Extracted from:

Functional Programming in Java

Harnessing the Power of Java 8 Lambda Expressions

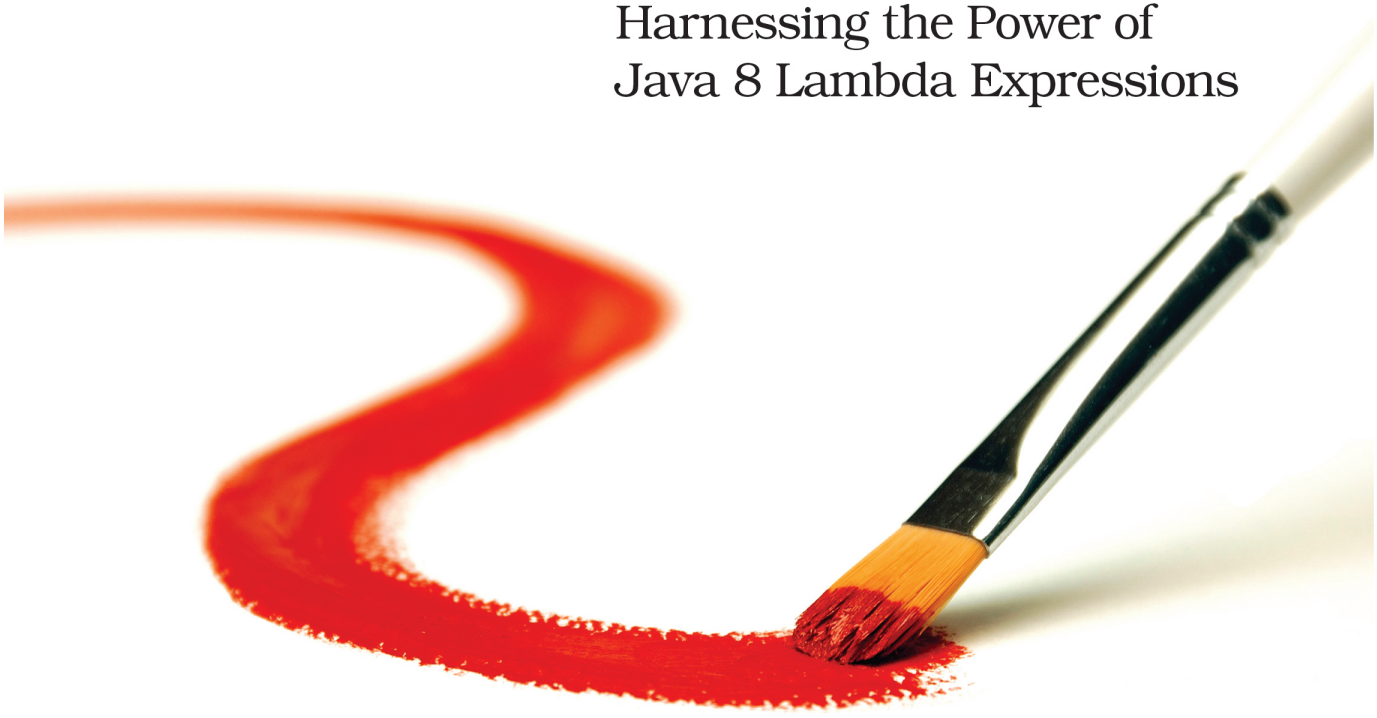
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Functional Programming in Java

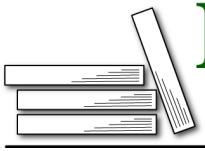
Harnessing the Power of Java 8 Lambda Expressions



Venkat Subramaniam

Foreword by Brian Goetz

Edited by Jacquelyn Carter



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://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:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (indexer)
Candace Cunningham (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

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

Printed in the United States of America.
ISBN-13: 978-1-937785-46-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2014

To the loving memory of my grandmothers,
Kuppammal and Jayalakshmi. I cherish my
wonder years under your care.

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.[1]

➤ *Sir Charles Antony Richard Hoare*

CHAPTER 1

Hello, Lambda Expressions!

Our Java coding style is ready for a remarkable makeover.

The everyday tasks we perform just got simpler, easier, and more expressive. The new way of programming in Java has been around for decades in other languages. With these facilities in Java we can write concise, elegant, and expressive code with fewer errors. We can use this to easily enforce policies and implement common design patterns with fewer lines of code.

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, and this new way to design and program, let's discuss why it's better.

Change the Way You Think

Imperative style—that's what Java has provided us 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 now 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

1. Hoare, Charles Antony Richard, "The Emperor's Old Clothes," *Communications of the ACM* 24, no. 2 (February 1981): 5–83, doi:10.1145/358549.358561.

listings in this book only have snippets of code (see *How to Read the Code Examples, on page ?*).

introduction/fpij/Cities.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 found 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

```
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 quite 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 very 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<BigDecimal> prices = Arrays.asList(
  new BigDecimal("10"), new BigDecimal("30"), new BigDecimal("17"),
  new BigDecimal("20"), new BigDecimal("15"), new BigDecimal("18"),
  new BigDecimal("45"), new BigDecimal("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

```
BigDecimal totalOfDiscountedPrices = BigDecimal.ZERO;

for(BigDecimal price : prices) {
  if(price.compareTo(BigDecimal.valueOf(20)) > 0)
    totalOfDiscountedPrices =
      totalOfDiscountedPrices.add(price.multiply(BigDecimal.valueOf(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. 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

Now 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 it, further reducing the chances of the requirements being misinterpreted.

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

introduction/fpij/DiscountFunctional.java

```
final BigDecimal totalOfDiscountedPrices =
  prices.stream()
        .filter(price -> price.compareTo(BigDecimal.valueOf(20)) > 0)
        .map(price -> price.multiply(BigDecimal.valueOf(0.9)))
        .reduce(BigDecimal.ZERO, BigDecimal::add);

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 before the method names, as in the previous example.

The code is concise, but we're using quite a number of new things from Java 8. First, we invoked 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're used to in 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 reduce() method to compute the total on the result of the map() method.

The looping is concealed much like it was under the contains() method. The map() method (and the filter() method), however, is more sophisticated. For each price in the prices list, it invokes the provided lambda expression and puts the responses from these calls into a new collection. The reduce() method is invoked on this collection to get the final result.

Here's the output from this version of code:

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

The Improvements

This is quite an improvement from 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 are the functional key to free us from the hassles of imperative programming. By changing the way we program, with a feature now baked into Java, 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.