Java Switch Statement

Introduction

In this article from my free Java 8 Course, I will be discussing the Java switch statement.

Switch Statements

The switch statement is another type of conditional. It is similar to an if-statement, but in some cases it can be more concise. You don't have to think about the exact meaning just yet, as we will get to it in the example below:

We're going to use this series of status enums (shown in Example 1), from my article about enums, for our examples. But instead of using a series of if-statement branches, we use switch to operate on the enum's values.

```
package com.marcusbiel.java8course;

public enum LoggingLevel {
     PENDING, PROCESSING, PROCESSED;
}
```

Example 1

The switch statement is shown in Example 2. Syntactically, it is similar to an if-statement, but instead of writing `if`, you write `switch`. Inside of the switch we write the variable we are trying to compare, in this case, state. Each case is a potential value of switch we are trying to compare to. That is to say, `case PENDING:` is roughly the same as saying: `if (status == Status.PENDING)`. The inside of each case is similar to inside the curly brackets of an if-statement. To terminate a case, you can type the word `break`, or write a `return` statement. If you don't have either of these, the code 'falls through', an idea that I'll explain in the next section.

If you only have one value, then an if-statement is more concise, as switch statements have the added overhead of writing `switch (status) {...}`. However, a switch statement is preferable when you iterate through multiple values with short code blocks, such as a method call, inside in each condition. The more conditions you have, the shorter your switch statement is compared to the equivalent branches of if-statements.

```
LoggingLevel state = LoggingLevel.PENDING;

switch (state) {
     case PENDING:
          onPending();
          break;
     case PROCESSING:
          onProcessing();
          break;
     case PROCESSED:
          onProcessed();
          break;
}
```

Example 2

Another alternative to avoid falling through is to return a value inside the switch statement to leave the surrounding method, as shown in Example 3. This exits the method, and therefore doesn't fall through to the rest of the switch statement. In this case, the return value of our process method is `String`. If no case is met, we will return a default `String` instead.

```
private String process(LoggingLevel state) {
     switch (state) {
          case PENDING:
               return pendingAsString();
          case PROCESSING:
               return processingAsString();
          case PROCESSED:
               return processedAsString();
     }
     return defaultString();
}
```

Example 3

Falling Through

The code in Example 4 is an example of a switch statement lacking a **break statement** for the case `PROCESSING`. Given a `PENDING` status, we would call the pending logic, leaving the switch altogether with the break statement. This works as intended.

```
switch (state) {
    case PENDING:
        onPending();
        break;
    case PROCESSING:
        onProcessing();
    case PROCESSED:
        onProcessed();
        break;
}
```

Example 4

However, given a PROCESSING status, we end up calling both the `onProcessing()` and the `onProcessed()` method. In other words, the `PROCESSING` case **falls through** to the `PROCESSED` case. In this example, both cases have their own independent code within, so it probably wasn't intended to fall through.

Falling through can be useful though when you want two cases to execute the same code. Since there's no way to say "and" in a switch statement, to avoid duplicate code you could fall through in the first case and then write the code in the second (See Example 5).

```
switch (state) {
    case PENDING:
        onPending();
        break;
    case PROCESSING:
        /*
        *falling through
        */
    case PROCESSED:
        onProcessingOrProcessed();
        break;
}
```

Example 5

Switch statements that fall through are a big source of bugs and are very dangerous as the lack of break statements can be difficult to spot. If you do intend to have your switch statement fall through, leave a comment saying so, as if to tell the next programmer, "this switch-case is falling through on purpose." - Otherwise he might be tempted to "fix" it.

The Default Clause

The syntax for switch statements doesn't require a case for all possible values. You can leave cases out. When no matching case is found, our switch statement finishes without doing anything. An alternative to not doing anything when no cases match is to add a default clause, as I did in Example 6, which matches when none of the other case clauses apply.

```
switch (state) {
     case PENDING:
          onPending();
          break;
     case PROCESSING:
          onProcessing();
          break;
     case PROCESSED:
          onProcessed();
          break;
     default:
          onDefault();
}
```

Example 6

The same rules and pitfalls of falling through apply here. On the surface, most switch statements with a default clause look innocent enough until you consider what can happen when the next guy comes along and adds a new value or two to our existing Status enum. Let's say he adds the ERROR value. Not knowing our enum's use in one or more switch statements, he would inadvertently cause unintended matches on the default clause due to the lack of an ERROR case! We usually don't want that, so I recommend that you either not use the default clause, or better yet, have it throw an error. (If you want to know more about this, read my article on checked and unchecked exceptions.)

A Few Caveats

Don't overuse switch — and by extension — don't overuse if-statements. These conditional statements, when overused, are infamous for overly complex code, often written with nested if, switch, or even for-loop statements.

I have seen too many codebases with methods having code margins up to eight levels or more deep, and almost always the culprits are poorly-organized, tedious, deep litanies of conditional statements and the blocks they contain.

These are relics of the past eras of procedural programming—a style of programming that builds upon procedures which makes you think like a machine. But humans don't usually think like this. The cool thing about object-oriented programming is that it's very close to how humans think. This allows us to talk to business guys, and when done properly, makes our code read very nicely.

The switch statement is not object-oriented. In Object-Oriented-Programming (OOP), there are some corner cases where it fits the bill as it is quick and easy to use. These are often instances where you would only have very few cases making up a small switch statement. However, there are too many instances where people would write hundreds of lines of code amounting to *one* long, crazy, hard-to-read switch statement. There are object-oriented techniques for reducing the complexity of such statements that help us express the same meaning. For example, we could represent the `PENDING` status with an object, the `PROCESSED` state with another, and so on. I won't discuss this in any more detail because these ideas are beyond the scope of this article; in summary, stick to only using switch statements for a very small number of cases.

Thanks for reading!