Java 21

Released on September 19, 2023, Java 21 is the latest long term support release of Java. This JDK release not only marks an increasing cadence in Java LTS releases from Oracle, but also includes several preview and permanent features in the language. Java 21 marks the first installment of Oracle's commitment to increasing the cadence of Java LTS versions. Java 17 was released in September 2021, marking the start of a two-year span between long term support versions. Previously, Java LTS versions were released every three years, and prior to that even more sporadically. This means that new features are maturing over a much quicker period of time.

A new version of JDK isn't impetus enough for many Java shops to drop what they're doing and upgrade to a new version. After all, upgrading is a lot of work and can cause disruption to mission-critical business applications. According to the 2023 Java Developer Productivity Report, 31% of developers are using Java 8 for their main application, 28% of developers are using Java 11, and only 19% of developers are using the newest Java version as of that survey. If that trend continues, adoption of Java 21 may be sluggish, but there are plenty of innovative features worth making the upgrade for.

What's a JEP and Why Does it Matter?

JEP is a Java development kit enhancement proposal. JEPs are a formalized way by which new features are added to the Java language on a preliminary or permanent basis. Further, there are three different types of JEPs:

- Incubator JEPs are a means of putting not-yet-final tools and APIs in the hands of developers while those tools and APIs progress toward either finalization or permanent removal in a future release. Incubator JEPs are a good way to test new features today, but put them into production at your own risk.
- Preview JEPs are features for which the design, specification, and implementation is finalized but not yet permanent, which means that they may yet be changed or removed at a future release. Preview JEPs must also be specifically enabled to be used.
- All other JEPs are essentially permanent. While it's not an official title, I'll refer to them as "Permanent JEPs" to differentiate.

Java 21 includes 15 JEPs: one is an incubator, five are previews and nine are permanent features. Here's the full list:

Incubator JEP

The incubator JEP included in Java 21 is:

- JEP 448 - Vector API (Sixth Incubator)

Preview JEPs

The six preview JEPs included in Java 21 are:

- JEP 430: String Templates
- JEP 442: Foreign Function & Memory API (Third Preview)
- JEP 443: Unnamed Patterns and Variables

- JEP 445: Unnamed Classes and Instance Main Methods
- JEP 446: Scoped Values
- JEP 453: Structured Concurrency

Permanent JEPs

The eight permanent JEPs included in Java 21 are:

- JEP 431: Sequenced Collections
- JEP 439: Generational ZGC
- JEP 440: Record Patterns
- JEP 441: Pattern Matching for switch
- JEP 444: Virtual Threads
- JEP 449: Deprecate the Windows 32-bit x86 Port for Removal
- JEP 451: Prepare to Disallow the Dynamic Loading of Agents
- JEP 452: Key Encapsulation Mechanism API

Hidden Gems in Java 21

With definitions out of the way, here's a look at some of our favorite features included in the latest Java LTS release. While not all of the JEPs qualify as "hidden gems" (some, like virtual threads, should be quite familiar unless you've been developing under a rock) they should all be on your list to test for production readiness. Here's a closer look at 11 of our favorites.

JEP 430: String Templates

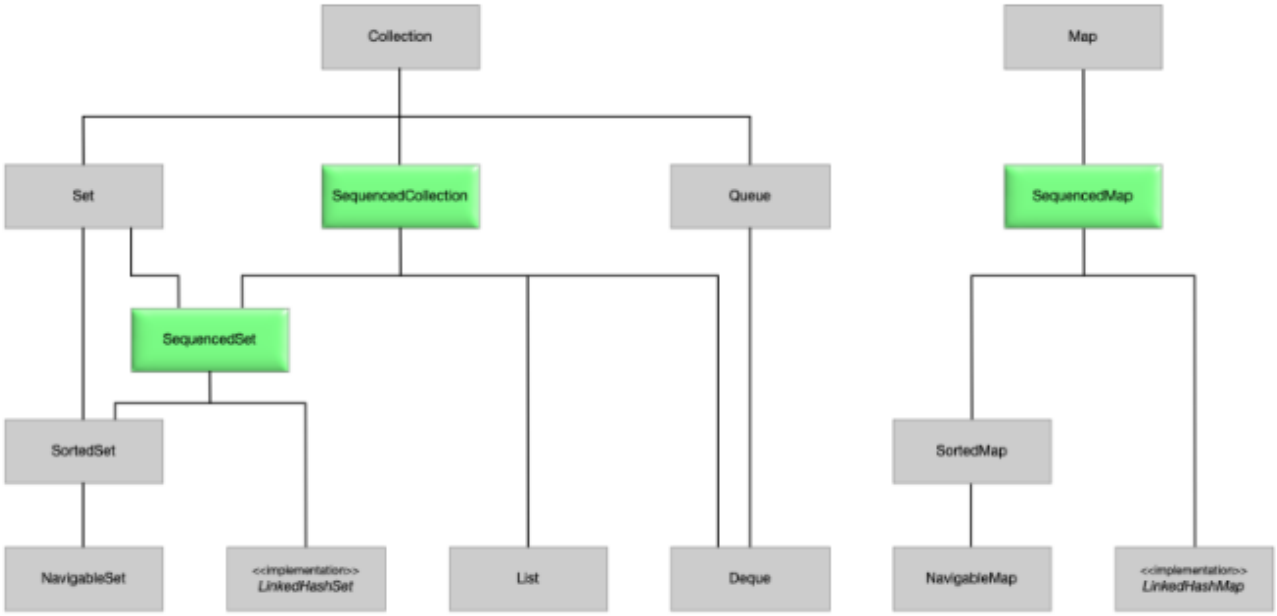
String templates couple literal text with embedded expressions and template processors to produce specialized results. This was theoretically possible prior to JEP 430 but the process was messy; now it's much more straightforward and comparable to other programming languages, like in the example below:

```
class StringTemplateExample {
        public static void main(String[] args) {
        String name = "Bleston";
        String info = STR."My name is \{name}";
        System.out.println(info);
        }
}
```

This feature makes it easier and more intuitive for Java developers to write strings. If you'd like to try string templates out yourself, you have to add compilation flags in order to do so.

JEP 431: Sequenced Collections

Sequenced collections represent collections with a defined encounter order. Using this feature, you should easily be able to get to the first and last elements, and process elements in reverse order. You might already be doing this today, but now the process is much easier. Prior to JEP 431, there wasn't a common interface for these collections and the operations were different. The diagram below illustrates the **SequencedSet**, **SequencedCollection**, and **SequencedMap** interfaces.




```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

import java.util.LinkedHashSet;
import java.util.SequencedCollection;

class SequencedCollectionTest {

    @Test
    void ordering() throws Exception {
        var list = LinkedHashSet.<String>newLinkedHashSet(100);
        if (list instanceof SequencedCollection<String> sequencedCollection) {
            sequencedCollection.add("ciao");
            sequencedCollection.add("hola");
            sequencedCollection.add("ni hao");
            sequencedCollection.add("salut");
            sequencedCollection.add("hello");
            sequencedCollection.addFirst("ola"); //<1>
            Assertions.assertEquals(sequencedCollection.getFirst(), "ola"); // ②
        }
    }
}
```

There are similar methods for getLast and addLast, and there's even support for reversing a collection, with the reverse method.

JEP 439: Generational Z Garbage Collector (ZGC)

The Generational ZGC improves the Z Garbage Collector to maintain separate generations for young and old objects. It also allows the ZGC to collect young objects more frequently, and older objects less frequently. If you're currently using the ZGC, you'll see significantly improved performance. Right now Generational ZGC has to be specifically enabled, but in the future it will be enabled by default.

JEP 440: Record Patterns

Record Patterns enable Java developers to deconstruct record values; they can be nested with type patterns to enable a powerful, declarative, and composable form of data navigation and

processing. Record patterns first appeared as a preview feature in Java 19 and were previewed for a second time with some tweaks in Java 20.

JEP 442: Foreign Function & Memory API

Foreign Function & Memory API allows Java programs to interoperate with code and data not managed by the JVM. JEP 442 enables foreign memory management for code outside of the JVM safely, without worrying about causing disruption to the underlying operating system.

JEP 444: Virtual Threads

Virtual threads could pose a significant time savings for Java programmers who are doing concurrent programming and dealing with threads. Instead of having to modify your code for concurrency and specify threads—a process that can quickly get messy—virtual threads enable the benefits of concurrency with greater performance and less complexity. That said, they may be complex to implement in existing large applications. They're also not a panacea; you can't substitute virtual threads for CPU threads and magically expect the same level of performance. Virtual threads are hardly a hidden gem—they've been previewed multiple times in previous JDK versions—but there's excitement to see how the Java community can incorporate this feature into new applications.

Finally, we get to Loom. You've no doubt heard a lot about Loom. The basic idea is to make scalable the code you wrote in college! What do I mean by that? Let's write a simple network service that prints out whatever is given to us. We must read from one InputStream and accrue everything into a new buffer (a ByteArrayOutputStream). Then, when the request finishes, we'll print the contents o the ByteArrayOutputStream. The problem is we might get a lot of data simultaneously. So, we will use threads to handle more than one request at the same time.

```
import java.io.ByteArrayOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.Executors;

class NetworkServiceApplication {

    public static void main(String[] args) throws Exception {
        try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
            try (var serverSocket = new ServerSocket(9090)) {
                while (true) {
                    var clientSocket = serverSocket.accept();
                    executor.submit(() -> {
                        try {
                            handleRequest(clientSocket);
                        } catch (Exception e) {
                            throw new RuntimeException(e);
                        }
                    });
                }
            }
        }
    }

    static void handleRequest(Socket socket) throws Exception {
        var next = -1;
        try (var baos = new ByteArrayOutputStream()) {
            try (var in = socket.getInputStream()) {
                while ((next = in.read()) != -1) {
                    baos.write(next);
                }
            }
            var inputMessage = baos.toString();
            System.out.println("request: %s".formatted(inputMessage));
        }
    }
}
```

It’s pretty trivial Networking-101 stuff. Create a ServerSocket, and wait for new clients (represented by instances of Socket) to appear. As each one arrives, hand it off to a thread from a threadpool. Each Thread reads data from the client Socket instance’s InputStream references. Clients might disconnect, experience latency, or have a large chunk of data to send, all of which is a problem because there are only so many threads available and we must not waste our precious little time on them.

We’re using threads to avoid a pileup of requests we can’t handle fast enough. But here again we’re defeated because, before Java 21, threads were expensive! They cost about two megabytes of RAM for each Thread. And so we pool them in a thread pool and reuse them. But even there, if we have too many requests, we’ll end up in a situation where none of the threads in the pool are available. They’re all stuck waiting on some request or another to finish. Well, sort of. Many are just sitting there, waiting for the next byte from the InputStream, but they’re unavailable for use.

The threads are blocked. They’re probably waiting for data from the client. The unfortunate state of things is that the server, waiting on that data, has no choice but to sit there, parked on a thread, not allowing anybody else to use it.

Until now, that is. Java 21 introduces a new sort of thread, a virtual thread. Now, we can create millions of threads for the heap. It’s easy. But fundamentally, the facts on the ground are that the actual threads, on which virtual threads execute, are expensive. So, how can the JRE let us have millions of threads for actual work? It has a vastly improved runtime that now notices when we block and suspend execution on the Thread until the thing we’re waiting for arrives. Then, it quietly puts us back on another thread. The actual threads act as carriers for virtual threads, allowing us to start millions of threads.

Java 21 has improvements in all the places that historically block threads, like blocking IO with InputStream and OutputStream, and Thread.sleep(), so now they correctly signal to the runtime that it is ok to reclaim the Thread and repurpose it for other virtual threads, allowing work to progress even when a virtual thread is 'blocked'. You can see that in this example, which I shamelessly stole from José Paumard, one of the Java Developer Advocates at Oracle whose work I love.

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

import java.io.ByteArrayOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.ConcurrentSkipListSet;
import java.util.concurrent.Executors;
import java.util.stream.IntStream;

class LoomTest {

    @Test
    void loom() throws Exception {

        var observed = new ConcurrentSkipListSet<String>();

        var threads = IntStream
                .range(0, 100)
                .mapToObj(index -> Thread.ofVirtual() // ⊙
                        .unstarted(() -> {
                            var first = index == 0;
                            if (first) {
                                observed.add(Thread.currentThread().toString());
                            }
```

```
                        try {
                            Thread.sleep(20);
                        } catch (InterruptedException e) {
                            throw new RuntimeException(e);
                        }
                        if (first) {
                            observed.add(Thread.currentThread().toString());
                        }
                        try {
                            Thread.sleep(20);
                        } catch (InterruptedException e) {
                            throw new RuntimeException(e);
                        }
                        if (first) {
                            observed.add(Thread.currentThread().toString());
                        }
                    }))
                .toList();

        for (var t : threads)
            t.start();

        for (var t : threads)
            t.join();

        System.out.println(observed);

        Assertions.assertTrue(observed.size() > 1);

    }

}
```

JEP 445: Unnamed Classes and Instance Main Methods

This preview JEP poses the most potential for Java developers who are new to the language, or for more experience developers who are teaching others the language. This feature has some limitations, mainly that unnamed classes can't reference named classes and vice versa. It's also somewhat controversial, since while it may make Java easier to learn for individuals coming from other languages, it could potentially cause newcomers to learn some bad habits.

JEP 448: Vector API (Sixth Incubator)

Vectors have been in the Java development language nearly since the beginning, but until now there hasn't been an API to utilize CPU hardware when doing so. That changes with JEP 448. Use cases for the Vector API include linear algebra, image processing, and character decoding, among others. As mentioned earlier, the main advantage of this API is that it's hardware supported. Take a closer look at the documentation before you dive into testing this feature, however, because certain CPUs are not supported.

JEP 449: Deprecate the Windows 32-bit x86 Port for Removal

End of life for the final Windows 10 (Version 22H2) is October 14, 2025. That date is important because Windows 10 is the last version that supports 32-bit operations. With JEP 449, Java developers will get an error when they configure the 32-bit x86 port.

JEP 451: Prepare to Disallow the Dynamic Loading of Agents

This new feature will issue a warning when agents are loaded dynamically into a running JVM; in the future it will issue an error. The main benefit of this is increased security. Libraries that currently load agents dynamically will need to be changed to load them at startup with --javaagent/--agentlib options or enable dynamic loading with --XX:+EnableDynamicAgentLoading

JEP 452: Key Encapsulation Mechanism API

This API allows for creating of encryption keys using key encapsulation mechanisms (KEMs) to comply with the latest security regulations including those proposed in RSA Key Encapsulation Mechanism (RSA-KEM), the Elliptic Curve Integrated Encryption Scheme (ECIES), and candidate KEM algorithms for the National Institute of Standards and Technology (NIST) Post-Quantum Cryptography standardization process.

JEP 453: Structured Concurrency

Structured concurrency simplifies concurrent programming by introducing an API that treats groups of related tasks running in different threads as a single unit of work. This is a preview JEP that was first introduced as an incubator JEP in Java 19.