# 概念

1. java支持四种类型：接口、类、数组、和基本类型。前三种通常被称为引用类型、类实例和数组是对象、而基本类型的值则不是对象；类的成员由它的域、方法、成员类和成员接口组成。方法的签名由它的名称和所有参数类型组成。签名不包括它的返回类型。

**方法签名：**由方法名称和一个参数列表（方法的参数的顺序和类型）组成。

1. 静态工厂方法通常更加适合，因此切忌第一反应就是提供共有构造器，而不先考虑静态工厂。

# 创建和销毁对象

1. 遇到许多构造器参数的时候，相较于采用重叠构造器模式（针对可选参数编写大量构造器），可以
2. 采用javaBean模式来替代，但是这种方法无法把类做成不可变，且线程不安全。
3. Builder模式:builder的setter方法返回builder本身，以便可以把调用链接起来，builder模式模拟了具名的可选参数，builder就像个构造器一样可以对其参数强加约束条件。

（方法中return this；this代表当前这个方法属于的对象）

public class NutritionFacts {  
 private final int sercingSize;  
 private final int servings;  
 private final int calories;  
 private final int fat;  
 private final int sodium;  
 private final int carbohydrate;  
  
 public static class Builder{  
 //Required parameters  
 private final int servingSize;  
 private final int servings;  
  
 //Option parameters  
 private int calories = 0;  
 private int fat = 0;  
 private int carbohydrate =0;  
 private int sodium = 0;  
  
 public Builder(int servingSize,int servings){  
 this.servingSize = servingSize;  
 this.servings = servings;  
 }  
  
 public Builder calories(int val){  
 calories = val ; return this;  
 }  
  
 public Builder fat(int fat){  
 this.fat = fat; return this;  
 }  
  
 public Builder carbohydrate(int val){  
 this.carbohydrate = val; return this;  
 }  
  
 public Builder sodium(int val){  
 this.sodium = val; return this;  
 }  
  
 public NutritionFacts build(){  
 return new NutritionFacts(this);  
 }  
  
 }  
  
 private NutritionFacts(Builder builder){  
 sercingSize = builder.servingSize;  
 servings = builder.servings;  
 calories = builder.calories;  
 fat = builder.fat;  
 sodium = builder.sodium;  
 carbohydrate = builder.carbohydrate;  
 }  
  
}

NutritionFacts cocaCola = new NutritionFacts.Builder(240,8).calories(100).sodium(35).carbohydrate(27).build();

1. 通过私有构造器强化不可实例化的能力。只有当类不包含显式构造器时，编译器才会生成缺省的构造器，因此只要让这个类包含私有构造器，他就不能被实例化了。
2. 避免创建不必要的对象：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。对于同时提供静态工厂方法和构造器的不可变类，通常使用静态工厂方法较优。如Boolean.valueOf(String)而不是new Boolean(String)
3. **String s = new String(“stringette”)**这种写法每次执行的时候都会创建一个新的String实例，而String s = “stringette”只用了一个String实例，且可以保证在同一台虚拟机中运行的代码，只要包含相同字符串字面常量，该对象就会被重用。
4. 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。如在下段代码：

Long sum = 0L;

for(long I = 0;i<Integer.MAX\_VALUE;i++){

sum+=I;

}

这段代码中，意味着程序每次从long到Long构造了大约2^31个多余的Long实例。

1. 使用静态块，初始化需要创建的一些常用不变实例，而避免在方法中去每次都创建一遍。
2. 内存泄漏：如果一个栈先增长，然后再收缩，那么从栈中弹出的对象不会被回收，因为栈内部维护着对这些对象的过期引用。应该在pop方法中弹出之后，手动回收：element[size] = null 。

# 对于所有对象都通用的方法

1. Object.equals方法的覆盖：
   1. 如果类具有自己特有的”逻辑相等”的概念，而且超类还没有覆盖equals实现期望的行为，这时就需要覆盖equals方法。
   2. 有一种”值类”不需要覆盖equals方法，即实例受控确保每个值最多存在一个对象，枚举类型就属于这种类
   3. 覆盖equals方法要遵循equals方法的特性：自反性、对称性、传递性、一致性（x.equals(null)必须返回false）。
2. 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. 把参数转换成正确的类型
   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. 覆盖equals方法时总要覆盖hashCode
   6. 不要将equals声明中的Object对象替换为其他的类型

例如：比较一个色点类，不仅比较类的对象，还要比较类中的颜色、和色点域

public class ColorPoint{

private final Point point;

private final Color color;

……

@Override

public boolean equals(Object o){

if(!(o instanceof ColorPoint))

return false;

ColorPoint cp = (ColorPoint)o;

return cp.point.equals(point) && cp.color.equals(color);

……

}

1. 假如重写了一个PhoneNumber类的equals函数，而没有重写hashCode，这时你企图将这个类与HashMap一起使用：  
   Map<PhoneNumber ,String> m = new HashMap<PhoneNumber , String>();

m.put(new PhoneNumber(707,867,5390),”jenny”);

这时你期望通过m.get(new PhoneNumber(707,867,5390))会返回””Jenny，但实际返回的是null。因为这里两个的PbonerNumber实例具有不同的散列码。

我们可以自定义适当的散列函数，重写PhonerNumber类的hashCode方法：如

@Override

public int hashCode(){

int result = 17;

result = 31\*result + areaCode;

result = 31\*result + prefix;

result = 31\*result + lineNumber;

return result;

}

1. 建议始终覆盖toString()方法,保证返回的字符串是一个“简洁的，但信息丰富的，并且易于阅读的表达形式”。
2. （谨慎覆盖clone方法）一般clone的含义是对于任何对象x，表达式：
   1. x.clone != x; true
   2. x.clone.getClass() == x.getClass(); true
   3. x.clone.equals(x); true
   4. clone方法返回的对象相当于构造器返回的一个新实例，默认的equals比较返回false（需要自定义覆写equals方法）
   5. 在继承结构中getClass()返回的就是this指针所代表的最真实的Class的对象，也即最上层的子类。
3. 实现Cloneable接口并重写提供公有的clone方法，由于Object.clone返回Object，所以需要在返回前将它转换：

@Override  
public NutritionFacts clone(){  
 try {  
 return (NutritionFacts)super.clone();  
 } catch (CloneNotSupportedException e) {  
 throw new AssertionError(); //Can't happen  
 }  
}

1. 考虑实现Comparable接口：  
   一个类中有多个关键域时，按照什么顺序来比较这些域是很关键的。必须从最关键的域开始，逐步到所有重要域在compareTo方法中进行比较。

# 类和接口

1. 使类的可访问性最小化，一个模块不需要知道其它模块内部的工作情况，这个概念被称为信息隐藏，或封装。（软件设计的基本原则之一，解耦、易于修改、优化、独立开发测试）
2. 如果方法覆盖了超类中的一个方法，子类中的访问级别不允许低于超类中的访问级别。确保了任何可使用此超类实例的地方也可使用此子类。
3. 在公有类中使用访问方法而非公有域。
4. 对于频繁用到的值，为他们提供公有的静态final常量。