## 1 考虑用静态工厂方法代替构造器

优势：

1. 有名称，用方法名标识获取对象的名称
2. 不必每次调用都创建新的对象
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力的时候，它们使代码变得更加简洁

缺点：

1. 类如果不含有公有的或者受保护的构造器，就不能被子类化
2. 静态工厂方法与其它静态方法没有任何区别

常用名称：

valueOf

of

getInstance

newInstance

getType

newType

## 2 遇到多个构造器参数时考虑使用构建器

当类的属性较多时，用一般的构造方法，重载情况多。用对象的构建器来创建对象。

如：

Component c=new Component.Builder(a,b).field1(c).field2(d).build();

其中 Builder为静态内部类，field方法都返回Builder(return this)，build方法返回Component对象。

缺点：创建对象需要先创建构造器，因此它只有在有很多参数的时候才使用。

## 3 用私有构造器或者枚举类型强化Singleton属性

用私有构造方法或枚举型来强化单例模式

单例模式：

枚举型：

public enum Elvis{

INSTANCE;

public void leaveTheBuilding(){...}

}

## 4 通过私有构造器强化不可实例化的能力

对于一些只提供静态方法的工具类，我们不希望它被实例化，则可以用私有构造器来避免。如：

public class ToolUtils{

private ToolUtils(){

throw new Exception(“本类内部实例化抛异常”);

}

// other static method.

}

## 5 避免创建不必要的对象

对于在类的方法中每次调用都需要创建相同的对象，那么就可以使用静态属性及静态代码块来初始化类时创建一次,例如A类中有个方法a要比较传入的日期是否在两个固定的日期之间，一种方法是在a中构建两个日期，再比较。这样每次比较都会创建两个日期对象。简化的方法就是在类中定义两个静态成员变量。

public class Person{

private static final Date START;

private static final Date END;

//other fields

static {

//initial code can move to method isInDateArea() and initialize for the first calling.

SimpleDateFormat sdf=new SimpleDateFormat(“yyyy-MM-dd HH:mm:ss”);

START=sdf.parse(“1946-01-01 00:00:01”);

END=sdf.parse(“1965-01-01 00:00:00”);

}

public boolean isInDateArea(Date birthday){

return birthday.compareTo(START) >=0 &&birthday.compareTo(END)<0;

}

}

基本类型优先于包装类型使用。当重用对象的代价大于创建对象时不建议重用对象，例如自己的不必要的对象池，维护对象池的开销大于创建对象。除非像数据库连接池那样，创建连接对象的代价非常大，则可以用连接池来复用连接对象。

## 6 消除过期的对象引用

异常内存泄漏的原因：

1. 自定义栈中过期元素没有清除。——不用的元素要设为null
2. 过期不用的缓存。
3. 无需长久存在的监听器和回调没有显示取消。

## 7 避免使用终结方法

终结方法 super.finalize()：不一定及时执行，也不一定会执行。可用于终止非关键的本地资源。如果用终结方法作为安全网，要记得记录终结方法的非法用法。

## 8 覆盖equals方法时遵守通用约定

约定：

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y.equals(x)=true时，x.equals(y)=true。
3. 传递性：非null引用值x,y,z 当x.equals(y)=true,y.equals(z)=true,则x.equals(z)=true。
4. 一致性：非null引用值x,y，不管怎样操作，只要x,y信息没修改，则多次调用x.equals(y)会一致返回true或false。
5. 非空性：所有引用不能为null。

告诫：

1. 覆盖equals时总要覆盖hashCode。
2. 不要企图让equals方法过于智能。
3. 不要将equals声明中的Object对象替换为其它类型方法参数类型要为Object。
4. 在每次覆盖equals方法都逐一审查equals 的五个约定

## 9 覆盖equals时总要覆盖hashCode

覆盖equals时如果不覆盖hashCode，会导致该类无法结合所有基于散列的集合一起正常运作，这样的集合包括HashMap,HashSet和HashTable。

覆盖hashCode约定：

1. 在应用程序执行期间，只要对象equals方法比较操作所用到的信息没有被修改，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。同一个应用程序多次执行过程中，每次执行所返回的整数可以不一致。
2. 如果两个对象equals比较是相等的，那么这两个对象任意一个的hashCode方法都必须产生同样的整数结果。
3. 如果两个对象equals比较是不相等的，那么这两个对象任意一个的hashCode方法都必须产生不同的整数结果。这样有可能提高散列表的性能。

利用equals比较的每个域来生成hashCode：

a 计算域f的散列值c

1. 如果该域为boolean类型，则计算(f?1:0)。
2. 如果该域为byte ,char,shot或者int类型，则计算(int)f。
3. 如果该域为long类型，则计算(int)(f^(f>>>32))。
4. 如果该域为float类型，则计算Float.floatToIntBits(f)。
5. 如果该域为double类型，则计算Double.doubleToLongBits(f)，然后按3为 得到的long类型值计算散列值。
6. 如果该域为一个对象引用，且该类的equals方法通过递归的调用equals方 法来比较这个域，则同样为这个域递归调用hashCode。
7. 如果该域为一个数组，则把每个元素当作一个单独的域来处理，每个域计 算一个散列码，然后根据b中的方法把这些散列值组合起来。

b 按下面的公式把a每个域得到的散列码c合并到result中

result=31\*reuslt+c

c 返回result。

## 10 始终要覆盖toString

注意：避免依赖于toString数据格式的细节进行编程。

## 11 谨慎地覆盖clone