***Scoped Values*** *- Scoped values are a modern alternative to thread locals that can be combined well with virtual threads. They allow storing a value for a limited time in such a way that only the thread that wrote the value can read it.*

*We’ve been using variables of the type ThreadLocal to allow components to share data.*

*But it has some consequences:*

* *A ThreadLocal variable is mutable. The ThreadLocal API allows access to both get() and set() methods of its variable type.*
* *We may face memory leak issues since the value of the ThreadLocal variables is retained until we explicitly call the remove() method on it or the thread exits. Thus, there’s no binding to the lifetime of these per-thread variables.*
* *They may lead to excessive memory footprints in case of using large numbers of threads. This is because the child threads can inherit the ThreadLocal variables of parent threads, thus allocating memory for every ThreadLocal variable.*

*To overcome these problems of ThreadLocal variables, Java 20 has introduced scoped values for sharing data within and across threads.*

***Scoped values provide a simple, immutable, and inheritable data-sharing option, specifically in situations where we’re working with a large number of threads.***

*A ScopedValue is an immutable value that is available for reading for a bounded period of execution by a thread. Since it’s immutable, it allows safe and easy data-sharing for a limited period of thread execution. Also, we need not pass the values as method arguments.*

*We can use the where() method of the class ScopedValue to set the value of a variable for the bounded period of thread execution. Moreover, once we get the data using the get() method, we cannot access it again.*

*Once the run() method of a thread finishes execution, the scoped value reverts to the unbound state. We can use the get() method to read the value of a scoped-value variable inside a thread.*

*Record Patterns - The following changes were made in Java 20 with JDK Enhancement Proposal 432:*

*Inference of Type Arguments of Generic Record Patterns - To explain this change, we need a more complex example. Given are a generic interface Multi<T> and two implementing records, Tuple<T> and Triple<T>, which contain two and three values of type T*

*interface Multi<T> {}*

*record Tuple<T>(T t1, T t2) implements Multi<T> {}*

*record Triple<T>(T t1, T t2, T t3) implements Multi<T> {}*

*we can check which concrete implementation a given Multi object is*

*Multi<String> multi = ...*

*if (multi instanceof Tuple<String>(var s1, var s2)) {*

*System.out.println("Tuple: " + s1 + ", " + s2);*

*} else if (multi instanceof Triple<String>(var s1, var s2, var s3)) {*

*System.out.println("Triple: " + s1 + ", " + s2 + ", " + s3);*

*}*

*As of Java 20, the compiler can infer the type so that we can omit it from the instanceof checks –*

*if (multi instanceof Tuple(var s1, var s2)) {*

*System.out.println("Tuple: " + s1 + ", " + s2);*

*} else if (multi instanceof Triple(var s1, var s2, var s3)) {*

*System.out.println("Triple: " + s1 + ", " + s2 + ", " + s3);*

*}*

***Record Patterns in for Loops*** *–*

*List<Position> positions = ...*

*for (Position p : positions) {*

*System.out.printf("(%d, %d)%n", p.x(), p.y());*

*}*

*with Java 20, we can also specify a record pattern in the for loop and then access x and y directly.*

*for (Position(int x, int y) : positions) {*

*System.out.printf("(%d, %d)%n", x, y);*

*}*

*Removal of Support for Named Record Patterns –*

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

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

*+ ", x = " + x + ", y = " + y);*

*}*

*This variant (“named record pattern”), there are two ways to access the fields of the record – either via the x and y variables – or via p.x() and p.y().*

*This variant was decided to be superfluous and removed again in Java 20.*

***Pattern Matching for Switch (JEP 433)*** *- Java 20 provides a refined version of pattern matching for switch expressions and statements, specifically about the grammar used in switch expressions. The main changes in this release include:*

* *Using a switch expression or a pattern over an enum class or sealed class now throws a MatchException. Earlier, we used to get an IncompatibleClassChangeError if no switch label was applied at run time.*

*For ex –*

*public sealed interface Shape permits Rectangle, Circle {}*

*public record Rectangle(Position topLeft, Position bottomRight) implements Shape {}*

*public record Circle(Position center, int radius) implements Shape {}*

*public class ShapeDebugger {*

*public static void debug(Shape shape) {*

*switch (shape) {*

*case Rectangle r -> System.out.println(*

*"Rectangle: top left = " + r.topLeft() + "; bottom right = " + r.bottomRight());*

*case Circle c -> System.out.println(*

*"Circle: center = " + c.center() + "; radius = " + c.radius());*

*}*

*}*

*}*

*Since the compiler knows all possible implementations of the sealed Shape interface, it can ensure that this switch expression is exhaustive.*

*Then we add another shape Oval, add it to the permits list of the Shape interface, and extend the main program.*

*It will immediately tell us that the switch statement in the ShapeDebugger does not cover all possible values.*

*The case needs to be added for third newly added Shape implementation i.e. Oval*

* *There are improvements in the grammar for switch labels.*
* *They have added support for type-inference of arguments for generic record patterns in switch expressions and statements, along with the other constructs that support patterns.*

*Ex –*

*Multi<String> multi = ...*

*switch(multi) {*

*case Tuple<String>(var s1, var s2) -> System.out.println(*

*"Tuple: " + s1 + ", " + s2);*

*case Triple<String>(var s1, var s2, var s3) -> System.out.println(*

*"Triple: " + s1 + ", " + s2 + ", " + s3);*

*...*

*}*

*Starting with Java 20, we can omit the <String> type arguments inside the switch statement:*

*switch(multi) {*

*case Tuple(var s1, var s2) -> System.out.println(*

*"Tuple: " + s1 + ", " + s2);*

*case Triple(var s1, var s2, var s3) -> System.out.println(*

*"Triple: " + s1 + ", " + s2 + ", " + s3);*

*...*

*}*

*Foreign Function and Memory API - The Foreign Function and Memory API allow Java developers to access code from outside the JVM and manage memory out of the heap (i.e., memory not managed by the JVM)*

*It includes the following refinements:*

1. *The MemorySegment and MemoryAddress abstractions are unified. This means that we can actually determine the range of memory addresses associated with a segment from its spatial bounds.*

* *They’ve also facilitated the use of pattern matching in switch expressions and statements by enhancing the sealed MemoryLayout hierarchy.*
* *Moreover, they’ve split the MemorySession into Arena and SegmentScope to facilitate sharing segments across maintenance boundaries.*

***Virtual Threads*** *- Virtual threads are lightweight threads that reduce the effort of writing, maintaining, and observing high-throughput concurrent applications. Thus, it makes it easy to debug and troubleshoot the server applications with existing JDK tools. This could be useful in the scaling of server applications.*

*However, we should note that the traditional Thread implementation still exists, and it doesn’t aim to replace the basic concurrency model of Java.*

*Below are a few of the minor changes since the first preview –*

* *They made the previously introduced methods in Thread – join(Duration), sleep(Duration), and threadId() – permanent in Java 20.*
* *Similarly, newly introduced methods in Future to examine the task state and result – state() and resultNow() – were made permanent in Java 20.*
* *Additionally, ExecutorService now extends AutoCloseable.*
* *The degradations to ThreadGroup, the legacy API for grouping threads, described in JEP 425, were made permanent in JDK 19. The ThreadGroup isn’t suitable for grouping virtual threads.*

***Structured Concurrency*** *- The goal is to simplify multithreaded programming by introducing an API for structured concurrency. It improves reliability by grouping multiple threads doing similar tasks into a single unit of work. As a result, there’s improved error handling and thread cancellations. Additionally, it promotes an improved way of concurrent programming aiming to protect from common risks arising from thread cancellation.*

*However, there’s one change in this re-incubated API. We get an updated StructuredTaskScope that supports the inheritance of scoped values by threads created in a task scope. Thus, we can now conveniently share immutable data across multiple threads.*

***Vector API*** *- This release doesn’t introduce any change to the API compared to Java 19. However, it includes a few bug fixes and performance enhancements.*

*Finally, development in the Vector API is closely aligned with Project Valhalla since it aims to adapt Project Valhalla’s enhancements in the future.*