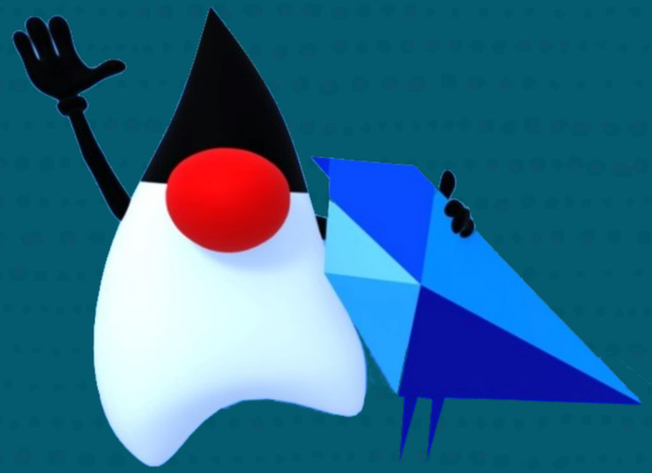
Helidon 4 on Virtual Threads

JCPEC24
Daniel Kec
Java Developer
Oracle



Agenda

- Quick Helidon introduction
- Optimizing server concurrency
- Remember reactive code?
- Virtual Threads
- Helidon 4 - blocking code is cool again!
- Pinning the carrier thread

Helidon

Quick introduction

What is Helidon

- Framework for developing cloud-native Java (micro)services
- K8s friendly
- Helidon is 100% Open Source, available on GitHub
- Open source Support: GitHub, Slack, Stack Overflow
- Commercial Support through Oracle Support for customers of WLS, Coherence

Helidon flavors

Helidon provides 2 programming models

Helidon SE

- Micro-framework
- Pure performance
- No Magic

Helidon MP

- MicroProfile
- Declarative (IOC)
- CDI, JAXRS
- Jakarta APIs
- Helidon SE under the hood

Helidon flavors

Imperative vs. Declarative style

Helidon SE

```
WebServer.builder()
        .addRouting(HttpRouting.builder()
                .get("/greet", (req, res)
                        -> res.send("Hello World!")))
        .build()
        .start();
```

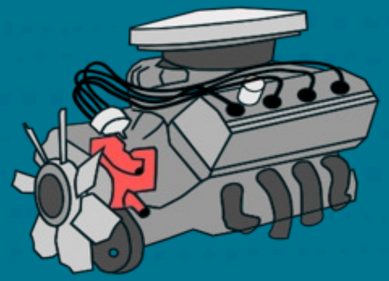
Helidon MP

```
@Path("/greet")
public class GreetService {

    @GET
    public String getMsg() {
        return "Hello World!";
     }
}
```

Helidon flavors

Helidon MP is under the hood powered by Helidon SE

Helidon SE



Helidon MP

Optimizing server concurrency

- Server's challenges

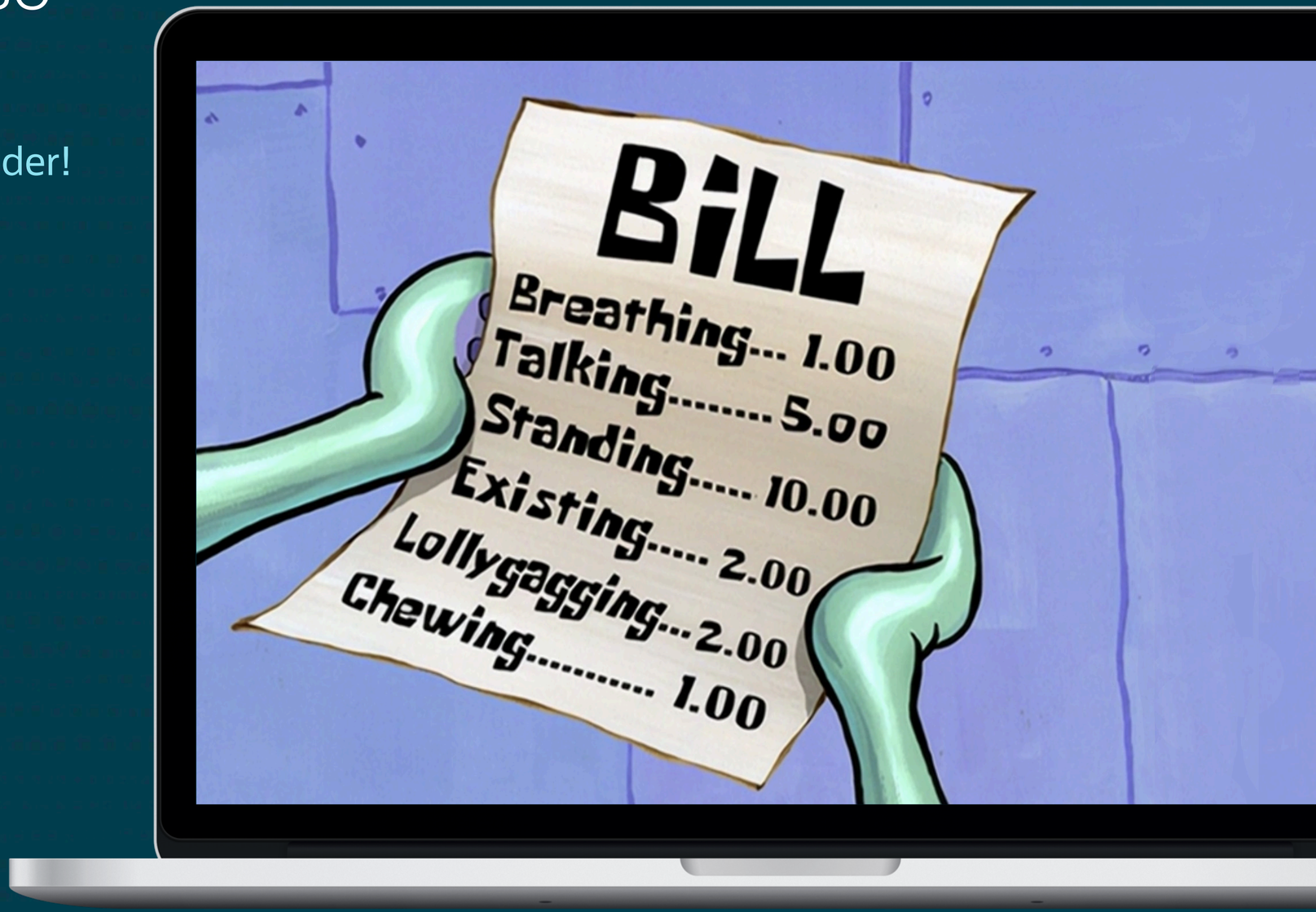
What problem do we solve?

- **Heavily concurrent environment**, usual for HTTP server
- Requirement to handle calls to other systems (database, messaging, other services [HTTP, grpc…])
- Requirement to return with **low latency** – requests are not designed to be long running
- **Limited memory**, CPU → limited number of platform threads
- Optimize, optimize, optimize …

Why is optimization so important?

Look at the bill from your cloud provider!

- CPU cycles ….$$$
- Memory ……. $$$
- Storage ……... $$$



Expensive Concurrency

- Java platform-threads are mapped one-to-one to the kernel threads
- Each kernel thread created by JVM needs megabytes of memory
- Kernel threads are scheduled by OS
- **Starting new kernel thread is expensive!**
- **Context switching is expensive!**

What can we do about it?

- Reusing threads - **thread pools**
- **“Don’t block the thread!”** - Keep one thread busy, rather than multiple threads waiting

Reactive programming

- Do you like reactive code?

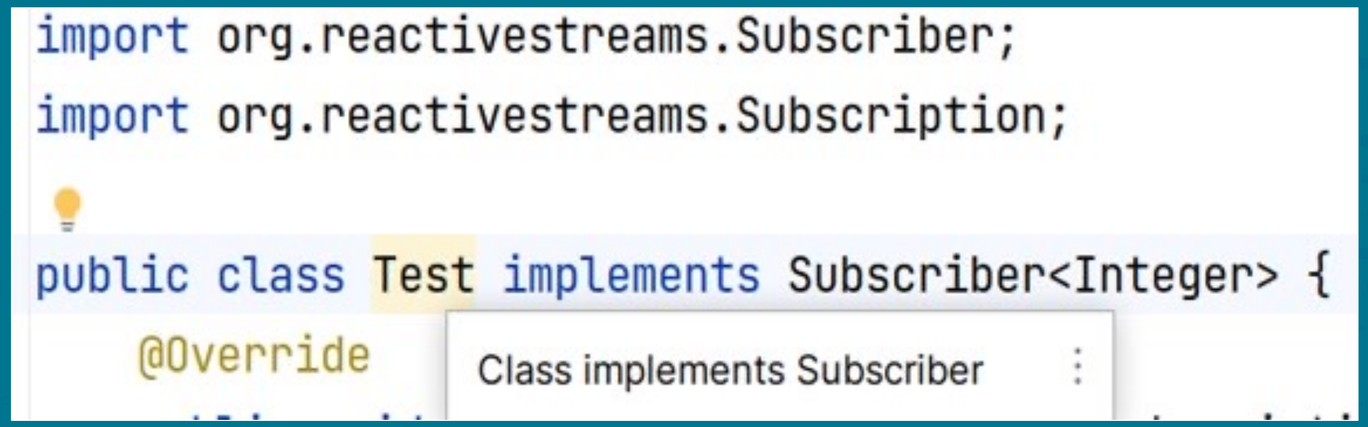
Reactive programming

- **Asynchronous** - we don’t wait for something to happen
- Just provide function to be called when it happens - callback function
- We have lost a flow control by giving up blocking, we need a means for backpressure control
- **Callback hell!**
- **Reactive Streams** API for callback orchestration

Reactive operators

Reactive Streams provides API for non-blocking back pressure control(request(1), request(5)...)

- Part of JDK since Java 9(Flow API)
- It's **hard to implement right**
- Reactive Streams spec rules are ridiculously complicated
- Even **IntelliJ warns you off!**



Reactive Streams implementations

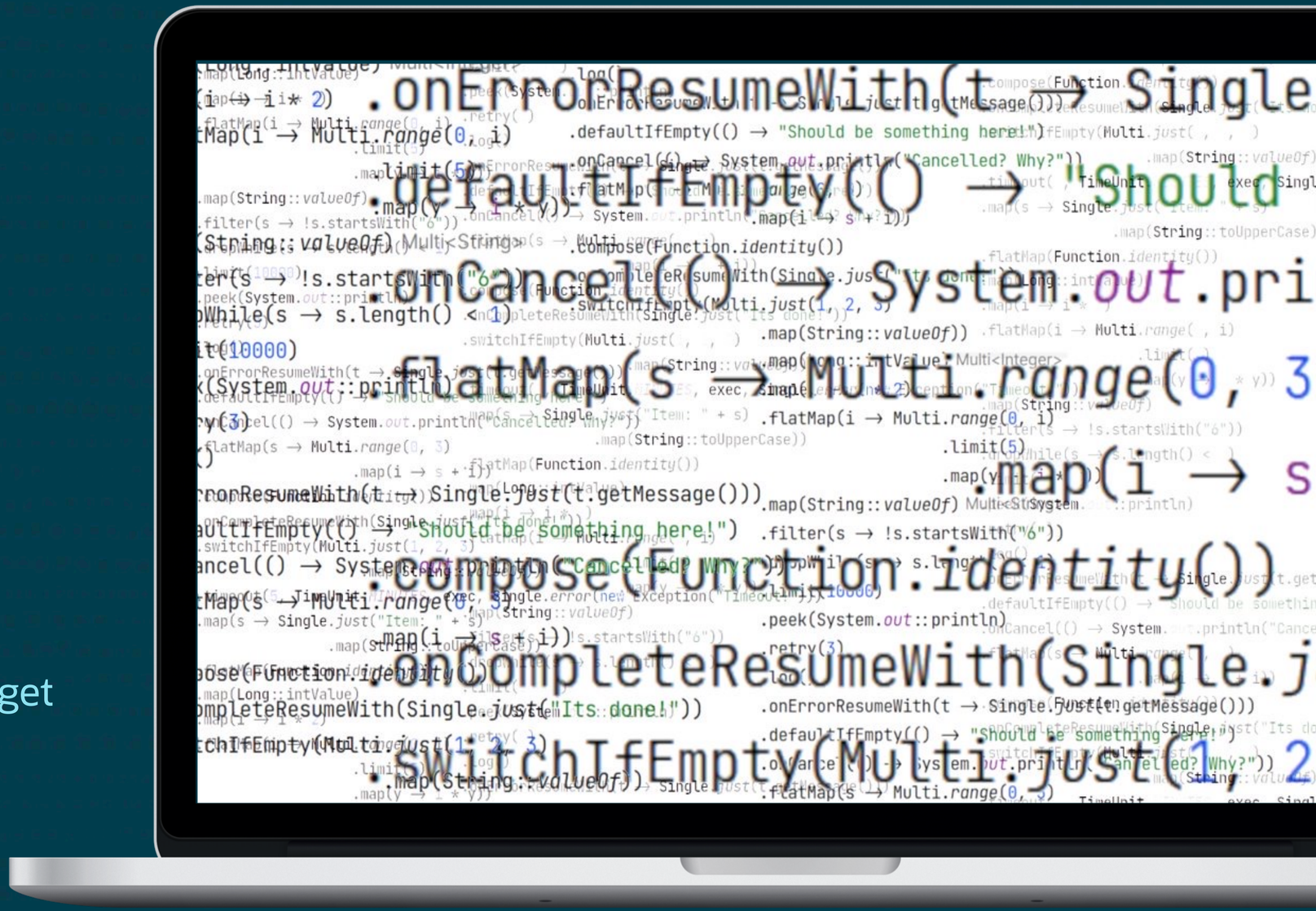
Composable reactive operators

- RxJava
- Reactor
- Akka-Streams
- Service-Talk
- Helidon
- Mutiny

So reactive operators are nice?

Reactive Operators

Is it confusing at the start? It doesn't get much better later.

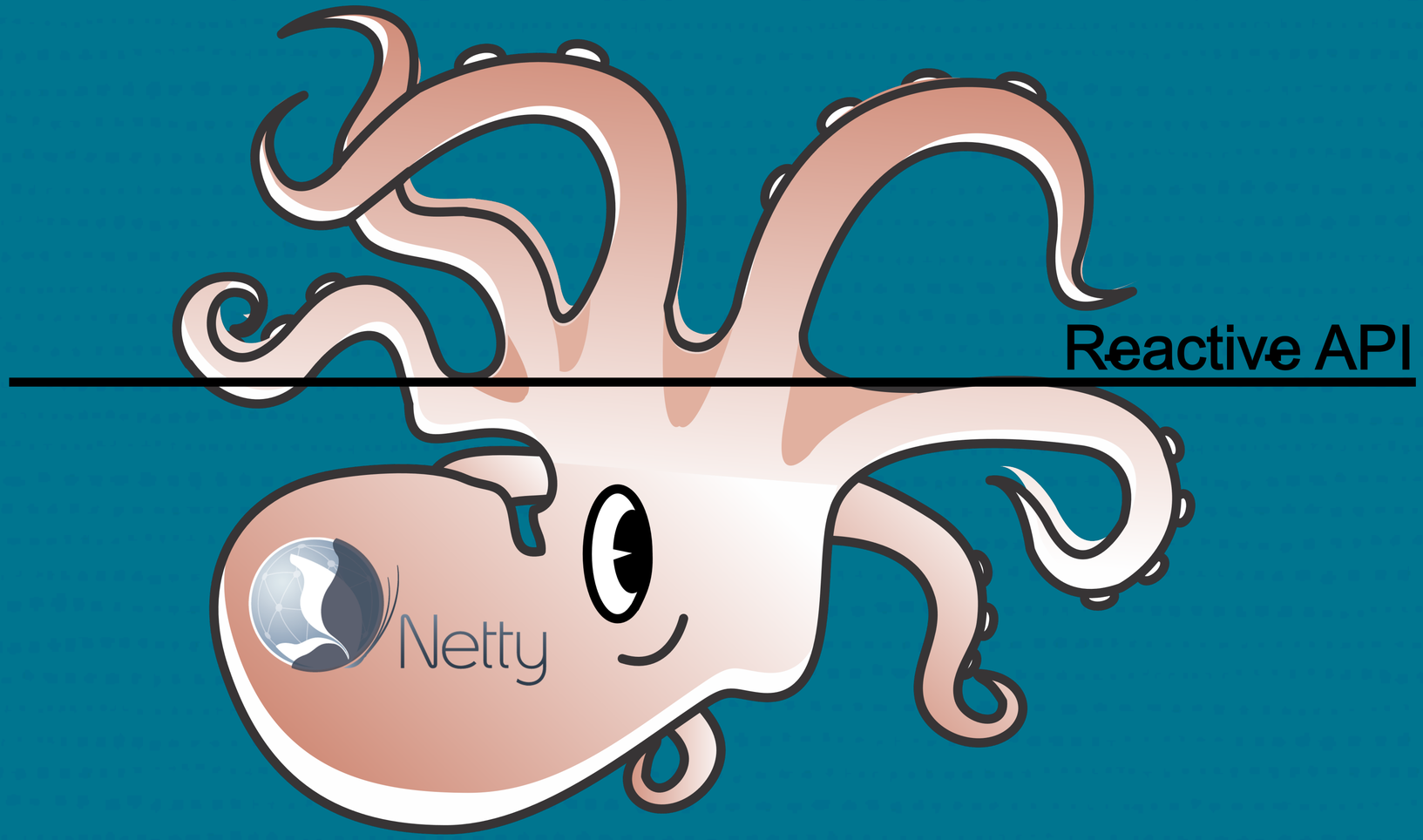


Reactive programming

- Steep learning curve
- Hard to get right™
  - Troubleshooting
  - No useful stack traces
  - More than one task in parallel is tough
- Using blocking code requires offloading

- “Callback Hell”

Reactive API



Virtual Threads

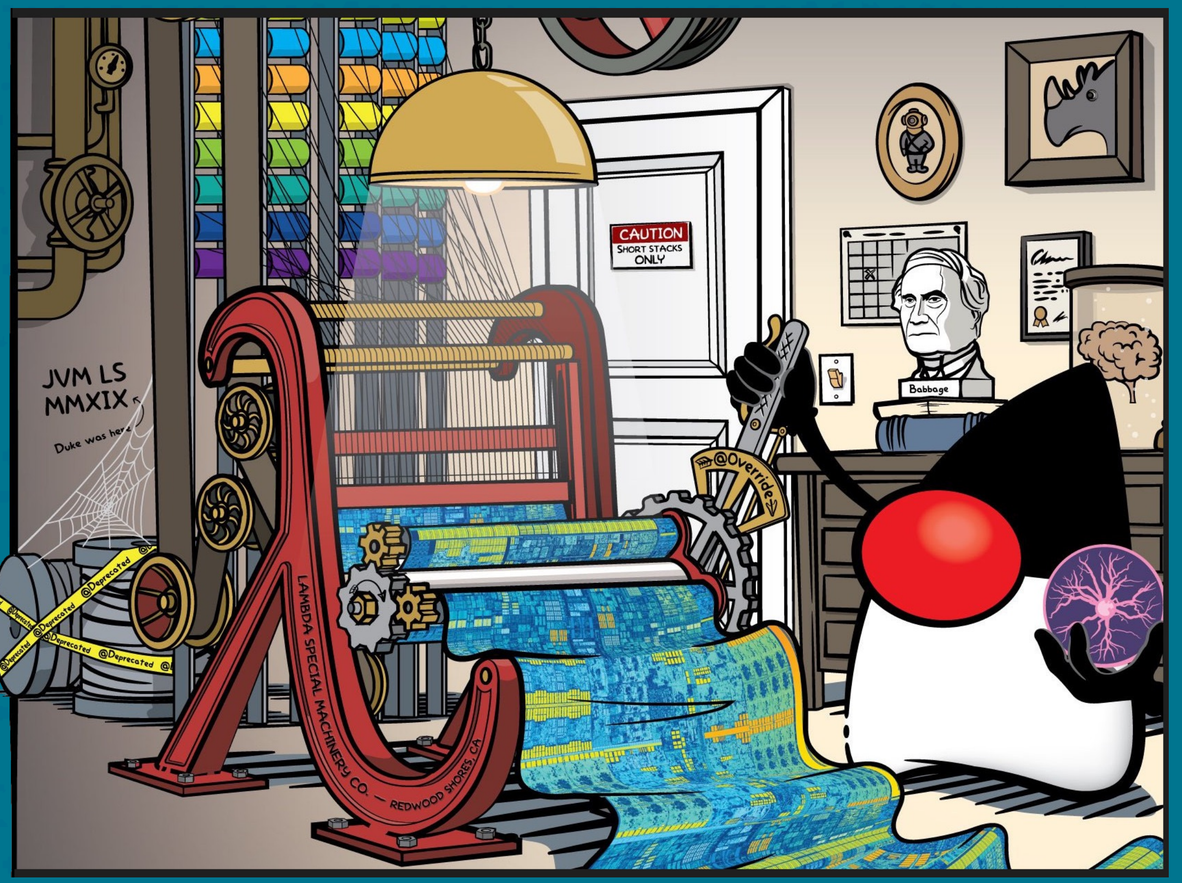
- Project Loom

Better Solution?

- Virtual Threads (Part of project Loom)
  - JEP-425 Preview feature since Java 19
  - JEP-444 Delivered in Java 21 (September 2023)
- Threads can now be either **Platform** or **Virtual**
- Blocking operations do not block a platform/carrier thread
- Can have a huge number of virtual threads
- Useful stack traces
- “Naive” approach to coding Java is back (and safe)

Virtual Threads

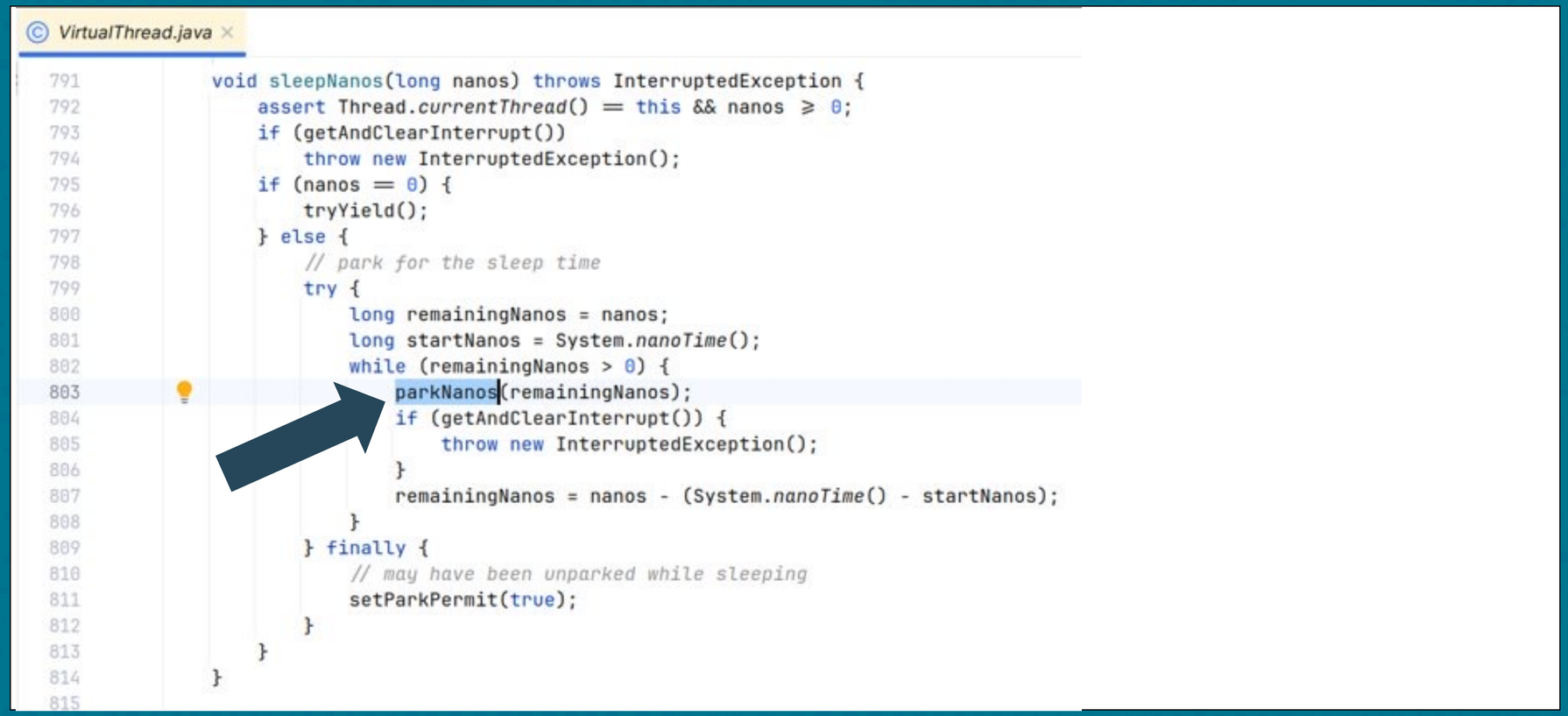
- We can block cheaply!
- Imperative code can achieve performance comparable with reactive constructs
- Green threads again? - Not really!
- Yielding happens under the hood(sleep)



java.lang.Thread.sleep()

```
07.05.22 Bateman   498      public static void sleep(long millis) throws InterruptedException {
07.05.22 Bateman   499          if (millis < 0) {
07.05.22 Bateman   500              throw new IllegalArgumentException("timeout value is negative");
07.05.22 Bateman   501          }
07.05.22 Bateman   502
07.05.22 Bateman   503          long nanos = MILLISECONDS.toNanos(millis);
11.04.23 Bateman   504          ThreadSleepEvent event = beforeSleep(nanos);
07.05.22 Bateman   505          try {
11.04.23 Bateman   506              if (currentThread() instanceof VirtualThread vthread) {
11.04.23 Bateman   507                  vthread.sleepNanos(nanos);
07.05.22 Bateman   508              } else {
07.05.22 Bateman   509                  sleep0(millis);
07.05.22 Bateman   510              }
11.04.23 Bateman   511          } finally {
11.04.23 Bateman   512              afterSleep(event);
11.04.23 Bateman   513          }
07.05.22 Bateman   514      }
07.05.22 Bateman   515
```

java.lang.VirtualThread.sleepNanos(long nanos)



java.lang.VirtualThread.parkNanos(long nanos)

VirtualThread.java

```
616  void parkNanos(long nanos) {
617      assert Thread.currentThread() == this;
618
619      // complete immediately if parking permit available or interrupted
620      if (getAndSetParkPermit(false) || interrupted)
621          return;
622
623      // park the thread for the waiting time
624      if (nanos > 0) {
625          long startTime = System.nanoTime();
626
627          boolean yielded = false;
628          Future<?> unparker = scheduleUnpark(this:::rk, nanos);
629          setState(PARKING);
630          try {
631              yielded = yieldContinuation();  // may throw
632          } finally {
633              assert (Thread.currentThread() == this) && (yielded == (state() == RUNNING));
634              if (!yielded) {
635                  assert state() == PARKING;
636                  setState(RUNNING);
637              }
638              cancel(unparker);
639          }
640
641          // park on carrier thread for remaining time when pinned
642          if (!yielded) {
643              long deadline = startTime + nanos;
644              if (deadline < 0L)
645                  deadline = Long.MAX_VALUE;
646              parkOnCarrierThread(true, deadline - System.nanoTime());
647          }
648      }
649  }
650
```

yieldContinuation();

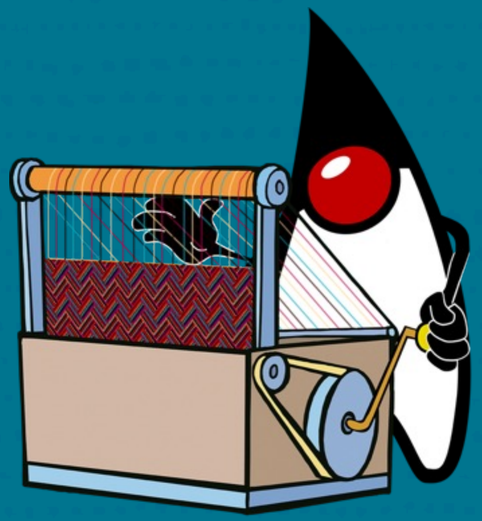
Continuations in Java!

Helidon 4

Blocking is cool again

Helidon 4

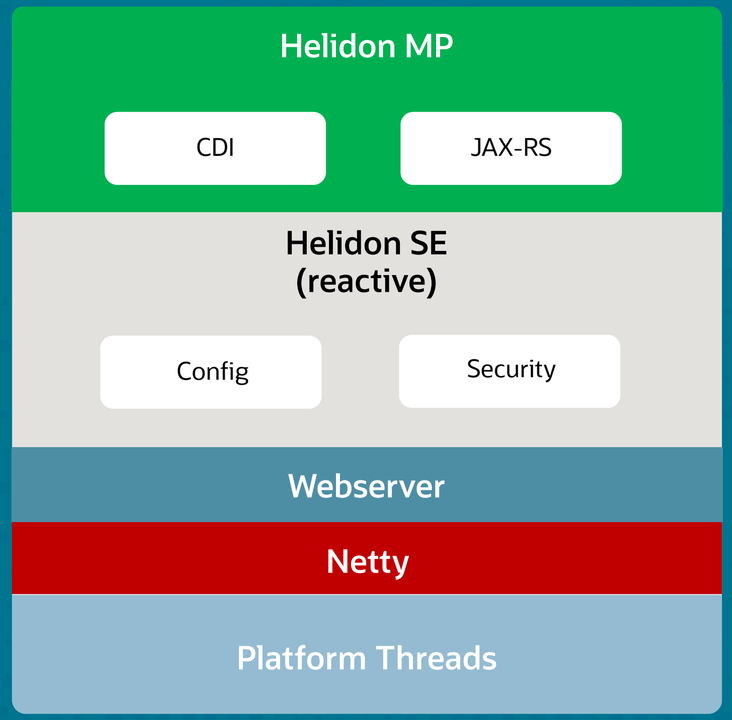
- Requires - Java 21
- Netty replaced with custom Web Server (Project Níma)
  - Designed for Virtual Threads
  - Created in cooperation with the Java team
  - Performance comparable to Netty
  - Heart of Helidon 4 release



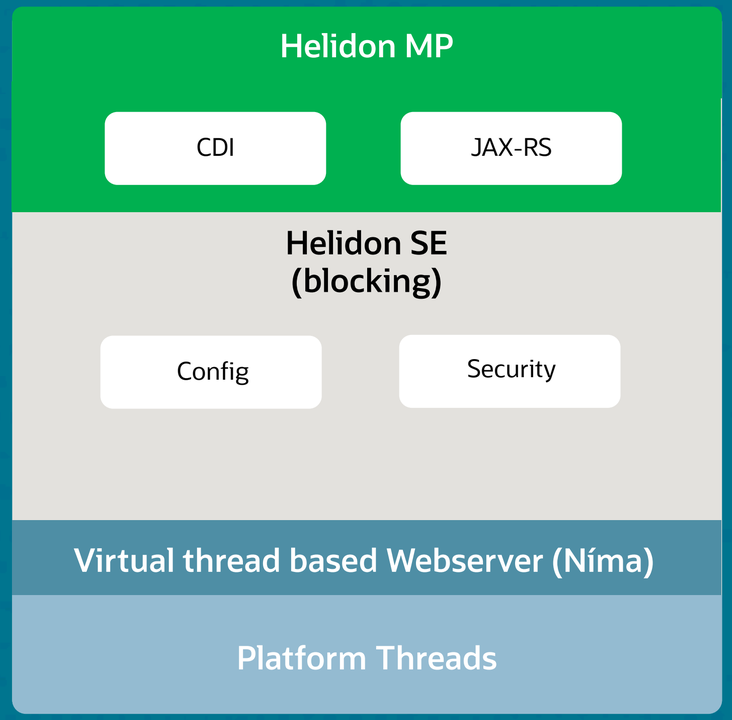
Architecture

Changes in Helidon 4

Helidon 1.x, 2.x, 3.x



Helidon 4.x



Helidon features timeline

Helidon 1

- Feb 14, 2019
- **Netty** based Web Server
- JDK >8
- **Javax** based MP
  - MicroProfile 3.2
  - Java EE 8

Helidon 2

- Jun 25, 2020
- **Netty** based Web Server
- JDK >11
- **Javax** based MP
  - **MicroProfile 3.3**
  - **Jakarta EE 8**

Helidon 3

- Jul 26, 2022
- **Netty** based Web Server
- **JDK >17**
- **Jakarta** based MP
  - MicroProfile 5.0
  - Jakarta EE 9.1

Helidon 4

- Oct 24, 2023
- **Virtual Thread** based Web Server(Project Níma)
- **JDK >21**
- **Jakarta** based MP
  - MicroProfile 6.0
  - Jakarta EE 10

Java 21

- Sep 19, 2023
- JEP 444 – **Virtual Threads**

Helidon 4 SE

- Switch to imperative code
  - Easier to debug
  - Easier to maintain
  - Easier to understand

Helidon 3 (reactive)

```
.get("/callOtherService", (req, res) → {
    seClient.get() WebClientRequestBuilder
            .request(JsonObject.class) Single<JsonObject>
            .map(jo → jo.getString("status")) Single<String>
            .map(String::toUpperCase)
            .onError(res::send)
            .forSingle(s → {
                res.send(s);
            });
})
```

Helidon 4 (imperative)

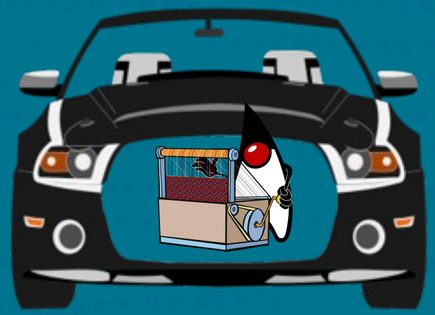
```
.get("/callOtherService", (req, res) → {
    String status = nimaClient.get() Http1ClientRequest
            .request(JsonObject.class) JsonObject
            .getString("status");

    String upperCaseStatus = status.toUpperCase();

    res.send(upperCaseStatus);
})
```

Helidon 4 MP

Helidon 4 MP is under the hood powered by Níma based Helidon 4 SE

Helidon 4 SE

Helidon 4 MP



Helidon 4 MP

- No change
- Just faster!

Helidon 3

```
@GET
public String callOtherService() {
    JsonObject jo = webTarget.request() Builder
            .buildGet() Invocation
            .invoke(JsonObject.class);

    return jo.getString("status").toUpperCase();
}
```

Helidon 4

```
@GET
public String callOtherService() {
    JsonObject jo = webTarget.request() Builder
            .buildGet() Invocation
            .invoke(JsonObject.class);

    return jo.getString("status").toUpperCase();
}
```

Helidon 4 Performance

TechEmpower Web Framework Benchmark

2023-10-17
Round 22

TechEmpower

Composite Framework Scores

Each framework's peak performance in each test type (shown in the colored columns below) is multiplied by the weights shown above. The results are then summed to yield a weighted score. Only frameworks that implement all test types are included. 159 total frameworks ranked, 5 visible, 154 hidden by filters. See filter panel above.

Rnk	Framework	JSON	1-query	20-query	Fortunes	Updates	Plaintext	Weighted score
37	helidon	429,240	268,833	30,291	238,545	9,390	3,035,006	**3,664** 45.3%
38	quarkus	903,185	318,897	17,610	214,275	6,697	2,861,479	**3,637** 45.0%
40	micronaut	568,955	221,741	28,171	179,741	15,209	1,327,013	**3,555** 44.0%
81	dropwizard	170,910	75,821	17,933	54,065	9,674	208,744	**1,608** 19.9%
88	spring	236,259	147,907	15,932	24,082	7,131	506,087	**1,507** 18.6%

https://www.techempower.com/benchmarks/#hw=ph&test=composite§ion=data-r22&f=zijunz-zik0zj-zik0zj-zik0zj-zik0zj-zik0zj-zik0zj-v2qiv3-xamxa7-zik0zj-zik0zj-zik0zj-zik0zj-zik0zj-1ekf

No Reactive layer

Helidon Webserver

Netty

Project Níma

No Netty!

- Got rid of numerous Netty CVEs

Helidon 4 – Why it is fundamentally different?

- Fully embrace and commit to Java 21 and Virtual Threads
- Brings back synchronous programming.
- Features a blocking, imperative API
- Easier to write, understand, debug, maintain
- Completely new Web Server, not a retrofit
- Helidon 4 WebServer implementation built from scratch, designed for virtual threads
- Virtual threads from the “socket up” (Project Níma)
- Worked in close collaboration with Project Loom JDK developers

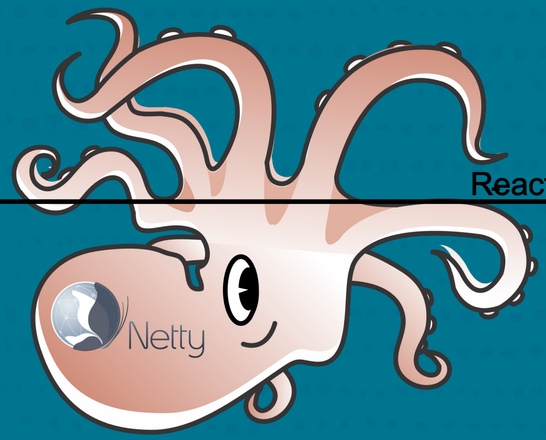
- Other frameworks support virtual threads by retrofitting

Helidon 4 – Why it is fundamentally different?

Reactive frameworks offloading to VTs

Imperative API

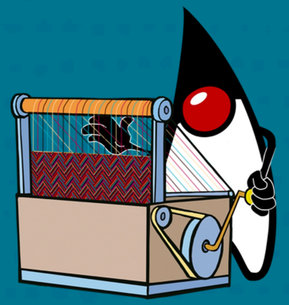
Offloading to Virtual Thread



Reactive API

Helidon 4

Imperative API

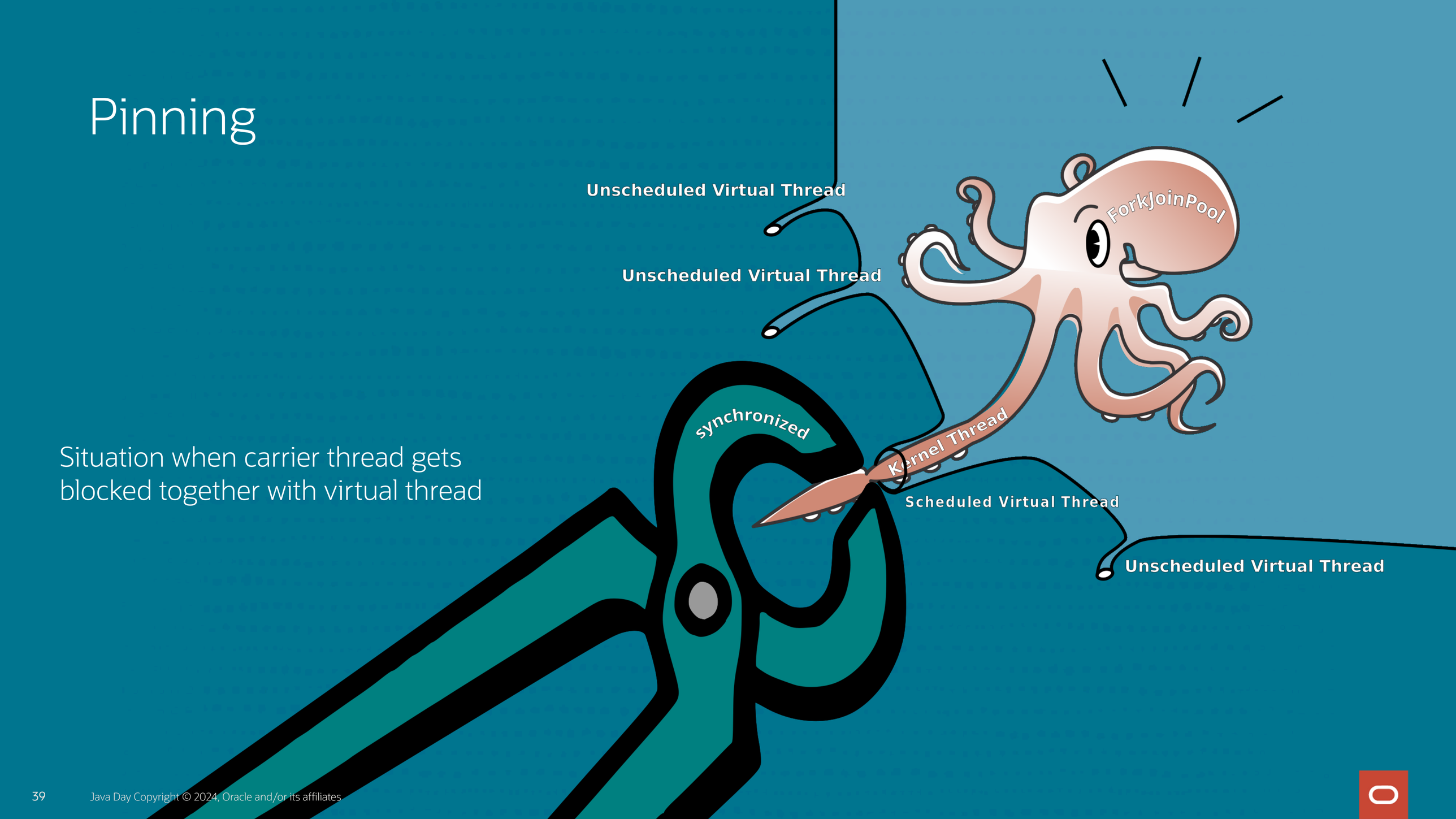


Pinning

Achilles' heel of Virtual Threads?

Pinning

Situation when carrier thread gets blocked together with virtual thread



Pinning

- Usual suspect is usage of **synchronized**
  - Not always harmful
  - Short-lived operations like in-memory operations are not harmful
- Carrier thread pool compensates by spinning up new carrier thread
  - Leads to degraded performance in case it happens frequently
- Usage of ReentrantLock does NOT cause pinning
  - ReentrantLock is VirtualThread friendly

Pinning example

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread.ofVirtual().start(() -> {
            synchronized (new Main()) {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {}
            }
        }).join();
    }
}
```


Pinning Detection #1

- jdk.tracePinnedThreads system property
  - Easy to use
  - -Djdk.tracePinnedThreads=short prints just problematic frame
  - Not recommended for production use with Helidon

```
➜ java -Djdk.tracePinnedThreads Main.java
Thread[#29,ForkJoinPool-1-worker-1,5,CarrierThreads]
    java.base/java.lang.VirtualThread$VThreadContinuation.onPinned(VirtualThread.java:183)
    java.base/jdk.internal.vm.Continuation.onPinned0(Continuation.java:393)
    java.base/java.lang.VirtualThread.parkNanos(VirtualThread.java:621)
    java.base/java.lang.VirtualThread.sleepNanos(VirtualThread.java:793)
    java.base/java.lang.Thread.sleep(Thread.java:507)
    me.daniel.se.quickstart.Main.lambda$main$0(Main.java:8) <== monitors:1
    java.base/java.lang.VirtualThread.run(VirtualThread.java:309)
```

Pinning Detection #2

- JDK Flight Recorder (JFR) **jdk.VirtualThreadPinned** event
  - Easy to use
  - Enabled by default on when operation takes longer 20ms

```
➜ java -XX:StartFlightRecording:jdk.VirtualThreadPinned#enabled=true,filename=pinning.jfr Main.java

➜ jfr print --events jdk.VirtualThreadPinned pinning.jfr
jdk.VirtualThreadPinned {
  startTime = 15:28:37.594 (2024-03-01)
  duration = 99.1 ms
  eventThread = "" (javaThreadId = 32, virtual)
  stackTrace = [
    java.lang.VirtualThread.parkOnCarrierThread(boolean, long) line: 677
    java.lang.VirtualThread.parkNanos(long) line: 636
    java.lang.VirtualThread.sleepNanos(long) line: 793
    java.lang.Thread.sleep(long) line: 507
    me.daniel.se.quickstart.Main.lambda$main$0() line: 8
    ...
  ]
}
```

Fixing pinning issue

- Offloading to physical thread
  - Universal solution for long running jobs, JNI with internal locking or legacy libraries
- Avoiding sychronized
  - Use ReentrantLock instead

Offloading in Helidon MP

org.glassfish.jersey.server.ManagedAsync

```java
@GET
@Path("/virtual")
public ThreadInfo getOnVirtualThread() {
    Thread t = Thread.currentThread();
    return new ThreadInfo(t.getName(), t.isVirtual());
}

public record ThreadInfo(String name, boolean virtual) {}
```

```java
@GET
@Path("/platform")
@ManagedAsync
public ThreadInfo getOnPlatformThread() {
    Thread t = Thread.currentThread();
    return new ThreadInfo(t.getName(), t.isVirtual());
}

public record ThreadInfo(String name, boolean virtual) {}
```

```
➜  offloading-mp curl -s localhost:8080/thread-name/ma/virtual | jq
{
  "name": "[0x4bb051b6 0x393d85a8] WebServer socket",
  "virtual": true
}
```

```
➜  offloading-mp curl -s localhost:8080/thread-name/ma/platform | jq
{
  "name": "jersey-server-managed-async-executor-0",
  "virtual": false
}
```

Offloading in Helidon SE

```
public static void main(String[] args) {
    var platformThreadPool = ThreadPoolSupplier.builder() Builder
            .virtualThreads(false)
            .build() ThreadPoolSupplier
            .get();

    WebServer.builder() Builder
            .port(8080)
            .routing(r → r
                    .get("/platform", (req, res) → {
                        res.send(platformThreadPool
                                .submit(() → {
                                    var t = Thread.currentThread();
                                    return new ThreadInfo(t.getName(), t.isVirtual());
                                })
                                .get()); //Block VT till offloaded work is done
                    }))
            .build() WebServer
            .start();
}
public record ThreadInfo(String name, boolean virtual) { } 1 usage
```

Future of synchronized

- Frameworks and libraries are replacing synchronized
- Pinning-less synchronize in Java is just around the corner

Project Loom Early-Access Builds

These builds are intended for developers looking to "kick the tyres" and provide feedback on using the API or by sending bug reports.

Warning: This build is based on an incomplete version of JDK 23.

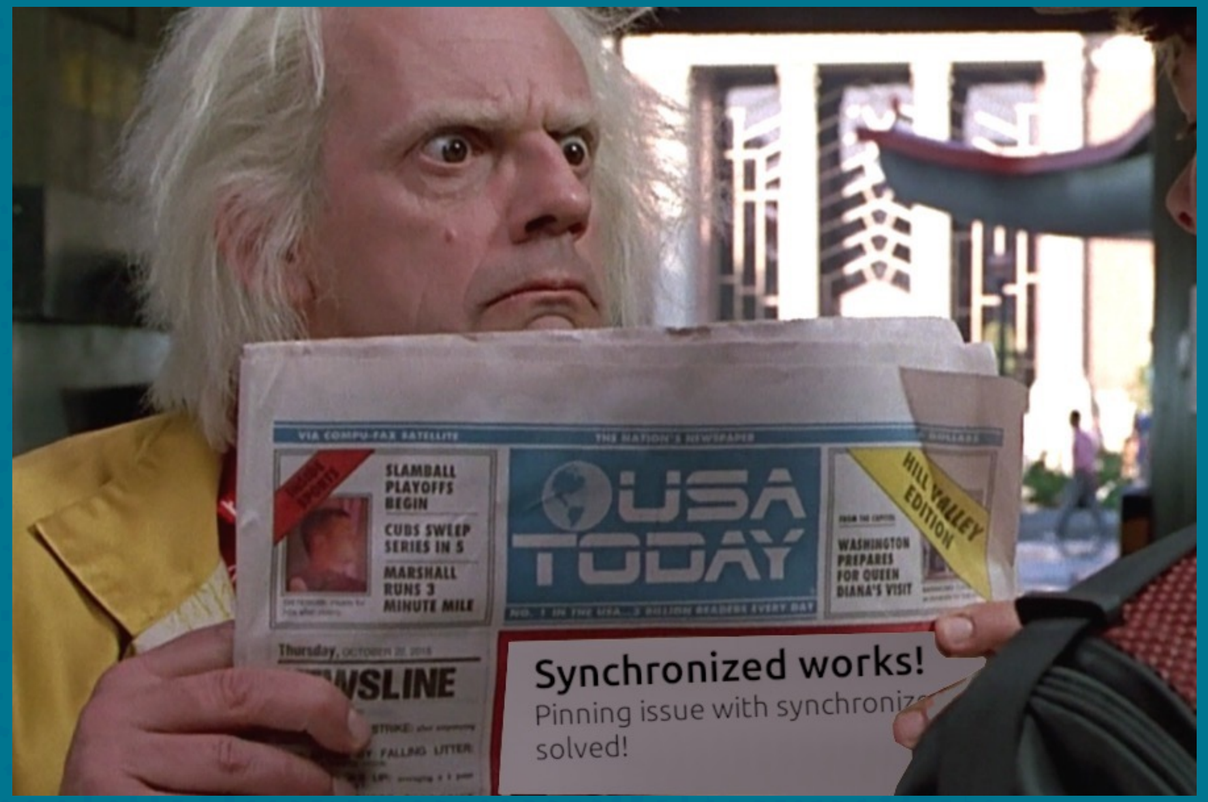
Build 23-loom+2-48 (2024/2/20)

These early-access builds are provided under the GNU General Public License, version 2, with the Classpath Exception.

Linux/AArch64	tar.gz (sha256)	200430266 bytes
Linux/x64	tar.gz (sha256)	202635705
macOS/AArch64	tar.gz (sha256)	196124359
macOS/x64	tar.gz (sha256)	198424761
Windows/x64	zip (sha256)	200759650

Notes

- These builds are based on jdk-23+10
- This build improves the implementation of Java monitors (synchronized methods) to work better with virtual threads.



No pinning with synchronize!

- Download EA build from https://jdk.java.net/loom and try it!

```
➜  /home/daniel/.sdkman/candidates/java/21.0.2-open/bin/java -Djdk.tracePinnedThreads=short Main.java
Thread[#29,ForkJoinPool-1-worker-1,5,CarrierThreads]
    Main.lambda$main$0(Main.java:6) <== monitors:1

➜  /opt/jdk/openjdk-23-loom+2-48/bin/java -Djdk.tracePinnedThreads=short Main.java
```

Pinning vs. Blocking in Reactive code

Pinning on Virtual Threads

Annoying

Blocking in Reactive Code

Destructive

More about Helidon …

@helidon_project

helidon.io

medium.com/helidon

github.com/helidon-io/helidon

youtube.com/Helidon_Project

helidon.slack.com