Search...

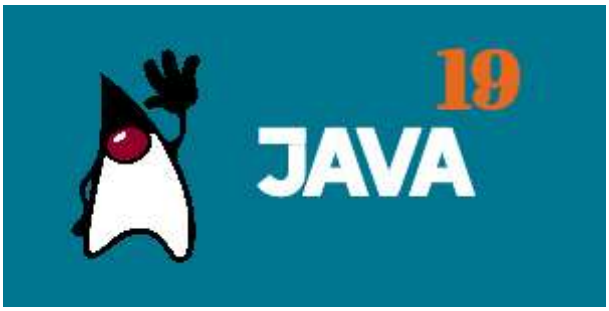
What is new in Java 19

By mkyong | Last updated: November 28, 2022

Viewed: 6,427 (+612 pv/w)

Tags: Concurrency | java 19 | new jdk | record | switch | virtual threads

Next >



Java 19 reached general availability on 20 September 2022, download Java 19 here.

Java 19 has 7 JEP items.

Java 19 developer features.

Record Patterns (preview), Foreign Function & Memory API (Preview), Virtual Threads (Preview), Pattern Matching for switch (Third Preview), Structure Concurrency APIs (Incubator)

1. JEP 405: Record Patterns (Preview)

This JEP improves the way of deconstructing or extracting the record values.

1.1 Normal Record Patterns Example

A typical record class JEP 395, and we need to deconstruct, get, or extract the record values manually.

JEP405.java

```
package com.mkyong.java19.jep405;

public class JEP405 {

  record Point(int x, int y) {
  }

  static void printSum(Object o) {
      if (o instanceof Point p) {
          int x = p.x();  // get x()
          int y = p.y();  // get y()
          System.out.println(x + y);
      }
  }

  public static void main(String[] args) {
      printSum(new Point(10, 20)); // outpu
  }
}
```


We now use the new record pattern to deconstruct (get or extract) the record values automati-

JEP405.java

```
package com.mkyong.java19.jep405;

public class JEP405 {

  record Point(int x, int y) {
  }

  static void printSumNew(Object o) {
      if (o instanceof Point(int x,int y)) {  // record pattern
          System.out.println(x + y);
      }
  }

  public static void main(String[] args) {
      printSumNew(new Point(10, 20)); // output 30
  }
}
```

1.2 Record Nested Patterns Example

JEP405_1.java

```
package com.mkyong.java19.jep405;

public class JEP405_1 {

  record Point(int x, int y) {
  }

  record Total(Point p1, Point p2) {
  }

  static void printSum(Object o) {
      // record nested pattern
      if (o instanceof Total(Point(int x,int
```

```
    public static void main(String[] args) {

        printSum(new Total(new Point(10, 5), new Point(2, 3)));

    }
}
```

Further Reading

- Java 16, JEP 394: Pattern Matching for instanceof
- Java 19, JEP 405: Record Patterns (Preview)

2. JEP 422: Linux/RISC-V Port

This JEP makes Java support RISC-V hardware, and the port of the JDK will integrate into the JDK mainline repository.

Further Reading

- JEP 422: Linux/RISC-V Port

3. JEP 424: Foreign Function

This JEP promotes the Foreign Function & Memory
preview stage. The Foreign Function & Memory AP
package of the `java.base` module.

```
int radixsort(const unsigned char **base, int nmemb,
  const unsigned        char *table, unsigned endbyte);
```

JEP424_SORT.java

```
                stdlib.lookup("radixsort").orElseThrow(),
                FunctionDescriptor.ofVoid(ADDRESS, JAVA_INT, ADDRESS, JAVA_CHAR));

        // 5. Sort the off-heap data by calling the foreign function
        radixSort.invoke(offHeap, javaStrings.length, MemoryAddress.NULL, '\0');

        // 6. Copy the (reordered) strings from off-heap to on-heap
        for (int i = 0; i < javaStrings.length; i++) {
            MemoryAddress cStringPtr = offHeap.getAtIndex(ValueLayout.ADDRESS, i);
            javaStrings[i] = cStringPtr.getUtf8String(0);
        }

        //print sort result
        for (String javaString : javaStrings) {
            System.out.println(javaString);
        }

    }

}
```

Output

```
Terminal

WARNING: A restricted method in java.lang.foreign.Linker has been called
WARNING: java.lang.foreign.Linker::nativeLinker has been called by the unnamed module
WARNING: Use --enable-native-access=ALL-UNNAMED to avoid a warning for this module

a
b
c
d
z
```


3.2 Below example shows how to use the Foreign Function & Memory API (FFM API) to call the

```
size_t strlen(const char *s);
```

JEP424_STRLEN.java

```
package com.mkyong.java19.jep424;

import java.lang.foreign.*;
import java.lang.invoke.MethodHandle;

import static java.lang.foreign.ValueLayout.ADDRESS;
import static java.lang.foreign.ValueLayout.JAVA_LONG;

public class JEP424_STRLEN {

  public static void main(String[] args) throws Throwable {

      String input = "Hello World";

      // 1. Find foreign function on the C library path
      SymbolLookup stdlib = Linker.nativeLinker().defaultLookup();

      // 2. Get a handle to the "strlen" function in the C standard library
      MethodHandle methodHandle = Linker.nativeLinker().downcallHandle(
              stdlib.lookup("strlen").orElseThrow(),
              FunctionDescriptor.of(JAVA_LONG, ADDRESS));

      // 3. Allocate off-heap memory to store strings
      MemorySegment memorySegment = SegmentAllocator
              .implicitAllocator().allocate

      // 4. Runs the foreign function "strl
      long length = (long) methodHandle.inv

      System.out.println("length = " + leng

  }

}
```

```
Terminal

WARNING: A restricted method in java.lang.foreign.Linker has been called
WARNING: java.lang.foreign.Linker::nativeLinker has been called by the unnamed module
WARNING: Use --enable-native-access=ALL-UNNAMED to avoid a warning for this module

length = 11
```

Further Reading

- JEP 424: Foreign Function & Memory API (Preview)

4. JEP 425: Virtual Threads (Preview)

This JEP introduces virtual threads, a lightweight implementation of threads provided by the JDK instead of the OS. **The number of virtual threads can be much larger than the number of OS threads**. These virtual threads help increase the throughput of the concurrent applications.

Let's review a case study:

An application with an average latency of 100ms ru
threads, and processing 20 requests concurrently, v


```
Terminal

This application can achieve a throughput of 200 requests per second
```

Let's say we scale the **throughput to 400 requests per second.**

We either need to process 40 requests concurrently (upgrade CPU processor to support 40 OS threads) or reduce the average latency of the application to 50ms; The limit is always the OS threads factor or CPU processor, which makes the application's throughput hardly scale up.

Platform Threads, OS Threads, and Virtual Threads

In Java, every instance of `java.lang.Thread` is a platform thread that runs Java code on an underlying OS thread. The number of platform threads is limited to the number of OS threads, like in the above case study.

A Virtual Thread is also an instance of `java.lang.Thread`, but it runs Java code on the same OS thread and shares it effectively, which makes the number of virtual threads can be much larger than the number of OS threads.

Because the number of OS threads does not limit virtual threads, we can quickly increase the concurrent requests to achieve higher throughput.

For example, the same existing CPU contains 10 cores and 20 OS threads, and we can convert the platform threads to virtual threads and increase concurrent requests to 40 to achieve a **throughput of 400 requests per second.**

```
Terminal
```

```
    This application can achieve a throughput of 400 requests per second.
```

The below example will run `10k` tasks on Virtual Threads, and the modern CPU may take less than 1 second to finish it.

```
// finish within 1 second
try (var executor = Executors.newVirtualT
    IntStream.range(0, 10_000).forEach(i
        executor.submit(() -> {
            Thread.sleep(Duration.ofSecon
            return i;
        });
    });
```

Try running the same code using the classic `newFixedThreadPool`, and we may need to manually terminal it because it will take a long time to finish.

```
// 10_000/20 = 500seconds, it takes 8 minutes and 33 seconds to finish it
try (var executor = Executors.newFixedThreadPool(20)) {
    IntStream.range(0, 10_000).forEach(i -> {
        executor.submit(() -> {
            Thread.sleep(Duration.ofSeconds(1));
            return i;
        });
    });
}
```

Further Reading

- JEP 425: Virtual Threads (Preview)

5. JEP 426: Vector API (Fourth Incubator)

This JEP improves the Vector API performance and other enhancements in response to feedback.

History

- Java 16, JEP 338 introduced new Vector API as an incubating API.
- Java 17, JEP 414 enhanced the Vector APIs, second incubator.
- Java 18, JEP 417 enhanced the Vector APIs, third incubator.

Further Reading

- JEP 426: Vector API (Fourth Incubator)

6. JEP 427: Pattern Matching for switch (Third Preview)

Guarded patterns are replaced with when clauses in switch blocks.

Below is a Java pattern matching for `switch` using the new `when` as the guarded pattern.

P.S The old `&&` was replaced with `when` in the guarded pattern.

JEP427.java

```
package com.mkyong.java19.jep427;

public class JEP427 {

  public static void main(String[] args) {

      testJava19("mkyong");
      testJava19("mkyongmkyong");
  }

  /* Old guarded pattern using &&
  static void test(Object o) {
      switch (o) {
          case String s && s.length() > 6 ->
              System.out.println("String's length longer than 10!");
          case String s ->
              System.out.println("String's length is " + s.length());
          default -> {
          }
      }
  }*/

  // new guarded pattern with when
  static void testJava19(Object o) {
      switch (o) {
          case String s
                  when s.length() > 10 ->
                    System.out.println("Str
          case String s ->
                    System.out.println("Str
          default -> {}
      }
  }
```

Output

```
Terminal
```

```
String's length is 6
String's length longer than 10!
```

This JEP also improves the runtime semantics of a pattern switch when the value of the selector expression is null are more closely aligned with legacy switch semantics.

History

- Java 17 JEP 406 introduced Pattern Matching for switch (Preview).
- Java 18 JEP 420 introduced Pattern Matching for switch (Second Preview).
- Java 19 JEP 427 introduced Pattern Matching for switch (Third Preview).

> **Further Reading**
>
> - JEP 427: Pattern Matching for switch (Third Preview)

7. JEP 428: Structured Concu

This JEP introduces Structured Concurrency APIs to

UnStructured Concurrency API
Review the below multithreaded code.

```
import java.time.Duration;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class JEP428 {

  public static void main(String[] args) throws ExecutionException, InterruptedExceptio
      JEP428 obj = new JEP428();
      obj.handleUnStructureAPI();
  }

  Response handleUnStructureAPI() throws ExecutionException, InterruptedException {
      try (var executor = Executors.newFixedThreadPool(10)) {
          Future<String> user = executor.submit(this::findUser);
          Future<Integer> order = executor.submit(this::fetchOrder);
          String theUser = user.get();   // Join findUser
          int theOrder = order.get();  // Join fetchOrder
          return new Response(theUser, theOrder);
      }
  }

  private String findUser() throws InterruptedException {
      Thread.sleep(Duration.ofSeconds(1));
      return "mkyong";
  }

  private Integer fetchOrder() throws InterruptedException {
      Thread.sleep(Duration.ofSeconds(1));
      return 1;
  }

  record Response(String x, int y) {
  }
}
```


The `Future` tasks `findUser()` and `fetchOrder()` exe
fail (throw exception) independently.

- If `fetchOrder()` throws an exception, the `findUser()` will continue running it, wasting resources.
- Assume `findUser()` takes 1 minute to finish, and the `fetchOrder()` is failed immediately, but we have no ways to tell `handle()` to stop or cancel the entire `handle()` process, the `handle()` will still wait 1 minute to process it.

Structured Concurrency API

This JEP introduces Structured Concurrency API `StructuredTaskScope`, which treats multiple tasks running in different threads as a single unit of work.

```
import jdk.incubator.concurrent.StructuredTaskScope;

  //...

  Response handleStructureAPI() throws ExecutionException, InterruptedException {
      try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
          Future<String> user = scope.fork(this::findUser);
          Future<Integer> order = scope.fork(this::fetchOrder);

          scope.join();           // Join both forks
          scope.throwIfFailed();  // ... and propagate errors

          // Here, both forks have succeeded, so compose their results
          return new Response(user.resultNow(), order.resultNow());
      }
  }
```

The `StructuredTaskScope.ShutdownOnFailure()` means if either the `findUser()` or `fetchOrder()` fails, the other will cancel if it has not yet been completed.

The `StructuredTaskScope` is in `incubator` module, a
the below commands:

```
Terminal

  javac --release XX --enable-preview --add
  java --enable-preview --add-modules jdk.incubator.concurrent Main
```