Extracted from:

Functional Programming in Java

Harnessing the Power of Java 8 Lambda Expressions

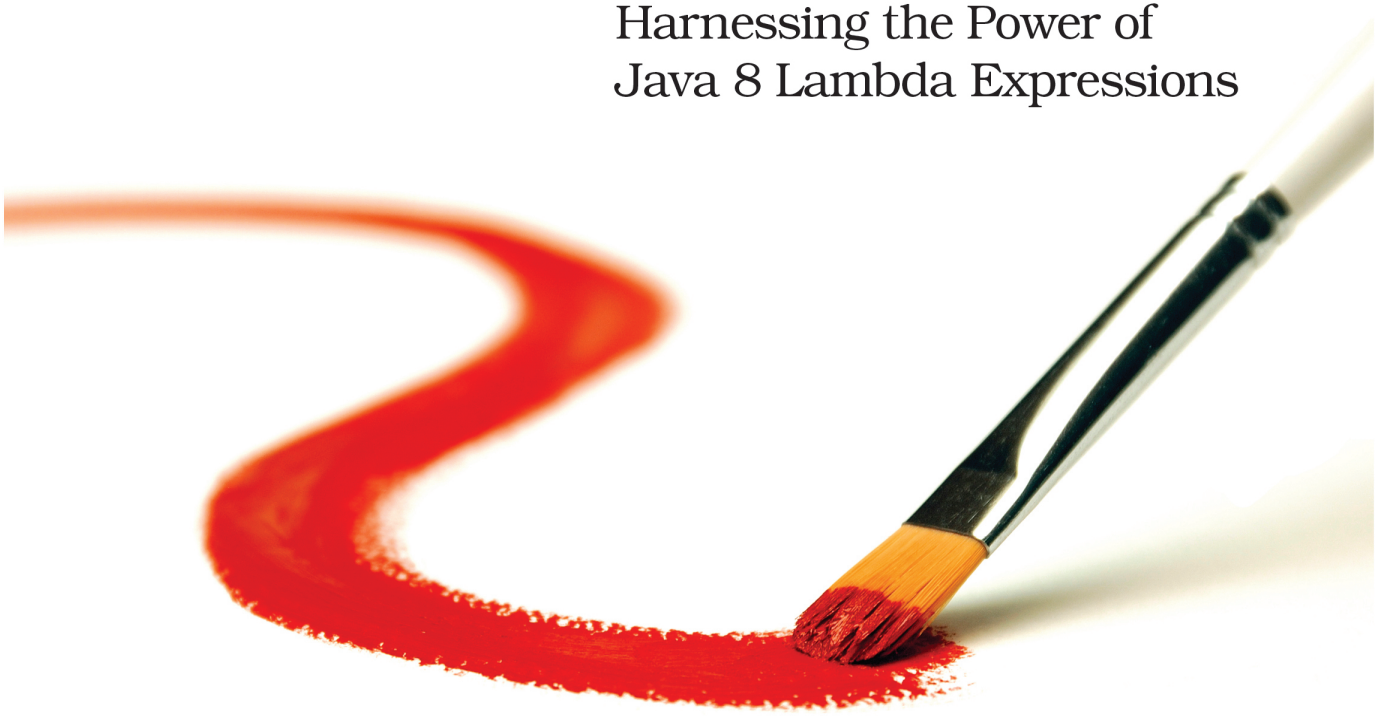
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Functional Programming in Java

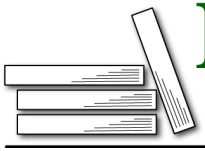
Harnessing the Power of Java 8 Lambda Expressions



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.

Make everything as simple as possible, but not simpler.

➤ *Albert Einstein*

CHAPTER 3

Strings, Comparators, and Filters

The JDK has evolved to include convenience methods that promote the functional style. When using familiar classes and interfaces from the library —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 tasks at hand. Techniques you pick up along the way will help turn long, mundane tasks into concise code snippets you can quickly write and easily maintain.

Iterating a String

The chars() method is a new one in the String class from the CharSequence interface. It's useful for fluently iterating 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 use it in an example to process a string. Along the way we'll discuss a few more handy 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 letters we're seeing numbers. That's because the chars() method returns a stream of Integers representing the letters 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 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 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 that 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 *A Little Sugar to Sweeten*, 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 on a class name (String). In this example, the method reference, again to an instance method, is based on 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 the method reference is quite concise, but we have to dig into it a bit more to understand what's going on. Once we get used to method references, our brains will know to autoparse these.

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

compare/fpij/IterateString.java
```
private static void printChar(int aChar) {
  System.out.println((char)(aChar));
}
```

We can use a 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 letters.

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

We used the internal iterator on the Stream that the chars() method returned, 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(), and so on, 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 use references to the respective methods.

compare/fpij/IterateString.java
```
str.chars()
   .filter(Character::isDigit)
   .forEach(IterateString::printChar);
```

The method references here helped remove the mundane parameter routing. In addition, 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 one for an instance method. Later we created one 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 one for an instance method and the one for a static method look the same structurally: for example, String::toUppercase and Character::isDigit. To decide how to route the parameter, the Java compiler will check whether the method is an instance method or a static method. If it's an instance method, then the synthesized method's parameter becomes the call's target, 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 static, 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-reference 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'll get a compilation error due to the reference's ambiguity. 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; we simply switch back to using the appropriate lambda-expression version to move on.

Once we get used to the functional style, we can switch between the lambda expressions and 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.