《EffectiveJava》学习笔记

**1.考虑静态工厂方法代替构造器：**

优势：

1）.有名称；

2）.不必在每次调用它们的时候都创建一个新对象；

3）.可以返回原返回类型的任何子类型的对象；

4）.在创建参数化类型实例的时候，它们使代码变得更加简洁。

缺点：

1）.类如果不含公有的或者受保护的构造器，就不能被子类化；

2）.它们与其他静态方法实际上没有区别。















**2.遇到多个构造器参数时要考虑用构建器：**

1）、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，并且难以阅读；

2）、JavaBean模式也可行，但是有个很严重的缺点，JavaBean模式阻止了把类做成不可变的可能，因此线程不安全；

3）、因此第三种方式，既可以保证像重叠构造器模式那样的安全性，也能保证像JavaBean模式那么好的可读性，也就是Builder模式。

Builder模式：

不直接生成想要的对象，而是让客户端利用所有必要的参数调用构造器，得到一个builder对象，然后客户端在builder对象上调用类似setter的方法，来设置每个相关的可选参数，最后客户端调用无参的builder方法生成不可变的对象。







**3.用私有构造器或者枚举类型强化Singleton属性：**

Singleton指仅仅被实例化一次的类。在jdk1.5前有多中方式实现单例模式，例如静态工厂方法、静态内部类等，但是它们各有弊端，比如线程问题或者反射问题，当然，抵御反射攻击时，可以修改构造器，让它在被要求创建第二个实例的时候抛出异常。



jdk1.5之后，单元素的枚举类型已经成为实现Singleton的最佳方法，既线程安全，又可避免反射攻击，同时代码也很优雅。

**4.通过私有构造器强化不可实例化的能力**

一些只包含静态方法或者静态域的类，比如工具类等，我们并不希望实例化该类，因为实例化并没有意义，此时就可以显示的私有化构造器来禁止类被实例化。但是也有副作用就是不能被子类继承了。

**5.避免创建不必要的对象**

一般来说，最好能重用对象而不是在每次需要的时候就创建一个相同功能的新对象。重用方式既快速又流行。如果对象是不可变的，它就始终可以被重用。

对于同时提供了静态工厂方法和构造器的不可变类，通常可以使用静态工厂方法而不是构造器，以避免创建不必要的对象。

除了重用不可变的对象之外，也可以重用那些已知不会被修改的可变对象。

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。

由于小对象的构造器只做很少量的显示工作，所以，小对象的创建和回收动作是非常廉价，特别是现代的JVM实现上更是如此。

**6.消除过期的对象引用**

过期引用：指永远也不会被解除的引用。

消除对象引用应该是一种例外，而不是一种规范行为。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。

只要类是自己管理内存，程序猿就应该警惕内存泄漏问题。

内存泄漏的另一个常见来源是缓存。





内存泄漏的第三个常见来源是监听器和其他回调。

**7.避免使用终结方法**

终结方法通常是不可预测的，也是很危险的，一般情况下是不必要的。使用终结方法会导致行为不稳定，降低性能，以及可移植性问题。

**8.覆盖equals（）方法时要遵守通用约定**

如果满足了以下任一条件，就可不用覆盖equals（）方法：

1）、类的每个实例本质上是唯一的。

2）、不关心类是否提供了“逻辑相等”的测试功能。

3）、超类已经覆盖了equals，从超类继承过来的行为对于子类是合适的。

4）、类是私有的或是包私有的，可以确定它的equals方法永远不会被调用。

如果类具有自己特有的“逻辑相等”概念（不同于对象等同的概念），而且超类还没有覆盖equals以实现期望的行为，这时就需要覆盖equals方法。这通常属于“值类”的情形。值类仅仅是一个表示值的类，例如Integer和Date。

有一种“值类”不需要覆盖equals方法，即用实例受控确保“每个值至多只存在一个对象”的类。枚举类型就属于这种类。对于这样的类，逻辑相同与对象等同是一回事，因此Object的equals方法等同于逻辑意义上的equals方法。

equals方法实现了等价关系：

1）、自反性：对于任何非null的引用值x，x.equals（x）必须返回true。

2）、对称性：对于任何非null的引用值x、y，当且仅当y.equals（x）返回true时，x.equals（y）必须返回true。



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

4）、一致性：对于任何非null的引用值x、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.equals（y）就会一致地返回true，或者一致的返回false。

5）、非空性：对于任何非null的引用值x，x.equals（null）必须返回false。

里氏替换原则：一个类型的任何重要属性也将适用于它的子类型，因此为该类型编写任何方法，在它的子类型上也应该同样运行得很好。

实现高质量equals方法的诀窍：

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）、把参数转化为正确的类型；

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；

5）、当编写完成了equals方法之后，应该问自己三个问题：它是否是对称的、传递的、一致的？并编写单元测试进行检验。

下面是String类的equals方法：



实现equals方法时要注意的问题：

1）、覆盖equals方法时总要覆盖hashCode方法；

2）、不要企图让equals方法过于智能；

3）、不要将equals声明中的Object对象替换为其他的类型。

**9.覆盖equals（）方法时总要覆盖hashCode方法**

在每个覆盖了equals方法的类中，也必须覆盖hashCode方法。

**hashCode规范：**

**1）、在应用程序的执行期间，只要对象的equals方法的比较操作所用的的信息没有被修改，那么对这同一个对象调用多次，hashCode方法都必须始终如一地返回同一个整数。在同一个应用程序的多次执行过程中，每次执行所返回的整数可以不一致。（也就是说在应用程序一次的执行期间，相同对象的hashCode值一直相同）**

**2）、如果两个对象根据equals方法比较是相等的，那么调用这两个对象中任意一个对象的hashCode方法都必须产生同样的整数结果。（也就是说相同对象的hashCode值一直相等。但是不同对象的hashCode值不一定不相等，比如String类型的相同值。）**

**3）、如果两个对象根据equals方法比较是不相等的，那么调用这两个对象中任意一个对象的hashCode方法时，则不一定要产生不同的散列码。但是，给不同的对象产生不同的整数结果，有可能提高散列表的性能。（比如HashMap类）**

**hashCode方法的解决方案：**

**1）、把某个非零的常数值，例如17，保存在一个名为result的int类型的常量中；**

**2）、对于对象中每个关键域f（指equals方法中涉及的每个域），完成以下步骤：**

**a、为该域计算int类型的散列码c：**

**Ⅰ、如果该域是boolean类型，则计算（f？1：0）；**

**Ⅱ、如果该域是byte、char、short或者int类型，则计算（int）f；**

**Ⅲ、如果该域是long类型，则计算（int）（f^（f>>>32））；**

**Ⅳ、如果该域是float类型，则计算Float.floatToIntBits（f）；**

**Ⅴ、如果该域是double类型，则计算Double.doubleToLongBits（f），然后按照步骤2.a.Ⅲ，为得到的long类型值计算散列值；**

**Ⅵ、如果该域是一个对象引用，并且该类的equals方法通过递归地调用equals方式来比较这个域，则同样为这个域递归地调用hashCode。如果需要更复杂的比较，则为这个域计算一个“范式”，然后针对这个范式调用hashCode，如果这个域的值为null，则返回0；**

**Ⅶ、如果该域是一个数组，则要把每一个元素当做单独的域来处理，也就是说，递归地应用上述规则，对每个重要的元素计算一个散列值，然后根据步骤2.b中的做法把这些散列值组合起来。如果数组域中的每个元素都很重要，可以利用发行版本1.5中增加的一个Arrays.hashCode方法。**

**b、按照下面的公式，把2.a中计算得到的散列码c合并到result中：**

**result = 31 \* result + c。**

**3）、返回result。**

**注意：在散列码的计算中，可以把冗余域排除在外，换句话说，如果一个域的值可以根据参与计算的其他域值计算出来，则可以把这样的域排除在外。必须排除equals比较计算中没有用到的任何域，否则很有可能违反hashCode约定的第二条。**

下面是String类的hashCode方法：



**10.始终要覆盖toString（）方法**

toString的约定进一步指出：建议所有的子类都覆盖这个方法。

当对象被传递给println、printlf、字符串联操作符（+）以及assert或者被调试器打印出来时，toString方法会被自动调用。