优先于数组

数组与泛型相比，有两个重要的不同点。首先，数组是协变的（covariant）。这个词听起来有点吓人，其实只是表示如果Sub为Super的子类型，那么数组类型Sub[]就是Super[]的子类型。相反，泛型则是不可变的（invariant）：对于任意两个不同的类型Type1和Type2，List<Type1>既不是List<Type2>的子类型，也不是List<Type2>的超类型[JLS，4.10；Naftalin07，2.5]。你可能认为，这意味着泛型是有缺陷的，但实际上可以说数组才是有缺陷的。

下面的代码片段是合法的：

```
// Fails at runtime!
Object[] objectArray = new Long[1];
objectArray[0] = "I don't fit in"; // Throws ArrayStoreException
```

但下面这段代码则不合法：

```
// Won't compile!
List<Object> ol = new ArrayList<Long>(); // Incompatible types
ol.add("I don't fit in");
```

这其中无论哪种方法，都不能将String放进Long容器中，但是利用数组，你会在运行时发现所犯的错误；利用列表，则可以在编译时发现错误。我们当然希望在编译时发现错误了。

数组与泛型之间的第二大区别在于，数组是具体化的（reified）[JLS，4.7]。因此数组会在运行时才知道并检查它们的元素类型约束。如上所述，如果企图将String保存到Long数组中，就会得到一个ArrayStoreException异常。相比之下，泛型则是通过擦除（erasure）[JLS，4.6]来实现的。因此泛型只在编译时强化它们的类型信息，并在运行时丢弃（或者擦除）它们的元素类型信息。擦除就是使泛型可以与没有使用泛型的代码随意进行互用（见第23条）。

由于上述这些根本的区别，因此数组和泛型不能很好地混合使用。例如，创建泛型、参数化类型或者类型参数的数组是非法的。这些数组创建表达式没有一个是合法的：new List<E>[]、new List<String>[]和new E[]。这些在编译时都会导致一个generic array creation（泛型数组创建）错误。

为什么创建泛型数组是非法的？因为它不是类型安全的。要是它合法，编译器在其他正确的程序中发生的转换就会在运行时失败，并出现一个**ClassCastException**异常。这就违背了泛型系统提供的基本保证。

为了更具体地对此进行说明，考虑以下代码片断：

```
// Why generic array creation is illegal - won't compile!
```

```
List<String>[] stringLists = new List<String>[1];  // (1)
List<Integer> intList = Arrays.asList(42);         // (2)
Object[] objects = stringLists;                    // (3)
objects[0] = intList;                              // (4)
String s = stringLists[0].get(0);                  // (5)
```

我们假设第1行是合法的，它创建了一个泛型数组。第2行创建并初始化了一个包含单个元素的List<Integer>。第3行将List<String>数组保存到一个Object数组变量中，这是合法的，因为数组是协变的。第4行将List<Integer>保存到Object数组里唯一的元素中，这是可以的，因为泛型是通过擦除实现的：List<Integer>实例的运行时类型只是List，List<String>[]实例的运行时类型则是List[]，因此这种安排不会产生ArrayStoreException异常。但现在我们有麻烦了。我们将一个List<Integer>实例保存到了原本声明只包含List<String>实例的数组中。在第5行中，我们从这个数组里唯一的列表中获取了唯一的元素。编译器自动地将获取到的元素转换成String，但它是一个Integer，因此，我们在运行时得到了一个ClassCastException异常。为了防止出现这种情况，（创建泛型数组的）第1行产生了一个编译时错误。

从技术的角度来说，像E、List<E>和List<String>这样的类型应称作不可具体化的（non-reifiable）类型[JLS，4.7]。直观地说，不可具体化的（non-reifiable）类型是指其运行时表示法包含的信息比它的编译时表示法包含的信息更少的类型。唯一可具体化的（reifiable）参数化类型是无限制的通配符类型，如List<?>和Map<?,?>（见第23条）。虽然不常用，但是创建无限制通配类型的数组是合法的。

禁止创建泛型数组可能有点讨厌。例如，这表明泛型一般不可能返回它的元素类型数组（部分解决方案请见第29条）。这也意味着在结合使用可变参数（varargs）方法（见第42条）和泛型时会出现令人费解的警告。这是由于每当调用可变参数方法时，就会创建一个数组来存放varargs参数。如果这个数组的元素类型不是可具体化的（reifialbe），就会得到一条警告。关于这些警告，除了把它们禁止（见第24条），并且避免在API中混合使用泛型与可变参数之外，别无他法。

当你得到泛型数组创建错误时，最好的解决办法通常是优先使用集合类型List<E>，而不是数组类型E[]。这样可能会损失一些性能或者简洁性，但是换回的却是更高的类型安全性和互用性。

例如，假设有一个（Collections.synchronizedList返回的那种）同步列表和一个函数（它有两了与该列表的元素同类型的参数值，并返回第三个值）。现在假设要编写一个方法reduce，并使用函数apply来处理这个列表。假设列表元素类型为整数，并且函数是用来做两个整数的求和运算，reduce方法就会返回列表中所有值的总和。如果函数是用来做两个整数求积的运算，该方法就会返回列表中值的乘积。如果列表包含字符串，并且函数连接两个字符串，该方法就会返回一个字符串，它按顺序包含了列表中的所有字符串。除了列表和函数之外，reduce方法还采用初始值进行减法运算，列表为空时会返回这个初始值。（初始值一般为函数的识别元

素，加法为0，乘法为1，字符串连接时是""。）以下是没有泛型时的代码：

```
// Reduction without generics, and with concurrency flaw!
static Object reduce(List list, Function f, Object initVal) {
    synchronized(list) {
        Object result = initVal;
        for (Object o : list)
            result = f.apply(result, o);
        return result;
    }
}

interface Function {
    Object apply(Object arg1, Object arg2);
}
```

假设你现在已经读过第67条，它告诉你不要从同步区域中调用“外来的（alien）方法”。因此，在持有锁的时候修改reduce方法来复制列表中内容，也可以让你在备份上执行减法。Java 1.5发行版本之前，要这么做一般是利用List的toArray方法（它在内部锁定列表）：

```
// Reduction without generics or concurrency flaw
static Object reduce(List list, Function f, Object initVal) {
    Object[] snapshot = list.toArray(); // Locks list internally
    Object result = initVal;
    for (E e: snapshot)
        result = f.apply(result, e );
    return result;
}
```

如果试图通过泛型来完成这一点，就会遇到我们之前讨论过的那种麻烦。以下是Function接口的泛型版：

```
interface Function<T> {
    T apply(T arg1, T arg2);
}
```

下面是一种天真的尝试，试图将泛型应用到修改过的reduce方法。这是一个泛型方法（generic method，见第27条）。如果你不理解这条声明，也不必担心。对于这个条目来说，应该把注意力集中在方法体上：

```
// Naive generic version of reduction - won't compile!
static <E> E reduce(List<E> list, Function<E> f, E initVal) {
    E[] snapshot = list.toArray();  // Locks list
    E result = initVal;
    for (E e : snapshot)
        result = f.apply(result, e);
    return result;
}
```

如果试着编译这个方法，就会得到下面的错误消息：

```
Reduce.java:12: incompatible types
found   : Object[], required: E[]
```

```
        E[] snapshot = list.toArray();  // Locks list
                                    ^
```

你会说，这没什么大不了的，我会将Object数组转换成一个E数组：

```
E[] snapshot = (E[]) list.toArray();
```

它是消除了那条错误，但是现在得到了一条警告：

```
Reduce.java:12: warning: [unchecked] unchecked cast
found   : Object[], required: E[]
        E[] snapshot = (E[]) list.toArray();  // Locks list
                                          ^
```

编译器告诉你，它无法在运行时检查转换的安全性，因为它在运行时还不知道E是什么——记住，元素类型信息会在运行时从泛型中被擦除。这段程序可以运行吗？结果表明，它可以运行，但是不安全。通过微小的修改，就可以让它在没有包含显式转换的行上抛出ClassCastException异常。 snapshot的编译时类型为E[]，它可以为String[]、Integer[]或者任何其他种类的数组。运行时类型为Object[]，这是很危险的。不可具体化的类型的数组转换只能在特殊情况下使用（见第26条）。

那么应该做些什么呢？用列表代替数组。下面的reduce方法编译时就没有任何错误或者警告：

```
// List-based generic reduction
static <E> E reduce(List<E> list, Function<E> f, E initVal) {
    List<E> snapshot;
    synchronized(list) {
        snapshot = new ArrayList<E>(list);
    }
    E result = initVal;
    for (E e : snapshot)
        result = f.apply(result, e);
    return result;
}
```

这个版本的代码比数组版的代码稍微冗长一点，但是可以确定在运行时不会得到ClassCastException异常，为此也值了。

总而言之，数组和泛型有着非常不同的类型规则。数组是协变且可以具体化的；泛型是不可变的且可以被擦除的。因此，数组提供了运行时的类型安全，但是没有编译时的类型安全，反之，对于泛型也一样。一般来说，数组和泛型不能很好地混合使用。如果你发现自己将它们混合起来使用，并且得到了编译时错误或者警告，你的第一反应就应该是用列表代替数组。

第26条：优先考虑泛型

一般来说，将集合声明参数化，以及使用JDK所提供的泛型和泛型方法，这些都不太困难。编写自己的泛型会比较困难一些，但是值得花些时间去学习如何编写。

考虑第6条中这个简单的堆栈实现：

```
// Object-based collection - a prime candidate for generics
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

这个类是泛型化（generification）的主要备选对象，换句话说，可以适当地强化这个类来利用泛型。根据实际情况来看，必须转换从堆栈里弹出的对象，以及可能在运行时失败的那些转换。将类泛型化的第一个步骤是给它的声明添加一个或者多个类型参数。在这个例子中有一个类型参数，它表示堆栈的元素类型，这个参数的名称通常为E（见第44条）。

下一步是用相应的类型参数替换所有的Object类型，然后试着编译最终的程序：

```
// Initial attempt to generify Stack - won't compile!
public class Stack<E> {
    private E[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
```

```
        elements = new E[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(E e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public E pop() {
        if (size==0)
            throw new EmptyStackException();
        E result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }
    ... // no changes in isEmpty or ensureCapacity
}
```

通常，你将至少得到一个错误或警告，这个类也不例外。幸运的是，这个类只产生一个错误，如下：

```
Stack.java:8: generic array creation
        elements = new E[DEFAULT_INITIAL_CAPACITY];
                   ^
```

如第25条中所述，你不能创建不可具体化的（non-reifiable）类型的数组，如E。每当编写用数组支持的泛型时，都会出现这个问题。解决这个问题有两种方法。第一种，直接绕过创建泛型数组的禁令：创建一个Object的数组，并将它转换成泛型数组类型。现在错误是消除了，但是编译器会产生一条警告。这种用法是合法的，但（整体上而言）不是类型安全的：

```
Stack.java:8: warning: [unchecked] unchecked cast
found   : Object[], required: E[]
        elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
                   ^
```

编译器不可能证明你的程序是类型安全的，但是你可以证明。你自己必须确保未受检的转换不会危及到程序的类型安全性。相关的数组（即elements变量）保存在一个私有的域中，永远不会被返回到客户端，或者传给任何其他方法。这个数组中保存的唯一元素，是传给push方法的那些元素，它们的类型为E，因此未受检的转换不会有任何危害。

一旦你证明了未受检的转换是安全的，就要在尽可能小的范围中禁止警告（见第24条）。在这种情况下，构造器只包含未受检的数组创建，因此可以在整个构造器中禁止这条警告。通过增加一条注解来完成禁止，Stack能够正确无误地进行编译，你就可以使用它了，无需显式的转换，也无需担心会出现ClassCastException异常：

```
// The elements array will contain only E instances from push(E).
// This is sufficient to ensure type safety, but the runtime
// type of the array won't be E[]; it will always be Object[]!
@SuppressWarnings("unchecked")
public Stack() {
```

```
        elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
    }
```

消除Stack中泛型数组创建错误的第二种方法是，将elements域的类型从E[]改为Object[]。这么做会得到一条不同的错误：

```
Stack.java:19: incompatible types
found   : Object, required: E
        E result = elements[--size];
                   ^
```

通过把从数组中获取到的元素由Object转换成E，可以将这条错误变成一条警告：

```
Stack.java:19: warning: [unchecked] unchecked cast
found   : Object, required: E
        E result = (E) elements[--size];
                       ^
```

由于E是一个不可具体化的（non-reifiable）类型，编译器无法在运行时检验转换。你还是可以自己证实未受检的转换是安全的，因此可以禁止该警告。根据第24条的建议，我们只要在包含未受检转换的任务上禁止警告，而不是在整个pop方法上就可以了，如下：

```
// Appropriate suppression of unchecked warning
public E pop() {
    if (size==0)
        throw new EmptyStackException();

    // push requires elements to be of type E, so cast is correct
    @SuppressWarnings("unchecked") E result =
        (E) elements[--size];

    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

具体选择这两种方法中的哪一种来处理泛型数组创建错误，则主要看个人的偏好了。所有其他的东西都一样，但是禁止数组类型的未受检转换比禁止标量类型（scalar type）的更加危险，所以建议采用第二种方案。但是在比Stack更实际的泛型类中，或许代码中会有多个地方需要从数组中读取元素，因此选择第二种方案需要多次转换成E，而不是只转换成E[]，这也是第一种方案之所以更常用的原因[Naftalin07，6.7]。

下面的程序示范了泛型Stack类的使用。程序以相反的顺序打印出它的命令行参数，并转换成大写字母。如果要在从堆栈中弹出的元素上调用String的toUpperCase方法，并不需要显式的转换，并且会确保自动生成的转换会成功：

```
// Little program to exercise our generic Stack
public static void main(String[] args) {
    Stack<String> stack = new Stack<String>();
    for (String arg : args)
```

```
        stack.push(arg);
    while (!stack.isEmpty())
        System.out.println(stack.pop().toUpperCase());
}
```

看来上述的示例与第25条相矛盾了，第25条鼓励优先使用列表而非数组。实际上并不可能总是或者总想在泛型中使用列表。Java并不是生来就支持列表，因此有些泛型如ArrayList，则必须在数组上实现。为了提升性能，其他泛型如HashMap也在数组上实现。

绝大多数泛型就像我们的Stack示例一样，因为它们的类型参数没有限制：你可以创建Stack<Object>、Stack<int[]>、Stack<List<String>>，或者任何其他对象引用类型的Stack。注意不能创建基本类型的Stack：企图创建Stack<int>或者Stack<double>会产生一个编译时错误。这是Java泛型系统根本的局限性。你可以通过使用基本包装类型（boxed primitive type）来避开这条限制（见第49条）。

有一些泛型限制了可允许的类型参数值。例如，考虑java.util.concurrent.DelayQueue，其声明如下：

```
class DelayQueue<E extends Delayed> implements BlockingQueue<E>;
```

类型参数列表（<E extends Delayed>）要求实际的类型参数E必须是java.util.concurrent.Delayed的一个子类型。它允许DelayQueue实现及其客户端在DelayQueue的元素上利用Delayed方法，无需显式的转换，也没有出现ClassCastException的风险。类型参数E被称作有限制的类型参数（bounded type parameter）。注意，子类型关系确定了，每个类型都是它自身的子类型[JLS，4.10]，因此创建DelayQueue<Delayed>是合法的。

总而言之，使用泛型比使用需要在客户端代码中进行转换的类型来得更加安全，也更加容易。在设计新类型的时候，要确保它们不需要这种转换就可以使用。这通常意味着要把类做成是泛型的。只要时间允许，就把现有的类型都泛型化。这对于这些类型的新用户来说会变得更加轻松，又不会破坏现有的客户端（见第23条）。

第27条：优先考虑泛型方法

就如类可以从泛型中受益一般，方法也一样。静态工具方法尤其适合于泛型化。Collections中的所有“算法”方法（例如binarySearch和sort）都泛型化了。

编写泛型方法与编写泛型类型相类似。例如下面这个方法，它返回两个集合的联合：

```
// Uses raw types - unacceptable! (Item 23)
public static Set union(Set s1, Set s2) {
    Set result = new HashSet(s1);
    result.addAll(s2);
    return result;
}
```

这个方法可以编译，但是有两条警告：

```
Union.java:5: warning: [unchecked] unchecked call to
HashSet(Collection<? extends E>) as a member of raw type HashSet
        Set result = new HashSet(s1);
                     ^
Union.java:6: warning: [unchecked] unchecked call to
addAll(Collection<? extends E>) as a member of raw type Set
        result.addAll(s2);
                      ^
```

为了修正这些警告，使方法变成是类型安全的，要将方法声明修改为声明一个类型参数，表示这三个集合的元素类型（两个参数和一个返回值），并在方法中使用类型参数。声明类型参数的类型参数列表，处在方法的修饰符及其返回类型之间。在这个示例中，类型参数列表为<E>，返回类型为Set<E>。类型参数的命名惯例与泛型方法以及泛型的相同（见第26条和第44条）：

```
// Generic method
public static <E> Set<E> union(Set<E> s1, Set<E> s2) {
    Set<E> result = new HashSet<E>(s1);
    result.addAll(s2);
    return result;
}
```

至少对于简单的泛型方法而言，就是这么回事了。现在该方法编译时不会产生任何警告，并提供了类型安全性，也更容易使用。以下是一个执行该方法的简单程序。程序中不包含转换，编译时不会有错误或者警告：

```
// Simple program to exercise generic method
public static void main(String[] args) {
    Set<String> guys = new HashSet<String>(
        Arrays.asList("Tom", "Dick", "Harry"));
    Set<String> stooges = new HashSet<String>(
        Arrays.asList("Larry", "Moe", "Curly"));
    Set<String> aflCio = union(guys, stooges);
    System.out.println(aflCio);
}
```

运行这段程序时，会打印出[Moe, Harry, Tom, Curly, Larry, Dick]。元素的顺序是依赖于实现的。

union方法的局限性在于，三个集合的类型（两个输入参数和一个返回值）必须全部相同。利用有限制的通配符类型（bounded wildcard type），可以使这个方法变得更加灵活（见第28条）。

泛型方法的一个显著特性是，无需明确指定类型参数的值，不像调用泛型构造器的时候是必须指定的。编译器通过检查方法参数的类型来计算类型参数的值。对于上述的程序而言，编译器发现union的两个参数都是Set<String>类型，因此知道类型参数E必须为String。这个过程称作类型推导（type inference）。

如第1条所述，可以利用泛型方法调用所提供的类型推导，使创建参数化类型实例的过程变得更加轻松。提醒一下：在调用泛型构造器的时候，要明确传递类型参数的值可能有点麻烦。类型参数出现在了变量声明的左右两边，显得有些冗余：

```
// Parameterized type instance creation with constructor
Map<String, List<String>> anagrams =
    new HashMap<String, List<String>>();
```

为了消除这种冗余，可以编写一个泛型静态工厂方法（generic static factory method），与想要使用的每个构造器相对应。例如，下面是一个与无参的HashMap构造器相对应的泛型静态工厂方法：

```
// Generic static factory method
public static <K,V> HashMap<K,V> newHashMap() {
    return new HashMap<K,V>();
}
```

通过这个泛型静态工厂方法，可以用下面这段简洁的代码来取代上面那个重复的声明：

```
// Parameterized type instance creation with static factory
Map<String, List<String>> anagrams = newHashMap();
```

在泛型上调用构造器时，如果语言所做的类型推导与调用泛型方法时所做的相同，那就好了。将来的某一天也许可以实现这一点，但截至Java 1.6发行版本还不行。

相关的模式是泛型单例工厂（generic singleton factory）。有时，会需要创建不可变但又适合于许多不同类型的对象。由于泛型是通过擦除（见第25条）实现的，可以给所有必要的类型参数使用单个对象，但是需要编写一个静态工厂方法，重复地给每个必要的类型参数分发对象。这种模式最常用于函数对象（见第21条），如Collections.reverseOrder，但也适用于像Collections.emptySet这样的集合。

假设有一个接口，描述了一个方法，该方法接受和返回某个类型T的值：

```
public interface UnaryFunction<T> {
    T apply(T arg);
}
```

现在假设要提供一个恒等函数（identity function）。如果在每次需要的时候都重新创建一个，这样会很浪费，因为它是无状态的（stateless）。如果泛型被具体化了，每个类型都需要一个恒等函数，但是它们被擦除以后，就只需要一个泛型单例。请看以下示例：

```
// Generic singleton factory pattern
private static UnaryFunction<Object> IDENTITY_FUNCTION =
    new UnaryFunction<Object>() {
        public Object apply(Object arg) { return arg; }
    };

// IDENTITY_FUNCTION is stateless and its type parameter is
// unbounded so it's safe to share one instance across all types.
@SuppressWarnings("unchecked")
public static <T> UnaryFunction<T> identityFunction() {
    return (UnaryFunction<T>) IDENTITY_FUNCTION;
}
```

IDENTITY_FUNCTION转换成(UnaryFunction<T>)，产生了一条未受检的转换警告，因为UnaryFunction<Object>对于每个T来说并非都是个UnaryFunction<T>。但是恒等函数很特殊：它返回未被修改的参数，因此我们知道无论T的值是什么，用它作为UnaryFunction<T>都是类型安全的。因此，我们可以放心地禁止由这个转换所产生的未受检转换警告。一旦禁止，代码在编译时就不会出现任何错误或者警告。

以下是一个范例程序，利用泛型单例作为UnaryFunction<String>和 UnaryFunction<Number>。像往常一样，它不包含转换，编译时没有出现错误或者警告：

```
// Sample program to exercise generic singleton
public static void main(String[] args) {
    String[] strings = { "jute", "hemp", "nylon" };
    UnaryFunction<String> sameString = identityFunction();
    for (String s : strings)
        System.out.println(sameString.apply(s));

    Number[] numbers = { 1, 2.0, 3L };
    UnaryFunction<Number> sameNumber = identityFunction();
    for (Number n : numbers)
        System.out.println(sameNumber.apply(n));
}
```

虽然相对少见，但是通过某个包含该类型参数本身的表达式来限制类型参数是允许的。这就是*递归类型限制*（**recursive type bound**）。递归类型限制最普遍的用途与Comparable接口有关，它定义类型的自然顺序：

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

类型参数T定义的类型，可以与实现Comparable<T>的类型的元素进行比较。实际上，几乎所有的类型都只能与它们自身的类型的元素相比较。因此，例如String实现Comparable<String>，Integer实现Comparable<Integer>，等等。

有许多方法都带有一个实现Comparable接口的元素列表，为了对列表进行排序，并在其中进行搜索，计算出它的最小值或者最大值，等等。要完成这其中的任何一项工作，要求列表中的每个元素要都能够与列表中的每个其他元素相比较，换句话说，列表的元素可以互相比较（mutually comparable）。下面是如何表达这种约束条件的一个示例：

```
// Using a recursive type bound to express mutual comparability
public static <T extends Comparable<T>> T max(List<T> list) {...}
```

类型限制<T extends Comparable<T>>，可以读作“针对可以与自身进行比较的每个类型T”，这与互比性的概念或多或少有些一致。

下面的方法就带有上述声明。它根据元素的自然顺序计算列表的最大值，编译时没有出现错误或者警告：

```
// Returns the maximum value in a list - uses recursive type bound
public static <T extends Comparable<T>> T max(List<T> list) {
    Iterator<T> i = list.iterator();
    T result = i.next();
    while (i.hasNext()) {
        T t = i.next();
        if (t.compareTo(result) > 0)
            result = t;
    }
    return result;
}
```

递归类型限制可能比这个要复杂得多，但幸运的是，这种情况并不经常发生。如果你理解了这种习惯用法及其通配符变量（见第28条），就能够处理在实践中遇到的许多递归类型限制了。

总而言之，泛型方法就像泛型一样，使用起来比要求客户端转换输入参数并返回值的方法来得更加安全，也更加容易。就像类型一样，你应该确保新方法可以不用转换就能使用，这通常意味着要将它们泛型化。并且就像类型一样，还应该将现有的方法泛型化，使新用户使用起来更加轻松，且不会破坏现有的客户端（见第23条）。

第28条：利用有限制通配符来提升API的灵活性

如第25条所述，参数化类型是不可变的（invariant）。换句话说，对于任何两个截然不同的类型Type1和Type2而言，List<Type1>既不是List<Type2>的子类型，也不是它的超类型。虽然List<String>不是List<Object>的子类型，这与直觉相悖，但是实际上很有意义。你可以将任何对象放进一个List<Object>中，却只能将字符串放进List<String>中。

有时候，我们需要的灵活性要比不可变类型所能提供的更多。考虑第26条中的堆栈下面就是它的公共API：

```
public class Stack<E> {
    public Stack();
    public void push(E e);
    public E pop();
    public boolean isEmpty();
}
```

假设我们想要增加一个方法，让它按顺序将一系列的元素全部放到堆栈中。这是第一次尝试，如下：

```
// pushAll method without wildcard type - deficient!
public void pushAll(Iterable<E> src) {
    for (E e : src)
        push(e);
}
```

这个方法编译时正确无误，但是并非尽如人意。如果Iterable src的元素类型与堆栈的完全匹配，就没有问题。但是假如有一个Stack<Number>，并且调用了push(intVal)，这里的intVal就是Integer类型。这是可以的，因为Integer是Number的一个子类型。因此从逻辑上来说，下面这个方法应该也可以：

```
Stack<Number> numberStack = new Stack<Number>();
Iterable<Integer> integers = ... ;
numberStack.pushAll(integers);
```

但是，如果尝试这么做，就会得到下面的错误消息，因为如前所述，参数化类型是不可变的：

```
StackTest.java:7: pushAll(Iterable<Number>) in Stack<Number>
cannot be applied to (Iterable<Integer>)
        numberStack.pushAll(integers);
                    ^
```

幸运的是，有一种解决办法。Java提供了一种特殊的参数化类型，称作有限制的通配符类型（bounded wildcard type），来处理类似的情况。pushAll的输入参数类型不应该为“E的

Iterable接口”，而应该为“E的某个子类型的Iterable接口”，有一个通配符类型正符合此意：Iterable<? Extends E>。（使用关键字extends有些误导：回忆一下第26条中的说法，确定了子类型（subtype）后，每个类型便都是自身的子类型，即便它没有将自身扩展。）我们修改一下pushAll来使用这个类型：

```
// Wildcard type for parameter that serves as an E producer
public void pushAll(Iterable<? extends E> src) {
    for (E e : src)
        push(e);
}
```

这么修改了之后，不仅Stack可以正确无误地编译，没有通过初始的pushAll声明进行编译的客户端代码也一样可以。因为Stack及其客户端正确无误地进行了编译，你就知道一切都是类型安全的了。

现在假设想要编写一个pushAll方法，使之与popAll方法相呼应。popAll方法从堆栈中弹出每个元素，并将这些元素添加到指定的集合中。初次尝试编写的popAll方法可能像下面这样：

```
// popAll method without wildcard type - deficient!
public void popAll(Collection<E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

如果目标集合的元素类型与堆栈的完全匹配，这段代码编译时还是会正确无误，运行得很好。但是，也并不意味着尽如人意。假设你有一个Stack<Number>和类型Object的变量。如果从堆栈中弹出一个元素，并将它保存在该变量中，它的编译和运行都不会出错，那你为何不能也这么做呢？

```
Stack<Number> numberStack = new Stack<Number>();
Collection<Object> objects = ... ;
numberStack.popAll(objects);
```

如果试着用上述的popAll版本编译这段客户端代码，就会得到一个非常类似于第一次用pushAll时所得到的错误：Collection<Object>不是Collection<Number>的子类型。这一次，通配符类型同样提供了一种解决办法。popAll的输入参数类型不应该为“E的集合”，而应该为“E的某种超类的集合”（这里的超类是确定的，因此E是它自身的一个超类型[JLS，4.10]）。仍然有一个通配符类型正是符合此意：Collection<? super E>。让我们修改popAll来使用它：

```
// Wildcard type for parameter that serves as an E consumer
public void popAll(Collection<? super E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

做了这个变动之后，Stack和客户端代码就都可以正确无误地编译了。

结论很明显。为了获得最大限度的灵活性，要在表示生产者或者消费者的输入参数上使用通配符类型。如果某个输入参数既是生产者，又是消费者，那么通配符类型对你就没有什么好处了：因为你需要的是严格的类型匹配，这是不用任何通配符而得到的。

下面的助记符便于让你记住要使用哪种通配符类型：

PECS表示producer-extends，consumer-super。

换句话说，如果参数化类型表示一个T生产者，就使用<? extends T>；如果它表示一个T消费者，就使用<? super T>。在我们的Stack示例中，pushAll的src参数产生E实例供Stack使用，因此src相应的类型为Iterable<? extends E>；popAll的dst参数通过Stack消费E实例，因此dst相应的类型为Collection<? super E>。PECS这个助记符突出了使用通配符类型的基本原则。Naftalin和Wadler称之为**Get and Put Principle** [Naftalin07，2.4]。

记住这个助记符，我们下面来看一些之前的条目中提到过的方法声明。第25条中的reduce方法就有这条声明：

```
static <E> E reduce(List<E> list, Function<E> f, E initVal)
```

虽然列表既可以消费也可以产生值，reduce方法还是只用它的list参数作为**E生产者（producer）**，因此它的声明就应该使用一个extends E的通配符类型。参数f表示既可以消费又可以产生E实例的函数，因此通配符类型不适合它。得到的方法声明如下：

```
// Wildcard type for parameter that serves as an E producer
static <E> E reduce(List<? extends E> list, Function<E> f,
                    E initVal)
```

这一变化实际上有什么区别吗？事实上，的确有区别。假设你有一个List<Integer>，想通过Function<Number>把它简化。它不能通过初始声明进行编译，但是一旦添加了有限制的通配符类型，就可以了。

现在让我们看看第27条中的union方法。下面是声明：

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
```

s1和s2这两个参数都是E消费者，因此根据PECS，这个声明应该是：

```
public static <E> Set<E> union(Set<? extends E> s1,
                               Set<? extends E> s2)
```

注意返回类型仍然是Set<E>。不要用通配符类型作为返回类型。除了为用户提供额外的

灵活性之外，它还会强制用户在客户端代码中使用通配符类型。

如果使用得当，通配符类型对于类的用户来说几乎是无形的。它们使方法能够接受它们应该接受的参数，并拒绝那些应该拒绝的参数。*如果类的用户必须考虑通配符类型，类的API或许就会出错。*

遗憾的是，类型推导（type inference）规则相当复杂，在语言规范中占了整整16页[JLS, 15.12.2.7-8]，而且它们并非总能完成需要它们完成的工作。看看修改过的union声明，你可能会以为可以像这样编写：

```
Set<Integer> integers = ... ;
Set<Double> doubles = ... ;
Set<Number> numbers = union(integers, doubles);
```

但这么做会得到下面的错误消息：

```
Union.java:14: incompatible types
found : Set<Number & Comparable<? extends Number &
                                Comparable<?>>>
required: Set<Number>
        Set<Number> numbers = union(integers, doubles);
                              ^
```

幸运的是，有一种办法可以处理这种错误。如果编译器不能推断你希望它拥有的类型，可以通过一个*显式的类型参数*（explicit type parameter）来告诉它要使用哪种类型。这种情况不太经常发生，这是好事，因为显式的类型参数不太优雅。增加了这个显式的类型参数之后，程序可以正确无误地进行编译：

```
Set<Number> numbers = Union.<Number>union(integers, doubles);
```

接下来，我们把注意力转向第27条中的max方法。以下是初始的声明：

```
public static <T extends Comparable<T>> T max(List<T> list)
```

下面是修改过的使用通配符类型的声明：

```
public static <T extends Comparable<? super T>> T max(
        List<? extends T> list)
```

为了从初始声明中得到修改后的版本，要应用PECS转换两次。最直接的是运用到参数list。它产生T实例，因此将类型从List<T>改成List<? extends T>。更灵活的是运用到类型参数T。这是我们第一次见到将通配符运用到类型参数。最初T被指定用来扩展Comparable<T>，但是T的comparable消费T实例（并产生表示顺序关系的整值）。因此，参数化类型Comparable<T>被有限制通配符类型Comparable<? super T>取代。comparable始终是消费者，因此*使用时始终应该是*Comparable<? super T>*优先于*Comparable<T>。对于comparator也一样，因此使

用时始终应该是Comparator<? super T>优先于Comparator<T>。

修改过的max声明可能是整本书中最复杂的方法声明了。所增加的复杂代码真的起作用了么？是的，起作用了。下面是一个简单的列表示例，在初始的声明中不允许这样，修改过的版本则可以：

```
List<ScheduledFuture<?>> scheduledFutures = ... ;
```

不能将初始方法声明运用给这个列表的原因在于，java.util.concurrent.ScheduledFuture没有实现Comparable<ScheduledFuture>接口。相反，它是扩展Comparable<Delayed>接口的Delayed接口的子接口。换句话说，ScheduleFuture实例并非只能与其他ScheduledFuture实例相比较；它可以与任何Delayed实例相比较，这就足以导致初始声明时就会被拒绝。

修改过的max声明有一个小小的问题：它阻止方法进行编译。下面的方法包含了修改过的声明：

```
// Won't compile - wildcards can require change in method body!
public static <T extends Comparable<? super T>> T max(
        List<? extends T> list) {
    Iterator<T> i = list.iterator();
    T result = i.next();
    while (i.hasNext()) {
        T t = i.next();
        if (t.compareTo(result) > 0)
            result = t;
    }
    return result;
}
```

以下是它编译时会产生的错误消息：

```
Max.java:7: incompatible types
found   : Iterator<capture#591 of ? extends T>
required: Iterator<T>
        Iterator<T> i = list.iterator();
                                    ^
```

这条错误消息意味着什么，我们又该如何修正这个问题呢？它意味着list不是一个List<T>，因此它的iterator方法没有返回Iterator<T>。它返回T的某个子类型的一个iterator，因此我们用它代替iterator声明，它使用了一个有限制的通配符类型：

```
Iterator<? extends T> i = list.iterator();
```

这是必须对方法体所做的唯一修改。迭代器的next方法返回的元素属于T的某个子类型，因此它们可以被安全地保存在类型T的一个变量中。

还有一个与通配符有关的话题值得探讨。类型参数和通配符之间具有双重性，许多方法都可以利用其中一个或者另一个进行声明。例如，下面是可能的两种静态方法声明，来交换列表中的

两个被索引的项目。第一个使用无限制的类型参数（见第27条），第二个使用无限制的通配符：

```
// Two possible declarations for the swap method
public static <E> void swap(List<E> list, int i, int j);
public static void swap(List<?> list, int i, int j);
```

你更喜欢这两种方法中的哪一种呢？为什么？在公共API中，第二种更好一些，因为它更简单。将它传到一个列表中——任何列表——方法就会交换被索引的元素。不用担心类型参数。一般来说，*如果类型参数只在方法声明中出现一次，就可以用通配符取代它。*如果是无限制的类型参数，就用无限制的通配符取代它；如果是有限制的类型参数，就用有限制的通配符取代它。

将第二种声明用于swap方法会有一个问题，它优先使用通配符而非类型参数：下面这个简单的实现都不能编译：

```
public static void swap(List<?> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

试着编译时会产生这条没有什么用处的错误消息：

```
Swap.java:5: set(int,capture#282 of ?) in List<capture#282 of ?>
cannot be applied to (int,Object)
        list.set(i, list.set(j, list.get(i)));
                            ^
```

不能将元素放回到刚刚从中取出的列表中，这似乎不太对劲。问题在于list的类型为List<?>，你不能把null之外的任何值放到List<?>中。幸运的是，有一种方式可以实现这个方法，无需求助于不安全的转换或者原生态类型（raw type）。这种想法就是编写一个私有的辅助方法来*捕捉*通配符类型。为了捕捉类型，辅助方法必须是泛型方法，像下面这样：

```
public static void swap(List<?> list, int i, int j) {
    swapHelper(list, i, j);
}

// Private helper method for wildcard capture
private static <E> void swapHelper(List<E> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

swapHelper方法知道list是一个List<E>。因此，它知道从这个列表中取出的任何值均为E类型，并且知道将E类型的任何值放进列表都是安全的。swap这个有些费解的实现编译起来却是正确无误的。它允许我们导出swap这个比较好的基于通配符的声明，同时在内部利用更加复杂的泛型方法。swap方法的客户端不一定要面对更加复杂的swapHelper声明，但是它们的确从中受益。

总而言之，在API中使用通配符类型虽然比较需要技巧，但是使API变得灵活得多。如果编写的是将被广泛使用的类库，则一定要适当地利用通配符类型。记住基本的原则：producer-extends, consumer-super (PECS)。还要记住所有的comparable和comparator都是消费者。

第29条：优先考虑类型安全的异构容器

泛型最常用于集合，如Set和Map，以及单元素的容器，如ThreadLocal和AtomicReference。在这些用法中，它都充当被参数化了的容器。这样就限制你每个容器只能有固定数目的类型参数。一般来说，这种情况正是你想要的。一个Set只有一个类型参数，表示它的元素类型；一个Map有两个类型参数，表示它的键和值类型；诸如此类。

但是，有时候你会需要更多的灵活性。例如，数据库行可以有任意多的列，如果能以类型安全的方式访问所有列就好了。幸运的是，有一种方法可以很容易地做到这一点。这种想法就是将键（key）进行参数化而不是将容器（container）参数化。然后将参数化的键提交给容器，来插入或者获取值。用泛型系统来确保值的类型与它的键相符。

简单地示范一下这种方法：考虑Favorites类，它允许其客户端从任意数量的其他类中，保存并获取一个“最喜爱”的实例。Class对象充当参数化键的部分。之所以可以这样，是因为类Class在Java 1.5版本中被泛型化了。类的类型从字面上来看不再只是简单的**Class**，而是**Class<T>**。例如，**String.class**属于**Class<String>**类型，**Integer.class**属于**Class<Integer>**类型。当一个类的字面文字被用在方法中，来传达编译时和运行时的类型信息时，就被称作**type token**[Brancha04]。

Favorites类的API很简单。它看起来就像一个简单的map，除了键（而不是map）被参数化之外。客户端在设置和获取最喜爱的实例时提交Class对象。下面就是这个API：

```
// Typesafe heterogeneous container pattern - API
public class Favorites {
    public <T> void putFavorite(Class<T> type, T instance);
    public <T> T getFavorite(Class<T> type);
}
```

下面是一个示例程序，检验一下Favorites类，它保存、获取并打印一个最喜爱的String、Integer和Class实例：

```
// Typesafe heterogeneous container pattern - client
public static void main(String[] args) {
    Favorites f = new Favorites();
    f.putFavorite(String.class, "Java");
    f.putFavorite(Integer.class, 0xcafebabe);
    f.putFavorite(Class.class, Favorites.class);
    String favoriteString = f.getFavorite(String.class);
    int favoriteInteger = f.getFavorite(Integer.class);
    Class<?> favoriteClass = f.getFavorite(Class.class);
    System.out.printf("%s %x %s%n", favoriteString,
        favoriteInteger, favoriteClass.getName());
}
```

正如所料，这段程序打印出的是Java cafebabe Favorites。

Favorites实例是*类型安全*（typesafe）的：当你向它请求String的时候，它从来不会返回一个Integer给你。同时它也是*异构的*（**heterogeneous**）：不像普通的map，它的所有键都是不同类型的。因此，我们将Favorites称作*类型安全的异构容器*（typesafe heterogeneous container）。

Favorites的实现小得出奇。它的完整实现如下：

```
// Typesafe heterogeneous container pattern - implementation
public class Favorites {
    private Map<Class<?>, Object> favorites =
        new HashMap<Class<?>, Object>();

    public <T> void putFavorite(Class<T> type, T instance) {
        if (type == null)
            throw new NullPointerException("Type is null");
        favorites.put(type, instance);
    }

    public <T> T getFavorite(Class<T> type) {
        return type.cast(favorites.get(type));
    }
}
```

这里发生了一些微妙的事情。每个Favorites实例都得到一个称作favorites的私有Map<Class<?>, Object>的支持。你可能认为由于无限制通配符类型的关系，将不能把任何东西放进这个Map中，但事实正好相反。要注意的是通配符类型是嵌套的：它不是属于通配符类型的Map的类型，而是它的键的类型。由此可见，每个键都可以有一个不同的参数化类型：一个可以是Class<String>，接下来是Class<Integer>等等。异构就是从这里来的。

第二件要注意的事情是，favorites Map的值类型只是Object。换句话说，Map并不能保证键和值之间的类型关系，即不能保证每个值的类型都与键的类型相同。事实上，Java的类型系统还没有强大到足以表达这一点。但我们知道这是事实，并在获取favorite的时候利用了这一点。

putFavorite方法的实现很简单：它只是把（从指定的Class对象到指定favorite实例的）一个映射放到favorites中。如前所述，这是放弃了键和值之间的“类型联系”，因此无法知道这个值是键的一个实例。但是没关系，因为getFavorites方法能够并且的确重新建立了这种联系。

getFavorite方法的实现比putFavorite的更难一些。它先从favorites映射中获得与指定Class对象相对应的值。这正是要返回的对象引用，但它的编译时类型是错误的。它的类型只是Object（favorites映射的值类型），我们需要返回一个T。因此，getFavorite方法的实现利用Class的cast方法，将对象引用*动态地转换*（dynamically cast）成了Class对象所表示的类型。

cast方法是Java的cast操作符的动态模拟。它只检验它的参数是否为Class对象所表示的类型的实例。如果是，就返回参数；否则就抛出ClassCastException异常。我们知道，getFavorite中的cast调用永远不会抛出ClassCastException异常，并假设客户端代码正确无误地进行了编译。也就是说，我们知道favorites映射中的值会始终与键的类型相匹配。

假设cast方法只返回它的参数，那它能为我们做什么呢？cast方法的签名充分利用了Class类被泛型化的这个事实。它的返回类型是Class对象的类型参数：

```
public class Class<T> {
    T cast(Object obj);
}
```

这正是getFavorite方法所需要的，也正是让我们不必借助于未受检地转换成T就能确保Favorites类型安全的东西。

Favorites类有两种局限性值得注意。首先，恶意的客户端可以很轻松地破坏Favorites实例的类型安全，只要以它的原生态形式（raw form）使用Class对象。但会造成客户端代码在编译时产生未受检的警告。这与一般的集合实现，如HashSet和HashMap并没有什么区别。你可以很容易地利用原生态类型HashSet（见第23条）将String放进HashSet<Integer>中。也就是说，如果愿意付出一点点代价，就可以拥有运行时的类型安全。确保Favorites永远不违背它的类型约束条件的方式是，让putFavorite方法检验instance是否真的是type所表示的类型的实例。我们已经知道这要如何进行了，只要使用一个动态的转换：

```
// Achieving runtime type safety with a dynamic cast
public <T> void putFavorite(Class<T> type, T instance) {
    favorites.put(type, type.cast(instance));
}
```

java.util.Collections中有一些集合包装类采用了同样的技巧。它们称作checkedSet、checkedList、checkedMap，诸如此类。除了一个集合（或者映射）之外，它们的静态工厂还采用一个（或者两个）Class对象。静态工厂属于泛型方法，确保Class对象和集合的编译时类型相匹配。包装类给它们所封装的集合增加了具体化。例如，如果有人试图将Coin放进你的Collection<Stamp>，包装类就会在运行时抛出ClassCastException异常。用这些包装类在混有泛型和遗留代码的应用程序中追溯“谁把错误的类型元素添加到了集合中”很有帮助。

Favorites类的第二种局限性在于它不能用在不可具体化的（non-reifiable）类型中（见第25条）。换句话说，你可以保存最喜爱的String或者String[]，但不能保存最喜爱的List<String>。如果试图保存最喜爱的List<String>，程序就不能进行编译。原因在于你无法为List<String>获得一个Class对象：List<String>.Class是个语法错误，这也是件好事。List<String>和List<Integer>共用一个Class对象，即List.class。如果从“字面（type literal）”

上来看，List<String>.class和List<Integer>.class是合法的，并返回了相同的对象引用，就会破坏Favorites对象的内部结构。

对于第二种局限性，还没有完全令人满意的解决办法。有一种方法称作super type token，它在解决这一局限性方面做了很多努力，但是这种方法仍有它自身的局限性[Gafter07]。

Favorites使用的类型令牌（type token）是无限制的：getFavorite和putFavorite接受任何Class对象。有时候，可能需要限制那些可以传给方法的类型。这可以通过有限制的类型令牌（bounded type token）来实现，它只是一个类型令牌，利用有限制类型参数（见第27条）或者有限制通配符（见第28条），来限制可以表示的类型。

注解API（见第35条）广泛利用了有限制的类型令牌。例如，这是一个在运行时读取注解的方法。这个方法来自AnnotatedElement接口，它通过表示类、方法、域及其他程序元素的反射类型来实现：

```
public <T extends Annotation>
    T getAnnotation(Class<T> annotationType);
```

参数annotationType是一个表示注解类型的有限制的类型令牌。如果元素有这种类型的注解，该方法就将它返回，如果没有，则返回null。被注解的元素本质上是个类型安全的异构容器，容器的键属于注解类型。

假设你有一个类型Class<?>的对象，并且想将它传给一个需要有限制的类型令牌的方法，例如getAnnotation。你可以将对象转换成Class<? extends Annotation>，但是这种转换是非受检的，因此会产生一条编译时警告（见第24条）。幸运的是，类Class提供了一个安全（且动态）地执行这种转换的实例方法。该方法称作asSubclass，它将调用它的Class对象转换成用其参数表示的类的一个子类。如果转换成功，该方法返回它的参数；如果失败，则抛出ClassCastException异常。

以下示范了如何利用asSubclass方法在编译时读取类型未知的注解。这个方法编译时没有出现错误或者警告：

```
// Use of asSubclass to safely cast to a bounded type token
static Annotation getAnnotation(AnnotatedElement element,
                                String annotationTypeName) {
    Class<?> annotationType = null; // Unbounded type token
    try {
        annotationType = Class.forName(annotationTypeName);
    } catch (Exception ex) {
        throw new IllegalArgumentException(ex);
    }
    return element.getAnnotation(
        annotationType.asSubclass(Annotation.class));
}
```

总而言之，集合API说明了泛型的一般用法，限制你每个容器只能有固定数目的类型参数。你可以通过将类型参数放在键上而不是容器上来避开这一限制。对于这种类型安全的异构容器，可以用Class对象作为键。以这种方式使用的Class对象称作类型令牌。你也可以使用定制的键类型。例如，用一个DatabaseRow类型表示一个数据库行（容器），用泛型Column<T>作为它的键。

第6章

枚举和注解

Java 1.5发行版本中增加了两个新的引用类型家族：一种新的类称作*枚举类型*（enum type），一种新的接口称作*注解类型*（annotation type）。本章讨论使用这两个新的类型家族的最佳实践。

第30条：用enum代替int常量

枚举类型（**enum type**）是指由一组固定的常量组成合法值的类型，例如一年中的季节、太阳系中的行星或者一副牌中的花色。在编程语言中还没有引入枚举类型之前，表示枚举类型的常用模式是声明一组具名的int常量，每个类型成员一个常量：

```
// The int enum pattern - severely deficient!
public static final int APPLE_FUJI         = 0;
public static final int APPLE_PIPPIN       = 1;
public static final int APPLE_GRANNY_SMITH = 2;

public static final int ORANGE_NAVEL  = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD  = 2;
```

这种方法称作**int**枚举模式（int enum pattern），存在着诸多不足。它在类型安全性和使用方便性方面没有任何帮助。如果你将apple传到想要orange的方法中，编译器也不会出现警告，还会用==操作符将apple与orange进行对比，甚至更糟糕：

```
// Tasty citrus flavored applesauce!
int i = (APPLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN;
```

注意每个apple常量的名称都以APPLE_作为前缀，每个orange常量则都以ORANGE_作为前缀。这是因为Java没有为int枚举组提供命名空间。当两个int枚举组具有相同的命名常量时，前缀可以防止名称发生冲突。

采用int枚举模式的程序是十分脆弱的。因为int枚举是编译时常量，被编译到使用它们的客户端中。如果与枚举常量关联的int发生了变化，客户端就必须重新编译。如果没有重新编译，程序还是可以运行，但是它们的行为就是不确定的。

将int枚举常量翻译成可打印的字符串，并没有很便利的方法。如果将这种常量打印出来，或者从调试器中将它显示出来，你所见到的就是一个数字，这没有太大的用处。要遍历一个组中的所有int枚举常量，甚至获得int枚举组的大小，这些都没有很可靠的方法。

你还可能碰到这种模式的变体，在这种模式中使用的是String常量，而不是int常量。这样的变体被称作**String**枚举模式，同样也是我们最不期望的。虽然它为这些常量提供了可打印的字符串，但是它会导致性能问题，因为它依赖于字符串的比较操作。更糟糕的是，它会导致初级用户把字符串常量硬编码到客户端代码中，而不是使用适当的域（field）名。如果这样的硬编码字符串常量中包含有书写错误，那么，这样的错误在编译时不会被检测到，但是在运行的时候却会报错。

幸运的是，从Java1.5发行版本开始，就提出了另一种可以替代的解决方案，可以避免int和String枚举模式的缺点，并提供许多额外的好处。这就是（JLS，8.9）。下面以最简单的形式演示了这种模式：

```
public enum Apple  { FUJI, PIPPIN, GRANNY_SMITH }
public enum Orange { NAVEL, TEMPLE, BLOOD }
```

表面上看来，这些枚举类型与其他语言中的没有什么两样，例如C、C++和C#，但是实际上并非如此。Java的枚举类型是功能十分齐全的类，功能比其他语言中的对等物要更强大得多，Java的枚举本质上是int值。

Java枚举类型背后的基本想法非常简单：它们就是通过公有的静态final域为每个枚举常量导出实例的类。因为没有可以访问的构造器，枚举类型是真正的final。因为客户端既不能创建枚举类型的实例，也不能对它进行扩展，因此很可能没有实例，而只有声明过的枚举常量。换句话说，枚举类型是实例受控的。它们是单例（Singleton）的泛型化（见第3条），本质上是单元素的枚举。对于熟悉本书第一版的读者来说，枚举类型为类型安全的枚举（**typesafe enum**）模式[Bloch01，见第21条]提供了语言方面的支持。

枚举提供了编译时的类型安全。如果声明一个参数的类型为Apple，就可以保证，被传到该参数上的任何非null的对象引用一定属于三个有效的Apple值之一。试图传递类型错误的值时，会导致编译时错误，就像试图将某种枚举类型的表达式赋给另一种枚举类型的变量，或者试图利用==操作符比较不同枚举类型的值一样。

包含同名常量的多个枚举类型可以在一个系统中和平共处，因为每个类型都有自己的命名

空间。你可以增加或者重新排列枚举类型中的常量，而无需重新编译它的客户端代码，因为导出常量的域在枚举类型和它的客户端之间提供了一个隔离层：常量值并没有被编译到客户端代码中，而是在int枚举模式之中。最终，可以通过调用toString方法，将枚举转换成可打印的字符串。

除了完善了int枚举模式的不足之外，枚举类型还允许添加任意的方法和域，并实现任意的接口。它们提供了所有Object方法（见第3章）的高级实现，实现了Comparable（见第12条）和Serializable接口（见第11章），并针对权举类型的可任意改变性设计了序列化方式。

那么我们为什么要将方法或者域添加到枚举类型中呢？首先，你可能是想将数据与它的常量关联起来。例如，一个能够返回水果颜色或者返回水果图片的方法，对于我们的Apple和Orange类型来说可能很有好处。你可以利用任何适当的方法来增强枚举类型。枚举类型可以先作为枚举常量的一个简单集合，随着时间的推移再演变成为全功能的抽象。

举个有关枚举类型的好例子，比如太阳系中的8颗行星。每颗行星都有质量和半径，通过这两个属性可以计算出它的表面重力。从而给定物体的质量，就可以计算出一个物体在行星表面上的重量。下面就是这个枚举。每个枚举常量后面括号中的数值就是传递给构造器的参数。在这个例子中，它们就是行星的质量和半径：

```
// Enum type with data and behavior
public enum Planet {
    MERCURY(3.302e+23, 2.439e6),
    VENUS  (4.869e+24, 6.052e6),
    EARTH  (5.975e+24, 6.378e6),
    MARS   (6.419e+23, 3.393e6),
    JUPITER(1.899e+27, 7.149e7),
    SATURN (5.685e+26, 6.027e7),
    URANUS (8.683e+25, 2.556e7),
    NEPTUNE(1.024e+26, 2.477e7);
    private final double mass;           // In kilograms
    private final double radius;         // In meters
    private final double surfaceGravity; // In m / s^2

    // Universal gravitational constant in m^3 / kg s^2
    private static final double G = 6.67300E-11;

    // Constructor
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
        surfaceGravity = G * mass / (radius * radius);
    }

    public double mass()           { return mass; }
    public double radius()         { return radius; }
    public double surfaceGravity() { return surfaceGravity; }

    public double surfaceWeight(double mass) {
        return mass * surfaceGravity;  // F = ma
    }
}
```

编写一个像Planet这样的枚举类型并不难。为了将数据与枚举常量关联起来，得声明实例域，并编写一个带有数据并将数据保存在域中的构造器。枚举天生就是不可变的，因此所有的域都应该为final的（见第15条）。它们可以是公有的，但最好将它们做成是私有的，并提供公有的访问方法（见第14条）。在Planet这个示例中，构造器还计算和保存表面重力，但这正是一种优化。每当surfaceWeight方法用到重力时，都会根据质量和半径重新计算，并返回它在该常量所表示的行星上的重量。

虽然Planet枚举很简单，它的功能却强大得出奇。下面是一个简短的程序，根据某个物体在地球上的重量（以任何单位），打印出一张很棒的表格，显示出该物体在所有8颗行星上的重量（用相同的单位）：

```
public class WeightTable {
   public static void main(String[] args) {
      double earthWeight = Double.parseDouble(args[0]);
      double mass = earthWeight / Planet.EARTH.surfaceGravity();
      for (Planet p : Planet.values())
          System.out.printf("Weight on %s is %f%n",
                            p, p.surfaceWeight(mass));
   }
}
```

注意Planet就像所有的枚举一样，它有一个静态的values方法，按照声明顺序返回它的值数组。还要注意toString方法返回每个枚举值的声明名称，使得println和printf的打印变得更加容易。如果你不满意这种字符串表示法，可以通过覆盖toString方法对它进行修改。下面就是用命令行参数175运行这个小小的WeightTable程序时的结果：

```
Weight on MERCURY is 66.133672
Weight on VENUS is 158.383926
Weight on EARTH is 175.000000
Weight on MARS is 66.430699
Weight on JUPITER is 442.693902
Weight on SATURN is 186.464970
Weight on URANUS is 158.349709
Weight on NEPTUNE is 198.846116
```

如果这是你第一次在实践中见到Java的printf方法，要注意它与C语言的区别，你在这里用的是%n，在C中则用\n。

与枚举常量关联的有些行为，可能只需要用在定义了枚举的类或者包中。这种行为最好被实现成私有的或者包级私有的方法。于是，每个枚举常量都带有一组隐蔽的行为，这使得包含该枚举的类或者包在遇到这种常量时都可以做出适当的反应。就像其他的类一样，除非迫不得已要将枚举方法导出至它的客户端，否则都应该将它声明为私有的，如有必要，则声明为包级私有的（见第13条）。

如果一个枚举具有普遍适用性，它就应该成为一个顶层类（top-level class）；如果它只

是被用在一个特定的顶层类中，它就应该成为该顶层类的一个成员类（见第22条）。例如，java.math.RoundingMode枚举表示十进制小数的舍入模式（rounding mode）。这些舍入模式用于BigDecimal类，但是它们提供了一个非常有用的抽象，这种抽象本质上又不属于BigDecimal类。通过使RoundingMode变成一个顶层类，库的设计者鼓励任何需要舍入模式的程序员重用这个枚举，从而增强API之间的一致性。

Planet示例中所示的方法对于大多数枚举类型来说就足够了，但你有时候会需要更多的方法。每个Planet常量都关联了不同的数据，但你有时需要将本质上不同的行为（**behavior**）与每个常量关联起来。例如，假设你在编写一个枚举类型，来表示计算器的四大基本操作（即加减乘除），你想要提供一个方法来执行每个常量所表示的算术运算。有一种方法是通过启用枚举的值来实现：

```
// Enum type that switches on its own value - questionable
public enum Operation {
    PLUS, MINUS, TIMES, DIVIDE;

    // Do the arithmetic op represented by this constant
    double apply(double x, double y) {
        switch(this) {
            case PLUS:   return x + y;
            case MINUS:  return x - y;
            case TIMES:  return x * y;
            case DIVIDE: return x / y;
        }
        throw new AssertionError("Unknown op: " + this);
    }
}
```

这段代码可行，但是不太好看。如果没有throw语句，它就不能进行编译，虽然从技术角度来看代码的结束部分是可以执行到的，但是实际上是不可能执行到这行代码的[JLS，14.2.1]。更糟糕的是，这段代码很脆弱。如果你添加了新的枚举常量，却忘记给switch添加相应的条件，枚举仍然可以编译，但是当你试图运用新的运算时，就会运行失败。

幸运的是，有一种更好的方法可以将不同的行为与每个枚举常量关联起来：在枚举类型中声明一个抽象的apply方法，并在特定于常量的类主体（**constant-specific class body**）中，用具体的方法覆盖每个常量的抽象apply方法。这种方法被称作特定于常量的方法实现（**constant-specific method implementation**）：

```
// Enum type with constant-specific method implementations
public enum Operation {
    PLUS   { double apply(double x, double y){return x + y;} },
    MINUS  { double apply(double x, double y){return x - y;} },
    TIMES  { double apply(double x, double y){return x * y;} },
    DIVIDE { double apply(double x, double y){return x / y;} };

    abstract double apply(double x, double y);
}
```

如果给Operation的第二种版本添加新的常量，你就不可能会忘记提供apply方法，因为该方法就紧跟在每个常量声明之后。即使你真的忘记了，编译器也会提醒你，因为枚举类型中的抽象方法必须被它所有常量中的具体方法所覆盖。

特定于常量的方法实现可以与特定于常量的数据结合起来。例如，下面的Operation覆盖了toString来返回通常与该操作关联的符号：

```
// Enum type with constant-specific class bodies and data
public enum Operation {
    PLUS("+") {
        double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        double apply(double x, double y) { return x / y; }
    };
    private final String symbol;
    Operation(String symbol) { this.symbol = symbol; }
    @Override public String toString() { return symbol; }

    abstract double apply(double x, double y);
}
```

在有些情况下，在枚举中覆盖toString非常有用。例如，上述的toString实现使得打印算术表达式变得非常容易，如这段小程序所示：

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    for (Operation op : Operation.values())
        System.out.printf("%f %s %f = %f%n",
                          x, op, y, op.apply(x, y));
}
```

用2和4作为命令行参数运行这段程序，会输出：

```
2.000000 + 4.000000 = 6.000000
2.000000 - 4.000000 = -2.000000
2.000000 * 4.000000 = 8.000000
2.000000 / 4.000000 = 0.500000
```

枚举类型有一个自动产生的valueOf(String)方法，它将常量的名字转变成常量本身。如果在枚举类型中覆盖toString，要考虑编写一个fromString方法，将定制的字符串表示法变回相应的枚举。下列代码（适当地改变了类型名称）可以为任何枚举完成这一技巧，只要每个常量都有一个独特的字符串表示法：

```
// Implementing a fromString method on an enum type
private static final Map<String, Operation> stringToEnum
    = new HashMap<String, Operation>();
static { // Initialize map from constant name to enum constant
    for (Operation op : values())
        stringToEnum.put(op.toString(), op);
}
// Returns Operation for string, or null if string is invalid
public static Operation fromString(String symbol) {
    return stringToEnum.get(symbol);
}
```

注意，在常量被创建之后，Operation常量从静态代码块中被放入到了stringToEnum的map中。试图使每个常量都从自己的构造器将自身放入到map中，会导致编译时错误。这是好事，因为如果这是合法的，就会抛出NullPointerException异常。枚举构造器不可以访问枚举的静态域，除了编译时常量域之外。这一限制是有必要的，因为构造器运行的时候，这些静态域还没有被初始化。

特定于常量的方法实现有一个美中不足的地方，它们使得在枚举常量中共享代码变得更加困难了。例如，考虑用一个枚举表示薪资包中的工作天数。这个枚举有一个方法，根据给定某工人的基本工资（按小时）以及当天的工作时间，来计算他当天的报酬。在五个工作日中，超过正常八小时的工作时间都会产生加班工资；在双休日中，所有工作都产生加班工资。利用switch语句，很容易通过将多个case标签分别应用到两个代码片断中，来完成这一计算。为了简洁起见，这个示例中的代码使用了double，但是注意double并不是适合薪资应用程序（见第48条）的数据类型。

```
// Enum that switches on its value to share code - questionable
enum PayrollDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY;
    private static final int HOURS_PER_SHIFT = 8;
    double pay(double hoursWorked, double payRate) {
        double basePay = hoursWorked * payRate;

        double overtimePay;     // Calculate overtime pay
        switch(this) {
          case SATURDAY: case SUNDAY:
            overtimePay = hoursWorked * payRate / 2;
          default: // Weekdays
            overtimePay = hoursWorked <= HOURS_PER_SHIFT ?
              0 : (hoursWorked - HOURS_PER_SHIFT) * payRate / 2;
            break;
        }

        return basePay + overtimePay;
    }
}
```

不可否认，这段代码十分简洁，但是从维护的角度来看，它非常危险。假设将一个元素添加到该枚举中，或许是一个表示假期天数的特殊值，但是忘记给switch语句添加相应的case。

程序依然可以编译，但pay方法会悄悄地将假期的工资计算成与正常工作日的相同。

为了利用特定于常量的方法实现安全地执行工资计算，你可能必须重复计算每个常量的加班工资，或者将计算移到两个辅助方法中（一个用来计算工作日，一个用来计算双休日），并从每个常量调用相应的辅助方法。这任何一种方法都会产生相当数量的样板代码，结果降低了可读性，并增加了出错的机率。

通过用计算工作日加班工资的具体方法代替PayrollDay中抽象的overtimePay方法，可以减少样板代码。这样，就只有双休日必须覆盖该方法了。但是这样也有着与switch语句一样的不足：如果又增加了一天而没有覆盖overtimePay方法，就会悄悄地延续工作日的计算。

你真正想要的就是每当添加一个枚举常量时，就强制选择一种加班报酬策略。幸运的是，有一种很好的方法可以实现这一点。这种想法就是将加班工资计算移到一个私有的嵌套枚举中，将这个策略枚举（**strategy enum**）的实例传到PayrollDay枚举的构造器中。之后PayrollDay枚举将加班工资计算委托给策略枚举，PayrollDay中就不需要switch语句或者特定于常量的方法实现了。虽然这种模式没有switch语句那么简洁，但更加安全，也更加灵活：

```
// The strategy enum pattern
enum PayrollDay {
    MONDAY(PayType.WEEKDAY), TUESDAY(PayType.WEEKDAY),
    WEDNESDAY(PayType.WEEKDAY), THURSDAY(PayType.WEEKDAY),
    FRIDAY(PayType.WEEKDAY),
    SATURDAY(PayType.WEEKEND), SUNDAY(PayType.WEEKEND);

    private final PayType payType;
    PayrollDay(PayType payType) { this.payType = payType; }

    double pay(double hoursWorked, double payRate) {
        return payType.pay(hoursWorked, payRate);
    }
    // The strategy enum type
    private enum PayType {
        WEEKDAY {
            double overtimePay(double hours, double payRate) {
                return hours <= HOURS_PER_SHIFT ? 0 :
                  (hours - HOURS_PER_SHIFT) * payRate / 2;
            }
        },
        WEEKEND {
            double overtimePay(double hours, double payRate) {
                return hours * payRate / 2;
            }
        };
        private static final int HOURS_PER_SHIFT = 8;

        abstract double overtimePay(double hrs, double payRate);

        double pay(double hoursWorked, double payRate) {
            double basePay = hoursWorked * payRate;
            return basePay + overtimePay(hoursWorked, payRate);
        }
    }
}
```

如果枚举中的switch语句不是在枚举中实现特定于常量的行为的一种很好的选择，那么它们还有什么用处呢？枚举中的**switch**语句适合于给外部的枚举类型增加特定于常量的行为。例如，假设Operation枚举不受你的控制，你希望它有一个实例方法来返回每个运算的反运算。你可以用下列静态方法模拟这种效果：

```
// Switch on an enum to simulate a missing method
public static Operation inverse(Operation op) {
    switch(op) {
        case PLUS:   return Operation.MINUS;
        case MINUS:  return Operation.PLUS;
        case TIMES:  return Operation.DIVIDE;
        case DIVIDE: return Operation.TIMES;
        default:  throw new AssertionError("Unknown op: " + op);
    }
}
```

一般来说，枚举会优先使用comparable而非int常量。与int常量相比，枚举有个小小的性能缺点，即装载和初始化枚举时会有空间和时间的成本。除了受资源约束的设备，例如手机和烤面包机之外，在实践中不必太在意这个问题。

那么什么时候应该使用枚举呢？每当需要一组固定常量的时候。当然，这包括“天然的枚举类型”，例如行星、一周的天数以及棋子的数目等等。但它也包括你在编译时就知道其所有可能值的其他集合，例如菜单的选项、操作代码以及命令行标记等。枚举类型中的常量集并不一定要始终保持不变。专门设计枚举特性是考虑到枚举类型的二进制兼容演变。

总而言之，与int常量相比，枚举类型的优势是不言而喻的。枚举要易读得多，也更加安全，功能更加强大。许多枚举都不需要显式的构造器或者成员，但许多其他枚举则受益于“每个常量与属性的关联”以及“提供行为受这个属性影响的方法”。只有极少数的枚举受益于将多种行为与单个方法关联。在这种相对少见的情况下，特定于常量的方法要优先于启用自有值的枚举。如果多个枚举常量同时共享相同的行为，则考虑策略枚举。

第31条：用实例域代替序数

许多枚举天生就与一个单独的int值相关联。所有的枚举都有一个ordinal方法，它返回每个枚举常量在类型中的数字位置。你可以试着从序数中得到关联的int值：

```
// Abuse of ordinal to derive an associated value - DON'T DO THIS
public enum Ensemble {
    SOLO,   DUET,   TRIO, QUARTET, QUINTET,
    SEXTET, SEPTET, OCTET, NONET,  DECTET;

    public int numberOfMusicians() { return ordinal() + 1; }
}
```

虽然这个枚举不错，但是维护起来就像一场恶梦。如果常量进行重新排序，numberOfMusicians方法就会遭到破坏。如果要再添加一个与已经用过的int值关联的枚举常量，就没那么走运了。例如，给双四重奏（**double quartet**）添加一个常量，它就像个八重奏一样，是由8位演奏家组成，但是没有办法做到。

要是没有给所有这些int值添加常量，也无法给某个int值添加常量。例如，假设想要添加一个常量表示三四重奏（**triple quartet**），它由12位演奏家组成。对于由11位演奏家组成的合奏曲并没有标准的术语，因此只好给没有用过的int值（11）添加一个虚拟（dummy）常量。这么做顶多就是不太好看。如果有许多int值都是从未用过的，可就不切实际了。

幸运的是，有一种很简单的方法可以解决这些问题。*永远不要根据枚举的序数导出与它关联的值，而是要将它保存在一个实例域中：*

```
public enum Ensemble {
    SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5),
    SEXTET(6), SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),
    NONET(9), DECTET(10), TRIPLE_QUARTET(12);

    private final int numberOfMusicians;
    Ensemble(int size) { this.numberOfMusicians = size; }
    public int numberOfMusicians() { return numberOfMusicians; }
}
```

Enum规范中谈到ordinal时这么写道："大多数程序员都不需要这个方法。它是设计成用于像EnumSet和EnumMap这种基于枚举的通用数据结构的。"除非你在编写的是这种数据结构，否则最好完全避免使用ordinal方法。

第32条：用EnumSet代替位域

如果一个枚举类型的元素主要用在集合中，一般就使用int枚举模式（见第30条），将2的不同倍数赋予每个常量：

```
// Bit field enumeration constants - OBSOLETE!
public class Text {
    public static final int STYLE_BOLD          = 1 << 0;  // 1
    public static final int STYLE_ITALIC        = 1 << 1;  // 2
    public static final int STYLE_UNDERLINE     = 1 << 2;  // 4
    public static final int STYLE_STRIKETHROUGH = 1 << 3;  // 8

    // Parameter is bitwise OR of zero or more STYLE_ constants
    public void applyStyles(int styles) { ... }
}
```

这种表示法让你用OR位运算将几个常量合并到一个集合中，称作位域（**bit field**）：

```
text.applyStyles(STYLE_BOLD | STYLE_ITALIC);
```

位域表示法也允许利用位操作，有效地执行像union（联合）和intersection（交集）这样的集合操作。但位域有着int枚举常量的所有缺点，甚至更多。当位域以数字形式打印时，翻译位域比翻译简单的int枚举常量要困难得多。甚至，要遍历位域表示的所有元素也没有很容易的方法。

有些程序员优先使用枚举而非int常量，他们在需要传递多组常量集时，仍然倾向于使用位域。其实没有理由这么做，因为还有更好的替代方法。java.util包提供了EnumSet类来有效地表示从单个枚举类型中提取的多个值的多个集合。这个类实现Set接口，提供了丰富的功能、类型安全性，以及可以从任何其他Set实现中得到的互用性。但是在内部具体的实现上，每个EnumSet内容都表示为位矢量。如果底层的枚举类型有64个或者更少的元素——大多如此——整个EnumSet就是用单个long来表示，因此它的性能比得上位域的性能。批处理，如removeAll和retainAll，都是利用位算法来实现的，就像手工替位域实现得那样。但是可以避免手工位操作时容易出现的错误以及不太雅观的代码，因为EnumSet替你完成了这项艰巨的工作。

下面是前一个范例改成用枚举代替位域后的代码，它更加简短、更加清楚，也更加安全：

```
// EnumSet - a modern replacement for bit fields
public class Text {
    public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }

    // Any Set could be passed in, but EnumSet is clearly best
    public void applyStyles(Set<Style> styles) { ... }
}
```

下面是将EnumSet实例传递给applyStyles方法的客户端代码。EnumSet提供了丰富的静态工厂来轻松创建集合，其中一个如这个代码所示：

```
text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

注意applyStyles方法采用的是Set<Style>而非EnumSet<Style>。虽然看起来好像所有的客户端都可以将EnumSet传到这个方法，但是最好还是接受接口类型而非接受实现类型。这是考虑到可能会有特殊的客户端要传递一些其他的Set实现，并且没有什么明显的缺点。

总而言之，正是因为枚举类型要用在集合（Set）中，所以没有理由用位域来表示它。EnumSet类集位域的简洁和性能优势及第30条中所述的枚举类型的所有优点于一身。实际上EnumSet有个缺点，即截止Java 1.6发行版本，它都无法创建不可变的EnumSet，但是这一点很可能在即将出来的版本中得到修正。同时，可以用Collections.unmodifiableSet将EnumSet封装起来，但是简洁性和性能会受到影响。

第33条：用EnumMap代替序数索引

有时候，你可能会见到利用ordinal方法（见第31条）来索引数组的代码。例如下面这个过于简化的类，用来表示一种烹饪用的香草：

```
public class Herb {
    public enum Type { ANNUAL, PERENNIAL, BIENNIAL }

    private final String name;
    private final Type type;

    Herb(String name, Type type) {
        this.name = name;
        this.type = type;
    }

    @Override public String toString() {
        return name;
    }
}
```

现在假设有一个香草的数组，表示一座花园中的植物，你想要按照类型（一年生、多年生或者两年生植物）进行组织之后将这些植物列出来。如果要这么做的话，需要构建三个集合，每种类型一个，并且遍历整座花园，将每种香草放到相应的集合中。有些程序员会将这些集合放到一个按照类型的序数进行索引的数组中来实现这一点。

```
// Using ordinal() to index an array - DON'T DO THIS!
Herb[] garden = ... ;

Set<Herb>[] herbsByType =  // Indexed by Herb.Type.ordinal()
    (Set<Herb>[]) new Set[Herb.Type.values().length];
for (int i = 0; i < herbsByType.length; i++)
    herbsByType[i] = new HashSet<Herb>();

for (Herb h : garden)
    herbsByType[h.type.ordinal()].add(h);

// Print the results
for (int i = 0; i < herbsByType.length; i++) {
    System.out.printf("%s: %s%n",
                      Herb.Type.values()[i], herbsByType[i]);
}
```

这种方法的确可行，但是隐藏着许多问题。因为数组不能与泛型（见第25条）兼容，程序需要进行未受检的转换，并且不能正确无误地进行编译。因为数组不知道它的索引代表着什么，你必须手工标注（label）这些索引的输出。但是这种方法最严重的问题在于，当你访问一个按照枚举的序数进行索引的数组时，使用正确的int值就是你的职责了；int不能提供枚举的类型安全。你如果使用了错误的值，程序就会悄悄地完成错误的工作，或者幸运的话，会抛出ArrayIndexOutOfBoundException异常。

幸运的是，有一种更好的方法可以达到同样的效果。数组实际上充当着从枚举到值的映射，因此可能还要用到Map。更具体地说，有一种非常快速的Map实现专门用于枚举键，称作java.util.EnumMap。以下就是用EnumMap改写后的程序：

```
// Using an EnumMap to associate data with an enum
Map<Herb.Type, Set<Herb>> herbsByType =
    new EnumMap<Herb.Type, Set<Herb>>(Herb.Type.class);
for (Herb.Type t : Herb.Type.values())
    herbsByType.put(t, new HashSet<Herb>());
for (Herb h : garden)
    herbsByType.get(h.type).add(h);
System.out.println(herbsByType);
```

这段程序更简短、更清楚，也更加安全，运行速度方面可以与使用序数的程序相媲美。它没有不安全的转换；不必手工标注这些索引的输出，因为映射键知道如何将自身翻译成可打印字符串的枚举；计算数组索引时也不可能出错。EnumMap在运行速度方面之所以能与通过序数索引的数组相媲美，是因为EnuMap在内部使用了这种数组。但是它对程序员隐藏了这种实现细节，集Map的丰富功能和类型安全与数组的快速于一身。注意EnumMap构造器采用键类型的Class对象：这是一个有限制的类型令牌（bounded type token），它提供了运行时的泛型信息（见第29条）。

你还可能见到按照序数进行索引（两次）的数组的数组，该序数表示两个枚举值的映射。例如，下面这个程序就是使用这样一个数组将两个阶段映射到一个阶段过渡中（从液体到固体称作凝固，从液体到气体称作沸腾，诸如此类）。

```
// Using ordinal() to index array of arrays - DON'T DO THIS!
public enum Phase { SOLID, LIQUID, GAS;
    public enum Transition {
        MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;
        // Rows indexed by src-ordinal, cols by dst-ordinal
        private static final Transition[][] TRANSITIONS = {
            { null,    MELT,     SUBLIME },
            { FREEZE,  null,     BOIL    },
            { DEPOSIT, CONDENSE, null    }
        };

        // Returns the phase transition from one phase to another
        public static Transition from(Phase src, Phase dst) {
            return TRANSITIONS[src.ordinal()][dst.ordinal()];
        }
    }
}
```

这段程序可行，看起来也比较优雅，但是事实并非如此。就像上面那个比较简单的香草花园的示例一样，编译器无法知道序数和数组索引之间的关系。如果在过渡表中出了错，或者在修改Phase或者Phase.Transition枚举类型的时候忘记将它更新，程序就会在运行时失败。这种失败的形式可能为ArrayIndexOutOfBoundsException、NullPointerException或者（更糟糕的是）没有任何提示的错误行为。这张表的大小是阶段个数的平方，即使非null项的数量比较少。

同样，利用EnumMap依然可以做得更好一些。因为每个阶段过渡都是通过一对阶段枚举进行索引的，最好将这种关系表示为一个map，这个map的键是一个枚举（起始阶段），值为另一个map，这第二个map的键为第二个枚举（目标阶段），它的值为结果（阶段过渡），即形成了Map（起始阶段，Map（目标阶段，阶段过渡））这种形式。一个阶段过渡所关联的两个阶段，最好通过“数据与阶段过渡枚举之间的关联”来获取，之后用该阶段过渡枚举来初始化嵌套的EnumMap。

```
// Using a nested EnumMap to associate data with enum pairs
public enum Phase {
   SOLID, LIQUID, GAS;

   public enum Transition {
     MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
     BOIL(LIQUID, GAS),   CONDENSE(GAS, LIQUID),
     SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID);

     private final Phase src;
     private final Phase dst;

     Transition(Phase src, Phase dst) {
        this.src = src;
        this.dst = dst;
     }
     // Initialize the phase transition map
     private static final Map<Phase, Map<Phase,Transition>> m =
       new EnumMap<Phase, Map<Phase,Transition>>(Phase.class);
     static {
       for (Phase p : Phase.values())
         m.put(p,new EnumMap<Phase,Transition>(Phase.class));
       for (Transition trans : Transition.values())
         m.get(trans.src).put(trans.dst, trans);
     }

     public static Transition from(Phase src, Phase dst) {
       return m.get(src).get(dst);
     }
   }
}
```

初始化阶段过渡map的代码看起来可能有点复杂，但是还不算太糟糕。map的类型为Map<Phase, Map<Phase, Transition>>，表示是由键为源Phase（即第一个phase）、值为另一个map组成的Map，其中组成值的Map是由键值对目标Phase（即第二个Phase）、Transition组成的。静态初始化代码块中的第一个循环初始化了外部map，得到了三个空的内容map。代码块中的第二个循环利用每个状态过渡常量提供的起始信息和目标信息初始化了内部map。

现在假设想要给系统添加一个新的阶段：**plasma**（离子）或者电离气体。只有两个过渡与这个阶段关联：电离化，它将气体变成离子；以及消电离化，将离子变成气体。为了更新基于数组的程序，必须给Phase添加一种新常量，给Phase.Transition添加两种新常量，用一种新的16个元素的版本取代原来9个元素的数组的数组。如果给数组添加的元素过多或者过少，或者元素放置不妥当，可就麻烦了：程序可以编译，但是会在运行时失败。为了更新基于

EnumMap的版本，所要做的就是必须将PLASMA添加到Phase列表，并将IONIZE（GAS，PLASMA）和DEIONIZE（PLASMA，GAS）添加到Phase.Transition的列表中。程序会自行处理所有其他的事情，你几乎没有机会出错。从内部来看，Map的Map被实现成了数组的数组，因此在提升了清楚性、安全性和易维护性的同时，在空间或者时间上还几乎不用任何开销。

总而言之，最好不要用序数来索引数组，而要使用**EnumMap**。如果你所表示的这种关系是多维的，就使用EnumMap<...，EnumMap<...>>。应用程序的程序员在一般情况下都不使用Enum.ordinal，即使要用也很少，因此这是一种特殊情况（见第31条）。

第34条：用接口模拟可伸缩的枚举

就几乎所有方面来看，枚举类型都优越于本书第一版中所述的类型安全枚举模式[Bloch01]。从表面上看，有一个异常与可伸缩性有关，这个异常可能处在原来的模式中，却没有得到语言构造的支持。换句话说，使用这种模式，就有可能让一个枚举类型去扩展另一个枚举类型；利用这种语言特性，则不可能这么做。这绝非偶然。枚举的可伸缩性最后证明基本上都不是什么好点子。扩展类型的元素为基本类型的实例，基本类型的实例却不是扩展类型的元素，这样很是混乱。目前还没有很好的方法来枚举基本类型的所有元素及其扩展。最终，可伸缩性会导致设计和实现的许多方面变得复杂起来。

也就是说，对于可伸缩的枚举类型而言，至少有一种具有说服力的用例，这就是操作码（**operation code**），也称作**opcode**。操作码是指这样的枚举类型：它的元素表示在某种机器上的那些操作，例如第30条中的Operation类型，它表示一个简单的计算器中的某些函数。有时候，要尽可能地让API的用户提供它们自己的操作，这样可以有效地扩展API所提供的操作集。

幸运的是，有一种很好的方法可以利用枚举类型来实现这种效果。由于枚举类型可以通过给操作码类型和（属于接口的标准实现的）枚举定义接口，来实现任意接口，基本的想法就是利用这一事实。例如，以下是第30条中的Operation类型的扩展版本：

```
// Emulated extensible enum using an interface
public interface Operation {
    double apply(double x, double y);
}

public enum BasicOperation, implements Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };
    private final String symbol;
    BasicOperation(String symbol) {
        this.symbol = symbol;
    }
    @Override public String toString() {
        return symbol;
    }
}
```

虽然枚举类型（BasicOperation）不是可扩展的，但接口类型（Operation）则是可扩展的，

它是用来表示API中的操作的接口类型。你可以定义另一个枚举类型，它实现这个接口，并用这个新类型的实例代替基本类型。例如，假设你想要定义一个上述操作类型的扩展，由求幂（exponentiation）和求余（remainder）操作组成。你所要做的就是编写一个枚举类型，让它实现Operation接口：

```
// Emulated extension enum
public enum ExtendedOperation implements Operation {
    EXP("^") {
        public double apply(double x, double y) {
            return Math.pow(x, y);
        }
    },
    REMAINDER("%") {
        public double apply(double x, double y) {
            return x % y;
        }
    };

    private final String symbol;
    ExtendedOperation(String symbol) {
        this.symbol = symbol;
    }
    @Override public String toString() {
        return symbol;
    }
}
```

在可以使用基础操作的任何地方，都可以使用新的操作，只要API是被写成采用接口类型（Operation）而非实现（BasicOperation）。注意，在枚举中，不必像在不可扩展的枚举中所做的那样，利用特定于实例的方法实现来声明抽象的apply方法。这是因为抽象的方法（apply）是接口（Operation）的一部分。

不仅可以在任何需要“基本枚举”的地方单独传递一个“扩展枚举”的实例，而且除了那些基本类型的元素之外，还可以传递完整的扩展枚举类型，并使用它的元素。例如，通过下面这个测试程序，体验一下上面定义过的所有扩展过的操作：

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(ExtendedOperation.class, x, y);
}
private static <T extends Enum<T> & Operation> void test(
        Class<T> opSet, double x, double y) {
    for (Operation op : opSet.getEnumConstants())
        System.out.printf("%f %s %f = %f%n",
                          x, op, y, op.apply(x, y));
}
```

注意扩展过的操作类型的类的字面文字（ExtendedOperation.class）从main被传递给了test方法，来描述被扩展操作的集合。这个类的字面文字充当有限制的类型令牌（见第29条）。opSet参数中公认很复杂的声明（<T extends Enum<T> & Operation> Class<T>）确保了Class

对象既表示枚举又表示Operation的子类型，这正是遍历元素和执行与每个元素相关联的操作时所需要的。

第二种方法是使用Collection<? Extends Operation>，这是个有限制的通配符类型（**bounded wildcard type**）（见第28条），作为opSet参数的类型：

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(Arrays.asList(ExtendedOperation.values()), x, y);
}

private static void test(Collection<? extends Operation> opSet,
        double x, double y) {
    for (Operation op : opSet)
        System.out.printf("%f %s %f = %f%n",
                          x, op, y, op.apply(x, y));
}
```

这样得到的代码没有那么复杂，test方法也比较灵活一些：它允许调用者将多个实现类型的操作合并到一起。另一方面，也放弃了在指定操作上使用EnumSet（见第32条）和EnumMap（见第33条）的功能，因此，除非需要灵活地合并多个实现类型的操作，否则可能最好使用有限制的类型令牌。

上面这两段程序用命令行参数2和4运行时，都会产生这样的输出：

```
4.000000 ^ 2.000000 = 16.000000
4.000000 % 2.000000 = 0.000000
```

用接口模拟可伸缩枚举有个小小的不足，即无法将实现从一个枚举类型继承到另一个枚举类型。在上述Operation的示例中，保存和获取与某项操作相关联的符号的逻辑代码，可以复制到BasicOperation和ExtendedOperation中。在这个例子中是可以的，因为复制的代码非常少。如果共享功能比较多，则可以将它封装在一个辅助类或者静态辅助方法中，来避免代码的复制工作。

总而言之，虽然无法编写可扩展的枚举类型，却可以通过编写接口以及实现该接口的基础枚举类型，对它进行模拟。这样允许客户端编写自己的枚举来实现接口。如果API是根据接口编写的，那么在可以使用基础枚举类型的任何地方，也都可以使用这些枚举。

第35条：注解优先于命名模式

Java 1.5发行版本之前，一般使用*命名模式*（**naming pattern**）表明有些程序元素需要通过某种工具或者框架进行特殊处理。例如，JUnit测试框架原本要求它的用户一定要用test作为测试方法名称的开头[Beck04]。这种方法可行，但是有几个很严重的缺点。首先，文字拼写错误会导致失败，且没有任何提示。例如，假设不小心将一个测试方法命名为tsetSafetyOverride而不是testSafetyOverride。JUnit不会出错，但也不会执行测试，造成错误的安全感（即测试方法没有执行，它没有报错的可能，从而给人以测试正确的假象）。

命名模式的第二个缺点是，无法确保它们只用于相应的程序元素上。例如，假设将某个类称作testSafetyMechanisms，是希望JUnit会自动地测试它所有的方法，而不管它们叫什么名称。JUnit还是不会出错，但也同样不会执行测试。

命名模式的第三个缺点是，它们没有提供将参数值与程序元素关联起来的好方法。例如，假设想要支持一种测试类别，它只在抛出特殊异常时才会成功。异常类型本质上是测试的一个参数。你可以利用某种具体的命名模式，将异常类型名称编码到测试方法名称中，但是这样的代码会很不雅观，也很脆弱（见第50条）。编译器不知道要去检验准备命名异常的字符串是否真正命名成功。如果命名的类不存在，或者不是一个异常，你也要到试着运行测试时才会发现。

注解[JLS，9.7]很好地解决了所有这些问题。假设想要定义一个注解类型来指定简单的测试，它们自动运行，并在抛出异常时失败。以下就是这样的一个注解类型，命名为Test：

```
// Marker annotation type declaration
import java.lang.annotation.*;

/**
 * Indicates that the annotated method is a test method.
 * Use only on parameterless static methods.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

Test注解类型的声明就是它自身通过Retention和Target注解进行了注解。注解类型声明中的这种注解被称作*元注解*（**meta-annotation**）。@Retention(RetentionPolicy.RUNTIME)元注解表明，Test注解应该在运行时保留。如果没有保留，测试工具就无法知道Test注解。@Target(ElementType.METHOD)元注解表明，Test注解只在方法声明中才是合法的；它不能运用到类声明、域声明或者其他程序元素上。

注意Test注解声明上方的注释："Use only on parameterless static method（只用于无参的

静态方法)”。如果编译器能够强制这一限制最好，但是它做不到。编译器可以替你完成多少错误检查，这是有限制的，即使是利用注解。如果将Test注解放在实例方法的声明中，或者放在带有一个或者多个参数的方法中，测试程序还是可以编译，让测试工具在运行时来处理这个问题。

下面就是现实应用中的Test注解，称作标记注解（**marker annotation**），因为它没有参数，只是“标注”被注解的元素。如果程序员拼错了Test，或者将Test注解应用到程序元素而非方法声明，程序就无法编译：

```
// Program containing marker annotations
public class Sample {
    @Test public static void m1() { }  // Test should pass
    public static void m2() { }
    @Test public static void m3() {    // Test Should fail
        throw new RuntimeException("Boom");
    }
    public static void m4() { }
    @Test public void m5() { } // INVALID USE: nonstatic method
    public static void m6() { }
    @Test public static void m7() {    // Test should fail
        throw new RuntimeException("Crash");
    }
    public static void m8() { }
}
```

Sample类有8个静态方法，其中4个被注解为测试。这4个中有2个抛出了异常：m3和m7，另外两个则没有：m1和m5。但是其中一个没有抛出异常的被注解方法：m5，是一个实例方法，因此不属于注解的有效使用。总之，Sample包含4项测试：一项会通过，两项会失败，另一项无效。没有用Test注解进行标注的4个方法会被测试工具忽略。

Test注解对Sample类的语义没有直接的影响。它们只负责提供信息供相关的程序使用。更一般地讲，注解永远不会改变被注解代码的语义，但是使它可以通过工具进行特殊的处理，例如像这种简单的测试运行类：

```
// Program to process marker annotations
import java.lang.reflect.*;

public class RunTests {
    public static void main(String[] args) throws Exception {
        int tests = 0;
        int passed = 0;
        Class testClass = Class.forName(args[0]);
        for (Method m : testClass.getDeclaredMethods()) {
            if (m.isAnnotationPresent(Test.class)) {
                tests++;
                try {
                    m.invoke(null);
                    passed++;
                } catch (InvocationTargetException wrappedExc) {
                    Throwable exc = wrappedExc.getCause();
                    System.out.println(m + " failed: " + exc);
                } catch (Exception exc) {
```

```
                    System.out.println("INVALID @Test: " + m);
                }
            }
        }
        System.out.printf("Passed: %d, Failed: %d%n",
                          passed, tests - passed);
    }
}
```

测试运行工具在命令行上使用完全匹配的类名，并通过调用Method.invoke反射式地运行类中所有标注了Test的方法。isAnnotationPresent方法告知该工具要运行哪些方法。如果测试方法抛出异常，反射机制就会将它封装在InvocationTargetException中。该工具捕捉到了这个异常，并打印失败报告，包含测试方法抛出的原始异常，这些信息是通过getCause方法从InvocationTargetException中提取出来的。

如果尝试通过反射调用测试方法时抛出InvocationTargetException之外的任何异常，表明编译时没有捕捉到Test注解的无效用法。这种用法包括实例方法的注解，或者带有一个或者多个参数的方法的注解，或者不可访问的方法的注解。测试运行类中的第二个catch块捕捉到了这些Test用法错误，并打印出相应的错误消息。下面就是RunTests在Sample上运行时打印的输出：

```
public static void Sample.m3() failed: RuntimeException: Boom
INVALID @Test: public void Sample.m5()
public static void Sample.m7() failed: RuntimeException: Crash
Passed: 1, Failed: 3
```

现在我们要针对只在抛出特殊异常时才成功的测试添加支持。为此我们需要一个新的注解类型：

```
// Annotation type with a parameter
import java.lang.annotation.*;
/**
 * Indicates that the annotated method is a test method that
 * must throw the designated exception to succeed.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception> value();
}
```

这个注解的参数类型是Class<? extends Exception>。这个通配符类型无疑很绕口。它在英语中的意思是：某个扩展Exception的类的Class对象，它允许注解的用户指定任何异常类型。这种用法是有限制的类型令牌（见第29条）的一个示例。下面就是实际应用中的这个注解。注意类名称被用作了注解的参数值：

```
// Program containing annotations with a parameter
public class Sample2 {
    @ExceptionTest(ArithmeticException.class)
```

```
    public static void m1() {  // Test should pass
        int i = 0;
        i = i / i;
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m2() {  // Should fail (wrong exception)
        int[] a = new int[0];
        int i = a[1];
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m3() { }  // Should fail (no exception)
}
```

现在我们要修改一下测试运行工具来处理新的注解。这其中包括将以下代码添加到main方法中：

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
    tests++;
    try {
        m.invoke(null);
        System.out.printf("Test %s failed: no exception%n", m);
    } catch (InvocationTargetException wrappedEx) {
        Throwable exc = wrappedEx.getCause();
        Class<? extends Exception> excType =
            m.getAnnotation(ExceptionTest.class).value();
        if (excType.isInstance(exc)) {
            passed++;
        } else {
            System.out.printf(
                "Test %s failed: expected %s, got %s%n",
                m, excType.getName(), exc);
        }
    } catch (Exception exc) {
        System.out.println("INVALID @Test: " + m);
    }
}
```

这段代码类似于用来处理Test注解的代码，但有一处不同：这段代码提取了注解参数的值，并用它检验该测试抛出的异常是否为正确的类型。没有显式的转换，因此没有出现ClassCastException的危险。编译过的测试程序确保它的注解参数表示的是有效的异常类型，需要提醒一点：有可能注解参数在编译时是有效的，但是表示特定异常类型的类文件在运行时却不再存在。在这种希望很少出现的情况下，测试运行类会抛出TypeNotPresentException异常。

将上面的异常测试示例再深入一点，想像测试可以在抛出任何一种指定异常时都得到通过。注解机制有一种工具，使得支持这种用法变得十分容易。假设我们将ExceptionTest注解的参数类型改成Class对象的一个数组：

```
// Annotation type with an array parameter
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception>[] value();
}
```

注解中数组参数的语法十分灵活。它是进行过优化的单元素数组。使用了ExceptionTest新版的数组参数之后，之前的所有ExceptionTest注解仍然有效，并产生单元素的数组。为了指定多元素的数组，要用花括号（{}）将元素包围起来，并用逗号（,）将它们隔开：

```
// Code containing an annotation with an array parameter
@ExceptionTest({ IndexOutOfBoundsException.class,
                 NullPointerException.class })
public static void doublyBad() {
    List<String> list = new ArrayList<String>();

    // The spec permits this method to throw either
    // IndexOutOfBoundsException or NullPointerException
    list.addAll(5, null);
}
```

修改测试运行工具来处理新的ExceptionTest相当简单。下面的代码代替了原来的代码：

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
    tests++;
    try {
        m.invoke(null);
        System.out.printf("Test %s failed: no exception%n", m);
    } catch (Throwable wrappedExc) {
        Throwable exc = wrappedExc.getCause();
        Class<? extends Exception>[] excTypes =
            m.getAnnotation(ExceptionTest.class).value();
        int oldPassed = passed;
        for (Class<? extends Exception> excType : excTypes) {
            if (excType.isInstance(exc)) {
                passed++;
                break;
            }
        }
        if (passed == oldPassed)
            System.out.printf("Test %s failed: %s %n", m, exc);
    }
}
```

本条目中开发的测试框架只是一个试验，但它清楚地示范了注解之于命名模式的优越性。它这还只是揭开了注解功能的冰山一角。如果是在编写一个需要程序员给源文件添加信息的工具，就要定义一组适当的注解类型。既然有了注解，就完全没有理由再使用命名模式了。

也就是说，除了“工具铁匠（toolsmiths——特定的程序员）”之外，大多数程序员都不必定义注解类型。但是所有的程序员都应该使用**Java**平台所提供的预定义的注解类型（见第36和24条）。还要考虑使用IDE或者静态分析工具所提供的任何注解。这种注解可以提升由这些工具所提供的诊断信息的质量。但是要注意这些注解还没有标准化，因此如果变换工具或者形成标准，就有很多工作要做了。

第36条：坚持使用Override注解

随着Java 1.5发行版本中增加注解，类库中也增加了几种注解类型[JLS，9.6]。对于传统的程序员而言，这里面最重要的就是Override注解了。这个注解只能用在方法声明中，它表示被注解的方法声明覆盖了超类型中的一个声明。如果坚持使用这个注解，可以防止一大类的非法错误。考虑下面的程序，这里的类Bigram表示一个双字母组或者有序的字母对：

```
// Can you spot the bug?
public class Bigram {
    private final char first;
    private final char second;
    public Bigram(char first, char second) {
        this.first  = first;
        this.second = second;
    }
    public boolean equals(Bigram b) {
        return b.first == first && b.second == second;
    }
    public int hashCode() {
        return 31 * first + second;
    }

    public static void main(String[] args) {
        Set<Bigram> s = new HashSet<Bigram>();
        for (int i = 0; i < 10; i++)
            for (char ch = 'a'; ch <= 'z'; ch++)
                s.add(new Bigram(ch, ch));
        System.out.println(s.size());
    }
}
```

主程序反复地将26个双字母组添加到集合中，每个双字母组都由两个相同的小写字母组成。随后它打印出集合的大小。你可能以为程序打印出的大小为26，因为集合不能包含重复。如果你试着运行程序，会发现它打印的不是26而是260。哪里出错了呢？

很显然，Bigram类的创建者原本想要覆盖equals方法（见第8条），同时还记得覆盖了hashCode。遗憾的是，不幸的程序员没能覆盖equals，而是将它重载了（见第41条）。为了覆盖Object.equals，必须定义一个参数为Object类型的equals方法，但是Bigram的equals方法的参数并不是Object类型，因此Bigram从Object继承了equals方法。这个equals方法测试对象的同一性，就像==操作符一样。每个bigram的10个备份中，每一个都与其余的9个不同，因此Object.equals认为它们不相等，这正解释了程序为什么会打印出260的原因。

幸运的是，编译器可以帮助你发现这个错误，但是只有当你告知编译器你想要覆盖Object.equals时才行。为了做到这一点，要用@Override标注Bigram.euqals，如下所示：

```
@Override public boolean equals(Bigram b) {
    return b.first == first && b.second == second;
}
```

如果插入这个注解，并试着重新编译程序，编译器就会产生一条像这样的错误消息：

```
Bigram.java:10: method does not override or implement a method
from a supertype
    @Override public boolean equals(Bigram b) {
    ^
```

你会立即意识到哪里错了，拍拍自己的头，恍然大悟，马上用正确的来取代出错的equals实现（见第8条）：

```
@Override public boolean equals(Object o) {
    if (!(o instanceof Bigram))
        return false;
    Bigram b = (Bigram) o;
    return b.first == first && b.second == second;
}
```

因此，*应该在你想要覆盖超类声明的每个方法声明中使用***Override***注解*。这一规则有个小小的例外。如果你在编写一个没有标注为抽象的类，并且确信它覆盖了抽象的方法，在这种情况下，就不必将Override注解放在该方法上了。在没有声明为抽象的类中，如果没有覆盖抽象的超类方法，编译器就会发出一条错误消息。但是，你可能希望关注类中所有覆盖超类方法的方法，在这种情况下，也可以放心地标注这些方法。

现代的IDE提供了坚持使用Override注解的另一种理由。这种IDE具有自动检查功能，称作*代码检验*（**code inspection**）。如果启用相应的代码检验功能，当有一个方法没有Override注解，却覆盖了超类方法时，IDE就会产生一条警告。如果坚持使用Override注解，这些警告就会提醒你警惕无意识的覆盖。这些警告补充了编译器的错误消息，提醒你警惕无意识的覆盖失败。IDE和编译器，可以确保你覆盖任何你想要覆盖的方法，无一遗漏。

如果你使用的是Java 1.6或者更新的发行版本，Override注解在查找Bug方面还提供了更多的帮助。在Java 1.6发行版本中，在覆盖接口以及类的方法声明中使用Override注解变成是合法的了。在被声明为去实现某接口的具体类中，不必标注出你想要这些方法来覆盖接口方法，因为如果你的类没有实现每一个接口方法，编译器就会产生一条错误消息。当然，你可以选择只包括这些注解，来标明它们是接口方法，但是这并非绝对必要。

但是在抽象类或者接口中，还是值得标注所有你想要的方法，来覆盖超类或者超接口方法，无论是具体的还是抽象的。例如，Set接口没有给Collection接口添加新方法，因此它应该在它的所有方法声明中包括Override注解，以确保它不会意外地给Collection接口添加任何新方法。

总而言之，如果在你想要的每个方法声明中使用Override注解来覆盖超类声明，编译器就可以替你防止大量的错误，但有一个例外。在具体的类中，不必标注你确信覆盖了抽象方法声明的方法（虽然这么做也没有什么坏处）。

第37条：用标记接口定义类型

标记接口（**marker interface**）是没有包含方法声明的接口，而只是指明（或者“标明”）一个类实现了具有某种属性的接口。例如，考虑Serializable接口（见第11章）。通过实现这个接口，类表明它的实例可以被写到ObjectOutputStream（或者“被序列化”）。

你可能听说过标记注解（见第35条）使得标记接口过时了。这种断言是不正确的。标记接口有两点胜过标记注解。首先，也是最重要的一点是，*标记接口定义的类型是由被标记类的实例实现的；标记注解则没有定义这样的类型。*这个类型允许你在编译时捕捉在使用标记注解的情况下要到运行时才能捕捉到的错误。

就Serializable标记接口而言，如果它的参数没有实现该接口，ObjectOutputStream.write(Object)方法将会失败。令人不解的是，ObjectOutputStream API的创建者在声明write方法时并没有利用Serializable接口。该方法的参数类型应该为Serializable而非Object。因此，试着在没有实现Serializable的对象上调用ObjectOutputStream.write，只会在运行时失败，但也并不一定如此。

标记接口胜过标记注解的另一个优点是，它们可以被更加精确地进行锁定。如果注解类型利用@Target（ElementType.TYPE）声明，它就可以被应用到任何类或者接口。假设有一个标记只适用于特殊接口的实现。如果将它定义成一个标记接口，就可以用它将唯一的接口扩展成它适用的接口。

Set接口可以说就是这种*有限制的标记接口*（**restricted marker interface**）。它只适用于Collection子类型，但是它不会添加除了Collection定义之外的方法。一般情况下，不把它当作是标记接口，因为它改进了几个Collection方法的契约，包括add、equals和hashCode。但是很容易想像只适用于某种特殊接口的子类型的标记接口，它没有改进接口的任何方法的契约。这种标记接口可以描述整个对象的某个约束条件，或者表明实例能够利用其他某个类的方法进行处理（就像Serializable接口表明实例可以通过ObjectOutputStream进行处理一样）。

标记注解胜过标记接口的最大优点在于，它可以通过默认的方式添加一个或者多个注解类型元素，给已被使用的注解类型添加更多的信息[JLS，9.6]。随着时间的推移，简单的标记注解类型可以演变成更加丰富的注解类型。这种演变对于标记接口而言则是不可能的，因为它通常不可能在实现接口之后再给它添加方法（见第18条）。

标记注解的另一个优点在于，它们是更大的注解机制的一部分。因此，标记注解在那些支持注解作为编程元素之一的框架中同样具有一致性。

那么什么时候应该使用标记注解，什么时候应该使用标记接口呢？很显然，如果标记是应用到任何程序元素而不是类或者接口，就必须使用注解，因为只有类和接口可以用来实现或者扩展接口。如果标记只应用给类和接口，就要问问自己：我要编写一个还是多个只接受有这种标记的方法呢？如果是这种情况，就应该优先使用标记接口而非注解。这样你就可以用接口作为相关方法的参数类型，它真正可以为你提供编译时进行类型检查的好处。

如果你对第一个问题的回答是否定的，就要再问问自己：我要永远限制这个标记只用于特殊接口的元素吗？如果是，最好将标记定义成该接口的一个子接口。如果这两个问题的答案都是否定的，或许就应该使用标记注解。

总而言之，标记接口和标记注解都各有用处。如果想要定义一个任何新方法都不会与之关联的类型，标记接口就是最好的选择。如果想要标记程序元素而非类和接口，考虑到未来可能要给标记添加更多的信息，或者标记要适合于已经广泛使用了注解类型的框架，那么标记注解就是正确的选择。如果你发现自己在编写的是目标为**ElementType.TYPE**的标记注解类型，就要花点时间考虑清楚，它是否真的应该为注解类型，想想标记接口是否会更加合适呢。

从某种意义上说，本条目与第19条中“如果不想定义类型就不要使用接口”的说法相反。本条目最接近的意思是说：如果想要定义类型，一定要使用接口。

第7章

方　法

本章要讨论方法设计的几个方面：如何处理参数和返回值，如何设计方法签名，如何为方法编写文档。本章中大多数内容既适用于构造器，也适用于普通的方法。与第5章一样，本章的焦点也集中在可用性、健壮性和灵活性上。

第38条：检查参数的有效性

绝大多数方法和构造器对于传递给它们的参数值都会有某些限制。例如，索引值必须是非负数，对象引用不能为null，等等，这些都是很常见的。你应该在文档中清楚地指明所有这些限制，并且在方法体的开头处检查参数，以强制施加这些限制。这是“应该在发生错误之后尽快检测出错误”这一普遍原则的一个具体情形。如果不能做到这一点，检测到错误的可能性就比较小，即使检测到错误了，也比较难以确定错误的根源。

如果传递无效的参数值给方法，这个方法在执行之前先对参数进行了检查，那么它很快就会失败，并且清楚地出现适当的异常（exception）。如果这个方法没有检查它的参数，就有可能发生几种情形。该方法可能在处理过程中失败，并且产生令人费解的异常。更糟糕的是，该方法可以正常返回，但是会悄悄地计算出错误的结果。最糟糕的是，该方法可以正常返回，但是却使得某个对象处于被破坏的状态，将来在某个不确定的时候，在某个不相关的点上会引发错误。

对于公有的方法，要用Javadoc的@throws标签（tag）在文档中说明违反参数值限制时会抛出的异常（见第62条）。这样的异常通常为IllegalArgumentException、IndexOutOfBoundsException或NullPointerException（见第60条）。一旦在文档中记录了对于方法参数的限制，并且记录了一旦违反这些限制将要抛出的异常，强加这些限制就是非常简单的事情了。下面是一个典型的例子：

```
/**
 * Returns a BigInteger whose value is (this mod m).  This method
 * differs from the remainder method in that it always returns a
 * non-negative BigInteger.
 *
 * @param  m the modulus, which must be positive
 * @return this mod m
 * @throws ArithmeticException if m is less than or equal to 0
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException("Modulus <= 0: " + m);
    ... // Do the computation
}
```

对于未被导出的方法（unexported method），作为包的创建者，你可以控制这个方法将在哪些情况下被调用，因此你可以，也应该确保只将有效的参数值传递进来。因此，非公有的方法通常应该使用断言（**assertion**）来检查它们的参数，具体做法如下所示：

```
// Private helper function for a recursive sort
private static void sort(long a[], int offset, int length) {
    assert a != null;
    assert offset >= 0 && offset <= a.length;
    assert length >= 0 && length <= a.length - offset;
    ... // Do the computation
}
```

从本质上讲，这些断言是在声称被断言的条件将会为真，无论外围包的客户端如何使用它。不同于一般的有效性检查，断言如果失败，将会抛出AssertionError。也不同于一般的有效性检查，如果它们没有起到作用，本质上也不会有成本开销，除非通过将-ea（或者-enableassertions）标记（flag）传递给Java解释器，来启用它们。关于断言的更多信息，请见Sun的教程[Asserts]。

对于有些参数，方法本身没有用到，却被保存起来供以后使用，检验这类参数的有效性尤为重要。例如，考虑第83页中的静态工厂方法，它的参数为一个int数组，并返回该数组的List视图。如果这个方法的客户端要传递null，该方法将会抛出一个NullPointerException，因为该方法包含一个显式的条件检查。如果省略了这个条件检查，它就会返回一个指向新建List实例的引用，一旦客户端企图使用这个引用，立即就会抛出NullPointerException。到那时，要想找到List实例的来源可能就非常困难了，从而使得调试工作极大地复杂化了。

如前所述，有些参数被方法保存起来供以后使用，构造器正是代表了这种原则的一种特殊情形。检查构造器参数的有效性是非常重要的，这样可以避免构造出来的对象违反了这个类的约束条件。

在方法执行它的计算任务之前，应该先检查它的参数，这一规则也有例外。一个很重要的例外是，在有些情况下，有效性检查工作非常昂贵，或者根本是不切实际的，而且有效性检查已隐含在计算过程中完成。例如，考虑一个为对象列表排序的方法：Collections.sort (List)。列表中的所有对象都必须是可以相互比较的。在为列表排序的过程中，列表中的每个对象将

与其他某个对象进行比较。如果这些对象不能相互比较，其中的某个比较操作就会抛出ClassCastException，这正是sort方法所应该做的事情。因此，提前检查列表中的元素是否可以相互比较，这并没有多大意义。然而，请注意，不加选择地使用这种方法将会导致失去失败原子性（failure atomicity）（见第64条）。

有时候，某些计算会隐式地执行必要的有效性检查，但是如果检查不成功，就会抛出错误的异常。换句话说，由于无效的参数值而导致计算过程抛出的异常，与文档中标明这个方法将抛出的异常并不相符。在这种情况下，应该使用第61条中讲述的异常转译（**exception translation**）技术，将计算过程中抛出的异常转换为正确的异常。

不要从本条目的内容中得出这样的结论：对参数的任何限制都是件好事。相反，在设计方法时，应该使它们尽可能地通用，并符合实际的需要。假如方法对于它能接受的所有参数值都能够完成合理的工作，对参数的限制就应该是越少越好。然而，通常情况下，有些限制对于被实现的抽象来说是固有的。

简而言之，每当编写方法或者构造器的时候，应该考虑它的参数有哪些限制。应该把这些限制写到文档中，并且在这个方法体的开头处，通过显式的检查来实施这些限制。养成这样的习惯是非常重要的。只要有效性检查有一次失败，你为必要的有效性检查所付出的努力便都可以连本带利地得到偿还了。

第39条：必要时进行保护性拷贝

使Java使用起来如此舒适的一个因素在于，它是一门安全的语言（**safe language**）。这意味着，它对于缓冲区溢出、数组越界、非法指针以及其他的内存破坏错误都自动免疫，而这些错误却困扰着诸如C和C++这样的不安全语言。在一门安全语言中，在设计类的时候，可以确切地知道，无论系统的其他部分发生什么事情，这些类的约束都可以保持为真。对于那些“把所有内存当作一个巨大的数组来看待”的语言来说，这是不可能的。

即使在安全的语言中，如果不采取一点措施，还是无法与其他的类隔离开来。假设类的客户端会尽其所能来破坏这个类的约束条件，因此你必须保护性地设计程序。实际上，只有当有人试图破坏系统的安全性时，才可能发生这种情形；更有可能的是，对你的API产生误解的程序员，所导致的各种不可预期的行为，只好由类来处理。无论是哪种情况，编写一些面对客户的不良行为时仍能保持健壮性的类，这是非常值得投入时间去做的事情。

没有对象的帮助时，虽然另一个类不可能修改对象的内部状态，但是对象很容易在无意识的情况下提供这种帮助。例如，考虑下面的类，它声称可以表示一段不可变的时间周期：

```
// Broken "immutable" time period class
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param  start the beginning of the period
     * @param  end the end of the period; must not precede start
     * @throws IllegalArgumentException if start is after end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0)
            throw new IllegalArgumentException(
                start + " after " + end);
        this.start = start;
        this.end   = end;
    }

    public Date start() {
        return start;
    }
    public Date end() {
        return end;
    }

    ...  // Remainder omitted
}
```

乍一看，这个类似乎是不可变的，并且强加了约束条件：周期的起始时间（start）不能在结束时间（end）之后。然而，因为Date类本身是可变的，因此很容易违反这个约束条件：

```
// Attack the internals of a Period instance
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
end.setYear(78);  // Modifies internals of p!
```

为了保护Period实例的内部信息避免受到这种攻击，对于构造器的每个可变参数进行保护性拷贝（**defensive copy**）是必要的，并且使用备份对象作为Period实例的组件，而不使用原始的对象：

```
// Repaired constructor - makes defensive copies of parameters
public Period(Date start, Date end) {
    this.start = new Date(start.getTime());
    this.end   = new Date(end.getTime());

    if (this.start.compareTo(this.end) > 0)
      throw new IllegalArgumentException(start +" after "+ end);
}
```

用了新的构造器之后，上述的攻击对于Period实例不再有效。注意，保护性拷贝是在检查参数的有效性（见第**38**条）之前进行的，并且有效性检查是针对拷贝之后的对象，而不是针对原始的对象。虽然这样做看起来有点不太自然，却是必要的。这样做可以避免在“危险阶段（window of vulnerability）”期间从另一个线程改变类的参数，这里的危险阶段是指从检查参数开始，直到拷贝参数之间的时间段。（在计算机安全社区中，这被称作**Time-Of-Check/Time-Of-Use**或者**TOCTOU**攻击[Viega01]。）

同时也请注意，我们没有用Date的clone方法来进行保护性拷贝。因为Date是非final的，不能保证clone方法一定返回类为java.util.Date的对象：它有可能返回专门出于恶意的目的而设计的不可信子类的实例。例如，这样的子类可以在每个实例被创建的时候，把指向该实例的引用记录到一个私有的静态列表中，并且允许攻击者访问这个列表。这将使得攻击者可以自由地控制所有的实例。为了阻止这种攻击，对于参数类型可以被不可信任方子类化的参数，请不要使用**clone**方法进行保护性拷贝。

虽然替换构造器就可以成功地避免上述的攻击，但是改变Period实例仍然是有可能的，因为它的访问方法提供了对其可变内部成员的访问能力：

```
// Second attack on the internals of a Period instance
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
p.end().setYear(78);  // Modifies internals of p!
```

为了防御这第二种攻击，只需修改这两个访问方法，使它返回可变内部域的保护性拷贝即可：

```
// Repaired accessors - make defensive copies of internal fields
```

```
public Date start() {
    return new Date(start.getTime());
}

public Date end() {
    return new Date(end.getTime());
}
```

采用了新的构造器和新的访问方法之后，Period真正是不可变的了。不管程序员是多么恶意，或者多么不合格，都绝对不会违反“周期的起始时间不能落后于结束时间”这个约束条件。确实如此，因为除了Period类自身之外，其他任何类都无法访问Period实例中的任何一个可变域。这些域被真正封装在对象的内部。

访问方法与构造器不同，它们在进行保护性拷贝的时候允许使用clone方法。之所以如此，是因为我们知道，Period内部的Date对象的类是java.util.Date，而不可能是其他某个潜在的不可信子类。也就是说，基于第11条中所阐述的原因，一般情况下，最好使用构造器或者静态工厂。

参数的保护性拷贝并不仅仅针对不可变类。每当编写方法或者构造器时，如果它要允许客户提供的对象进入到内部数据结构中，则有必要考虑一下，客户提供的对象是否有可能是可变的。如果是，就要考虑你的类是否能够容忍对象进入数据结构之后发生变化。如果答案是否定的，就必须对该对象进行保护性拷贝，并且让拷贝之后的对象而不是原始对象进入到数据结构中。例如，如果你正在考虑使用由客户提供的对象引用作为内部Set实例的元素，或者作为内部Map实例的键（key），就应该意识到，如果这个对象在插入之后再被修改，Set或者Map的约束条件就会遭到破坏。

在内部组件被返回给客户端之前，对它们进行保护性拷贝也是同样的道理。不管类是否为不可变的，在把一个指向内部可变组件的引用返回给客户端之前，也应该加倍认真地考虑。解决方案是，应该返回保护性拷贝。记住长度非零的数组总是可变的。因此，在把内部数组返回给客户端之前，应该总要进行保护性拷贝。另一种解决方案是，给客户端返回该数组的不可变视图（immutable view）。这两种方法在第13条中都已经演示过了。

可以肯定地说，上述的真正启示在于，只要有可能，都应该使用不可变的对象作为对象内部的组件，这样就不必再为保护性拷贝（见第15条）操心。在前面的Period例子中，值得一提的是，有经验的程序员通常使用Date.getTime()返回的long基本类型作为内部的时间表示法，而不是使用Date对象引用。他们之所以这样做，主要因为Date是可变的。

保护性拷贝可能会带来相关的性能损失，这种说法并不总是正确的。如果类信任它的调用者不会修改内部的组件，可能因为类及其客户端都是同一个包的双方，那么不进行保护性拷贝也是可以的。在这种情况下，类的文档中就必须清楚地说明，调用者绝不能修改受到影响的参数或者返回值。

即使跨越包的作用范围，也并不总是适合在将可变参数整合到对象中之前，对它进行保护性拷贝。有一些方法和构造器的调用，要求参数所引用的对象必须有个显式的交接（**handoff**）过程。当客户端调用这样的方法时，它承诺以后不再直接修改该对象。如果方法或者构造器期望接管一个由客户端提供的可变对象，它就必须在文档中明确地指明这一点。

如果类所包含的方法或者构造器的调用需要移交对象的控制权，这个类就无法让自身抵御恶意的客户端。只有当类和它的客户端之间有着互相的信任，或者破坏类的约束条件不会伤害到除了客户端之外的其他对象时，这种类才是可以接受的。后一种情形的例子是包装类模式（wrapper class pattern）（见第16条）。根据包装类的本质特征，客户端只需在对象被包装之后直接访问它，就可以破坏包装类的约束条件，但是，这么做往往只会伤害到客户端自己。

简而言之，如果类具有从客户端得到或者返回到客户端的可变组件，类就必须保护性地拷贝这些组件。如果拷贝的成本受到限制，并且类信任它的客户端不会不恰当地修改组件，就可以在文档中指明客户端的职责是不得修改受到影响的组件，以此来代替保护性拷贝。

第40条：谨慎设计方法签名

本条目是若干API设计技巧的总结，它们都还不足以单独开设一个条目。综合来说，这些设计技巧将有助于使你的API更易于学习和使用，并且比较不容易出错。

谨慎地选择方法的名称。方法的名称应该始终遵循标准的命名习惯（见第56条）。首要目标应该是选择易于理解的，并且与同一个包中的其他名称风格一致的名称。第二个目标应该是选择与大众认可的名称（如果存在的话）相一致的名称。如果还有疑问，请参考Java类库的API。尽管Java类库的API中也有大量不一致的地方，考虑到这些Java类库的规模和范围，这是不可避免的，但它还是得到了相当程度的认可。

不要过于追求提供便利的方法。每个方法都应该尽其所能。方法太多会使类难以学习、使用、文档化、测试和维护。对于接口而言，这无疑是正确的，方法太多会使接口实现者和接口用户的工作变得复杂起来。对于类和接口所支持的每个动作，都提供一个功能齐全的方法。只有当一项操作被经常用到的时候，才考虑为它提供快捷方式（shorthand）。*如果不能确定，还是不提供快捷为好。*

避免过长的参数列表。目标是四个参数，或者更少。大多数程序员都无法记住更长的参数列表。如果你编写的许多方法都超过了这个限制，你的API就不太便于使用，除非用户不停地参考它的文档。现代的IDE会有所帮助，但最好还是使用简短的参数列表。*相同类型的长参数序列格外有害。*API的用户不仅无法记住参数的顺序，而且，当他们不小心弄错了参数顺序时，他们的程序仍然可以编译和运行，只不过这些程序不会按照作者的意图进行工作。

有三种方法可以缩短过长的参数列表。第一种是把方法分解成多个方法，每个方法只需要这些参数的一个子集。如果不小心，这样做会导致方法过多。但是通过提升它们的正交性（orthogonality），还可以*减少*（**reduce**）方法的数目。例如，考虑java.util.List接口。它并没有提供“在子列表（sublist）中查找元素的第一个索引和最后一个索引”的方法，这两个方法都需要三个参数。相反，它提供了subList方法，这个方法带有两个参数，并返回子列表的一个*视图*（**view**）。这个方法可以与indexOf或者lastIndexOf方法结合起来，获得期望的功能，而这两个方法都分别只有一个参数。而且，subList方法也可以与其他任何“针对List实例进行操作”的方法结合起来，在子列表上执行任意的计算。这样得到的API就有很高的“功能－重量”（power-to-weight）比。

缩短长参数列表的第二种方法是创建*辅助类*（**helper class**），用来保存参数的分组。这些辅助类一般为静态成员类（见第22条）。如果一个频繁出现的参数序列可以被看作是代表了某个独特的实体，则建议使用这种方法。例如，假设你正在编写一个表示纸牌游戏的类，你会

发现，经常要传递一个两参数的序列来表示纸牌的点数和花色。如果增加辅助类来表示一张纸牌，并且把每个参数序列都换成这个辅助类的单个参数，那么这个纸牌游戏类的API以及它的内部表示都可能会得到改进。

结合了前两种方法特征的第三种方法是，从对象构建到方法调用都采用Builder模式（请见第2条）。如果方法带有多个参数，尤其是当它们中有些是可选的时候，最好定义一个对象来表示所有参数，并允许客户端在这个对象上进行多次“setter”调用，每次调用都设置一个参数，或者设置一个较小的相关的集合。一旦设置了需要的参数，客户端就调用对象的“执行（execute）”方法，它对参数进行最终的有效性检查，并执行实际的计算。

对于参数类型，要优先使用接口而不是类（请见第52条）。只要有适当的接口可用来定义参数，就优先使用这个接口，而不是使用实现该接口的类。例如，没有理由在编写方法时使用HashMap类来作为输入，相反，应当使用Map接口作为参数。这使你可以传入一个Hashtable、HashMap、TreeMap、TreeMap的子映射表（submap），或者任何有待于将来编写的Map实现。如果使用的是类而不是接口，则限制了客户端只能传入特定的实现，如果碰巧输入的数据是以其他的形式存在，就会导致不必要的、可能非常昂贵的拷贝操作。

*对于***boolean***参数，要优先使用两个元素的枚举类型。*它使代码更易于阅读和编写，尤其当你在使用支持自动完成功能的IDE的时候。它也使以后更易于添加更多的选项。例如，你可能会有一个Thermometer类型，它带有一个静态工厂方法，而这个静态工厂方法的签名需要传入这个枚举的值：

```
public enum TemperatureScale { FAHRENHEIT, CELSIUS }
```

Thermometer.newInstance(TemperatureScale.CELSIUS)不仅比Thermometer.newInstance(true)更有用，而且你还可以在未来的发行版本中将KELVIN添加到TemperatureScale中，无需非得给Thermometer添加新的静态工厂。你还可以将依赖于温度刻度单位的代码重构到枚举常量的方法中（见第30条）。例如，每个刻度单位都可以有一个方法，它带有一个double值，并将它规格化成摄氏度。

第41条：慎用重载

下面这个程序的意图是好的，它试图根据一个集合（collection）是Set、List，还是其他的集合类型，来对它进行分类：

```
// Broken! - What does this program print?
public class CollectionClassifier {
    public static String classify(Set<?> s) {
        return "Set";
    }

    public static String classify(List<?> lst) {
        return "List";
    }

    public static String classify(Collection<?> c) {
        return "Unknown Collection";
    }

    public static void main(String[] args) {
        Collection<?>[] collections = {
            new HashSet<String>(),
            new ArrayList<BigInteger>(),
            new HashMap<String, String>().values()
        };

        for (Collection<?> c : collections)
            System.out.println(classify(c));
    }
}
```

你可能期望这个程序会打印出“Set”，紧接着是“List”，以及“Unknown Collection”，但实际上不是这样。它是打印“Unknown Collection”三次。为什么会这样呢？因为classify方法被重载（**overloaded**）了，而要调用哪个重载（**overloading**）方法是在编译时做出决定的。对于for循环中的全部三次迭代，参数的编译时类型都是相同的：Collection<?>。每次迭代的运行时类型都是不同的，但这并不影响对重载方法的选择。因为该参数的编译时类型为Collection<?>，所以，唯一合适的重载方法是第三个：classify(Collection<?>)，在循环的每次迭代中，都会调用这个重载方法。

这个程序的行为有悖常理，因为对于重载方法（**overloaded method**）的选择是静态的，而对于被覆盖的方法（**overridden method**）的选择则是动态的。选择被覆盖的方法的正确版本是在运行时进行的，选择的依据是被调用方法所在对象的运行时类型。这里重新说明一下，当一个子类包含的方法声明与其祖先类中的方法声明具有同样的签名时，方法就被覆盖了。如果实例方法在子类中被覆盖了，并且这个方法是在该子类的实例上被调用的，那么子类中的覆盖方法（**overriding method**）将会执行，而不管该子类实例的编译时类型到底是什么。为了进行更具体的说明，考虑下面这个程序：

```
class Wine {
    String name() { return "wine"; }
}

class SparklingWine extends Wine {
    @Override String name() { return "sparkling wine"; }
}

class Champagne extends SparklingWine {
    @Override String name() { return "champagne"; }
}

public class Overriding {
    public static void main(String[] args) {
        Wine[] wines = {
            new Wine(), new SparklingWine(), new Champagne()
        };
        for (Wine wine : wines)
            System.out.println(wine.name());
    }
}
```

name方法是在类Wine中被声明的，但是在类SparklingWine和Champagne中被覆盖。正如你所预期的那样，这个程序打印出“wine，sparkling wine和champagne”，尽管在循环的每次迭代中，实例的编译时类型都为Wine。当调用被覆盖的方法时，对象的编译时类型不会影响到哪个方法将被执行；“最为具体的（most specific）”那个覆盖版本总是会得到执行。这与重载的情形相比，对象的运行时类型并不影响“哪个重载版本将被执行”；选择工作是在编译时进行的，完全基于参数的编译时类型。

在CollectionClassifier这个示例中，该程序的意图是：期望编译器根据参数的运行时类型自动将调用分发给适当的重载方法，以此来识别出参数的类型，就好像Wine的例子中的name方法所做的那样。方法重载机制完全没有提供这样的功能。假设需要有个静态方法，这个程序的最佳修正方案是，用单个方法来替换这三个重载的classify方法，并在这个方法中做一个显式的instanceof测试：

```
public static String classify(Collection<?> c) {
    return c instanceof Set  ? "Set" :
           c instanceof List ? "List" : "Unknown Collection";
}
```

因为覆盖机制是规范，而重载机制是例外，所以，覆盖机制满足了人们对于方法调用行为的期望。正如CollectionClassifier例子所示，重载机制很容易使这些期望落空。如果编写出来的代码的行为可能使程序员感到困惑，它就是很糟糕的实践。对于API来说尤其如此。如果API的普通用户根本不知道“对于一组给定的参数，其中的哪个重载方法将会被调用”，那么，使用这样的API就很可能导致错误。这些错误要等到运行时发生了怪异的行为之后才会显现出来，许多程序员无法诊断出这样的错误。因此，应该避免胡乱地使用重载机制。

到底怎样才算胡乱使用重载机制呢？这个问题仍有争议。安全而保守的策略是，永远不要

导出两个具有相同参数数目的重载方法。如果方法使用可变参数（**varargs**），保守的策略是根本不要重载它，除第42条中所述的情形之外。如果你遵守这些限制，程序员永远也不会陷入到“对于任何一组实际的参数，哪个重载方法是适用的”这样的疑问中。这项限制并不麻烦，因为你始终可以给方法起不同的名称，而不使用重载机制。

例如，考虑ObjectOutputStream类。对于每个基本类型，以及几种引用类型，它的write方法都有一种变形。这些变形方法并不是重载write方法，而是具有诸如writeBoolean(boolean)、writeInt(int)和writeLong(long)这样的签名。与重载方案相比较，这种命名模式带来的好处是，有可能提供相应名称的读方法，比如readBoolean()、readInt()和readLong()。实际上，ObjectInputStream类正是提供了这样的读方法。

对于构造器，你没有选择使用不同名称的机会；一个类的多个构造器总是重载的。在许多情况下，可以选择导出静态工厂，而不是构造器（见第1条）。对于构造器，还不用担心重载和覆盖的相互影响，因为构造器不可能被覆盖。或许你有可能导出多个具有相同参数数目的构造器，所以有必要了解一下如何安全地做到这一点。

如果对于“任何一组给定的实际参数将应用于哪个重载方法上”始终非常清楚，那么，导出多个具有相同参数数目的重载方法就不可能使程序员感到混淆。如果对于每一对重载方法，至少有一个对应的参数在两个重载方法中具有“根本不同（radically different）”的类型，就属于这种情形。如果显然不可能把一种类型的实例转换为另一种类型，这两种类型就是根本不同的。在这种情况下，一组给定的实际参数应用于哪个重载方法上就完全由参数的运行时类型来决定，不可能受到其编译时类型的影响，所以主要的混淆根源就消除了。例如，ArrayList有一个构造器带一个int参数，另一个构造器带一个Collection参数。难以想像在什么情况下，会不清楚要调用哪一个构造器。

在Java 1.5发行版本之前，所有的基本类型都根本不同于所有的引用类型，但是当自动装箱出现之后，就不再如此了，它会导致真正的麻烦。考虑下面这个程序：

```
public class SetList {
    public static void main(String[] args) {
        Set<Integer> set = new TreeSet<Integer>();
        List<Integer> list = new ArrayList<Integer>();

        for (int i = -3; i < 3; i++) {
            set.add(i);
            list.add(i);
        }
        for (int i = 0; i < 3; i++) {
            set.remove(i);
            list.remove(i);
        }
        System.out.println(set + " " + list);
    }
}
```

程序将－3至2之间的整数添加到了排好序的集合和列表中，然后在集合和列表中都进行3次相同的remove调用。如果像大多数人一样，希望程序从集合和列表中去除非整数值（0，1和2），并打印出[－3，－2，－1] [－3，－2，－1]。事实上，程序从集合中去除了非整数，还从列表中去除了奇数值，打印出[－3，－2，－1] [－2，0，2]。将这种行为称之为混乱，已是保守的说法。

实际发生的情况是：set.remove(i)调用选择重载方法remove(E)，这里的E是集合（Integer）的元素类型，将i从int自动装箱到Integer中。这是你所期待的行为，因此程序不会从集合中去除正值。另一方面，list.remove(i)调用选择重载方法remove(int i)，它从列表的指定位置上去除元素。如果从列表[－3，－2，－1，0，1，2]开始，去除第零个元素，接着去除第一个、第二个，得到的是[－2，0，2]，这个秘密被揭开了。为了解决这个问题，要将list.remove的参数转换成Integer，迫使选择正确的重载方法。另一种方法是，可以调用Integer.valueOf(i)，并将结果传给list.remove。这两种方法都如我们所料，打印出[－3，－2，－1] [－3，－2，－1]：

```
for (int i = 0; i < 3; i++) {
    set.remove(i);
    list.remove((Integer) i);  // or remove(Integer.valueOf(i))
}
```

前一个范例中所示的混乱行为在这里也出现了，因为List<E>接口有两个重载的remove方法：remove(E)和remove(int)。当它在Java 1.5发行版本中被泛型化之前，List接口有一个remove(Object)而不是remove(E)，相应的参数类型：Object和int，则根本不同。但是自从有了泛型和自动装箱之后，这两种参数类型就不再根本不同了。换句话说，Java语言中添加了泛型和自动装箱之后，破坏了List接口。幸运的是，Java类库中几乎再没有API受到同样的破坏，但是这种情形清楚地说明了，自动装箱和泛型成了Java语言的一部分之后，谨慎重载显得更加重要了。

数组类型和Object之外的类截然不同。数组类型和Serializable与Cloneable之外的接口也截然不同。如果两个类都不是对方的后代，这两个独特的类就是不相关的（**unrelated**）[JLS, 5.5]。例如，String和Throwable就是不相关的。任何对象都不可能是两个不相关的类的实例，因此不相关的类是根本不同的。

还有其他一些“类型对”的例子也是不能相互转换的[JLS, 5.1.12]，但是，一旦超出了上述这些简单的情形，大多数程序员要想搞清楚“一组实际的参数应用于哪个重载方法上”就会非常困难。确定选择哪个重载方法的规则是非常复杂的。这些规则在语言规范中占了33页的篇幅[JLS, 15.12.1-3]，很少有程序员能够理解其中的所有微妙之处。

有时候，尤其在更新现有类的时候，可能会被迫违反本条目的指导原则。例如，自从Java 1.4发行版本以来，String类就已经有一个contentEquals（StringBuffer）方法。在Java 1.5发行

版本中，新增了一个称作CharSequence的接口，用来为StringBuffer、StringBuilder、String、CharBuffer以及其他类似的类型提供公共接口，为实现这个接口，对它们全都进行了改造。在Java平台中增加CharSequence的同时，String也配备了重载的contentEquals方法，即contentEquals（CharSequence）方法，这个方法表示，当且仅当此String表示与CharSequence序列相同的char值时，返回true。

尽管这样的重载很显然违反了本条目的指导原则，但是只要当这两个重载方法在同样的参数上被调用时，它们执行相同的功能，重载就不会带来危害。程序员可能并不知道哪个重载函数会被调用，但只要这两个方法返回相同的结果就行。确保这种行为的标准做法是，让更具体化的重载方法把调用转发给更一般化的重载方法：

```
public boolean contentEquals(StringBuffer sb) {
    return contentEquals((CharSequence) sb);
}
```

虽然Java平台类库很大程度上遵循了本条目中的建议，但是也有诸多的类违背了。例如，String类导出两个重载的静态工厂方法：valueOf(char[])和valueOf(Object)，当这两个方法被传递了同样的对象引用时，它们所做的事情完全不同。没有正当的理由可以解释这一点，它应该被看作是一种反常行为，有可能会造成真正的混淆。

简而言之，“能够重载方法”并不意味着就“应该重载方法”。一般情况下，对于多个具有相同参数数目的方法来说，应该尽量避免重载方法。在某些情况下，特别是涉及构造器的时候，要遵循这条建议也许是不可能的。在这种情况下，至少应该避免这样的情形：同一组参数只需经过类型转换就可以被传递给不同的重载方法。如果不能避免这种情形，例如，因为正在改造一个现有的类以实现新的接口，就应该保证：当传递同样的参数时，所有重载方法的行为必须一致。如果不能做到这一点，程序员就很难有效地使用被重载的方法或者构造器，他们就不能理解它为什么不能正常地工作。

第42条：慎用可变参数

Java 1.5发行版本中增加了可变参数（varargs）方法，一般称作**variable arity method**（可匹配不同长度的变量的方法）[JLS，8.4.1]。可变参数方法接受0个或者多个指定类型的参数。可变参数机制通过先创建一个数组，数组的大小为在调用位置所传递的参数数量，然后将参数值传到数组中，最后将数组传递给方法。

例如，下面就是一个可变参数方法，带有int参数的一个序列，并返回它们的总和。正如你所期望的，sum(1, 2, 3)的值为6，sum()的值为0：

```
// Simple use of varargs
static int sum(int... args) {
    int sum = 0;
    for (int arg : args)
        sum += arg;
    return sum;
}
```

有时候，有必要编写需要1个或者多个某种类型参数的方法，而不是需要0个或者多个。例如，假设想要计算多个int参数的最小值。如果客户端没有传递参数，那这个方法的定义就不太好了。你可以在运行时检查数组长度：

```
// The WRONG way to use varargs to pass one or more arguments!
static int min(int... args) {
    if (args.length == 0)
        throw new IllegalArgumentException("Too few arguments");
    int min = args[0];
    for (int i = 1; i < args.length; i++)
        if (args[i] < min)
            min = args[i];
    return min;
}
```

这种解决方案有几个问题。其中最严重的问题是，如果客户端调用这个方法时，并没有传递参数进去，它就会在运行时而不是编译时失败。另一个问题是，这段代码很不美观。你必须在args中包含显式的有效性检查，除非将min初始化为Integer.MAX_VALUE，否则将无法使用for-each循环，这样的代码也不美观。

幸运的是，有一种更好的方法可以实现想要的这种效果。声明该方法带有两个参数，一个是指定类型的正常参数，另一个是这种类型的varargs参数。这种解决方案解决了前一个示例中的所有不足：

```
// The right way to use varargs to pass one or more arguments
static int min(int firstArg, int... remainingArgs) {
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
```

```
            min = arg;
    return min;
}
```

如你所见，当你真正需要让一个方法带有不定数量的参数时，可变参数就非常有效。可变参数是为printf而设计的，它是在Java 1.5发行版本中添加到平台中的，为了核心的反射机制（见第53条），在该发行版本中被改造成利用可变参数。printf和反射机制都从可变参数中极大地受益。

可以将以数组当作final参数的现有方法，改造成以可变参数代替，而不影响现有的客户端。但是可以并不意味着应该这么做！考虑Arrays.asList的情形。这个方法从来都不是设计成用来将多个参数集中到一个列表中的，但是当平台中增加了可变参数时，将它改造成这么做似乎是个好办法。因此，变成可以这样：

```
List<String> homophones = Arrays.asList("to", "too", "two");
```

这种用法有效，但是这么用就是个大错误。在Java 1.5发行版本之前，打印数组内容的常见做法如下：

```
// Obsolete idiom to print an array!
System.out.println(Arrays.asList(myArray));
```

这种做法在当时是必需的，因为数组从Object继承了它们的toString实现，因此直接在数组上调用toString，会产生没有意义的字符串，如[Ljava.lang.Integer;@3e25a5。这种做法只在对象引用类型的数组上才有用，但是如果不小心在基本类型的数组上尝试这么做，程序将无法编译。例如，这个程序：

```
public static void main(String[] args) {
    int[] digits = { 3, 1, 4, 1, 5, 9, 2, 6, 5, 4 };
    System.out.println(Arrays.asList(digits));
}
```

在发行版本1.4中，这个程序会产生这条错误消息：

```
Va.java:6: asList(Object[]) in Arrays can't be applied to (int[])
        System.out.println(Arrays.asList(digits));
                                         ^
```

由于在Java 1.5发行版本中，令人遗憾地决定将Arrays.asList改造成可变参数方法，现在这个程序可以通过编译，并且没有错误或者警告。但是运行这个程序时，会输出无意识的也是无意义的结果：[[I@3e25a5]。Arrays.asList方法现在“增强”为使用可变参数，将int类型的数组digits的对象引用集中到数组的单个元素数组中，并忠实地将它包装到List<int[]>实例中。打印这个列表会导致在列表中调用toString，从而导致在它唯一的元素int数组上调用toString，产生上述令人遗憾的结果。

从好的方面看，将数组转变成字符串的Arrays.asList做法现在是过时的，当前的做法要健壮得多。也是在Java 1.5发行版本中，Arrays类得到了补充完整的Arrays.toString方法（不是可变参数方法！），专门为了将任何类型的数组转变成字符串而设计的。如果用Arrays.toString代替Arrays.asList，这个程序就会产生想要的结果：

```
// The right way to print an array
System.out.println(Arrays.toString(myArray));
```

如果不改造Arrays.asList，更好的办法则是给Collections添加一个新的方法，专门用来将它的参数集中到列表中：

```
public static <T> List<T> gather(T... args) {
    return Arrays.asList(args);
}
```

这种方法可以提供收集功能，而不会危及对现有Arrays.asList方法的类型检查。

这个教训很明显。*不必改造具有**final**数组参数的每个方法；只当确实是在数量不定的值上执行调用时才使用可变参数。*

有两个方法签名特别可疑：

```
ReturnType1 suspect1(Object... args) { }
<T> ReturnType2 suspect2(T... args) { }
```

带有上述任何一种签名的方法都可以接受任何参数列表。改造之前进行的任何编译时的类型检查都会丢失，Arrays.asList发生的情形正是说明了这一点。

在重视性能的情况下，使用可变参数机制要特别小心。可变参数方法的每次调用都会导致进行一次数组分配和初始化。如果凭经验确定无法承受这一成本，但又需要可变参数的灵活性，还有一种模式可以让你如愿以偿。假设确定对某个方法95%的调用会有3个或者更少的参数，就声明该方法的5个重载，每个重载方法带有0至3个普通参数，当参数的数目超过3个时，就使用一个可变参数方法：

```
public void foo() { }
public void foo(int a1) { }
public void foo(int a1, int a2) { }
public void foo(int a1, int a2, int a3) { }
public void foo(int a1, int a2, int a3, int... rest) { }
```

现在你知道了，所有调用中只有5%参数数量超过3个的调用需要创建数组。就像大多数的性能优化一样，这种方法通常不太恰当，但是一旦真正需要它时，它可就帮上大忙了。

EnumSet类对它的静态工厂使用这种方法，最大限度地减少创建枚举集合的成本。当时这

么做是有必要的，因为枚举集合为位域提供在性能方面有竞争力的替代方法，这是很重要的(见第32条)。

简而言之，在定义参数数目不定的方法时，可变参数方法是一种很方便的方式，但是它们不应该被过度滥用。如果使用不当，会产生混乱的结果。

第43条：返回零长度的数组或者集合，而不是null

像下面这样的方法并不少见：

```
private final List<Cheese> cheesesInStock = ...;

/**
 * @return an array containing all of the cheeses in the shop,
 *     or null if no cheeses are available for purchase.
 */
public Cheese[] getCheeses() {
    if (cheesesInStock.size() == 0)
        return null;
    ...
}
```

把没有奶酪（cheese）可买的情况当作是一种特例，这是不合常理的。这样做会要求客户端中必须有额外的代码来处理null返回值，例如：

```
Cheese[] cheeses = shop.getCheeses();
if (cheeses != null &&
    Arrays.asList(cheeses).contains(Cheese.STILTON))
    System.out.println("Jolly good, just the thing.");
```

而不是下面这段代码：

```
if (Arrays.asList(shop.getCheeses()).contains(Cheese.STILTON))
    System.out.println("Jolly good, just the thing.");
```

对于一个返回null而不是零长度数组或者集合的方法，几乎每次用到该方法时都需要这种曲折的处理方式。这样做很容易出错，因为编写客户端程序的程序员可能会忘记写这种专门的代码来处理null返回值。这样的错误也许几年都不会被注意到，因为这样的方法通常返回一个或者多个对象。返回null而不是零长度的数组也会使返回数组或者集合的方法本身变得更加复杂，这一点虽然不是特别重要，但是也值得注意。

有时候会有人认为：null返回值比零长度数组更好，因为它避免了分配数组所需要的开销。这种观点是站不住脚的，原因有两点。第一，在这个级别上担心性能问题是不明智的，除非分析表明这个方法正是造成性能问题的真正源头（见第55条）。第二，对于不返回任何元素的调用，每次都返回同一个零长度数组是有可能的，因为零长度数组是不可变的，而不可变对象有可能被自由地共享（见第15条）。实际上，当你使用标准做法（standard idiom）把一些元素从一个集合转存到一个类型化的数组（typed array）中时，它正是这样做的：

```
// The right way to return an array from a collection
private final List<Cheese> cheesesInStock = ...;

private static final Cheese[] EMPTY_CHEESE_ARRAY = new Cheese[0];
```

```
/**
 * @return an array containing all of the cheeses in the shop.
 */
public Cheese[] getCheeses() {
    return cheesesInStock.toArray(EMPTY_CHEESE_ARRAY);
}
```

在这种习惯用法中，零长度数组常量被传递给toArray方法，以指明所期望的返回类型。正常情况下，toArray方法分配了返回的数组，但是，如果集合是空的，它将使用零长度的输入数组，Collection.toArray(T[])的规范保证：如果输入数组大到足够容纳这个集合，它就将返回这个输入数组。因此，这种做法永远也不会分配零长度的数组。

同样地，集合值的方法也可以做成在每当需要返回空集合时都返回同一个不可变的空集合。Collections.emptySet、emptyList和emptyMap方法提供的正是你所需要的，如下所示：

```
// The right way to return a copy of a collection
public List<Cheese> getCheeseList() {
  if (cheesesInStock.isEmpty())
    return Collections.emptyList(); // Always returns same list
  else
    return new ArrayList<Cheese>(cheesesInStock);
}
```

简而言之，返回类型为数组或集合的方法没理由返回**null**，而不是返回一个零长度的数组或者集合。这种习惯做法（指返回null）很有可能是从C程序设计语言中沿袭过来的，在C语言中，数组长度是与实际的数组分开返回的。在C语言中，如果返回的数组长度为零，再分配一个数组就没有任何好处。

第44条：为所有导出的API元素编写文档注释

如果要想使一个API真正可用，就必须为其编写文档。传统意义上的API文档是手工生成的，所以保持文档与代码同步是一件很繁琐的事情。Java语言环境提供了一种被称为Javadoc的实用工具，从而使这项任务变得很容易。Javadoc利用特殊格式的文档注释（**documentation comment**，通常被写作**doc comment**），根据源代码自动产生API文档。

如果你对文档注释的规范还不太熟悉，应该去了解这些规范。虽然这些规范还没有正式成为Java程序设计语言的一部分，但它们已经构成了每个程序员都应该知道的事实API。这些规范的内容在**Sun**公司关于如何编写文档注释（**How to Write Doc Comments**）的网页上进行了说明[Javadoc-guide]。虽然这个网页在Java 1.4发行版本之后还没有进行更新，但它仍然是个很有价值的资源。Java 1.5发行版本中，为Javadoc新增了两个重要的Javadoc标签：{@literal}和{@code}[Javadoc-5.0]。本条目中会讨论到这些标签。

为了正确地编写**API**文档，必须在每个被导出的类、接口、构造器、方法和域声明之前增加一个文档注释。如果类是可序列化的，也应该对它的序列化形式编写文档（见第75条）。如果没有文档注释，Javadoc所能够做的也就是重新生成该声明，作为受影响的API元素的唯一文档。使用没有文档注释的API是非常痛苦的，也很容易出错。为了编写出可维护的代码，还应该为那些没有被导出的类、接口、构造器、方法和域编写文档注释。

方法的文档注释应该简洁地描述出它和客户端之间的约定。除了专门为继承而设计的类中的方法（见第17条）之外，这个约定应该说明这个方法做了什么，而不是说明它是如何完成这项工作的。文档注释应该列举出这个方法的所有前提条件（**precondition**）和后置条件（**postcondition**），所谓前提条件是指为了使客户能够调用这个方法，而必须要满足的条件；所谓后置条件是指在调用成功完成之后，哪些条件必须要满足。一般情况下，前提条件是由@throws标签针对未受检的异常所隐含描述的；每个未受检的异常都对应一个前提违例（precondition violation）。同样地，也可以在一些受影响的参数的@param标记中指定前提条件。

除了前提条件和后置条件之外，每个方法还应该在文档中描述它的副作用（**side effect**）。所谓副作用是指系统状态中可以观察到的变化，它不是为了获得后置条件而明确要求的变化。例如，如果方法启动了后台线程，文档中就应该说明这一点。最后，文档注释也应该描述类或者方法的线程安全性（**thread safety**），正如第70条中所述。

为了完整地描述方法的约定，方法的文档注释应该让每个参数都有一个@param标签，以及一个@return标签（除非这个方法的返回类型为void），以及对于该方法抛出的每个异常，无论是受检的还是未受检的，都有一个@throws标签（见第62条）。按惯例，跟在@param标签

或者@return标签后面的文字应该是一个名词短语，描述了这个参数或者返回值所表示的值。跟在@throws标签之后的文字应该包含单词“if”（如果），紧接着是一个名词短语，它描述了这个异常将在什么样的条件下会被抛出。有时候，也会用算术表达式来代替名词短语。按惯例，@param、@return或者@throws标签后面的短语或者子句都不用句点来结束。下面这个简短的文档注释演示了所有这些习惯做法：

```
/**
 * Returns the element at the specified position in this list.
 *
 * <p>This method is <i>not</i> guaranteed to run in constant
 * time. In some implementations it may run in time proportional
 * to the element position.
 *
 * @param  index index of element to return; must be
 *         non-negative and less than the size of this list
 * @return the element at the specified position in this list
 * @throws IndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= this.size()})
 */
E get(int index);
```

注意，这份文档注释中使用了HTML标签（<p>和<i>）。Javadoc工具会把文档注释翻译成HTML，文档注释中包含的任意HTML元素都会出现在结果HTML文档中。有时候，程序员会把HTML表格嵌入到它们的文档注释中，但是这种做法并不多见。

还要注意，@throws子句的代码片段中到处使用了Javadoc的{@code}标签。它有两个作用：造成该代码片段以代码字体（code font）进行呈现，并限制HTML标记和嵌套的Javadoc标签在代码片段中进行处理。后一种属性正是允许我们在代码片段中使用小于号（<）的东西，虽然它是一个HTML元字符。在Java 1.5发行版本之前，是通过使用HTML标签和HTML转义，将代码片段包含在文档注释中。现在再也没有必要在文档注释中使用**HTML <code>或者<tt>**标签了：**Javadoc {@code}**标签更好，因为它避免了转义**HTML**元字符。为了将多个代码示例包含在一个文档注释中，要使用包在HTML的<pre>标签里面的Javadoc {@code}标签。换句话说，是先在多行的代码示例前使用字符<pre>{@code，然后在代码后面加上}</pre>。

最后，要注意这个文档注释中用到了单词“this”。按惯例，当“this”被用在实例方法的文档注释中时，它应该始终是指方法调用所在的对象。

不要忘记，为了产生包含HTML元字符的文档，比如小于号（<）、大于号（>）以及“与”号（&），必须采取特殊的动作。让这些字符出现在文档中的最佳办法是用{@literal}标签将它们包围起来，这样就限制了HTML标记和嵌套的Javadoc标签的处理。除了它不以代码字体渲染文本之外，其余方面就像{@code}标签一样。例如，这个Javadoc片段：

```
 * The triangle inequality is {@literal |x + y| < |x| + |y|}.
```

产生了这样的文档："The triangle inequality is |x + y| < |x| + |y|." {@literal}标签也可以只是包住小于号，而不是整个不等式，所产生的文档是一样的，但是在源代码中见到的文档注释的可读性就会更差。这说明了一条通则：文档注释在源代码和产生的文档中都应该是易于阅读的。如果无法让两者都易读，产生的文档的可读性要优先于源代码的可读性。

每个文档注释的第一句话（如下所示）成了该注释所属元素的概要描述（**summary description**）。例如，第177页中文档注释中的概要描述为"返回这个列表中指定位置上的元素"。概要描述必须独立地描述目标元素的功能。为了避免混淆，同一个类或者接口中的两个成员或者构造器，不应该具有同样的概要描述。特别要注意重载的情形，在这种情况下，往往很自然地在描述中使用同样的第一句话（但在文档注释中这是不可接受的）。

注意所期待的概要描述中是否包括句点，因为句点会过早地终止这个描述。例如，一个以"A college degree, such as B.S., M.S., or Ph.D."开头的文档注释，会产生这样的概要描述："A college degree, such as B.S, M.S."问题在于，概要描述在后面接着空格、跳格或者行终结符的第一个句点处（或者在第一个块标签处）结束[Javadoc-ref]。在这种情况下，缩写"M.S."中的第二个句点就要接着用一个空格。最好的解决方法是，将讨厌的句点以及任何与{@literal}关联的文本都包起来，因此在源代码中，句点后面就不再是空格了：

```
/**
 * A college degree, such as B.S., {@literal M.S.} or Ph.D.
 * College is a fountain of knowledge where many go to drink.
 */
public class Degree { ... }
```

说概要描述是文档注释中的第一个句子（**sentence**），这似乎有点误导人。规范指出，概要描述很少是个完整的句子。对于方法和构造器而言，概要描述应该是个完整的动词短语（包含任何对象），它描述了该方法所执行的动作。例如：

- ArrayList(int initialCapacity)—Constructs an empty list with the specified initial capacity.（用指定的初始容量构造一个空的列表）
- Collection.size()—Returns the number of elements in this collection.（返回该集合中元素的数目）

对于类、接口和域，概要描述应该是一个名词短语，它描述了该类或者接口的实例，或者域本身所代表的事物。例如：

- TimerTask—A task that can be scheduled for one-time or repeated execution by a Timer.（可以调度一次的任务，或者被Timer重复执行的任务）

- Math.PI—The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.（非常接近于PI（圆周长度与直径的比值）的double值）

Java 1.5发行版本中增加的三个特性在文档注释中需要特别小心：泛型、枚举和注解。当为泛型或者方法编写文档时，确保要在文档中说明所有的类型参数。

```
/**
 * An object that maps keys to values.  A map cannot contain
 * duplicate keys; each key can map to at most one value.
 *
 * (Remainder omitted)
 *
 * @param <K> the type of keys maintained by this map
 * @param <V> the type of mapped values
 */
public interface Map<K, V> {
    ... // Remainder omitted
}
```

当为枚举类型编写文档时，要确保在文档中说明常量，以及类型，还有任何公有的方法。注意，如果文档注释很简短，可以将整个注释放在一行上：

```
/**
 * An instrument section of a symphony orchestra.
 */
public enum OrchestraSection {
    /** Woodwinds, such as flute, clarinet, and oboe. */
    WOODWIND,

    /** Brass instruments, such as french horn and trumpet. */
    BRASS,

    /** Percussion instruments, such as timpani and cymbals */
    PERCUSSION,

    /** Stringed instruments, such as violin and cello. */
    STRING;
}
```

为注解类型编写文档时，要确保在文档中说明所有成员，以及类型本身。带有名词短语的文档成员，就像是域一样。对于该类型的概要描述，要使用一个动词短语，说明当程序元素具有这种类型的注解时它表示什么意思。

```
/**
 * Indicates that the annotated method is a test method that
 * must throw the designated exception to succeed.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    /**
     * The exception that the annotated test method must throw
     * in order to pass. (The test is permitted to throw any
     * subtype of the type described by this class object.)
     */
```

```
    Class<? extends Exception> value();
}
```

从Java 1.5发行版本开始，包级私有的文档注释就应该放在一个称作package-info.java的文件中，而不是放在package.html中。除了包级私有的文档注释之外，package-info.java也可以（但并非必需）包含包声明和包注解。

类的导出API有两个特征经常被人忽视，即线程安全性和可序列化性。类是否是线程安全的，应该在文档中对它的线程安全级别进行说明，如第70条中所述。如果类是可序列化的，就应该在文档中说明它的序列化形式，如第75条中所述。

Javadoc具有“继承”方法注释的能力。如果API元素没有文档注释，Javadoc将会搜索最为适用的文档注释，接口的文档注释优先于超类的文档注释。搜索算法的细节可以在《*The Javadoc Reference Guide*》[Javadoc-ref]中找到。也可以利用{@inheritDoc}标签从超类型中继承文档注释的部分内容。这意味着，不说别的，类还可以重用它所实现的接口的文档注释，而不需要拷贝这些注释。这项机制有可能减轻维护多个几乎相同的文档注释的负担，但它使用起来比较需要一些小技巧（tricky），并具有一些局限性。关于这一点的详情超出了本书的范围，在此不做讨论。

为了降低文档注释中出错的可能性，一种简单的办法是通过一个**HTML有效性检查器**（**HTML validity checker**）来运行由Javadoc产生的HTML文件。这样可以检测出HTML标签的许多不正确用法，以及应该被转义的HTML元字符。Internet上有一些HTML有效性检查器可供下载，并且可以在线检验HTML[W3C-validator]。

关于文档注释有一点需要特别注意。虽然为所有导出的API元素提供文档注释是必要的，但是这样做并非永远就足够了。对于由多个相互关联的类组成的复杂API，通常有必要用一个外部文档来描述该API的总体结构，对文档注释进行补充。如果有这样的文档，相关的类或者包文档注释就应该包含一个对这个外部文档的链接。

本条目中所述的内容涵盖了基本的惯例。关于编写文档注解最权威的指导是Sun公司的《*How to Write Doc Comments*（如何编写文档注释）》[Javadoc-guide]。

简而言之，要为API编写文档，文档注释是最好、最有效的途径。对于所有可导出的API元素来说，使用文档注释应该被看作是强制性的。要采用一致的风格来遵循标准的约定。记住，在文档注释内部出现任何HTML标签都是允许的，但是HTML元字符必须要经过转义。

第8章

通用程序设计

本章主要讨论Java语言的具体细节，讨论了局部变量的处理、控制结构、类库的用法、各种数据类型的用法，以及两种不是由语言本身提供的机制（**reflection**和**native method**，*反射机制和本地方法*）的用法。最后讨论了优化和命名惯例。

第45条：将局部变量的作用域最小化

本条目与第13条（使类和成员的可访问性最小化）本质上是类似的。将局部变量的作用域最小化，可以增强代码的可读性和可维护性，并降低出错的可能性。

较早的程序设计语言（如C语言）要求局部变量必须在一个代码块的开头处进行声明，出于习惯，有些程序员们目前还是继续这样做。这个习惯应该改正。在此提醒，Java允许你在任何可以出现语句的地方声明变量。

*要使局部变量的作用域最小化，最有力的方法就是在第一次使用它的地方声明。*如果变量在使用之前进行声明，这只会造成混乱——对于试图理解程序功能的读者来说，这又多了一种只会分散他们注意力的因素。等到用到该变量的时候，读者可能已经记不起该变量的类型或者初始值了。

过早地声明局部变量不仅会使它的作用域过早地扩展，而且结束得也过于晚了。局部变量的作用域从它被声明的点开始扩展，一直到外围块（block）的结束处。如果变量是在“使用它的块”之外被声明的，当程序退出该块之后，该变量仍是可见的。如果变量在它的目标使用区域之前或者之后被意外地使用的话，后果将可能是灾难性的。

*几乎每个局部变量的声明都应该包含一个初始化表达式。*如果你还没有足够的信息来对一个变量进行有意义的初始化，就应该推迟这个声明，直到可以初始化为止。这条规则有个例外的情况与try-catch语句有关。如果一个变量被一个方法初始化，而这个方法可能会抛出一个

受检的异常（checked exception），该变量就必须在try块的内部被初始化。如果变量的值必须在try块的外部被使用到，它就必须在try块之前被声明，但是在try块之前，它还不能被“有意义地初始化”。请参照第202页中的例子。

循环中提供了特殊的机会来将变量的作用域最小化。（无论是传统的还是for-each形式的）for循环，都允许声明循环变量（**loop variable**），它们的作用域被限定在正好需要的范围之内。（这个范围包括循环体，以及循环体之前的初始化、测试、更新部分。）因此，如果在循环终止之后不再需要循环变量的内容，**for循环就优先于while循环**。

例如，下面是一种遍历集合的首选做法（见第46条）：

```
// Preferred idiom for iterating over a collection
for (Element e : c) {
    doSomething(e);
}
```

在Java 1.5发行版本之前，首选的做法如下（现在仍然有适用之处）：

```
// No for-each loop or generics before release 1.5
for (Iterator i = c.iterator(); i.hasNext(); ) {
    doSomething((Element) i.next());
}
```

为了弄清楚为什么这个for循环比while循环更好，请考虑下面的代码片断，它包含两个while循环，以及一个Bug：

```
Iterator<Element> i = c.iterator();
while (i.hasNext()) {
    doSomething(i.next());
}
...

Iterator<Element> i2 = c2.iterator();
while (i.hasNext()) {              // BUG!
    doSomethingElse(i2.next());
}
```

第二个循环中包含一个“剪切-粘贴”错误：它本来是要初始化一个新的循环变量i2，却使用了旧的循环变量i，遗憾的是，这时i仍然还在有效范围之内。结果代码仍然可以通过编译，运行的时候也不会抛出异常，但是它所做的事情却是错误的。第二个循环并没有在c2上迭代，而是立即终止，造成c2为空的假象。因为这个程序的错误是悄然发生的，所以可能在很长时间内都不会被发现。

如果类似的“剪切-粘贴”错误发生在前面任何一种for循环中，结果代码就根本不能通过编译。在第二个循环开始之前，第一个循环的元素（或者迭代器）变量已经不在它的作用域范围之内了：

```
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    doSomething(i.next());
}
    ...

// Compile-time error - cannot find symbol i
for (Iterator<Element> i2 = c2.iterator(); i.hasNext(); ) {
    doSomething(i2.next());
}
```

而且，如果使用for循环，犯这种“剪切-粘贴”错误的可能性就会大大降低，因为通常没有必要在两个循环中使用不同的变量名。循环是完全独立的，所以重用元素（或者迭代器）变量的名称不会有任何危害。实际上，这也是很流行的做法。

使用for循环与使用while循环相比还有另外一个优势：更简短，从而增强了可读性。

下面是另外一种对局部变量的作用域进行最小化的循环做法：

```
for (int i = 0, n = expensiveComputation(); i < n; i++) {
    doSomething(i);
}
```

关于这种做法要关注的一点是，它具有两个循环变量：i和n，两者具有完全相同的作用域。第二个变量n被用来保存第一个变量的极限值，从而避免在每次迭代中执行冗余计算的开销。通常，如果循环测试中涉及方法调用，它可以保证在每次迭代中都会返回同样的结果，就应该使用这种做法。

最后一种“将局部变量的作用域最小化”的方法是使方法小而集中。如果把两个操作（activity）合并到同一个方法中，与其中一个操作相关的局部变量就有可能会出现在执行另一个操作的代码范围之内。为了防止这种情况发生，只要把这个方法分成两个，每个方法各执行一个操作。

第46条：for-each循环优先于传统的for循环

在Java 1.5发行版本之前，对集合进行遍历的首选做法如下：

```
// No longer the preferred idiom to iterate over a collection!
for (Iterator i = c.iterator(); i.hasNext(); ) {
    doSomething((Element) i.next()); // (No generics before 1.5)
}
```

遍历数组的首选做法如下：

```
// No longer the preferred idiom to iterate over an array!
for (int i = 0; i < a.length; i++) {
    doSomething(a[i]);
}
```

这些做法都比while循环（见第45条）更好，但是它们也并不完美。迭代器和索引变量都会造成一些混乱。而且，它们也代表着出错的可能。迭代器和索引变量在每个循环中出现三次，其中有两次让你很容易出错。一旦出错，就无法保证编译器能够发现错误。

Java 1.5发行版本中引入的for-each循环，通过完全隐藏迭代器或者索引变量，避免了混乱和出错的可能。这种模式同样适用于集合和数组：

```
// The preferred idiom for iterating over collections and arrays
for (Element e : elements) {
    doSomething(e);
}
```

当见到冒号（:）时，可以把它读作“在…里面”。因此上面的循环可以读作“对于元素中的每个元素e。”注意，利用for-each循环不会有性能损失，甚至用于数组也一样。实际上，在某些情况下，比起普通的for循环，它还稍有些性能优势，因为它对数组索引的边界值只计算一次。虽然可以手工完成这项工作（见第45条），但程序员并不总会这么做。

在对多个集合进行嵌套式迭代时，for-each循环相对于传统for循环的这种优势还会更加明显。下面就是人们在试图对两个集合进行嵌套迭代时经常会犯的错误：

```
// Can you spot the bug?
enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
            NINE, TEN, JACK, QUEEN, KING }
...
Collection<Suit> suits = Arrays.asList(Suit.values());
Collection<Rank> ranks = Arrays.asList(Rank.values());

List<Card> deck = new ArrayList<Card>();
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
        deck.add(new Card(i.next(), j.next()));
```

如果之前没有发现这个Bug也不必难过。许多专家级的程序员偶尔也会犯这样的错误。问题在于，在迭代器上对外部的集合（suits）调用了太多次next方法了。它应该从外部的循环进行调用，以便每种花色调用一次，但它却是从内部循环调用，因此它是每张牌调用一次。在用完所有花色之后，循环就会抛出NoSuchElementException异常。

如果真的那么不幸，并且外部集合的大小是内部集合大小的几倍——可能因为它们是相同的集合——循环就会正常终止，但是不会完成你想要的工作。例如，下面是个考虑不周的尝试，要打印一对骰子的所有可能的滚法。

```
// Same bug, different symptom!
enum Face { ONE, TWO, THREE, FOUR, FIVE, SIX }
...
Collection<Face> faces = Arrays.asList(Face.values());

for (Iterator<Face> i = faces.iterator(); i.hasNext(); )
    for (Iterator<Face> j = faces.iterator(); j.hasNext(); )
        System.out.println(i.next() + " " + j.next());
```

这个程序不会抛出异常，而是只打印6个重复的词（从“ONE ONE”到“SIX SIX”），而不是预计的36种组合。

为了修正这些示例中的Bug，必须在外部循环的作用域中添加一个变量来保存外部元素：

```
// Fixed, but ugly - you can do better!
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); ) {
    Suit suit = i.next();
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
        deck.add(new Card(suit, j.next()));
}
```

如果使用的是嵌套的for-each循环，这个问题就会完全消失。产生的代码就如你所希望得那样简洁。

```
// Preferred idiom for nested iteration on collections and arrays
for (Suit suit : suits)
    for (Rank rank : ranks)
        deck.add(new Card(suit, rank));
```

for-each循环不仅让你遍历集合和数组，还让你遍历任何实现Iterable接口的对象。这个简单的接口由单个方法组成，与for-each循环同时被增加到Java平台中。下面就是这个接口的示例：

```
public interface Iterable<E> {
    // Returns an iterator over the elements in this iterable
    Iterator<E> iterator();
}
```

实现Iterable接口并不难。如果你在编写的类型表示的是一组元素，即使你选择不让它实现Collection，也要让它实现Iterable。这样可以允许用户利用for-each循环遍历你的类型，会令用户永远感激不尽的。

总之，for-each循环在简洁性和预防Bug方面有着传统的for循环无法比拟的优势，并且没有性能损失。应该尽可能地使用for-each循环。遗憾的是，有三种常见的情况无法使用for-each循环：

1. **过滤**——如果需要遍历集合，并删除选定的元素，就需要使用显式的迭代器，以便可以调用它的remove方法。
2. **转换**——如果需要遍历列表或者数组，并取代它部分或者全部的元素值，就需要列表迭代器或者数组索引，以便设定元素的值。
3. **平行迭代**——如果需要并行地遍历多个集合，就需要显式地控制迭代器或者索引变量，以便所有迭代器或者索引变量都可以得到同步前移（就如上述关于有问题的牌和骰子的示例中无意中所示范的那样）。

在以上任何一种情况下，就要使用普通的for循环，要警惕本条目中提到的陷阱，并且要确保做到最好。

第47条：了解和使用类库

假设你希望产生位于0和某个上界之间的随机整数。面对这个常见的任务，许多程序员会编写出如下所示的方法：

```
private static final Random rnd = new Random();

// Common but deeply flawed!
static int random(int n) {
    return Math.abs(rnd.nextInt()) % n;
}
```

这个方法看起来可能不错，但是却有三个缺点。第一个缺点是，如果n是一个比较小的2的乘方，经过一段相当短的周期之后，它产生的随机数序列将会重复。第二个缺点是，如果n不是2的乘方，那么平均起来，有些数会比其他的数出现得更为频繁。如果n比较大，这个缺点就会非常明显。这可以通过下面的程序直观地体现出来，它会产生一百万个经过细心指定的范围内的随机数，并打印出有多少个数字落在随机数取值范围的前半部分：

```
public static void main(String[] args) {
    int n = 2 * (Integer.MAX_VALUE / 3);
    int low = 0;
    for (int i = 0; i < 1000000; i++)
        if (random(n) < n/2)
            low++;

    System.out.println(low);
}
```

如果random方法工作正常的话，这个程序打印出来的数将接近于一百万的一半，但是如果真正运行这个程序，就会发现它打印出来的数接近于666 666。由random方法产生的数字有2/3落在随机数取值范围的前半部分。

random方法的第三个缺点是，在极少数情况下，它的失败是灾难性的，返回一个落在指定范围之外的数。之所以如此，是因为这个方法试图通过调用Math.abs，将rnd.nextInt()返回的值映射为一个非负整数int。如果nextInt()返回Integer.MIN_VALUE，那么Math.abs也会返回Integer.MIN_VALUE，假设n不是2的乘方，那么取模操作符（%）将返回一个负数。这几乎肯定会使程序失败，而且这种失败很难重现。

为了编写能修正这三个缺点的random方法，有必要了解关于伪随机数生成器、数论和2的求补算法的相关知识。幸运的是，你并不需要自己来做这些工作——已经有现成的成果可以为你所用。它被称为Random.nextInt(int)，自Java 1.2发行版本以来，它已经成了Java平台的一部分。

你无需关心nextInt(int)的实现细节（如果你有强烈的好奇心，可以研究它的文档或者源代

码)。具有算法背景的高级工程师已经花了大量的时间来设计、实现和测试这个方法，然后经过这个领域中的专家的审查，以确保它的正确性。然后，标准类库经过了Beta测试、发行和近十年的成千上万程序员的广泛使用。在这个方法中还没有发现过缺陷，但是，如果将来发现有缺陷，在下一个发行版本中就会修正这些缺陷。通过使用标准类库，可以充分利用这些编写标准类库的专家的知识，以及在你之前的其他人的使用经验。

使用标准类库的第二个好处是，不必浪费时间为那些与工作不太相关的问题提供特别的解决方案。就像大多数程序员一样，应该把时间花在应用程序上，而不是底层的细节上。

使用标准类库的第三个好处是，它们的性能往往会随着时间的推移而不断提高，无需你做任何努力。因为许多人在使用它们，被当作工业标准在使用，所以，提供这些标准类库的组织有强烈的动机要使它们运行得更快。这些年来，许多Java平台类库已经被重新编写了，有时候是重复编写，从而导致性能上有了显著的提高。

标准类库也会随着时间的推移而增加新的功能。如果类库中漏掉了某些功能，开发者社区(developer community)就会把这些缺点告示出来，漏掉的功能就会添加到后续的发行版本中。Java平台类库始终是在这个社区的推动下不断发展的。

使用标准类库的最后一个好处是，可以使自己的代码融入主流。这样的代码更易读、更易维护、更易被大多数的开发人员重用。

既然有那么多的优点，使用标准类库机制而不选择专门的实现，这显然是符合逻辑的，然而还是有相当一部分的程序员没有这样做。为什么呢？可能他们并不知道有这些类库机制的存在。在每个重要的发行版本中，都会有许多新的特性被加入到类库中，所以与这些新特性保持同步是值得的。每次Java平台有重要的发行时，Sun公司都会发布一个网页，说明新的特性。这些网页值得好好读一读[Java5-feat，Java6-feat]。这些标准类库太庞大了，以至于不可能去学习所有的文档[JavaSE6]，但是每个程序员都应该熟悉**java.lang**、**java.util**，某种程度上还有**java.io**中的内容。关于其他类库的知识可以根据需要随时学习。

本条目不可能总结类库中所有的便利工具，但是有两种工具值得特别一提。在1.2发行版本中，**Collections Framework**（集合框架）被加入到了java.util包中。它应该成为每个程序员基本工具箱中的一部分。Collections Framework是一个统一的体系结构，用来表示和操作集合，允许它们对集合进行独立于表示细节的操作。它减轻了编程的负担，同时还提升了性能。它考虑到不相关的API之间的互操作性，减少了为设计和学习新的API所要付出的努力，并且鼓励软件重用。如果想要了解更多这方面的细节，请参见Sun公司网站上的文章[Collections]，或者阅读有关的教程[Bloch06]。

1.5发行版本中，在**java.util.concurrent**包中增加了一组并发实用工具。这个包既包含高

级的并发工具来简化多线程的编程任务，还包含低级别的并发基本类型，允许专家们自己编写更高级的并发抽象。java.util.concurrent的高级部分，也应该是每个程序员基本工具箱中的一部分（见第68条和第69条）。

在有些情况下，一个类库工具并不能满足你的需要。你的需求越是特殊，这种情形就越有可能发生。虽然你的第一个念头应该是使用标准类库，但是，如果你在观察了它们在某些领域所提供的功能之后，确定它不能满足需要，你就得使用其他的实现。任何一组类库所提供的功能总是难免会有遗漏。如果你所需要的功能不存在，那么，就只能自己实现这些功能，别无选择。

总而言之，不要重新发明轮子。如果你要做的事情看起来是十分常见的，有可能类库中已经有某个类完成了这样的工作。如果确实是这样，就使用现成的；如果还不清楚是否存在这样的类，就去查一查。一般而言，类库的代码可能比你自己编写的代码更好一些，并且会随着时间的推移而不断改进。这并不是在影射你作为一个程序员的能力。从经济角度的分析表明：类库代码受到的关注远远超过大多数普通程序员在同样的功能上所能够给予的投入。