

Overview of Java 20 new features

 [Guide](#)  Java  New Java Features  About 2883 words  About 10 minutes

JDK 20 was released on March 21, 2023 and is not a long-term support version.

According to the development plan, the next LTS version will be JDK 21, which will be released in September 2023.

OpenJDK JDK 20 General-Availability Release

This page provides production-ready open-source builds of the Java Development Kit, version 20, an implementation of the [Java SE 20 Platform](#) under the [GNU General Public License, version 2, with the Classpath Exception](#).

Commercial builds of JDK 20 from Oracle, under a [non-open-source license](#), can be found [here](#).

JDK 20 has only 7 new features:

- [JEP 429: Scoped Values](#) (First Incubation)
- [JEP 432: Record Patterns](#) (Second Preview)
- [JEP 433: Pattern Matching in Switches](#) (Fourth Preview)
- [JEP 434: Foreign Function & Memory API](#) (Second Preview)
- [JEP 436: Virtual Threads](#) (Second Preview)
- [JEP 437: Structured Concurrency](#) (Second Incubation)
- [JEP 432: Vector API](#) (Fifth Incubation)

JEP 429: Scoped Values (First Incubation)

Scoped Values: They can share immutable data within and between threads and are superior to thread-local variables, especially when using a large number of virtual threads.

```
1 final static ScopedValue<...> V = new ScopedValue<>();  
2  
3 // In some method  
4 ScopedValue.where(V, <value>  
5     .run(() -> { ... V.get() ... call methods ... }));  
6  
7  
8
```

java

```
// In a method called directly or indirectly from the lambda  
expression  
... V.get() ...
```

Scoped values allow data to be shared safely and efficiently between components in a larger program without resorting to method parameters.

For a detailed introduction to scope values, we recommend reading [the Scope Values FAQ](#) article.

JEP 432: Record Mode (Second Preview)

Record Patterns allow you to deconstruct record values, making it easier to extract data from record classes. Furthermore, you can nest record patterns and combine them with type patterns to achieve powerful, declarative, and composable forms of data navigation and processing.

Record patterns cannot be used alone, but must be used with instanceof or switch pattern matching.

Let's take instanceof as an example to demonstrate it briefly.

Simply define a record class:

```
1 record Shape(String type, long unit){} java
```

Before recording mode:

```
1 Shape circle = new Shape("Circle", 10); java  
2 if (circle instanceof Shape shape) {  
3     System.out.println("Area of " + shape.type() + " is : " + Math.PI  
4     * Math.pow(shape.unit(), 2));  
5 }
```

With the record mode:



```
1 Shape circle = new Shape("Circle", 10);
2 if (circle instanceof Shape(String type, long unit)) {
3     System.out.println("Area of " + type + " is : " + Math.PI *
4     Math.pow(unit, 2));
5 }
```

java

Let's take a look at the use of recording mode and switch.

Define some classes:

```
1 interface Shape {}
2 record Circle(double radius) implements Shape { }
3 record Square(double side) implements Shape { }
4 record Rectangle(double length, double width) implements Shape { }
```

java

Before recording mode:

```
1 Shape shape = new Circle(10);
2 switch (shape) {
3     case Circle c:
4         System.out.println("The shape is Circle with area: " +
5         Math.PI * c.radius() * c.radius());
6         break;
7
8     case Square s:
9         System.out.println("The shape is Square with area: " +
10        s.side() * s.side());
11        break;
12
13    case Rectangle r:
14        System.out.println("The shape is Rectangle with area: + " +
15        r.length() * r.width());
16        break;
17
18    default:
19        System.out.println("Unknown Shape");
20        break;
21 }
```

java

With the record mode:



```
1 Shape shape = new Circle(10);
2 switch(shape) {
3
4     case Circle(double radius):
5         System.out.println("The shape is Circle with area: " + Math.PI
6 * radius * radius);
7         break;
8
9     case Square(double side):
10        System.out.println("The shape is Square with area: " + side *
11 side);
12        break;
13
14    case Rectangle(double length, double width):
15        System.out.println("The shape is Rectangle with area: + " +
16 length * width);
17        break;
18
19    default:
20        System.out.println("Unknown Shape");
21        break;
22 }
```

Record mode can avoid unnecessary conversions, making the code more concise and easier to read. Moreover, after using record mode, you no longer have to worry about `null` or `NullPointerException`, and the code is safer and more reliable.

Record mode was first previewed in Java 19, proposed by [JEP 405](#). It was previewed again in JDK 20, proposed by [JEP 432](#). This preview includes:

- Add support for generic record pattern type parameter inference,
- Add support for record mode to appear in the title of the enhancement statement for
- Remove support for named record mode.

Note : Do not confuse the record mode with the record class officially introduced in [JDK16](#)



JEP 433: Pattern Matching in Switches (Fourth Preview)

Just `instanceof` like, switch the type matching automatic conversion function was also added.


`instanceof` Code example:

```
1 // Old code
2 if (o instanceof String) {
3     String s = (String)o;
4     ... use s ...
5 }
6
7 // New code
8 if (o instanceof String s) {
9     ... use s ...
10 }
```

java

switch Code example:

```
1 // Old code
2 static String formatter(Object o) {
3     String formatted = "unknown";
4     if (o instanceof Integer i) {
5         formatted = String.format("int %d", i);
6     } else if (o instanceof Long l) {
7         formatted = String.format("long %d", l);
8     } else if (o instanceof Double d) {
9         formatted = String.format("double %f", d);
10    } else if (o instanceof String s) {
11        formatted = String.format("String %s", s);
12    }
13    return formatted;
14 }
15
16 // New code
17 static String formatterPatternSwitch(Object o) {
    return switch (o) {
```

java

```
18         case Integer i -> String.format("int %d", i);
19         case Long l    -> String.format("long %d", l);
20         case Double d   -> String.format("double %f", d);
21         case String s   -> String.format("String %s", s);
22         default         -> o.toString();
23     };
24 }
25
```

switch Pattern matching was previewed in Java 17, Java 18, and Java 19, with Java 20 being the fourth preview. Each preview generally includes minor improvements, which I won't detail here.

JEP 434: Foreign Function and Memory API (Second Preview)

This API enables Java programs to interoperate with code and data outside the Java runtime. By efficiently calling external functions (that is, code outside the JVM) and safely accessing external memory (that is, memory not managed by the JVM), this API enables Java programs to call native libraries and process native data without the risks and brittleness of JNI.

The foreign function and memory API had its first incubation phase in Java 17, with [JEP 412](#). It had its second incubation phase in Java 18, with [JEP 419](#). It was first previewed in Java 19, with [JEP 424](#).

JDK 20 is the second preview, proposed by [JEP 434](#). This time the improvements include:

- `MemorySegment` and `MemoryAddress` abstract unity
- Enhanced `MemoryLayout` Hierarchy
- `MemorySession` Split into `Arena` and `SegmentScope` to facilitate segment sharing across maintenance boundaries.

In [the Java 19 New Features Overview](#), I introduced the external function and memory API in detail, so I will not give additional introductions here.

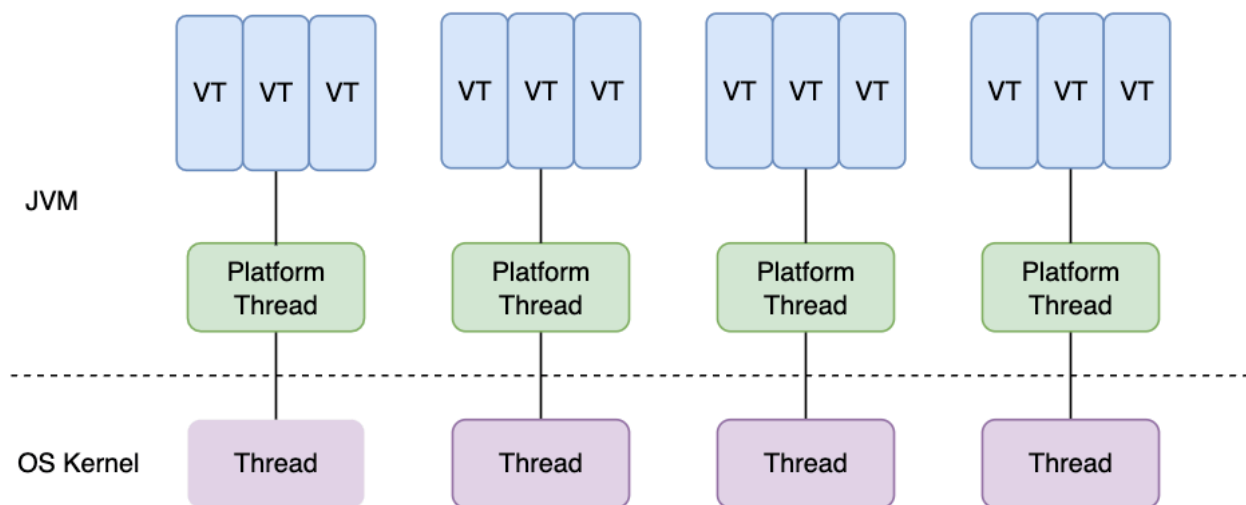


JEP 436: Virtual Threads (Second Preview)

A virtual thread is a lightweight process (LWP) implemented by the JDK rather than the OS and scheduled by the JVM. Many virtual threads share the same OS thread, and the number of virtual threads can be much greater than the number of OS threads.

Before the introduction of virtual threads, `java.lang.Thread` the package already supported platform threads, which are the threads we've been using since the days of virtual threads. The JVM scheduler manages virtual threads through platform threads (carrier threads). A platform thread can execute different virtual threads at different times (multiple virtual threads are attached to a platform thread). When a virtual thread is blocked or waiting, the platform thread can switch to executing another virtual thread.

The relationship diagram of virtual threads, platform threads, and system kernel threads is shown below (Source: [How to Use Java 19 Virtual Threads](#)):



A few more points about the correspondence between platform threads and kernel threads: In mainstream operating systems like Windows and Linux, Java threads use a one-to-one thread model, meaning one platform thread corresponds to one kernel thread. Solaris is a special case, where the HotSpot VM supports both many-to-many and one-to-one threading. For more details, please refer to R's answer: [Is the thread model in the JVM user-level](#) ?

Compared to platform threads, virtual threads are inexpensive and lightweight, destroyed immediately after use. Therefore, they do not need to be reused or pooled. Each task can run in its own dedicated virtual thread. Virtual threads can be paused and resumed to



switch between threads, avoiding the overhead of context switching while maintaining the advantages of multithreading. This simplifies the complexity of high-concurrency programs and effectively reduces the workload of writing, maintaining, and monitoring high-throughput concurrent applications.

Virtual threads have proven to be very useful in other multithreaded languages, such as Goroutine in Go and processes in Erlang.

There is a discussion on Java 19 virtual threads on Zhihu. If you are interested, you can check it out: <https://www.zhihu.com/question/536743167> .

For a detailed explanation and principle of Java virtual threads, please refer to the following articles:

- [A Simple Introduction to Virtual Threads](#)
- [Java 19 is now GA! See how virtual threads significantly improve system throughput](#)
- [Virtual Thread - VirtualThread Source Code Perspective](#)

Virtual threads were first previewed in Java 19, proposed by [JEP 425](#). [JDK 20](#) saw a second preview with some minor changes that I won't detail here.

Finally, let's look at four ways to create virtual threads:

```
1 // 1、通过 Thread.ofVirtual() 创建
2 Runnable fn = () -> {
3     // your code here
4 };
5
6 Thread thread = Thread.ofVirtual(fn)
7                     .start();
8
9 // 2、通过 Thread.startVirtualThread() 、创建
10 Thread thread = Thread.startVirtualThread(() -> {
11     // your code here
12 });
13
14 // 3、通过 Executors.newVirtualThreadPerTaskExecutor() 创建
15 var executorService = Executors.newVirtualThreadPerTaskExecutor();
16
17 executorService.submit(() -> {
18     // your code here
19 });
20
```

java


```
21 class CustomThread implements Runnable {
22     @Override
23     public void run() {
24         System.out.println("CustomThread run");
25     }
26 }
27
28 //4、通过 ThreadFactory 创建
29 CustomThread customThread = new CustomThread();
30 // 获取线程工厂类
31 ThreadFactory factory = Thread.ofVirtual().factory();
32 // 创建虚拟线程
33 Thread thread = factory.newThread(customThread);
34 // 启动线程
35 thread.start();
```

From the four ways of creating virtual threads listed above, it can be seen that in order to lower the threshold of virtual threads, the official has tried its best to reuse the original `Thread` thread class, so that a smooth transition to the use of virtual threads can be achieved.

JEP 437: Structured Concurrency (Second Incubation)

Java 19 introduces structured concurrency, a multi-threaded programming method. Its purpose is to simplify multi-threaded programming through structured concurrency APIs, not to replace them `java.util.concurrent`. It is currently in the incubator stage.

Structured concurrency treats multiple tasks running in different threads as a single unit of work, simplifying error handling, improving reliability, and enhancing observability. In other words, structured concurrency preserves the readability, maintainability, and observability of single-threaded code.

The basic API of structured concurrency is [StructuredTaskScope](#) to

`StructuredTaskScope` support splitting a task into multiple concurrent subtasks, which are executed in their own threads, and the subtasks must complete before the main task continues.

`StructuredTaskScope` The basic usage is as follows:



```
1      try (var scope = new StructuredTaskScope<Object>()) {      java
2          // 使用fork方法派生线程来执行子任务
3          Future<Integer> future1 = scope.fork(task1);
4          Future<String> future2 = scope.fork(task2);
5          // 等待线程完成
6          scope.join();
7          // 结果的处理可能包括处理或重新抛出异常
8          ... process results/exceptions ...
9      } // close
```

Structured concurrency is well suited for virtual threads, which are lightweight threads implemented by the JDK. Many virtual threads share the same operating system thread, allowing for a very large number of virtual threads.

The only change to structured concurrency in JDK 20 is an update to support thread-inherited scope values created within a task scope. `StructuredTaskScope` This simplifies sharing immutable data across threads, as detailed in [JEP 429](#) .

JEP 432: Vector API (Fifth Incubation)

Vector computations consist of a series of operations on vectors. The Vector API is used to express vector computations that can be reliably compiled into optimal vector instructions on the supported CPU architecture at runtime, achieving performance superior to equivalent scalar computations.

The goal of the Vector API is to provide users with a wide range of vector computations that are concise, easy to use, and platform-independent.

The Vector API was originally proposed by [JEP 338](#) and integrated into Java 16 as [an incubating API](#). [The second round of incubation was proposed by JEP 414](#) and integrated into Java 17, the third round was proposed by [JEP 417](#) and integrated into Java 18, and the fourth round was proposed by [JEP 426](#) and integrated into Java 19.

This incubation of Java20 basically did not change the vector API, but only made some bug fixes and performance enhancements, see [JEP 438](#) for details .



JavaGuide官方公众号

(微信搜索JavaGuide)



- 1、公众号后台回复“**PDF**”获取原创PDF面试手册
- 2、公众号后台回复“**学习路线**”获取Java学习路线最新版
- 3、公众号后台回复“**开源**”获取优质Java开源项目合集
- 4、公众号后台回复“**八股文**”获取Java面试真题+面经

Recently Updated 2023/10/30 12:32
Contributors: Guide , Mr.Hope

Copyright © 2025 Guide

