What the heck is Project Loom?

Deepu K Sasidharan

@deepu105 | deepu.tech

auth0
by Okta

Hi, I'm Deepu K Sasidharan

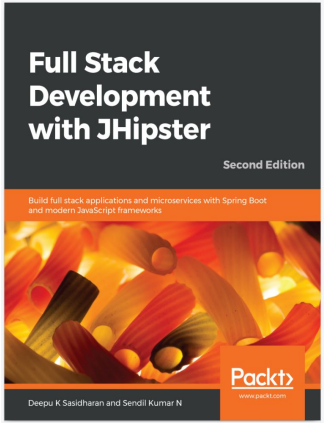
- ➔ *JHipster co-lead developer*
- ➔ *Java Champion*
- ➔ *Creator of KDash, JDL Studio*
- ➔ *Developer Advocate @ Auth0 by Okta*
- ➔ *OSS aficionado, polyglot dev, author, speaker*


@deepu105@mastodon.online


deepu.tech


@deepu105


deepu05



auth0
by Okta

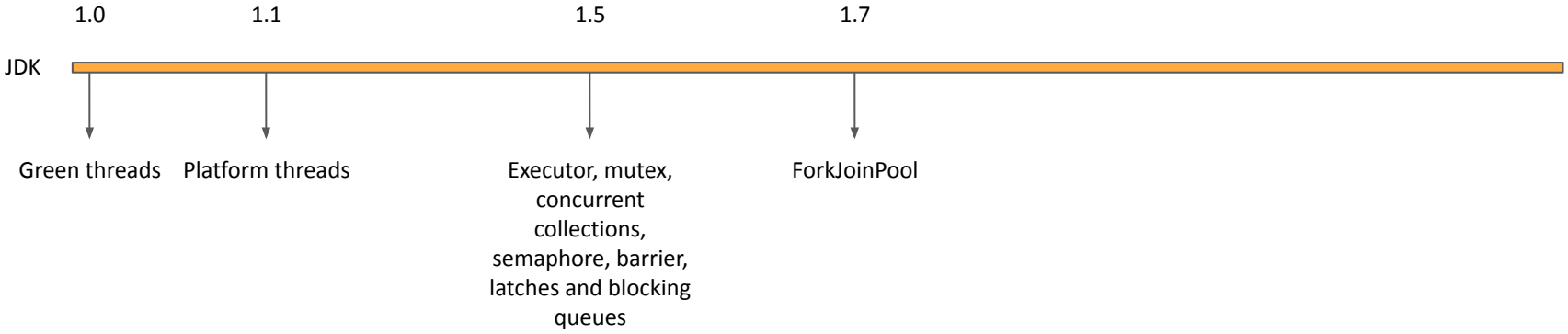
Concurrency in Java

https://deepu.tech/concurrency-in-modern-languages-java/

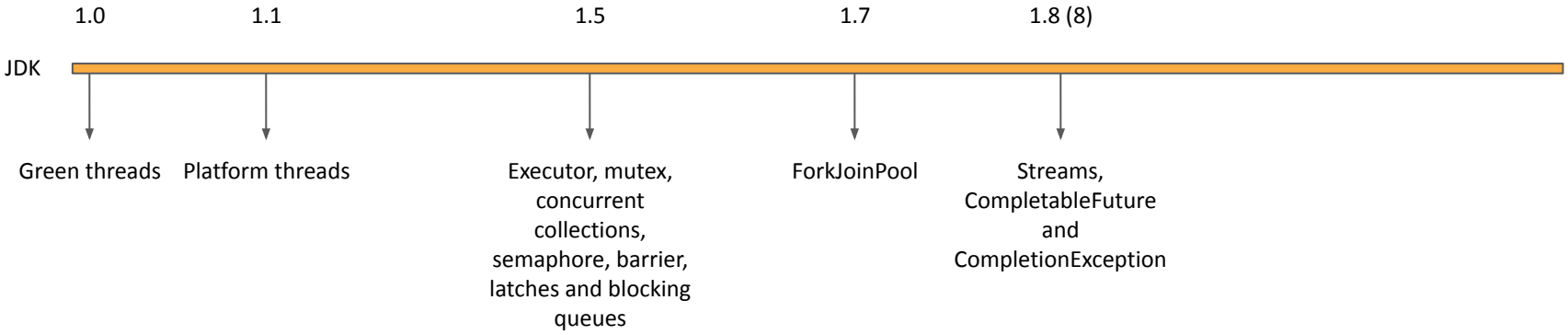
JDK Evolution

JDK Evolution

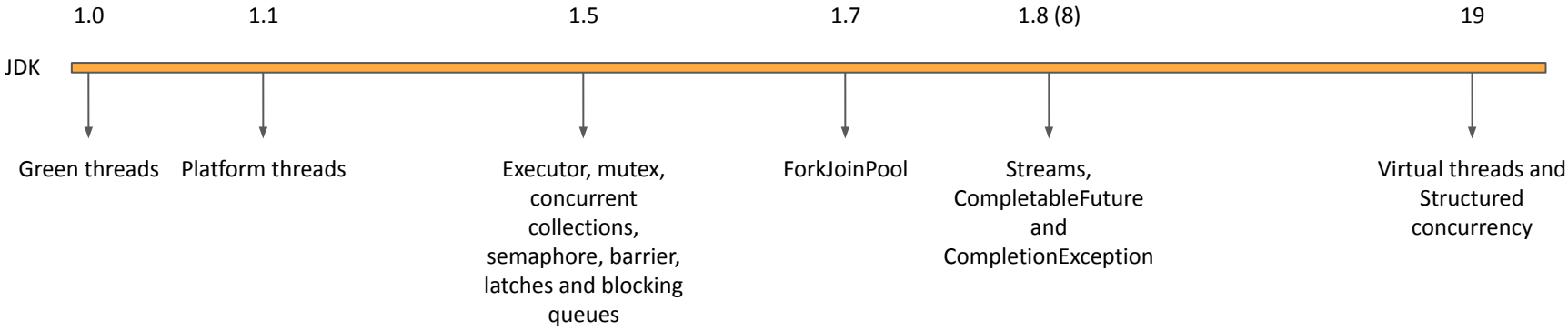
JDK Evolution



JDK Evolution



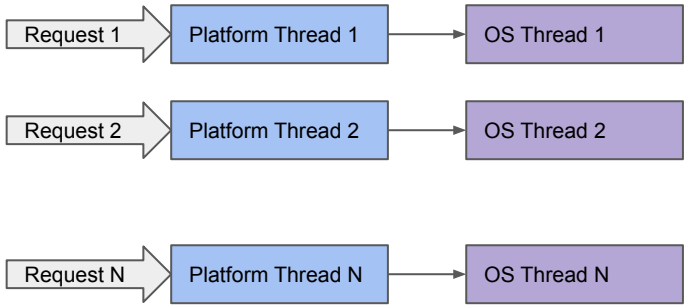
JDK Evolution



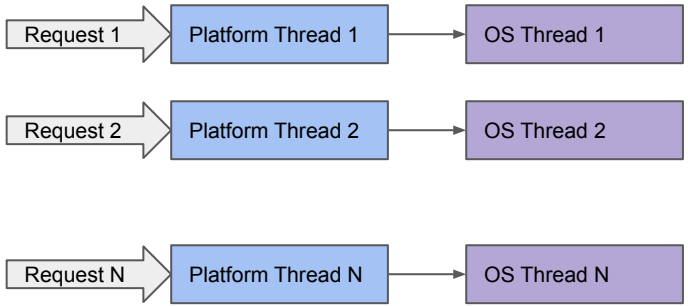
Platform Threads

- Platform threads == OS threads
- Platforms threads are mapped 1:1 to OS threads

Thread-per-request model



Thread-per-request model

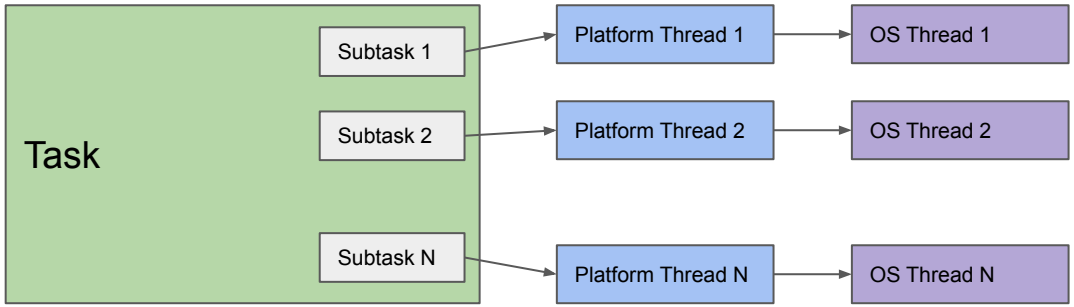


Little's law

$\lambda = L/W$

λ = Throughput (average rate of requests)
L = Average concurrency (number of requests concurrently processed by the server)
W = Latency (average duration of processing each request)

Parallel processing



- Should handle data races and data corruption
- Thread synchronization might be needed
- Thread leaks and cancellation delays
- Fragile
- A lot of responsibility on the developer

Project Loom

https://developer.okta.com/blog/2022/08/26/state-of-java-project-loom

Project Loom

Project Loom aims to drastically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications that make the best use of available hardware.

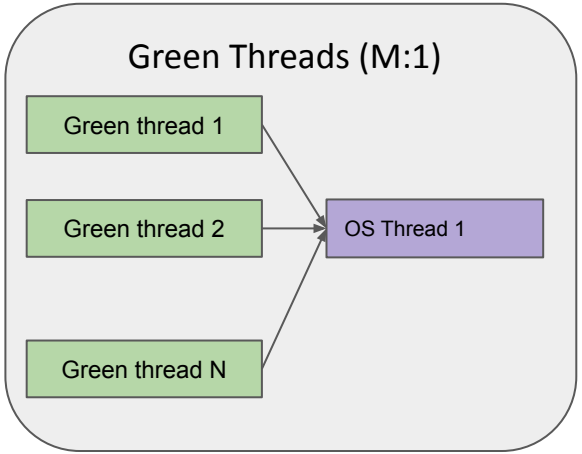
— Ron Pressler (Tech lead, Project Loom)

Virtual threads

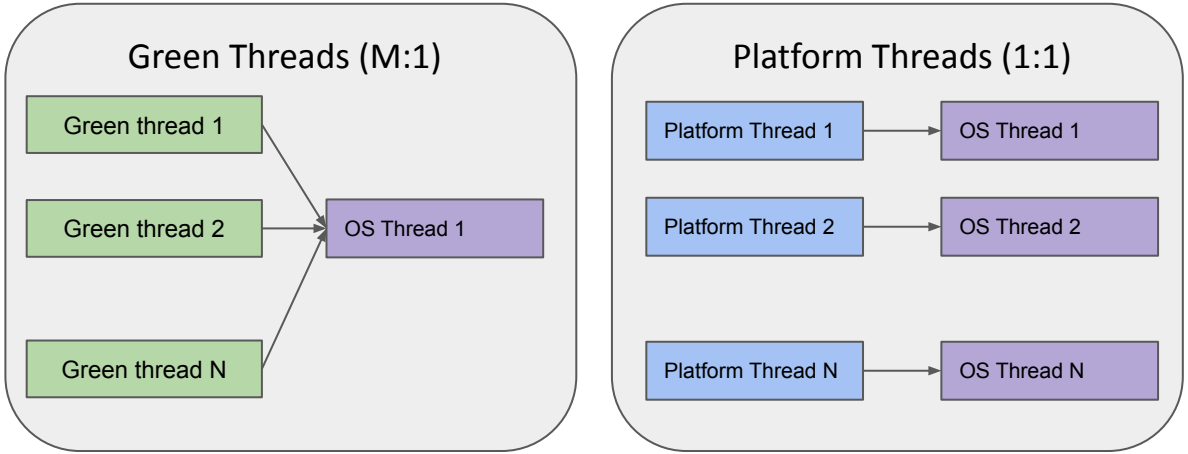
a.k.a User mode threads

a.k.a Coroutines

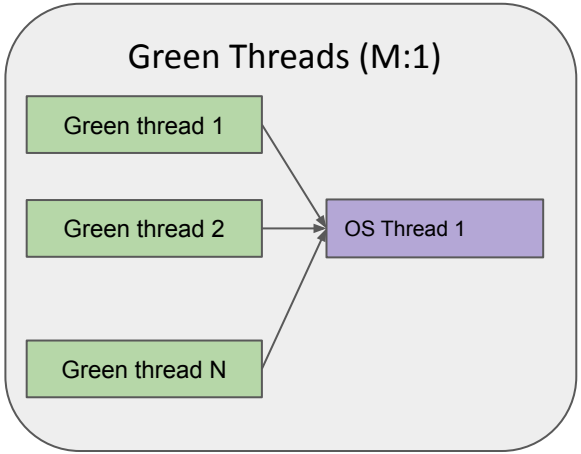
Green threads mapping

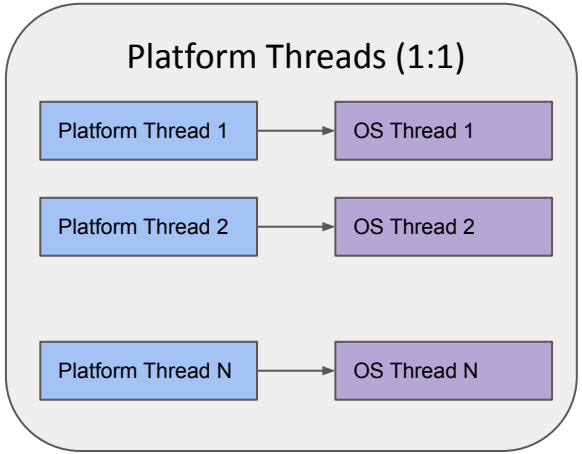


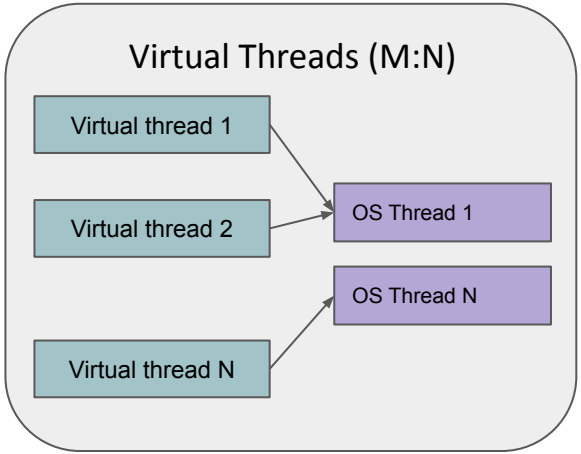
Platform threads mapping



Virtual threads mapping







Goroutines

```
go func() {
    println("Hello, Goroutines!")
}()
```

Kotlin coroutines

```
runBlocking {
    launch {
        println("Hello, Kotlin coroutines!")
    }
}
```

Java virtual thread

```
Thread.startVirtualThread(() -> {
    System.out.println("Hello, Project Loom!");
});
```

Virtual thread features

- It is a `Thread` in code, runtime, debugger, and profiler
- It's a Java entity and not a wrapper around an OS thread
- Creating and blocking them are cheap operations
- They should not be pooled
- Virtual threads use a work-stealing `ForkJoinPool` scheduler
- Pluggable schedulers can be used for asynchronous programming
- A virtual thread will have its own stack memory
- The virtual threads API is very similar to platform threads and hence easier to adopt/migrate

Total number of platform threads

```
var counter = new AtomicInteger();
while (true) {
    new Thread(() -> {
        int count = counter.incrementAndGet();
        System.out.println("Thread count = " + count);
        LockSupport.park();
    }).start();
}
```

Total number of virtual threads

```
var counter = new AtomicInteger();
while (true) {
    Thread.startVirtualThread(() -> {
        int count = counter.incrementAndGet();
        System.out.println("Thread count = " + count);
        LockSupport.park();
    });
}
```

Task throughput for platform threads

```
try (var executor = Executors.newThreadPerTaskExecutor(Executors.defaultThreadFactory())) {
    IntStream.range(0, 100_000).forEach(i -> executor.submit(() -> {
        Thread.sleep(Duration.ofSeconds(1));
        System.out.println(i);
        return i;
    }));
}

# 'newThreadPerTaskExecutor' with 'defaultThreadFactory'
0:18.77 real,   18.15 s user,   7.19 s sys,     135% 3891pu,    0 amem,         743584 mmem
# 'newCachedThreadPool' with 'defaultThreadFactory'
0:11.52 real,   13.21 s user,   4.91 s sys,     157% 6019pu,    0 amem,         2215972 mmem
```

Task throughput for virtual threads

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 100_000).forEach(i -> executor.submit(() -> {
        Thread.sleep(Duration.ofSeconds(1));
        System.out.println(i);
        return i;
    }));
}
```

```
0:02.62 real,   6.83 s user,    1.46 s sys,     316% 14840pu,   0 amem,         350268 mmem
```

JMH Benchmarks

```
# Throughput (more is better)
Benchmark                                  Mode  Cnt   Score   Error  Units
LoomBenchmark.platformThreadPerTask       thrpt    5   0.362 ± 0.079  ops/s
LoomBenchmark.platformThreadPool          thrpt    5   0.528 ± 0.067  ops/s
LoomBenchmark.virtualThreadPerTask        thrpt    5   1.843 ± 0.093  ops/s

# Average time (less is better)
Benchmark                                  Mode  Cnt   Score   Error  Units
LoomBenchmark.platformThreadPerTask        avgt    5   5.600 ± 0.768   s/op
LoomBenchmark.platformThreadPool           avgt    5   3.887 ± 0.717   s/op
LoomBenchmark.virtualThreadPerTask         avgt    5   1.098 ± 0.020   s/op
```

https://github.com/deepu105/java-loom-benchmarks

More benchmarks

- An interesting benchmark using ApacheBench on GitHub by Elliot Barlas
- A benchmark using Akka actors on Medium by Alexander Zakusylo
- JMH benchmarks for I/O and non-I/O tasks on GitHub by Colin Cachia

Structured concurrency

Without structured concurrency

```
void handleOrder() throws ExecutionException, InterruptedException {
    try (var esvc = new ScheduledThreadPoolExecutor(8)) {
        Future<Integer> inventory = esvc.submit(() -> updateInventory());
        Future<Integer> order = esvc.submit(() -> updateOrder());

        int theInventory = inventory.get();    // Join updateInventory
        int theOrder = order.get();            // Join updateOrder

        System.out.println("Inventory " + theInventory + " updated for order " + theOrder);
    }
}
```

Without structured concurrency

```
void handleOrder() throws ExecutionException, InterruptedException {
    try (var esvc = new ScheduledThreadPoolExecutor(8)) {
        Future<Integer> inventory = esvc.submit(() -> updateInventory()); // failed
        Future<Integer> order = esvc.submit(() -> updateOrder()); // runs in background

        int theInventory = inventory.get();    // Join updateInventory // fails
        int theOrder = order.get();            // Join updateOrder // unreachable

        System.out.println("Inventory " + theInventory + " updated for order " + theOrder);
    }
}
```

Without structured concurrency

```
void handleOrder() throws ExecutionException, InterruptedException {
    try (var esvc = new ScheduledThreadPoolExecutor(8)) {
        Future<Integer> inventory = esvc.submit(() -> updateInventory()); // expensive task
        Future<Integer> order = esvc.submit(() -> updateOrder()); // failed

        int theInventory = inventory.get();    // Join updateInventory // task blocked
        int theOrder = order.get();            // Join updateOrder // will fail

        System.out.println("Inventory " + theInventory + " updated for order " + theOrder);
    }
}
```

Without structured concurrency

```
void handleOrder() throws ExecutionException, InterruptedException { // interrupted
    try (var esvc = new ScheduledThreadPoolExecutor(8)) {
        Future<Integer> inventory = esvc.submit(() -> updateInventory()); // runs in bg
        Future<Integer> order = esvc.submit(() -> updateOrder()); // runs in bg

        int theInventory = inventory.get();    // Join updateInventory
        int theOrder = order.get();            // Join updateOrder

        System.out.println("Inventory " + theInventory + " updated for order " + theOrder);
    }
}
```

Structured concurrency

```
void handleOrder() throws ExecutionException, InterruptedException {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        Future<Integer> inventory = scope.fork(() -> updateInventory());
        Future<Integer> order = scope.fork(() -> updateOrder());

        scope.join();           // Join both forks
        scope.throwIfFailed();  // ... and propagate errors

        // Here, both forks have succeeded, so compose their results
        System.out.println("Inventory " + inventory.resultNow() + " updated for order " +
order.resultNow());
    }
}
```

Structured concurrency

```
void handleOrder() throws ExecutionException, InterruptedException {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        Future<Integer> inventory = scope.fork(() -> updateInventory()); // failed
        Future<Integer> order = scope.fork(() -> updateOrder()); // cancelled

        scope.join();           // Join both forks
        scope.throwIfFailed();  // ... and propagate errors

        // Here, both forks have succeeded, so compose their results
        System.out.println("Inventory " + inventory.resultNow() + " updated for order " +
order.resultNow());
    }
}
```

State of Project Loom

Impact for regular developers

- No breaking changes
- Very low API surface and hence easy to adopt/migrate
- Rely on underlying libraries to switch to virtual threads
- Debugging virtual threads would need some getting used to
- Can easily switch to virtual threads from thread pools
- Structured concurrency could help to eliminate a lot of failsafe code
- At the moment need to use preview and incubator modules
- Some unlearning to do (no pooling, no reusing, no shared pool executors)
- Proliferation of virtual threads in simple use cases.

Impact for libraries

- Performance and throughput increases
- Early adoption
- Simpler codebase
- Server software like tomcat, Undertow and Jetty will see improvements
- Frameworks like Spring, Micronaut and Quarkus will see improvements
- Libraries like RxJava and Akka might benefit from structured concurrency
- Asynchronous and reactive programming will still be around but in many use cases virtual threads could replace them and give same benefits with less complexity

Early adoption

- GraalVM
  - Support added (https://github.com/oracle/graal/pull/4802)
- Quarkus
  - Support added (https://github.com/quarkusio/quarkus/pull/24942)
- Micronaut
  - Being discussed (https://github.com/micronaut-projects/micronaut-core/issues/7724)
- Spring
  - https://spring.io/blog/2022/10/11/embracing-virtual-threads

Caveats

Resources

- https://www.infoq.com/articles/java-virtual-threads/
- https://inside.java/2020/08/07/loom-performance/
- http://cr.openjdk.java.net/~rpressler/loom/loom/sol1_part1.html
- https://foojay.io/today/thinking-about-massive-throughput-meet-virtual-threads/

Get the Slides

Thank You

Deepu K Sasidharan

@deepu105 | deepu.tech

https://deepu.tech/tags#java

https://developer.auth0.com