例子，就农场库存程序而言，你必须实现一个从列表中筛选绿苹果的功能。

#### 1、筛选绿苹果

public static List<Apple> filterGreenApples(List<Apple> inventory) {

List<Apple> result = new ArrayList<>();  
 for (Apple apple : inventory) {  
 if ("green".equals(apple.getColor())) {  
 result.add(apple);  
 }  
 }  
 return result;  
}

突出显示的行就是筛选绿苹果所需的条件。但是现在农民改主意了，他还想要筛选红苹果。你该怎么做呢?简单的解决办法就是复制这个方法，把名字改成filterRedApples，然后更改 if条件来匹配红苹果。然而，要是农民想要筛选多种颜色：浅绿色、暗红色、黄色等，这种方法 就应付不了了。一个良好的原则是在编写类似的代码之后，尝试将其抽象化。

#### 2、把颜色作为参数

一种做法是给方法加一个参数，把颜色变成参数，这样就能灵活地适应变化了。

public static List<Apple> filterApplesByColor(List<Apple> inventory, String color) {

List<Apple> result = new ArrayList<>();  
 for (Apple apple : inventory) {  
 if (apple.getColor().equals(color)) {  
 result.add(apple);  
 }  
 }  
 return result;  
}

这时要是能区分轻的苹果就太好了。可能会想到用另外一个参数来应对不同的重量：

public static List<Apple> filterApplesByWeight(List<Apple> inventory, int weight) {

List<Apple> result = new ArrayList<>();  
 for (Apple apple : inventory) {  
 if (apple.getWeight() > weight) {  
 result.add(apple);  
 }  
 }  
 return result;  
}

但是这样复制了大部分的代码来实现遍历库存，并对每个苹果应用少选条件。这打破了DRY（Don’t Repeat Yourself，不要重复自己）的软件工程原则。如果想要改变筛选变量方式来提升性能呢？那就得修改所有方法的实现，而不是只改一个。从工程工作量的角度来看，这代价太大了。

#### 3、对你能想到的每个属性做筛选

你可以将颜色和重量结合为一个方法，称为filter。不过就算这样，你还是需要一种方式来区分想要筛选哪个属性。你可以加上一个标志来区分对颜色和重量的查询（但绝不要这样做！）一种把所有属性结合起来的笨拙尝试如下：

public static List<Apple> filterApples(List<Apple> inventory, String color,

int weight, boolean flag) {  
 List<Apple> result = new ArrayList<>();  
 for (Apple apple : inventory) {  
 boolean existed = (flag && apple.getColor().equals(color)) ||  
 (!flag && apple.getWeight() > weight);  
 if (existed) {  
 result.add(apple);  
 }  
 }  
 return result;  
}

可以这么用（但很笨拙）：

List<Apple> greenApples = filterApples(inventory, "green", 0, true);

List<Apple> heavyApples = filterApples(inventory, "", 150, false);