目前，当要把新的行为传递给filterApples方法的时候，你不得不声明好几个实现ApplePredicate对象。Java有一个机制称为匿名类，它可以让你同时声明和实例化一个类。它可以帮助你进一步改善代码，让它变得更简洁。但这也不完全令人满意。

#### 1、匿名类

匿名类和你熟悉的Java局部类（块中定义的类）差不多，但匿名类没有名字。它允许你同时声明并实例化一个类。换句话说，它允许你随建随用。

#### 2、使用匿名类

下面代码展示了如何通过创建一个匿名类实现ApplePredicate的对象，重写筛选的例子：

//使用匿名类实现ApplePredicate的对象

List<Apple> redApples = *filterApples1*(inventory, new ApplePredicate() {  
 @Override  
 public boolean test(Apple apple) {  
 return "red".equals(apple.getColor());  
 }  
});

直接内联参数化filterApples1方法的行为。

GUI应用程序中经常使用匿名类来创建事件处理器对象（下面的例子使用的是Java FX API，一种现代的Java UI平台）：

botton.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent event) {  
 System.out.println("Woooo a click!!");  
 }  
});

但是匿名类还是不够好。

* 第一，它往往很笨重，因为它占用了很多空间。
* 第二，很多程序员觉得它用起来很让人费解。如下经典的Java谜题：

public class MeaningOfThis {

public final int value = 4;  
  
 public void doIt() {  
 int value = 6;  
 Runnable r = new Runnable() {  
 public final int value = 5;  
 @Override  
 public void run() {  
 int value = 10;  
 System.*out*.println(this.value);  
 }  
 };  
 r.run();  
 }  
  
 public static void main(String... args) {  
 MeaningOfThis m = new MeaningOfThis();  
 m.doIt();  
 }  
}

this指的是包含它的Runnable，而不是外面的类MeaningOfThis。

即使匿名类处理在某种程度上改善了为一个接口声明好几个实体类的啰嗦问题，但它仍不能令人满意。在只需要传递一段简单的代码时（例如表示选择标准的boolean表达式），你还是要创建一个对象，明确地实现一个方法来定义一个新的行为（例如Predicate中的test方法或是EventHandler中的handler方法）。

在理想情况下，我们想鼓励程序员使用行为参数化模式，因为正如你前面看到的，它让代码更能适应需求的变化。

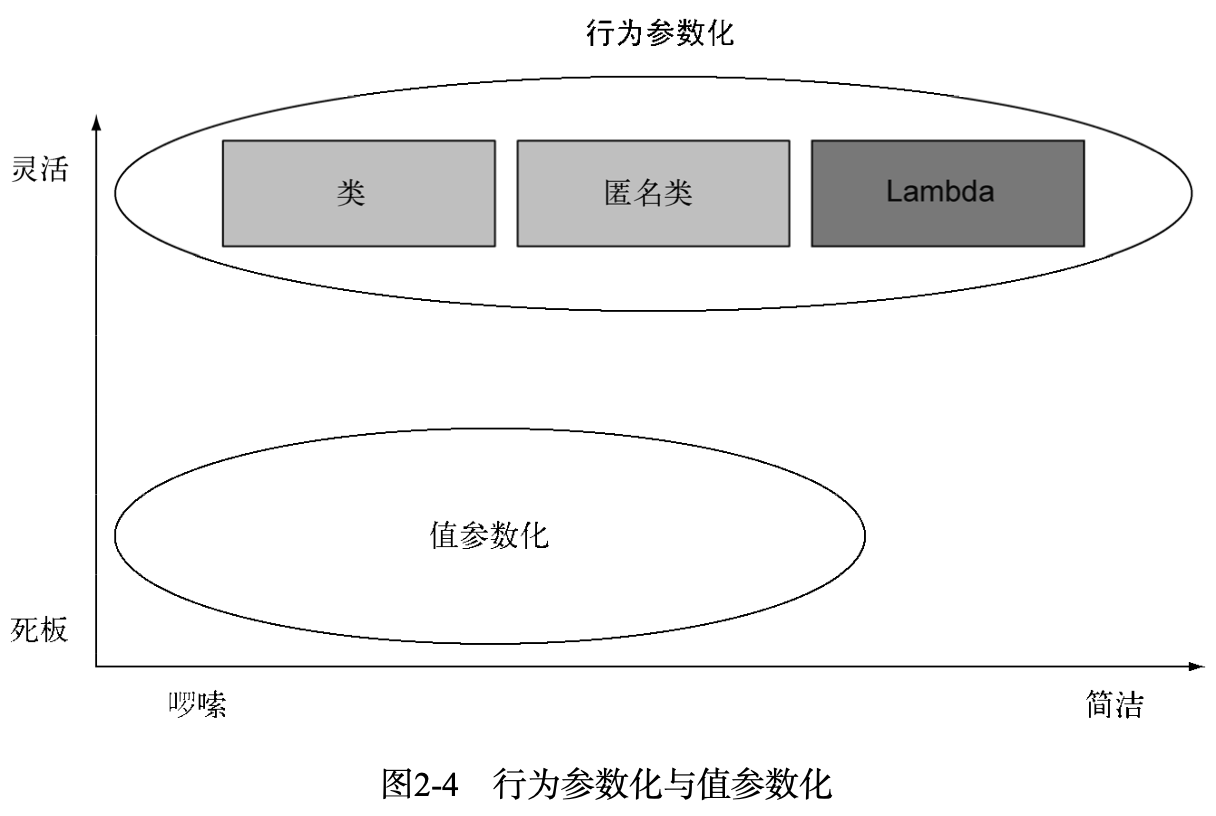
#### 3、适应Lambda表达式

上面的代码在Java 8里可以用Lambada表达式重写为下面的样子：

List<Apple> result = *filterApples*(inventory,

(Apple apple) -> "red".equals(apple.getColor()));

不得不承认这代码看上去比先前干净很多。它看起来更像问题陈述本身了。做个小结：



#### 4、将List类型抽象化

在通往抽象的路上，我们还可以更进一步。目前，filterApples方法还只适用于Apple。还可以将List类型抽象化：

public interface Predicate<T> {

boolean test(T t);  
default Predicate<T> and(Predicate<? super T> other) {  
 Objects.*requireNonNull*(other);  
 return (t) -> test(t) && other.test(t);  
 }  
default Predicate<T> negate() {  
 return (t) -> !test(t);  
 }  
default Predicate<T> or(Predicate<? super T> other) {  
 Objects.*requireNonNull*(other);  
 return (t) -> test(t) || other.test(t);  
 }  
static <T> Predicate<T> isEqual(Object targetRef) {  
 return (null == targetRef)  
 ? Objects::*isNull* : object -> targetRef.equals(object);  
 }  
}

public static <T> List<T> filter(List<T> list, Predicate<T> p) {  
 List<T> result = new ArrayList<>();  
 for (T e : list) {  
 if (p.test(e)) {  
 result.add(e);  
 }  
 }  
 return result;  
}

现在可以把filter方法用在香蕉、橘子、Integer或是String的列表上了。下面是一个使用Lambda表达式的例子：

List<Integer> evenNumbers = *filter*(numbers,

(Integer i) -> i % 2 == 0);

现在在灵活性和简洁性之间找到了最佳的平衡点，这在Java 8之前是不可能做到的！