Java 8 idioms: An easier path to functional programming in Java

Think declaratively to adopt functional techniques in your Java programs

Venkat Subramaniam

February 24, 2017
(First published February 15, 2017)

Learning to program declaratively, rather than imperatively, is an easy first step to adopting functional techniques in your Java programs.

Java™ developers are well accustomed to imperative and object-oriented programming, as these styles have been supported by the Java language since its first release. In Java 8, we gained access to a set of powerful new functional features and syntax. Functional programming has been around for decades, and it's generally more concise and expressive, less error prone, and easier to parallelize than object-oriented programming. So there are good reasons to introduce functional features into your Java programs. Still, programming in the functional style requires some changes in how you design your code.

About this series
Java 8 is the most significant update to the Java language since its inception — packed so full of new features that you might wonder where to start. In this series, author and educator Venkat Subramaniam offers an idiomatic approach to Java 8: short explorations that invite you to rethink the Java conventions you've come to take for granted, while gradually integrating new techniques and syntax into your programs.

I've found that thinking declaratively, rather than imperatively, can ease the transition to a more functional style of programming. In this first article in the new *Java 8 idioms* series, I explain the difference and overlap between imperative, declarative, and functional programming styles. Then, I show you how to use declarative thinking to gradually integrate functional techniques into your Java programs.

The imperative style

Developers trained in the imperative style of programming are accustomed to telling programs what to do, as well as how to do it. Here's a simple example:

Listing 1. findNemo in the imperative style

```
import java.util.*;

public class FindNemo {
  public static void main(String[] args) {
    List<String> names =
      Arrays.asList("Dory", "Gill", "Bruce", "Nemo", "Darla", "Marlin", "Jacques");

    findNemo(names);
  }

  public static void findNemo(List<String> names) {
    boolean found = false;
    for(String name : names) {
      if(name.equals("Nemo")) {
        found = true;
        break;
      }
    }

    if(found)
      System.out.println("Found Nemo");
    else
      System.out.println("Sorry, Nemo not found");
  }
}
```

The `findNemo()` method starts by initializing a mutable *flag* variable, also known as a *garbage variable*. Developers often give such variables throwaway names like `f`, `t`, or `temp`, conveying our general attitude that they shouldn't exist. In this case, the variable is named `found`.

Next, the program loops through one element at a time in the given `names` list. It checks whether the name at hand equals the value that it's looking for, in this case, `Nemo`. If the value matches, it sets the flag to `true` and instructs the flow of control to "break" out of the loop.

Because this is an imperative-style program — the most familiar style for many Java developers — you define every step of the program: you tell it to iterate over each element, compare the value, set the flag, and break out of the loop. The imperative style gives you full control, which is sometimes a good thing. On the other hand, you do all the work. In many cases, you could be more productive by doing less.

The declarative style

Declarative programming means that you still tell the program what to do, but you leave the implementation details to the underlying library of functions. Let's see what happens when we rework the `findNemo` method from Listing 1 using the declarative style:

Listing 2. findNemo in the declarative style

```
  public static void findNemo(List<String> names) {
    if(names.contains("Nemo"))
      System.out.println("Found Nemo");
    else
      System.out.println("Sorry, Nemo not found");
  }
```

Note first that you don't have any garbage variables in this version. You are also not wasting effort on looping through the collection. Instead, you let the built-in `contains()` method do the work. You've still told the program what to do — check whether the collection holds the value we're seeking — but you've left the details to the underlying method.

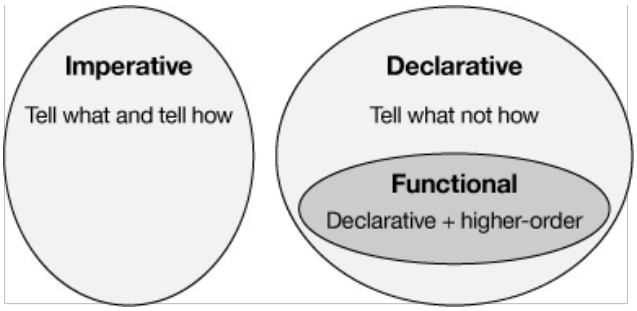
In the imperative example, you controlled the iteration and the program did exactly as instructed. In the declarative version, you don't care how the work happens, as long as it gets done. The implementation of `contains()` may vary, but as long as the result is what you expect, you are happy with that. Less effort to get the same result.

Training yourself to think in a declarative style will greatly ease your transition to functional programming in Java. The reason? The functional style builds on the declarative style. Thinking declaratively offers a gradual transition from imperative to functional programming.

The functional style

While programming in a functional style is always declarative, simply using a declarative style doesn't equate to functional programming. This is because functional programming combines declarative methods with higher order functions. Figure 1 visualizes the relationships between the imperative, declarative, and functional programming styles.

Figure 1. Connecting the imperative, declarative, and functional styles



Higher order functions in Java

In Java, you pass objects to methods, create objects within methods, and return objects from methods. You can do the same with functions. That is, you can pass functions to methods, create functions within methods, and return functions from methods.

In this context, a *method* is part of a class — static or instance — but a function can be local to a method and cannot be intentionally associated with a class or an instance. A method or a function that can receive, create, or return a function is considered a *higher order function*.

A functional programming example

Adopting a new programming style requires changing how you think about your programs. It's a process that you can practice with simple examples, building up to more complex programs.

Listing 3. A Map in the imperative style

```
import java.util.*;

public class UseMap {
  public static void main(String[] args) {
    Map<String, Integer> pageVisits = new HashMap<>();

    String page = "https://agiledeveloper.com";

    incrementPageVisit(pageVisits, page);
    incrementPageVisit(pageVisits, page);

    System.out.println(pageVisits.get(page));
  }

  public static void incrementPageVisit(Map<String, Integer> pageVisits, String page) {
    if(!pageVisits.containsKey(page)) {
       pageVisits.put(page, 0);
    }

    pageVisits.put(page, pageVisits.get(page) + 1);
  }
}
```

In Listing 3, the `main()` function creates a `HashMap` that holds the number of page visits for a website. Meanwhile, the `incrementPageVisit()` method increases a count for each visit to the given page. We'll focus on this method.

The `incrementPageVisit()` method is written in an imperative style: Its job is to increment a count for the given page, which it stores in the `Map`. The method doesn't know if there's already a count for the given page, so it first checks to see if a count exists. If not, it inserts a "0" as the count for that page. It then gets the count, increments it, and stores the new value in the `Map`.

Thinking declaratively requires you to shift the design of this method from "how" to "what." When the method `incrementPageVisit()` is called, you want to either initialize the count for the given page to 1 or increment the running value by one count. That's the *what*.

Because you're programming declaratively, the next step is to scan the JDK library for methods in the `Map` interface that can accomplish your goal — that is, you're looking for a built-in method that knows *how* to accomplish your given task.

It turns out that the `merge()` method suits your purposes perfectly. Listing 4 uses your new declarative approach to modify the `incrementPageVisit()` method from Listing 3. In this case, however, you're not just adopting a more declarative style by choosing a smarter method; because `merge()` is a higher order function, the new code is actually a good example of the functional style:

Listing 4. A Map in the functional style

```
public static void incrementPageVisit(Map<String, Integer> pageVisits, String page) {
    pageVisits.merge(page, 1, (oldValue, value) -> oldValue + value);
}
```

In Listing 4, `page` is passed as the first argument to `merge()`: the key whose value should be updated. The second argument serves as the initial value that will be given for that key *if* it doesn't

already exist in the `Map` (in this case, "1"). The third argument, a lambda expression, receives as parameters the current value in the map for the key and a variable holding the value passed as the second argument to merge. The lambda expression returns the sum of its parameters, in effect incrementing the count. (*Editorial note*: Thanks to István Kovács for correcting an error in the code.)

Compare the single line of code in the `incrementPageVisit()` method in Listing 4 with the multiple lines of code from Listing 3. While the program in Listing 4 is an example of the functional style, thinking through the problem declaratively helped us make the leap.

Conclusion

You have much to gain from adopting functional techniques and syntax in your Java programs: the code is concise, more expressive, has fewer moving parts, is easier to parallelize, and is generally easier to understand than object-oriented code. The challenge is to change your thinking from the imperative style of programming — which is familiar to the vast majority of developers — to thinking declaratively.

While there's nothing easy or straightforward about functional programming, you can take a big leap by learning to focus on *what* you want your programs to do, rather than *how* you want it done. By allowing the underlying library of functions to manage execution, you will gradually and intuitively get to know the higher order functions that are the building blocks of functional programming.

Related topics

- Java 8 language changes
- Java 8 concurrency basics
- Functional thinking, Part 1: Learn to think like a functional programmer
- Functional Programming in Java: The Pragmatic Bookshelf, 2014
- IBM Code: Java journeys