The Pragmatic Programmers

Functional Programming in Java

Harnessing the Power of Java 8 Lambda Expressions

Venkat Subramaniam

Edited by Jacquelyn Carter

Functional Programming in Java

Harnessing the Power of Java 8 Lambda Expressions

Venkat Subramaniam

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

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*.

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: B1.0—March 6, 2013

Preface

You're in for a treat. One of the most prominent and widely used languages in the world has evolved, for the better. Until now Java gave us one set of tools—OO—and we did the best we could with it. Now, in addition, there's another way to solve more elegantly the common problems we encounter when developing applications. We can now do quite effectively in Java what was possible on the JVM only using other languages—this means truly more power for everyone using Java.

In the past few decades, I'm thankful for the privilege to program with a few languages: C, C++, Java, C#, F#, Ruby, Groovy, Scala, Clojure, Erlang, JavaScript... When asked which one's my favorite, my resounding answer has been: it's not the language that excites me, but the way we program.

The science and engineering in programming is what drew me in, but it's the art in programming that keeps me. Coding has a lot in common with writing —there's more than one way to express our ideas. Java helped us so far to write code using objects. Now we have an additional way to implement our designs and ideas.

This is a new way in Java, one that will make our code more expressive, easier to write, less error-prone, and easier to parallelize than we were able to do with Java until now. This way has been around for decades and widely used in languages like Lisp, Clojure, Erlang, Scala, Groovy, Ruby... This is not only a new way in Java, but you'll find it to be a better way as well.

Since coding is like writing, we can learn a few things from that field. William Zinsser recommends simplicity, clarity, and brevity in *On Writing Well [Zin01]*. To create better applications, we can start by making the code simpler, clearer, and concise. The new style of programming in Java provides exactly that, as we will explore throughout this book.

Who's this book for

This book is for programmers well versed with object-oriented programming in Java who are keen to learn and apply the new facilities of lambda expressions. You'll need good experience programming in previous versions of Java, especially Java 5, to make the best use of this book.

Programmers mostly interested in other JVM languages like Scala, Groovy, JRuby, and Closure can benefit from the examples in this book and can relate back to the facilities offered in the respective languages. They can also use the examples to help fellow Java programmers in their teams.

Programmers experienced with the functional style of programming in other languages who are now involved in Java projects can use this book as well. They can learn how what they know translates to the specifics of the lambda expressions usage in Java. Programmers who're familiar with lambda expressions in Java can use this book to help coach and train their team members who are getting up to speed in this area.

What's in this book

This book will help you get up to speed with Java 8 lambda expressions, to think in the elegant style, and benefit from the additions to the Java JDK library. We'll take an example driven approach to explore the concepts. Rather than discuss the theory of functional programming, we'll dive into specific day-to-day tasks to apply the elegant style. This approach will quickly help to get these concepts under our belts, so we can put them to real use on projects right away.

On the first read, take the time to go over the chapters sequentially, as we build upon previously discussed concepts and examples. Later, when working on applications, take a quick glance at any relevant example or section in the book. There's also an appendix of syntax for quick reference.

Here's how the rest of the book is organized:

We discuss the functional style of programming, its benefits, and how it differs from the prevalent imperative style in Chapter 1, *Hello Lambda Expressions*, on page ?. We also look into how Java supports lambda expressions in this chapter.

The JDK collections have received some special treatment in Java 8, with new interfaces, classes, and methods that support functional style operations. We will explore these in Chapter 2, *Using Collections*, on page ?.

In Chapter 3, *Strings, Comparators, and Filters*, on page ? we exploit functional style and lambda expressions to work with strings, implement the Comparator interface, and filters for file selection.

In addition to using the functional style facilities in the JDK, we can benefit from applying the elegant style in the design of methods and classes we create. We'll pick up functional style design techniques in Chapter 4, *Designing with Lambda Expressions*, on page ?.

The lambda expressions facilitate a code structure that helps delineate operations to manage object lifetimes and resource clean up as we'll see in Chapter 5, *Working with Resources*, on page ?.

We'll see lambda expressions shine in Chapter 6, *Being Lazy*, on page ?, to provide us the ability to postpone instance creation and method evaluations, create infinite lazy collections, and thereby improve the performance of the code.

In Chapter 7, *Optimizing Recursions*, on page ? we will use lambda expressions to optimize recursions and achieve stellar performance using memoization techniques.

We'll put the techniques we cover in the book to some real use in Chapter 8, *Composing with Lambda Expressions*, on page ? where we will transform objects, implement map-reduce, and safely parallelize a program with little effort.

Finally, in Chapter 9, *Bringing it all Together*, on page ? we will go over the key concepts and the practices needed to adopt them.

In Appendix 1, *Starter Set of Functional Interfaces*, on page ? we'll take a glance at some of the most popular functional interfaces.

A quick overview of the Java 8 syntax for functional interfaces, lambda expressions, and method/constructor references is in Appendix 2, *Syntax Overview*, on page ?.

The URLs mentioned throughout the book are gathered together for convenience in Appendix 3, *Web Resources*, on page ?.

Java Version used in this book

To run the examples in this book you need Java 8 with support for lambda expressions. Using automated scripts, the examples in this book have been tried out with the following version of Java:

```
openjdk version "1.8.0-ea"
OpenJDK Runtime Environment (build 1.8.0-ea-lambda-nightly-h3419-20130219-b78-b00)
OpenJDK 64-Bit Server VM (build 25.0-b15, mixed mode)
```

Take a few minutes to download the appropriate version of Java depending on the system. This will help you practice the examples in the book as you follow along.

How to read the code examples

When writing code in Java, we place classes in packages and executable statements and expressions in methods. In order to reduce clutter and save pages in the book, we'll skip the package names and imports in the code listing. All code in this book belong to a package:

```
package fpij;
```

Any executable code not listed within a method is part of an undisplayed main() method. When going through the code listings, if there's an urge to look at the full source code, remember it's only a click away at the website for the book.

Online Resources

A number of web resources referenced through out the book are collected in Appendix 3, *Web Resources*, on page ?. Here are a few that will help you get started with this book:

The Oracle website for downloading the version of Java used in this book: http://jdk8.java.net/lambda

The official homepage for this book at the Pragmatic Bookshelf website is http://www.pragprog.com/titles/vsjava8. From there you can download all the example source code for this book. You can also offer feedback by submitting errata entries or posting your comments and questions in the forum for the book.

If you're reading the book in the PDF form, you can click on the link above a code listing to view or download the specific examples.

Now for some fun with lambda expressions…

Our Java coding style is ready for a remarkable makeover. The common everyday tasks we perform just got simpler, easier, and more expressive. The new way of programming that's now part of 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 make use of 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 this change is better.

1.1 Why embrace another paradigm?

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 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 take a look at a few examples, to see the benefits and the differences in style.

Let's start from familiar grounds 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 Section 4, *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.

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 so:

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

That's one example of declarative style. Instead of beating around a mutable variable with low level commands, the contains() method helped us get directly to our business. It wrapped under the covers the steps necessary to loop through the elements; the declarative style removed the clutter and let us focus on the core behavior we like to implement. The benefit—the code reads pretty close to our business intent. Fewer lines of infrastructure code means the code is less error-prone and easier to maintain.

That 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, *concurrent programming*, etc. Java now makes it possible to write concise, elegant, less error-prone code, not just for simple cases, but throughout our applications.

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, 15, 20, 25, 30, 45, 50);
```

Suppose we're asked to total the prices 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) {
  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, compute the discounted value for each price, one at a time, and add that 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: 175.5
```

It worked, but writing it feels dirty. It's no fault of ours, we had to make use of what was available. But, the code is fairly low level, it suffers from "primitive obsession." 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?"

Now we can do better, a lot better, in such a way that the code resembles 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 that works in Java 8.

introduction/fpij/DiscountFunctional.java

```
final double totalOfDiscountedPrices =
  prices.stream().map((Integer price) -> price * 0.9).sum();

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

Let's read that aloud—map the prices to discounted values and then sum them up. The code flows along with logic in the same way we'd describe the requirements.

The code is concise, but we're making use of 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 will discuss later.

Instead of explicitly iterating through the prices list, we're using a special map method. Unlike the methods we're used to in Java and the JDK, this method takes an anonymous function—a lambda expression—as a parameter, within the parenthesis (). We'll soon explore this further. On the result of the map() method we invoke the sum() method to compute the total.

Much like the way the looping was concealed under the contains() method, the looping is concealed in this version also. The map() method, however, is more sophisticated, for each price in the prices list, it invokes the provided lambda expression and collects back the responses from these calls into a new collection. The sum() method is finally invoked on this collection to get the final result.

Here's the output from this version of code:

```
Total of discounted prices: 175.5
```

Now that we've gotten a taste of the declarative and functional style, let's visit their benefits.

The JDK has evolved to include convenience methods that promote functional style. When using familiar classes and interfaces from the library, like String for example, we need to look for opportunities to use these newer functions in place of the old style. Also, anywhere we used an anonymous inner class with just one method, we can now use lambda expressions to reduce clutter and ceremony.

In this chapter we'll use lambda expressions and method references to iterate over a String, to implement Comparators, to list files in a directory, and to observe file and directory changes. Quite a few methods introduced in the previous chapter will appear here again to help with the current tasks on hand. Techniques we pick up along the way will help turn long mundane tasks into concise code snippets we can quickly write and easily maintain.

3.1 Iterating a String

chars() is a new method in the String class from the CharSequence interface. It's useful to fluently iterate over the String's characters. We can use this convenient internal iterator to apply an operation on the individual characters that make up the string. Let's make use of it in an example to process a string. Along the way we will pick up a few more useful ways to use method references.

compare/fpij/IterateString.java

```
final String str = "w00t";

str.chars().forEach(ch -> System.out.println(ch));
```

The chars() method returns a Stream over which we can iterate, using the forEach() internal iterator. We get direct read access to the characters in the String within the iterator. Here's the result when we iterate and print each character.

```
119
48
48
116
```

The result is not quite what we'd expect. Instead of seeing characters we're seeing some numbers. That's because the chars() method returns a stream of Integers, representing the characters instead of a stream of Characters. Let's explore the API a bit further before we fix the output.

In the previous code we created a lambda expression in the argument list for the forEach() method. The implementation was a simple call where we routed the parameter directly as an argument to the println() method. Since this is a trivial operation, we can eliminate this mundane code with the help of the Java compiler. We can rely on it to do this parameter routing for us, using a method reference like we did back in *Using Method References*, on page ?.

We already saw how to create a method reference for an instance method. For example, for the call name.toUpperCase(), the method reference is String::toUpperCase. In this example, however, we have a call on a static reference System.out. We can use either a class name or an expression to the left of the double colon in method references. Using this flexibility, it's quite easy to provide a method reference to the println() method, as we see next.

compare/fpij/IterateString.java

```
str.chars().forEach(System.out::println);
```

In this example we see the smarts of the Java compiler for parameter routing. Recall lambda expressions and method references may stand in where implementations of functional interfaces are expected and the Java compiler synthesizes the appropriate method in-place (see Section 1.5, *Syntax Sugar with Functional Interfaces*, on page ?). In the earlier method reference we used, String::toUppercase; the parameter to the synthesized method turned into the target of the method call, like so: parameter.toUppercase();. That's because the method reference is based off a class name (String). In this example, the method reference, again to an instance method, is based off an expression, an instance of PrintStream accessed through the static reference System.out. Since we already provided a target for the method, the Java compiler decided to use the parameter of the synthesized method as an argument to the referenced method, like so: System.out.println(parameter);. Sweet.

The code with method reference is quite concise, but we have to dig into it a bit more to understand what's going on here. Once we get used to method references, our brains will know to auto-parse these.

In this example, while the code is concise, the output is not satisfactory. We want to see characters and not numbers in their place. To fix that, let's write a convenience method that takes an int and prints it as a character.

compare/fpij/IterateString.java

```
private static void printChar(int aChar) {
  System.out.println((char)(aChar));
}
```

We can use a method reference to this convenience method to fix the output.

compare/fpij/IterateString.java

```
str.chars().forEach(IterateString::printChar);
```

We can continue to use the result of chars() as an int and when it's time to print we can convert it to a character. The output of this version will display characters.

```
w
0
0
t
```

If we want to process characters and not int from the start, we can convert the ints to characters right after the call to the chars() method, like so:

compare/fpij/IterateString.java

```
str.chars()
.map(ch -> Character.valueOf((char)ch))
.forEach(System.out::println);
```

We used the internal iterator on the Stream returned by the chars() method, but we're not limited to that method. Once we get a Stream we can use any methods available on it, like map(), filter(), reduce(), etc. to process the characters in the string. For example, we can filter out only digits from the string, like so,

compare/fpij/IterateString.java

```
str.chars()
  .filter(ch -> Character.isDigit(ch))
  .forEach(ch -> printChar(ch));
```

We can see the filtered digits in the next output.

```
0
0
```

Once again, instead of the lambda expressions we passed to the filter() method and the forEach() method, we can replace them with method references to the respective methods.

compare/fpij/IterateString.java

```
str.chars().filter(Character::isDigit).forEach(IterateString::printChar);
```

The method references helped here again to remove the mundane parameter routing. In addition to that, in this example we see yet another variation of method references compared to the previous two instances where we used them. When we first saw method references, we created a method reference for an instance method. Later we created it for a call on a static reference. Now we're creating a method reference for a static method—method references seem to keep on giving.

The method reference for an instance method and a static method structurally look the same: for example, String::toUppercase and Character::isDigit. To decide how to route the parameter, the Java compiler will look up to see if the method is an instance method or a static method. If it's an instance method, then the parameter of the synthesized method becomes the target of the call, like in parameter.toUppercase(); (the exception to this rule is if the target is already specified like in System.out::println). On the other hand, if the method is a static method, then the parameter to the synthesized method is routed as an argument to this method, like in Character.isDigit(parameter);. See Appendix 2, *Syntax Overview*, on page ? for a listing of method references variations and their syntax.

While this parameter routing is quite convenient, there is one caveat—method collisions and the resulting ambiguity. If there's both a matching instance method and a static method, we will get a compilation error due to the ambiguity of the reference. For example, if we write Double::toString to convert an instance of Double to a String, the compiler would get confused whether to use the public String toString() instance method or the static method public static String toString(double value), both from the Double class. If we run into this, no sweat, simply switch back to using the appropriate lambda expression version to move on.

Once we get used to the functional style, we can gradually switch between either the lambda expressions or the more concise method references, based on our comfort level.

We used a new method in Java 8 to easily iterate over characters. Next we'll explore the enhancements to the Comparator interface.