#### ---遇到多个构造器参数时考虑用构建器---

静态工厂和构造器有个共同的局限性：不能很好地扩展到大量的可选参数。

比如：

public class NutritionFacts {

private final int servingSize; // required

private final int servings; // required

private final int calories; // optional

private final int fat; // optional

private final int sodium; // optional

private final int carbohydrate; // optional

public NutritionFacts(int servingSize, int servings) {

this(servingSize, servings, 0);

}

public NutritionFacts(int servingSize, int servings, int calories) {

this(servingSize, servings, calories, 0);

}

public NutritionFacts(int servingSize, int servings, int calories, int fat) {

this(servingSize, servings, calories, fat, 0);

}

public NutritionFacts(int servingSize, int servings, int calories, int fat, int sodium) {

this(servingSize, servings, calories, fat, sodium, 0);

}

public NutritionFacts(int servingSize, int servings, int calories, int fat, int sodium, int cabohydrate) {

this.servingSize = servingSize;

this.srvings = servings;

this.calories = calories;

this.fat = fat;

this.sodium = sodium;

this.carbohydrate = carbohydrate;

}

}

当你要创建NutritionFacts的实例的时候，就利用参数列表最短的构造器，但该列表包含了要设置的所有参数。

这个构造器调用通常需要许多你本不想设置的参数，但是还是不得不为他们传递值。

由此可见，重叠构造器模式可行，但是当有许多参数的时候，客户端代码就很难编写，并且难以阅读。

改进的方法---JavaBeans模式

调用一个无参构造器来创建对象，然后调用setter方法来设置每个必要的参数：

public class NutritionFacts {

private final int servingSize = -1; // required

private final int servings = -1; // required

private final int calories = 0; // optional

private final int fat = 0; // optional

private final int sodium = 0; // optional

private final int carbohydrate = 0; // optional

public NutritionFacts(){}

// setters

public void setServingSize(int val) { servingSize = val;}

// .....

}

这种模式下创建实例：

NutritionFacts fact = new NutritionFacts();

fact.setServingSize(250);

//....

这种模式下严重的缺点是，构造过程被分到了几个调用中，在构造中JavaBean可能处于不一致的状态。

终极方法---Builder模式，示例：

public class NutritionFacts {

private final int servingSize; // required

private final int servings; // required

private final int calories; // optional

private final int fat; // optional

private final int sodium; // optional

private final int carbohydrate; // optional

public static class Builder {

private final int servingSize; // required

private final int servings; // required

private int calories = 0; // optional

private int fat = 0; // optional

private int sodium = 0; // optional

private int carbohydrate = 0; // optional

publilc Builder(int servingSize, int servings) {

this.servingSize = servingSize;

this.servings = servings;

}

public Builder calories(int val) {

calories = val;

return this;

}

public Builder fat(int val) {

fat = val;

return this;

}

public Builder carbohydrate(int val) {

carbohydrate = val;

return this;

}

public Builder sodium(int val) {

sodium = val;

return this;

}

public NutritionFacts build() {

return new NutritionFacts(this);

}

}

private NutritionFacts(Builder builder) {

servingSize = builder.servingSize;

servings = builder.servings;

calories = builder.calories;

fat = builder.fat;

sodium = builder.sodium;

carbohydrate = builer.carbohydrate;

}

}

这里NutritionFacts是不可变的，所有默认的参数值都单独放在一个地方，builder的setter方法返回builder本身，以便可以把调用链接起来。创建对象实例：

NutritionFacts coca = new

NutritionFacts.Builder(290,5).calories(90),sodium(33).carbohydrate(22).builder();

#### 实现高质量equals方法的诀窍

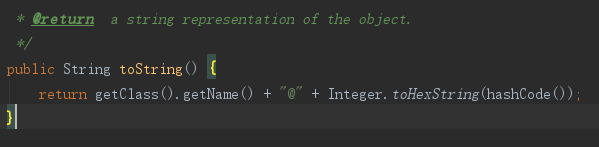
1. 使用“==”操作符检查参数是否为这个对象的引用
2. 使用instanceof检查参数是否为正确的类型
3. 将参数转化为正确的类型
4. 对于该类中的每个关键域，检查参数中的域是否与该对象中对应的域匹配
5. 域的比较顺序也会影响equals方法的性能，应该最先比较最有可能不一致的域，或者开销最低的域

equals方法三个特性：对称性、传递性、一致性

#### 覆盖equals时总要覆盖hashCode

#### 始终要覆盖toString()

java.lang.Object中的toString()方法



覆盖原始的toString()方法增加可读性