1. 创建和销毁对象
2. 用静态工厂方法代替构造器

public static Boolean valueOf(Boolean b){

return b ? Boolean.true:Boolean.false;

}

一优势：它们有名称。

二优势：不必在每次调用它们的时候都创建一个对象。

三优势：它们可以返回原返回类型的任何子类型的对象。

四优势：所返回的对象的类可以随着每次调用而发生变化，这取决于静态工厂方法的 参数值（只要是已声明的返回类型的子类型）。

五优势：方法返回的对象所属的类，在编写包含该静态工厂方法的类时可以不存在。

一缺点：类如果不含公有的或者受保护的构造器，就不能被子类化。

二缺点：程序员很难发现它们。

第二条 遇到多个构造器参数时要考虑使用构建器

new Test(String a, String b, String c);

简而言之，重叠构造器模式可行，但是当有许多参数的时候，客户端代码会很难编写并且仍然较难以读懂。

JavaBeans模式：先调用一个无参构造器来创建对象，然后在调用setter方法来设置每个必要的参数。

JavaBeans不足之处：在构造过程中JavaBean可能处于不一致的状态。

JavaBeans模式使得把类作出不可变的可能性不复存在。

Builder模式模拟了具名的可选参数。

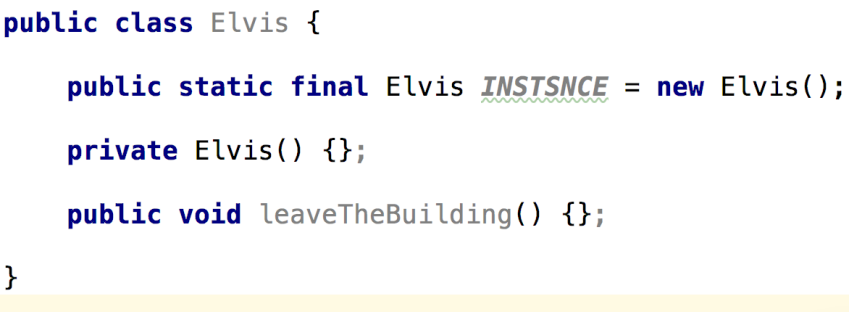
Builder模式也适用于类层次结构：子类方法声明返回超级类中声明的返回类型的子类型，这被称为协变返回类型。

如果类的构造器或者静态工厂中具有多个参数，设计这种类时，Builder模式就是一种不错的选择。

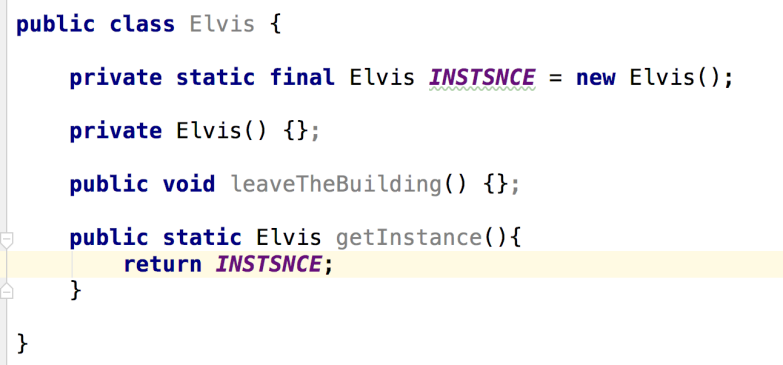
第三条 用私有构造器或者枚举类型强化Singleton属性

Singleton是指仅仅被实例化一次的类。Singleton通常被用来代表一个无状态的对象，如函数，或者那些本质上唯一的系统组件。使类成为Singleton会使它的客户端测试变得十分困难。

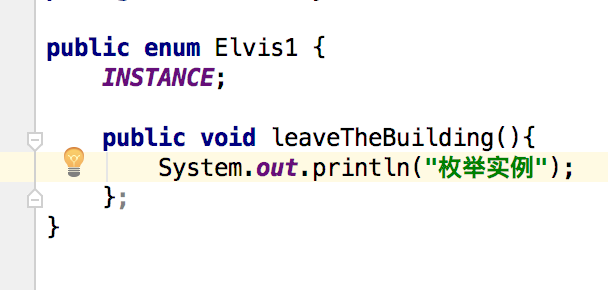
实现Singleton有两种常见的方法，这两种方法都要保持构造器为私有的，并导出公有的静态成员，以便允许客户端能够访问该类的唯一实例。



二，都会返回同一个对象引用，所以，永远不会创建其他的Elvis实例



单元素的枚举类型经常成为实现Singleton的最佳方法。



1. ：通过私有构造器强化不可实例化的能力

企图通过将类做成抽象类来强制该类不可被实例化是行不通的。该类可以被子类化，并且该子类也可以被实例化。由于只有当类不包含显示的构造器时，编译器才会生产缺省的构造器，因此只要让这个类包含一个私有构造器，它就不能被实例化。

public class Test {

//为非实例性禁用默认构造函数

private Test(){

//避免不小心在类的内部调用构造器。

throw new AssertionError();

}

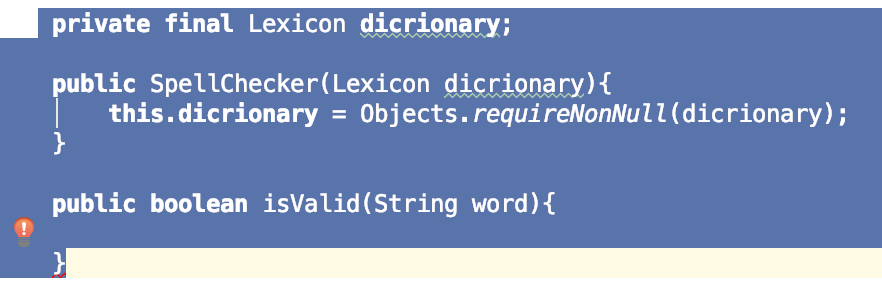
}

副作用：它使得一个类不能被子类化，所有的构造器都必须显式的或隐式的调用超类构造器，在这种情况下，子类就没有可访问的超类构造器可调用了。

1. ：优先考虑依赖注入来引用资源

静态工具类和Singleton类不适合于需要引用底层资源的类。

当创建一个新的实例时，就将该资源传到构造器中，这是依赖注入的一种形式：词典就是拼写检查器的一个依赖，在创建拼写检查器时就将词典注入其中。



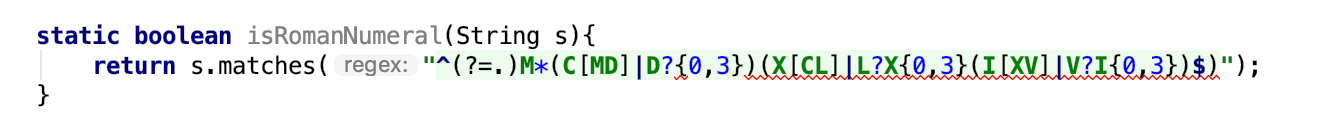
这个程序模式的另一种有用的变体是，将资源工厂(factory)传给构造器。工厂是可以被重复调用来创建类型实例的一个对象。

总而言之，不要用Singleton和静态工具类来实现依赖一个或多个底层资源的类，且该资源的行为会影响到该类的行为；也不要直接用这个类来创建这些资源。而应该将这些资源或者工厂传给构造器，通过它们来创建类。这个实践就被称作依赖注入，它极大地提升了类的灵活性，可重用性和可测试性。

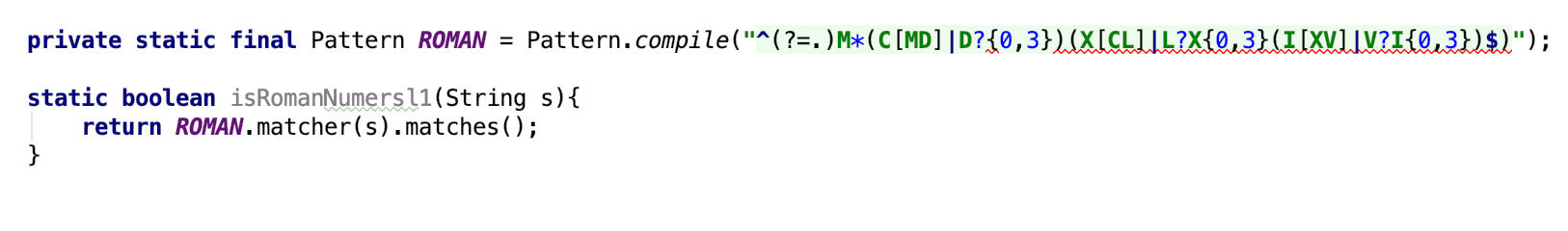
1. ：避免创建不必要的对象

例子：String s = new String(“Hello”);//该语句每次执行的时候都创建一个新的String实例，但是这些创建对象的动作全都是不必要的。

String s = “Hello”;//这个版本只用了一个String实例，而不是每次执行的时候都创建一个新的实例。而且，它可以保证，对于所有在同一台虚拟机中运行的代码，只要它们包含相同的字符串字面常量，该对象就会被重用。



虽然String.matches方法最易于查看一个字符串是否与正则表达式相匹配，但并不适合在注重性能的情形中使用。（在加载的时候初始化）



适配器是指这样一个对象：它把功能委托给一个后备对象，从而为后备对象提供一个可以替代的接口。

另一种创建多余对象的方法，称作自动装箱，它允许程序员将基本类型和装箱基本类型混用，按需要自动装箱和拆箱，按需要自动装箱和拆箱。自动装箱使得基本类型和装箱基本类型之间的差别变得模糊起来，但是并没有完全消除。

结论：要优先使用基本类型而不是装箱基本类型，要当心无意识的自动装箱。

1. ：消除过期的对象引用

如果一个栈先是增长了，然后在收缩，那么，从栈中弹出来的对象将不会被当作垃圾回收，即使使用栈的程序不再引用这些对象，它们也不会被回收。这是因为栈内部维护着对这些对象的过期引用(obsolete reference)，所谓的过期引用，是指永远也不会在被解除的引用。

在支持垃圾回收的语言中，内存泄漏是很隐蔽的(称这类内存泄漏为“无意识的对象保持”)。如果一个对象引用被无意识地保留起来了，那么垃圾回收机制不仅不会处理这个对象，而且也不会处理被这个对象所引用的所有其他对象。

清空对象引用应该是一种例外，而不是一种规范。消除过期引用最好的方法是让包含该引用的变量结束其生命周期。

内存泄漏的第三个常见来源是监听器和其他回调。如果你实现了一个API，客户端在这个API中注册回调，却没有显示地取消注册，那么除非你采用某些动作，否则它们就会不断地堆积起来。

由于内存泄漏通常不会表现成明显的失败，所以它们可以在一个系统中存在很多年。往往只有通过仔细检查代码，或者借助于Heap分析工具才能发现内存泄漏问题。

第八条：避免使用终结方法和清除方法

终结方法(finalizer)通常是不可预测的，也是很危险的，一般情况下是不必要的。在Java 9中用清除方法代替了终结方法。清除方法没有终结方法那么危险，但仍然是不可预测，运行缓慢，一般情况下也是不必要的。

在Java中，当一个对象变得不可到达的时候，垃圾回收器会回收与该对象相关联的存储空间。

终结方法和清除方法的缺点在于不能保证会被及时执行。从一个对象变得不可到达开始，到它的终结方法被执行，所花费的时间是任意长的。注重时间的任务不应该由终结方法或者清除方法来完成。

终结方法线程的优先级比该应用程序的其他线程的优先级要低的多。Java语言规范不仅不保证终结方法或者清除方法会被及时地执行，而且根本就不保证它们会被执行。当一个程序中止的时候，某些已经无法访问的对象上的终结方法却根本没有被执行，这完全有可能，结论是：永远不应该依赖终结方法或者清除方法来更新重要的持久状态。

不要被System.gc和System.runFinalization这两个方法所诱惑，它们确实增加终结方法或者清除方法被执行的机会，但是它们并不保证终结方法或者清除方法一定会被执行。

使用终结方法和清除方法有一个非常严重的性能损失。

1. ：try-with-resources优先于try-finally

必须实现AutoCloseable接口，其中包含了单个返回void的close方法。

try(FileInputStream fis = new FileInpuptStream(file)){

}catch(Exception e){}

在处理必须关闭的资源时，始终要优先考虑用try-with-resources,而不是用try-finally.这样得到的代码将更加简洁，清晰，产生的异常也更有价值。

1. 对于所有对象都通用的方法

第十条：覆盖equals时请遵守通用约定

类的每个实例本质上都是唯一的。

类没有必要提供“逻辑相等”的测试功能。

超类已经覆盖了equals，超类的行为对于这个类也是合适。

类是私有的，或者是包级私有的，可以确定它的equals方法永远不会被调用。

在覆盖equals方法实现等价关系

自反性:对于任何非null的引用x，x.equals(x)必须返回true。（对象必须等于其自身）

对称性:对于任何非null的引用x和y，当且仅当y.equals(x)返回为true，并且x.equals(y)也必须返回true。

传递性:对于任何非null的引用x,y和z，如果x.equals(y)返回true，y.euqals(z)返回true，那么x.euqals(z)也必须返回true。

一致性:对于任何非null的引用值x和y，只要equals的比较操作在对象中所用的信息没有被修改，多次调用x.euqals(y)就会一致的返回true，或者一致地返回false。（相等的对象永远相等，不相等的对象永远不相等）

非空性:对于所有的非null的引用值x，x.equals(null)必须返回false。(所有的对象都不能等于null。尽管很难想象在什么情况下o.equals(null)调用会意外返回true，但是意外爬抛出NullPointException异常的情形却不难想像)

一旦违反了equals约定，当其他对象面对你的对象时，你完全不知道这些对象的行为会怎么样。

里氏替换原则认为，一个类型的任何重要属性也将适用于它的自类型，因此为该类编写任何方法，在它的自类型上也同喜那个运行的很好。

综合这些要求，得出了以下实现高质量equals方法的窍诀

1）使用==操作符检查“参数是否为这个对象的引用”

2）使用instanceof操作符检查“参数是否为正确的类型”。

3）把参数转换为正确的类型。因为转换之前进行过instanceof测试，所以确保会成功。

4）对于该类中的每个“关键”域，检查参数中的域是否与该对象中对应的域相匹配。

对于既不是float和double类型的基本类型域，可以使用==操作符进行比较;对于对象引用域，可以递归地调用equals方法;对于float域，可以使用静态Float.compare(float, float)方法，对于double域，则使用Double.compare(double, double)。对float和double域进行特殊的处理是有必要的，因为存在着Float.NaN, -0.0f以及类似的double常量: