Java 19 Features

**Date 16/05/2023**

# Overview of Java 19 (Sept 20, 2022)

1. New System Properties for System.out and System.err
2. New Methods to Create Pre-allocated HashMaps
3. JEP 427 - Pattern Matching for switch (Third Preview)
4. JEP 405 - Record Patterns (Preview)
5. JEP 425 - Virtual Threads (Preview)
6. JDK 428 - Structured Concurrency (Incubator)
7. JDK 424 - Foreign Function & Memory API (Preview)
8. Locale class New Static Method

# New System Properties for System.out and System.err

One change that you won't find in the Java 19 feature announcements but only kept deep in the release notes is something **every Java developer should know about**.

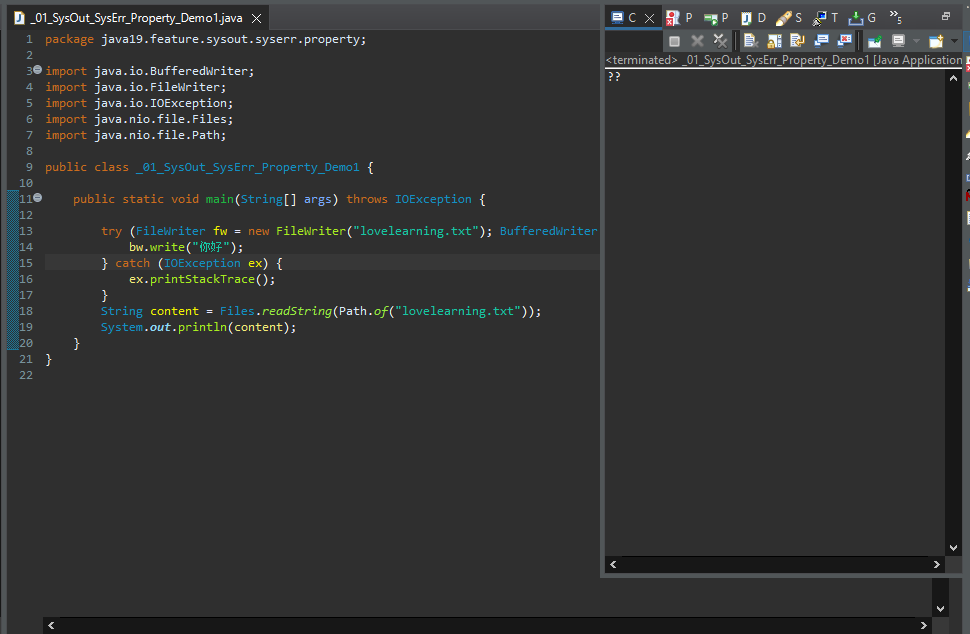
If you run an existing application with Java 19, you may see **question marks** on the console instead of **special characters**.

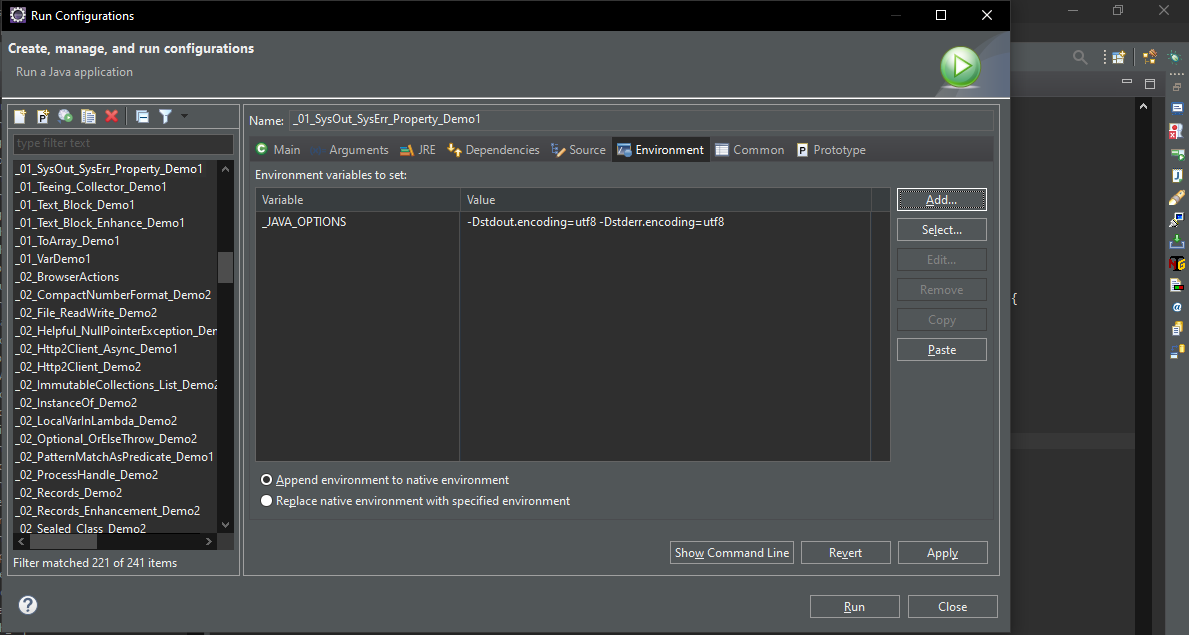
This is because, as of Java 19, the operating system's default encoding is used for printing to System.out and System.err - for example, "Cp1252" on Windows. To change the output to UTF-8, you have to add the following VM options when calling the application,

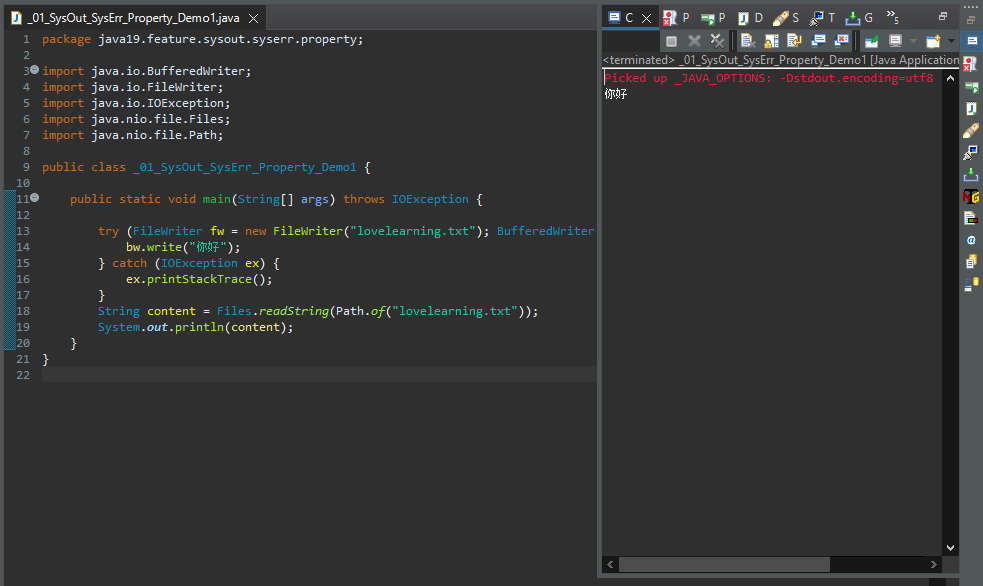
**-Dstdout.encoding=utf8 -Dstderr.encoding=utf8**

If you don't want to do this every time you start the program, you can also set these settings globally by defining the following environment variable (yes, it begins with an underscore),

**\_JAVA\_OPTIONS="-Dstdout.encoding=utf8 -Dstderr.encoding=utf8"**







# New Methods to Create Pre-allocated HashMaps

If we want to create an ArrayList for a known number of elements (e.g., 120), we can do it as follows,

**List<String> list = new ArrayList<>(120);**

Thus, the array underlying the ArrayList is allocated directly for 120 elements and does not have to be enlarged several times (i.e., newly created and copied) to insert the 120 elements.

Similarly, we have always been able to generate a HashMap as follows,

**Map<String, Integer> map = new HashMap<>(120);**

Intuitively, one would think that this HashMap offers space for 120 mappings.

However, this is not the case!

This is because the HashMap is initialized with a default load factor of 0.75. This means that as soon as the HashMap is 75% full, it is rebuilt ("rehashed") with double the size. This ensures that the elements are distributed as evenly as possible across the HashMap's buckets and that as few buckets as possible contain more than one element.

Thus, the HashMap initialized with a capacity of 120 can only hold 120 × 0.75 = 90 mappings.

To create a HashMap for 120 mappings, you had to calculate the capacity by dividing the number of mappings by the load factor, 120 ÷ 0.75 = 160.

So, a HashMap for 120 mappings had to be created as follows,

For 120 mappings, 120 / 0.75 = 160

Map<String, Integer> map = new HashMap<>(160);

Java 19 makes it easier for us - we can now write the following instead,

**Map<String, Integer> map = HashMap.newHashMap(120);**

public class \_01\_HashMap\_Create\_Method\_Demo1 {

*@SuppressWarnings*("unused")

public static void main(String[] args) {

// Before Java 19

Map<String, Integer> map = new HashMap<>(120);

// From Java 19

Map<Integer, String> newMap = HashMap.*newHashMap*(120);

}

}

# JEP 427 - Pattern Matching for switch (Third Preview)

In Java 19, JDK Enhancement Proposal 427 changed the syntax of the so-called "Guarded Pattern" (in the example below "String s && s.length() > 5"). Instead of &&, we now have to use the new keyword when.

Example,

switch (obj) {

case String s when s.length() > 5 -> System.out.println(s.toUpperCase());

case String s -> System.out.println(s.toLowerCase());

case Integer i -> System.out.println(i \* i);

default -> {}

}

**when** is a so-called "contextual keyword" and therefore only has a meaning within a case label. If you have variables or methods with the name "when" in your code, you don't need to change them.

public class \_01\_Switch\_Exp\_Enhancement\_Demo1 {

public static void main(String[] args) {

*testSwitchExp*("Java19SwitchExpEnhacement");

*testSwitchExp*("Java19");

}

*@SuppressWarnings*("preview")

private static void testSwitchExp(Object obj) {

switch (obj) {

case String str when str.length() > 10 -> System.***out***.println("String length greater than 10");

case String str -> System.***out***.println("String length " + str.length());

case default -> {

}

}

}

}

# JEP 405 - Record Patterns (Preview)

As of Java 19, JDK Enhancement Proposal 405 allows us to use a so-called "record pattern". This allows us to write the code as follows:

**private void print(Object object) {**

**if (object instanceof Position(int x, int y)) {**

**System.out.println("object is a position, x = " + x + ", y = " + y);**

**}**

**// else ...**

**}**

Instead of matching on "Position pstn" and accessing pstn in the above code, we now match on "Position(int x, int y)" and can then access x and y directly.

public class \_01\_Records\_PatternMatch\_Demo1 {

record Car(String name, String model) {

}

public static void main(String[] args) {

*printCarDetail*(new Car("Audi", "A6100"));

}

// Before Java19

// @SuppressWarnings("preview")

// private static void printCarDetail(Object obj) {

// if (obj instanceof Car car) {

// System.out.println(car.name + " " + car.model);

// }

// }

*@SuppressWarnings*("preview")

private static void printCarDetail(Object obj) {

if (obj instanceof Car(String name,String model)) {

System.***out***.println(name + " " + model);

}

}

}

We can now also use a record pattern in the switch statement.

public class \_02\_Records\_PatternMatch\_Switch\_Demo2 {

record Car(String name, String model) {

}

public static void main(String[] args) {

*printCarDetail*(new Car("Audi", "A6100"));

}

*@SuppressWarnings*("preview")

private static void printCarDetail(Object obj) {

switch (obj) {

case Car(String name, String model) ->

System.***out***.println("Object is Car, Name = " + name + " and Model = " + model);

case default -> {

}

}

}

}

It is also possible to match nested records, let me demonstrate this with below example.

public class \_03\_Records\_Nested\_PatternMatch\_Demo3 {

record JourneyStartDay(int sDay) {

}

record JourneyEndDay(int eDay) {

}

record JourneyTotalDays(JourneyStartDay jsd, JourneyEndDay jed) {

}

public static void main(String[] args) {

*calculateTotalJourneyDays*(new JourneyTotalDays(new JourneyStartDay(1), new JourneyEndDay(10)));

}

*@SuppressWarnings*("preview")

private static void calculateTotalJourneyDays(Object obj) {

if (obj instanceof JourneyTotalDays(JourneyStartDay(int sDay),JourneyEndDay(int eDay))) {

System.***out***.println("Total number of journey days is " + (eDay - sDay));

}

}

}

# JEP 425 - Virtual Threads (Preview)

The most exciting innovation **in Java 19 for me is "Virtual Threads".** Virtual threads have been developed in **Project Loom** for several years and could only be tested with a self-compiled JDK so far.

With **JDK Enhancement Proposal 425**, virtual threads finally make their way into the official JDK, and they do so directly in the preview stage, so no more significant changes to the API are expected.

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 runs on a CPU containing 10 cores, 20 OS threads, and processing 20 requests concurrently, which will fully utilize the 20 OS threads.

**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.newVirtualThreadPerTaskExecutor()) {

IntStream.range(0, 10\_000).forEach(i -> {

executor.submit(() -> {

Thread.sleep(Duration.ofSeconds(1));

return i;

});

});

}

Try running the same code using the classic newFixedThreadPool, and we may need to manually terminate 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;

});

});

}

public class \_01\_VirtualThread\_Demo1 {

public static void main(String[] args) {

*virtualThreadDemo*();

*nonVirtualThreadDemo*();

}

// The below example will run 5k tasks on Virtual Threads, and the modern CPU

// may take less than 1 second to finish it.

private static void virtualThreadDemo() {

try (var executor = Executors.*newVirtualThreadPerTaskExecutor*()) {

IntStream.*range*(0, 5000).forEach(n -> {

executor.submit(() -> {

try {

Thread.*sleep*(Duration.*ofSeconds*(1));

} catch (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println(n);

});

});

}

}

// Try running the above code using the classic newFixedThreadPool, and we may

// need to manually terminate it because it will take a long time to finish.

private static void nonVirtualThreadDemo() {

try (var executor = Executors.*newFixedThreadPool*(20)) {

IntStream.*range*(0, 5000).forEach(n -> {

executor.submit(() -> {

try {

Thread.*sleep*(Duration.*ofSeconds*(1));

} catch (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println(n);

});

});

}

}

}

# JDK 428 - Structured Concurrency (Incubator)

Structured Concurrency developed in Project Loom and initially released as an incubator feature in Java 19 with **JDK Enhancement Proposal 428** is the so-called "Structured Concurrency."

When a task consists of several subtasks that can be processed in parallel, Structured Concurrency allows us to implement this in a particularly readable and maintainable way.

public class \_01\_Structured\_Concurrency\_Demo1 {

public static void main(String[] args) throws InterruptedException, ExecutionException {

\_01\_Structured\_Concurrency\_Demo1 obj = new \_01\_Structured\_Concurrency\_Demo1();

ResultSet result1 = obj.handleUnstructuredAPI();

System.***out***.println(result1.x);

System.***out***.println(result1.y);

ResultSet result2 = obj.handleStructuredAPI();

System.***out***.println(result2.x);

System.***out***.println(result2.y);

}

record ResultSet(String x, int y) {

}

private String findUser() throws InterruptedException {

Thread.*sleep*(Duration.*ofSeconds*(1));

return "John";

}

private Integer fetchOrder() throws InterruptedException {

Thread.*sleep*(Duration.*ofSeconds*(1));

return 1;

}

/\*

\* The Future tasks findUser() and fetchOrder() execute concurrency, and each

\* task can succeed or fail (throw exception) independently.

\*

\* If findUser() throws an exception, the fetchOrder() will continue running it,

\* wasting resources.

\*

\* 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.

\*/

ResultSet handleUnstructuredAPI() throws InterruptedException, ExecutionException {

try (var executor = Executors.*newFixedThreadPool*(10)) {

Future<String> user = executor.submit(this::findUser);

Future<Integer> order = executor.submit(this::fetchOrder);

String theUser = user.get();

int theOrder = order.get();

return new ResultSet(theUser, theOrder);

}

}

/\*

\* This JEP introduces Structured Concurrency API StructuredTaskScope, which

\* treats multiple tasks running in different threads as a single unit of work.

\*

\* 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, and we can compile and run

\* the program with the below commands.

\*

\* javac --release XX --enable-preview --add-modules jdk.incubator.concurrent

\* Main.java

\*

\* java --enable-preview --add-modules jdk.incubator.concurrent Main

\*/

ResultSet handleStructuredAPI() {

try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {

Future<String> user = scope.fork(this::findUser);

Future<Integer> order = scope.fork(this::fetchOrder);

scope.join();

scope.throwIfFailed();

return new ResultSet(user.resultNow(), order.resultNow());

}

}

}

# JDK 424 - Foreign Function & Memory API (Preview)

This JEP promotes the Foreign Function & Memory API (FFM API) from the incubator stage to the preview stage. The Foreign Function & Memory API (FFM API) resides in the **java.lang.foreign** package of the **java.base** module.

In Project **Panama**, a replacement for the cumbersome, error-prone, and slow Java Native Interface (JNI) has been in the works for a long time.

The "Foreign Memory Access API" and the "Foreign Linker API" were already introduced in Java 14 and Java 16 - both initially individually in the incubator stage. In Java 17, these APIs were combined to form the "Foreign Function & Memory API" (FFM API), which remained in the incubator stage until Java 18.

In Java 19, **JDK Enhancement Proposal 424** finally promoted the new API to the preview stage, which means that only minor changes and bug fixes will be made. So, it's time to introduce the new API!

The Foreign Function & Memory API enables access to native memory (i.e., memory outside the Java heap) and access to native code (e.g., C libraries) directly from Java.

Here is a simple example that stores a string in off-heap memory and calls the "strlen" function of the C standard library on it.

public class \_01\_ForeignFunction\_Memory\_API\_Demo1 {  
  
 public static void main(String[] args) throws Throwable {  
  
 /\* String input = "Java19 Learning";  
 // 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().allocateUtf8String(input);  
 // 4. Runs the foreign function "strlen"  
 long length = (long)methodHandle.invoke(memorySegment);  
 System.out.println("Length = "+length);\*/  
 }  
}

# Locale class New Static Method

In Java 19, the public constructors of the Locale class were marked as "deprecated".

Instead, we should use the **new static factory method Locale.of()**. This ensures that there is only one instance per Locale configuration.

The following example shows the use of the factory method compared to the constructor.

public class \_01\_Locale\_Of\_Demo1 {

public static void main(String[] args) {

// Deprecated

// Locale locale = new Locale();

Locale locale1 = Locale.*of*("de");

Locale locale2 = Locale.*of*("de", "DE");

System.***out***.println(locale1 == Locale.***GERMAN***);

System.***out***.println(locale2 == Locale.***GERMANY***);

}

}

# References

<https://www.happycoders.eu/java/java-19-features/>

<https://mkyong.com/java/what-is-new-in-java-19/>