2 Passing code with behavior parameterization

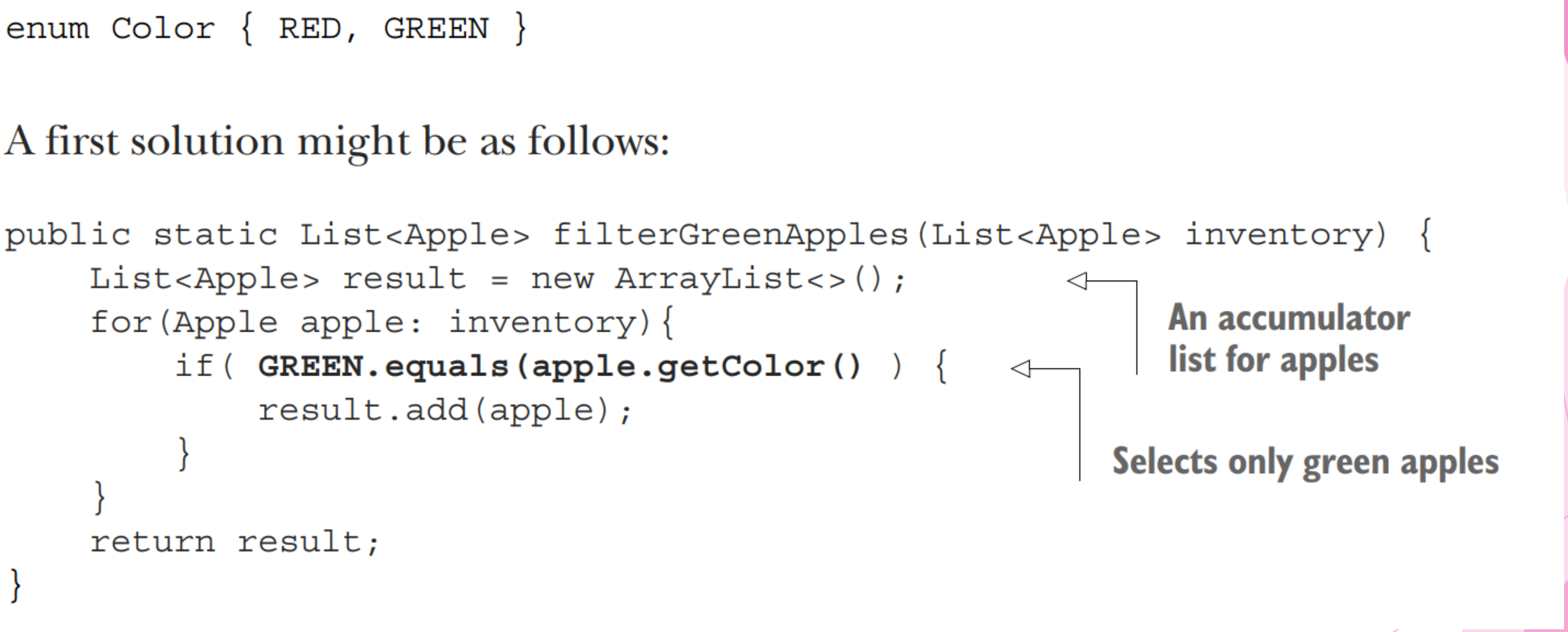
This chapter covers

- Coping with changing requirements
- Behavior parameterization
- Anonymous classes
- Preview of lambda expressions
- Real-world examples: Comparator, Runnable, and GUI

2.1 Coping with changing requirements

- A well-know problem in software engineering is that no matter what you do, user requirements will change.
- *Behavior parameterization* is a software development pattern that lets you handle frequent requirement changes.
  - pass the block of code as an argument to another method that will execute it later.
  - Java API examples: to sort a List, to filter names of files, to tell a Thread to execute a block of code, or even perform GUI event handling.

First attempt: filtering green apples



```
enum Color { RED, GREEN }
```

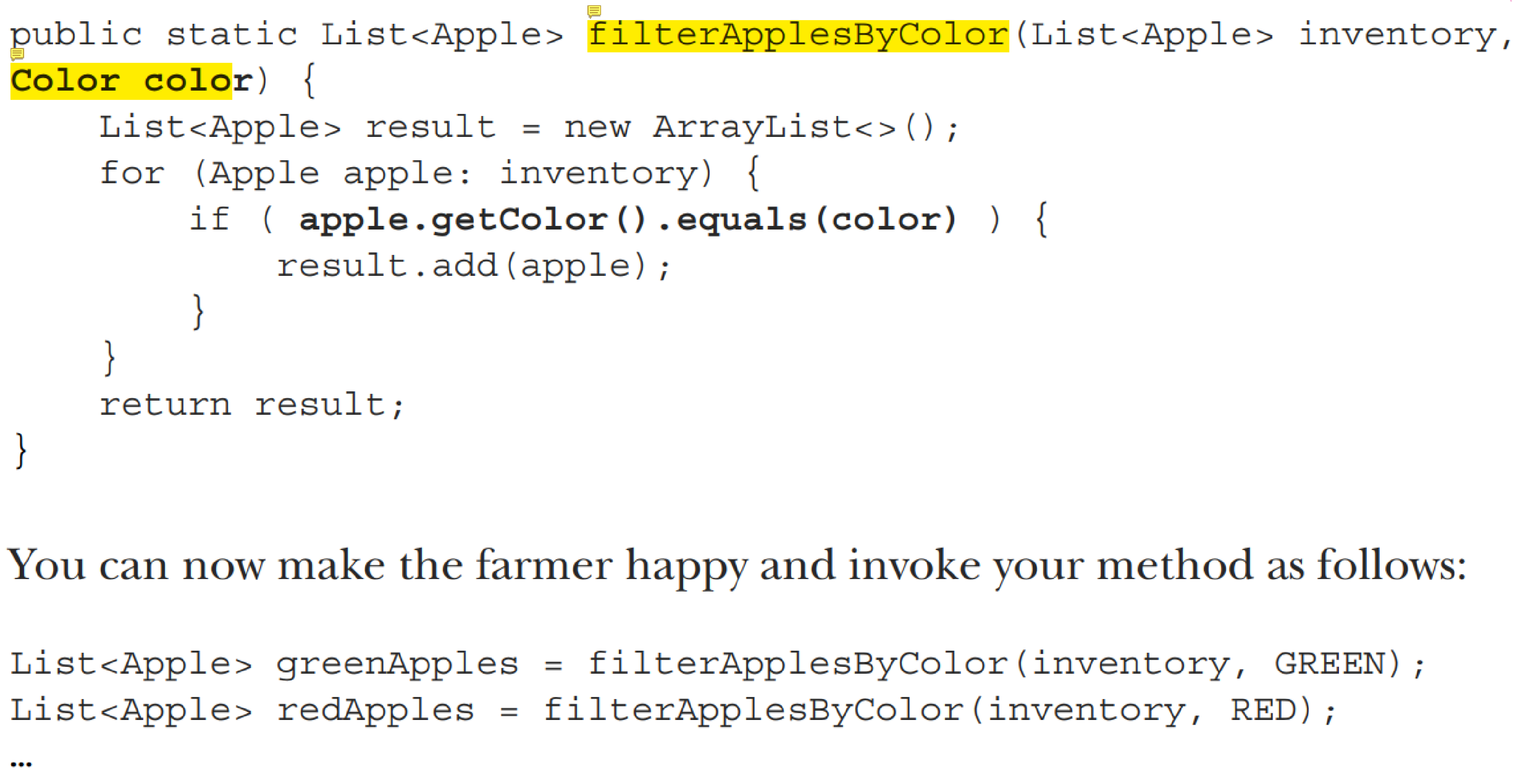
A first solution might be as follows:

```
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;
}
```

An accumulator list for apples

Selects only green apples

Second attempt: parameterizing the color

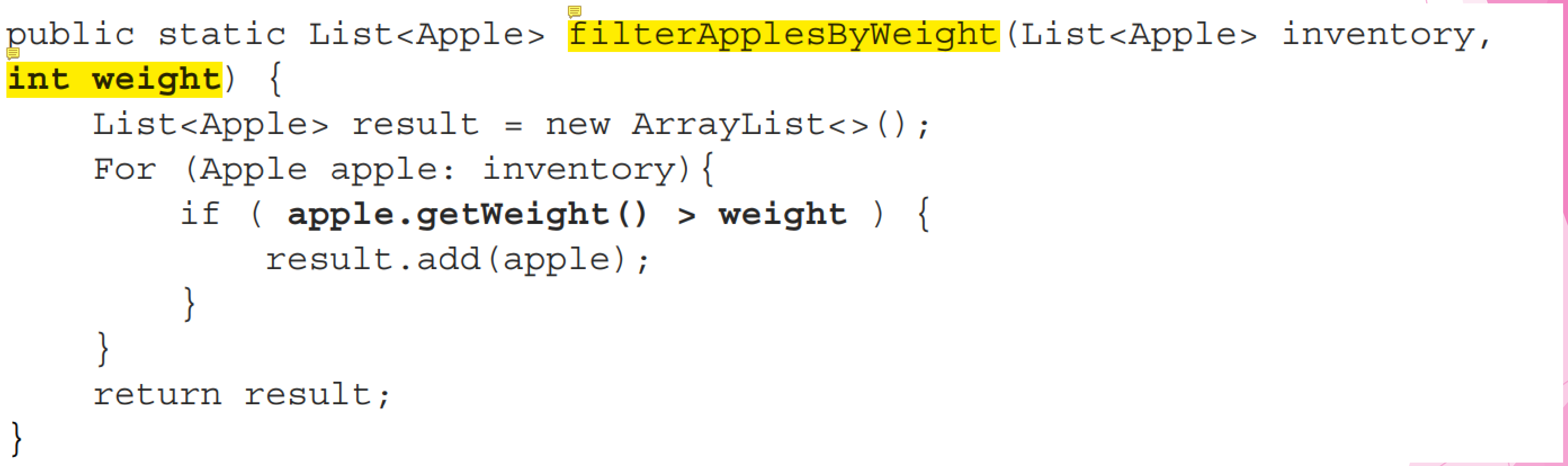


```
public static List<Apple> filterApplesByColor(List<Apple> inventory,
Color color) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory) {
        if ( apple.getColor().equals(color) ) {
            result.add(apple);
        }
    }
    return result;
}
```

You can now make the farmer happy and invoke your method as follows:

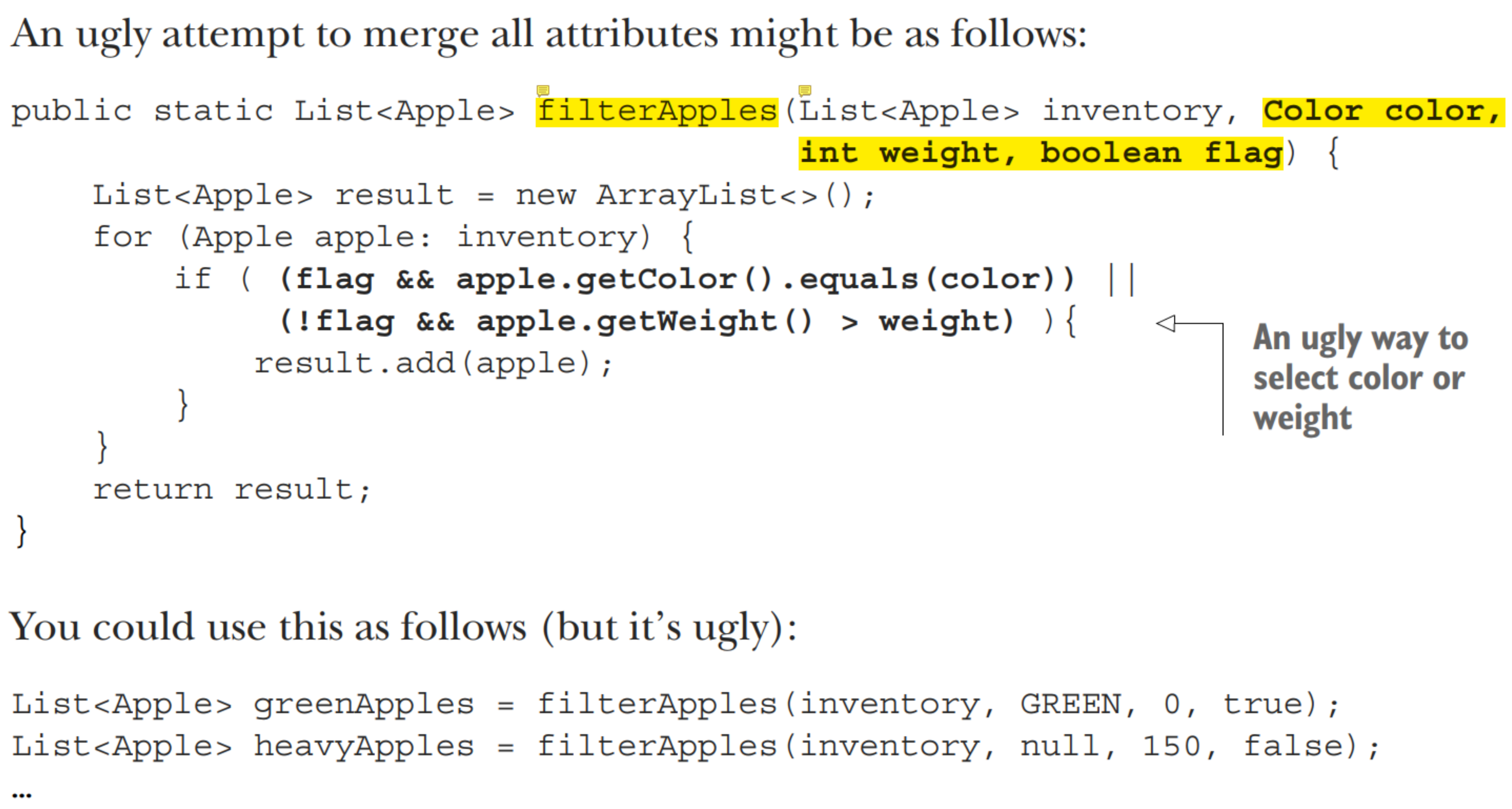
```
List<Apple> greenApples = filterApplesByColor(inventory, GREEN);
List<Apple> redApples = filterApplesByColor(inventory, RED);
…
```

The following method can cope with various weights through an additional parameter:



```
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;
}
```

Third attempt: filtering with every attribute you can think of



An ugly attempt to merge all attributes might be as follows:

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

An ugly way to select color or weight

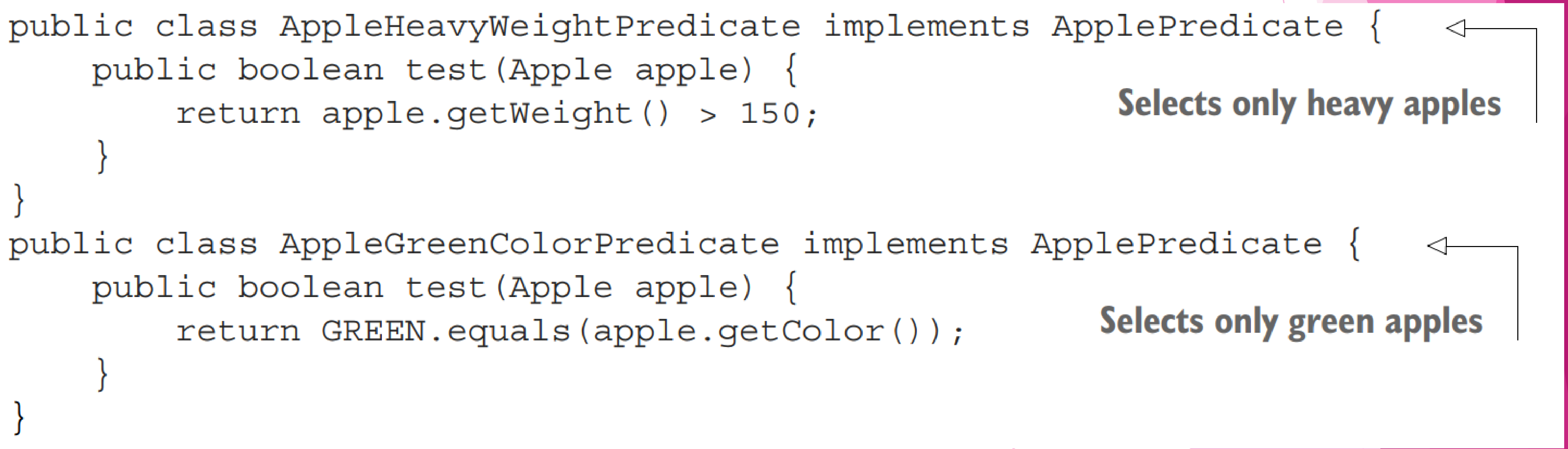
You could use this as follows (but it's ugly):

```
List<Apple> greenApples = filterApples(inventory, GREEN, 0, true);
List<Apple> heavyApples = filterApples(inventory, null, 150, false);
…
```

2.2 Behavior parameterization

- *Predicate* – a function that returns a Boolean

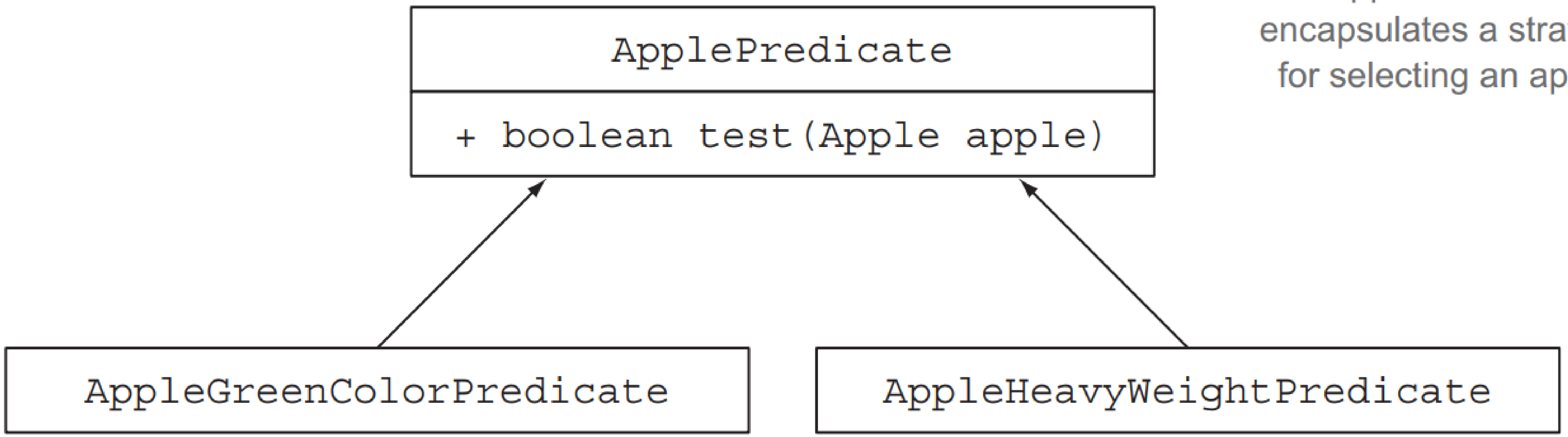
```
public interface ApplePredicate {
    Boolean test (Apple apple);
}
```



```
public class AppleHeavyWeightPredicate implements ApplePredicate {
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
}
public class AppleGreenColorPredicate implements ApplePredicate {
    public boolean test(Apple apple) {
        return GREEN.equals(apple.getColor());
    }
}
```

Selects only heavy apples

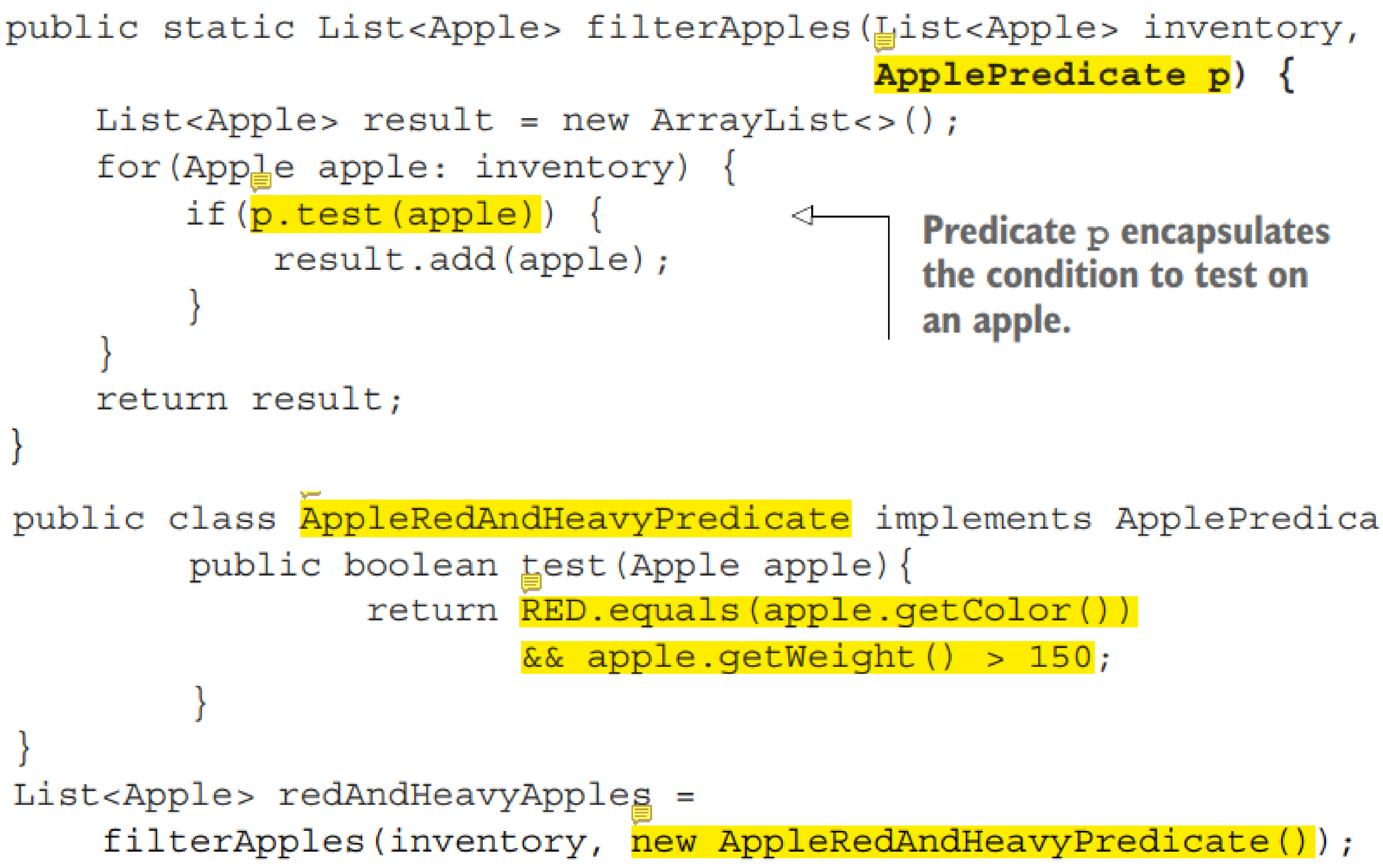
Selects only green apples



ApplePredicate encapsulates a strategy for selecting an apple

Figure 2.1 Different strategies for selecting an `Apple`

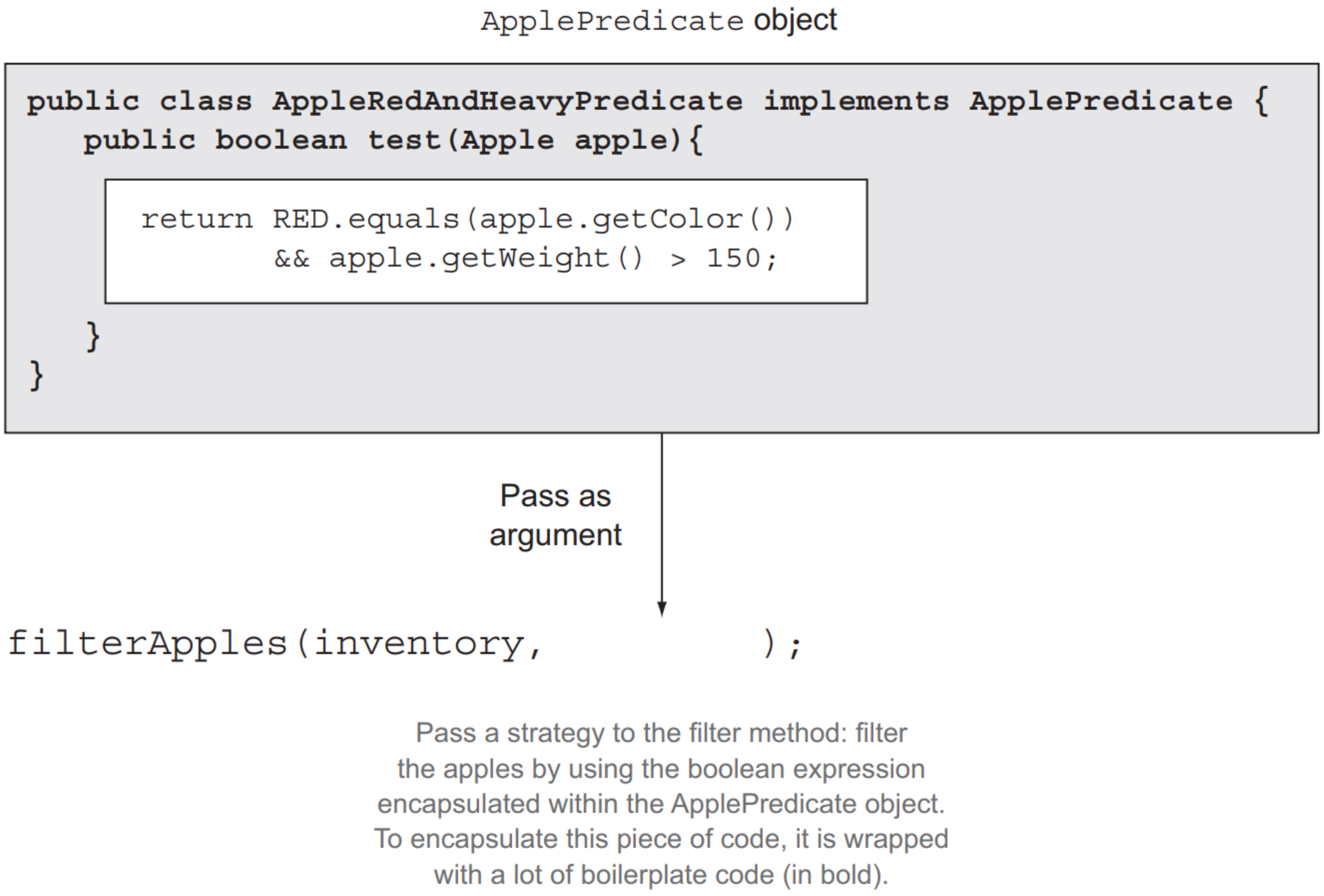
Fourth attempt: filtering by abstract criteria



```
public static List<Apple> filterApples(List<Apple> inventory,
                                        ApplePredicate p) {
    List<Apple> result = new ArrayList<>();
    for(Apple apple: inventory) {
        if(p.test(apple)) {
            result.add(apple);
        }
    }
    return result;
}
```

Predicate p encapsulates the condition to test on an apple.

```
public class AppleRedAndHeavyPredicate implements ApplePredicate {
        public boolean test(Apple apple){
                return RED.equals(apple.getColor())
                       && apple.getWeight() > 150;
        }
}
List<Apple> redAndHeavyApples =
    filterApples(inventory, new AppleRedAndHeavyPredicate());
```

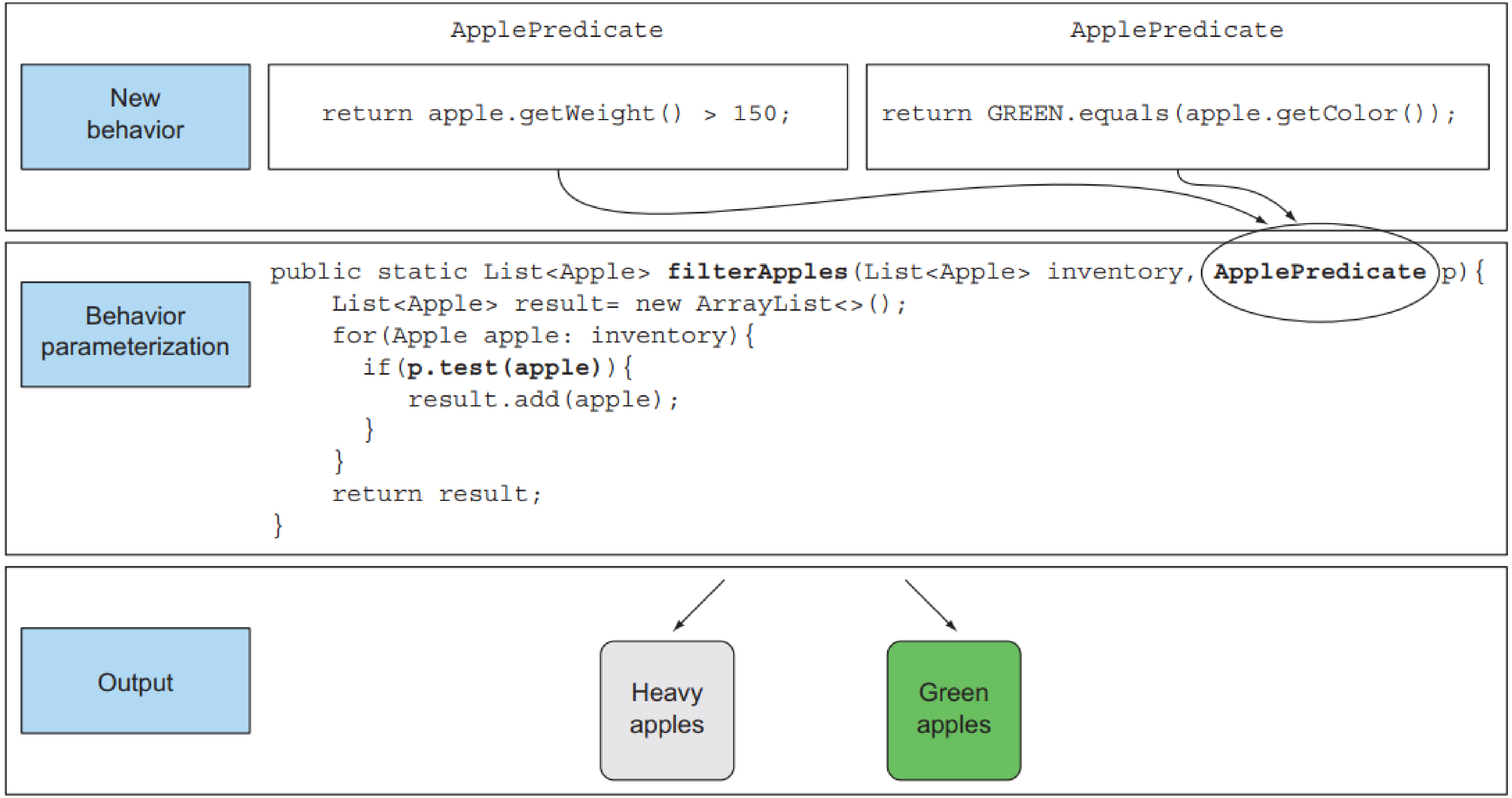


```
public class AppleRedAndHeavyPredicate implements ApplePredicate {
   public boolean test(Apple apple){
      return RED.equals(apple.getColor())
             && apple.getWeight() > 150;
   }
}
```

```
filterApples(inventory,           );
```

Pass a strategy to the filter method: filter the apples by using the boolean expression encapsulated within the ApplePredicate object. To encapsulate this piece of code, it is wrapped with a lot of boilerplate code (in bold).

Figure 2.2 Parameterizing the behavior of `filterApples` and passing different filter strategies



```
return apple.getWeight() > 150;
```

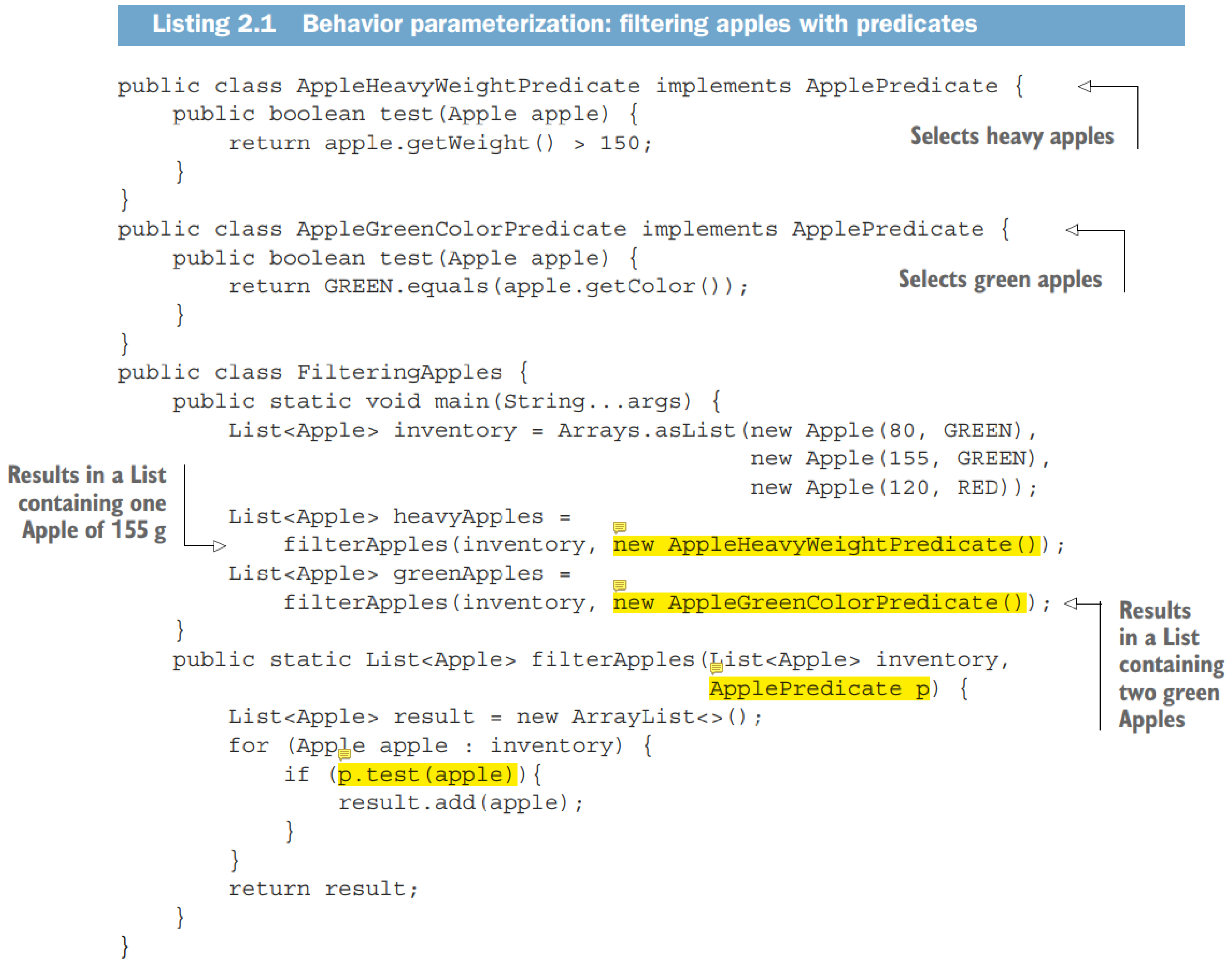
```
return GREEN.equals(apple.getColor());
```

```
public static List<Apple> filterApples(List<Apple> inventory, ApplePredicate p){
    List<Apple> result= new ArrayList<>();
    for(Apple apple: inventory){
      if(p.test(apple)){
         result.add(apple);
      }
    }
    return result;
}
```

Figure 2.3 Parameterizing the behavior of `filterApples` and passing different filter strategies

2.3 Tackling verbosity

- *Anonymous classes* are like the local classes (a class defined in a block) that you're
  already familiar with in Java. But anonymous classes don't have a name.
- They allow you to declare and instantiate a class at the same time. In short, they allow you to create ad hoc implementations.



Listing 2.1 Behavior parameterization: filtering apples with predicates

```
public class AppleHeavyWeightPredicate implements ApplePredicate {
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
}
public class AppleGreenColorPredicate implements ApplePredicate {
    public boolean test(Apple apple) {
        return GREEN.equals(apple.getColor());
    }
}
public class FilteringApples {
    public static void main(String...args) {
        List<Apple> inventory = Arrays.asList(new Apple(80, GREEN),
                                              new Apple(155, GREEN),
                                              new Apple(120, RED));
        List<Apple> heavyApples =
            filterApples(inventory, new AppleHeavyWeightPredicate());
        List<Apple> greenApples =
            filterApples(inventory, new AppleGreenColorPredicate());
    }
    public static List<Apple> filterApples(List<Apple> inventory,
                                           ApplePredicate p) {
        List<Apple> result = new ArrayList<>();
        for (Apple apple : inventory) {
            if (p.test(apple)){
                result.add(apple);
            }
        }
        return result;
    }
}
```

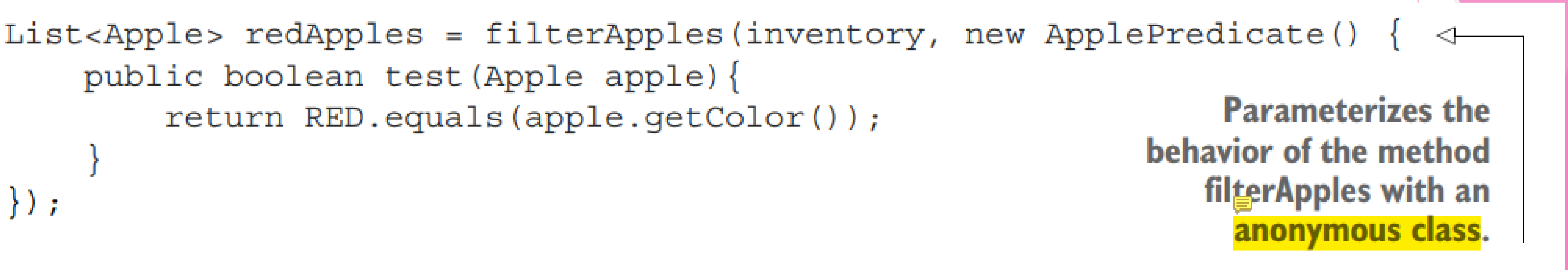
Selects heavy apples

Selects green apples

Results in a List containing one Apple of 155 g

Results in a List containing two green Apples

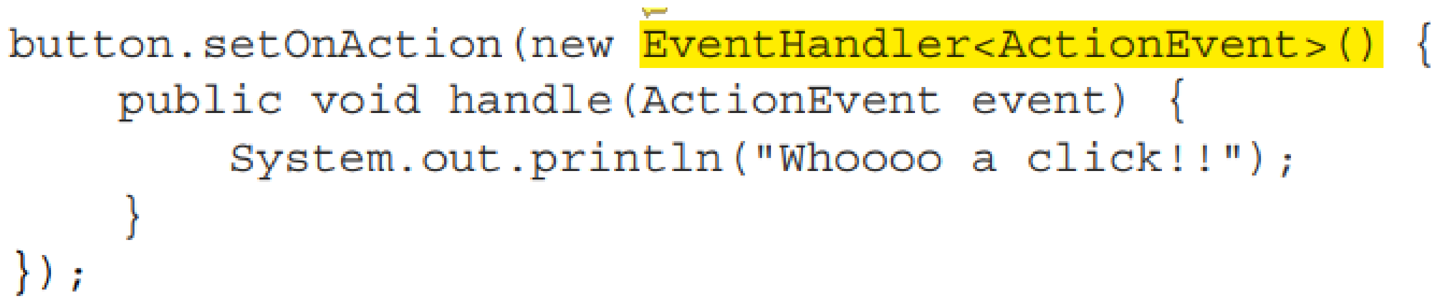
Fifth attempt: using an anonymous class



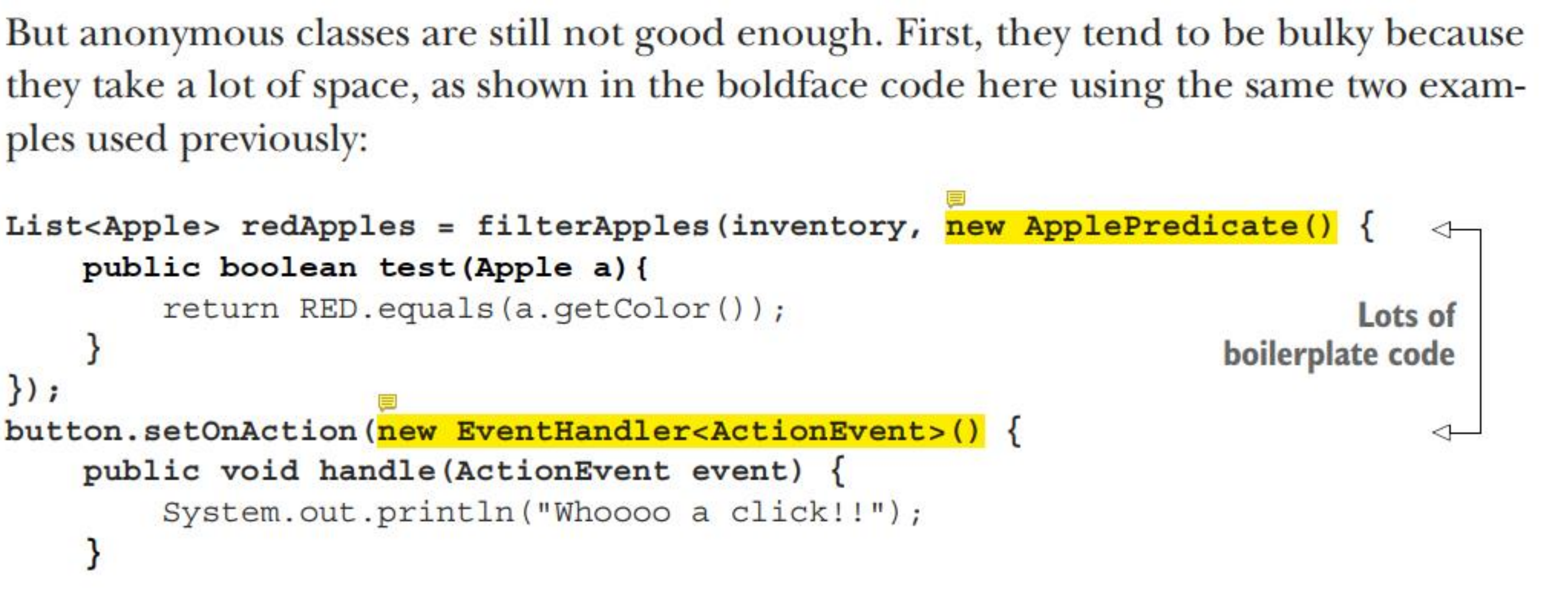
```
List<Apple> redApples = filterApples(inventory, new ApplePredicate() {
    public boolean test(Apple apple){
        return RED.equals(apple.getColor());
    }
});
```

Parameterizes the behavior of the method filterApples with an anonymous class.

Using the JavaFX API, a modern UI platform for Java:



```
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        System.out.println("Whoooo a click!!");
    }
});
```



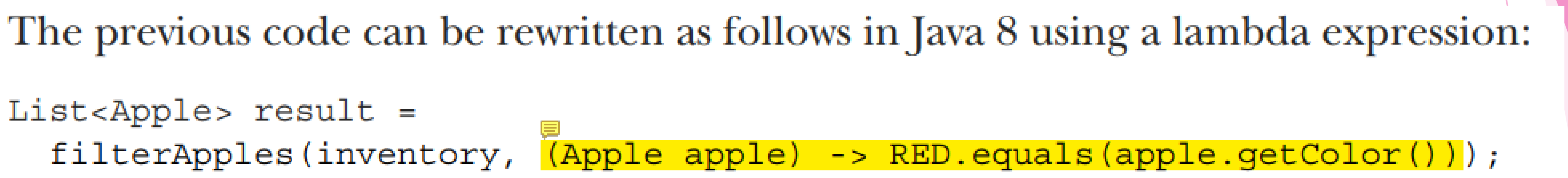
But anonymous classes are still not good enough. First, they tend to be bulky because they take a lot of space, as shown in the boldface code here using the same two examples used previously:

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate() {
    public boolean test(Apple a){
        return RED.equals(a.getColor());
    }
});
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        System.out.println("Whoooo a click!!");
    }
```

Lots of boilerplate code

Good code should be easy to comprehend at a glance.

Sixth attempt: using a lambda expression



The previous code can be rewritten as follows in Java 8 using a lambda expression:

```
List<Apple> result =
  filterApples(inventory, (Apple apple) -> RED.equals(apple.getColor()));
```

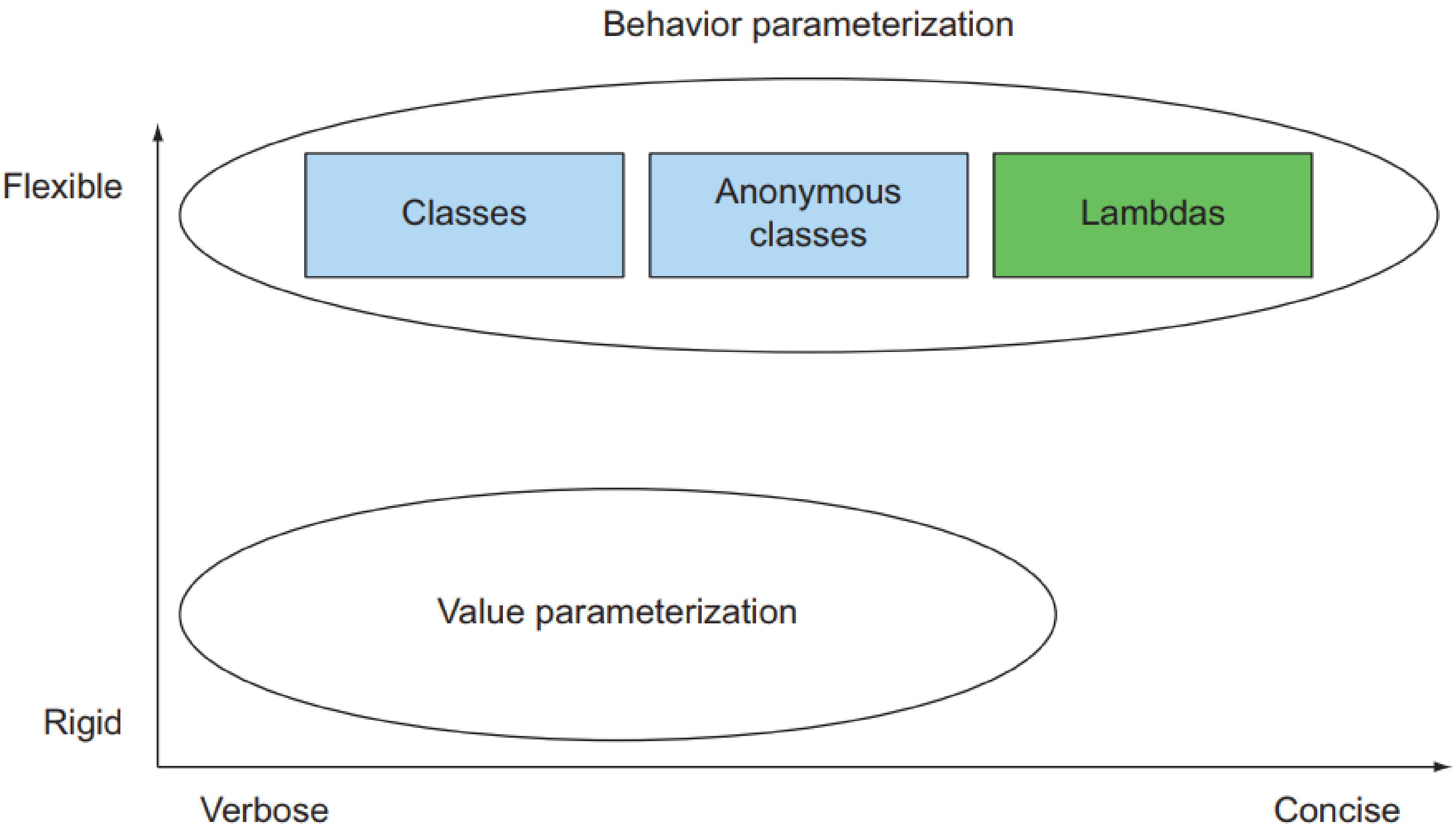
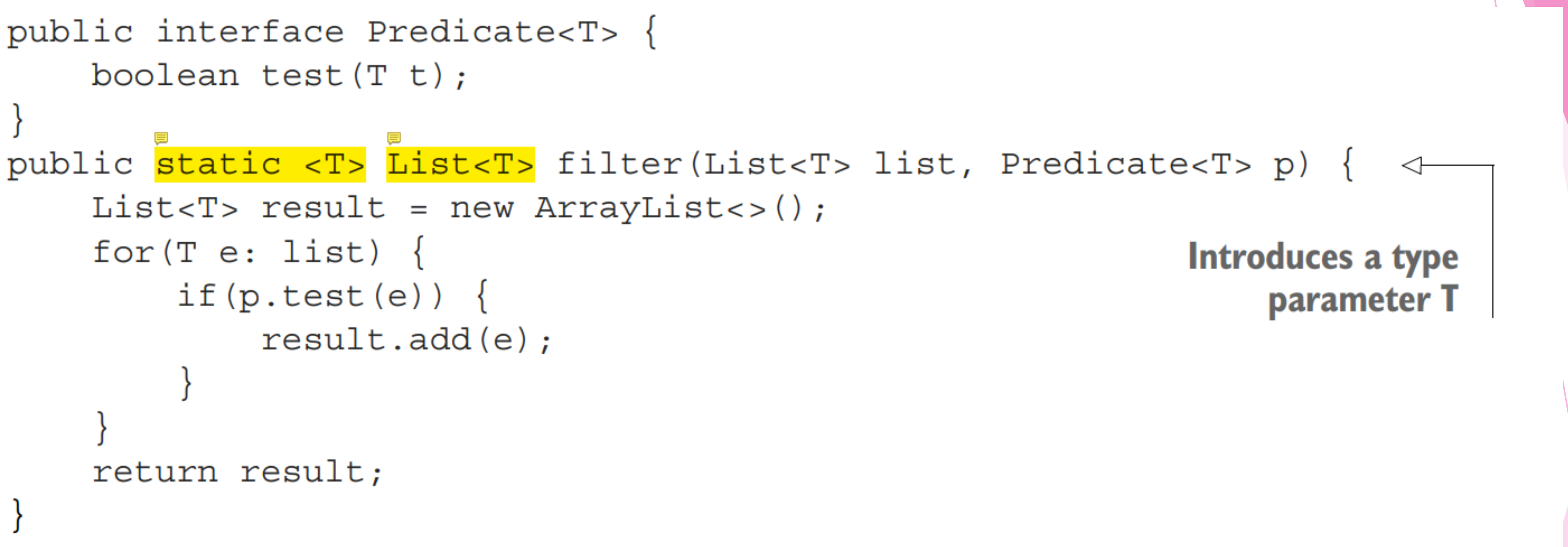


Figure 2.4 Behavior parameterization versus value parameterization

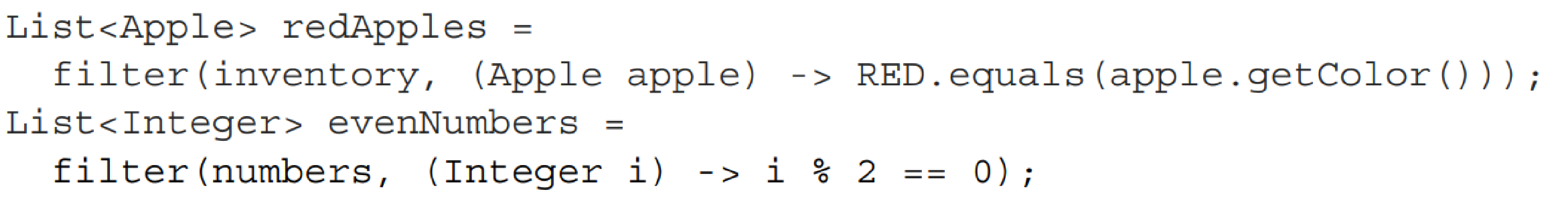
Seventh attempt: abstracting over List type



```
public interface Predicate<T> {
    boolean test(T t);
}
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;
}
```

Introduces a type parameter T

Here’s an example, using lambda expressions:

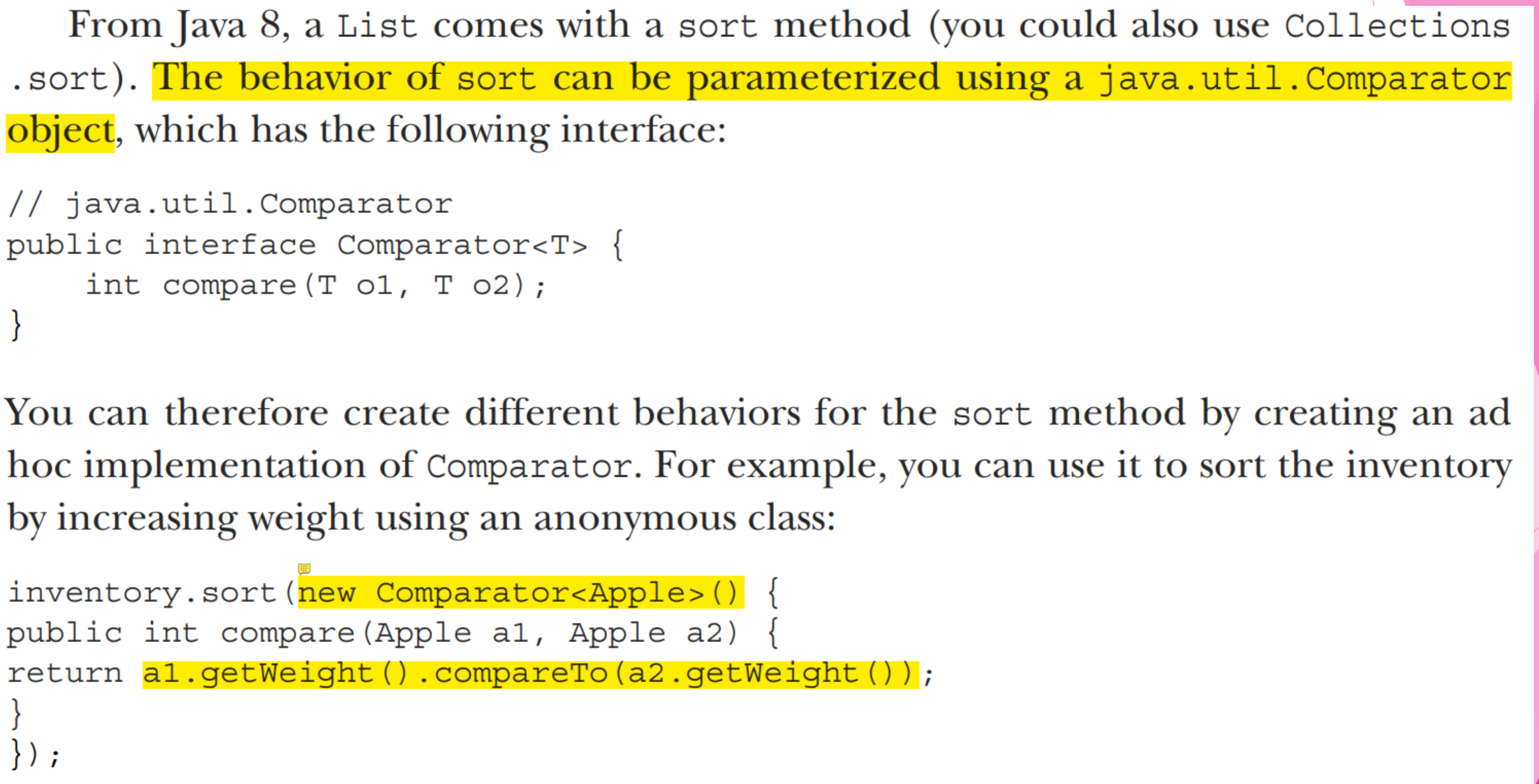


```
List<Apple> redApples =
  filter(inventory, (Apple apple) -> RED.equals(apple.getColor()));
List<Integer> evenNumbers =
  filter(numbers, (Integer i) -> i % 2 == 0);
```

2.4 Real-world examples

- Sorting with a Comparator
- Executing a block of code with Runnable
- Returning a result from a task using Callable
- GUI event handling

Sorting with a Comparator



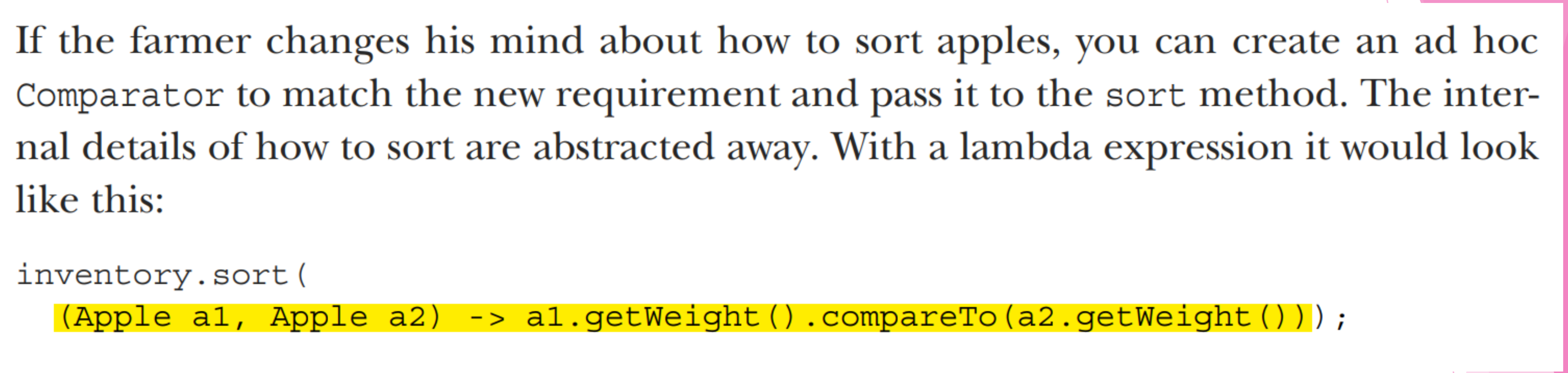
From Java 8, a List comes with a sort method (you could also use Collections.sort). The behavior of sort can be parameterized using a java.util.Comparator object, which has the following interface:

```
// java.util.Comparator
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

You can therefore create different behaviors for the sort method by creating an ad hoc implementation of Comparator. For example, you can use it to sort the inventory by increasing weight using an anonymous class:

```
inventory.sort(new Comparator<Apple>() {
public int compare(Apple a1, Apple a2) {
return a1.getWeight().compareTo(a2.getWeight());
}
});
```

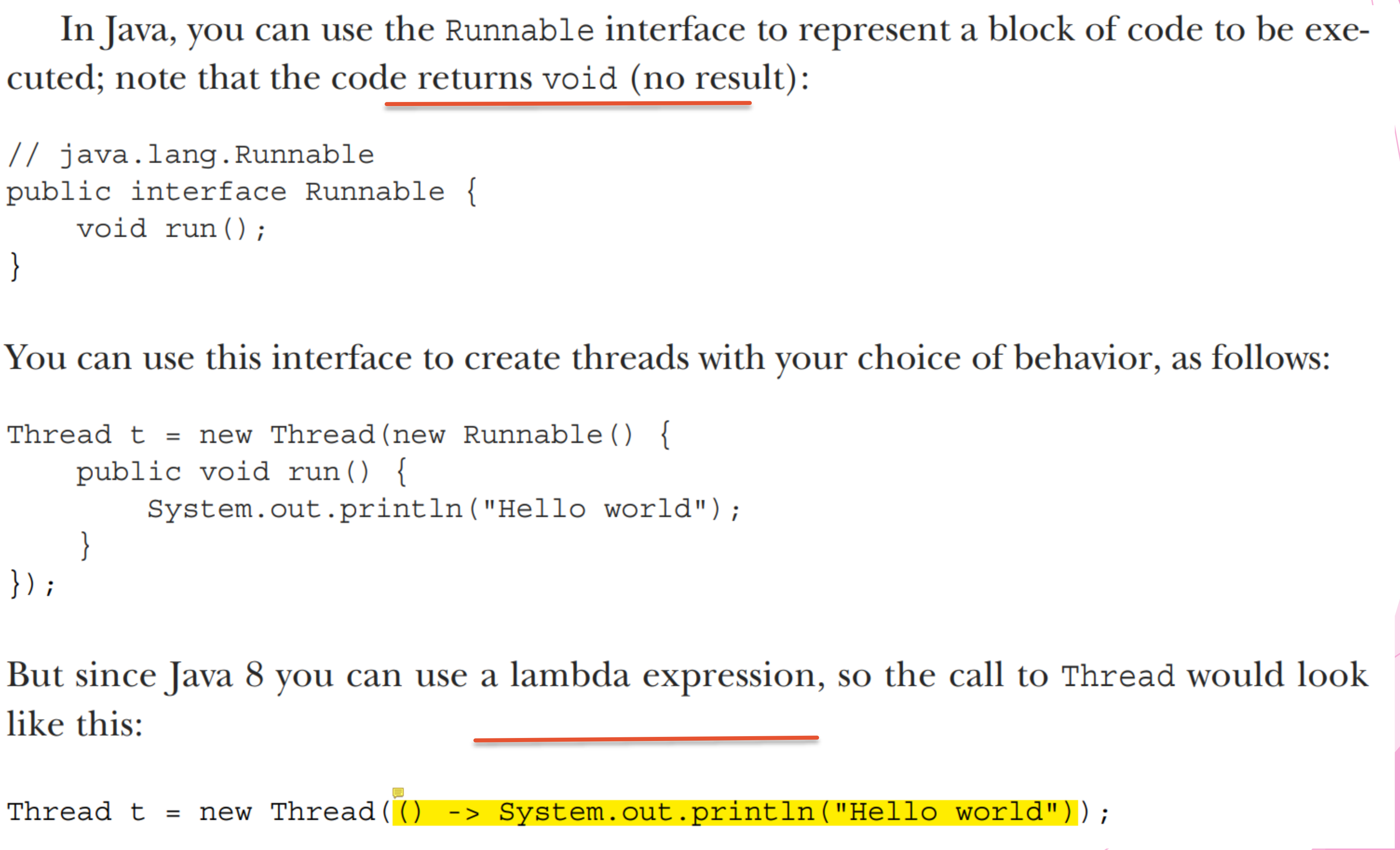
Sorting with a Comparator (Cont.)



If the farmer changes his mind about how to sort apples, you can create an ad hoc `Comparator` to match the new requirement and pass it to the `sort` method. The internal details of how to sort are abstracted away. With a lambda expression it would look like this:

```
inventory.sort(
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

Executing a block of code with Runnable



In Java, you can use the Runnable interface to represent a block of code to be executed; note that the code returns void (no result):

```
// java.lang.Runnable
public interface Runnable {
    void run();
}
```

You can use this interface to create threads with your choice of behavior, as follows:

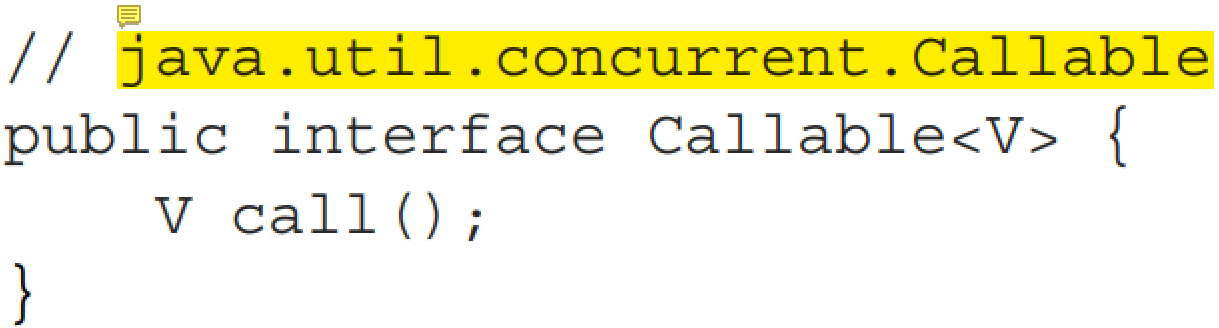
```
Thread t = new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello world");
    }
});
```

But since Java 8 you can use a lambda expression, so the call to Thread would look like this:

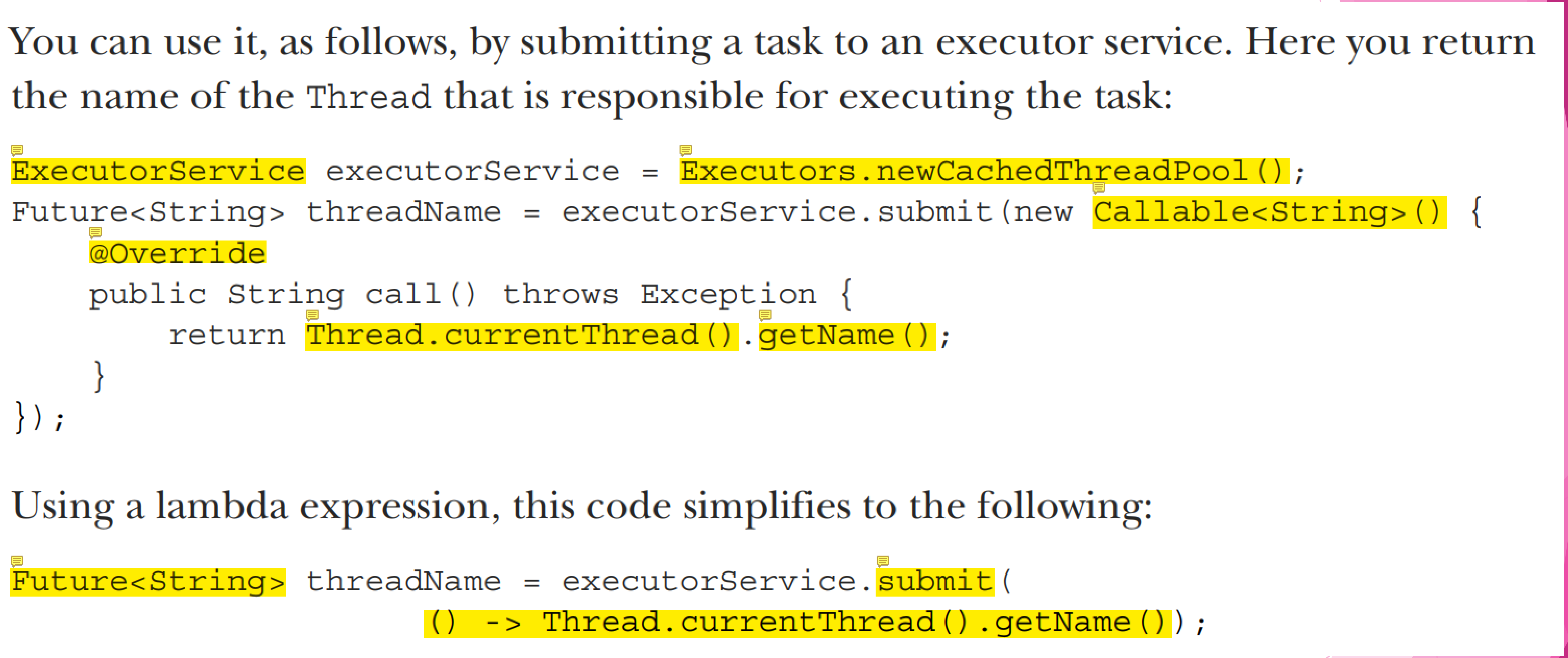
```
Thread t = new Thread(() -> System.out.println("Hello world"));
```

Returning a result from a task using Callable

- The ExecutorService abstraction introduced inJava 5.
- The ExecutorService interface decouples how tasks are submitted and executed.
- By using an ExecutorService you can send a task to a pool of threads and have its result stored in a Future.
- The Callable interface is used to model a task that returns a result.



```
// java.util.concurrent.Callable
public interface Callable<V> {
    V call();
}
```



You can use it, as follows, by submitting a task to an executor service. Here you return the name of the Thread that is responsible for executing the task:

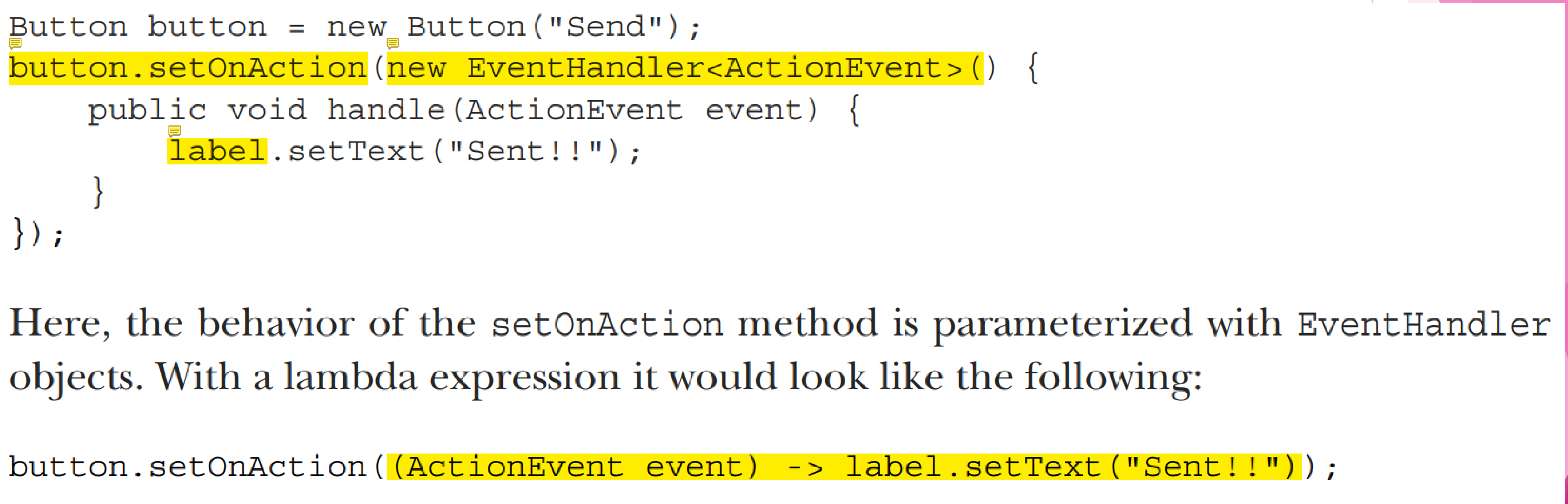
```
ExecutorService executorService = Executors.newCachedThreadPool();
Future<String> threadName = executorService.submit(new Callable<String>() {
    @Override
    public String call() throws Exception {
        return Thread.currentThread().getName();
    }
});
```

Using a lambda expression, this code simplifies to the following:

```
Future<String> threadName = executorService.submit(
                    () -> Thread.currentThread().getName());
```

GUI event handling

- In JavaFX, you can use an EventHandler to represent a response to an event by passing it to setOnAction:



```
Button button = new Button("Send");
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        label.setText("Sent!!");
    }
});
```

Here, the behavior of the setOnAction method is parameterized with EventHandler objects. With a lambda expression it would look like the following:

```
button.setOnAction((ActionEvent event) -> label.setText("Sent!!"));
```

Summary

- Behavior parameterization is the ability for a method to *take* multiple different behaviors as parameters and use them internally to *accomplish* different behaviors.
- Behavior parameterization lets you make your code more adaptive to changing requirements and saves on engineering efforts in the future.
- Passing code is a way to give new behaviors as arguments to a method. But it’s verbose prior to Java 8. Anonymous classes helped a bit before Java 8 to get rid of the verbosity associated with declaring multiple concrete classes for an interface that are needed only once.
- The Java API contains many methods that can be parameterized with different behaviors, which include sorting, threads, and GUI handling.