JAVA SE

New switch Expressions in Java 12

A new preview feature makes switch statements friendlier and less error-prone.

by Raoul-Gabriel Urma and Richard Warburton

[Note: This article applies only to Java 12. Switches operate somewhat differently in Java 13 and later releases. —*Ed.*]

JDK 12 was released in March 2019. It's the third release using the six-month release cycle that was announced with the release of Java 9. What's in store this time? This article focuses on the new language feature available in preview mode: switch expressions. Our next article will cover the changes in the JDK, including the garbage collectors G1 and Shenandoah.

Switch Expressions

For those who get excited about new language features, you are going to be pleased with Java 12, which contains enhanced switch expressions. However, this feature is available only in preview mode currently. This means that simply running the Java compiler as usual won't work if you use the new switch expression syntax. To enable this feature, you'll need to use the flags `--enable-preview` and `--release 12` when you compile your code. To compile the code snippets in this article, make sure you have JDK 12 installed and use the following command:

```
javac --enable-preview --release 12 Example.java
```

To run the generated class file, you'll need to pass the `--enable-preview` flag to the Java launcher:

```
java --enable-preview Example
```

Before looking at the new feature, let's review what preview mode is. According to the official documentation, "A preview language or VM feature is a new feature of the Java SE Platform that is fully specified, fully implemented, and yet *impermanent*. It is available in a JDK feature release to provoke developer feedback based on real-world use; this may lead to it becoming permanent in a future Java SE Platform. In other words, it can still be refined or even removed." [Emphasis added. —*Ed.*]

So what's wrong with the switch statement as you currently know it? There are four improvements that we discuss: fall-through issue, compound form, exhaustiveness, and expression form.

Fall-through issue. Let's start with the fall-through behavior. In Java, you typically write a switch as follows:

```
switch(event) {
    case PLAY:
        //do something
        break;
```

```
    case STOP:
        //do something
        break;
    default:
        //do something
        break;
}
```

Note all the `break` statements within each block that handles a specific `case` clause. The `break` statement ensures that the next block in the `switch` statement is not executed. What happens if you miss the `break` statement, though? Will the code still compile? Yes it will. As a quiz, try to guess the console output of the following code:

```
var event = Event.PLAY;

switch (event) {
    case PLAY:
        System.out.println("PLAY event!");
    case STOP:
        System.out.println("STOP event");
    default:
        System.out.println("Unknown event");
}
```

The code prints the following:

```
PLAY event!
STOP event
Unknown event
```

This behavior in a switch is called *fall through*. As described in Oracle's Java SE documentation, "All statements after the matching `case` label are executed in sequence, regardless of the expression of subsequent `case` labels, until a `break` statement is encountered."

The fall-through behavior can lead to subtle bugs when you simply forget to include a `break` statement. Consequently, the behavior of the program could be incorrect. In fact, the Java compiler warns you of suspicious fall through if you compile with `- Xint:fallthrough`. The issue is also picked up by code checkers, such as Error Prone.

This new switch form uses the lambda-style syntax introduced in Java 8 consisting of the arrow between the label and the code that returns a value.

This issue is also mentioned in JDK Enhancement Proposal (JEP) 325 as a motivation for the enhanced form of `switch`: "The current design of Java's `switch` statement follows closely languages such as C and C++, and supports fall-through semantics by default. Whilst this traditional control flow is often useful for writing low-level code (such as parsers for binary encodings), as `switch` is used in higher-level contexts, its error-prone nature starts to outweigh its flexibility."

Now in Java 12 (with `--enable-preview` activated), there's a new syntax for switch that has no fall through and, as a result, can help reduce the scope for bugs. Here's how you'd refactor the previous code to make use of this new `switch` form:

```
switch (event) {
    case PLAY -> System.out.println("PLAY event!");
    case STOP -> System.out.println("STOP event");
    default -> System.out.println("Unknown event");
};
```

This new `switch` form uses the lambda-style syntax introduced in Java 8 consisting of the arrow between the label and the code that returns a value. Note that these are not actual lambda expressions; it's just that the syntax is lambda-like. You can use single-line expressions or curly-braced blocks just as you can with the body of a lambda expression. Here's an example that shows the syntax of mixing single-line expressions and curly-braced blocks:

```
switch (event) {
    case PLAY -> {
        System.out.println("PLAY event!");
        counter++;
    }
    case STOP -> System.out.println("STOP event");
    default -> System.out.println("Unknown event");
};
```

Compound cases. Next is dealing with multiple case labels. Before Java 12, you could use only one label for each case. For example, in the following code, despite the fact that the logic for `STOP` and `PAUSE` is the same, you'd need to handle two separate cases unless you use fall through:

```
switch (event) {
    case PLAY:
        System.out.println("User has triggered the p
        break;
    case STOP:
        System.out.println("User needs to relax");
        break;
    case PAUSE:
        System.out.println("User needs to relax");
        break;
}
```

A typical way to reduce the verbosity is to use the fall-through semantics of `switch` as follows:

```
switch (event) {
    case PLAY:
        System.out.println("User has triggered the p
        break;
    case STOP:
    case PAUSE:
        System.out.println("User needs to relax");
        break;
}
```

However, as discussed earlier, this style can lead to bugs because it's not clear whether the `break` statement is missing or intentional. If there were a way to specify that the handling is the same for the cases `STOP` and `PAUSE`, that would provide more clarity. That's exactly what is now possible in Java 12. Using the arrow-syntax form, you can specify multiple `case` labels. The previous code can be refactored like this:

```
switch (event) {
    case PLAY ->
        System.out.println("User has triggered the p
    case STOP, PAUSE ->
        System.out.println("User needs to relax");
};
```

In this code, the labels are simply listed consecutively. The code is now more concise and the intent clear.

Exhaustiveness. Another benefit of the new `switch` form is exhaustiveness. This means that when you use `switch` with an enum,

the compiler checks that for any possible value there is a matching `switch` label.

For example, if you have the following `enum` type:

```
public enum Event {
    PLAY, PAUSE, STOP
}
```

And you create a switch that covers some but not all the values, such as the following:

```
switch (event) {
    case PLAY -> System.out.println("User has trigge
    case STOP -> System.out.println("User needs to 
}; // compile error
```

Then, in Java 12, the compiler will generate this error:

```
error: the switch expression does not cover all pos
```

This error is a useful reminder that a default clause is missing or that you've forgotten to deal with all possible values.

Expression form. The expression form is the other improvement of the old `switch` statement: To understand what an "expression form" means, it's worthwhile reviewing the difference between a statement and an expression.

Statements are essentially "actions." Expressions, however, are "requests" that produce a value. Expressions are fundamental and simple to understand, which leads to better code comprehension and easier maintenance.

In Java, you can clearly see the distinction as it exists between an `if` statement and the ternary operator, which is an expression. The following code highlights this difference.

```
String message =  "";
if (condition) {
    message = "Hello!";
}
else {
    message = "Welcome!";
}
```

This code could be rewritten as the following expression:

```
String message = condition ? "Hello!" : "Welcome!";
```

Before Java 12, `switch` was a statement only. Now, though, you can also have switch expressions. For example, take a look at this code that processes various user events:

```
String message = "No event to log";
switch (event) {
    case PLAY:
        message = "User has triggered the play butt
        break;
    case STOP:
        message = "User needs a break";
        break;
}
```

This code can be written as a concise switch expression form that better indicates the intent of the code:

```
var log = switch (event) {
    case PLAY -> "User has triggered the play buttor
    case STOP -> "User needs a break";
    default -> "No event to log";
};
```

Note how the `switch` now returns a value—it's an expression. This expression form for a switch also gives you the ability to use a `return` statement within a `case` block. However, we recommend extracting complex code logic into a private helper method with a meaningful name if you feel that things are becoming difficult to understand. You can then simply call this method using the expression-style syntax.

Toward Pattern Matching

You may be wondering what the motivation is for yet another language feature. Since Java 8, functional programming has clearly influenced the evolution of Java:

- Java 8 brought lambda expressions and streams.
- Java 9 included the Flow API to support reactive streams.
- Java 10 introduced local variable type inference.

All these ideas have been available in functional programming languages such as Scala and Haskell for a long time. So what's the latest idea for Java? It's (structural) *pattern matching*, which should not be confused with regular expressions.

Switch expressions are a helpful addition that will enable you to write code that is a bit more concise and less error-prone.

Pattern matching, as it's meant here, is the idea that you test whether an object is of a particular structure before extracting certain parts of that structure to do some processing. This is typically accomplished in Java by using a combination of `instanceof` checks and cast expressions. A common situation in which to do this is when you need to write a parser for a domain-specific language. For example, the following code checks whether an object is of a particular type before casting it so that information can be extracted from it.

```
if(o instanceof BinOp) {
    var binop = (BinOp) o;
    // use specific methods of the BinOp class
}
else if (o instanceof Number) {
    var number = (Number) o;
    // use specific methods of the Number class
}
```

Java doesn't yet support a language feature to provide full pattern matching, but that is currently being discussed as a potential addition, as described in JEP 305.

As an initial step in that direction, Java 12 needed to enhance the switch functionality that has existed since the first version of Java by making it an expression form. Through the introduction of this new feature, Java augments the switch syntax to enable future enhancements.

Conclusion

Java 12 doesn't bring any new language feature that you can readily use. However, it brings switch expressions, which are available as a preview

language feature. Switch expressions are a helpful addition that will enable you to write code that is a bit more concise and less error-prone. In particular, switch expressions provide four improvements: fall-through semantics, compound form, exhaustiveness, and expression form. Finally, and perhaps more exciting, the syntax available to `switch` has now become richer.

At the moment, in Java 12, the switch cases support only switching on enum, String, byte, short, char, int, and their wrapper classes. However, in the future there may well be more sophisticated forms and support for structural pattern matching on arbitrary "switchable" types.

Acknowledgments. The authors wish to thank Oracle's Java langtools team for providing feedback on this article.

Also in This Issue

Raoul-Gabriel Urma

Raoul-Gabriel Urma (@raoulUK) is the CEO and cofounder of Cambridge Spark, a leading learning community for data scientists and developers in the UK. He is also chairman and cofounder of Cambridge Coding Academy, a community of young coders and students. Urma is coauthor of the best-selling programming book *Java 8 in Action* (Manning Publications, 2015). He holds a PhD in computer science from the University of Cambridge.

Richard Warburton

Richard Warburton (@richardwarburto) is a software engineer, teacher, author, and Java Champion. He is the author of the best-selling *Java 8 Lambdas* (O'Reilly Media, 2014) and helps developers learn via Iteratr Learning and at Pluralsight. Warburton has delivered hundreds of talks and training courses. He holds a PhD from the University of Warwick.

Share this Page