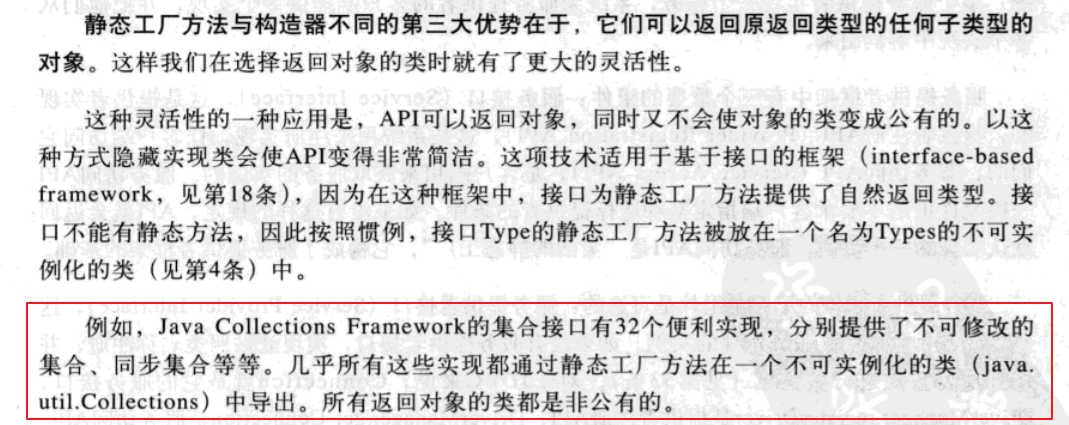
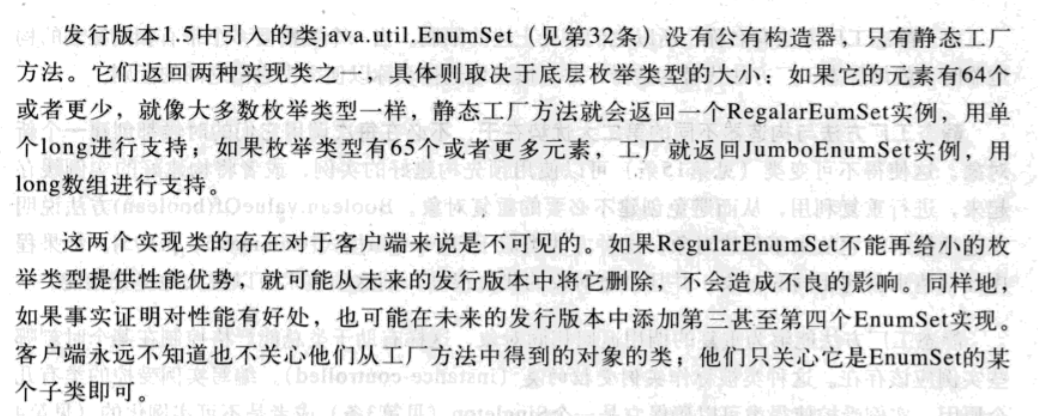
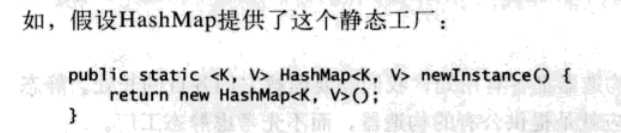
### 静态工厂方法：

优点：

* 与构造器相比，有名称；
* 与构造器相比，不必每次都创建新对象；
* 与构造器相比，可以返回原返回类型的任何子类型；（如：java集合框架的Collections类）这样对于那些非公开的子类，后期版本做更改后也不会影响到用户，比如：
* 与构造器相比，创建参数化类型实例时，代码更加简洁——这一点好像在jdk8中，并没有区别（），或者是jvm版本原因

**书中举例**：使用Map<String, List<String>> map = HashMap.newInstance()代替Map<String, List<String>> map = new HashMap<String, List<String>>()，理由是静态工厂方法可以进行类型推导（值得推荐）

缺点：

* 类如果不包含公有的或受保护的构造器，就不能被子类化；
* 名字不好找，不像构造器那样被特别标注；一般可以遵守默认命名规则：valueOf/of（一般用于类型转换）、getInstance、getType/newType（当静态工厂方法在不同类中时）等；

### **多参数时，考虑用构建器（builder）**

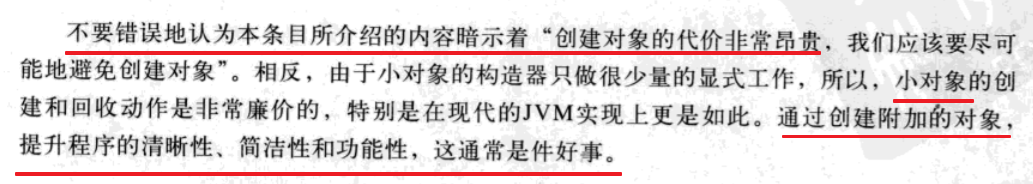
* 重叠构造器模式：参数少时，比较好；参数太多后，果断放弃；
* JavaBeans模式：缺点-使保证bean一致性变得困难；
* Builder模式：builder是它构建类的静态内部类，可先通过builder的setter方法设置属性，然后调用builder.build()方法，构建对象；

### **枚举实现单例模式**

* 抵御通过反射机制生成第二实例的方法：构建第二实例的时候抛出异常
* 单例模式中，如果类是可序列化的（实现Serializable接口），必须重写readResolve方法，不然，每次反序列化都会产生一个实例；
* 单元素的枚举类是最好的实现单例模式的方法——既可防止反射攻击，也可防止反序列化产生多实例

### **使用私有构造器强化不可实例化能力**

### **避免创建不必要的对象**

* String a = new String(“string”); 此句创建了2次实例：参数“string”就是一个实例
* 优先使用基本类型，而不是封装类型
* 有时候重用对象会导致代码很乱，逻辑糟糕，比重建对象的代价更大

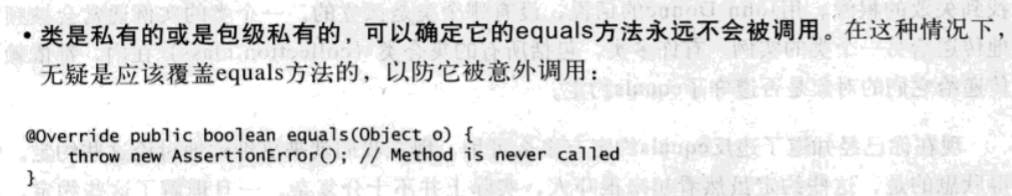
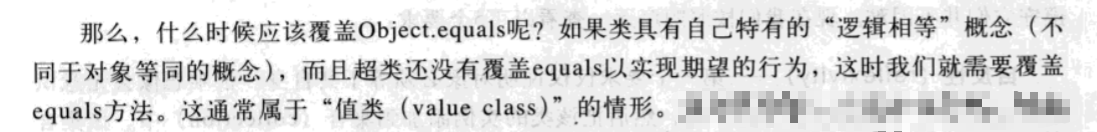
### **及时清除过期引用**

* 缓存、监听器/其他回调都比较容易发生内存泄漏

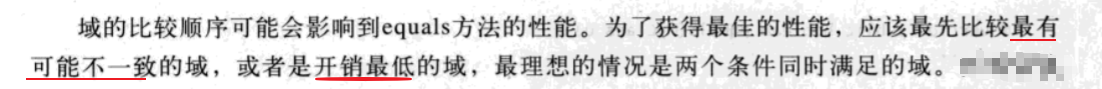
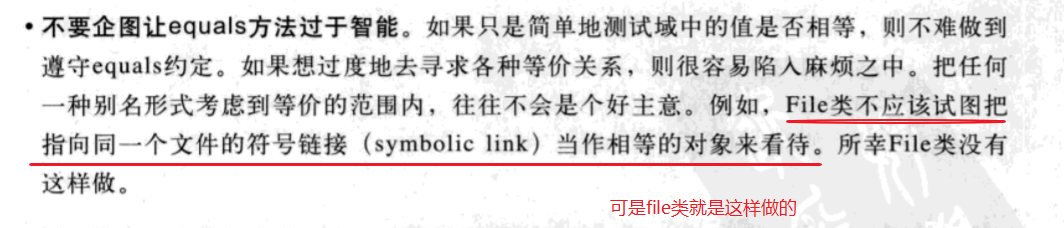
### **避免使用终结方法**

* 从一个对象变得不可到达开始，到它的终结方法执行，所花费的时间是任意长的；所以，类似在终结方法中关闭文件的做法的错误的。
* 何时执行终结方法也是垃圾回收算法的一个功能，而垃圾回收算法在不同的jvm实现中会大相径庭，如果以来finallizer，那么不同jvm中实现会截然不同。
* 有时候finallizer是否执行都不能保证：程序终止，而finallizer方法却没执行。
* 不要被System.runFinalizersOnExit()和Runtime.runFinalizersOnExit()诱惑，它们都有致命缺陷（多线程情况）；
* Finallizer中的异常不会被打印，容易被忽略；
* Finallizer增加性能损耗；
* 建议使用try...finally
* 子类如果重写了终结方法（finallizer），则必须再调用超类的终结方法；**终结方法守卫者**可以防止粗心大意而没有执行super.finallizer

### **覆盖equals方法的通用约定**

* 类的每个实例实质上都是唯一的；
* 不关心类是否提供“逻辑相等”的测试功能；
* 超类的equals方法也适合子类；
* 不明白：
* 什么时候需要覆盖equals方法：对于枚举类，逻辑相等和对象相等时一个意思，所以没必要覆盖equals
* 覆盖equals需要遵守几个特性：自反性、对称性、传递性、一致性、以及null
* **氏替换原则**简单粗暴的理解：任何基类可以出现的地方，子类一定可以出现。
* Timestamp类（Date的子类，增加了nanoseconds域）和Date类不要混合使用，混合情况下回违反equals的自反性
* Equals优化：

1. 使用“==”检查对象引用
2. 使用instanceof检查类型
3. 把参数转化成正确的类型（如：date转成long）

* 
* 重写equals的时候也要重写hashcode
* 不要将equals(Object obj)中的Object替换为其他类型（如：MyClass），这样就不是重写了，而是重载；添加@Override可以避免
* 尴尬：

### 重写equals的时候也要重写hashcode

* Hashcode也有一致性
* 如果equals返回true，那么hashcode也要相等，反之，不一定，但是不equals的对象，返回不同的hash值，有可能提高hash性能；
* 如果一个类是不可变的，并且计算hash码的开销也比较大，应该考虑将hash值缓存起来
* 要慎重考虑计算hash值得时候舍弃某些字段的得与失

### 最好子类都重写toString方法

如果tostring方法用于持久化，那么请确定长久规范

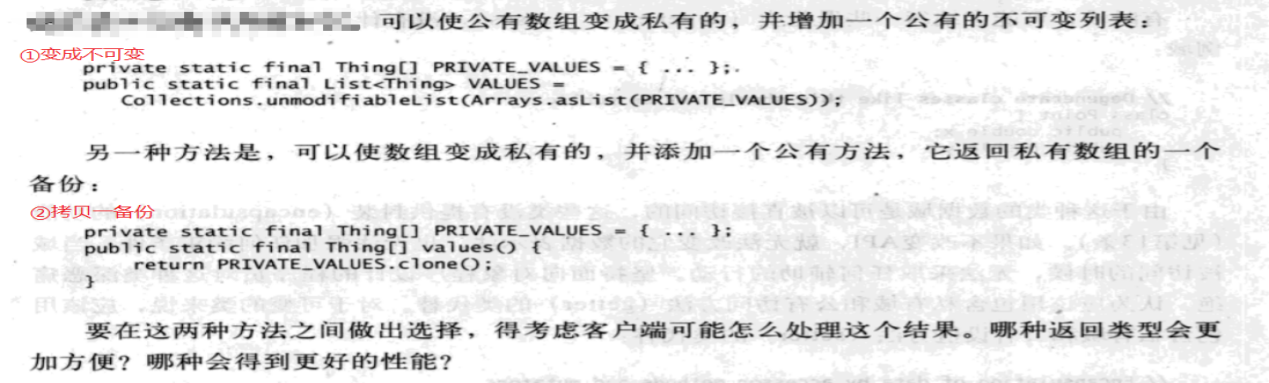
### 谨慎重写clone

* 数组上调用clone返回的数组类型和原类型一样
* clone结构与指向可变对象的final域的正常用法是不兼容的（文中例子是一个数组elemnts的clone）
* 如果一个专为继承设计的类重写了clone方法，那么应该效仿object.clone：声明为protected、抛出CloneNotSupportedException异常、不能实现Cloneable，留给子类选择的空间
* 线程安全的类，也要保证clone方法和其他方法一样——线程安全

### 考虑实现Comparable

* 如果创建的类是一个值类，具有明显的内排序，就应该坚定地实现Comparable接口
* compareTo和equals不需要必须等效，比如：BigDecimal(“1.0”)和BigDecimal(“1.00”);但是如果使用treeSet之类的集合，则只算一个元素

### 类和成员可访问性最小

* 最大透明度，称为信息隐藏或封装，软件设计原则之一
* 好处：解耦（开发、理解、测试、维护都比较容易）
* 实例的域决不能是公有的
* Final域应当只包含基本类型的值或不可变对象的引用；
* final修饰的数组几乎总是错误的，解决这种矛盾的方法有2种：

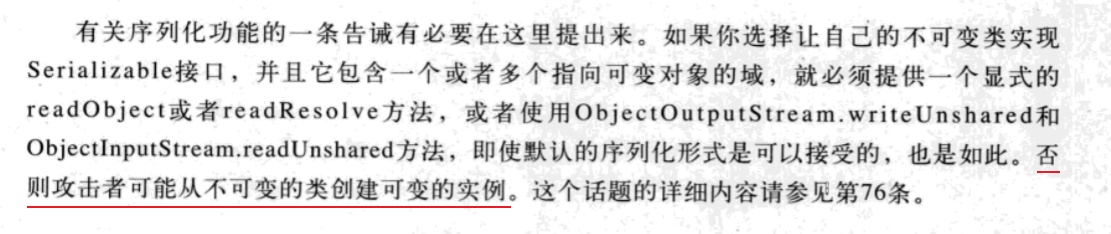
### 避免直接访问域

* 如果公有类暴露了它的数据域，要想在将来改变其内部表示法是不可能的

### 使可变性最小化

* 不可变类：实例化后不再改变——string、基本类型的包装类、BigInteger、BigDecimal等
* 不可变类更加容易设计、实现和使用，不易出错，更加安全
* 设计不可变类原则：

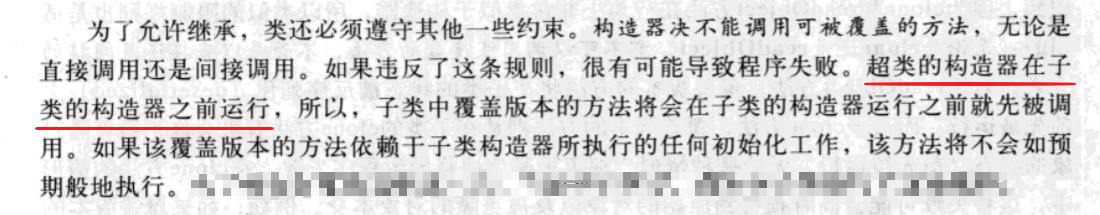
1. 不提供修改状态的方法；
2. 保证类不被扩展（final）
3. 所有域设为private
4. 所有域设为final
5. 避免引用可变组件（如其他可变类引用）

* 对于不可变类，本质上就没有拷贝的必要，所以是实现clone是不必要的，String就是反面教材
* 不可变类会造成性能的浪费（MutableBigInteger就是BigInteger的性能优化版）
* 

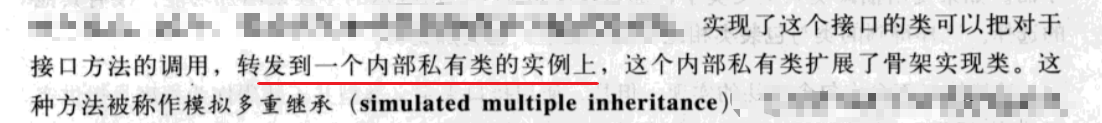
### **复合（composition）优先于继承：**

继承比较脆弱，如果新版本中添加新的方法很可能对子类造成很大影响，导致不稳定。复合不存在这种问题，复合类似于适配器模式

### **要么为继承而设计，并提供文档说明，要么禁止继承**

* 关于文档：好的api文档应该描述一个给定的方法做了什么，而不是如何做的
* 构造器决不能调用可被重写的方法
* 为继承而设计的类，应该慎重考虑实现Cloneable和Serializable接口

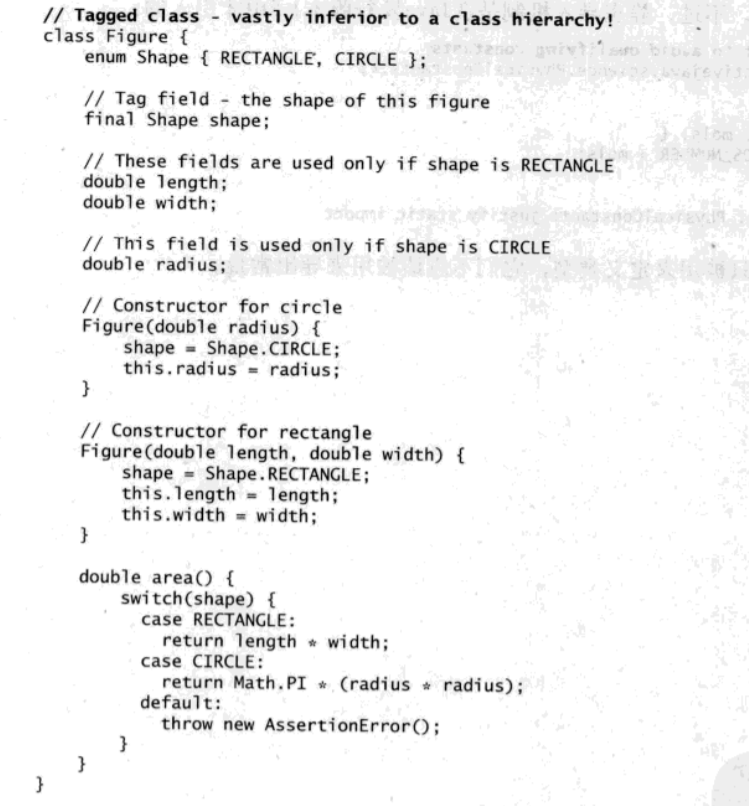
### **接口优于抽象类**

* 现有类易被更新，以实现新的接口；比如jdk添加Comparable接口的时候
* 接口是定义mixin（混合类型）的理想选择
* 接口定义类型，抽象类（一般命名AbstractXXX，如AbstractList）搭建骨架
* 公有接口的设计一定要谨慎，一旦公开发行，并被广泛实现，再想修改接口，几乎是不可能的
* 接口实现起来比抽象类灵活，但设计了接口，最好定义一个骨架（抽象类）

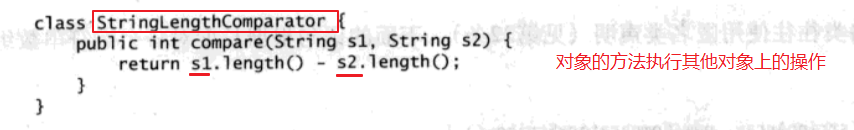
### **接口只用来定义类型**

* 常量接口（只有final的静态域）是对接口的不良使用

### **类层次优于标签类**

* 标签类（没有具体定义，文中给了个列子，如下图）：标签类过于冗长，易出错，且效率低下

### **用函数对象表示策略**

* 函数对象：

### **优先考虑静态成员类**

**唠唠**：

* 永远不要让客户去做任何类库能够替客户完成的事