Java 20 Release

New Features :

- The vector API proposal
- Virtual threads
- Structured concurrency
- Scoped values
- Record patterns
- Foreign function and memory API
- Pattern matching for switch statements and expressions

1. Vector API (Fifth Incubator) - https://openjdk.org/jeps/438 :

- API to express vector computations that reliably compile at runtime to optimal vector instructions on supported CPU architectures
- Incubated in JDK 16, 17, 18 and 19.
- Goals : Clear and concise API, Platform agnostic, Reliable runtime compilation and performance on x64 and AArch64 architectures, Graceful degradation, Alignment with Project Valhalla
- A vector is represented by the abstract class `Vector<E>`
- A vector also has a *shape* which defines the size, in bits, of the vector. The set of shapes supported correspond to vector sizes of 64, 128, 256, and 512 bits
- The set of element types (`E`) supported is `Byte`, `Short`, `Integer`, `Long`, `Float` and `Double`, corresponding to the scalar primitive types `byte`, `short`, `int`, `long`, `float` and `double`, respectively.
- Operations on vectors are classified as either *lane-wise* or *cross-lane*.
- The combination of element type and shape determines a vector's *species*, represented by `VectorSpecies<E>` and operations on vectors are classified as either *lane-wise* or *cross-lane.*
- To support control flow, some vector operations optionally accept masks represented by the public abstract class `VectorMask<E>`. Each element in a mask is a boolean value corresponding to a vector lane.
- To support cross-lane permutation operations, some vector operations accept shuffles represented by the public abstract class `VectorShuffle<E>`.

Example :

Here is a simple scalar computation over elements of arrays:

```
void scalarComputation(float[] a, float[] b, float[] c) {
   for (int i = 0; i < a.length; i++) {
        c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
   }
}
```

(We assume that the array arguments are of the same length.)

Here is an equivalent vector computation, using the Vector API:

```
static final VectorSpecies<Float> SPECIES = FloatVector.SPECIES_PREFERRED;

void vectorComputation(float[] a, float[] b, float[] c) {
    for (int i = 0; i < a.length; i += SPECIES.length()) {
        // VectorMask<Float>  m;
        var m = SPECIES.indexInRange(i, a.length);
        // FloatVector va, vb, vc;
        var va = FloatVector.fromArray(SPECIES, a, i, m);
        var vb = FloatVector.fromArray(SPECIES, b, i, m);
        var vc = va.mul(va)
                   .add(vb.mul(vb))
                   .neg();
        vc.intoArray(c, i, m);
    }
}
```

2. Virtual Threads (Second Preview) - https://openjdk.org/jeps/436 :

- Virtual threads are lightweight threads that dramatically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications.
- **Goals :** 1. Simple thread-per-request style to scale with near-optimal hardware utilization, 2. Enable existing code that uses the `java.lang.Thread` API to adopt virtual threads with minimal change, 3. Enable easy troubleshooting, debugging, and profiling of virtual threads with existing JDK tools.
- *The thread-per-request style*
- *Improving scalability with the asynchronous style*
- *Preserving the thread-per-request style with virtual threads*

Using virtual threads vs. platform threads

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 10_000).forEach(i -> {
        executor.submit(() -> {
            Thread.sleep(Duration.ofSeconds(1));
            return i;
        });
    });
}  // executor.close() is called implicitly, and waits
```

Virtual threads are a [preview API](), disabled by default

The programs above use the `Executors.newVirtualThreadPerTaskExecutor()` method, so to run them on JDK 20 you must enable preview APIs as follows:

- Compile the program with `javac --release 20 --enable-preview Main.java` and run it with `java --enable-preview Main`; or,
- When using the [source code launcher](), run the program with `java --source 20 --enable-preview Main.java`; or,
- When using [jshell](), start it with `jshell --enable-preview`.

- Observing virtual threads – `$ jcmd <pid> Thread.dump_to_file -format=json <file>`

3. Structured Concurrency(Second Incubator)- https://openjdk.org/jeps/437:

- Structured concurrency treats multiple tasks running in different threads as a single unit of work, thereby streamlining error handling and cancellation, improving reliability, and enhancing observability.
- *Unstructured concurrency with* `ExecutorService`

```
Response handle() throws ExecutionException, InterruptedException {
    Future<String>  user  = esvc.submit(() -> findUser());
    Future<Integer> order = esvc.submit(() -> fetchOrder());
    String theUser  = user.get();   // Join findUser
    int    theOrder = order.get();  // Join fetchOrder
    return new Response(theUser, theOrder);
}
```

- The principal class of the structured concurrency API is StructuredTaskScope. This class allows developers to structure a task as a family of concurrent subtasks, and to coordinate them as a unit. Subtasks are executed in their own threads by *forking* them individually and then *joining* them as a unit and, possibly, cancelling them as a unit.

Here is the `handle()` example from earlier, written to use `StructuredTaskScope` (`ShutdownOnFailure` is explained below,):

```
Response handle() throws ExecutionException, InterruptedException {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        Future<String>  user  = scope.fork(() -> findUser());
        Future<Integer> order = scope.fork(() -> 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());
    }
}
```

- `ShutdownOnSuccess`
- Error handling with short-circuiting
- Cancellation propagation
- Clarity
- Observability

4. Scoped Values (Incubator) - https://openjdk.org/jeps/429 :

- *scoped values*, which enable the sharing of immutable data within and across threads.
- *Goals :* Ease of use, Comprehensibility, Robustness, Performance

Thread-local variables for sharing :

```
Thread 1                                 Thread 2
--------                                 --------
8. DBAccess.newConnection()              8. throw new
InvalidPrincipalException()
7. DBAccess.open() <----------+          7. DBAccess.open() <----------+
   ...                        |             ...                        |
   ...                  Principal(ADMIN)    ...               Principal(GUEST)
2. Application.handle(..)     |          2. Application.handle(..)     |
1. Server.serve(..) ----------+          1. Server.serve(..) ----------+
```

```
class Server {
    final static ThreadLocal<Principal> PRINCIPAL = new ThreadLocal<>();
// (1)
    void serve(Request request, Response response) {
        var level     = (request.isAuthorized() ? ADMIN : GUEST);
        var principal = new Principal(level);
        PRINCIPAL.set(principal);
// (2)
        Application.handle(request, response);
    }
}
class DBAccess {
    DBConnection open() {
        var principal = Server.PRINCIPAL.get();
// (3)
        if (!principal.canOpen()) throw new InvalidPrincipalException();
        return newConnection(...);
// (4)
    }
}
```

Problems with thread-local variables :

- Unconstrained mutability
- Unbounded lifetime
- Expensive inheritance

Toward lightweight sharing :

- A *scoped value* allows data to be safely and efficiently shared between components in a large program without resorting to method arguments. It is a variable of type `ScopedValue`.

```
final static ScopedValue<...> V = new ScopedValue<>();
// In some method
ScopedValue.where(V, <value>)
           .run(() -> { ... V.get() ... call methods ... });
// In a method called directly or indirectly from the lambda expression
... V.get() ...
```

5. Record Patterns (Second Preview) - https://openjdk.org/jeps/432 :

- Record patterns and type patterns can be nested to enable a powerful, declarative, and composable form of data navigation and processing.
- Goals : Extend pattern matching to express more sophisticated, composable data queries, Do not change the syntax or semantics of type patterns.

```
// Old code
if (obj instanceof String) {
    String s = (String)obj;
    ... use s ...
}
// New code
if (obj instanceof String s) {
    ... use s ...
}
```

Pattern matching and record classes :

```
record Point(int x, int y) {}
static void printSum(Object obj) {
    if (obj instanceof Point p) {
        int x = p.x();
        int y = p.y();
        System.out.println(x+y);
    }
}
```

The true power of pattern matching is that it scales elegantly to match more complicated object graphs. For example, consider the following declarations:

```
record Point(int x, int y) {}
enum Color { RED, GREEN, BLUE }
record ColoredPoint(Point p, Color c) {}
record Rectangle(ColoredPoint upperLeft, ColoredPoint lowerRight) {}
```

If we want to extract the color from the upper-left point, we could write:

```
static void printUpperLeftColoredPoint(Rectangle r) {
    if (r instanceof Rectangle(ColoredPoint ul, ColoredPoint lr)) {
         System.out.println(ul.c());
    }
}
```

Nested patterns can, of course, fail to match:

```
record Pair(Object x, Object y) {}

Pair p = new Pair(42, 42);

if (p instanceof Pair(String s, String t)) {
    System.out.println(s + ", " + t);
} else {
    System.out.println("Not a pair of strings");
}
```

Record patterns :

If a record class is generic then it can be used in a record pattern as either a parameterized type or as a raw type. For example:

```
record Box<T>(T t) {}

static void test1(Box<String> bo) {
    if (bo instanceof Box<String>(var s)) {
        System.out.println("String " + s);
    }
}
```

Nested - `if (bo instanceof Box<Box<String>>(Box(var s)))`

Record patterns and exhaustive switch :

```
class A {}
class B extends A {}
sealed interface I permits C, D {}
final class C implements I {}
final class D implements I {}
record Pair<T>(T x, T y) {}
Pair<A> p1;
Pair<I> p2;
```

The following `switch` is not exhaustive, since there is no match for a pair containing two values both of type `A`:

```
switch (p1) {                  // Error!
    case Pair<A>(A a, B b) -> ...
    case Pair<A>(B b, A a) -> ...
}
```

These two switches are exhaustive, since the interface `I` is `sealed` and so the types `C` and `D` cover all possible instances:

```
switch (p2) {
    case Pair<I>(I i, C c) -> ...
    case Pair<I>(I i, D d) -> ...
}
switch (p2) {
    case Pair<I>(C c, I i) -> ...
    case Pair<I>(D d, C c) -> ...
    case Pair<I>(D d1, D d2) -> ...
}
```

In contrast, this `switch` is not exhaustive since there is no match for a pair containing two values both of type `D`:

```
switch (p2) {                         // Error!
    case Pair<I>(C fst, D snd) -> ...
    case Pair<I>(D fst, C snd) -> ...
    case Pair<I>(I fst,
```

Record patterns and enhanced for statements :

If `R` is a record pattern then an enhanced `for` statement of form

```
for (R : e) S
```

is equivalent to the following enhanced `for` statement, which has no record pattern in the header:

```
for (var tmp : e) {
    switch(tmp) {
```

```
        case null -> throw new MatchException(new NullPointerException());
        case R -> S;
    }
}
```

This translation has the following consequences:

- The record pattern `R` must be *applicable* to the element type of the array or `Iterable`.
- The record pattern `R` must be *exhaustive* for the element type of the array or `Iterable`.
- Should any element of `e` be `null` then the execution of the enhanced `for` statement results in `MatchException` being thrown

6. Pattern Matching for switch (Fourth Preview) - https://openjdk.org/jeps/433 :

- Enhance the Java programming language with pattern matching for switch expressions and statements.

```
static String formatterPatternSwitch(Object obj) {
    return switch (obj) {
        case Integer i -> String.format("int %d", i);
        case Long l    -> String.format("long %d", l);
        case Double d  -> String.format("double %f", d);
        case String s  -> String.format("String %s", s);
        default        -> obj.toString();
    };
}
```

Switches and null :

```
static void testFooBar(String s) {
    if (s == null) {
        System.out.println("Oops!");
        return;
    }
    switch (s) {
        case "Foo", "Bar" -> System.out.println("Great");
        default           -> System.out.println("Ok");
    }
}
```

Refactoring too -

```
static void testFooBar(String s) {
    switch (s) {
        case null         -> System.out.println("Oops");
        case "Foo", "Bar" -> System.out.println("Great");
        default           -> System.out.println("Ok");
    }
}
```

Case refinement :

```
class Shape {}
class Rectangle extends Shape {}
class Triangle  extends Shape { int calculateArea() { ... } }

static void testTriangle(Shape s) {
    switch (s) {
        case null:
            break;
        case Triangle t:
            if (t.calculateArea() > 100) {
                System.out.println("Large triangle");
                break;
            }
        default:
            System.out.println("A shape, possibly a small triangle");
    }
}
```

Can refactor to -

```
static void testTriangle(Shape s) {
    switch (s) {
        case null ->
            { break; }
        case Triangle t
        when t.calculateArea() > 100 ->
            System.out.println("Large triangle");
        case Triangle t ->
            System.out.println("Small triangle");
        default ->
            System.out.println("Non-triangle");     }}
```

Patterns in switch labels :

There are five major language design areas to consider when supporting patterns in `switch`:

1. Enhanced type checking
2. Exhaustiveness of `switch` expressions and statements
3. Scope of pattern variable declarations
4. Dealing with `null`
5. Errors

1. Enhanced type checking

1a. Selector expression typing

```
record Point(int i, int j) {}

enum Color { RED, GREEN, BLUE; }


static void typeTester(Object obj) {

    switch (obj) {

        case null     -> System.out.println("null");

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

        case Color c  -> System.out.println("Color: " + c.toString());

        case Point p  -> System.out.println("Record class: " +
p.toString());

        case int[] ia -> System.out.println("Array of ints of length" +
ia.length);

        default       -> System.out.println("Something else");

    }

}
```

1b. Dominance of case labels :

```
static void first(Object obj) {

    switch (obj) {


        case String s ->

            System.out.println("A string: " + s);


        case CharSequence cs ->

            System.out.println("A sequence of length " + cs.length());

        default -> {

            break;

        }

    }

}
```

1c. Inference of type arguments in record patterns :

```
record MyPair<S,T>(S fst, T snd){};


static void recordInference(MyPair<String, Integer> pair){
    switch (pair) {
        case MyPair(var f, var s) ->
            ... // Inferred record Pattern MyPair<String,Integer>(var f,
var s)
        ...
    }
}
```

2. Exhaustiveness of switch expressions and statements :

Consider this (erroneous) pattern `switch` expression:

```
static int coverage(Object obj) {
    return switch (obj) {         // Error - not exhaustive
        case String s -> s.length();
    };
}
```

Consider this (still erroneous) example:

```
static int coverage(Object obj) {
    return switch (obj) {         // Error - still not exhaustive
        case String s  -> s.length();
        case Integer i -> i;
    };
}
```

The type coverage of a `default` label is all types, so this example is (at last!) legal:

```
static int coverage(Object obj) {
    return switch (obj) {
        case String s  -> s.length();
        case Integer i -> i;
        default -> 0;
    };
}
```

3. Scope of pattern variable declarations :

We extend this flow-sensitive notion of scope for pattern variable declarations to encompass pattern declarations occurring in `case` labels with three new rules:

1. The scope of a pattern variable declaration which occurs in a switch label includes any `when` clause of that label.
2. The scope of a pattern variable declaration which occurs in a `case` label of a `switch` rule includes the expression, block, or `throw` statement that appears to the right of the arrow.
3. The scope of a pattern variable declaration which occurs in a `case` label of a `switch` labeled statement group includes the block statements of the statement group. Falling through a `case` label that declares a pattern variable is forbidden.

This example shows the first rule in action:

```
static void test(Object obj) {
    switch (obj) {
        case Character c
        when c.charValue() == 7:
            System.out.println("Ding!");
            break;
        default:
            break;
        }
    }
}
```

This variant shows the second rule in action:

```
static void test(Object obj) {
    switch (obj) {
        case Character c -> {
            if (c.charValue() == 7) {
                System.out.println("Ding!");
            }
            System.out.println("Character");
        }
        case Integer i ->
            throw new IllegalStateException("Invalid Integer argument: "
                                                + i.intValue());
        default -> {
            break;
        }
    }
}
```

The third rule is more complicated. Let us first consider an example where there is only one `case` label for a `switch` labeled statement group:

```
static void test(Object obj) {
    switch (obj) {
```

```
        case Character c:
            if (c.charValue() == 7) {
                System.out.print("Ding ");
            }
            if (c.charValue() == 9) {
                System.out.print("Tab ");
            }
            System.out.println("Character");
        default:
            System.out.println();
    }
}
```

We forbid the possibility of falling through a `case` label that declares a pattern variable. Consider this erroneous example:

```
static void test(Object obj) {
    switch (obj) {
        case Character c:
            if (c.charValue() == 7) {
                System.out.print("Ding ");
            }
            if (c.charValue() == 9) {
                System.out.print("Tab ");
            }
            System.out.println("character");
        case Integer i:                 // Compile-time error
            System.out.println("An integer " + i);
        default:
            break;
    }
}
```

On the other hand, falling through a label that does not declare a pattern variable is safe, as this example shows:

```
void test(Object obj) {
    switch (obj) {
        case String s:
            System.out.println("A string");
        default:
            System.out.println("Done");
    }
}
```

4. *Dealing with null:*

- If the selector expression evaluates to `null` then any `null` case label is said to match. If there is no such label associated with the switch block then the `switch` throws `NullPointerException`, as before.
- If the selector expression evaluates to a non-`null` value then we select a matching `case` label, as normal. If no `case` label matches then any `default` label is considered to match.

For example, given the declaration below, evaluating `test(null)` will print `null!` rather than throw `NullPointerException`:

```
static void test(Object obj) {
    switch (obj) {
        case null     -> System.out.println("null!");
        case String s -> System.out.println("String");
        default       -> System.out.println("Something else");
    }
}
```

this code:

```
static void test(Object obj) {
    switch (obj) {
        case String s  -> System.out.println("String: " + s);
        case Integer i -> System.out.println("Integer");
        default        -> System.out.println("default");
    }
}
```

is equivalent to:

```
static void test(Object obj) {
    switch (obj) {
        case null      -> throw new NullPointerException();
        case String s  -> System.out.println("String: "+s);
        case Integer i -> System.out.println("Integer");
        default        -> System.out.println("default");
    }
}
```

5. *Errors :*

If no label in a pattern `switch` matches the value of the selector expression then the `switch` completes abruptly by throwing a `MatchException`, since pattern switches must be exhaustive.

For example:

```
record R(int i){
    public int i(){      // accessor method for i
        return i / 0;
    }
}
static void exampleAnR(R r) {
    switch(r) {
        case R(var i): System.out.println(i);
    }
}
```

The invocation `exampleAnR(new R(42))` causes a `MatchException` to be thrown.

By contrast:

```
static void example(Object obj) {
    switch (obj) {
        case R r when (r.i / 0 == 1): System.out.println("It's an R!");
        default: break;
    }
}
```

The invocation `example(new R(42))` causes an `ArithmeticException` to be thrown.

7. Foreign Function & Memory API (Second Preview) - https://openjdk.org/jeps/434 :

- API by which Java programs can interoperate with code and data outside of the Java runtime.
- The Foreign Function & Memory (FFM) API combines two earlier incubating APIs: the Foreign-Memory Access API (JEPs 370, 383, and 393) and the Foreign Linker API (JEP 389).

Foreign memory

- The ByteBuffer API provides *direct* byte buffers, which are Java objects backed by fixed-size regions of off-heap memory
- The sun.misc.Unsafe API provides low-level access to on-heap memory that also works for off-heap memory.

Foreign functions

- JNI involves several tedious artifacts: a Java API (native methods), a C header file derived from the Java API, and a C implementation that calls the native library of interest.
- JNI can only interoperate with libraries written in languages, typically C and C++, that use the calling convention of the operating system and CPU for which the JVM was built.
- JNI does not reconcile the Java type system with the C type system.

Description :

The Foreign Function & Memory API (FFM API) defines classes and interfaces so that client code in libraries and applications can

- Allocate foreign memory (`MemorySegment` and `SegmentAllocator`),
- Manipulate and access structured foreign memory (`MemoryLayout` and `VarHandle`),
- Control the allocation and deallocation of foreign memory (`SegmentScope` and `Arena`),
- Call foreign functions (`Linker`, `FunctionDescriptor`, and `SymbolLookup`).

The FFM API resides in the `java.lang.foreign` package of the `java.base` module.

Example :

```
// 1. Find foreign function on the C library path

Linker linker          = Linker.nativeLinker();

SymbolLookup stdlib    = linker.defaultLookup();

MethodHandle radixsort = linker.downcallHandle(stdlib.find("radixsort"),
...);

// 2. Allocate on-heap memory to store four strings

String[] javaStrings = { "mouse", "cat", "dog", "car" };

// 3. Use try-with-resources to manage the lifetime of off-heap memory

try (Arena offHeap = Arena.openConfined()) {

    // 4. Allocate a region of off-heap memory to store four pointers

    MemorySegment pointers = offHeap.allocateArray(ValueLayout.ADDRESS,
javaStrings.length);

    // 5. Copy the strings from on-heap to off-heap

    for (int i = 0; i < javaStrings.length; i++) {

        MemorySegment cString = offHeap.allocateUtf8String(javaStrings[i]);

        pointers.setAtIndex(ValueLayout.ADDRESS, i, cString);

    }

    // 6. Sort the off-heap data by calling the foreign function

    radixsort.invoke(pointers, javaStrings.length, MemorySegment.NULL,
'\0');
```

```
    // 7. Copy the (reordered) strings from off-heap to on-heap
    for (int i = 0; i < javaStrings.length; i++) {
        MemorySegment cString = pointers.getAtIndex(ValueLayout.ADDRESS,
i);
        javaStrings[i] = cString.getUtf8String(0);
    }
} // 8. All off-heap memory is deallocated here
assert Arrays.equals(javaStrings, new String[] {"car", "cat", "dog",
"mouse"});  // true
```

- The Linker interface enables both *downcalls* (calls from Java code to native code) and *upcalls* (calls from native code back to Java code).

Suppose we wish to downcall from Java to the `strlen` function defined in the standard C library:

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

Clients can link C functions using the *native linker* (see `Linker::nativeLinker`), a `Linker` implementation that conforms to the ABI determined by the OS and CPU on which the JVM is running. A downcall method handle that exposes `strlen` can be obtained as follows (the details of `FunctionDescriptor` will be described shortly):

```
Linker linker = Linker.nativeLinker();
MethodHandle strlen = linker.downcallHandle(
    linker.defaultLookup().find("strlen").get(),
    FunctionDescriptor.of(JAVA_LONG, ADDRESS)
);
```

Invoking the downcall method handle will run `strlen` and make its result available in Java. For the argument to `strlen`, we use a helper method to convert a Java string into an off-heap memory segment (using a confined arena) which is then passed by-reference:

```
try (Arena arena = Arena.openConfined()) {
    MemorySegment str = arena.allocateUtf8String("Hello");
    long len          = strlen.invoke(str);  // 5
}
```

Reference Link:

https://www.infoworld.com/article/3676699/jdk-20-the-new-features-in-java-20.html