Java Update
For the JCP EC

Aurelio Garcia-Ribeyro
Senior Director Product Management
Java Platform Group
Dec, 2023

Copyright © 2023, Oracle and/or its affiliates
Agenda

- Java Release Model – With most recent changes
- Future of Java – Active OpenJDK Projects
Java Release Model – Major Releases

Major Releases

Feature Releases

OpenJDK

Minor Release

Critical Patch Update
Java Release Model – Six month cadence

- JDK 7
- JDK 8
- JDK 11
- JDK 12
- JDK 13
- JDK 14
- JDK 15
- JDK 16
- JDK 17

Major Releases
Feature Releases
OpenJDK

2019 2020 2021 2022 2023 2024 2025 2026 2027 2028 2029 2030 2031
Java Release Model - NFTC Releases

JDK 8

JDK 11

JDK 17

JDK 21

JDK 22

JDK 23

JDK 24

JDK 25
With LTS Blinders

JDK 8
JDK 11
JDK 17
JDK 21

JEPs
56
120
74
37

Mar 2014
Sept 2018
Sept 2021
Sept 2023

4 + Years
3 years
2 Years

Copyright © 2023, Oracle and/or its affiliates
What did the six-month release cadence give us

- 12 On-time Feature Releases in 6 years, 3 of them with Long Term Support offered by Oracle
- No delayed features *
- Ability to adjust feature priority at any moment
- Higher quality releases
  - No irresistible need to slip in features under the wire
  - No overwhelming urge to backport new features to older releases
- Ability to incubate and/or preview features before making them final
- More engagement from Java Developers and System Administrators on non-final features
- Smaller features no longer wait for larger "release drivers"
- Faster adoption of new releases by tools and libraries

* Features are not scheduled into a release until they are ready
Agenda

• Java Release Model – With most recent changes
• Future of Java – Active OpenJDK Projects
# Active projects in the OpenJDK community

<table>
<thead>
<tr>
<th></th>
<th>Summary</th>
<th>Pain point</th>
<th>“Obvious” Competition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loom</td>
<td>Lightweight concurrency</td>
<td>“Threads are too expensive, don’t scale”</td>
<td>Go, Elixir</td>
</tr>
<tr>
<td>Amber</td>
<td>Right-sizing language ceremony</td>
<td>“Java is too verbose”</td>
<td>C#, Kotlin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Java is hard to teach”</td>
<td></td>
</tr>
<tr>
<td>ZGC</td>
<td>Sub-millisecond GC pauses</td>
<td>“GC pauses are too long”</td>
<td>C, Rust</td>
</tr>
<tr>
<td>Panama</td>
<td>Native code and memory interop</td>
<td>“Using native libraries is too hard”</td>
<td>Python, C</td>
</tr>
<tr>
<td></td>
<td>SIMD Vector support</td>
<td>“Numeric loops are too slow”</td>
<td></td>
</tr>
<tr>
<td>Leyden</td>
<td>Faster startup and warmup</td>
<td>“Java starts up too slowly”</td>
<td>Go</td>
</tr>
<tr>
<td>Valhalla</td>
<td>Value types and specialized generics</td>
<td>“Cache misses are too expensive”</td>
<td>C, C#</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Generics and primitives don’t mix”</td>
<td></td>
</tr>
<tr>
<td>Babylon</td>
<td>Foreign programming model interop</td>
<td>“Using GPUs is too hard”</td>
<td>LinQ, Julia</td>
</tr>
</tbody>
</table>
1 Core
1 Thread
100% cpu use
8 Cores
8 Threads
100% cpu

CPU Bound Application
8 Cores
8 Threads at 1/2 use
50% cpu
8 Cores
8 16 Threads at 1/2 use
100% cpu
8 Core
32 Threads at 1/4 use
100% cpu
8 Core
800 Threads at 1/100
100% cpu

IO Bound Application
But...

If it were this simple we wouldn't be talking about this right?
Pre-Loom: 1 Java Thread = 1 Operating System Thread

OS Threads are relatively expensive

- 2+kB of memory for metadata
- 1 MB+ of heap usage *

* Java Applications are limited to a few thousand threads by (mostly unused) memory

Java [OS] Threads are NOT enough for many IO Bound applications....
Threads are great!
• Readable, sequential code with understandable control flow
• Great debugging and serviceability, with comprehensible stack traces
• Natural unit of scheduling for operating systems

But, threads are heavyweight
• Expensive to create, megabyte-scale stacks, can only create a few thousand
• The convenient thread-per-task model can bump into this ceiling

Reactive frameworks promise better scaling, but at a significant cost
• Contorted programming model, hard to debug, incomprehensible stack traces
Virtual Threads – JDK 21

Loom introduces *virtual threads*
- Lighter threads, which don’t drag around huge thread stacks
- Pay-as-you-go stacks (minimum size 200-300 bytes), stored in the heap
- Scales to 1M+ concurrent connections on commodity hardware

Virtual threads are real threads!
- Implement `java.lang.Thread`, support `ThreadLocal`
- Clean stack traces, thread dumps
- Sequential-step debugging, profiling
- All your threaded code just works
- “Threads without the baggage”

“Carrier” OS threads transparently managed by a JVM scheduler
Most server requests spend more time in IO than compute

If requests are bound to threads, then we’ll likely run out of threads before we run out of CPU

- Run out of threads because we run out of memory
- Artificial throughput limit, raising cost of deployment
- With virtual threads, can keep taking load until CPU is saturated
Virtual threads are designed to model a single task, rather than a mechanism for running tasks

- Cheap enough to have a thread for every user request and async task
- Can keep the happy “thread per request” model and still scale
- Pooling them is counterproductive!

Obviates the need for complex and ill-fitting async or “reactive” frameworks

- No need to change paradigms, just make threads better

Virtual threads transparently suspended / resumed when they block

- Blocking APIs throughout the JDK retrofitted to be aware of virtual threads
### Active projects in the OpenJDK community

<table>
<thead>
<tr>
<th></th>
<th>Summary</th>
<th>Pain point</th>
<th>“Obvious” Competition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loom</td>
<td>Lightweight concurrency</td>
<td>“Threads are too expensive, don’t scale”</td>
<td>Go, Elixir</td>
</tr>
<tr>
<td>Amber</td>
<td>Right-sizing language ceremony</td>
<td>“Java is too verbose”</td>
<td>C#, Kotlin</td>
</tr>
<tr>
<td>Amber</td>
<td></td>
<td>“Java is hard to teach”</td>
<td></td>
</tr>
<tr>
<td>ZGC</td>
<td>Sub-millisecond GC pauses</td>
<td>“GC pauses are too long”</td>
<td>C, Rust</td>
</tr>
<tr>
<td>Panama</td>
<td>Native code and memory interop</td>
<td>“Using native libraries is too hard”</td>
<td>Python, C</td>
</tr>
<tr>
<td>Panama</td>
<td>SIMD Vector support</td>
<td>“Numeric loops are too slow”</td>
<td></td>
</tr>
<tr>
<td>Leyden</td>
<td>Faster startup and warmup</td>
<td>“Java starts up too slowly”</td>
<td>Go</td>
</tr>
<tr>
<td>Valhalla</td>
<td>Value types and specialized generics</td>
<td>“Cache misses are too expensive”</td>
<td>C, C#</td>
</tr>
<tr>
<td>Valhalla</td>
<td></td>
<td>“Generics and primitives don’t mix”</td>
<td></td>
</tr>
<tr>
<td>Babylon</td>
<td>Foreign programming model interop</td>
<td>“Using GPUs is too hard”</td>
<td>LinQ, Julia</td>
</tr>
</tbody>
</table>
Project Amber progress

JEPs delivered *
- Local Variable Type Inference – JDK 10
- Local Variable Syntax for Lambda Parameters - JDK 11
- Switch Expressions - JDK 14
- Text Blocks - JDK 15
- Pattern Matching for instanceof - JDK 16
- Records - JDK 16
- Sealed classes - JDK 17
- Record Patterns - JDK 21
- Pattern Matching for switch - JDK 21
- String Templates - Preview, JDK 21
- Unnamed Patterns and Variables - Preview, JDK 21
- Unnamed Classes and Instance Main Methods - Preview, JDK 21

Work in progress…
- Type patterns for primitive types
- Reconstruction expressions for records (and eventually, classes)
- Deconstruction patterns for classes and interfaces
- Relaxed constructor ordering

* Details on each of Amber's JEP can be found in this presentation's appendix
# Active projects in the OpenJDK community

<table>
<thead>
<tr>
<th></th>
<th>Summary</th>
<th>Pain point</th>
<th>“Obvious” Competition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loom</td>
<td>Lightweight concurrency</td>
<td>“Threads are too expensive, don’t scale”</td>
<td>Go, Elixir</td>
</tr>
<tr>
<td>Amber</td>
<td>Right-sizing language ceremony</td>
<td>“Java is too verbose”</td>
<td>C#, Kotlin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Java is hard to teach”</td>
<td></td>
</tr>
<tr>
<td>ZGC</td>
<td>Sub-millisecond GC pauses</td>
<td>“GC pauses are too long”</td>
<td>C, Rust</td>
</tr>
<tr>
<td>Panama</td>
<td>Native code and memory interop</td>
<td>“Using native libraries is too hard”</td>
<td>Python, C</td>
</tr>
<tr>
<td></td>
<td>SIMD Vector support</td>
<td>“Numeric loops are too slow”</td>
<td></td>
</tr>
<tr>
<td>Leyden</td>
<td>Faster startup and warmup</td>
<td>“Java starts up too slowly”</td>
<td>Go</td>
</tr>
<tr>
<td>Valhalla</td>
<td>Value types and specialized generics</td>
<td>“Cache misses are too expensive”</td>
<td>C, C#</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Generics and primitives don’t mix”</td>
<td></td>
</tr>
<tr>
<td>Babylon</td>
<td>Foreign programming model interop</td>
<td>“Using GPUs is too hard”</td>
<td>LinQ, Julia</td>
</tr>
</tbody>
</table>

Copyright © 2023, Oracle and/or its affiliates

December 2023
**ZGC**

The “Z” garbage collector was introduced in JDK 15

- **Terabyte-scale heaps, sub-millisecond pauses**
  - Pauses do not scale with heap size or live-set
  - All the buzzwords – Concurrent, Parallel, Compacting, Region-based, Numa-Aware, Auto-tuning

- No longer have to worry about GC pauses

What’s the catch?

- The cost of this near-pauseless operation is about a 2% throughput reduction
- And, uses more memory

Copyright © 2023, Oracle and/or its affiliates

December 2023
GC pause times

Lower is better

<table>
<thead>
<tr>
<th>Milliseconds</th>
<th>G1</th>
<th>ZGC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>200</td>
<td>95</td>
</tr>
<tr>
<td>95th Percentile</td>
<td>300</td>
<td>150</td>
</tr>
<tr>
<td>99th Percentile</td>
<td>400</td>
<td>250</td>
</tr>
<tr>
<td>99.9th Percentile</td>
<td>500</td>
<td>300</td>
</tr>
<tr>
<td>Max</td>
<td>600</td>
<td>450</td>
</tr>
</tbody>
</table>
The chart illustrates measured GC pause times for G1 and ZGC generations. The y-axis represents time in microseconds, and the x-axis categorizes the collections as G1 and ZGC. The chart indicates that ZGC consistently shows lower GC pause times compared to G1, with specific metrics highlighted at average, 95th percentile, 99th percentile, and 99.9th percentile. The goal for both is lower values, indicating improved performance and responsiveness.
GC pause times

Lower is better

microseconds

G1

ZGC

Average
95th Percentile
99th Percentile
99.9th Percentile
Max
ZGC Improvements Over Time

GC Pause Times

(Lower is better)

JDK 11

JDK 15

JDK 17

GC Pause Times (ms)

128G Heap

40 Hyper-threads (Intel)
Generational ZGC

ZGC has been here for a while
  • But has been single-generation
JDK 21 adds *generational* capability to ZGC
  • Generational ZGC offers the same throughput with significantly less memory
  • 75% less memory for same throughput on Cassandra benchmark
## Active projects in the OpenJDK community

<table>
<thead>
<tr>
<th></th>
<th>Summary</th>
<th>Pain point</th>
<th>“Obvious” Competition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Loom</strong></td>
<td>Lightweight concurrency</td>
<td>“Threads are too expensive, don’t scale”</td>
<td>Go, Elixir</td>
</tr>
<tr>
<td><strong>Amber</strong></td>
<td>Right-sizing language ceremony</td>
<td>“Java is too verbose”</td>
<td>C#, Kotlin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Java is hard to teach”</td>
<td></td>
</tr>
<tr>
<td><strong>ZGC</strong></td>
<td>Sub-millisecond GC pauses</td>
<td>“GC pauses are too long”</td>
<td>C, Rust</td>
</tr>
<tr>
<td><strong>Panama</strong></td>
<td>Native code and memory interop</td>
<td>“Using native libraries is too hard”</td>
<td>Python, C</td>
</tr>
<tr>
<td></td>
<td>SIMD Vector support</td>
<td>“Numeric loops are too slow”</td>
<td></td>
</tr>
<tr>
<td><strong>Leyden</strong></td>
<td>Faster startup and warmup</td>
<td>“Java starts up too slowly”</td>
<td>Go</td>
</tr>
<tr>
<td><strong>Valhalla</strong></td>
<td>Value types and specialized generics</td>
<td>“Cache misses are too expensive”</td>
<td>C, C#</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Generics and primitives don’t mix”</td>
<td></td>
</tr>
<tr>
<td><strong>Babylon</strong></td>
<td>Foreign programming model interop</td>
<td>“Using GPUs is too hard”</td>
<td>LinQ, Julia</td>
</tr>
</tbody>
</table>
Project Panama

Project Panama is (partly) about better access to native (off-heap) memory and native code.

In the early days of Java, native code was actively discouraged:
- Pure Java FTW!

But, there are some great native libraries that won’t be – and don’t need to be – rewritten in Java:
- Off-CPU computing (Cuda, OpenCL)
- Machine learning (Blas, Blis, ONNX, Tensorflow)
- Graphics (OpenGL, DirectX, Vulkan)
- Many others (CRIU, fuse, io_uring, OpenSSL, V8, SQLite, ucx)
Project Panama

We can access native libraries with JNI, but it is painful to use, unsafe
• Code in a brittle combination of Java and C
• Expensive to maintain, error-prone, poor error checking
• JNI errors can crash the JVM

Java developers often resort to ByteBuffer (or Unsafe) to manage “big data” off-heap
• ByteBuffers are clumsy, limited to 2GB
• Unsafe is, well, unsafe (and will eventually go away)

Panama is built for safety and performance from the ground up
• Highly optimized temporal and spatial bounds checking
Project Panama

Panama gives us a better, safer, performant alternative to JNI, ByteBuffer, and Unsafe
• Final preview in JDK 21
• Based on newer, more optimizable VM facilities (MethodHandle, VarHandle)
• Safe, supported alternative for off-heap operations currently in Unsafe

Panama makes it easy to wrap native libraries with Java bindings and access them from Java code
• Bring native libraries into the Java ecosystem
• Encourage building and distributing Java bindings for popular native libraries
<table>
<thead>
<tr>
<th>Summary</th>
<th>Pain point</th>
<th>“Obvious” Competition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loom</td>
<td>“Threads are too expensive, don’t scale”</td>
<td>Go, Elixir</td>
</tr>
<tr>
<td>Lightweight concurrency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amber</td>
<td>“Java is too verbose”</td>
<td>C#, Kotlin</td>
</tr>
<tr>
<td>Right-sizing language ceremony</td>
<td>“Java is hard to teach”</td>
<td></td>
</tr>
<tr>
<td>ZGC</td>
<td>“GC pauses are too long”</td>
<td>C, Rust</td>
</tr>
<tr>
<td>Sub-millisecond GC pauses</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Panama</td>
<td>“Using native libraries is too hard”</td>
<td>Python, C</td>
</tr>
<tr>
<td>Native code and memory interop</td>
<td>“Numeric loops are too slow”</td>
<td></td>
</tr>
<tr>
<td>SIMD Vector support</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leyden</td>
<td>“Java starts up too slowly”</td>
<td>Go</td>
</tr>
<tr>
<td>Faster startup and warmup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Valhalla</td>
<td>“Cache misses are too expensive”</td>
<td>C, C#</td>
</tr>
<tr>
<td>Value types and specialized generics</td>
<td>“Generics and primitives don’t mix”</td>
<td></td>
</tr>
<tr>
<td>Babylon</td>
<td>“Using GPUs is too hard”</td>
<td>LinQ, Julia</td>
</tr>
<tr>
<td>Foreign programming model interop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active projects in the OpenJDK community</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A look ahead – Project Leyden

Project Leyden is about improving the *startup* and *warmup* of Java applications

- *Startup* is the time it takes to get to the first useful unit of work
- *Warmup* is the time it takes for the application to reach peak performance

Java has historically favored long-term peak performance over startup

- A good tradeoff for many applications

Java does a lot of work at startup – processing classfiles, interpretation, profile gathering, callsite linkage, JIT compilation

- Dynamic compilation produces better code than static compilation
- Good peak performance, but at the cost of startup and warmup
Startup and warmup

Class initialization activity unique to first iteration (CPU milliseconds)

Online JIT activity for warmup (CPU seconds)
To improve startup, push the first point down

To improve warmup, push the entire curve down
Shifting computation

To push these curves down, we have to shift work off the critical path
  • Could shift work later in time, such as by laziness
  • Could shift work earlier in time, from run time to build time

The JDK already employs many computation-shifting techniques
  • Constant folding, garbage collection, class loading, JIT compilation

Let’s shift more!
  • Adapted the existing JIT compilers and Class Data Sharing (CDS) to precompute and store compilation profiles, compiled code, callsite linkage
  • No changes to user code, no loss of dynamism
  • Just a “training run” at build time
Experimental Leyden result: javac

- Repeatedly compile 100 small source files
- 2x startup improvement, significant warmup improvement
- No change to existing code
Experimental Leyden result: Spring Boot

Startup time (s)

Spring Boot “Pet Clinic”
4.1x startup improvement with no change to existing code
## Active projects in the OpenJDK community

<table>
<thead>
<tr>
<th>Project</th>
<th>Summary</th>
<th>Pain point</th>
<th>“Obvious” Competition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loom</td>
<td>Lightweight concurrency</td>
<td>“Threads are too expensive, don’t scale”</td>
<td>Go, Elixir</td>
</tr>
<tr>
<td>Amber</td>
<td>Right-sizing language ceremony</td>
<td>“Java is too verbose”</td>
<td>C#, Kotlin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Java is hard to teach”</td>
<td></td>
</tr>
<tr>
<td>ZGC</td>
<td>Sub-millisecond GC pauses</td>
<td>“GC pauses are too long”</td>
<td>C, Rust</td>
</tr>
<tr>
<td>Panama</td>
<td>Native code and memory interop</td>
<td>“Using native libraries is too hard”</td>
<td>Python, C</td>
</tr>
<tr>
<td></td>
<td>SIMD Vector support</td>
<td>“Numeric loops are too slow”</td>
<td></td>
</tr>
<tr>
<td>Leyden</td>
<td>Faster startup and warmup</td>
<td>“Java starts up too slowly”</td>
<td>Go</td>
</tr>
<tr>
<td>Valhalla</td>
<td>Value types and specialized generics</td>
<td>“Cache misses are too expensive”</td>
<td>C, C#</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Generics and primitives don’t mix”</td>
<td></td>
</tr>
<tr>
<td>Babylon</td>
<td>Foreign programming model interop</td>
<td>“Using GPUs is too hard”</td>
<td>LinQ, Julia</td>
</tr>
</tbody>
</table>
And to conclude..
Shameless plug to ask for your help in evolving Java while protecting current programs

Test... test... test

- Preview/Incubator features
  Even if only to say "no issues"

- Early Access of upcoming Feature Versions
  You can test JDK 22 EA today

- Early Access of Project Builds
Appendix

Project Amber Features
Local-Variable Type Inference  JDK 10

URL url = new URL("http://www.oracle.com/");

URLConnection con = url.openConnection();

InputStreamReader is = new InputStreamReader(con.getInputStream());

Reader reader = new BufferedReader(is);
Local-Variable Type Inference  JDK 10

```java
var url = new URL("http://www.oracle.com/");

var con = url.openConnection();

var is = new InputStreamReader(con.getInputStream());

var reader = new BufferedReader(is);
```

Style Guide: https://openjdk.java.net/projects/amber/LVTIstyle.html
int numLetters;
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        numLetters = 6;
        break;
    case TUESDAY:
        numLetters = 7;
        break;
    case THURSDAY:
    case SATURDAY:
        numLetters = 8;
        break;
    case WEDNESDAY:
        numLetters = 9;
        break;
    default:
        throw new IllegalArgumentException("Not a day: " + day);
}
return numLetters;
Switch Expressions

JDK 14

```java
return switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY -> 7;
    case THURSDAY, SATURDAY -> 8;
    case WEDNESDAY -> 9;
};
```
var html += "<tr><br>" +
    "<td>Retweets: " + t.getRetweetCount() + "</td><br>" +
    "<td>Likes: " + t.getLikeCount() + "</td><br>" +
    "<tr><br>";
var html += """
<tr>
    <td>Retweets: %s</td>
    <td>Likes: %s</td>
</tr>
""".formatted(t.getRetweetCount(),
            t.getLikeCount());
var html += """

............ <tr>

............ <td>Retweets: %s</td>

............ <td>Likes: %s</td>

............ <tr>

"""".formatted(t.getRetweetCount(), t.getLikeCount());
if (obj instanceof String) {
    String s = (String) obj;
    // use s
}

1) a test: is obj a String
2) declaration of a new variable s
3) casting of obj to String into variable s
if (obj instanceof String s) {
    // use s
}
Pattern Matching for `instanceof` JDK 16

```java
if (obj instanceof String s) {
    // use s
} else {
    // s is out of scope here!
}
```
class Point {
    final int x;
    final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Point point = (Point) o;
        if (x != point.x) return false;
        return y == point.y;
    }

    @Override
    public int hashCode() {
        int result = x;
        result = 31 * result + y;
        return result;
    }

    @Override
    public String toString() {
        return "Point{x=", x + "}, y=" + y + "}';
    }

    public int x() { return x; }
    public int y() { return y; }
}
record Point (int x, int y) {}
Sealed Types (classes and interfaces)  JDK 17

package com.example.geometry;

public abstract sealed class Shape permits Circle, Rectangle, Square {...}

public final class Circle extends Shape {...}

public sealed class Rectangle extends Shape permits TransparentRectangle, FilledRectangle {...}
public final class TransparentRectangle extends Rectangle {...}
public final class FilledRectangle extends Rectangle {...}

public non-sealed class Square extends Shape {...}
Record Patterns – JDK 21

Before

```java
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);
    }
}
```
Record Patterns

After

```java
record Point(int x, int y) { }

static void printSum(Object obj) {
  if (obj instanceof Point(int x, int y) {
    System.out.println(x+y);
  }
}
```
More complicated Object Graphs

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

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

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

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

Type Inference

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

static void printUpperLeftColoredPoint(Rectangle r) {
    if (r instanceof Rectangle(ColoredPoint (var p, Color c),
                               var lr)) {
        System.out.println(c);
    }
}
```
Pattern Matching for `switch` - JDK 21

**JEP 441**

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

Allows an expression to be tested against a number of patterns, each with a specific action, so that complex data-oriented queries can be expressed concisely and safely.
Pattern Matching for switch

Before

```java
String formatter(Object o) {
    String formatted = "unknown";
    if (o instanceof Integer i) {
        formatted = String.format("int %d", i);
    } else if (o instanceof Long l) {
        formatted = String.format("long %d", l);
    } else if (o instanceof Double d) {
        formatted = String.format("double %f", d);
    } else if (o instanceof String s) {
        formatted = String.format("String %s", s);
    }
    return formatted;
}
```
Pattern Matching for `switch`

After

```java
String formatter(Object o) {
    return switch (o) {
        case null -> "null";
        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 -> o.toString();
    }
}
```
static void test(Object o) {
    switch (o) {
    case String s:
        if (s.length() == 1)
            {//handle single character strings}
        else
            {//handle all other strings}
        break;
    ...
    }
}
Pattern Matching for switch – Optional when clause

```java
static void test(Object o) {
    switch (o) {
        case String s when s.length() == 1 -> //single character strings
        case String s -> //all other strings
            ...
    }
}
```
String Templates (Preview)

JEP 430

String templates complement Java's existing string literals and text blocks by coupling literal text with embedded expressions and template processors to produce specialized results.

Goals

• Simplify how to express strings that include values computed at run time
• Enhance the readability of expressions that mix text and expressions
• Improve the security of programs that compose strings from user-provided values and pass them to other systems
String Templates - Motivation

String s = x + " + " + y + " equals " + (x + y);
//hard to read

String s = new StringBuilder(
   .append(x)
   .append(" + ")
   .append(y)
   .append(" equals ")
   .append(x + y)
   .toString();
//verbose

String s = String.format("%1$d + %2$d equals %3$d", x, y, x + y);
String t = "%1$d + %2$d equals %3$d".formatted(x, y, x + y);
//invites arity and type mismatch

MessageFormat mf = new MessageFormat("{0} + {1} equals {2}"");
String s = mf.format(x, y, x + y);
//too much ceremony, unfamiliar syntax
Why not add String Interpolation?

String Interpolation offers string literals that combine embedded expression as well as literal text.

```javascript
const title = "My Web Page";
const text = "Hello, world";

var html = `\n  <html>
    <head>
      <title>${title}</title>
    </head>
    <body>
      <p>${text}</p>
    </body>
  </html>`;
```
String Interpolation

Simplified assumptions meet real world

https://imgs.xkcd.com/comics/exploits_of_a_mom.png
As easy to use... but better

A little more work gets you a lot more safety

String Templates allow domain-specific validation and transformations to be built into the Template

```java
String name = "Robert"); DROP TABLE Students; --";
String query = "INSERT INTO Students VALUES ('\{name\}')";
```

With String Interpolation:

- `INSERT INTO Students VALUES ('Robert'); DROP TABLE Students; --')`

Using String Templates:

- `INSERT INTO Students VALUES ('Robert\'); DROP TABLE Students; --')`
String Templates

Description

```java
String name = "Joan";

String info = STR."My name is \{name}";

assert info.equals("My name is Joan");
```
String Templates

Description

1) Template Processor

String info = STR."My name is \{name}\";

2) Dot (U+002E)

3) Template with a embedded expression
String Templates

STR Template Processor

```java
int x = 10, y = 20;
String s = STR."\{x\} + \{y\} = \{x + y\}"
// "10 + 20 = 30"

String t = STR."Access at \{req.date\} \{req.time\} from \{req.ipAddress\}";
// "Access at 2022-03-25 15:34 from 8.8.8.8"
```
```
String time = STR."The time is \{ \\
// The java.time.format package is very useful
DateTimeFormatter
 .ofPattern("HH:mm:ss")
 .format(LocalTime.now())
} right now";

// "The time is 12:34:56 right now"
```
String Templates

String title = "My Web Page"; String text = "Hello, world";

String html = STR.""
   <html>
      <head>
         <title>\{title\}</title>
      </head>
      <body>
         <p>\{text\}</p>
      </body>
   </html>
.""
String Templates

The FMT template processor

FMT is like STR but it also interprets format specifiers to the left of the embedded expressions. Format specifiers are the same as those defined in java.util.Formatter.

```java
double gallons = 12.34
double pricePerGallon = 3.865

FMT."Purchasing %1.2f\{gallons\} gallons of gasoline at $%1.3f\{pricePerGallon\} would cost $%1.2f\{gallons * pricePerGallon\}"

// "Purchasing 12.34 gallons of gasoline at $3.865 per gallon would cost $47.69"
Unnamed Patterns and Variables (Preview)

JEP 443

Enhance the Java language with unnamed patterns, which match a record component without stating the component's name or type, and unnamed variables, which can be initialized but not used. Both are denoted by an underscore character: _
Pattern Matching with unused variables

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

static void printUpperLeftColoredPoint(Rectangle r) {
    if (r instanceof Rectangle(ColoredPoint (var p, Color c),
                                var lr)) {
        System.out.println(c);
    }
}
```
Pattern Matching with Unnamed Patterns

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

static void printUpperLeftColoredPoint(Rectangle r) {
    if (r instanceof Rectangle(ColoredPoint (_, Color c), _)) {
        System.out.println(c);
    }
}
```
Unnamed Variables

String s = ...;

try {
    int i = Integer.parseInt(s);
    ... i ...
} catch (NumberFormatException ex) {
    System.out.println("Bad number: " + s);
}
Unnamed Variables

String s = ...;

try {
    int i = Integer.parseInt(s);
    ... i ...
} catch (NumberFormatException _) {
    System.out.println("Bad number: " + s);
}
Unnamed Classes and Instance Main Methods (Preview)

JEP 445

Make it possible for students to write their first programs without needing to understand language features designed for large programs.
Unnamed Classes and Instance Main Methods

My first Java program

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

"Ignore all of this... you will understand it later"
Allow instance main methods

My first Java program

class HelloWorld {
    void main() {
        System.out.println("Hello, World!");
    }
}

Introduce unnamed classes

My Java first program

```java
void main() {
    System.out.println("Hello, World!");
}
```
Introduce unnamed classes

My Java first program

class <unnamed> {
    void main() {
        System.out.println("Hello, World!");
    }
}

Introduce unnamed classes

My Java first program

class <unnamed> { 
    String greeting() { return "Hello, World!"; }
    void main() {
        System.out.println(greeting());
    }
}
Introduce unnamed classes

My Java first program

class <unnamed> {
    String greeting = "Hello, World!";
    void main() {
        System.out.println(greeting);
    }
}