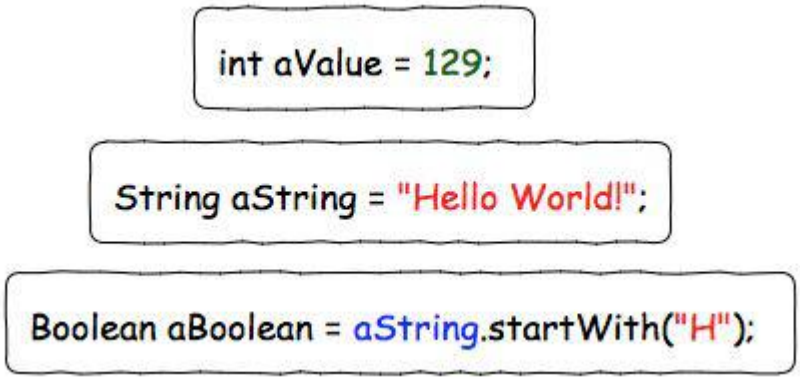
Lambda 表达式有何用处？如何使用？

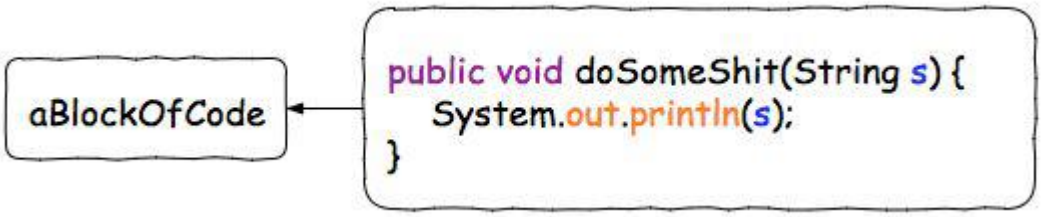
一、什么是 Lambda?

我们知道，对于一个 Java 变量，我们可以赋给其一个**“值”**。

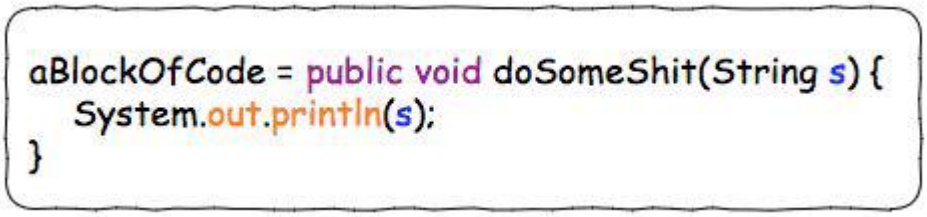


如果你想把**“一块代码”**赋给一个 Java 变量，应该怎么做呢？

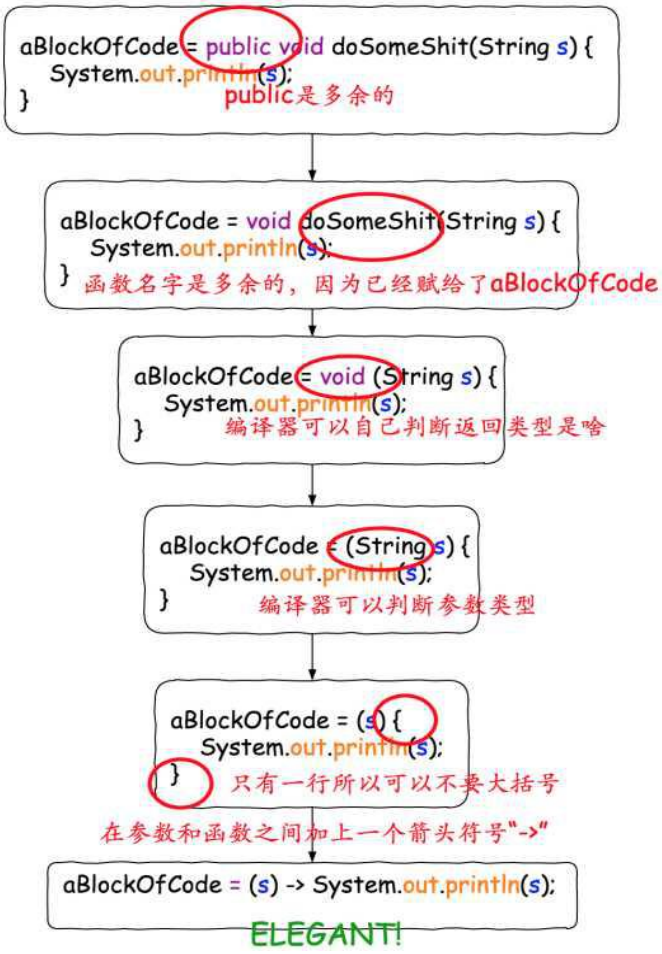
比如，我想把右边那块代码，赋给一个叫做 aBlockOfCode 的 Java 变量：



在 Java 8 之前，这个是做不到的。但是 Java 8 问世之后，利用 Lambda 特性，就可以做到了。



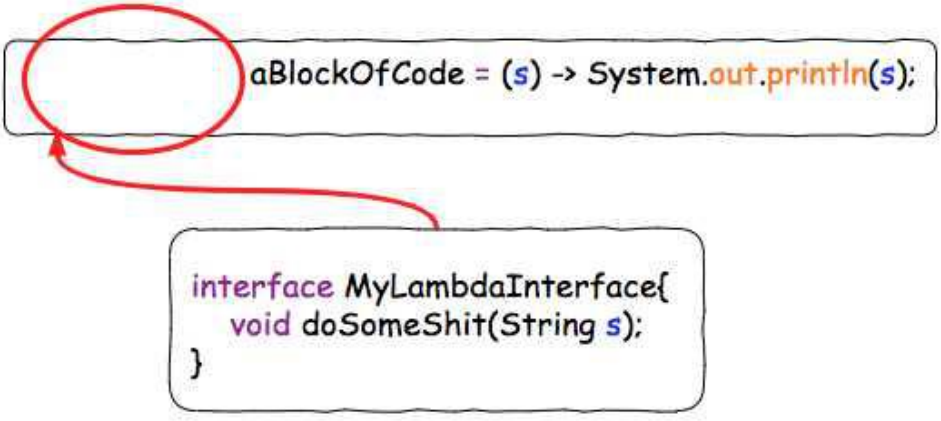
当然，这个并不是一个很简洁的写法。所以，为了使这个赋值操作更加 elegant，我们可以移除



这样，我们就成功的非常优雅的把“一块代码”赋给了一个变量。**而“这块代码”，或者说“这个被赋给一个变量的函数”，就是一个 Lambda 表达式**。

但是这里仍然有一个问题，就是变量 aBlockOfCode 的类型应该是什么？

在 Java 8 里面，**所有的 Lambda 的类型都是一个接口，而 Lambda 表达式本身，也就是”那段代码“，需要是这个接口的实现。**这是我认为理解 Lambda 的一个关键所在，简而言之就是，**Lambda 表达式本身就是一个接口的实现**。直接这样说可能还是有点让人困扰，我们继续看看例子。我们给上面的 aBlockOfCode 加上一个类型：



这种只有**一个接口函数需要被实现的接口类型，我们叫它”函数式接口“。**为了避免后来的人在这个接口中增加接口函数导致其有多个接口函数需要被实现，变成"非函数接口”，我们可以在这个上面加上一个声明@FunctionalInterface，这样别人就无法在里面添加新的接口函数了：

```
@FunctionalInterface
interface MyLambdaInterface{
    void doSomeShit(String s);
}
```

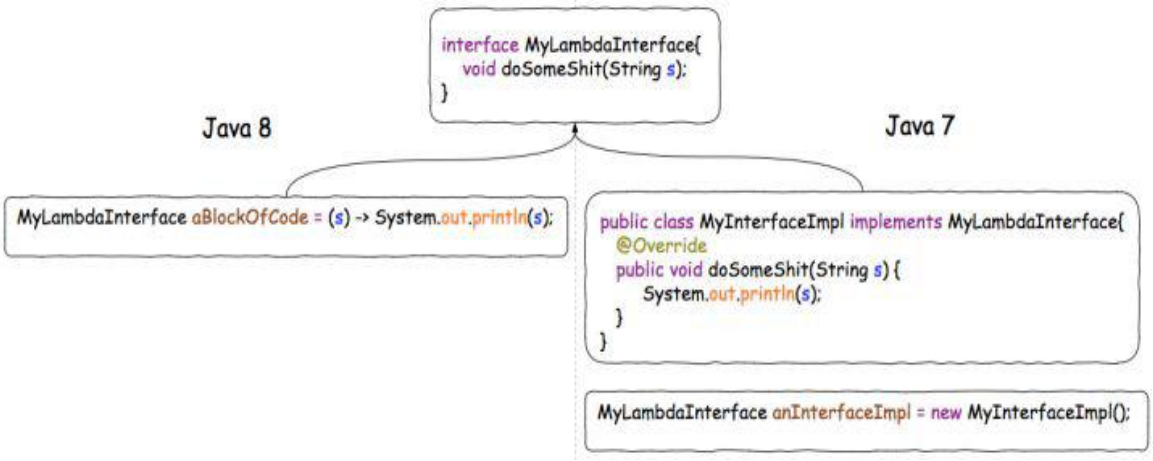
这样，我们就得到了一个完整的 Lambda 表达式声明：

```
MyLambdaInterface aBlockOfCode = (s) -> System.out.println(s);
```

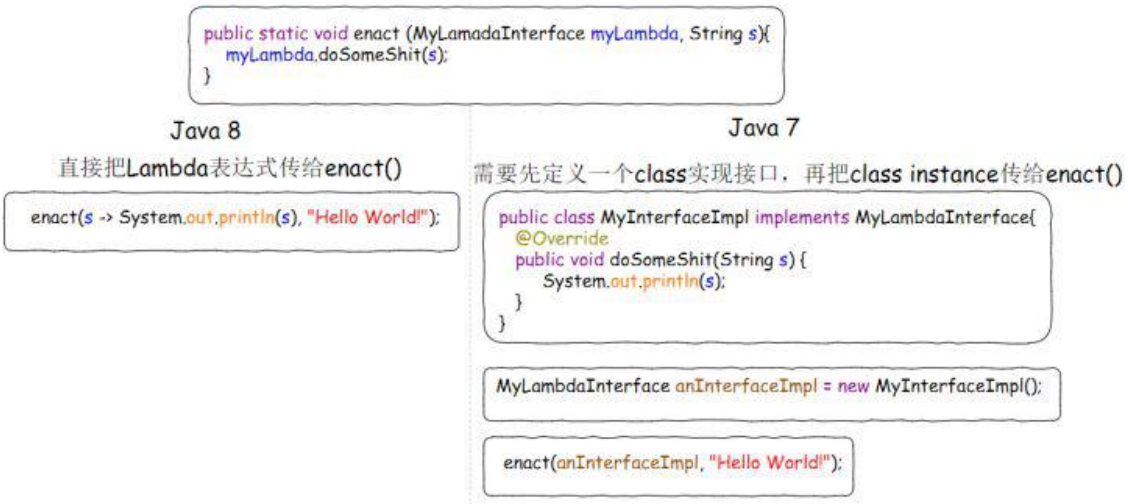
二、Lambda 表达式有什么作用?

最直观的作用就是使得代码变得异常简洁。

我们可以对比一下 Lambda 表达式和传统的 Java 对同一个接口的实现：



这两种写法本质上是等价的。但是显然，Java 8 中的写法更加优雅简洁。并且，由于 Lambda 可以直接赋值给一个变量，**我们就可以直接把 Lambda 作为参数传给函数，而传统的 Java 必须有明确的接口实现的定义，初始化才行：**

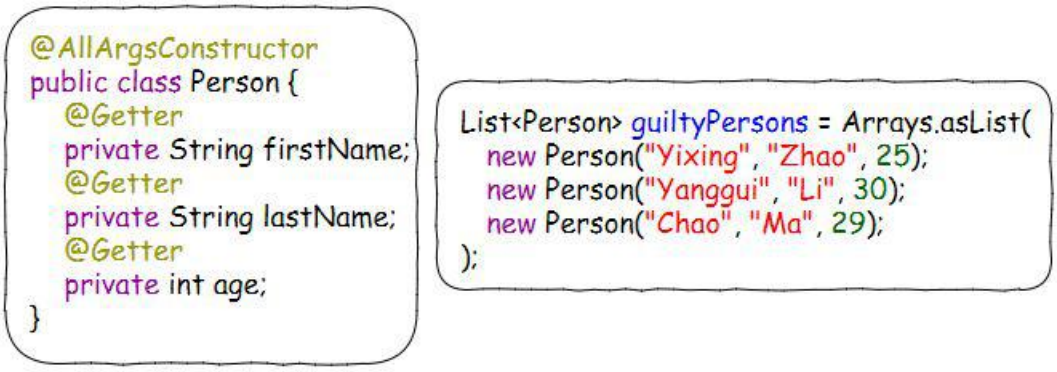


有些情况下，这个接口实现只需要用到一次。传统的 Java 7 必须要求你定义一个“污染环境”的接口实现 MyInterfaceImpl，而相较之下 Java 8 的 Lambda，就显得干净很多。

三、Lambda 结合 FunctionalInterface Lib，forEach，stream()，method reference 等新特性可以使代码变的更加简洁！

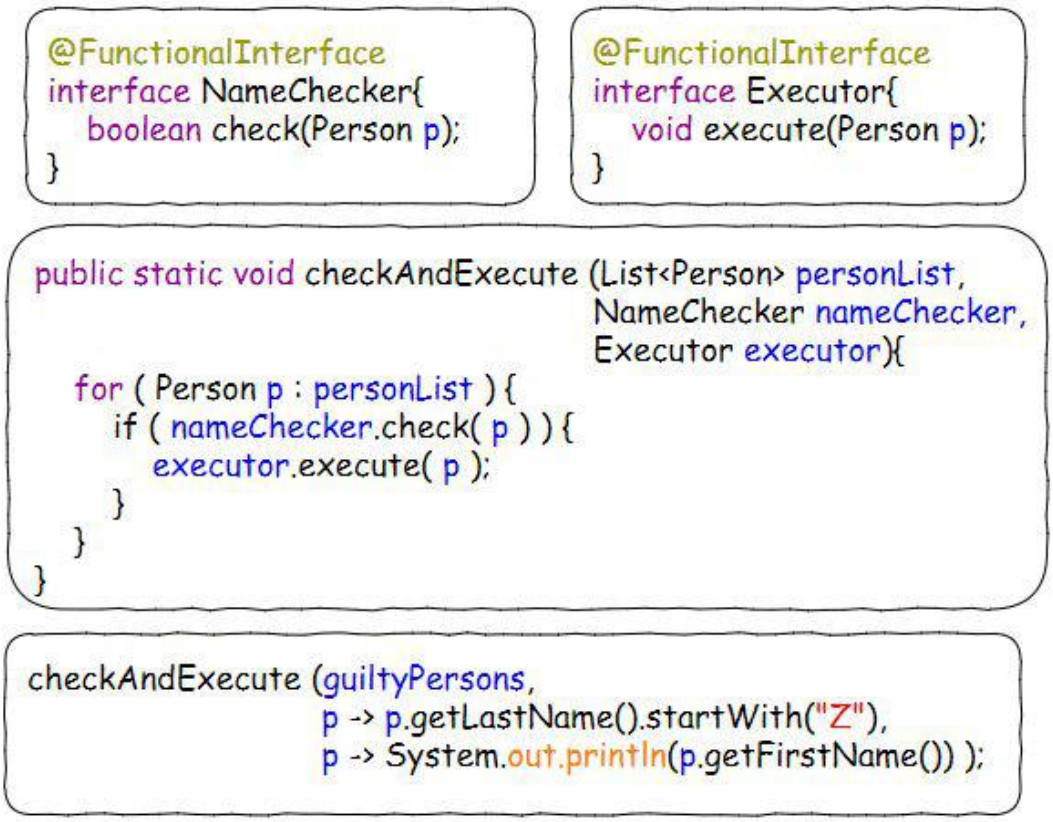
直接上例子。

假设 Person 的定义和 List<Person>的值都给定。



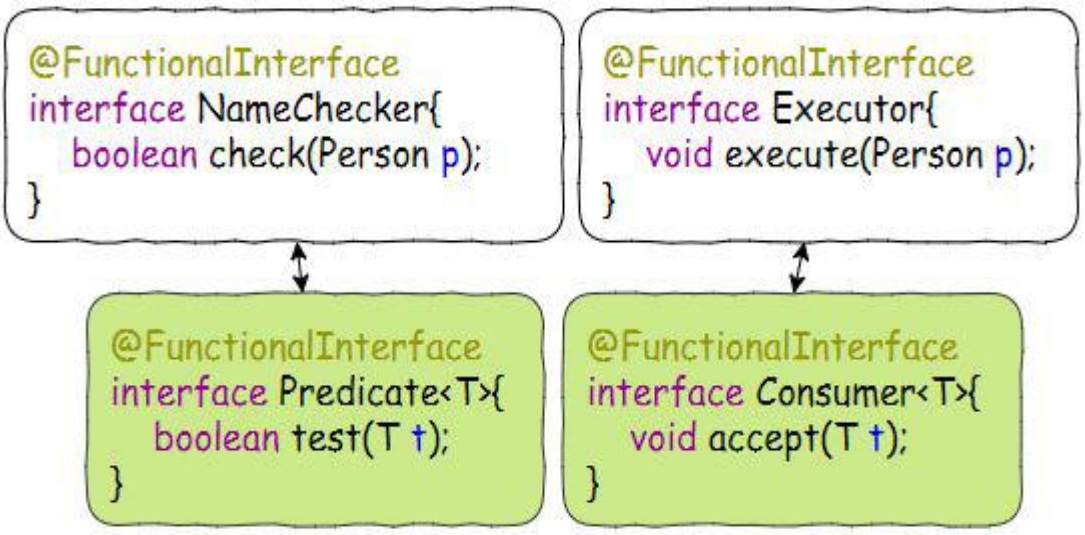
现在需要你打印出 guiltyPersons List 里面所有 LastName 以"Z"开头的人的 FirstName。

原生态 Lambda 写法：定义两个函数式接口，定义一个静态函数，调用静态函数并给参数赋值 Lambda 表达式。



这个代码实际上已经比较简洁了，但是我们还可以更简洁么？

当然可以。在 Java 8 中有一个函数式接口的包，里面定义了大量可能用到的函数式接口（java.util.function (Java Platform SE 8)）。所以，我们在这里压根都不需要定义 NameChecker 和 Executor 这两个函数式接口，直接用 Java 8 函数式接口包里的 Predicate<T>和 Consumer<T>就可以了——因为他们这一对的接口定义和 NameChecker/Executor 其实是一样的。



第一步简化 - 利用函数式接口包：

```
public static void checkAndExecute (List<Person> personList,
                                    Predicate<Person> predicate,
                                    Consumer<Person> consumer){
   for ( Person p : personList ) {
      if ( predicate.test( p ) )
         consumer.accept( p );
   }
}
```

```
checkAndExecute (guiltyPersons,
                 p -> p.getLastName().startWith("Z"),
                 p -> System.out.println(p.getFirstName()) );
```

静态函数里面的 for each 循环其实是非常碍眼的。这里可以利用 Iterable 自带的 forEach()来替代。forEach()本身可以接受一个 Consumer<T> 参数。

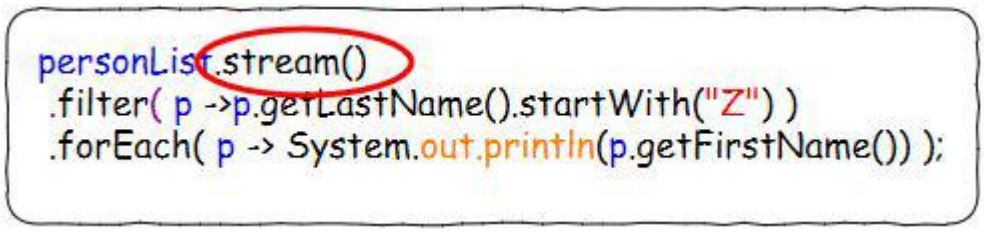
第二步简化 - 用 Iterable.forEach()取代 foreach loop：

```
public static void checkAndExecute (List<Person> personList,
                                    Predicate<Person> predicate,
                                    Consumer<Person> consumer){
   personList.forEach( p -> { if( predicate.test(p) )
                                 consumer.accept(p); });
}
```

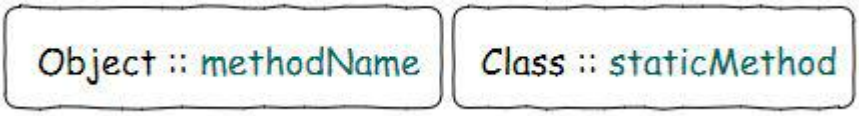
```
checkAndExecute (guiltyPersons,
                 p -> p.getLastName().startWith("Z"),
                 p -> System.out.println(p.getFirstName()) );
```

由于静态函数其实只是对 List 进行了一通操作，这里我们可以甩掉静态函数，直接使用 stream()特性来完成。stream()的几个方法都是接受 Predicate<T>，Consumer<T>等参数的（java.util.stream (Java Platform SE 8)）。你理解了上面的内容，stream()这里就非常好理解了，并不需要多做解释。

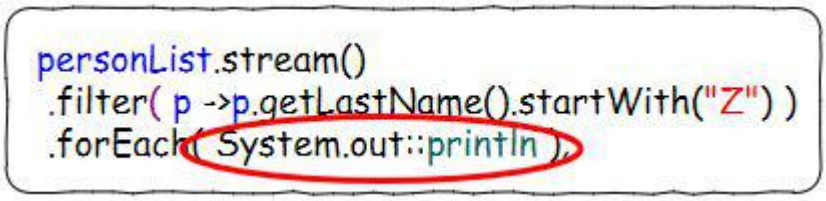
第三步简化 - 利用 stream()替代静态函数：



对比最开始的 Lambda 写法，这里已经非常非常简洁了。但是如果，我们的要求变一下，变成 print 这个人的全部信息，及 p -> System.out.println(p); 那么还可以利用 Method reference 来继续简化。所谓 Method reference，就是用已经写好的别的 Object/Class 的 method 来代替 Lambda expression。格式如下：



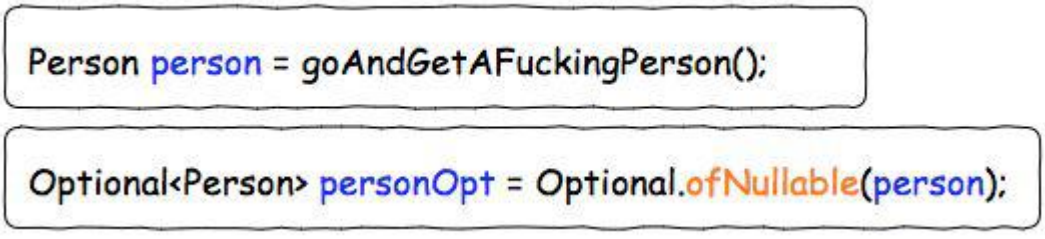
第四步简化 - 如果是 println(p)，则可以利用 Method reference 代替 forEach 中的 Lambda 表达式：



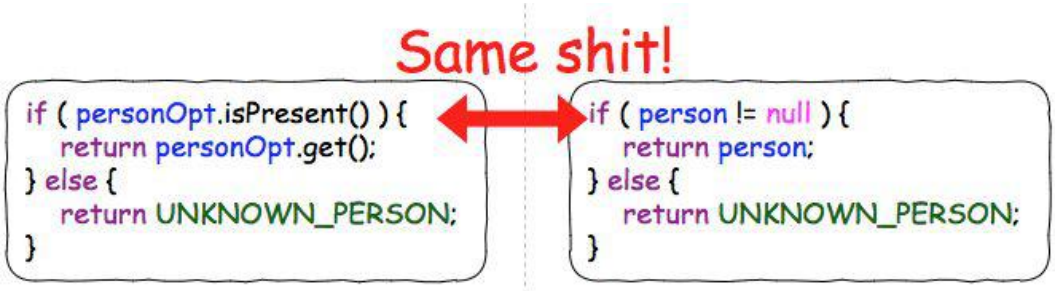
这基本上就是能写的最简洁的版本了。

四、Lambda 配合 Optional<T>可以使 Java 对于 null 的处理变的异常优雅

这里假设我们有一个 person object，以及一个 person object 的 Optional wrapper:



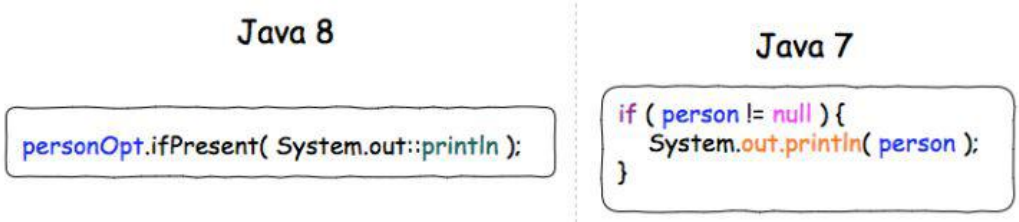
Optional<T>如果不结合 Lambda 使用的话，并不能使原来繁琐的 null check 变的简单。



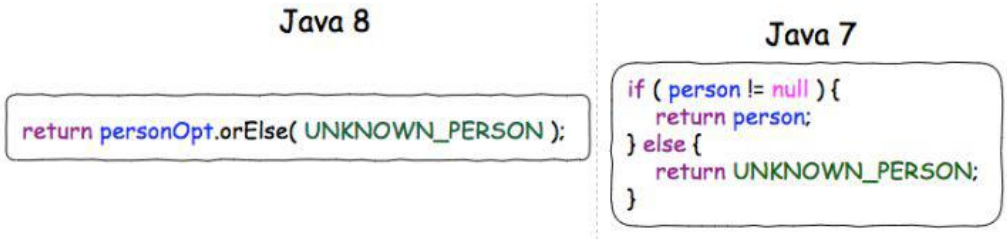
只有当 Optional<T>结合 Lambda 一起使用的时候，才能发挥出其真正的威力！

我们现在就来对比一下下面四种常见的 null 处理中，Java 8 的 Lambda+Optional<T>和传统 Java 两者之间对于 null 的处理差异。

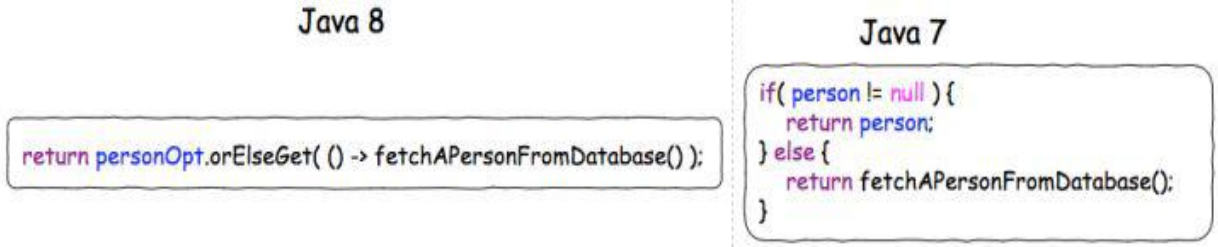
情况一 - 存在则开干



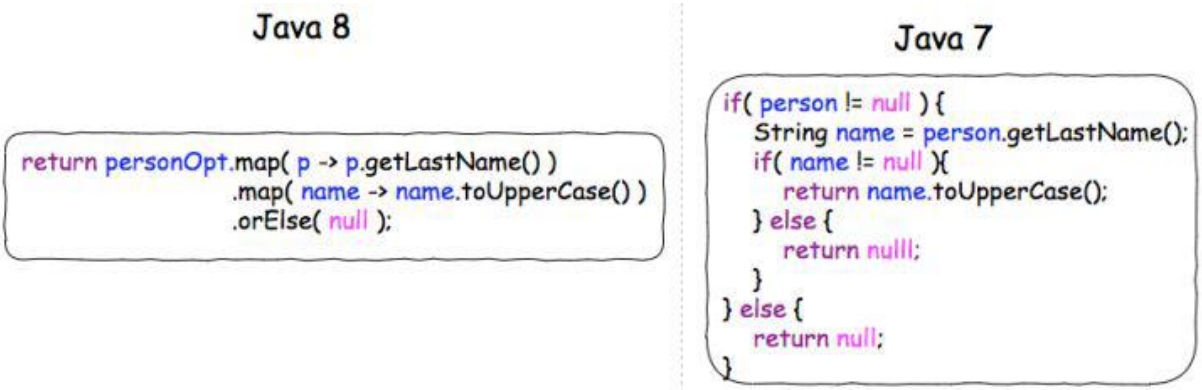
情况二 - 存在则返回，无则返回屁



情况三 - 存在则返回，无则由函数产生



情况四 - 夺命连环 null 检查



由上述四种情况可以清楚地看到，Optional<T>+Lambda 可以让我们少写很多 ifElse 块。尤其是对于情况四那种夺命连环 null 检查，传统 java 的写法显得冗长难懂，而新的 Optional<T>+Lambda 则清新脱俗，清楚简洁。

关于 Java 的 Lambda，还有东西需要讨论和学习。比如如何 handle lambda exception，如何利用 Lambda 的特性来进行 parallel processing 等。总之，我只是一如既往地介绍个大概，让你大概知道，哦！原来是这样子就 OK 了。网上关于 Lambda 有很多相关的教程，多看多练。假以时日，必定有所精益。