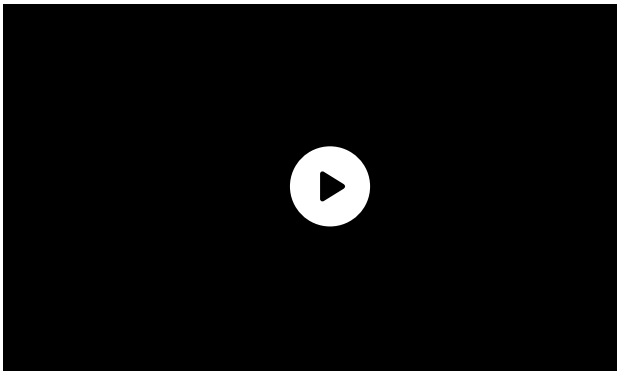
New Features in Java 21

FEATURED VIDEOS



Last updated: April 12, 2024

Written by: Somnath Musib
(https://www.baeldung.com/author/somnathmusib)

Java (https://www.baeldung.com/category/java) +

>= Java 21 (https://www.baeldung.com/tag/jdk21-and-later)

1. Overview

Oracle released Java 21 on September 19, 2023. It's the latest Java LTS release after Java 17.

This article will discuss the **new features and enhancements added to Java 21**.

2. List of JEPs

Let's talk about the notable enhancements introduced to the language in Java 21 through various Java Enhancement Proposals (JEP).

2.1. Record Patterns (JEP 440 (https://openjdk.org/jeps/440))

Record patterns (/java-19-record-patterns) were included in Java 19 and Java 20 as preview features, and Java 21 further refines this feature.

This JEP extends the existing pattern-matching feature to destructure the record class instances which enables writing sophisticated data queries. There is also added support for nested patterns for more composable data queries.

Here's an example:

```
record Point(int x, int y) {}

public static int beforeRecordPattern(Object obj) {
    int sum = 0;
    if(obj instanceof Point p) {
        int x = p.x();
        int y = p.y();
        sum = x+y;
    }
    return sum;
}

public static int afterRecordPattern(Object obj) {
    if(obj instanceof Point(int x, int y)) {
        return x+y;
    }
    return 0;
}
```

We can also use nested record patterns. The following example demonstrates this:

```
enum Color {RED, GREEN, BLUE}
record ColoredPoint(Point point, Color color) {}
```

```
record RandomPoint(ColoredPoint cp) {}
public static Color getRamdomPointColor(RandomPoint r) {
    if(r instanceof RandomPoint(ColoredPoint cp)) {
        return cp.color();
    }
    return null;
}
```

In the above code snippet, we destructure the *ColoredPoint* to access the *color()* method.

2.2. Pattern Matching for *switch* (JEP 441 (https://openjdk.org/jeps/441))

Pattern matching for *switch* (/java-switch-pattern-matching) was introduced in JDK 17 and refined in JDK 18, 19, 20, and JDK 21.

The main goal of this feature is to allow patterns in *switch case* labels, to improve the expressiveness of *switch* statements and expressions. Besides, there is also an enhancement to handle *NullPointerException* by allowing a *null case* label.

Let us explore this with an example:

```
class Account{
    double getBalance() {
        return 0;
    }
}

class SavingsAccount extends Account {
    double getBalance() {
        return 100;
    }
}
class TermAccount extends Account {
    double getBalance() {
        return 1000;
    }
}
class CurrentAccount extends Account {
    double getBalance() {
        return 10000;
    }
}
```

Before Java 21, we can use the below code to get the balance:

```
static double getBalanceWithOutSwitchPattern(Account account) {
    double balance = 0;
    if(account instanceof SavingsAccount sa) {
        balance = sa.getBalance();
    }
    else if(account instanceof TermAccount ta) {
        balance = ta.getBalance();
    }
    else if(account instanceof CurrentAccount ca) {
        balance = ca.getBalance();
    }
    return balance;
}
```

The above code isn't very expressive. With Java 21, we can leverage patterns in *case* labels:

```
static double getBalanceWithSwitchPattern(Account account) {
    double result = 0;
    switch (account) {
        case null -> throw new RuntimeException("Oops, account is
null");
        case SavingsAccount sa -> result = sa.getBalance();
        case TermAccount ta -> result = ta.getBalance();
        case CurrentAccount ca -> result = ca.getBalance();
        default -> result = account.getBalance();
    };
    return result;
}
```

A pattern *case* label also supports many values. Let us elaborate on this with an example:

```
static String processInputOld(String input) {
    String output = null;
    switch(input) {
        case null -> output = "Oops, null";
        case String s -> {
            if("Yes".equalsIgnoreCase(s)) {
                output = "It's Yes";
            }
            else if("No".equalsIgnoreCase(s)) {
                output = "It's No";
            }
            else {
                output = "Try Again";
            }
        }
    }
    return output;
}
```

The above code can be enhanced using Java 21's *when* clauses with *case* labels:

```
static String processInputNew(String input) {
    String output = null;
    switch(input) {
        case null -> output = "Oops, null";
        case String s when "Yes".equalsIgnoreCase(s) -> output = "It's
Yes";
        case String s when "No".equalsIgnoreCase(s) -> output = "It's
No";
        case String s -> output = "Try Again";
    }
    return output;
}
```

2.3. String Literal (JEP 430 (https://openjdk.org/jeps/430))

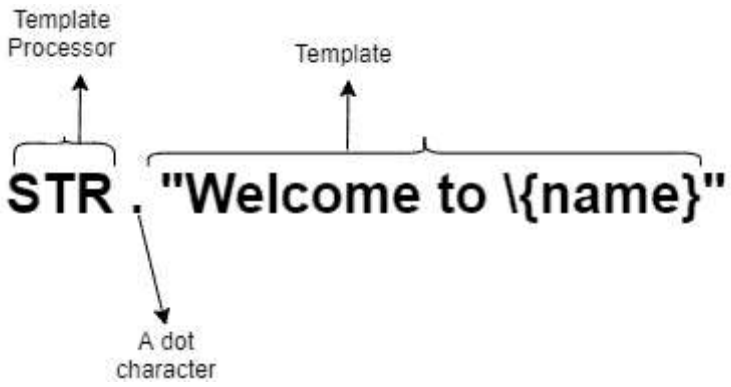
Java offers several mechanisms to compose strings with string literals and expressions. Some of these are String concatenation, *StringBuilder* class, *String* class *format()* method, and the *MessageFormat* class.

Java 21 introduces the string templates. This complements Java's existing string literals and text blocks by coupling literal text with template expressions and template processors to produce the desired results.

Let see an example:

```
String name = "Baeldung";
String welcomeText = STR."Welcome to \{name}";
System.out.println(welcomeText);
```

The above code snippet prints the text "*Welcome to Baeldung*".



(/wp-content/uploads/2024/04/StringTemplates.png)

In the above text, we have a template processor (STR), a dot character, and a template that contains an embedded expression (\{*name*}). At runtime, when the template processor evaluates the template expression, it combines the literal text in the template with the values of the embedded expression to produce the result.

The STR is one of the template processors provided by Java and is automatically imported into all Java source files. The other template processors offered by Java are FMT and RAW.

2.4. Virtual Threads (JEP 444 (https://openjdk.org/jeps/444))

Virtual threads were originally introduced to the Java language as a preview feature in Java 19, and further refined in Java 20. Java 21 introduced some new changes.

Virtual threads are lightweight threads. One of the main purposes of these threads is to reduce the effort of developing high-concurrent applications. The traditional threads also called the platform threads are thin wrappers around OS threads. One of the major issues with platform threads is that they run the code on the OS thread and capture the OS thread throughout its lifetime. There is a limit on the number of OS threads and this creates a scalability bottleneck.

Like platform threads, a virtual thread is also an instance of *java.lang.Thread* class but it isn't tied to a specific OS thread. It runs the code on a specific OS thread but does not capture the thread for an entire lifetime. Therefore, many virtual threads can share OS threads to run their code.

Let us see the use of the virtual thread with an example:

```
try(var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.rangeClosed(1, 10_000).forEach(i -> {
        executor.submit(() -> {
            System.out.println(i);
            try {
                Thread.sleep(Duration.ofSeconds(1));
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
    });
}
```

In the above code snippet, we're using the static *newVirtualThreadPerTaskExecutor()* method. This executor creates a new virtual thread for each task. Therefore, in the above example, we created *10000* virtual threads.

Java 21 introduced two notable changes to the virtual threads:

- Virtual threads now always support thread-local variables.
- Virtual threads are created through the *Thread.Builder* API are also monitored through their lifetime and observable in the new thread dump

2.5. Sequenced Collections (JEP 431 (https://openjdk.org/jeps/431))

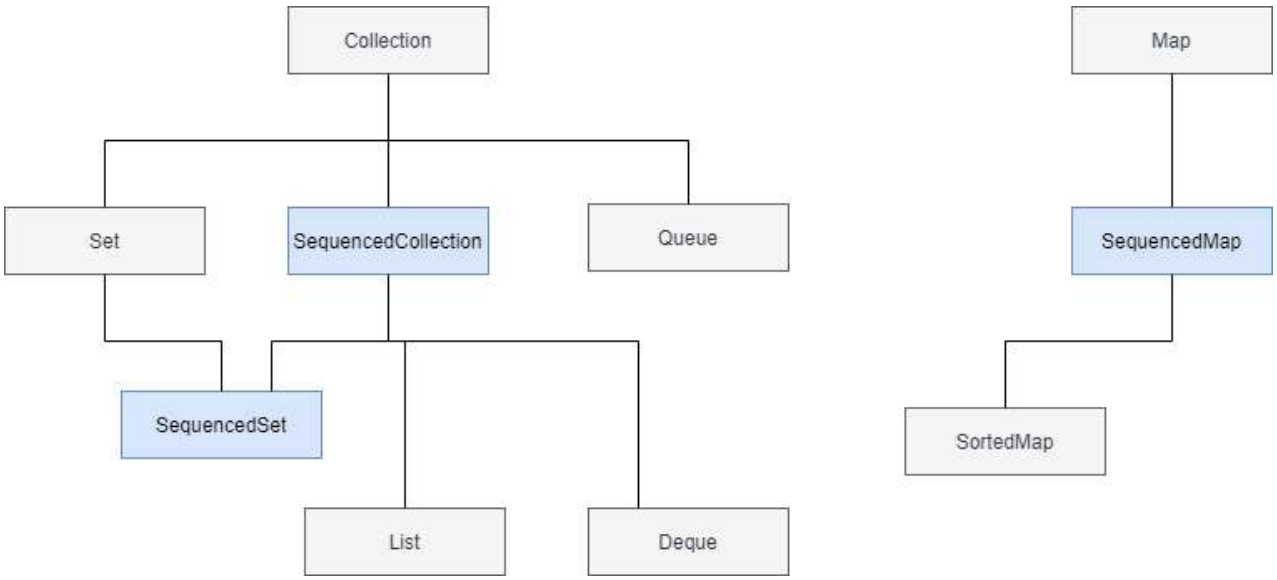
In the Java collection framework, no collection type represents a sequence of elements with a defined encountered order. For instance, *List* and *Deque* interfaces define an encounter order, but their common super type *Collection* doesn't. In the same way, *Set* doesn't define an encounter order, but subtypes such as *LinkedHashSet* or *SortedSet* do.

Java 21 introduced three new interfaces to represent sequenced collections, sequenced sets, and sequenced maps.

A sequenced collection (/java-21-sequenced-collections) is a collection whose elements have a defined encounter order. It has first and last elements, and the elements between them have successors and

predecessors. A sequenced set is a set that is a sequenced collection with no duplicate elements. A sequenced map is a map whose entries have a defined encountered order.

The following diagram shows the retrofitting of the newly introduced interfaces in the collection framework hierarchy:



(/wp-content/uploads/2024/04/SequencedCollections.png)

2.6. Key Encapsulation Mechanism API (JEP 452 (https://openjdk.org/jeps/452))

Key encapsulation is a technique to secure symmetric keys using asymmetric keys or public key cryptography.

The traditional approach uses a public key to secure a randomly generated symmetric key. However, this approach requires padding which is difficult to prove secure.

A key encapsulation mechanism (KEM) uses the public key to derive the symmetric key that doesn't require any padding.

Java 21 has introduced a new KEM API to enable applications to use KEM algorithms.

3. Conclusion

In this article, we discussed a few notable changes delivered in Java 21.

We discussed record patterns, pattern matching for switches, string templates, sequenced collections, virtual threads, string templates, and the new KEM API.

There are other enhancements and improvements spread across JDK 21 packages and classes. However, this article should be a good starting point for exploring Java 21 new features.

As always, the source code for this article is available over on GitHub (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-21).

Get started with Spring Boot and with core Spring, through the *Learn Spring* course:

>> CHECK OUT THE COURSE (/ls-course-end)

Learning to build your API **with Spring**?

Download the E-book (/rest-api-spring-guide)

Comments are open for 30 days after publishing a post. For any issues past this date, use the Contact form on the site.

COURSES

ALL COURSES (/ALL-COURSES)
ALL BULK COURSES (/ALL-BULK-COURSES)
ALL BULK TEAM COURSES (/ALL-BULK-TEAM-COURSES)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
APACHE HTTPCLIENT TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)
JAVA ARRAY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/JAVA-ARRAY)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)

JOBS (/TAG/ACTIVE-JOB/)
OUR PARTNERS (/PARTNERS)
PARTNER WITH BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)