

10 Example of Lambda Expressions and Streams in Java 8

Java 8 release is just a couple of weeks away, scheduled at 18th March 2014, and there is lot of buzz and excitement about this path breaking release in Java community. One of feature, which is synonymous to this release is lambda expressions, which will provide ability to pass behaviours to methods. Prior to Java 8, if you want to pass behaviour to a method, then your only option was Anonymous class, which will take 6 lines of code and most important line, which defines the behaviour is lost in between. *Lambda expression replaces anonymous classes* and removes all boiler plate, enabling you to write code in functional style, which is some time more readable and expression.



Teach 3 hours a day
Earn upto Rs. 40,000/month

This mix of bit of functional and full of object oriented capability is very exciting development in Java eco-system, which will further enable development and growth of parallel third party libraries to take advantage of multi-processor CPUs.

Though industry will take its time to adopt Java 8, I don't think any serious Java developer can overlook key features of Java 8 release e.g. lambda expressions, functional interface, stream API, default methods and new Date and Time API.

As a developer, I have found that best way to learn and master lambda expression is to try it out, do as many examples of lambda expressions as possible. Since biggest impact of Java 8 release will be on Java Collections framework its best to try examples of Stream API and lambda expression to extract, filter and sort data from Lists and Collections.

I have been writing about Java 8 and have shared some [useful resources to master Java 8](#) in past. In this post, I am going to share you 10 most useful ways to use lambda expressions in your code, these examples are simple, short and clear, which will help you to pick lambda expressions quickly.



Teach 3 hours a day
Earn upto Rs. 40,000/month

Java 8 Lambda Expressions Examples

I am personally very excited about Java 8, particularly lambda expression and stream API. More and more I look them, it makes me enable to write more clean code in Java. Though it was not like this always; when I first saw a Java code written using lambda expression, I was



Interview Questions

[core java interview question \(161\)](#)

[data structure and algorithm \(45\)](#)

[Coding Interview Question \(32\)](#)

[SQL Interview Questions \(24\)](#)

[thread interview questions \(20\)](#)

[database interview questions \(18\)](#)

[servlet interview questions \(17\)](#)

[collections interview questions \(15\)](#)

[spring interview questions \(9\)](#)

[Programming interview question \(4\)](#)

[hibernate interview questions \(4\)](#)

[Translate this blog](#)

very disappointed with cryptic syntax and thinking they are making Java unreadable now, but I was wrong.

After spending just a day and doing *couple of examples of lambda expression and stream API*, I was happy to see more cleaner Java code then before. It's like the [Generics](#), when I first saw I hated it. I even continued using old Java 1.4 way of dealing with Collection for few month, until one of my friend explained me benefits of using Generics.

Bottom line is, don't afraid with initial cryptic impression of lambda expressions and method reference, you will love it once you do couple of examples of extracting and filtering data from Collection classes. So let's start this wonderful journey of learning lambda expressions in Java 8 by simple examples.

Example 1 - implementing Runnable using Lambda expression

One of the first thing, I did with Java 8 was trying to replace anonymous class with lambda expressions, and what could have been best example of anonymous class then implementing Runnable interface. Look at the code of implementing runnable prior to Java 8, it's taking four lines, but with lambda expressions, it's just taking one line. What we did here? the whole [anonymous class](#) is replaced by `() -> {}` code block.

```
//Before Java 8:
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Before Java8, too much code for too little to do");
    }
}).start();

//Java 8 way:
new Thread( () -> System.out.println("In Java8, Lambda expression rocks !!") ).start();
```

Output:
too much code, for too little to do
Lambda expression rocks !!

This example brings us syntax of lambda expression in Java 8. You can write following kind of code using lambdas :

```
(params) -> expression
(params) -> statement
(params) -> { statements }
```

for example, if your method don't change/write a parameter and just print something on console, you can write it like this :

```
() -> System.out.println("Hello Lambda Expressions");
```

If your method accept two parameters then you can write them like below :

```
(int even, int odd) -> even + odd
```

By the way, it's general practice to **keep variable name short inside lambda expressions**. This makes your code shorter, allowing it to fit in one line. So in above code, choice of variable names as `a,b` or `x,y` is better than `even` and `odd`.

Example 2 - Event handling using Java 8 Lambda expressions

If you have ever done coding in Swing API, you will never forget writing event listener code. This is another classic use case of plain old Anonymous class, but no more. You can write better event listener code using lambda expressions as shown below.

```
// Before Java 8:
```

Java Tutorials

[date and time tutorial \(18\)](#)

[FIX protocol tutorial \(16\)](#)

[java collection tutorial \(53\)](#)

[java IO tutorial \(25\)](#)

[Java JSON tutorial \(6\)](#)

[Java multithreading Tutorials \(33\)](#)

[Java Programming Tutorials \(27\)](#)

[Java xml tutorial \(9\)](#)

Search This Blog

```

JButton show = new JButton("Show");
show.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Event handling without lambda expression is boring");
    }
});

```

```

// Java 8 way:
show.addActionListener((e) -> {
    System.out.println("Light, Camera, Action !! Lambda expressions Rocks");
});

```

Another place where Java developers frequently use anonymous class is for providing [custom Comparator](#) to `Collections.sort()` method. In Java 8, you can replace your ugly anonymous class with more readable lambda expression. I leave that to you for exercise, should be easy if you follow the pattern, I have shown during implementing [Runnable](#) and `ActionListener` using lambda expression.

Example 3 - Iterating over List using Lambda expressions

If you are doing Java for few years, you know that most common operation with Collection classes are iterating over them and applying business logic on each elements, for example processing a list of orders, trades and events. Since Java is an imperative language, all code looping code written prior to Java 8 was sequential i.e. their is on simple way to do parallel processing of list items. If you want to do parallel filtering, you need to write your own code, which is not as easy as it looks. Introduction of lambda expression and default methods has separated what to do from how to do, which means now Java Collection knows how to iterate, and they can now provide parallel processing of Collection elements at API level. In below example, I have shown you [how to iterate over List](#) using with and without lambda expressions, you can see that now List has a `forEach()` method, which can iterate through all objects and can apply whatever you ask using lambda code.

```

//Prior Java 8 :
List features = Arrays.asList("Lambdas", "Default Method", "Stream API", "Date and Time API");
for (String feature : features) {
    System.out.println(feature);
}

```

```

//In Java 8:
List features = Arrays.asList("Lambdas", "Default Method", "Stream API", "Date and Time API");
features.forEach(n -> System.out.println(n));

```

```

// Even better use Method reference feature of Java 8
// method reference is denoted by :: (double colon) operator
// looks similar to scope resolution operator of C++
features.forEach(System.out::println);

```

Output:
Lambdas
Default Method
Stream API
Date and Time API

The last example of [looping over List](#) shows *how to use method reference in Java 8*. You see the double colon, scope resolution operator form C++, it is now used for method reference in Java 8.

Example 4 - Using Lambda expression and Functional interface Predicate

Apart from providing support for functional programming idioms at language level, Java 8 has also added a new package called `java.util.function`, which contains lot of classes to enable functional programming in Java. One of them is `Predicate`, By using `java.util.function.Predicate` functional interface and lambda expressions, you can provide logic to API methods to add lot of dynamic behaviour in less code. Following *examples of Predicate in Java 8* shows lot of common ways to [filter Collection data in Java code](#). `Predicate` interface is great for filtering.

```

public static void main(args[]){
    List languages = Arrays.asList("Java", "Scala", "C++", "Haskell", "Lisp");
}

```

Follow by Email

Submit

Followers

Blog Archive

► 2017 (26)

► 2016 (166)

► 2015 (126)

▼ 2014 (101)

► December (6)

► November (8)

► October (4)

► September (9)

► August (8)

► July (6)

► June (8)

► May (11)

► April (10)

► March (11)

▼ February (11)

Why Catching Throwable or Error is bad?

10 Example of Lambda Expressions and Streams in Ja...

How to Format and Display Number to Currency in Ja...

Fixing java.net.BindException: Cannot assign requ...

Top 30 Java Phone Interview Questions Answers for ...

How to Create Tabs UI using HTML, CSS, jQuery, JSP...

Why Static Code Analysis is Important?

Display tag Pagination, Sorting Example in JSP and...

Java Comparable Example for Natural Order Sorting

Difference between Association, Composition and Ag...

StringTokenizer Example in Java with Multiple Deli...

► January (9)

► 2013 (127)

► 2012 (214)

► 2011 (135)

► 2010 (30)

Pages

[Privacy Policy](#)

Copyright by Javin Paul 2010-2016. Powered by

```

System.out.println("Languages which starts with J :");
filter(languages, (str)->str.startsWith("J"));

System.out.println("Languages which ends with a ");
filter(languages, (str)->str.endsWith("a"));

System.out.println("Print all languages :");
filter(languages, (str)->true);

System.out.println("Print no language : ");
filter(languages, (str)->false);

System.out.println("Print language whose length greater than 4:");
filter(languages, (str)->str.length() > 4);
}

public static void filter(List names, Predicate condition) {
    for(String name: names) {
        if(condition.test(name)) {
            System.out.println(name + " ");
        }
    }
}
}

```

Output:

```

Languages which starts with J :
Java
Languages which ends with a
Java
Scala
Print all languages :
Java
Scala
C++
Haskell
Lisp
Print no language :
Print language whose length greater than 4:
Scala
Haskell

```

```

//Even better
public static void filter(List names, Predicate condition) {
    names.stream().filter((name) -> (condition.test(name))).forEach((name) -> {
        System.out.println(name + " ");
    });
}

```

You can see that filter method from Stream API also accept a Predicate, which means you can actually replace our custom filter() method with the in-line code written inside it, that's the power of lambda expression. By the way, Predicate interface also allows you test for multiple condition, which we will see in our next example.

Example 5 : How to combine Predicate in Lambda Expressions

As I said in previous example, java.util.function.Predicate allows you to combine two or more Predicate into one. It provides methods similar to logical operator AND and OR named as and(), or() and xor(), which can be used to combine the condition you are passing to filter() method. For example, In order to get all languages, which starts with J and are four character long, you can define two separate Predicate instance covering each condition and then combine them using Predicate.and() method, as shown in below example :

```

// We can even combine Predicate using and(), or() And xor() logical functions
// for example to find names, which starts with J and four letters long, you
// can pass combination of two Predicate
Predicate<String> startsWithJ = (n) -> n.startsWith("J");
Predicate<String> fourLetterLong = (n) -> n.length() == 4;

names.stream()
    .filter(startsWithJ.and(fourLetterLong))
    .forEach((n) -> System.out.print("\nName, which starts with 'J' and four letter long is : "

```

Similarly you can also use or() and xor() method. This example also highlight important fact about using Predicate as individual condition and then combining them as per your need.

In short, you can use Predicate interface as traditional Java imperative way, or you can take advantage of lambda expressions to write less and do more.

Example 6 : Map and Reduce example in Java 8 using lambda expressions

This example is about one of the popular functional programming concept called map. It allows you to transform your object. Like in this example we are transforming each element of costBeforeTax list to including Value added Tax. We passed a lambda expression $x \rightarrow x * x$ to `map()` method which applies this to all elements of the stream. After that we use `forEach()` to print the all elements of list. You can actually get a List of all cost with tax by using Stream API's Collectors class. It has methods like `toList()` which will combine result of map or any other operation. Since Collector perform terminal operator on Stream, you can't re-use Stream after that. You can even use `reduce()` method from Stream API to reduce all numbers into one, which we will see in next example

```
// applying 12% VAT on each purchase
// Without lambda expressions:
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
for (Integer cost : costBeforeTax) {
    double price = cost + .12*cost;
    System.out.println(price);
}

// With Lambda expression:
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
costBeforeTax.stream().map((cost) -> cost + .12*cost).forEach(System.out::println);
```

Output

```
112.0
224.0
336.0
448.0
560.0
112.0
224.0
336.0
448.0
560.0
```

Example 6.2 - Map Reduce example using Lambda Expressions in Java 8

In previous example, we have seen how map can transform each element of a Collection class e.g. List. There is another function called `reduce()` which can combine all values into one. Map and Reduce operations are core of functional programming, reduce is also known as fold operation because of its folding nature. By the way reduce is not a new operation, you might have been already using it. If you can recall SQL aggregate functions like `sum()`, `avg()` or `count()`, they are actually reduce operation because they accept multiple values and return a single value. Stream API defines `reduce()` function which can accept a lambda expression, and combine all values. Stream classes like `IntStream` has built-in methods like `average()`, `count()`, `sum()` to perform reduce operations and `mapToLong()`, `mapToDouble()` methods for transformations. It doesn't limit you, you can either use built-in reduce function or can define yours. In this Java 8 Map Reduce example, we are first applying 12% VAT on all prices and then calculating total of that by using `reduce()` method.

```
// Applying 12% VAT on each purchase
// Old way:
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
double total = 0;
for (Integer cost : costBeforeTax) {
    double price = cost + .12*cost;
    total = total + price;
}
System.out.println("Total : " + total);

// New way:
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
double bill = costBeforeTax.stream().map((cost) -> cost + .12*cost).reduce((sum, cost) -> sum + cost, 0);
System.out.println("Total : " + bill);
```

```
Output
Total : 1680.0
Total : 1680.0
```

Example 7: Creating a List of String by filtering

Filtering is one of the common operation Java developers perform with large collections, and you will be surprise how much easy it is now to filter bulk data/large collection using lambda expression and stream API. Stream provides a `filter()` method, which accepts a Predicate object, means you can pass lambda expression to this method as filtering logic. Following examples of filtering collection in Java with lambda expression will make it easy to understand.

```
// Create a List with String more than 2 characters
List<String> filtered = strList.stream().filter(x -> x.length() > 2).collect(Collectors.toList());
System.out.printf("Original List : %s, filtered list : %s %n", strList, filtered);
```

```
Output :
Original List : [abc, , bcd, , defg, jk], filtered list : [abc, bcd, defg]
```

By the way, their is a common confusion regarding `filter()` method. In real world, when we filter, we left with something which is not filtered, but in case of using `filter()` method, we get a new list which is actually filtered by satisfying filtering criterion.

Example 8: Applying function on Each element of List

We often need to apply certain function to each element of List e.g. multiplying each element by certain number or dividing it, or doing anything with that. Those operations are perfectly suited for `map()` method, you can supply your transformation logic to `map()` method as lambda expression and it will transform each element of that collection, as shown in below example.

```
// Convert String to Uppercase and join them using coma
List<String> G7 = Arrays.asList("USA", "Japan", "France", "Germany", "Italy", "U.K.", "Canada");
String G7Countries = G7.stream().map(x -> x.toUpperCase()).collect(Collectors.joining(", "));
System.out.println(G7Countries);
```

```
Output :
USA, JAPAN, FRANCE, GERMANY, ITALY, U.K., CANADA
```

Example 9: Creating a Sub List by Copying distinct values

This example shows how you can take advantage of `distinct()` method of Stream class to filter duplicates in Collection.

```
// Create List of square of all distinct numbers
List<Integer> numbers = Arrays.asList(9, 10, 3, 4, 7, 3, 4);
List<Integer> distinct = numbers.stream().map(i -> i*i).distinct().collect(Collectors.toList());
System.out.printf("Original List : %s, Square Without duplicates : %s %n", numbers, distinct);
```

```
Output :
Original List : [9, 10, 3, 4, 7, 3, 4], Square Without duplicates : [81, 100, 9, 16, 49]
```

Example 10 : Calculating Maximum, Minimum, Sum and Average of List elements

There is a very useful method called `summaryStatistics()` in stream classes like `IntStream`, `LongStream` and `DoubleStream`. Which returns returns an `IntSummaryStatistics`, `LongSummaryStatistics` or `DoubleSummaryStatistics` describing various summary data about the elements of this stream. In following example, we have used this method to calculate maximum and minimum number in a List. It also has `getSum()` and `getAverage()` which can give sum and average of all numbers from List.

```
//Get count, min, max, sum, and average for numbers
List<Integer> primes = Arrays.asList(2, 3, 5, 7, 11, 13, 17, 19, 23, 29);
IntSummaryStatistics stats = primes.stream().mapToInt((x) -> x).summaryStatistics();
System.out.println("Highest prime number in List : " + stats.getMax());
System.out.println("Lowest prime number in List : " + stats.getMin());
System.out.println("Sum of all prime numbers : " + stats.getSum());
System.out.println("Average of all prime numbers : " + stats.getAverage());
```

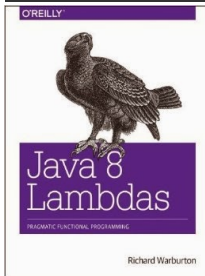
Output :

```
Highest prime number in List : 29
Lowest prime number in List : 2
Sum of all prime numbers : 129
Average of all prime numbers : 12.9
```

Lambda Expression vs Anonymous class

Since lambda expression is effectively going to replace Anonymous inner class in new Java code, its important to do a comparative analysis of both of them. One key difference between using Anonymous class and Lambda expression is the use of [this keyword](#). For anonymous class 'this' keyword resolves to *anonymous class*, whereas for lambda expression 'this' keyword resolves to *enclosing class* where lambda is written. Another difference between lambda expression and anonymous class is in the way these two are compiled. Java compiler compiles lambda expressions and convert them into [private method](#) of the class. It uses `invokedynamic` byte code instruction from Java 7 to bind this method dynamically.

Things to remember about Lambdas in Java 8



So far we have see 10 examples of lambda expression in Java 8, this is actually a good dose of lambdas for beginners, you will probably need to run examples by your own to get most of it. Try changing requirement, and create your own examples to learn quickly. I would also like to suggest using Netbeans IDE for practising lambda expression, it has got really good Java 8 support. Netbeans shows hint for converting your code into functional style as and when it sees opportunity. It's extremely easy to *convert an Anonymous class to lambda expression*, by simply following Netbeans hints. By the way, If you love to read books then don't forget to check Java 8 Lambdas,

pragmatic functional programming by Richard Warburton, or you can also see Manning's Java 8 in Action, which is not yet published but I guess have free PDF of first chapter available online. But before you busy with other things, let's revise some of the important things about Lambda expressions, default methods and functional interfaces of Java 8.

1) Only predefined Functional interface using `@Functional` annotation or method with one abstract method or SAM (Single Abstract Method) type can be used inside lambda expressions. These are actually known as target type of lambda expression and can be used as return type, or parameter of lambda targeted code. For example, if a method accepts a `Runnable`, [Comparable](#) or `Callable` interface, all has single abstract method, you can pass lambda expression to them. Similarly if a method accept interface declared in `java.util.function` package e.g. `Predicate`, `Function`, `Consumer` or `Supplier`, you can pass lambda expression to them.

2) You can use method reference inside lambda expression if method is not modifying the parameter supplied by lambda expression. For example following lambda expression can be replaced with a method reference since it is just a single method call with the same parameter:

```
list.forEach(n -> System.out.println(n));
```

```
list.forEach(System.out::println); // using method reference
```

However, if there's any transformations going on with the argument, we can't use method references and have to type the full lambda expression out, as shown in below code :


```
list.forEach((String s) -> System.out.println("*" + s + "*"));
```

You can actually omit the type declaration of lambda parameter here, compiler is capable to infer it from generic type of List.

3) You can use both [static](#), non static and [local variable](#) inside lambda, this is called **capturing variables inside lambda**.

4) Lambda expressions are also known as **closure** or **anonymous function** in Java, so don't be surprise if your colleague calling closure to lambda expression.

5) Lambda methods are internally translated into private methods and `invokedynamic` byte code instruction is now issued to dispatch the call:. You can use [javap tool](#) to decompile class files, it comes with JDK. Use command `javap -p` or `javap -c -v` to take a look at byte bode generated by lambda expressions. It would be something like this

```
private static java.lang.Object lambda$0(java.lang.String);
```

6) One restriction with lambda expression is that, you can only reference either final or effectively final local variables, which means you cannot modified a variable declared in the outer scope inside a lambda.

```
List<Integer> primes = Arrays.asList(new Integer[]{2, 3,5,7});
int factor = 2;
primes.forEach(element -> { factor++; });
```

Compile time error : *"local variables referenced from a lambda expression must be final or effectively final"*

By the way, simply accessing them, without modifying is Ok, as shown below :

```
List<Integer> primes = Arrays.asList(new Integer[]{2, 3,5,7});
int factor = 2;
primes.forEach(element -> { System.out.println(factor*element); });
```

Output

```
4
6
10
14
```

So its more like a closure with immutable capture, similar to Python.

That's all in this **10 examples of lambda expressions in Java 8**. This is going to be one of the biggest change in Java's history and will have a huge impact on how Java developer use Collections framework going forward. Anything I can think of similar scale was Java 5 release, which brings lots of goodies to improve code quality in Java, e.g. Generics, Enum, Autoboxing, Static imports, Concurrent API and variable arguments. Just like above all feature helped to write clean code in Java, I am sure lambda expression will take it to next level. One of the thing I am expecting is development of parallel third-party libraries, which will make writing high performance application slightly easier than today.

Posted by Javin Paul 

Labels: [core java](#), [Java 8](#), [Lambda expression](#), [programming](#)

12 comments :

[Sushil nagpal](#) said...

I loved it.. Awesome Post buddy. Thanks :)

February 26, 2014 at 10:33 PM

Anonymous said...

Thanks for this wonderful article, Most important thing, which I know I come to know is that if a method accept type of an interface, which has just one abstract method e.g. Comparator or Comparable, we can use a lambda expression instead of defining and then creating a new instance of a class that implements Comparator, and if that lambda expression does nothing but call an existing method, we can use method reference. This really helped me to understand where can we use lambda expression, and answered my long had doubt about why everyone talk about Comparable and Runnable when they talk Java 8 and lambdas.

February 26, 2014 at 11:19 PM

[ella aslev](#) said...

Thanks for sharing this information. Java program examples are quite useful for me. In General [Java Lambda Expressions and Streams](#) are one of the great feature added in Java 8. Lambda Expressions are used to represent functions and streams are nothing but wrappers, arrays and collections.

February 27, 2014 at 10:50 PM

[Vasi](#) said...

Oh no, this will make debugging a nightmare. Java was awesome because of it's simplicity; to bad it's loosing that...

March 3, 2014 at 1:16 AM

Anonymous said...

I have a question.

Suppose that an interface has 2 methods with the same signature but different names:

```
public void actionPerformed(ActionEvent e);
```

```
public void actionWhatever(ActionEvent e);
```

If you use:

```
show.addActionListener((e) -> { System.out.println("Light, Camera, Action !! Lambda expressions Rocks"); });
```

How will it know which one of the two methods to replace with it?

March 15, 2014 at 2:03 PM

Anonymous said...

Nice article. clearly explained about lambda expressions. Thanks for posting. Great Job.

March 17, 2014 at 12:03 PM

Anonymous said...

If an interface has more then one method, then that will not be a functional interface, and therefore will not be supportes by lambda expressions...

March 19, 2014 at 1:57 PM

[Amit Jain](#) said...

wonderful, perfect way to understand and learn. Thanks

April 14, 2014 at 7:41 AM

Anonymous said...

Better way to filter non null values in a List in Java 8 :

```
stockList.stream().filter(Objects::nonNull).forEach(o -> {...});
```

better use of static method reference instead of lambdas and functional interface.

April 14, 2014 at 8:33 AM

Anonymous said...

hello sir, can you please share how to use FlatMap in Java 8? I am looking for a decent example but not able to find a good one. FlatMap seems to be very useful, please provide a decedn FlatMap example using Java 8. thanks

January 22, 2015 at 11:05 PM

Anonymous said...

Hi,
in example4, how do the codes get compiled?

eg: filter(languages, (str)->str.startsWith("J"));

variable (str) is not a String, so str.StartsWith("J") will get compiled error.
unless it get casted into String as ((String)str).startsWith("J");

thanks.

April 13, 2015 at 3:57 PM

[Razvan Manolescu](#) said...

// With Lambda expression:

```
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
```

This list should be parametrized, as the following lambda expression won't be able figure out how to the arithmetic operation on the Object.

November 22, 2015 at 9:09 AM

Post a Comment

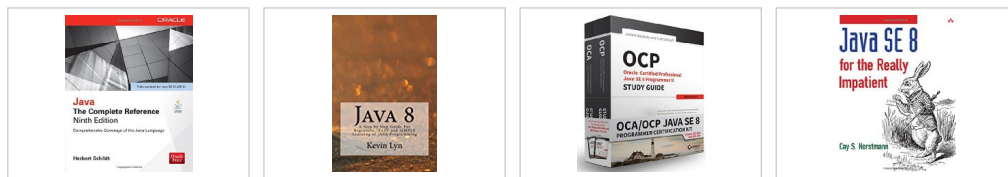
[Newer Post](#)




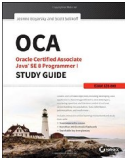
[Home](#)

[Older Post](#)

Subscribe to: [Post Comments \(Atom \)](#)

Related Books for Further Reading



<p>Java: The Complete Reference, Ninth Edition</p> <p>\$37.02 \$60.00</p> <p>(271)</p>	<p>Java 8: A Step by Step Guide For Beginners FAST and S...</p> <p>\$9.49</p> <p>(6)</p>	<p>OCA / OCP Java SE 8 Programmer Certification...</p> <p>\$60.48 \$90.00</p> <p>(11)</p>	<p>Java SE8 for the Really Impatient: A Short Cour...</p> <p>\$31.99 \$39.00</p> <p>(37)</p>
 <p>Java 8 in Action: Lambdas, Streams, and functional-st...</p> <p>\$37.71 \$49.00</p> <p>(76)</p>	 <p>JAVA 8 Programming: Step by Step Java 8 Course Progra...</p> <p>\$9.45</p> <p>(15)</p>	 <p>Java 8 Lambdas: Functional Programming For The M...</p> <p>\$31.84 \$39.00</p> <p>(29)</p>	 <p>OCA: Oracle Certified Associate Java SE 8 Pro...</p> <p>\$38.68 \$59.00</p> <p>(104)</p>

All ▼

Search Amazon

Go

Ads by Amazon