Extracted from:

Functional Programming in Java, Second Edition

Harness the Power of Streams and Lambda Expressions

Functional Programming in Java

Second Edition

Harness the Power of Streams and Lambda Expressions

Venkat Subramaniam

Edited by Jacquelyn Carter

Functional Programming in Java, Second Edition

Harness the Power of Streams and Lambda Expressions

Venkat Subramaniam

Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

Sir Charles Antony Richard Hoare's quote is used by permission of the ACM.
Abelson and Sussman's quote is used under Creative Commons license.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Jacquelyn Carter
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

To perseverance.

Hello, Lambda Expressions!

Java has been evolving at a rapid pace in recent years and Java 8 has become the baseline version in many organizations. We can write object-oriented code in Java with the imperative style or functional style and even mix them within the same application as we see fit. The imperative style has been the most popular and practiced way of writing code in Java. But this style, albeit most familiar to developers, is packed with accidental complexity. Many developers are transitioning to use the functional style of programming in Java, more and more each day. Some of the major reasons are the functional style code's reduced complexity and ease of reading and understanding, once the developers get used to the syntax and the paradigm, of course.

With the functional style, the everyday tasks we perform get simpler, easier, and more expressive. We can quickly write concise, elegant, and expressive code with fewer lines of code and errors. We don't have to figure out what the code is doing, it becomes obvious and easy to understand. The benefit is that we can focus on the domain and the inherent capabilities of the application rather than trying to decipher what the code does. We can also use this to easily enforce policies and implement common design patterns without being drowned in verbose long-winded code. The net result is that we can be more productive in creating and delivering applications faster.

In this book, we'll explore the functional style of programming using direct examples of everyday tasks we do as programmers. Before we take the leap to this elegant style, which is a better way to design and program, let's discuss the reasons to change from the familiar imperative style.

Change the Way You Think

Imperative style—that's what Java has provided us with since its inception. In this style, we tell Java every step of what we want it to do and then we

watch it faithfully exercise those steps. That's worked fine, but it's a bit low-level. The code tends to get verbose, and we often wish the language were a tad more intelligent; we could then tell it—declaratively—*what* we want rather than delve into *how* to do it. Thankfully, Java can help us do that. Let's look at a few examples to see the benefits and the differences in style.

The Habitual Way

Let's start on familiar ground to see the two paradigms in action. Here's an imperative way to find if Chicago is in a collection of given cities—remember, the listings in this book only have snippets of code (see How to Read the Code Examples, on page ?).

introduction/fpij/Cities.java
```java
boolean found = false;
for(String city : cities) {
  if(city.equals("Chicago")) {
    found = true;
    break;
  }
}

System.out.println("Found chicago?:" + found);
```

This imperative version is noisy and low-level; it has several moving parts. We first initialize a smelly boolean flag named found and then walk through each element in the collection. If we find the city we're looking for, then we set the flag and break out of the loop. Finally, we print out the result of our finding.

A Better Way

As observant Java programmers, the minute we set our eyes on this code we'd quickly turn it into something more concise and easier to read, like this:

introduction/fpij/Cities.java
```java
System.out.println("Found chicago?:" + cities.contains("Chicago"));
```

That's one example of declarative style—the contains() method helped us get directly to our business.

Tangible Improvements

That change improved our code in a few ways:

- No messing around with mutable variables
- Iteration steps wrapped under the hood
- Less clutter
- Better clarity; retains our focus

- Less impedance; code closely trails the business intent
- Less error prone
- Easier to understand and maintain

Beyond Simple Cases

That was simple—the declarative function to check if an element is present in a collection has been around in Java for a long time. Now imagine not having to write imperative code for more advanced operations, like parsing files, working with databases, making calls to web services, *programming concurrency*, and so on. Java now makes it possible to write concise, elegant, less error-prone code, not just for simple cases, but throughout our applications.

The Old Way

Let's look at another example. We'll define a collection of prices and try out a few ways to total discounted price values.

```
final List<Integer> prices = Arrays.asList(10, 30, 17, 20, 18, 45, 12);
```

Suppose we're asked to total the prices greater than $20, discounted by 10%. Let's do that in the habitual Java way first.

```
introduction/fpij/DiscountImperative.java
double totalOfDiscountedPrices = 0.0;

for(int price : prices) {
  if(price > 20) {
    totalOfDiscountedPrices += price * 0.9;
  }
}
System.out.println("Total of discounted prices: " + totalOfDiscountedPrices);
```

That's familiar code; we start with a mutable variable to hold the total of the discounted prices. We then loop through the prices, pick each price greater than $20, compute each item's discounted value, and add those to the total. Finally, we print the total value of the discounted prices.

And here's the output from the code.

```
Total of discounted prices: 67.5
```

It worked, but writing it feels dirty. It's no fault of ours; we had to use what was available. But the code is fairly low-level—it suffers from "primitive obsession" and defies the single-responsibility principle (SRP). And the code also violates the single level of abstraction principle (SLAP) since it delves into multiple nested levels of details. It first works with the collection of prices, then for each price it examines the value, and then for each selected value

(three levels down in nesting) it computes the products and performs the sum. In addition to having to work with a poor design, those of us working from home have to keep this code away from the eyes of kids aspiring to be programmers, for they may be dismayed and sigh, "That's what you do for a living?"

A Better Way, Again

We can do better—a lot better. Our code can resemble the requirement specification. This will help reduce the gap between the business needs and the code that implements them, further reducing the chances of the requirements being misinterpreted.

Rather than tell Java to create a mutable variable and then to repeatedly assign a value to it, let's talk with it at a higher level of abstraction, as in the next code.

```
introduction/fpij/DiscountFunctional.java
final double totalOfDiscountedPrices =
  prices.stream()
        .filter(price -> price > 20)
        .mapToDouble(price -> price * 0.9)
        .sum();

System.out.println("Total of discounted prices: " + totalOfDiscountedPrices);
```

Let's read that aloud—filter prices greater than $20, map the prices to discounted values, and then add them up. The code flows along with logic in the same way we'd describe the requirements. As a convention in Java, we wrap long lines of code and line up the dots vertically before the method names, as in the previous example.

The code is concise, but we're using a number of new things from modern Java. To start with, we invoke a stream() method on the prices list. This opens the door to a *special* iterator with a wealth of convenience functions, which we'll discuss later.

Instead of explicitly iterating through the prices list, we're using a few special methods, such as filter() and map(). Unlike the methods we were used to in older versions of Java and the Java Development Kit (JDK), these methods take an anonymous function—a lambda expression—as a parameter, within the parentheses (). (We'll soon explore this further.) We invoke the sum() method, which is a special form of the reduce() operation, to compute the total on the result of the mapToDouble() which is a special form of the map() method.

The looping is concealed much like it was under the `contains()` method. But the `map()` method (and the `filter()` method) is more sophisticated. For each price from the `prices` list that passes through the `filter()`, the `mapToDouble()` method invokes the provided lambda expression to transform the given price to a discounted price. The `sum()` method, which is a `reduce` operation, adds up the transformed values to provide the final result.

Here's the output from this version of code:

```
Total of discounted prices: 67.5
```

The Improvements

You can see several improvements compared to the habitual way:

- Nicely composed, not cluttered
- Free of low-level operations
- Easier to enhance or change the logic
- Iteration controlled by a library of methods
- Efficient; lazy evaluation of loops
- Easier to parallelize where desired

Later we'll discuss how Java provides these improvements.

Lambdas to the Rescue

Lambdas, which are anonymous functions, are the functional key to relieve us from the hassles of imperative programming. Unlike methods, which belong to a class, lambdas are free-standing functions we can create within methods. By changing the way we program and by making use of lambdas, we can write code that's not only elegant and concise but also less prone to errors; more efficient; and easier to optimize, enhance, and parallelize.