第13条：使类和成员的可访问性最小化

-->将API与它的实现清晰的隔离开来-->解除系统的各个模块之间的耦合关系-->实现这些模块可以独立开发、测试、优化、使用、理解和修改。

使类的可访问性最下化的规则：
1.对于顶层的（非嵌套类）的类和接口，只有两种可能的访问级别：包级私有的和公有的
2.如果一个包级私有的顶层类（或者接口）只是在某一个类的内部被用到，就应该考虑使用它成为唯一使用它的那个类私有嵌套类。这样可以将它的访问范围从包中的所有类缩小到使用它的那个类。

对于成员有四种访问级别：
1.私有的，只有在声明该成员的顶层内部才可以访问这个成员。
2.包级私有的，声明该成员的类的包内部的任何类都可以访问这个成员。从技术上讲，它被称为“缺省访问级别”，如果没有为成员指定访问修饰符，就采用这个访问级别。
3.受保护的，声明该成员类的子类可以访问这个成员（但有一些限制），并且，声明该成员的包内部的任何类可以访问这个成员。
4.公有的，在任何地方都可以访问该成员。

使成员可访问性最小化的规则：
1.对于公有的类的成员，当访问级别变成包级私有时，会大大增强访问性。受保护的成员是类的导出的API的一部分，必须永远得到支持。导出的类的受的受保护成员也代表了该类对于某个实现细节的公开承诺。受保护的成员应该少量使用。
2.如果子类方法覆盖了超类中的一个方法，子类的访问级别就不允许低于超类中的访问级别。如果一个类实现了一个接口，那么接口中所有的类方法在这个类中也都必须声明为公有的。
3.不能为了测试，而将类、接口、或者成员变成包的导出的API的一部分。
4.实例域决不能是公有的。如果域是非final的，或者是一个指向可变对象的final引用，那么一旦使这个域变成公有的，就放弃了对存储在这个域中的值进行限制的能力。
5.如果类具有公有的静态final数组域，应该修改为：
51.可以使公有数组变成私有的，并增加一个公有的不可变列表。

5.2可以使数组变成私有的，并添加一个公有方法，它返回私有数组的一个备份。

第14条：在公有类中使用访问方法而非公有域

公有类永远不要暴露可变的域：

1.让公有类暴露不可变的域其危害比较小。

2.有时候需要包级私有的或者私有的嵌套类来暴露域，无论这个类是可变的还是不可变的。这样，改变域的作用范围被进一步限制在外部类中。

第15条：使可变性最小化

不可变的类比可变类更加易于设计、实现和使用。

为了使类成为不可变，要遵循5条规则：

1.不要提供任何会修改对象状态的方法。

2.保证类不会被扩展，防止子类化，破坏类的不可变行为。

3.使所有的域都是final的。

4.使所有的域称为私有的。

5.确保对于任何可变组件的互斥访问。

不可变对象的优点：

1.不可变对象本质上是线程安全的，它们不要求同步。

2.不可变对象可以被自由的共享，永远也不要进行保护性拷贝。

3.不仅可以共享不可变对象，甚至也可以共享它们的内部信息。

4.实际应用中，不可变对象为其他对象提供了大量的构件。

不可变对象的缺点：

不可变对象真正唯一的缺点是，对于每个不同的值都需要一个单独的对象。创建这种对象的代价可能很高，特别是对于大型对象的情形。

优化：

1.如果能够精确地预测客户端将要在不可变的类上执行哪些复杂的多阶段操作，可以提供一个包级私有的可变配套类。如果无法预测，最好的办法是提供一个公有的可变配套类。

2.为了提高性能，许多不可变的类拥有一个或者多个非final的域，它们在第一次被

请求执行这些计算的时候，把一些开销昂贵的计算结果缓存在这些域中。

第16条：复合优先于继承

对于专门为了继承而设计，而且具有很好的文档说明的类来说，使用继承是非常安全的。然而，对于普通的具体类进行跨越包边界的继承，则是非常危险的。
为什么？
继承打破了封装性，子类依赖于其超类中特定的实现细节。超类的实现可能会随着发行版本的不同而有所变化，如果真的发生了变化，子类可能遭到破坏，即使它的代码完全没有改变。

复合：新的类中增加一个私有域，它的引用指向现有类的一个实例。
转发：新类中的每个实例方法都可以调用被包含的现有类实例中对应的方法，并返回它的结果集。

只有当子类真正是超类的子类型时，才适合用继承。

第17条：要么为继承而设计，并提供文档说明，要么就禁止继承

好的API文档应该描述一个给定的方法做了什么工作，而不是描述它是如何工作的。
为了允许继承，类必须遵守一些约束：
1.类必须通过某种形式提供适当的钩子，以便能够进入到它的内部工作流程中，这种形式可以是精心选择的受保护的方法。
2.对于为了继承而设计的类，唯一的测试方法就是编写子类。
3.构造器决不能调用可被覆盖的方法。
4.无论是clone还是readObject，都不可以调用可覆盖的方法。
5.如果你认为必须允许从这样的类继承，一种合理的办法是确保这个类永远不会调用它的任何可覆盖方法。

第18条：接口优先于抽象类

接口的优势：
1.现有类很容易被更新，以实现新的接口。
2.接口是定义mixin（混合类型）的理想选择。

3.接口允许我们构造非层次结构的类型框架。
接口的缺点：
接口一旦被公开发行，并且被广泛实现，再想改变这个接口几乎是不可能的。
抽象类的优势：
抽象类的演变比接口的演变要容易的多。如果在后续的发行版本中，你希望在抽象类中增加新的方法，始终可以可以增加具体的方法，它包含合理的默认实现。
于是：通过对你导出的每个重要接口都提供一个抽象的骨架实现类，把接口和抽象的优点结合起来。
如果预置的类无法扩展骨架实现，这个类始终可以手工实现这个接口，可以使用内部私有类扩展骨架实现类，这种方法称为模拟多重继承。

第19条：接口只用于定义类型

常量接口是对接口的不良使用
原因：内部的实现细节不应该出现在导出的API中。
如果要导出常量，可以有下面几种方案：
1.如果这个常量与现有的某个类和接口密切相关，就应该把这个类和接口添加到这个类或者接口中。
2.如果这些常量最好看作枚举类型的成员，就应该使用枚举类型来导出这些常量。
3.最后可以使用不可以实例化的工具类来导出这些常量。

第20条：类层次优先于标签类

标签类缺点：
标签类是类层次的一种简单仿效，将代码充斥在单个类中，代码冗长，可读性差，容易出错，并且效率低下。
可以考虑使用类层次来重构标签类。

第21条：用函数对象表示策略

1.如果一个类仅仅导出这样的方法，它的实例实际上等同于一个指向该方法的指针，这样的指针称为函数对象。
2.函数指针的主要用途就是实现策略模式。
3.当一个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的策略类，当一个具体策略是设计用来重复使用时候，
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域来导出，

其类型为该策略接口。

第22条：优先考虑静态成员类

静态成员类、非静态成员类、匿名类、局部类
静态成员类与非静态成员类的区别：
1.静态成员类的声明中包含修饰法static
2.非静态成员类实例都隐含着一个外部类的实例引用，这种隐含关系需要消耗非静态成员类实例的空间，并增加构造的时间开销。
如果嵌套的实例可以在它外部类的实例之外独立存在，这个嵌套类就必须是静态成员类；在没有外部类的情况下，要想创建非静态成员类的实例是不可能。
3.静态成员类的一种最常见的用法是作为公有辅助类，非静态成员类的一种最常见用法是定义一个Adapter。

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。

匿名类：
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使用，而是在使用的时候被声明和实例化。
当且仅当匿名类出现在非静态的环境中时，它才拥有外围实例。即使出现在静态的环境中，也不可能拥有静态成员。

匿名类受到的限制：
1.除了在它们的被声明的时候之外，是无法将它们实例化的。
2.你不能执行instanceof测试，或者需要任何要命名类的其他事情。
3.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。