JAVA SE

Java 14 Arrives with a Host of New Features

Java 14 contains more new features than the previous two releases—most of them aimed at easing coding.

by Raoul-Gabriel Urma

February 27, 2020

Java 14 is scheduled for release on March 17. Version 14 includes more JEPs (Java Enhancement Proposals) than Java 12 and 13 combined. So what is in store and what is most relevant for Java developers who write and maintain code on a daily basis?

In this article, I examine the following major advances:

- Improved switch expressions, which first appeared in Java 12 and Java 13 as previews and are now fully part of Java 14
- Pattern matching for `instanceof` (a language feature)
- Helpful `NullPointerException`s (a JVM feature)

If you read this article and try some of these features in your codebase, I encourage you to share your experience by providing feedback to the Java team. By doing so, you have the opportunity to contribute to Java's development.

Switch Expressions

In Java 14, switch expressions become permanent. If you need a refresher on what switch expressions are, they were covered extensively in two previous articles.

In earlier releases, switch expressions were a "preview" feature. As a reminder, features are designated as "preview" to gather feedback and it is possible that they might change or even be removed based on the feedback; but it is expected that most will eventually become permanent in Java.

The benefits of the new switch expressions include reduced scope for bugs due to the absence of fall-through behavior, exhaustiveness, and ease of writing thanks to the expression and compound form. As a refresher example, a switch expression now can leverage the arrow syntax, such as in this example:

```
var log = switch (event) {
    case PLAY -> "User has triggered the play buttor
    case STOP, PAUSE -> "User needs a break";
```

```
    default -> {
        String message = event.toString();
        LocalDateTime now = LocalDateTime.now();
        yield "Unknown event " + message +
              " logged on " + now;
    }
};
```

Text Blocks

Java 13 introduced text blocks as a preview feature. Text blocks make it easier to work with multiline string literals. This feature is going through a second round of preview with Java 14 and incorporates a couple of tweaks. As a refresher, it is quite common to write code with many string concatenations and escape sequences in order to provide adequate multiline text formatting. The code below shows an example for HTML formatting:

```
String html = "<HTML>" +
"\n\t" + "<BODY>" +
"\n\t\t" + "<H1>\"Java 14 is here!\"</H1>" +
"\n\t" + "</BODY>" +
"\n" + "</HTML>";
```

With text blocks, you can simplify this process and write more-elegant code using the three quotation marks that delimit the beginning and end of a text block:

```
String html = """
<HTML>
  <BODY>
    <H1>"Java 14 is here!"</H1>
  </BODY>
</HTML>""";
```

Text blocks also offer greater expressiveness compared to normal string literals. You can read more about this in an earlier article.

Two new escape sequences were added in Java 14. First, you can use the new `\s` escape sequence to mean a single space. Second, you can use a backslash, `\`, as a way to suppress the insertion of a new line character at the end of a line. This is helpful when you have a very long line that you want to split up for ease of readability inside a text block.

For example, the current way of doing multiline strings is

```
String literal =
         "Lorem ipsum dolor sit amet, consectetur a
         "elit, sed do eiusmod tempor incididunt ut
         "et dolore magna aliqua.";
```

With the `\` escape sequence in text blocks, this can be expressed as follows:

```
String text = """
                Lorem ipsum dolor sit amet, consecte
                elit, sed do eiusmod tempor incidid
                et dolore magna aliqua.\
                """;
```

Pattern Matching for instanceof

Java 14 introduces a preview feature that helps eliminate the need for explicit casts that are preceded by conditional `instanceof` checks. For example, consider the following code:

```
if (obj instanceof Group) {
  Group group = (Group) obj;

  // use group specific methods
  var entries = group.getEntries();
}
```

It can be refactored using the preview feature to this:

```
if (obj instanceof Group group) {
  var entries = group.getEntries();
}
```

Because the condition check asserts that `obj` is of type `Group`, why should you need to say that `obj` is of type `Group` again with the condition block in the first snippet? This need potentially increases the scope for errors.

The shorter syntax will remove many casts from typical Java programs. (A research paper from 2011 proposing a related language feature reported that approximately 24% of all casts follow an `instanceof` in a conditional statement.)

JEP 305 covers this change and points out an example from the *Effective Java* book by Joshua Bloch, which illustrates this with the following equality method:

```
@Override public boolean equals(Object o) {
    return (o instanceof CaseInsensitiveString) &&
            ((CaseInsensitiveString) o).s.equalsIgn
}
```

The previous code can be reduced to the following form by removing the redundant explicit cast to `CaseInsensitiveString`:

```
@Override public boolean equals(Object o) {
    return (o instanceof CaseInsensitiveString cis)
            cis.s.equalsIgnoreCase(s);
}
```

This is an interesting preview feature to experiment with because it opens the door for more general pattern matching. The idea of pattern matching is to provide a language feature with convenient syntax for extracting components of objects based on certain conditions. This is the case with the `instanceof` operator, because the condition is a type check and the extraction is calling an appropriate method or accessing a specific field.

In other words, this preview feature is just the beginning and you can look forward to a language feature that can help further reduce verbosity and thereby reduce the likelihood of bugs.

Records

There's another preview language feature in store: records. Like other ideas floated so far, this feature follows the trend of reducing verbosity in Java and helping developers write code that is more concise. Records focus on certain domain classes whose purpose is only to store data in fields and that do not declare any custom behaviors.

To jump straight to the problem, take a simple domain class, `BankTransaction`, that models a transaction with three fields: a date, an amount, and a description. You need to worry about multiple components when you declare the class:

- The constructor
- Getter methods
- `toString()`
- `hashCode()` and `equals()`

The code for such components is often generated automatically by the IDE and takes a lot of space. Here's a fully generated implementation for the `BankTransaction` class:

```
public class BankTransaction {
    private final LocalDate date;
    private final double amount;
    private final String description;


    public BankTransaction(final LocalDate date,
                           final double amount,
                           final String description
        this.date = date;
        this.amount = amount;
        this.description = description;
    }

    public LocalDate date() {
        return date;
    }

    public double amount() {
        return amount;
    }

    public String description() {
        return description;
    }

    @Override
    public String toString() {
        return "BankTransaction{" +
                "date=" + date +
                ", amount=" + amount +
                ", description='" + description + '
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()
        BankTransaction that = (BankTransaction) o;
        return Double.compare(that.amount, amount) =
                date.equals(that.date) &&
                description.equals(that.description
    }

    @Override
    public int hashCode() {
```

```
        return Objects.hash(date, amount, descripti
    }
}
```

Java 14 provides a way to remove the verbosity and make the intent clear that all you want is a class that only aggregates data together with implementations of the `equals`, `hashCode`, and `toString` methods. You can refactor `BankTransaction` as follows:

```
public record BankTransaction(Date date,
                              double amount,
                              String description) {
```

With a record, you "automatically" get the implementations of `equals`, `hashCode`, and `toString` in addition to the constructor and getters.

To try this example out, remember that you need to compile the file using the preview flag:

```
javac --enable-preview --release 14 BankTransaction
```

Fields of a record are implicitly `final`. This means you cannot reassign them. Note, however, that it doesn't mean the whole record is immutable; the objects themselves that are stored in the fields can be mutable.

If you are interested in a more detailed article about records, check out Ben Evans' recent article in *Java Magazine*.

Stay tuned. Records also raise interesting questions from an educational standpoint for the next generation of Java developers. For example, if you mentor junior developers, when should records be introduced in the curriculum: before introducing OOP and classes or after?

Helpful NullPointerExceptions

Some people say that throwing `NullPointerException`s should be the new "Hello world" in Java because you can't escape from them. Jokes aside, they cause frustrations because they often appear in application logs when code is running in a production environment, which can make debugging difficult because the original code is not readily available. For example, consider the code below:

```
var name = user.getLocation().getCity().getName();
```

Before Java 14, you might get the following error:

```
Exception in thread "main" java.lang.NullPointerExc
    at NullPointerExample.main(NullPointerExample.j
```

Unfortunately, if at line 5, there's an assignment with multiple method invocations— `getLocation()` and `getCity()`—either one could be returning null. In fact, the variable `user` could also be null. So, it's not clear what is causing the `<strong>NullPointerException</strong>`.

Now, with Java 14, there's a new JVM feature through which you can receive more-informative diagnostics:

```
Exception in thread "main" java.lang.NullPointerExc
    at NullPointerExample.main(NullPointerExample.j
```

The message now has two clear components:

- **The consequence:** `Location.getCity()` cannot be invoked.
- **The reason:** The return value of `User.getLocation()` is null.

The enhanced diagnostics work only when you run Java with the following flag:

```
-XX:+ShowCodeDetailsInExceptionMessages
```

Here's an example:

```
java -XX:+ShowCodeDetailsInExceptionMessages NullPo
```

In a future version of Java, this feature might be enabled by default, as reported here.

This enhancement is available not just for method invocations—it also works in other places that could lead to a `NullPointerException`, including field accesses, array accesses, and assignments.

Conclusion

In Java 14, there are new preview language features and updates that help developers in their daily work. For example, Java 14 introduces `instanceof` pattern matching, which is a way to reduce explicit casts. Also, Java 14 introduces records, which are a new construct to concisely declare classes that are used solely to aggregate data. In addition, `NullPointerException` messages have been enhanced with better diagnostics and switch expressions are now part of Java 14. Text blocks, a feature that helps you work with multiline string values, are going through another preview round after introducing two new escape sequences. One other change that will be of interest to a subset of technicians in Java operations is event streaming in the JDK Flight Recorder. That option is discussed in this article by Ben Evans.

As you can see, Java 14 brings a lot of innovation to the table. You should definitely give it a whirl and send feedback on the preview features to the Java team.

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.

Share this Page