


The
Pragmatic
Programmers

Cruising Along with Java

Modernize and Modularize
with the Latest Features



Venkat Subramaniam
Edited by Jacquelyn Carter

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit <https://www.pragprog.com>.

Copyright © The Pragmatic Programmers, LLC.

An experienced programmer doesn't write more code, faster. Quite the opposite, as they gain experience, they find ways to reduce code, clutter, complexity, and chances of error. They write code to solve the problem on hand and quickly refactor it to make it concise, expressive, easier to understand, and easier to maintain. A common concept that often comes to their aid during such refactoring efforts is trading statements for expressions.

Statements perform actions but don't return any results to the caller. By nature, they promote side-effects and often require mutating variables. The result is code that's generally verbose, hard to reason, hard to change, error-prone, and sometimes outright unpleasant to work with. They also don't compose—each statement is executed in isolation from the next. There's no way to chain statements.

Expressions perform computations and return their results to the caller. Well-written expressions are pure and idempotent, don't cause side-effects, and don't mutate any variables. We can also build on the results of one expression to execute another expression; they compose really well. The benefit is highly concise code that's easier to express, easier to understand, less error-prone, and often pleasant to work with.

In Java, `switch` has been used only as a statement for years. Now, in addition, it can also be used as an expression. This capability, introduced in Java 12, can reduce the size and complexity in existing code and will also change the way we write new code, so we can take advantage of all the benefits of using expressions.

In this chapter we will start by converting from a `switch` statement to an expression, discuss the benefits, and dive into the different capabilities and differences of `switch` expression compared to the old `switch` statement. This chapter will prepare you for the greater good that's waiting in the next chapter where `switch` turns from an expression to a full-blown pattern matching syntax.

From Statements to Expressions

Let's see how the `switch` expression can deliver a better signal-to-noise ratio. We'll start with a piece of code that uses `if-else`, refactor that to use the `switch` statement, discuss why that's not sufficient, and move forward into transforming that code to use the `switch` expression.

In some languages, like Ruby, Scala, and Kotlin, `if` is an expression. In Java, `if` is a statement (though there `if` is the ternary operator which is an expression). That means you can't assign the result of the evaluation of `if` to a variable

because it doesn't yield a result of its execution. This often forces us to create a variable and mutate it.

Where there's an if, there's probably an else that tags along to perform an alternative action if the condition provided to if isn't met. That's more code we write, to discern between the if and the else parts.

The logic in many applications usually doesn't end with a plain either-or situation. The if-else often flows along into a long series of if-else statements. Looking back at the code we create, we often realize how noisy and cluttered the code is, with all the recurring if and else. We end up with rather icky code, to say the least. Such code is often a great candidate for refactoring to switch. Let's look at an example.

Let's take the familiar example of computing the grades from scores, where different values in a range from 0 to 100 map to grades of 'A' to 'F'. Here's a class `Grade` with a `main()` method that prints the grades of a few different scores.

```
switch/vsca/Grade.java
```

```
public class Grade {  
    public static void main(String[] args) {  
        List.of(59, 64, 76, 82, 89, 94, 100)  
            .stream()  
            .map(Grade::gradeFor)  
            .forEach(System.out::println);  
    }  
}
```

It transforms the given scores to a string representation of the grade for each score and prints it, using a yet-to-be-written method `gradeFor()`. We want to implement the `gradeFor()` method so the program will output a result in the format:

```

Grade for score 59 is F
Grade for score 64 is D
Grade for score 76 is C
Grade for score 82 is B
Grade for score 89 is B
Grade for score 94 is A
Grade for score 100 is A

```

If we were using an older version of Java, we'd have to unleash a series of if-else statements to implement the `gradeFor()` method, like so:

`switch/vsca/Grade.java`

```

public static String gradeFor(int score) {
    String letterGrade = "";

    if(score >= 90) {
        letterGrade = "A";
    } else if(score >= 80) {
        letterGrade = "B";
    } else if(score >= 70) {
        letterGrade = "C";
    } else if(score >= 60) {
        letterGrade = "D";
    } else {
        letterGrade = "F";
    }

    return "Grade for score %d is %s".formatted(score, letterGrade);
}

```

The logic is rather simple, but the number of if and else in the code is the source of the noise in it. That reminds me of a project with many functions that had sequences of if-else that ran more than 70 levels deep. That project was filled with several pieces of code that can't be unseen.

Not all if-else sequences can be refactored to using a switch. But this example can be converted to using a switch, due to the nice structure in the range of values being compared. Let's see how the traditional switch statement holds up in comparison to the if-else maze.

```

public static String gradeFor(int score) {
    String letterGrade = "";

    switch(Math.min(score / 10, 10)) {
        case 10:
            letterGrade = "A";
            break;
        case 9:
            letterGrade = "A";
            break;

```

```

    case 8:
        letterGrade = "B";
        break;
    case 7:
        letterGrade = "C";
        break;
    case 6:
        letterGrade = "D";
        break;
    default:
        letterGrade = "F";
        break;
}

return "Grade for score %d is %s".formatted(score, letterGrade);
}

```

That probably leaves you with a mixed feeling. The good news is it's less noisy and looks less cluttered compared to the if-else version. Kudos for the refactoring effort for that. But this version has more lines of code and still mutates the `letterGrade` variable. There's also an additional risk: if we forget the `break` statements, then the grade computed would be incorrect—such bugs may start a campus riot if left undetected. (In my youth I actually started a few riots on campuses, but that's a story for another book.)

The use of `switch` is in the right direction. But the `switch` statement brings its own set of problems with it. First, it's a statement and, thus, has side-effects, such as mutability, and all the smells we're often told to avoid in good programming practices. Second, the flow has to be controlled explicitly using `break`, and it's a common mistake among programmers to forget that, especially when altering code.

The `switch` expression greatly improves upon the `switch` statement. Instead of using a *colon case*, a `switch` expression uses an *arrow case* where each path is an expression and has an auto-break. In other words, a `switch` expression is like a rectified `switch` statement with the ill behaviors removed.

Let's refactor the code to turn the `switch` statement into a `switch` expression:

```

public static String gradeFor(int score) {
    final String letterGrade = switch(Math.min(score / 10, 10)) {
        case 10 -> "A";
        case 9 -> "A";
        case 8 -> "B";
        case 7 -> "C";
        case 6 -> "D";
        default -> "F";
    };
}

```

```
    return "Grade for score %d is %s".formatted(score, letterGrade);  
}
```

Seeing that is like feeling a breath of fresh air. That code has the perfect logic-to-break ratio. You may place the cases in any order in this example, with the default at the end. Depending on the value of the expression passed as an argument (within the parentheses) to `switch()`, one and exactly one case path is taken. Once the expression in a path is evaluated, it's immediately returned as the result of the `switch` expression. You're not allowed to place a `break` statement in the middle of a `switch` expression—good riddance. The arrow case has the label, followed by an arrow `->`, and then the expression that should be evaluated if the value passed in matches the given label.

When compared to the `if-else` version and the `switch statement` version, this version is less noisy, shorter, crisp, easy to read, easy to follow, avoids mutability, has no side-effect, and is overall pleasant to work with. It's far easier to reason about than the other versions as well.

Let's venture further into the features of `switch` expression so you can reap its full benefits.