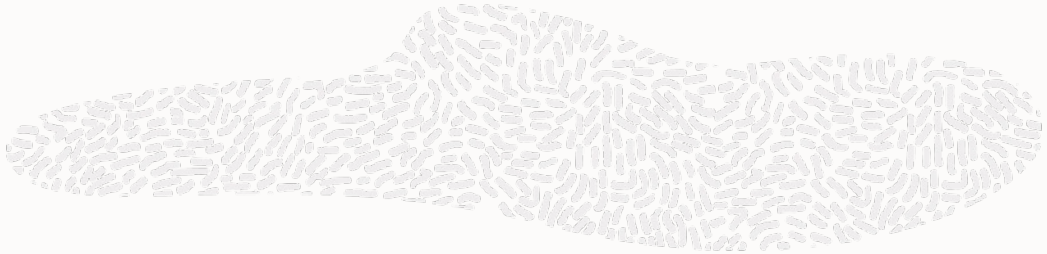
ORACLE

Java Update

For the JCP EC

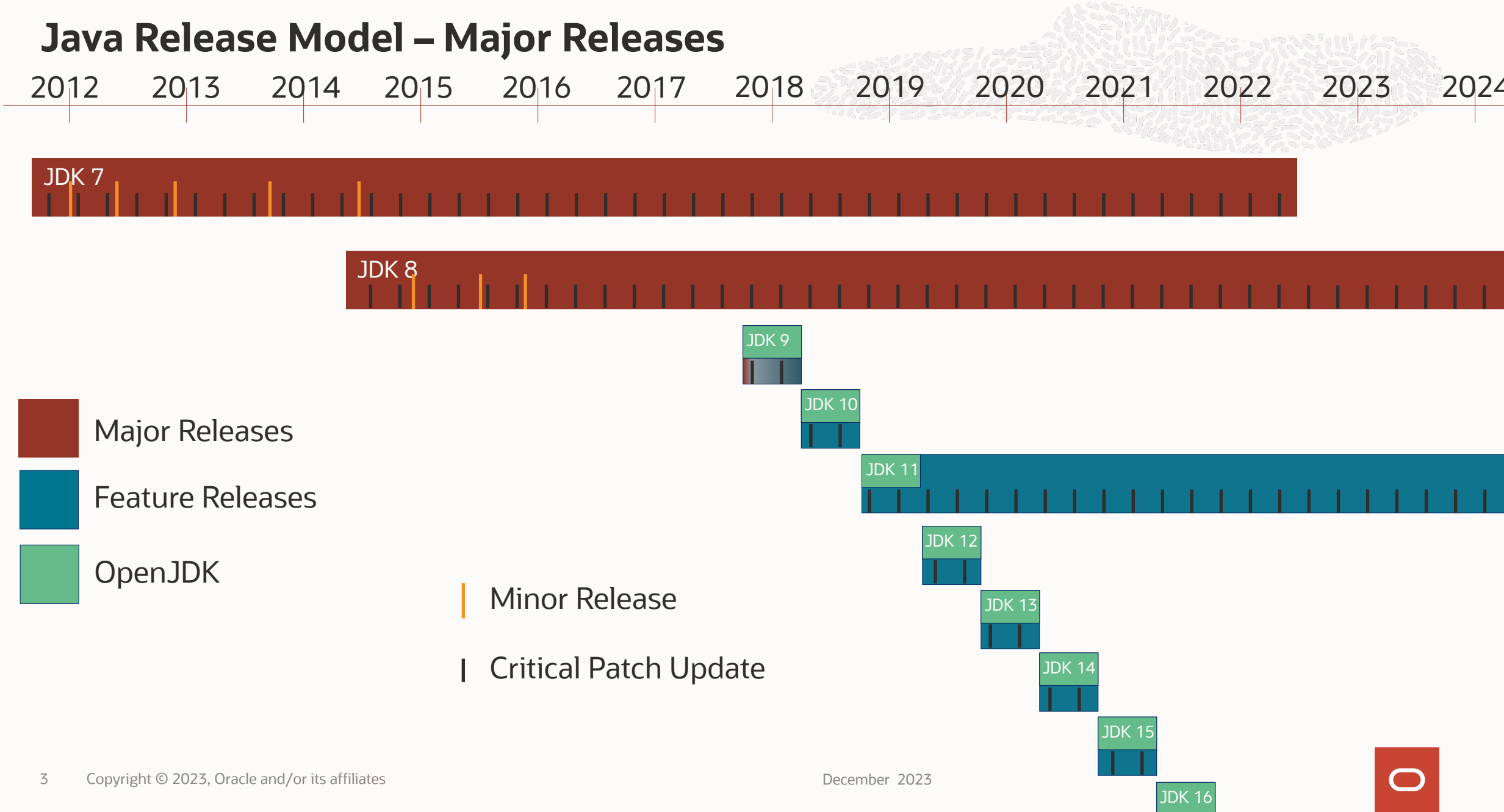
Aurelio Garcia-Ribeyro
Senior Director Product Management
Java Platform Group
Dec, 2023

Agenda

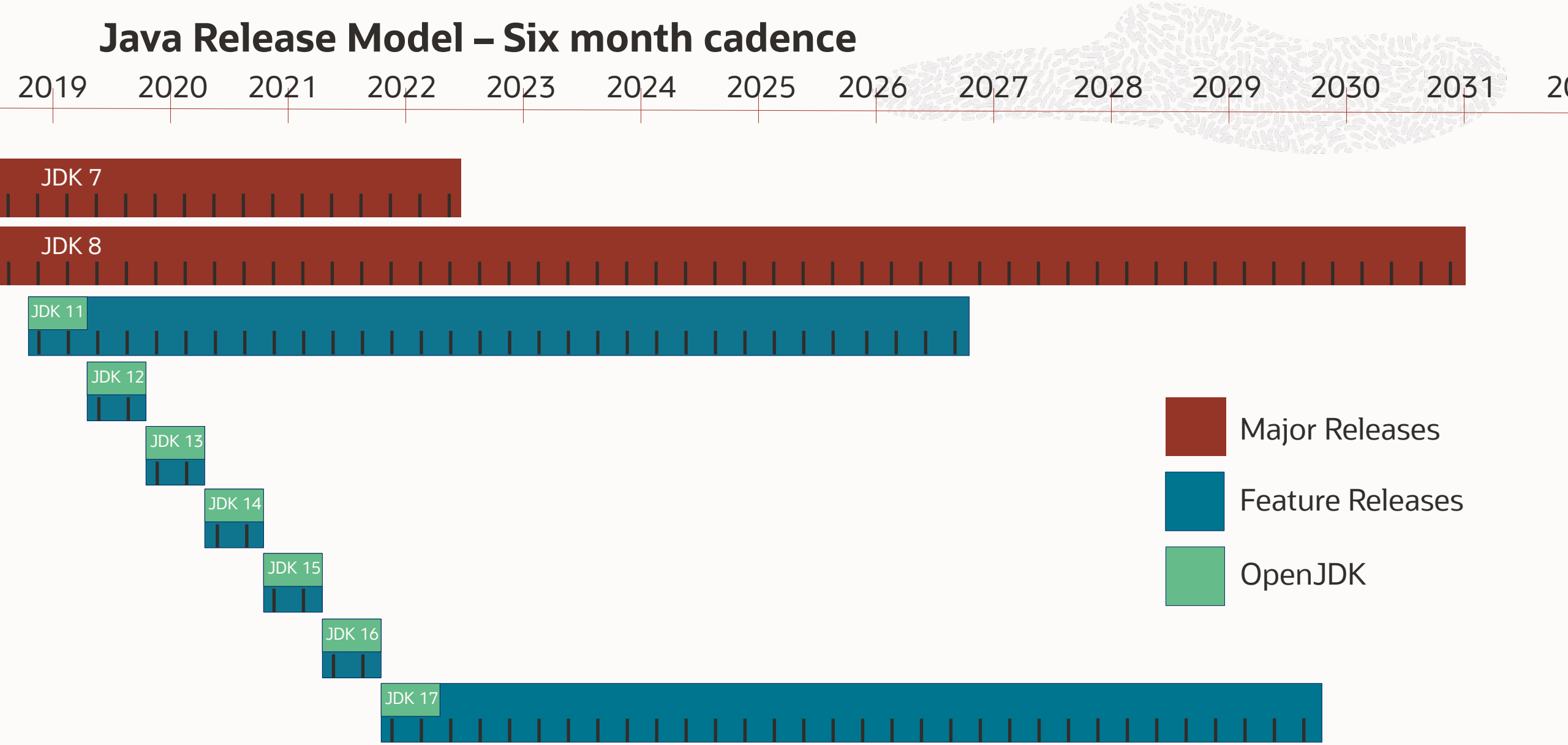


- Java Release Model – With most recent changes
- Future of Java – Active OpenJDK Projects

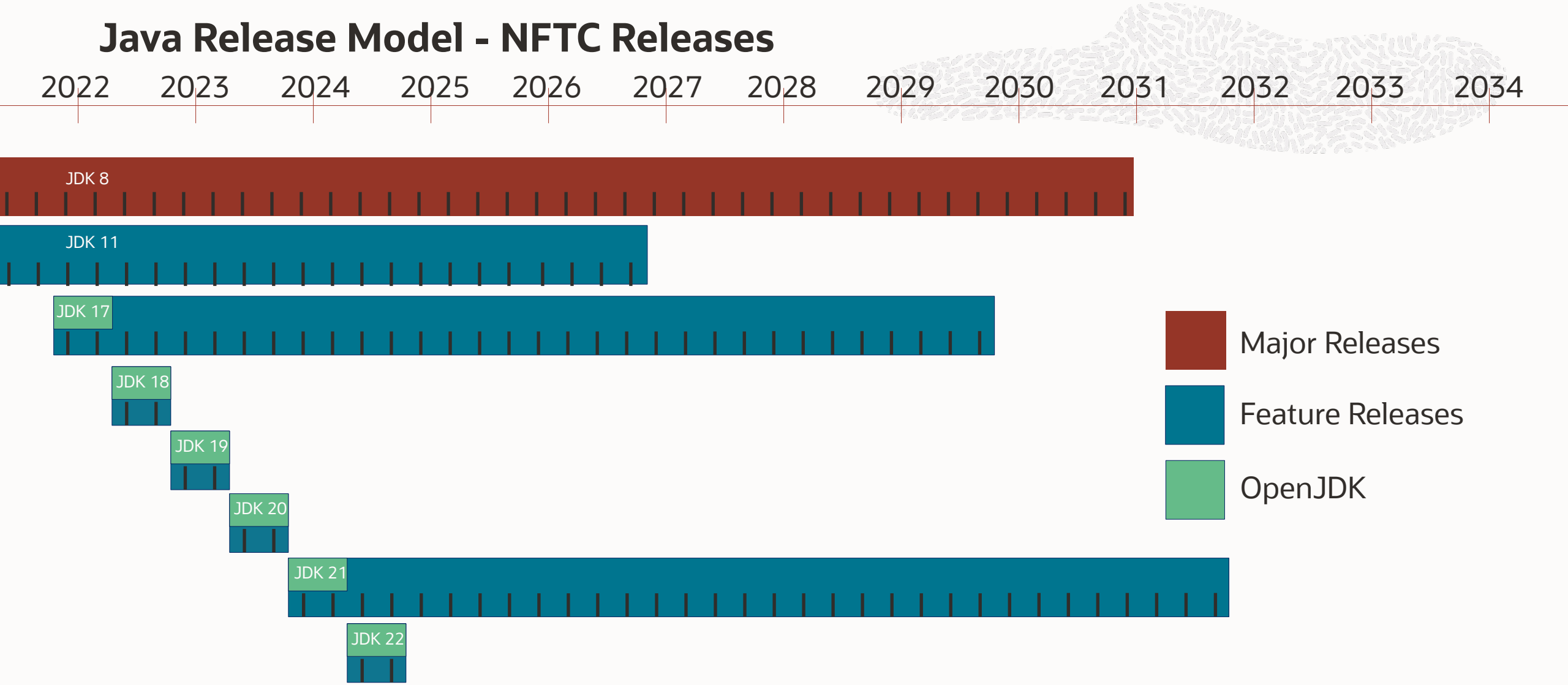
Java Release Model – Major Releases



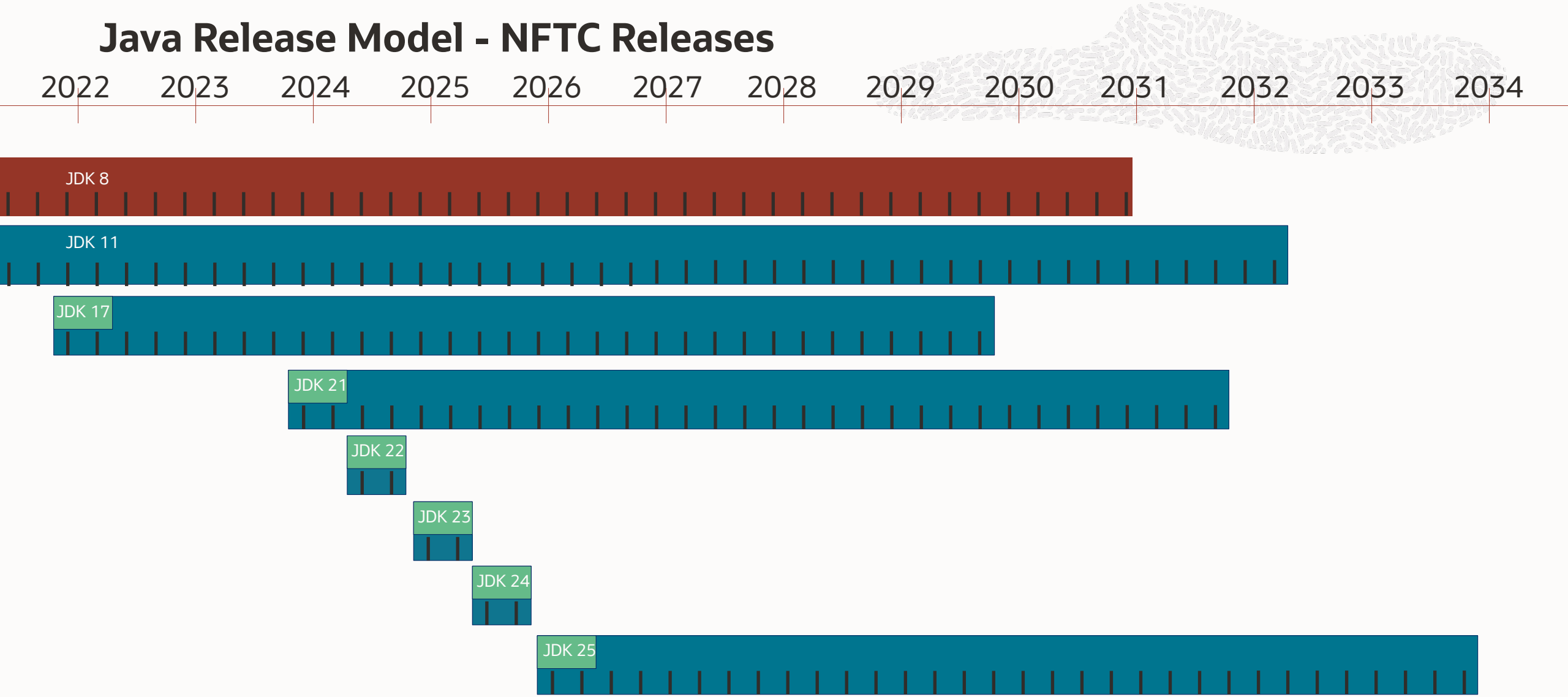
Java Release Model – Six month cadence



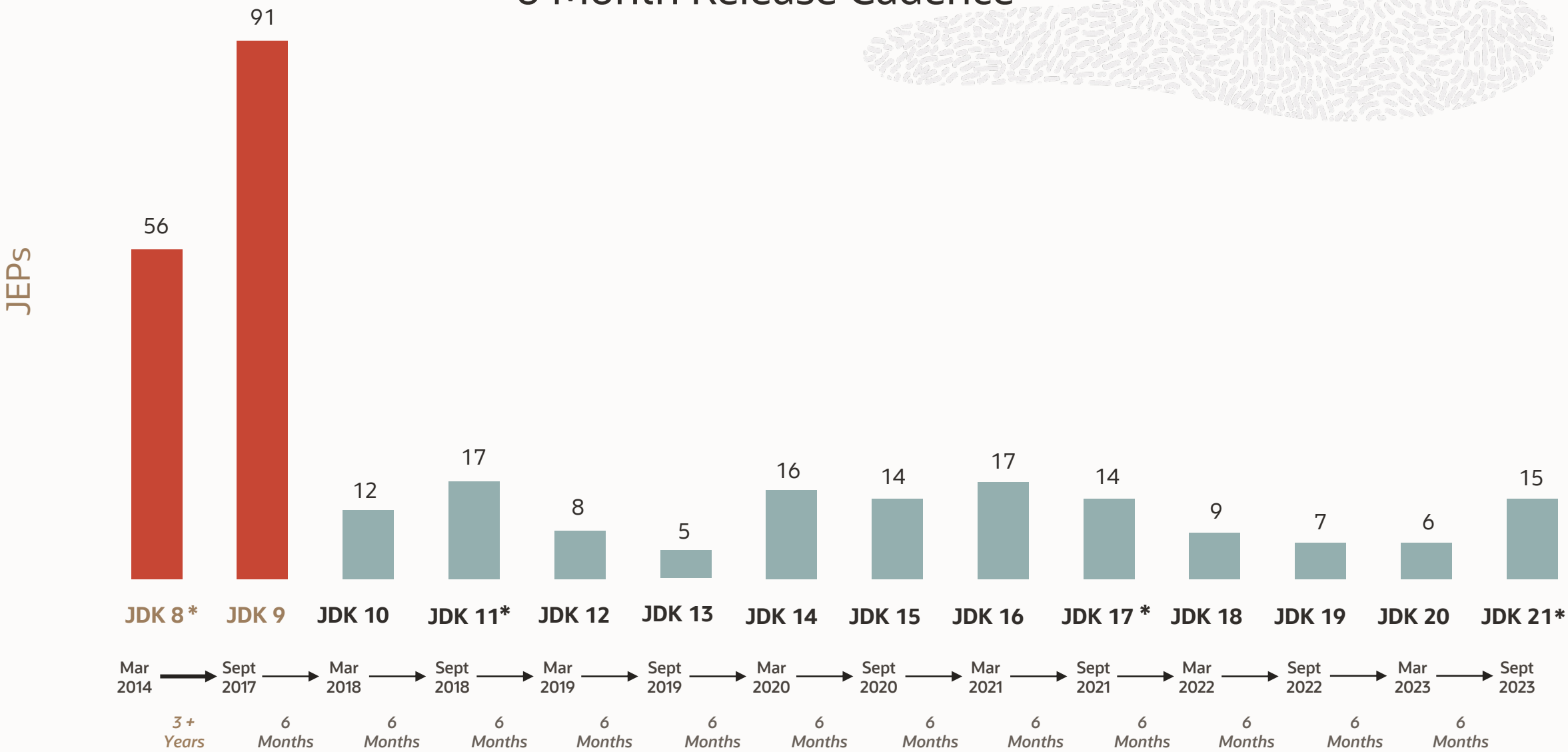
Java Release Model - NFTC Releases



Java Release Model - NFTC Releases

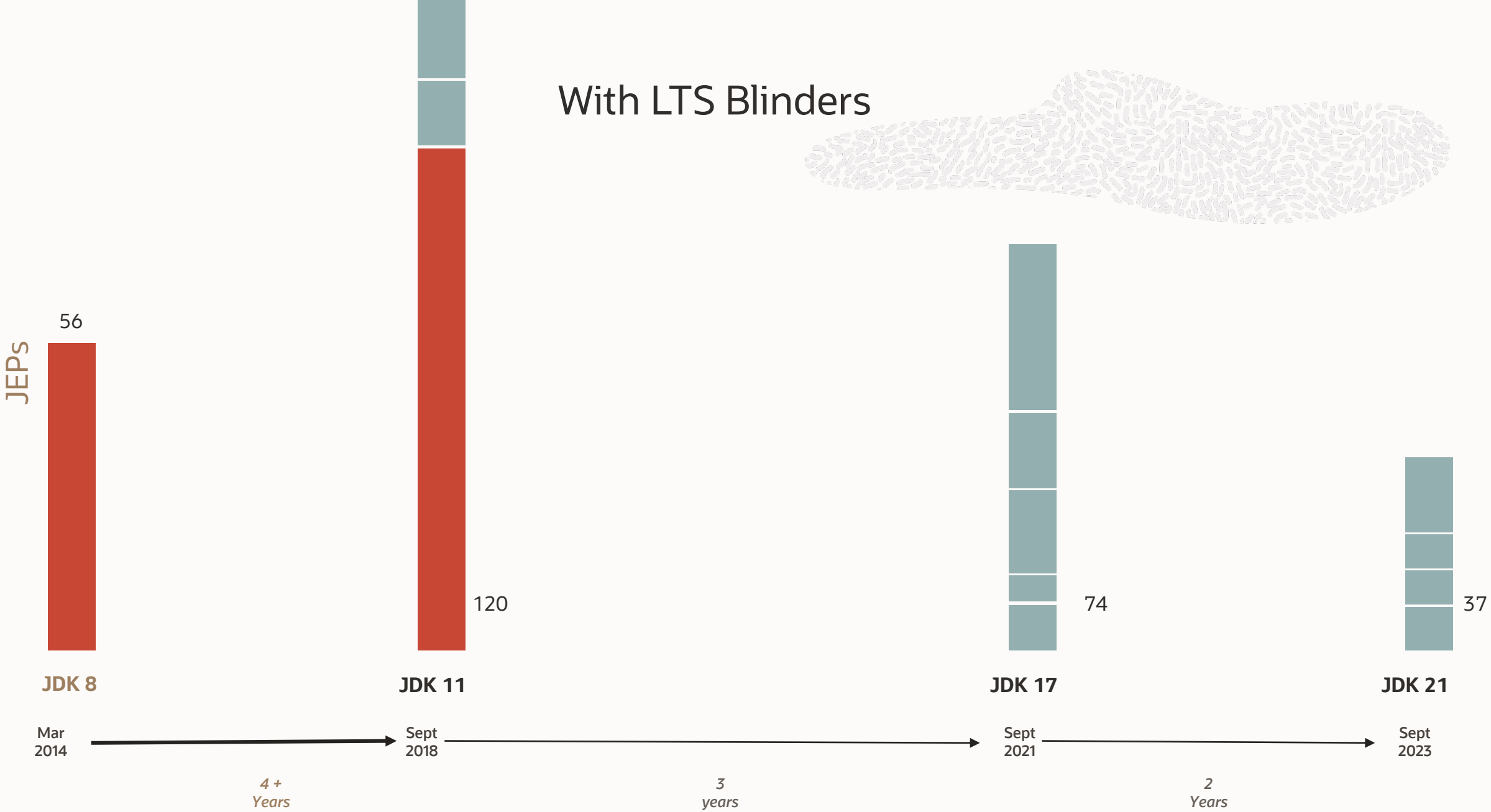


6 Month Release Cadence



* Oracle offers LTS for this version

With LTS Blinders

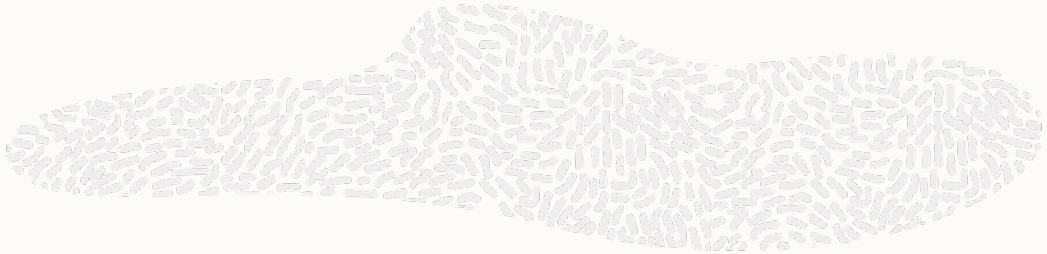


What did the six-month release cadence give us

- 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

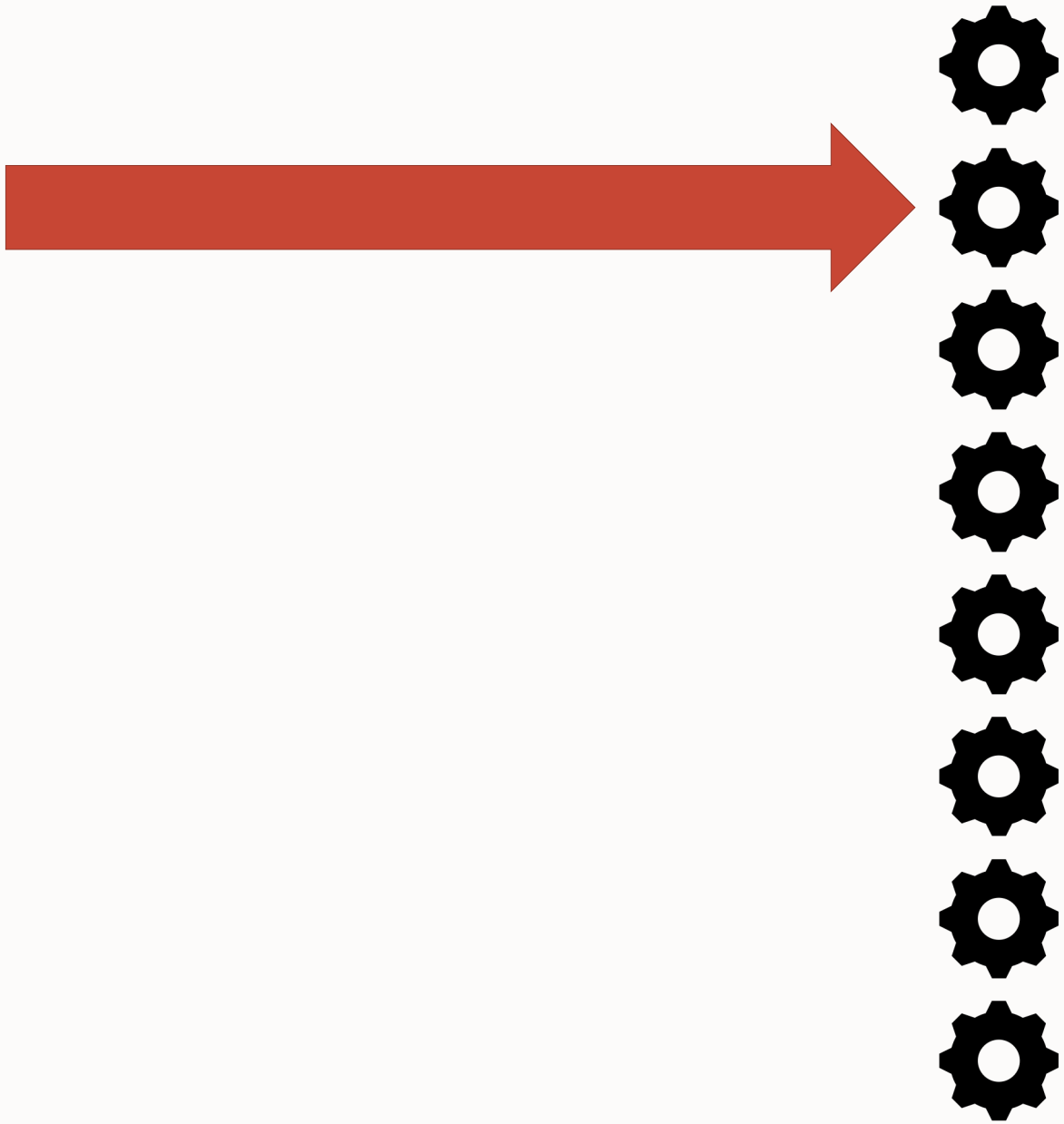


- Java Release Model – With most recent changes
- Future of Java – Active OpenJDK Projects

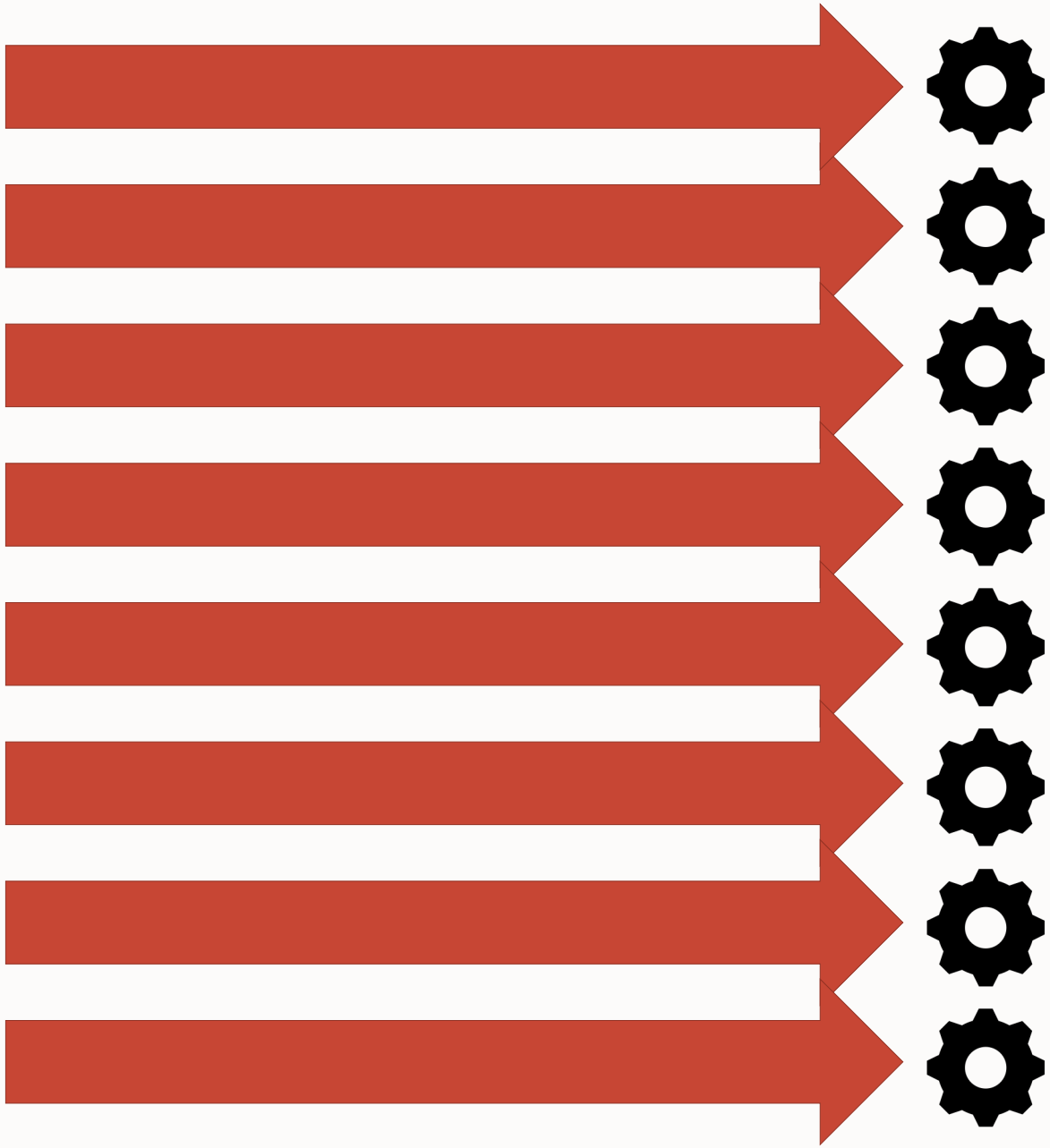
Active projects in the OpenJDK community

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

1 Core
1 Thread
100% cpu use

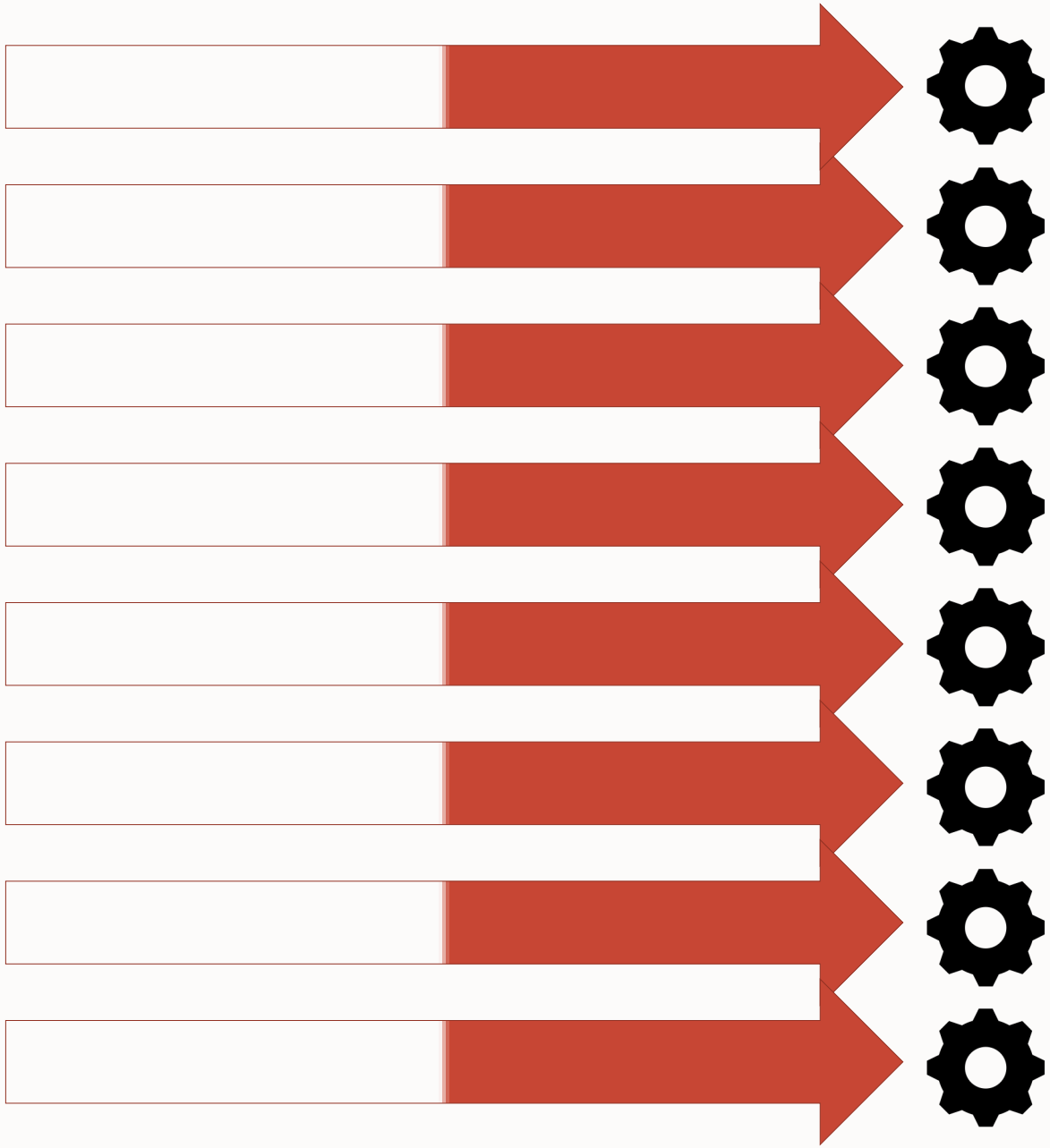


8 Cores
? Threads
100% cpu

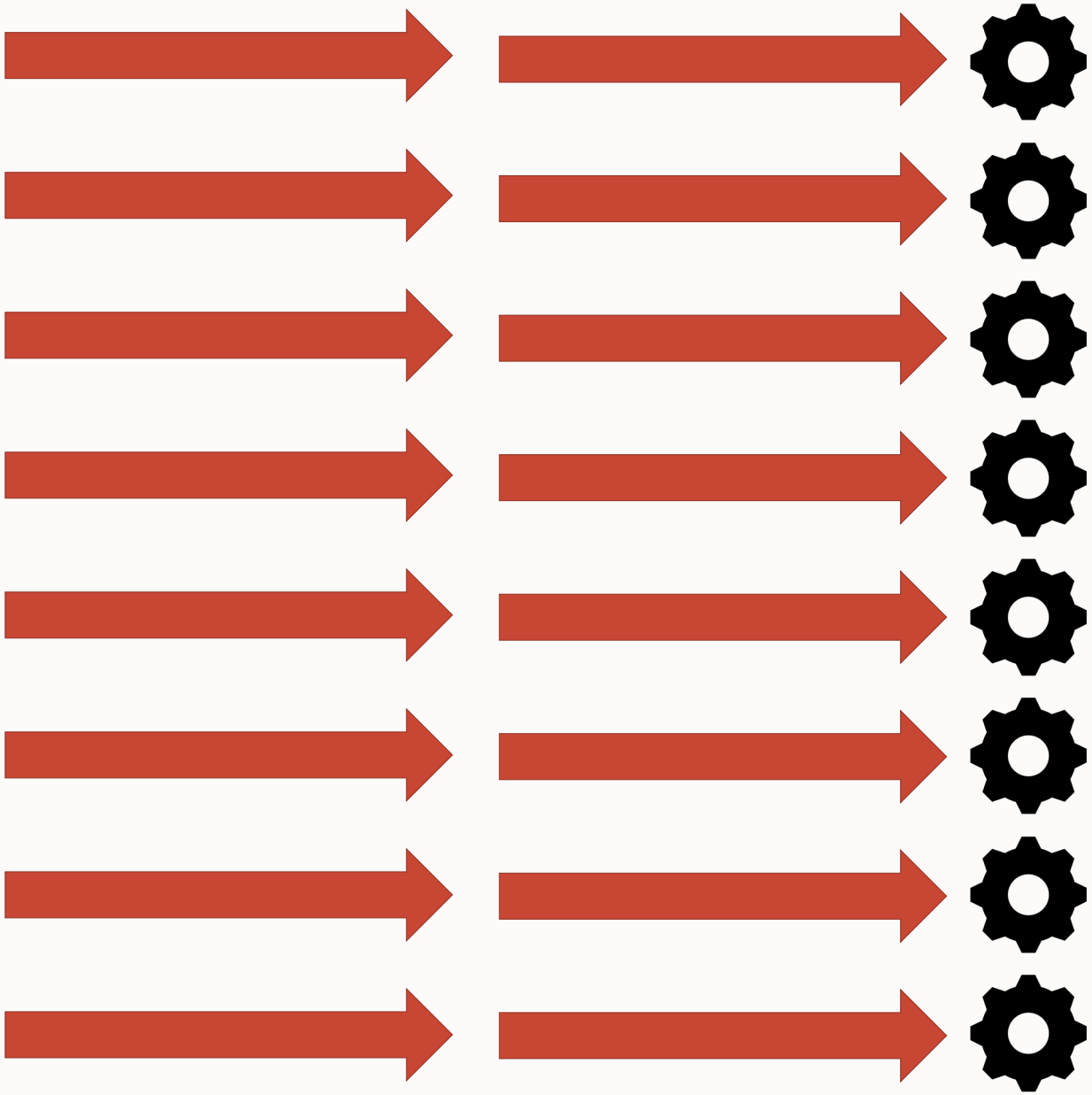


8 Cores
8 Threads
100% cpu

CPU Bound Application



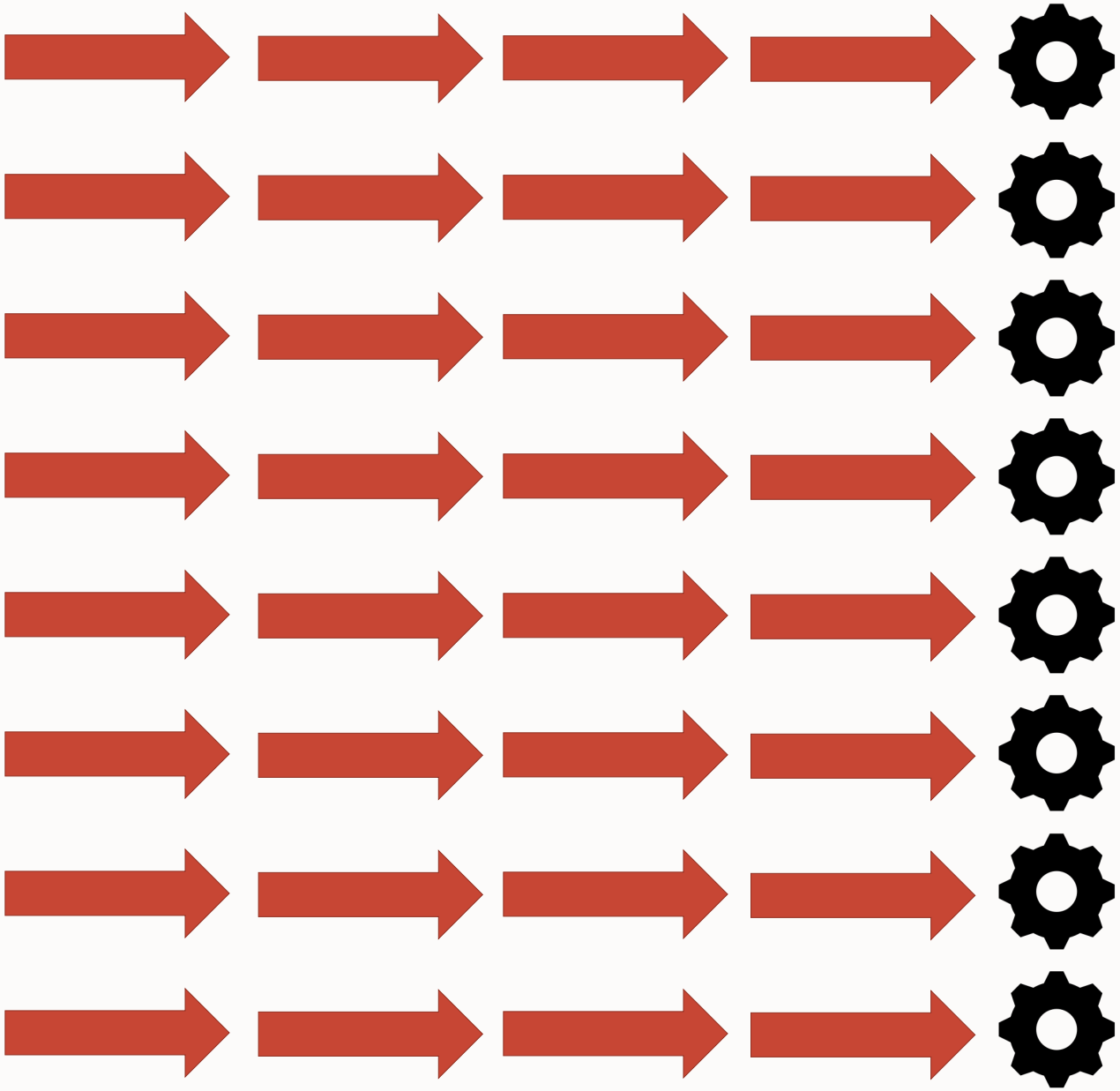
8 Cores
8 Threads at 1/2 use
50% cpu



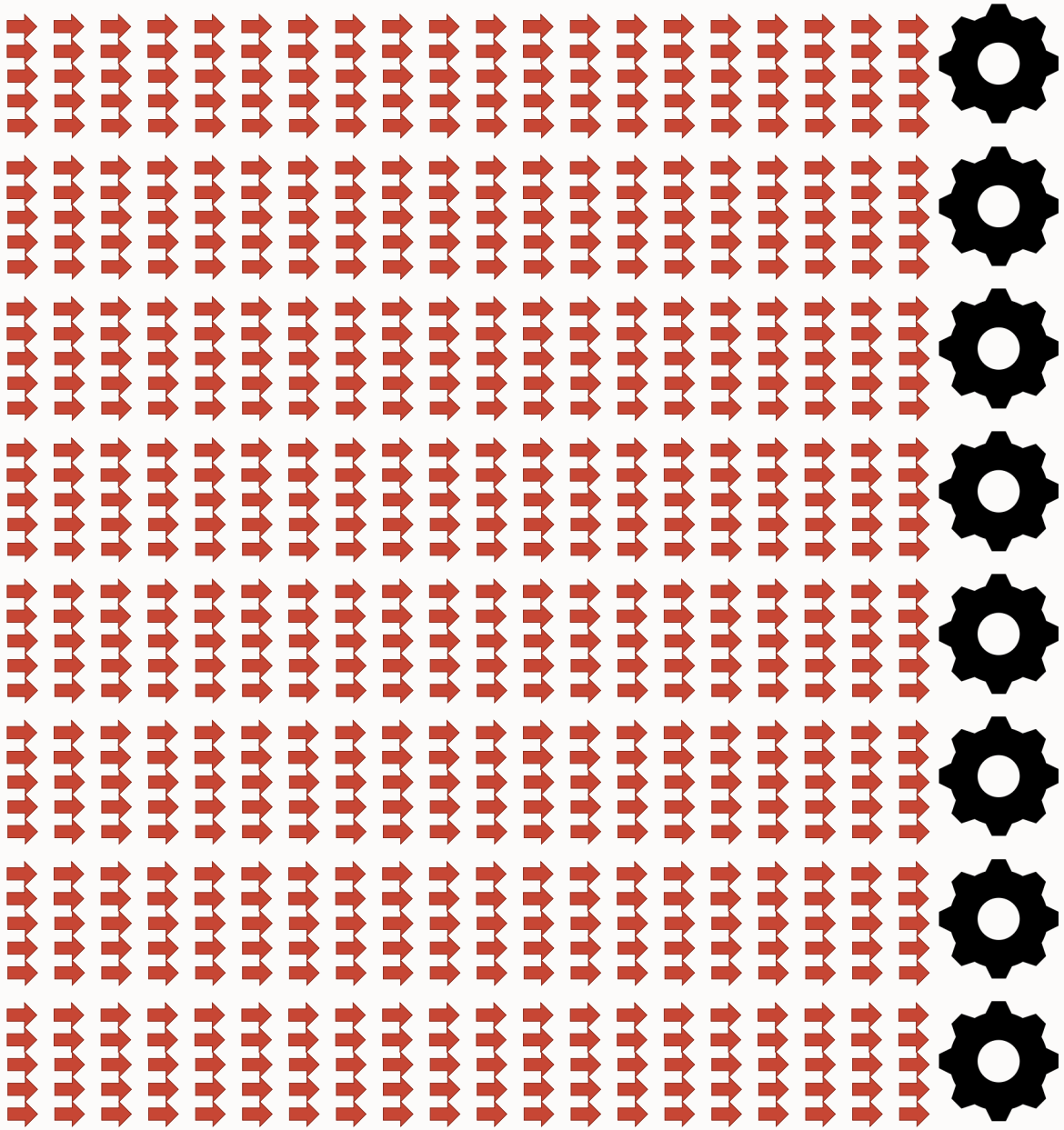
8 Cores

~~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….

Project Loom

Don't make users choose between efficient development and efficient deployment!

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

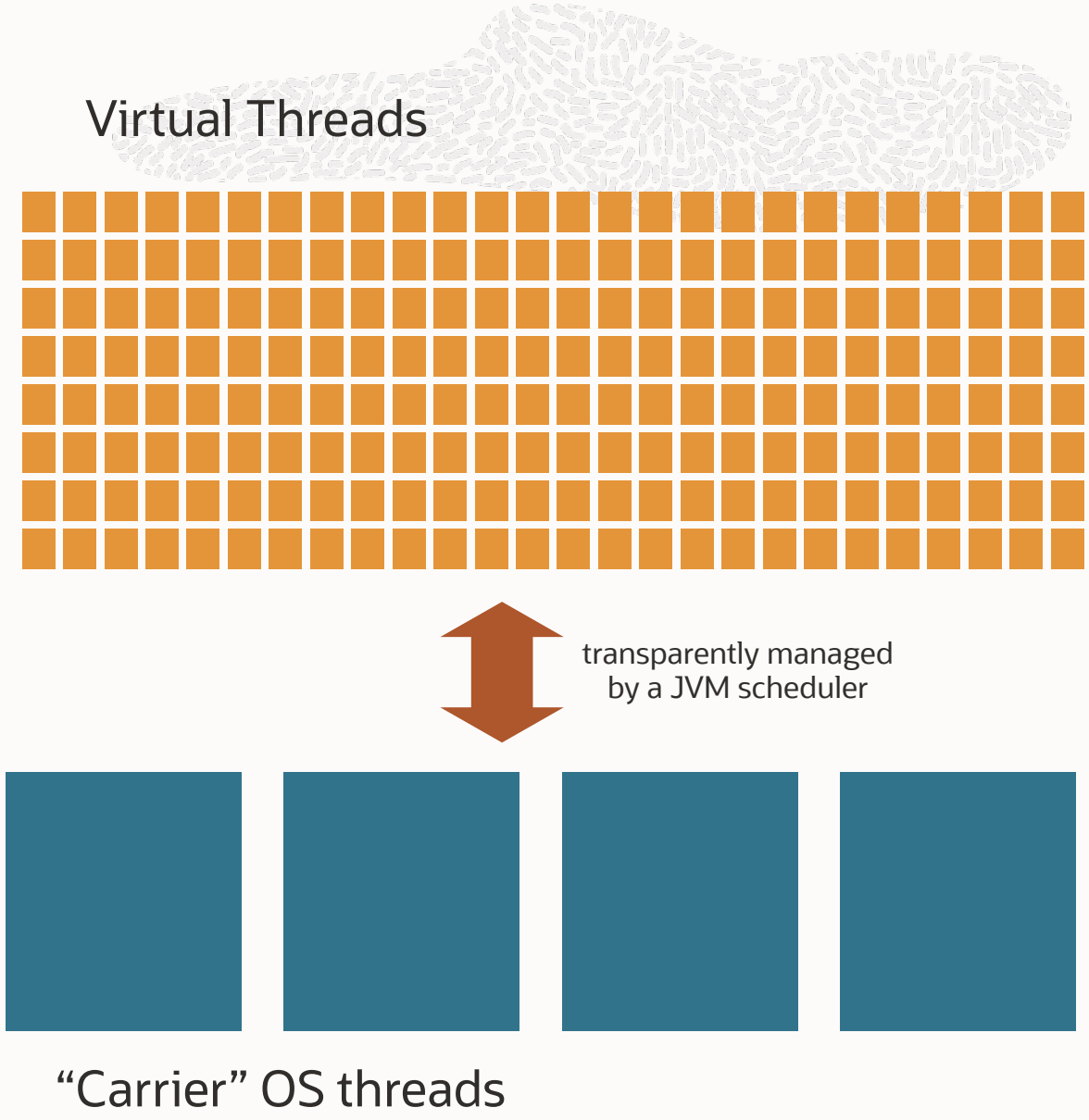
Virtual Threads – JDK 21

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"

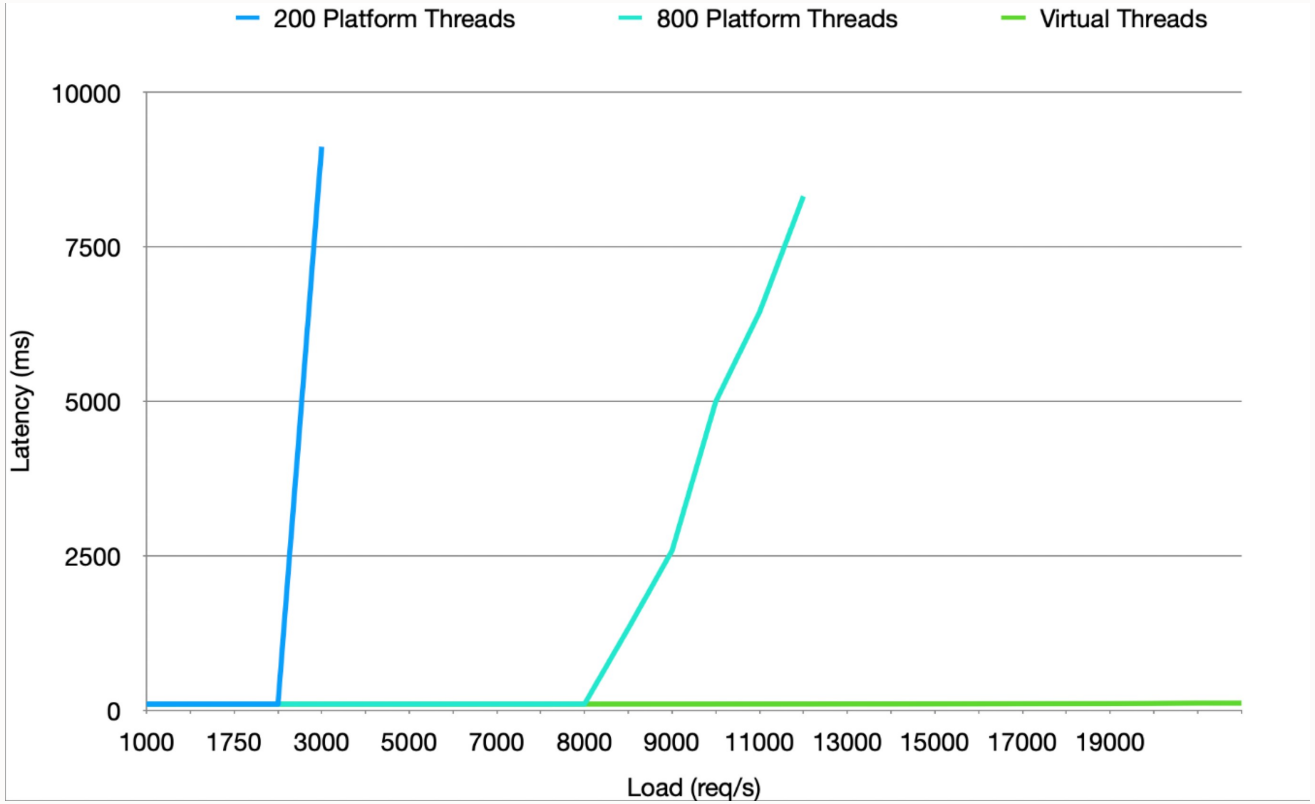


Breaking the bottleneck

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



Same abstraction, new mindset

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

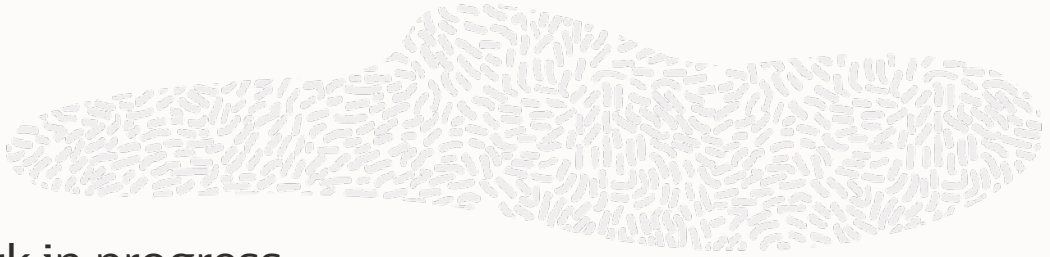
Virtual threads transparently suspended / resumed when they block

- Blocking APIs throughout the JDK retrofitted to be aware of virtual threads

Active projects in the OpenJDK community

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

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

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

ZGC

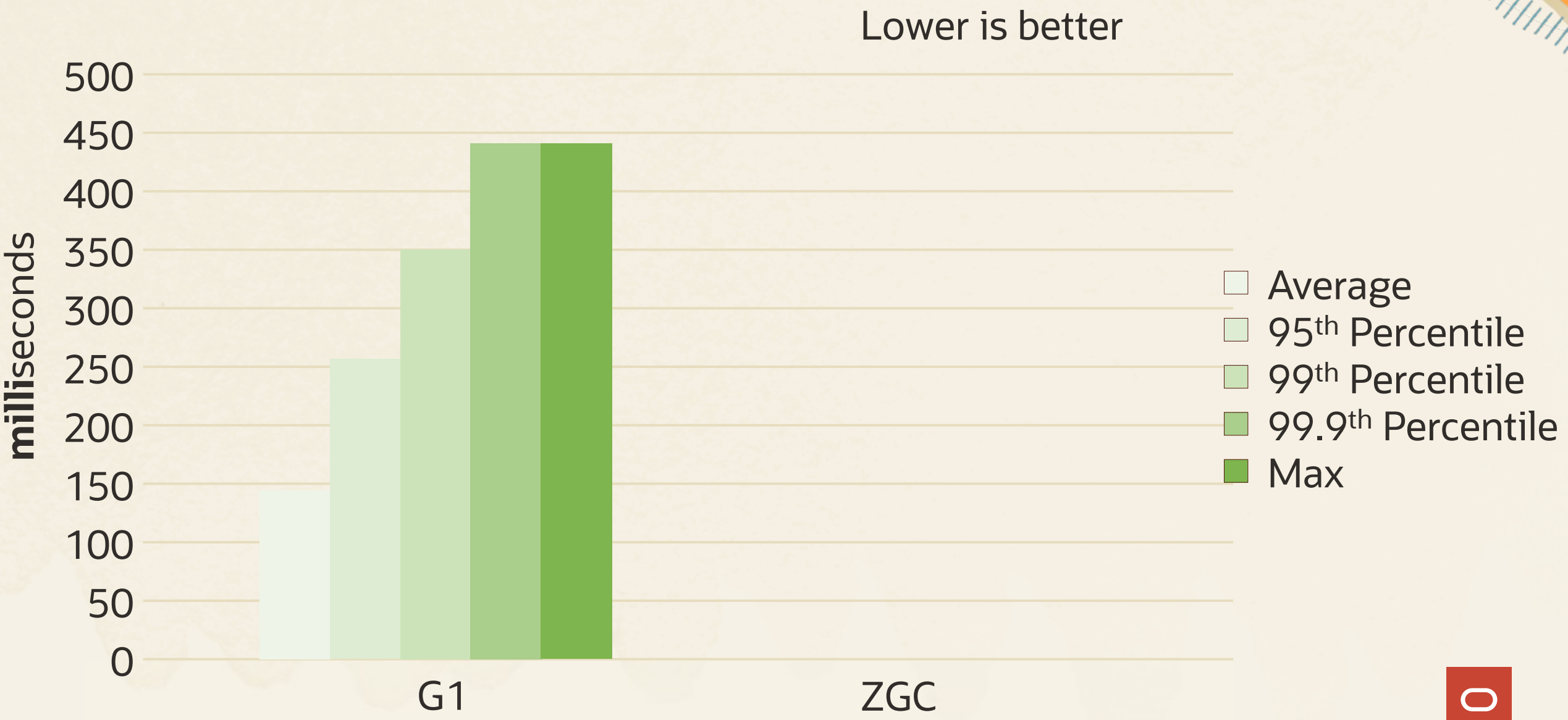
The “Z” garbage collector was introduced in JDK 15

- 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

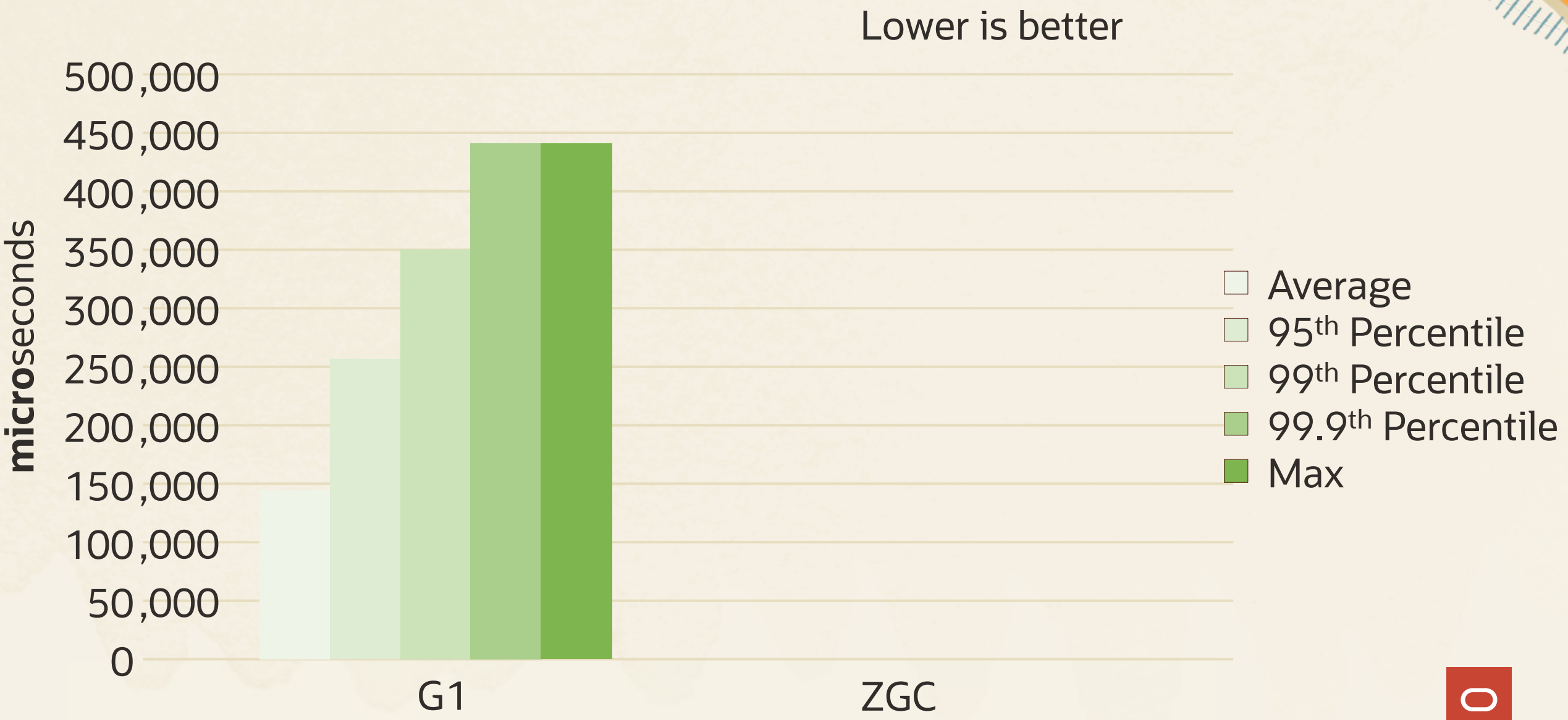
What’s the catch?

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

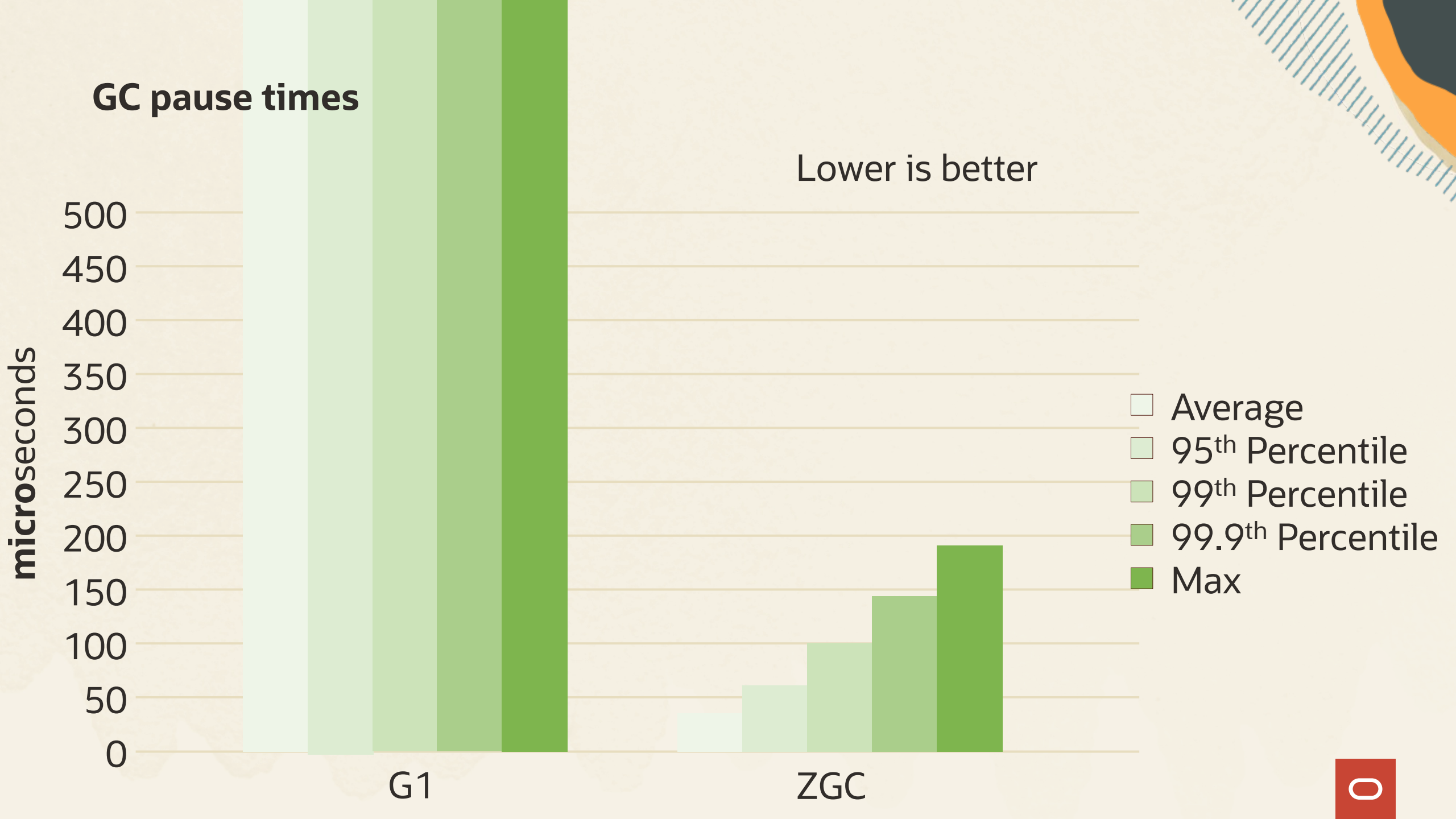
GC pause times



GC pause times

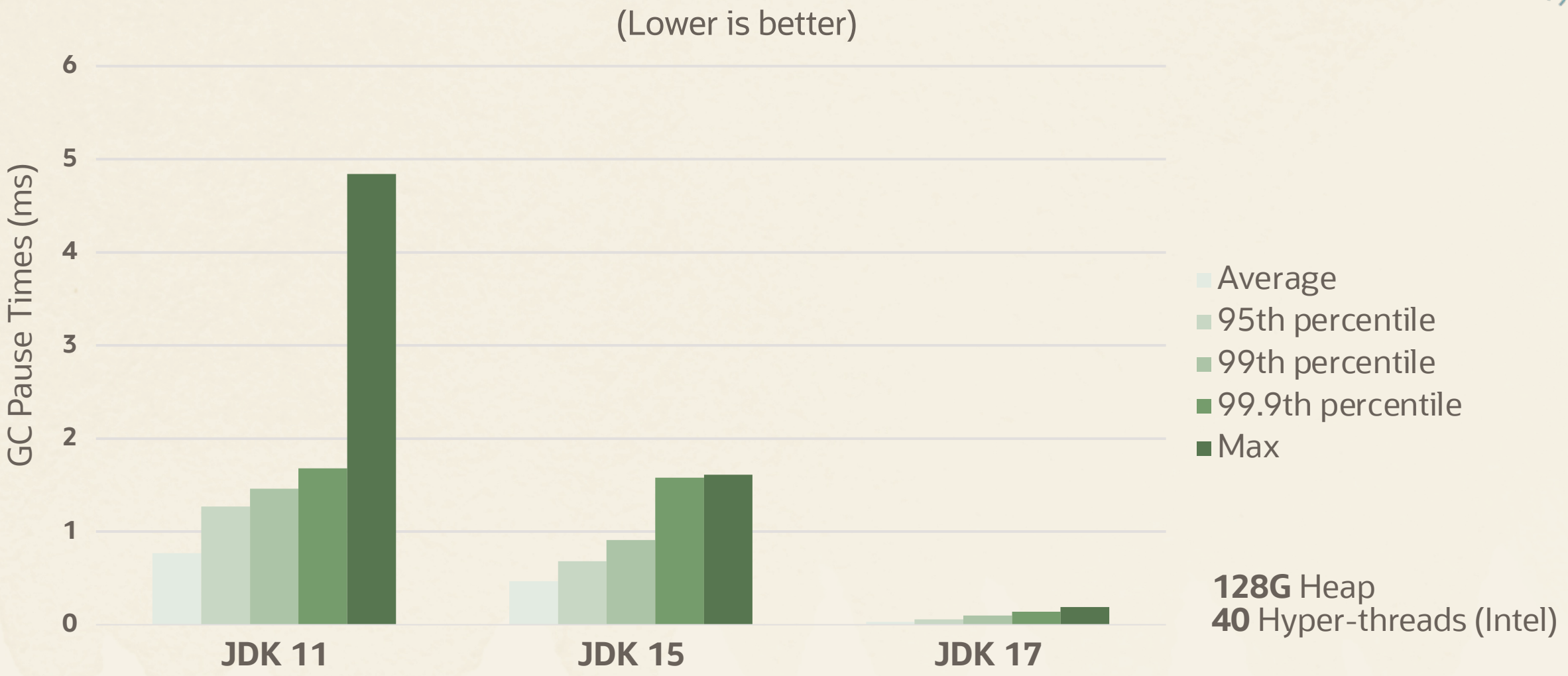


GC pause times



ZGC Improvements Over Time

GC Pause Times



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

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

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

Active projects in the OpenJDK community

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

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

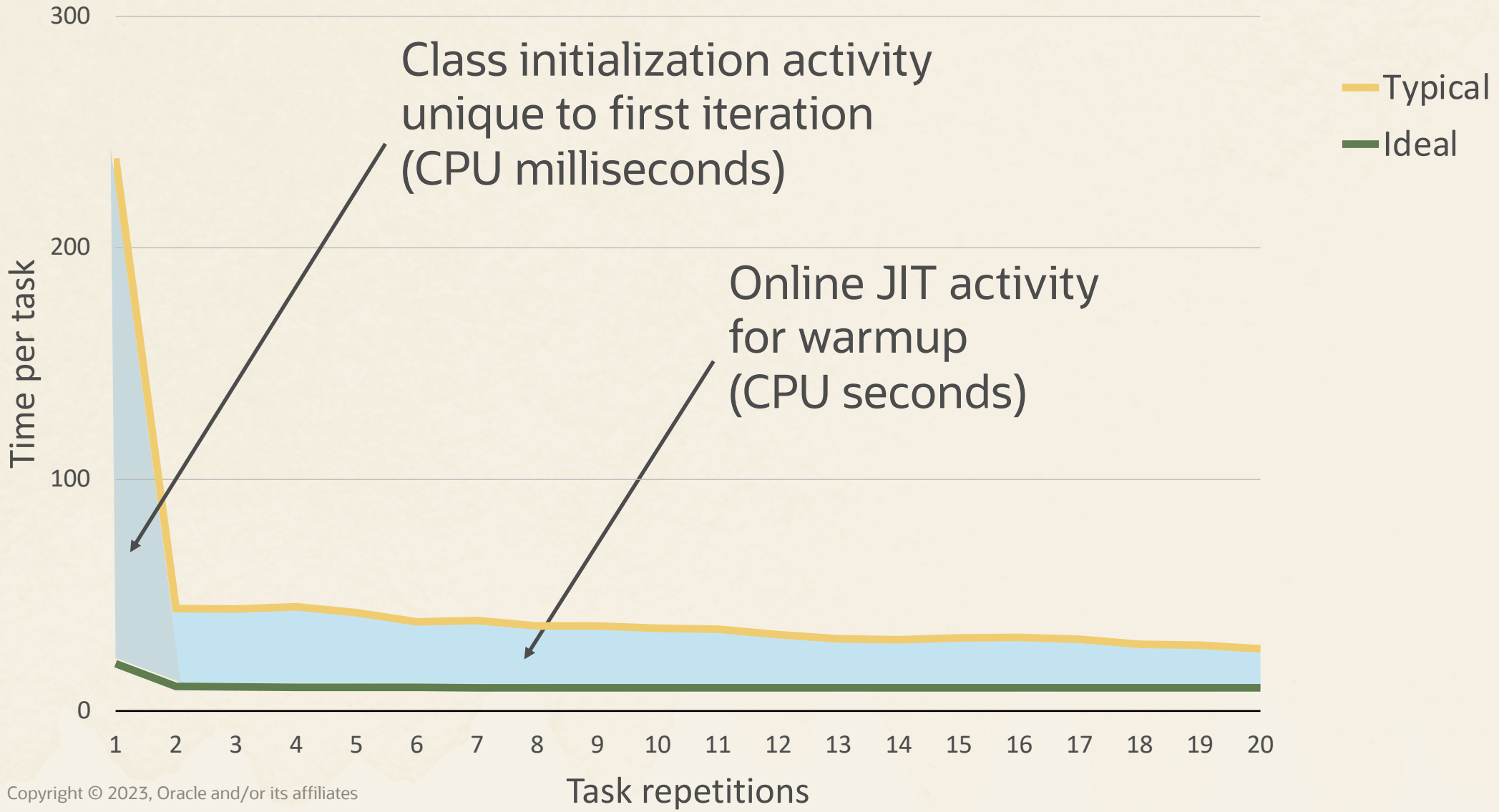
Java has historically favored long-term peak performance over startup

- A good tradeoff for many applications

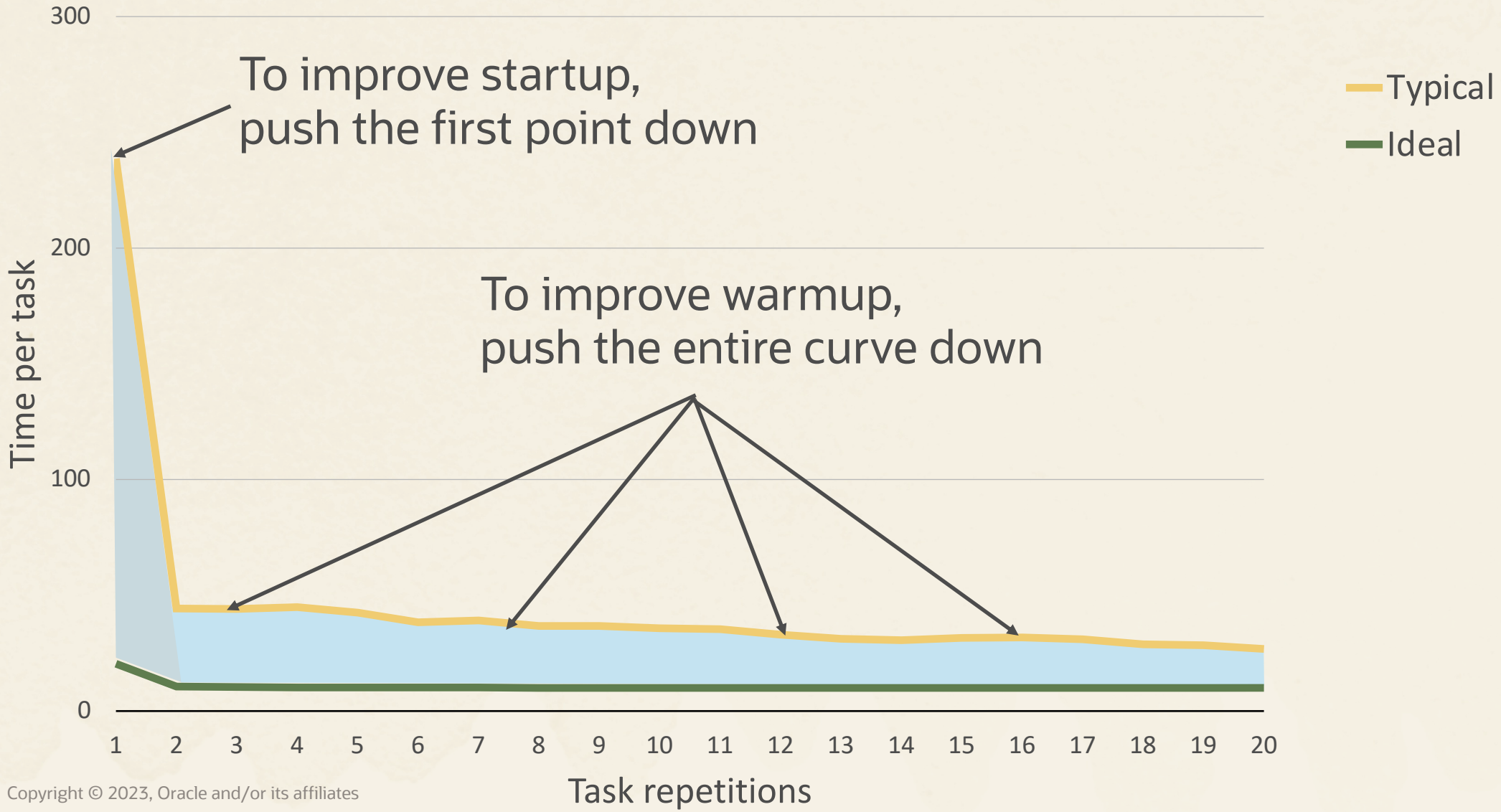
Java does a lot of work at startup – processing classfiles, interpretation, profile gathering, callsite linkage, JIT compilation

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

Startup and warmup



Startup and warmup



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

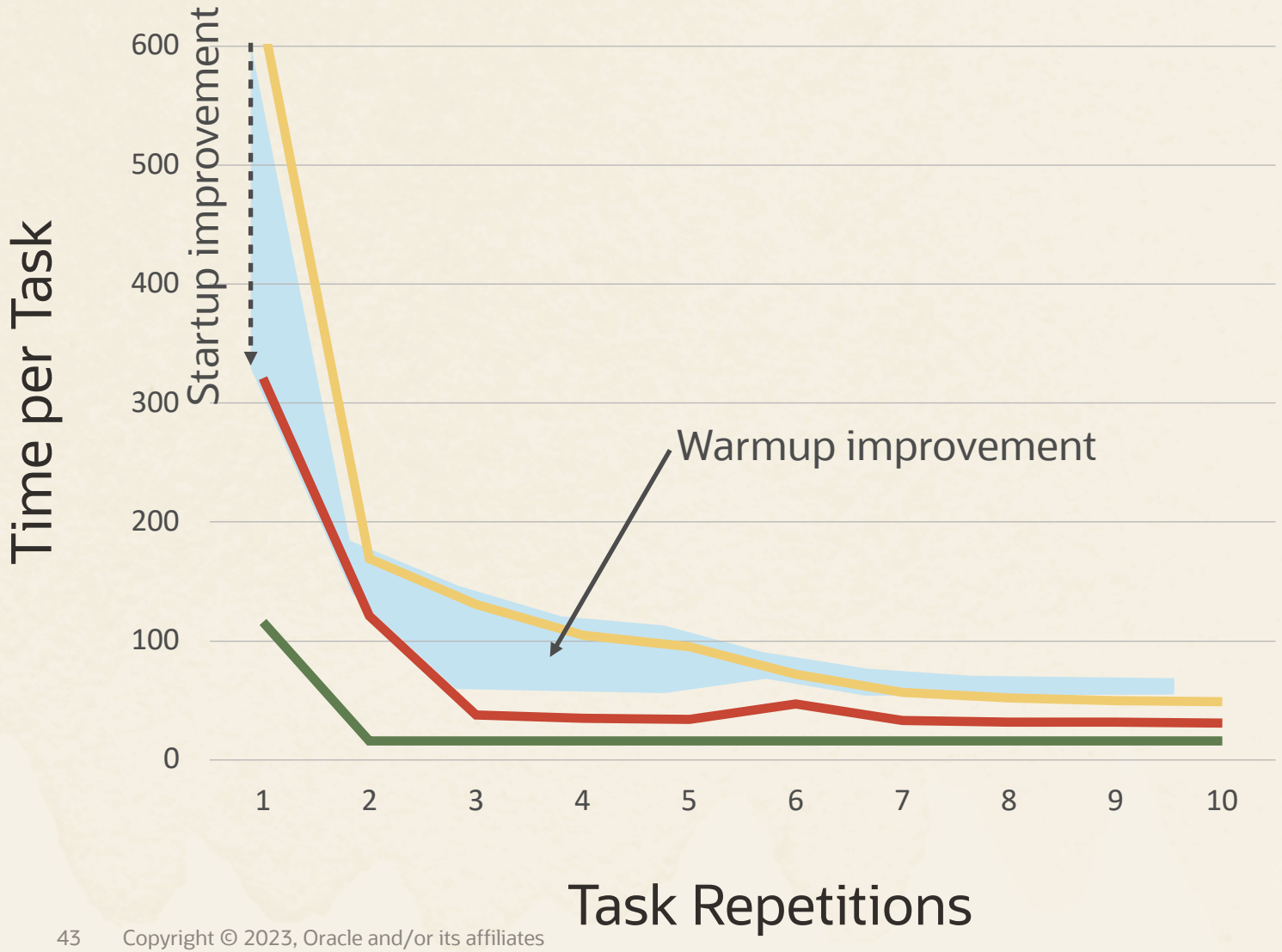
The JDK already employs many computation-shifting techniques

- Constant folding, garbage collection, class loading, JIT compilation

Let's shift more!

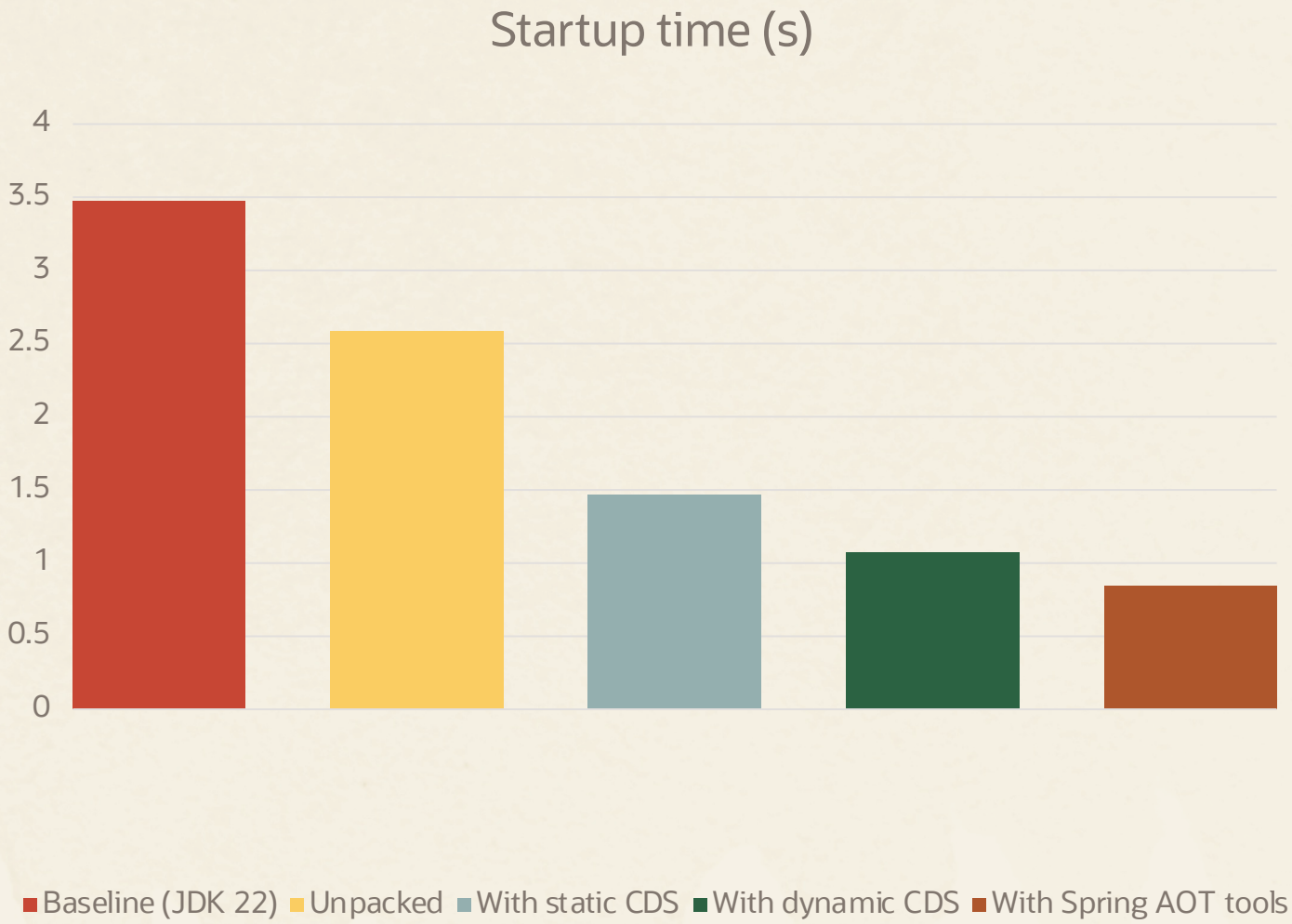
- Adapted the existing JIT compilers and Class Data Sharing (CDS) to precompute and store compilation profiles, compiled code, callsite linkage
- No changes to user code, no loss of dynamism
- Just a "training run" at build time

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



Spring Boot "Pet Clinic"

4.1x startup improvement with no change to existing code

Active projects in the OpenJDK community

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

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

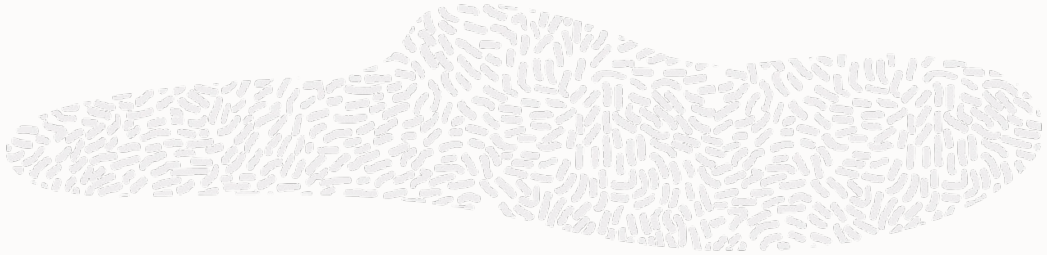
- Early Access of Project Builds


ORACLE

Appendix

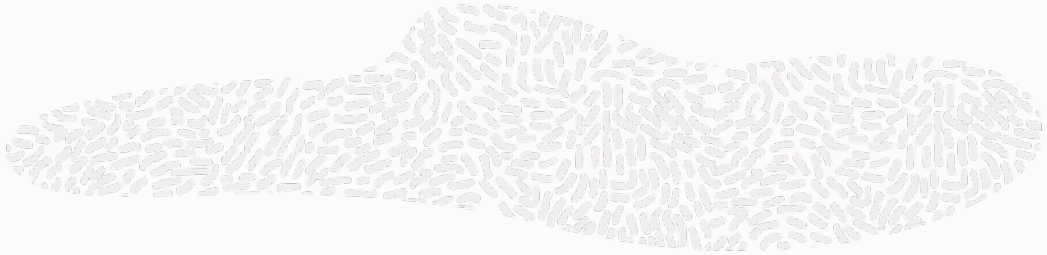
Project Amber Features

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

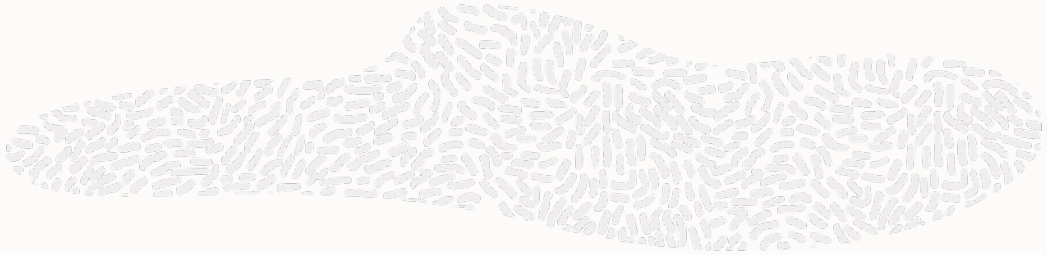


```
var url = new URL("http://www.oracle.com/");

var con = url.openConnection();

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

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

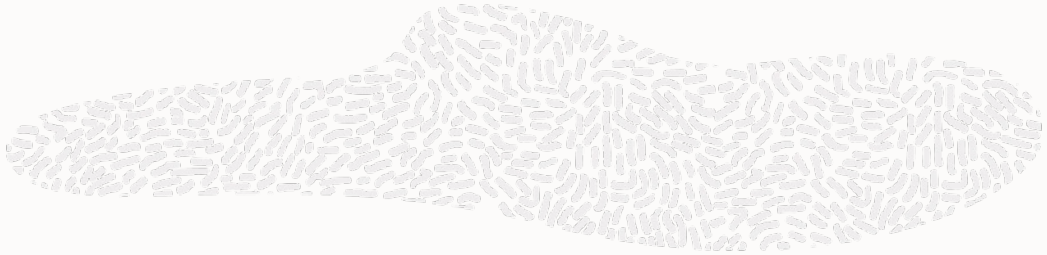
Style Guide: https://openjdk.java.net/projects/amber/LVTIstyle.html

Switch Expressions
JDK 14

```
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;
```

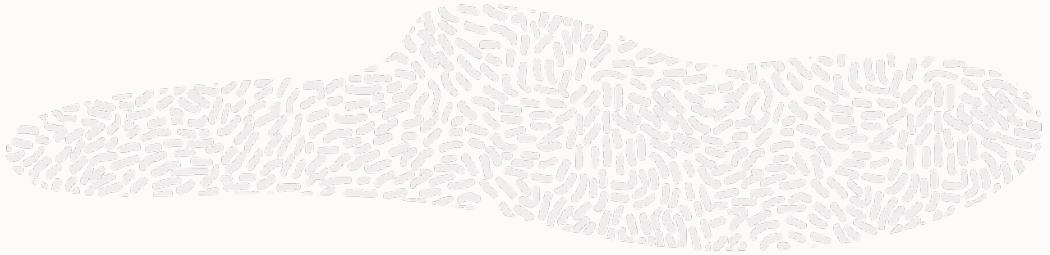


Switch Expressions
JDK 14



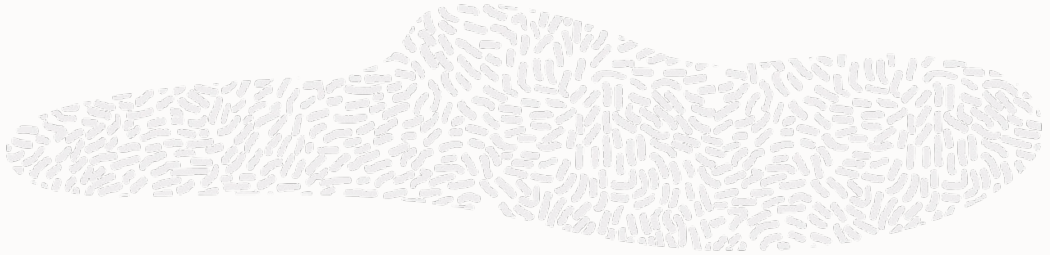
```
return switch (day) {
    case MONDAY, FRIDAY, SUNDAY  -> 6;
    case TUESDAY                 -> 7;
    case THURSDAY, SATURDAY      -> 8;
    case WEDNESDAY               -> 9;
};
```

Text Blocks JDK 15



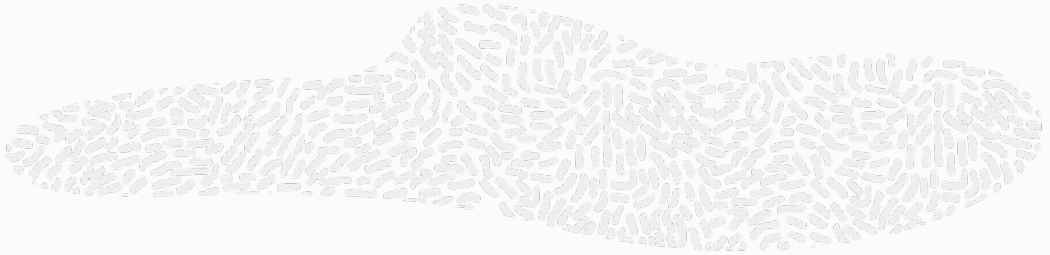
```
var html += "<tr>\n" +
                "<td>Retweets: " + t.getRetweetCount() + "</td>\n" +
                "<td>Likes: " + t.getLikeCount() + " </td>\n" +
                "<tr>\n ";
```

Text Blocks JDK 15



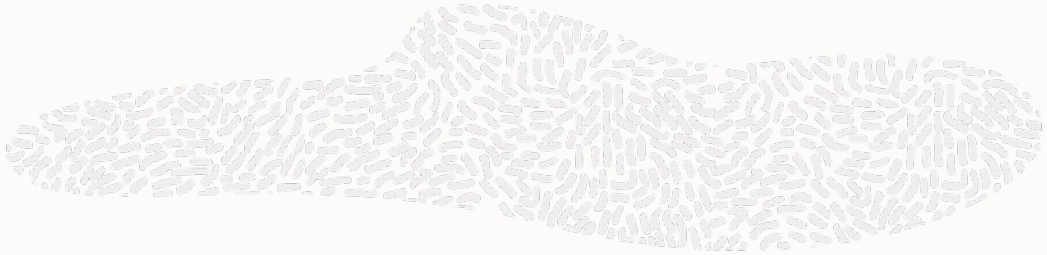
```
var html += """
                <tr>
                    <td>Retweets: %s</td>
                    <td>Likes: %s</td>
                <tr>
                """.formatted(t.getRetweetCount(),
                              t.getLikeCount());
```

Text Blocks JDK 15



```
var html += """
            <tr>
                <td>Retweets: %s</td>
                <td>Likes: %s</td>
            <tr>
            """.formatted(t.getRetweetCount(),
                          t.getLikeCount());
```

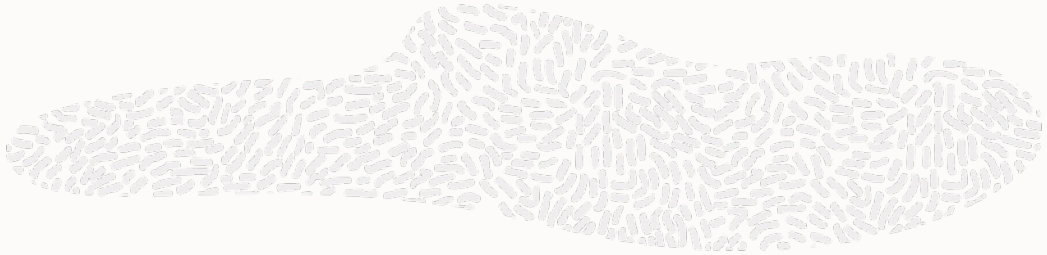
Pattern Matching for instanceof JDK 16



```
if (obj instanceof String) {
    String s = (String) obj;
    // use s
}
```

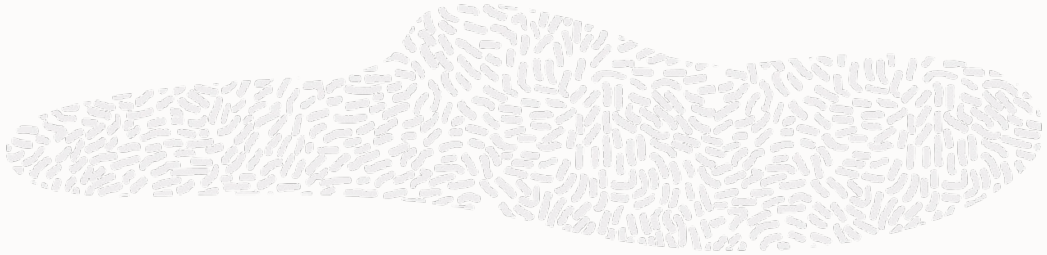
1) a test: *is obj a String*
2) declaration of a new variable **`s`**
3) casting of **`obj`** to String into variable **`s`**

Pattern Matching for instanceof JDK 16



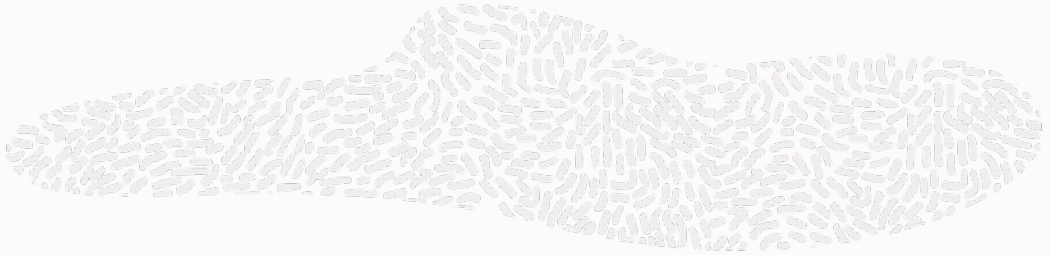
```
if (obj instanceof String s) {
    // use s
}
```

Pattern Matching for instanceof JDK 16



```
if (obj instanceof String s) {
    // use s
} else {
  //s is out of scope here!
}
```

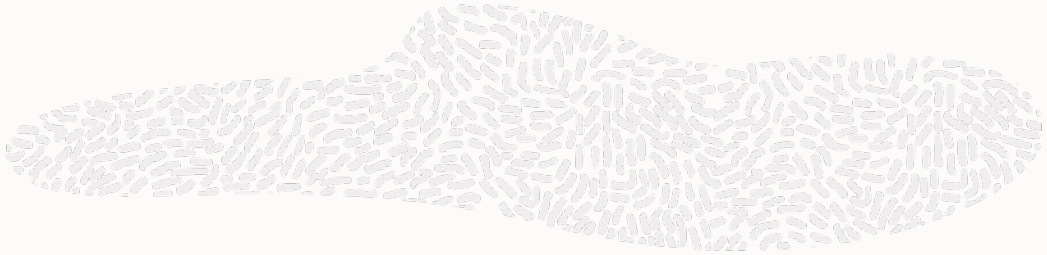
Record Classes JDK 16



```
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 Classes JDK 16



```
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

```
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

```
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

```
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

```
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 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

```
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

```
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();
    };
}
```

Pattern Matching for `switch` – Case Refinement

```
static void test(Object o) {
    switch (o) {
        case String s:
            if (s.length() == 1)
                {//handle single character strings}
            else
                {//handle all other strings}
                break;
        ...
    };
}
```

The desired test: [if o is a String of length 0] is split between the case and the if statement

Pattern Matching for `switch` – Optional `when` clause

```
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
```

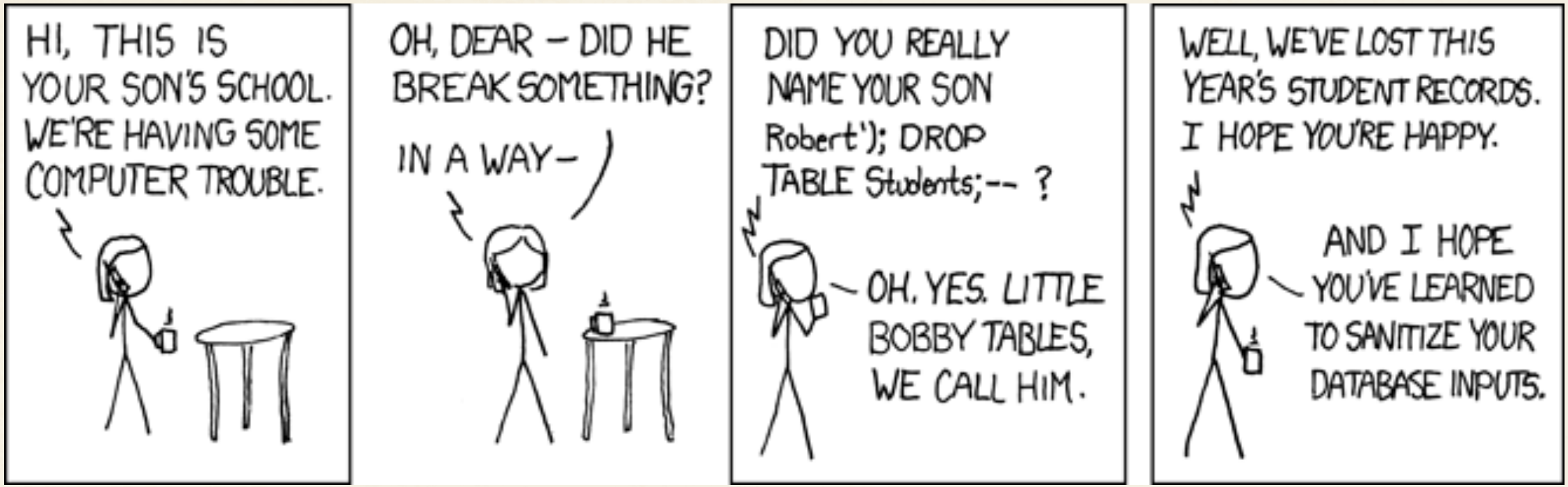
Why not add String Interpolation?

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

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

var html = `<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

```
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

```
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

```
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 Templates

Multi Line Embedded Expressions

```
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>
    """;
```

```
"""
<html>
  <head>
    <title>My Web Page</title>
  </head>
  <body>
    <p>Hello, world</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`

```
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

```
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

```
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

Class declaration and public access modifier

Parameters to interface with OS's shell

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

static modifier is part of class-and-object model

"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

```
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);
    }
}
```