1、考虑用静态工厂方法代替构造器。

静态工厂方法是类提供的用于创建对象的方法，它和构造器相比具有以下几个优势：

（1）有名称。如果构造器的参数本身没有确切地描述正被返回的对象，那么具有适当名称的静态工厂方法会更容易使用。例：BigInteger(int,int,Random)返回一个BigInteger对象，这个对象的值可能是素数。如果我们想获取一个值为素数的BigInteger对象，利用BigInteger.probablePrime()方法显然更加清楚。

（2）不必在每次调用的时候都创建一个新对象。利用静态工厂方法创建对象时可以实现对象缓存，或是使用事先创建好的对象。如果程序经常请求创建完全相同的对象，那么这项技术可以极大地提升性能。如果程序中不存在完全相同的两个对象这一点得到了保证，那么就可以利用==代替equals()方法，这样能简化客户端代码的编写，提升程序性能。一个典型的应用就是enum类型。

（3）可以返回原返回类型的任何子类型的对象。一种应用是，方法可以返回对象，同时又不会让对象的类变成公有的。Collectons中的如[synchronizedMap](mk:@MSITStore:C:\\Users\\swt369\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Collections.html" \l "synchronizedMap(java.util.Map))、[unmodifiableMap](mk:@MSITStore:C:\\Users\\swt369\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Collections.html" \l "unmodifiableMap(java.util.Map))等都利用了这一点。对于用户而言只需要知道它们是一个Map即可正常使用。

静态工厂方法的主要缺点在于：

（1）类如果不含public或是protected的构造器，就不能被子类化。

（2）与其他的静态方法没有本质上的区别，在类的静态方法较多时会造成麻烦。这个缺点可以利用命名规范部分弥补。常用的静态工厂方法命名如下：

valueOf/of——返回的实例与参数具有相同的值，常用于类型转换。

getInstance——根据传入参数返回对象。

newInstance——与getInstance相同，但用该名称命名的方法应保证每次返回的都是新对象。

getType——一般用于工厂方法处于不同的类的情况下，其他与getInstance相同。

newType——一般用于工厂方法处于不同的类的情况下，其他与newInstance相同。

一般而言，静态工厂方法更加灵活，可操作性更强，因此在编写创建对象的方法时，应该首先想想使用静态工厂方法是否合适。

2、遇到多个构造器参数时要考虑用构建器。

当遇到构造器需要大量参数时，传统的做法有以下两种：

（1）利用重叠构造器。假设某个类Example的构造器需要A~F共7种参数，可以这样写：

**class** Example{

//省略A~F七个域

**public** Example() {

**this**(**null**);

}

**public** Example(A a) {

**this**(**a**,**null**);

}

**public** Example(A a,B b) {

**this(a,b,null);**

}

...

**public** Example(A a,B b,C c,D d,E e,F f,G g){

//省略this.A = A...等

}

}

这种做法在已知参数较多时极其容易导致编程错误（如将两个参数顺序写反），尤其是在构造器中有相同类型的参数时。

（2）利用JavaBean模式：

JavaBean模式规定如下：

1、所有属性为private。  
2、提供默认构造方法。  
3、提供getter和setter。  
4、实现serializable接口。

所有属性均利用setter设置，与构造器无关。这种做法的问题在于构造过程被分到了多个调用中，使得在构造过程中对象可能处于不一致的状态。如果试图使用不一致的对象，将会导致错误。并且JavaBean模式组织了将类做成不科比那的可能，需要付出额外的努力确保线程安全。

在这种情况下，最合适的方法是利用Builder模式，特别是许多参数都是可选参数的情况下。

3、通过私有构造器或者枚举类型强化Singleton属性。

基本上所有的实现单例模式的方法的第一步都是将类的构造器声明为私有的。这可以保证正常情况下外部无法直接new出一个新的对象。但是利用反射机制可以绕过这种限制。如果需要抵御这种攻击，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。

为了保证反序列安全，需要在实现Serializeable接口的同时，覆盖readResolve()方法：

**private** Object readResolve() **throws** ObjectStreamException{

**return** instance;

}

不考虑enum的额外性能消耗的前提下，用单元素的枚举类型实现单例模式是最佳方法。

4、通过私有构造器强化不可实例化的能力。

对于一些工具类（如Math、Collections、Arrays），实例对它没有任何意义，此时就可以将其构造器声明为private的。

不能通过将类声明为abstract的来阻止实例化，因为即使你这样做，还是可以通过子类来获取实例，而且这还有误导性，让人认为这个类是为继承而设计的。

私有构造器的副作用是阻止了子类化，因为所有构造器都必须显式或隐式地调用超类构造器，在这种情况下子类就没有可用的超类构造器了。

5、避免创建不必要的对象。

最经典的一个例子是String的使用。任何情况下都不要写出如下代码：

String str = **new** String("This is a string.");

而应该这样写：

String str = "This is a string.";

这是因为对于在同一台JVM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。

如果在某个方法中涉及到比较昂贵的不可变对象的创建，而这个方法的调用非常频繁，则最好将这些对象声明为对象（或类）的域，在构建对象（或加载类）时完成初始化，并在方法中复用。

在做数值计算的时候，除了必要的情况下，应尽可能地使用基本类型，避免创建不必要的包装类对象。

小型对象的复用在现代JVM的垃圾回收器的支持下往往意义不大，如果创建小型对象能够提升代码的准确度、清晰度，则应当放心地使用。

6、消除过期的对象引用。

JVM的垃圾回收器会自动处理掉不再被引用的对象，但这不代表Java程序中不存在内存泄漏问题。在有些情况下，某些对象虽然不再会被使用了，但是仍有其他对象持有它的引用（过期引用），在这种情况下它就得不到回收。这种情况常出现在容器类对象（如各种缓存池）中。

虽然不手动清除引用可能会造成以上问题，但在日常编程中不应当将清空对象引用作为一种规范，而作为一种特殊情况下的手段来使用，因为大量的清除引用语句会使得代码变得很乱。清除过期引用最有效的方法是让所有对象都在它的最紧凑的作用域内被创建。一般来说，当程序中出现以下情况的时候才需要考虑手动清除引用：

（1）有大量的对象不是在最紧凑的作用域内声明的。

（2）缓存。如果缓存项的生命周期是由它的键的外部引用决定的，那么可以使用WeakHashMap实现缓存。

（3）监听器和其他回调。

7、避免使用终结方法。

Java中的终结方法finalize()不是C++中析构函数的替代品。Java语言规范不仅不保证终结方法会及时地得到执行，甚至不保证它一定会得到执行。因此，如果将资源清理放在终结方法中，非常容易导致程序运行失败（OOM等）。如果确实需要终止线程或关闭文件等，应当提供一个显式的方法（如close()），并要求客户端在需要的时候调用这些方法。

显式的终止方法通常放在try-finally块的finally部分中，以确保及时终止。

终结方法合理的使用场合有这么两个：一是对象极其持有关键资源，必须得到释放，此时可以编写终结方法作为安全网。二是调用了本地方法，而本地方法可能是由其他语言可直接操作内存的语言（如C++）编写的，那么需要在finalize()方法中进行处理。

如果类有终结方法，而其子类需要覆盖终结方法，那么子类应当在它的终结方法的try块中放置子类自己的终结方法，而在finally块中放置超类的终结方法。需要注意的是Java没有任何机制保证这个终结方法链的完整性，如果对这一点不放心，可以在类中增加一个附加对象：

**private** **final** Object finalizerGuardian = **new** Object(){

@Override

**protected** **void** finalize() **throws** Throwable {

...

}

};

8、实现equals方法时请遵守通用约定。

如果一个类没有实现equals()方法，并且也没有从父类继承得到的equals()方法，那么它的equals()方法会是从Object()继承得到的，比较的是两个对象在内存中的地址。

在以下几种情况下可以不覆盖equals()方法：

（1）类的每个实例本质上都是唯一的。

（2）不关心类是否提供了“逻辑相等”的测试功能。

（3）父类已经实现了equals方法，而且对子类同样有效。

（4）类是私有的或是包级私有的，可以保证它的equals()方法永远不会被调用。这种情况下也可覆盖equals()方法，以防意外调用：

@Override

**public** **boolean** equals(Object obj) {

**throw** **new** AssertionError();

}

equals()方法编写指南：

只选取有价值的域参与比较。正确的equals()方法应满足以下条件：

（1）自反性。x.equals(x) == true;

（2）对称性。x.equals(y) == y.equals(x);

（3）传递性。x.equals(y) == true && y.equals(z) == true

→ x.equals(z) == true;

（4）一致性。只要x和y用于等价比较的域未改变，那么x.equal(y)的结果不变。

（5）x.equals(null) == false。

实现方法：

（1）首先使用==操作符检查“参数是否为这个对象的引用”，如果是则返回true。

（2）使用instanceof检查“参数是否为正确的类型”。如果不是，则返回false。这里的“正确的类型”一般是equals()方法所在的类，特殊情况下也可是父类或是实现的接口。这一步同时检查了参数是否为null（null instanceof 任何class == false）。

（3）把参数转换成正确的类型。

（4）对于类中每个关键域，检查参数中的域是否与该对象中对应的域相匹配。这里需要注意的是，有些域是null可能是合法的，为了避免NullPointerException异常，可以用这样的判断语句：

(field == **null** ? o.field == **null** : field.equals(o.field))

9、覆盖equals时总要覆盖hashCode

在每个覆盖了equals方法的类中，也必须覆盖hashCode方法。如果不这样做的话，就会违反Object.hashCode的通用约定，从而导致该类无法结合所有基于散列的集合一起正常运作。

hashCode()方法编写指南：

hashCode()方法应满足以下条件：

（1）只要对象的变化不会影响到equals()方法的结果，那么hashCode()方法的结果不变；

（2）x.equals(y) == true → x.hashCode() == y.hashCode()；

（3）x.equals(y) == false → x.hashCode() =/!= y.hashCode()。

简而言之就是等价的对象散列码必须相同，不等价的对象散列码不一定不同。

一般根据以下步骤计算散列码值：

（1）int result = 某个非零常量，推荐17；

（2）根据以下方法计算每个有意义的域var的散列码c：

|  |  |
| --- | --- |
| 重要字段var的类型 | 生成的hash分量 |
| byte, char, short ,int | C=(int)var |
| Long | C=(int)(var ^ (var >>> 32)) |
| boolean | C=var?1:0 |
| float | C= Float.floatToIntBits(var) |
| double | long bits = Double.doubleToLongBits(var);  C= (int)(bits ^ (bits >>> 32)); |
| 引用类型 | C=(null == var ? 0 : var.hashCode()) |
| 容器 | C=所有元素hashCode()的和 |

注：

<<:左移运算符，num << 1,相当于num乘以2

>>:右移运算符，num >> 1,相当于num除以2

>>>:无符号右移，忽略符号位，空位都以0补齐

（3）对每个c执行以下计算：result = 31 \* result + c

（4）result即为最终的hashCode()返回值。

如果类是不可变的，并且常被用作Map的键，可以考虑用延迟初始化的方法缓存hashCode的结果。

10、始终要覆盖toString

默认的toString方法返回的是类名@散列码（无符号十六进制表示）。

toString方法会被用在如print、printf、字符串联操作符（+）等等场合。

一个好的toString方法应当是一个“简洁的，但信息丰富，并且易于阅读的表达形式”。

在代码中添加关于toString方法输出的字符串的格式的说明。

11、谨慎地覆盖clone

如果想要对象具有拷贝自身的功能，就应当实现Cloneable接口，并覆盖clone方法。

clone方法通用约定：

x.clone() != x

x.clone().getClass() == x.getClass()

x.clone().equals(x)

1、由于x.clone()与x不是同一个对象的引用，因此x.clone() != x；

2、拷贝前后的两个对象必然是同一个类的实例，因此x.clone().getClass() == x.getClass()；

3、需要提前实现equals()方法。由于新对象是原对象的拷贝，因而x.clone().equals(x) == true。

如果覆盖了非final类的clone方法，则应该返回一个通过super.clone而得到的对象，并将其转换成正确的类型。

Object的clone方法返回的对象是原对象的逐域拷贝，对于引用类型直接复制引用，而不会重新创建对象。因此必须考虑浅拷贝与深拷贝的问题，防止拷贝出来的对象对原对象造成破坏。

12、考虑实现Comparable接口

一个类实现了Comparable，就表明它的实例具有内在的排序关系。对于实现了Comparable接口的对象数组排序只需使用Arrays.sort()或是Collections.sort()。

compareTo方法的通用约定：将这个对象与指定的对象进行比较，当该对象小于、等于或大于指定对象时，分别返回一个负整数、零或是一个正整数。如果由于指定对象的类型而无法与该对象进行比较，则抛出ClassCastException异常。

对于具有多个关键域的对象的比较，应当从最关键的域开始，如果不相同则返回这个域的compareTo的结果，一直到全部比较完成，类似于字典序。

一般情况下应当保证(x.equals(y) == true) == (x.compareTo(y) == 0)，如果不是则应注释说明。

当用某两个域相减的结果作为compareTo的返回值时，需要小心溢出问题，因此这种方法不推荐。

13、使类和成员的可访问性最小化。

区别设计良好的模块与设计不好的模块最重要的因素在于，这个模块对于外部的模块而言是否隐藏其内部数据和其他实现细节。下面是几条规则：

（1）在设计每个类时，应尽可能地使每个类或者成员不被外界访问。

顶层类只有两种可能的访问权限：包级私有与公有。如果一个类或是接口仅仅在包内被使用，则应将其设置成包级私有的。如果一个类或接口仅仅在某个类中被使用，则应将其设置为那个类的私有嵌套类。

成员（域、方法、嵌套类、嵌套接口）有四种可能的访问权限：

私有的（private）——只有在声明该成员的顶层类内部才可以访问这个成员。

包级私有的（package-private）——声明该成员的包内部的任何类都可以访问这个成员。为默认访问级别。

受保护的（protected）——声明该成员的类的子类与同一个包内的类可以访问这个成员。

公有的（public）——任何地方都可以访问。

（2）受保护的成员属于类提供的API的一部分，必须永远得到支持。

（3）如果方法覆盖了超类的一个方法，子类中该方法的访问级别就不得低于超类中的访问级别，这一点由编译器保证。这样可以保证任何使用超类实例的地方都可以使用子类实例。

（4）实例域绝不能是公有的。如果域是非final的，或是一个指向可变对象的final引用，那么一旦这个域成为公有的，就无法对存储在这个域中的值进行限制。包含公有可变域的类不是线程安全的。

（5）如果需要提供静态常量，可以通过公有的静态final域来实现，但必须保证这些域要么是基本类型，要么是指向不可变对象的引用。这种域的名称由大写字母组成，单词之间用下划线隔开。

（6）长度非零的数组总是可变的，因此如果类具有公有的静态final数组域，或者返回这种域的访问方法，几乎总是错误的。正确的方法有两种，一是利用Collections提供的不可变列表，二是返回列表的拷贝（利用clone方法（内置））。

14、在公有类中使用访问方法而非公有域。

公有类的任何非常量域都应当使用getter与setter来访问。如果选择直接暴露域，那么就无法在不修改客户端代码的情况下改变其内部表示。

15、使可变性最小化。