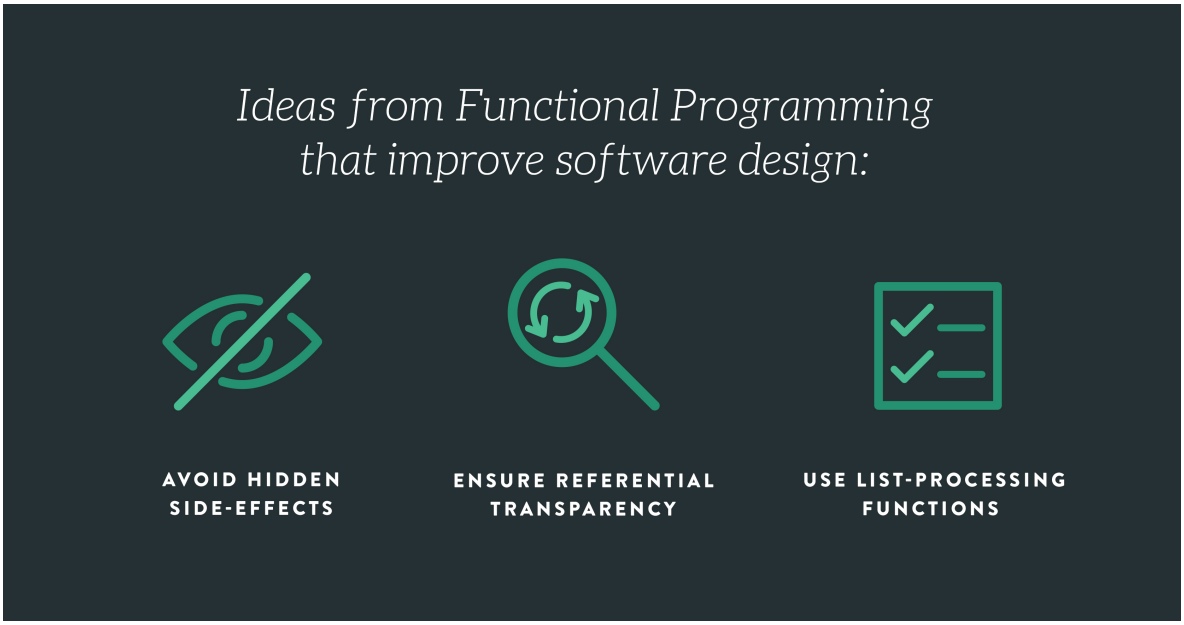
DZone

# When Functional Programming Isn't

**by Dave Nicolette · Mar. 07, 18 · Java Zone · Tutorial**

Take 60 minutes to understand the Power of the Actor Model with "Designing Reactive Systems: The Role Of Actors In Distributed Architecture". Brought to you in partnership with Lightbend.




When functional programming started to become a “thing” in the software industry, I had a lot of difficulty understanding the fundamentals. I’ve always worked primarily in business application software and in system administration. The mathematical orientation of functional programming languages represented an unfamiliar paradigm for me.

I mentioned this to a colleague, and he told me that he fully grokked FP after learning to use lambda expressions in C#. His reply puzzled me, as I had been using lambda expressions in Java, Ruby, and JavaScript for some time, and I knew that similar features were present in other languages, like Python, SmallTalk, and Lisp. I had also done the usual tutorials for languages that had some FP features, like Scala and Clojure. Could it be that FP was really nothing more than this, decorated with arcane buzzwords?

Surely there was more to it. My attempts to teach myself Haskell had not gone as easily as learning other languages that were based on familiar paradigms. Haskell code was clearly quite different from lambda expressions in Java or Ruby. It seemed, instead, that support for lambda expressions in object-oriented languages had been *inspired* by functional languages, but that there was much more to FP than just that.

## Mathematical Programming

## Mathematical Programming

As I read more about FP, it became clear that the purpose of functional languages is to model mathematical expressions. It might have been better if the inventors of FP had named it "mathematical programming" instead. It isn't just about writing "functions," after all.

APL, a language developed in the 1960s, uses non-ASCII symbols in its source code to represent mathematical concepts. A special keyboard mapping was designed to support working in APL. It looks like this:


Here's an example of Conway's Game of Life implemented in APL, from StackOverflow:

```
life←{↑1 ⍵∨.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂⍵}
```

Other functional languages, such as J, substitute ASCII symbols for mathematical symbols so that mathematical expressions can be entered using a standard keyboard. Here's an example of Conway's Game of Life implemented in J:

```
1   step =: ((]+.&(3&=)+)(+/@(((4&{.,(_4&{.))(>,{,~>i:1))&|.)))~
```

Even if you aren't familiar with those languages, you can probably see the two code snippets are roughly similar. Personally, I find the J code harder to read, but I appreciate the practicality of being able to enter source code with a standard keyboard.

One of the key uses of mathematical expressions is to process *lists* or *series*, like this formula for calculating the area under a curve (image borrowed from here):

$$\int_a^b f(x)\,dx = \lim_{mesh \to 0} \sum_{i=1}^{n} f(c_i)\Delta x_i$$

So, there's an integral and a summation and so forth. Functional programming languages are designed to solve this sort of problem efficiently based on concise source code. The language may have a number of these common calculations built in or available through libraries.

Lambda expressions in object-oriented languages serve the same purpose. The examples you find online

Lambda expressions in object-oriented languages serve the same purpose. The examples we find online can be confusing for a business application programmer because we don't often face mathematical problems. Programmers who write data analytics code and those who work on scientific/engineering solutions will find the usual examples more relatable.

In business application programming we *do* quite often process collections of objects, either filtering a subset of members, combining two collections, or performing an operation on all members (like *map* and *reduce* operations). Lambda expressions often enable us to express these operations in a concise and readable way, as compared with (for instance) nested *for* loops. The code also tends to be less error-prone than a complicated loop structure.

For example, here's an iterative solution to the Fibonacci series in Java (snagged from a StackOverflow question):

```
public int fib(int n) {
    int x = 0, y = 1, z = 1;
    for (int i = 0; i < n; i++) {
        x = y;
        y = z;
        z = x + y;
    }
    return x;
}
```

For contrast, here's a solution using lambda expressions (thanks to Artem Lovan):

```
public static List generate(int series) {
    return Stream.iterate(new int[]{0, 1}, s -> new int[]{s[1], s[0] + s[1]})
                .limit(series)
                .map(n -> n[0])
                .collect(toList());
}
```

(Yes, I was lazy and didn't write my own examples. How different could they have been, anyway?)

You can see for yourself that the lambda version is more concise and expresses its intent effectively.

# Practical Software Development Takeaways

Is it true that when you've mastered lambda expressions in an OO language, you've also mastered functional programming? I don't think so. But I *do* think it's a good idea to master lambda expressions.

General business application programming has taken several ideas from FP that improve our software design. These include (at least):

- avoid hidden side-effects
- ensure referential transparency
- use list-processing functions rather than loops where it makes sense

Those guidelines are not really different from conventional OO design guidelines. Avoiding hidden side-effects is another way to express the idea of *separation of concerns* or *single responsibility principle*. When you do that, you also achieve *referential transparency*. Lambda expressions that operate on collections make the code more expressive of intent and more concise. It has always been a goal of OO design to make the code expressive of intent.

# Microtests

Contemporary development practices include the idea of *microtests*, which are very small examples of functionality that exercise (in an OO language) a single logical path through a single method. They are often used to support emergent design through test-driven development as well as to explore the functionality of an existing code base by probing it with small test cases.

When I use lambda expressions in an OO language, I find that the smallest logical chunk of code that can be exercised by a micro-example is larger than when I use iteration or recursion for the same solution. Let's look at the two Java examples above, and write microtests for them.

If we take that iterative example and flesh it out a bit so that we can obtain a Fibonacci series, we get this:

```
1  public class FibIterative {
2      public int fib(int n) {
3          int x = 0, y = 1, z = 1;
4          for (int i = 0; i < n; i++) {
5              x = y;
6              y = z;
7              z = x + y;
8          }
9          return x;
10     }
11
12     public List<Integer> fibSeries(int limit) {
13         List<Integer> result = new ArrayList<Integer>();
14         for (int i = 0 ; i < limit ; i++ ) {
15             result.add(fib(i));
16         }
17         return result;
18     }
19 }
```

Some microtests for that code could look like this:

```
@Test
public void the_2nd_value_is_1() {
    assertEquals(1, fib.fib(1));
}

@Test
public void the_5th_value_is_3() {
    assertEquals(3, fib.fib(4));
}

@Test
public void first_10_values() {
    List expected = Arrays.asList(new Integer[] { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 });
    assertEquals(expected, fib.fibSeries(10));
}
```

The key point here is that the smallest chunk of logic we can check is the code that determines the next number in the series. The third example shows that we *could* verify the functionality by checking the entire list. However, if there is an error in the single-number routine the whole-list check won't take us directly to the offending line in the source. (You may have to imagine something more complicated than this to visualize the problem.) There are special rules for the first few numbers in the Fibonacci series, and that's where an error is most likely.

An advantage of using a lambda expression is that the correctness of the series can be assured by the lambda expression itself. The smallest microtest example we need is one that covers the whole series.

Adjusting the sample lambda solution so it will compile, and making the method non-static, we have this:

```
public class FibLambda {
    public List generate(int series) {
        return Stream.iterate(new int[]{0, 1}, s -> new int[]{s[1], s[0] + s[1]})
                    .limit(series)
                    .map(n -> n[0])
                    .collect(Collectors.toList());
    }
}
```

You can already see a couple of advantages, even before we consider the microtests. This solution involves less code, and therefore less probability of an error occurring. The code is also a little more self-describing.

We don't need microtest examples to verify that individual numbers in the series will be calculated

We don't need microtest examples to verify that individual numbers in the series will be calculated correctly, as that is baked into the lambda expression. We only need to verify that the series comes out as expected. So, we have fewer test cases to maintain, without getting into the ice cream cone anti-pattern of test automation.

```
@Test
public void it_generates_the_first_10_values() {
    List expected = Arrays.asList(new Integer[] { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 });
    FibLambda fib = new FibLambda();
    assertEquals(expected, fib.generate(10));
}
```

# Programmer Effectiveness

It takes a programmer about the same amount of time and effort to read/understand, design/create, comprehend/modify a line of source code regardless of the programming language involved. When a language offers more power per source expression, solutions can be built using fewer lines of code. Programmers' time is used more effectively than when they must write in a language that doesn't offer much power per expression.

Functional languages pack a lot of power into each line of source code. When we use elements borrowed from FP in our OO languages, we can gain some of those benefits, too.

To illustrate, here are some code snippets that sum the values in an array. First, let's look at Intel assembly language. This example is borrowed from here.

```
; find the sum of the elements of an array
SECTION .data ; start of data segment
global x
x:
      dd      1
      dd      5
      dd      2
      dd      18
sum:
      dd   0
SECTION .text ; start of code segment
      mov  eax,4     ; EAX will serve as a counter for
                    ; the number words left to be summed
      mov  ebx,0     ; EBX will store the sum
      mov  ecx, x    ; ECX will point to the current
                     ; element to be summed
top:  add  ebx, [ecx]
      add  ecx,4     ; move pointer to next element
      dec  eax   ; decrement counter
```

```
        dec  sum   ; decrement counter
        jnz top  ; if counter not 0, then loop again
done: mov  [sum],ebx  ; done, store result in "sum"
```

So, we have 21 lines of source code to understand and maintain. Due to the nature of the language, we also have to include quite a few source comments to communicate the intent of the code. Those things increase the time and effort programmers must expend to live with this solution. The dependence on comments introduces a risk that the comments and code will no longer match, after the solution has been modified during its long years of production life.

The equivalent code in Java, using iteration to sum the values, looks like this:

```
public int sumArrayIterative(List myFineList) {
    int sum = 0;
    for (Integer value : myFineList) {
        sum += value;
    }
    return sum;
}
```

That's seven lines (four of which are "meat"), which is a lot more understandable than 21 lines. We don't need any source comments to explain the intent of the code, either.

Now let's do the same thing using a lambda expression.

```
public int sumArrayStream(List myFineList) {
    return myFineList.stream().mapToInt(i -> i).sum();
}
```

Using a lambda expression reduces the number of source lines to three (two of which are "meat"). It makes programmers' time just a little bit more impactful, and makes the code just a little bit more habitable.

Another advantage: Those "non-meat" lines in source code are clutter that makes it harder to follow what the code is doing. With the Java iterative solution, almost half the source lines are clutter. With the lambda solution, there's only one line of clutter, and it's nothing more distracting than a closing curly brace.

# Learning Resources

Martin Fowler has written a very clear introduction to *collection pipelines*, which are in essence what we are writing when we use the lambda expression features of OO languages. That article contains numerous examples in several languages.

He also wrote a step-by-step guide to refactoring a loop into a collection pipeline in a way that safely preserves behavior at each step. I suggest this is a useful skill to practice in a dojo setting or on your own.

The ever-practical Brian Marick has been spending considerable time delving into FP in the past couple of years. He has shared his learning journey on Twitter, and has produced a couple of very useful e-books that can help a general software developer get a handle on FP.

*Functional Programming for the Object-Oriented Programmer* and *An Outsider's Guide to Statically-Typed Functional Programming* introduce FP in a way an everyday software development practitioner can relate to, as opposed to the abstract approach offered by most other sources. The latter book seems to have offended some in the FP community for reasons I don't understand. In any case, these are great resources for learning about FP, if you're already an experienced OO developer.

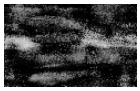
# Conclusion

Using lambda expressions in an OO language doesn't make you a functional programmer, but using ideas from the FP community to produce clean OO code is a Good Thing. OO code that uses a functional *style* will tend to have fewer problems of the kinds that arise from hidden side effects and poor separation of concerns. Lambda expressions are readable and expressive of intent for sections of code that process lists or collections; often moreso than iteration. In Java, for instance, using the explicitly-defined functional interfaces helps us produce reliable code that is compatible with other useful design techniques such as immutable objects.



## Like This Article? Read More From DZone

**Java 8 Concepts: FP, Streams, and Lambda Expressions**



**Don't Fear the Lambda**

**A Little Lambda Tutorial**


Free DZone Refcard
**Getting Started With Kotlin**

Topics: JAVA , FUNCTIONAL PROGRAMMING , LAMBDA EXPRESSIONS , ANTI-PATTERNS , TUTORIAL

# Hello OpenJ9 on Windows, I Didn't Expect You so Soon!

**by Holger Voormann** MVB · **Mar 14, 18 · Java Zone · Tutorial**



Faster startup time, lower memory footprint, and higher application throughput only by replacing the Java Virtual Machine? That sounds too good to be true. So far, there has been no real alternative to Oracle's Java HotSpot VM on Windows. With Eclipse OpenJ9, which emerged from open-sourcing IBM's J9 VM, there is now the alternative that promises exactly this.

At the end of January, the first OpenJDK 9 with Eclipse OpenJ9 nightly builds for Windows were published, but they were not very stable at that time. This week, I tested the nightly builds again to run the Eclipse IDE and I was pleasantly surprised: OpenJ9 ran without crashing. Here are my results: the **start time** of the Eclipse Oxygen.2 Java IDE **improves with OpenJ9 from 20** to 17 seconds, with some tuning (see below) even **to 12 seconds** compared to the Java 9 JDK with Oracle's HotSpot VM on my more than six-year-old laptop. Also, the Windows Task Manager shows **less memory** used by the Eclipse IDE and tasks like compiling a large project are a **bit faster** with OpenJ9.

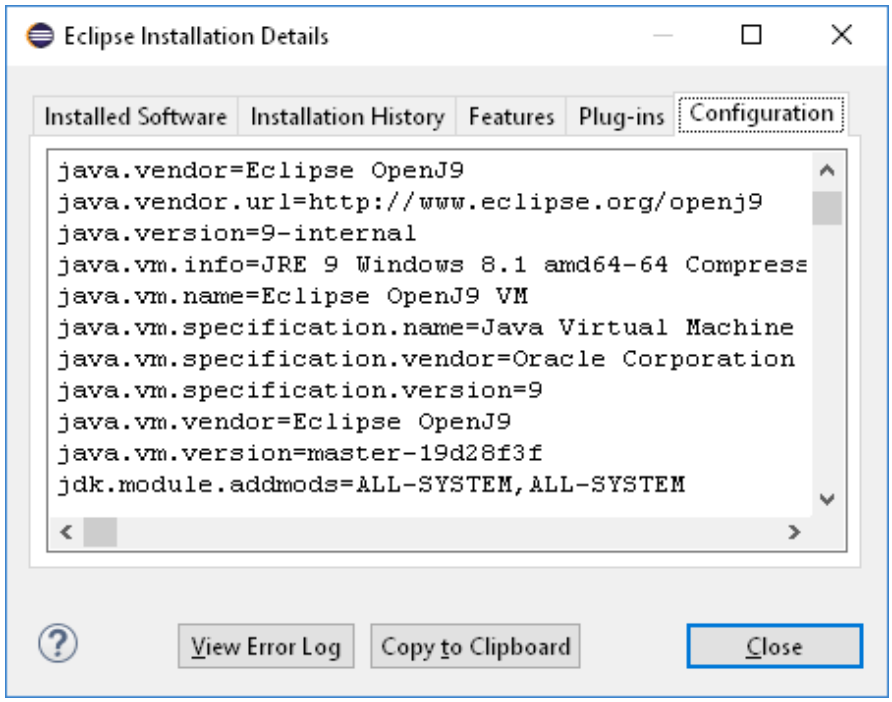
To start the Eclipse IDE with OpenJ9, in `eclipse.ini` add the following two lines above `-vmargs`:

```
-vm
C:\path\to\jdk-9+181\bin\javaw.exe
```

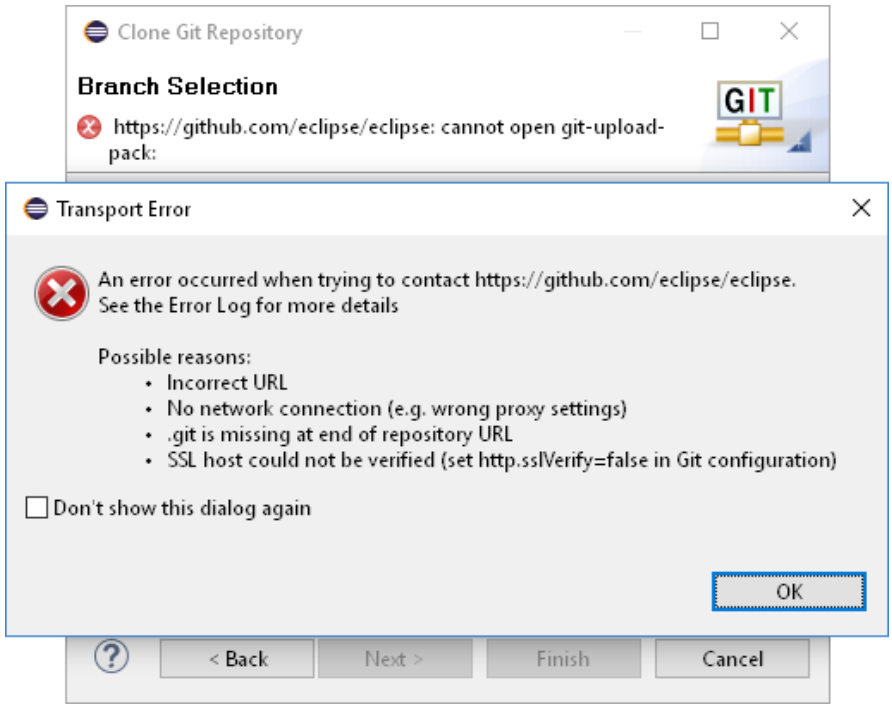
Embedding the JDK into an Eclipse installation directory as a `jre` subdirectory does not yet work, but as long as you do not start the Eclipse IDE from the command line from another directory, you can use `-vm` with ... To further improve the startup time, add the following two lines below

with `jre\bin\javaw.exe`. To further improve the startup time, add the following two lines below `-vmargs`:

```
-Xtune:virtualized
-Xshareclasses:cacheDir=C:\path\to\shareclasses
```

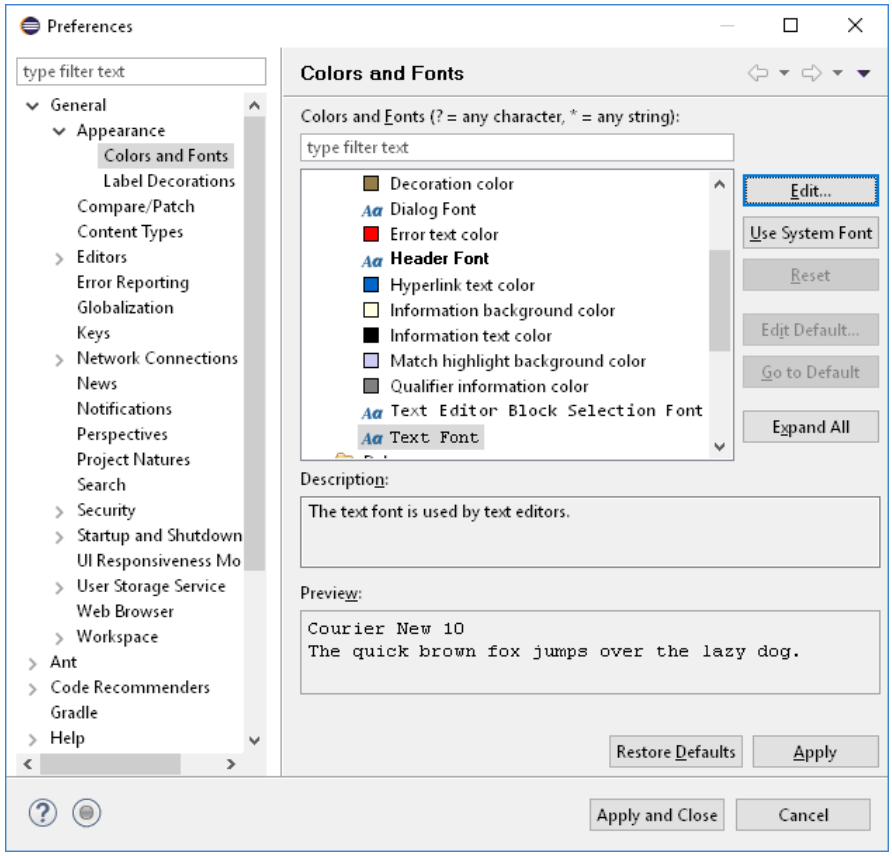



The cloning of a GitHub repository fails due to missing certificate authority (CA) certificates. You can fix this OpenJDK 9 issue by replacing the `lib\security` directory (which contains the `cacerts` file) with the same directory of an OpenJDK 10 early access build.




In the Eclipse IDE that is running on OpenJDK the standard basic text font defaults (for reasons I don't know) to *Courier New* 10 instead of *Consolas* 10. You can change this in *Window* > *Preferences*: *General*

know) to *Courier New* 10 instead of *Consolas* 10. You can change this in *Window* > *Preferences*. *General* > *Appearance* > *Colors and Fonts* by selecting *Basic* > *Text Font* and pressing *Edit*... (if you like, you can also use *Source Code Pro* 9 like the Clean Sheet theme does).




I have not noticed so far any further differences between the Eclipse IDE running on Oracle's JDK 9 and the Eclipse IDE running on the current OpenJDK 9 with OpenJ9 nightly build. Debugging and hot code replace works as expected.

Many thanks to the OpenJ9 team! I look forward to the final release. It's great to have two good open source Java virtual machines for Windows. Who knows, but with only one of the two, neither of the two might be open source today.

PS: If the Eclipse IDE still starts too slowly for you, have a look at a developer build of the upcoming Eclipse Photon IDE.

---



---

**Breakpoints**


**Getting Started With Kotlin**

Topics: JAVA, JVM, ECLIPSE IDE, OPENJ9, TUTORIAL