IBM

developerWorks®

# Java 8 idioms: Function composition and the Collection Pipeline pattern

## Functional patterns for iterating collections in Java

Venkat Subramaniam

March 30, 2017

Endless looping isn't the only way to iterate collections in your code. Function Composition and Collection Pipeline are two patterns that let you use expressions, rather than statements, to sort collections in Java.

Certain functional design patterns will naturally show up in your programs when you begin to adopt the functional style of programming, but you will still need to work at mastering them. This article introduces Function Composition and Collection Pipeline, two functional-style patterns that you can combine to iterate collections in your code. Understanding the mechanics of these patterns will help you structure your Java™ programs to make the most of higher-order functions and lambda expressions.

**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.

## Statements and expressions

If you do a quick grep of `for` in your codebase, you might be surprised to realize how often you use `for` loops in your code. I call this the *`for` hammer*: anytime we need to loop, we seem to reach for `for`.

In Java, both `for` and `while` are statements. *Statements* perform an action but do not yield any results. By their nature, any statement that does something useful will result in a mutation of data. That's the only way statements can convey their effect. *Expressions* are the opposite: they can yield results without causing mutation.

Using statements in your code is like collaborating on a piece of work without being able to directly pass that work between team members. The only way to share something is by placing it on a table or a shelf, where another team member can then retrieve it. Expressions work more like a chain: when one person completes a task, they hand off the item to the next person in the chain.

Expressions facilitate the Collection Pipeline pattern, which Martin Fowler described as a sequence of operations where one operation's collected output is fed into the next. While the Collection Pipeline pattern is used in object-oriented programming (you've probably seen it in code that uses object builders) it's more prominent in functional programming.

Function Composition and Collection Pipeline patterns are two patterns that work together. In the next section we'll solve a problem using the familiar `for` statement. Then you'll see how to solve the same problem more efficiently using these two patterns.

## Iterating and sorting with statements

Let's start with a `Car` class that has the properties of make, model, and year:

### Listing 1. A Car class

```
package agiledeveloper;

public class Car {
  private String make;
  private String model;
  private int year;

  public Car(String theMake, String theModel, int yearOfMake) {
    make = theMake;
    model = theModel;
    year = yearOfMake;
  }

  public String getMake() { return make; }
  public String getModel() { return model; }
  public int getYear() { return year; }
}
```

We can add a collection of `Car` instances, like so:

### Listing 2. A collection of Car instances

```
package agiledeveloper;

import java.util.Collections;
import java.util.Comparator;
import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;
import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;

public class Iterating {
  public static List<Car> createCars() {
    return Arrays.asList(
      new Car("Jeep", "Wrangler", 2011),
      new Car("Jeep", "Comanche", 1990),
      new Car("Dodge", "Avenger", 2010),
      new Car("Buick", "Cascada", 2016),
      new Car("Ford", "Focus", 2012),
      new Car("Chevrolet", "Geo Metro", 1992)
    );
  }
```

In Listing 3, we use imperative-style programming to iterate over the list and get the names of cars made later than the year 2000. We then sort the models in ascending order by year.

## Listing 3. Models sorted by year using 'for'

```
public static List<String> getModelsAfter2000UsingFor(List<Car> cars) {
    List<Car> carsSortedByYear = new ArrayList<>();

    for(Car car : cars) {
      if(car.getYear() > 2000) {
        carsSortedByYear.add(car);
      }
    }

    Collections.sort(carsSortedByYear, new Comparator<Car>() {
      public int compare(Car car1, Car car2) {
        return new Integer(car1.getYear()).compareTo(car2.getYear());
      }
    });

    List<String> models = new ArrayList<>();
    for(Car car : carsSortedByYear) {
      models.add(car.getModel());
    }

    return models;
  }
```

As you can see, there's a lot of looping in this code. First, the `getModelsAfter2000UsingFor` method takes a list of cars as its parameter. It extracts or filters out cars made after the year 2000, putting them into a new list named `carsSortedByYear`. Next, it sorts that list in ascending order by year-of-make. Finally, it loops through the list `carsSortedByYear` to get the model names and returns them in a list.

Could we make the code more efficient? We might be able to eliminate one `for` loop by sorting all of the cars at once, but then we would be sorting through a larger list of objects.

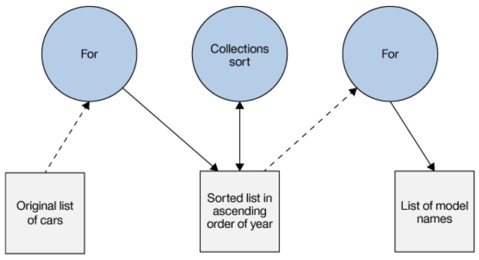
Here is the output for the example list of cars:

## Listing 4. Output of getModelsAfter2000UsingFor

```
Avenger, Wrangler, Focus, Cascada
```

This short example demonstrates what I call the *effect of statements*. While functions and methods in general can be used as expressions, the `Collectionssort` method doesn't return a result. Because it is used as a statement, it mutates the list given as argument. Both of the `for` loops also mutate lists as they iterate. Being statements, that's just how these elements work. As a result, the code contains unnecessary garbage variables, as shown in Figure 1.

**Figure 1. The effect of statements**



## Iterating and sorting with a collection pipeline

In functional programming, it's common to sequence complex operations through a series of smaller modular functions or operations. The series is called a *composition of functions*, or a function composition. When a collection of data flows through a function composition, it becomes a collection pipeline. Function Composition and Collection Pipeline are two design patterns frequently used in functional-style programming.

Instead of using one big `for` hammer we can use multiple specialized tools, depending on the problem at hand. Rather than use statements for everything, which is common in imperative-style programming, functional-style programming encourages us to use expressions. Expressions don't have the side-effect that statements do, of mutating objects. An expression like `filter` or `map` also returns a result that we can pass to another function, which is how we create a collection pipeline. Consider the code in Listing 5.

## Listing 5. Function composition in the Collection Pipeline pattern

```
public static List<String> getModelsAfter2000UsingPipeline(
   List<Car> cars) {
   return
     cars.stream()
         .filter(car -> car.getYear() > 2000)
         .sorted(Comparator.comparing(Car::getYear))
         .map(Car::getModel)
         .collect(toList());
 }
```

The method `getModelsAfter2000UsingPipeline` produces the same result as the method `getModelsAfter2000UsingFor`, from Listing 3, but note the differences in the code:

- The functional-style code is more concise than the imperative-style code.
- The functional-style code doesn't perform explicit mutability and uses fewer garbage variables.
- The functions/methods used in the second method are all expressions returning values. Contrast this to the `Collections.sort` method in Listing 3.
- The `getModelsAfter2000UsingPipeline` uses the Collection Pipeline pattern and is very expressive.

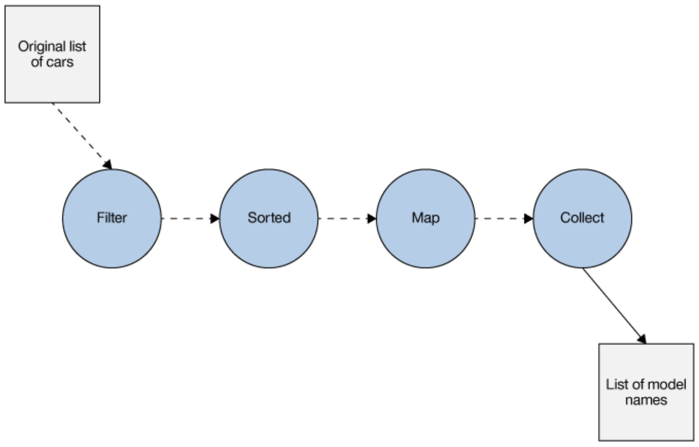
With just a few lines, the intent of the code is clear—given a collection of cars, filter or extract only those made in or after the year 2000; then sort by year, map or transform the objects to their model names, and collect the results into a list.

Part of what makes the code in Listing 5 concise and expressive is the use of method references. Passing a lambda expression to the `filter` makes sense because it gets the year property of the given object and compares it to the year 2000. Passing a lambda expression to the `map` would not be so effective, however. The expression passed to the `map` method would be `car -> car.getModel()`, which is rather trivial. The lambda expression merely returns a property of the given object and does not perform any real computation or operation. We are better off replacing it with a method reference.

Instead of passing a lambda expression to the `map` method, we passed the method reference `Car::getModel`. Likewise, instead of passing the lambda expression `car -> car.getYear()` to the comparing method, we passed the method reference `Car::getYear`. Method references are short, concise, and expressive. It is best to use them wherever possible.

Figure 2 shows the collection pipeline from Listing 5.

### Figure 2. The charm of Collection Pipeline



Looking at Figure 2, you see how the `getModelsAfter2000UsingPipeline` function executes the collection pipeline, transforming the given input through a series of functions. As the data moves through the functions, Java 8's lazy evaluation and function fusing capabilities (see *Functional Programming In Java*, 2014) can help us avoid creating intermediate objects in some cases. Functions also do not make intermediate objects visible or available as the data transitions through the pipeline.

## Conclusion

In imperative-style programming, it is common to use `for` and `while` loops for most kinds of data processing. In this article you've learned an alternative approach that is very popular in functional-style programming. Function composition is a simple technique that lets you sequence modular functions to create more complex operations. When you run data through the sequence, you have a collection pipeline. Together, the Function Composition and Collection Pipeline patterns enable you to create sophisticated programs where data flow from upstream to downstream and is passed through a series of transformations.

## Related topics

- Martin Fowler on the Collection Pipeline pattern
- Lambda expressions and method references in Java 8
- Functional design patterns, Part 1: How patterns manifest in the functional world
- Functional Programming in Java: The Pragmatic Bookshelf, 2014
- IBM Code: Java journeys