**Java 19**

*The much-anticipated Java 19 is now available.In this blog you will learn more about Java 19.*

Targeted Audience: Developers, Architects.

*JDK 19 will receive updates under these terms, until****March 2023****when it will be superseded by JDK 20.*

In this article I will share the key new features introduced in Java 19 and other details.

**Concurrency Model Update: Preview**

This new feature is in preview mode, but I am mentioning it here to draw attention to this useful feature.

I hope these preview features can come as features in Java 20. 😇

In the current JDK, java.lang.Thread is a *platform thread*. A platform thread runs Java code on an underlying OS thread and captures the OS thread for the code’s entire lifetime. The number of platform threads is limited to the number of OS threads.

Let me explain with an example.

If you want to run 100 threads using the ExecutorService, one of the methods, e.g., Executors.newCachedThreadPool(), will create 100 Java threads, which will create 100 platform threads, meaning 100 CPU threads. If I increase the count to 10,000 or 20,000, then it will create 20,000 Java threads and it will have 20,000 CPU threads. And based on your machine and with modern hardware, it can crash or hang.

But see the below code. If you use this code and run it on your machine, it will work and not crash with modern hardware.

This big difference comes with this new feature, *newVirtualThreadPerTaskExecutor*. **This method creates 20,000 Java threads but doesn’t create 20,000 CPU/platform threads.**

Behind the scenes, the JDK runs the code on a small number of OS threads, perhaps as few as one.

try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {  
 IntStream.range(0, 20\_000).forEach(i -> {  
 executor.submit(() -> {  
 Thread.sleep(Duration.ofMillis(100));  
 return i;  
 });  
 });  
}

Yes, you can use ExecutorService to reduce the number of threads, but if you use Executors.newFixedThreadPool(200), it will create 200 threads and 200 platform threads.

Then so many tasks will work in sequential rather than in parallel due to the fact that there are only 200 threads and 20,000 tasks.

**When Virtual Thread is useful,**

* The number of concurrent tasks is high.
* The workload is not CPU-bound, since having many more threads than processor cores cannot improve throughput in that case.

*To run preview features code in Java you have to use argument when compile and run.*

java — source 19 — enable-preview [Main.java](http://main.java/)

**New Date-Time Formats: Feature**

Additional Date-Time formats are introduced in Java 19. In the java.time.format classes, DateTimeFormatter and DateTimeFormatterBuilder.

DateTimeFormatter.ofLocalizedPattern

Below is a use case for that. It helps to pass string patterns for your locale.

Locale.setDefault(Locale.CHINA);  
LocalDateTime date = LocalDateTime.now();  
String text = date.format(DateTimeFormatter.ofLocalizedPattern(“yMMM”));  
System.out.println(“DATE: “ + text);

Now the users can specify their own flexible style with this.

**Preallocated HashMaps and HashSets: Feature**

There is new static factory method introduced in below classes,

* HashMap.newHashMap
* LinkedHashMap.newLinkedHashMap
* WeakHashMap.newWeakHashMap
* HashSet.newHashSet
* LinkedHashSet.newLinkedHashSet

This factory method, such as HashMap.newHashMap, will create a HashMap with **preallocated space to accommodate an expected number of mappings.**

You think there is another way: the int argument constructor. But it’s not the same.

These constructors for these classes set the “capacity” ( internal table size), which is not the same as the number of elements that can be accommodated.

HashMap<String, Integer> map = HashMap.newHashMap(5);  
IntStream.of(1,2,3,4,5,6,7).forEach(val -> map.put(Integer.toString(val), val));  
map.forEach((k,v) -> System.out.println(k));

**When to use:**

When you know how many elements or maps you need to store in a map or set, use this new factory static method compared to the int-argument constructor.