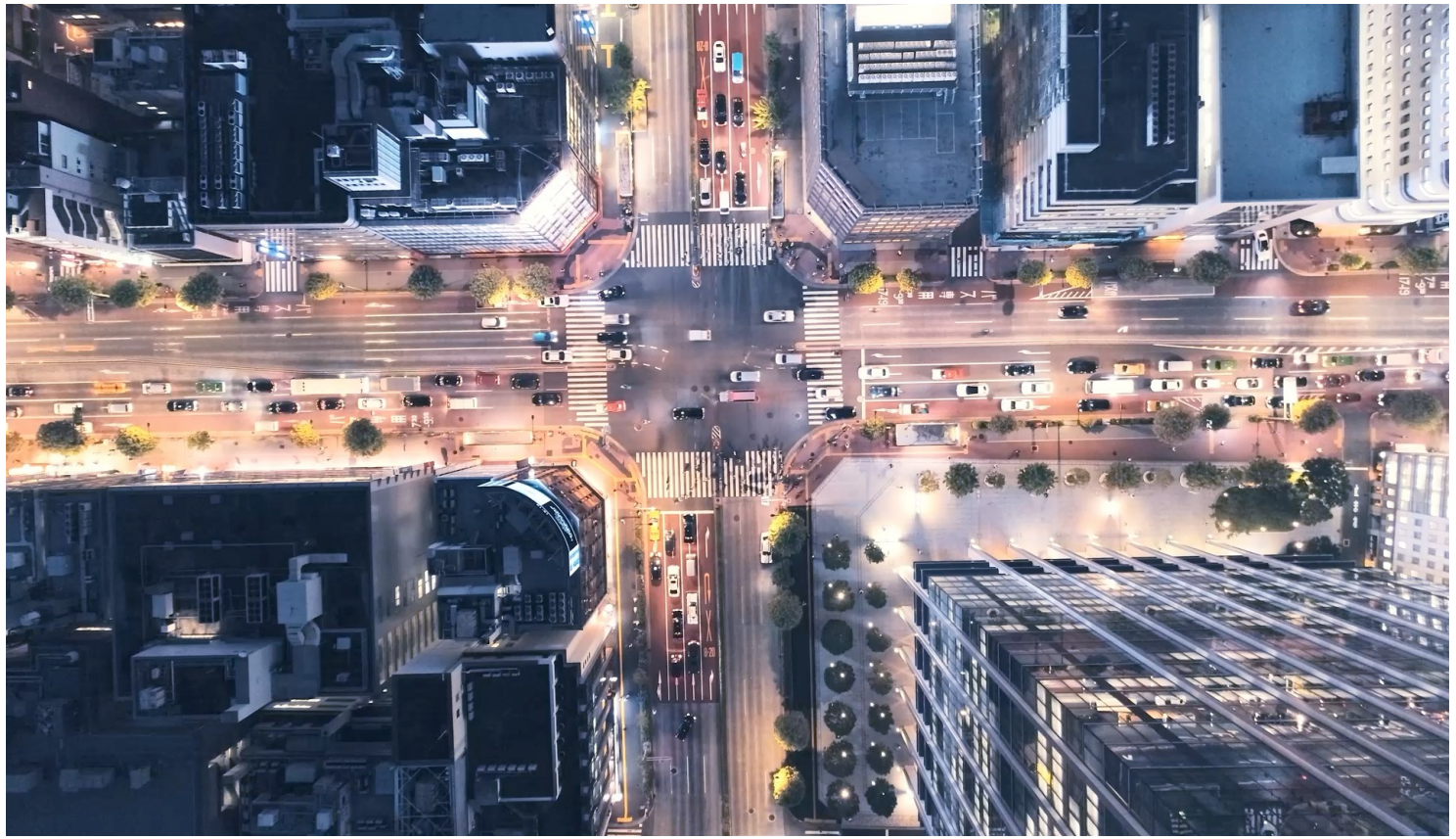
A Tale of Two Cities: Blocking Code VS Non-Blocking Code

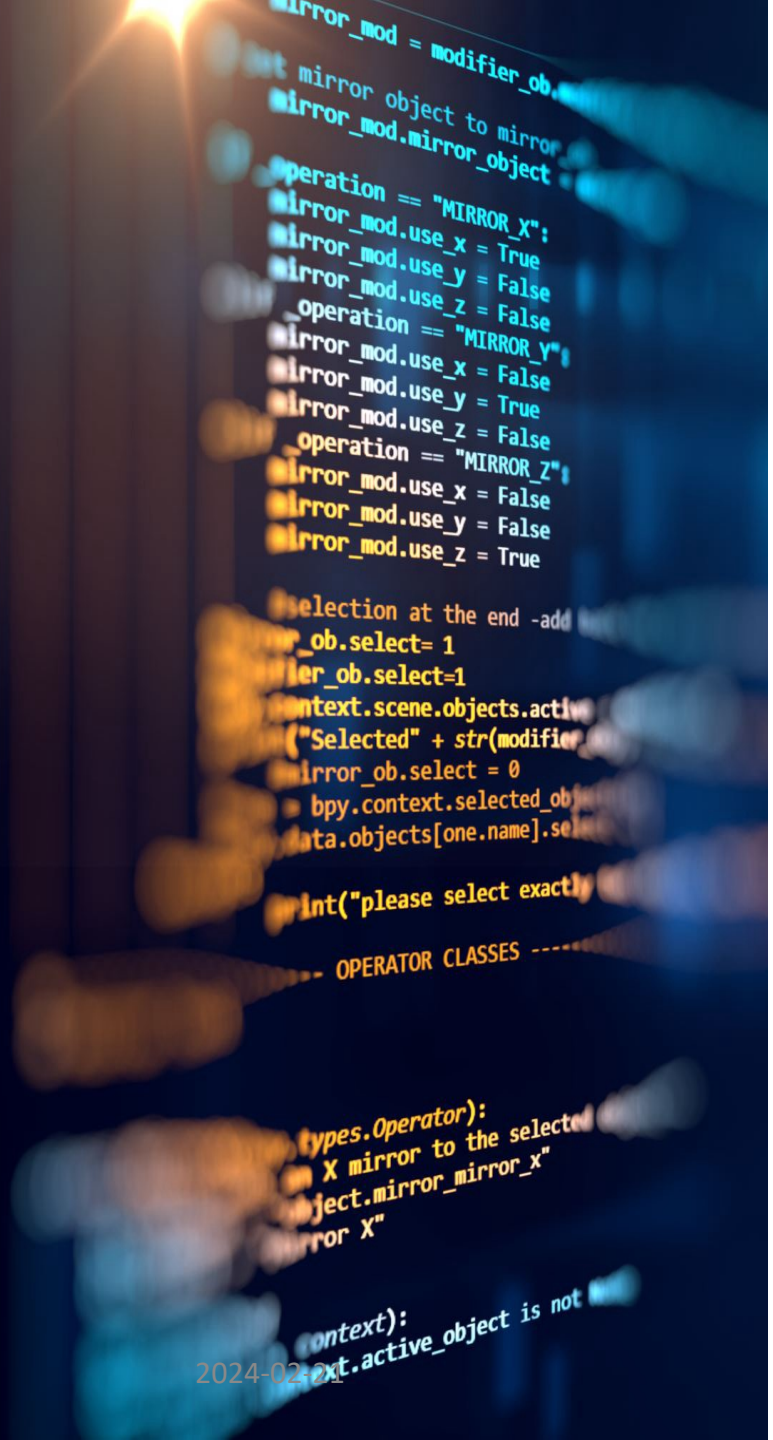
Bazlur Rahman

Staff Software Developer

DNAStack

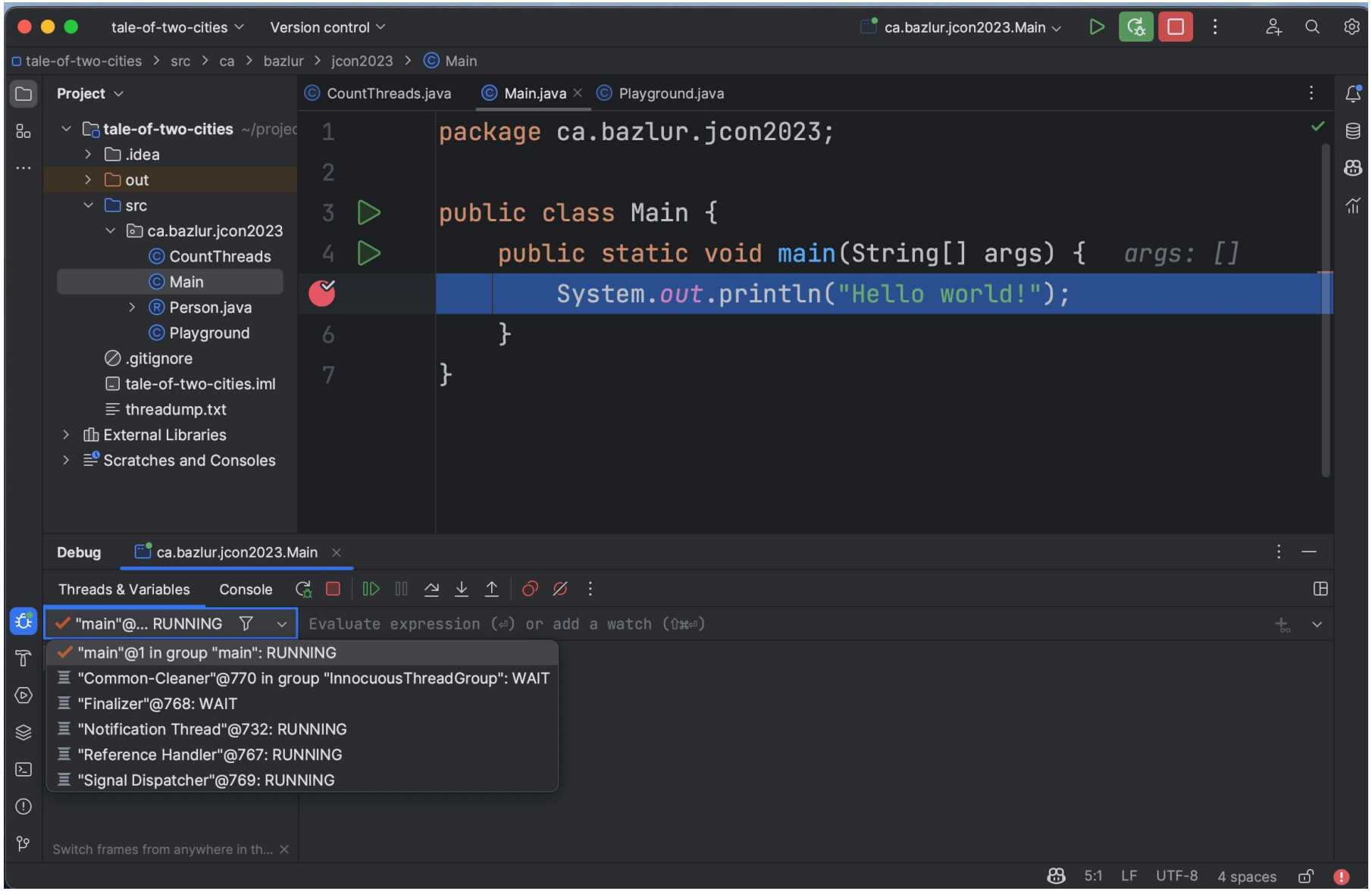


Agenda

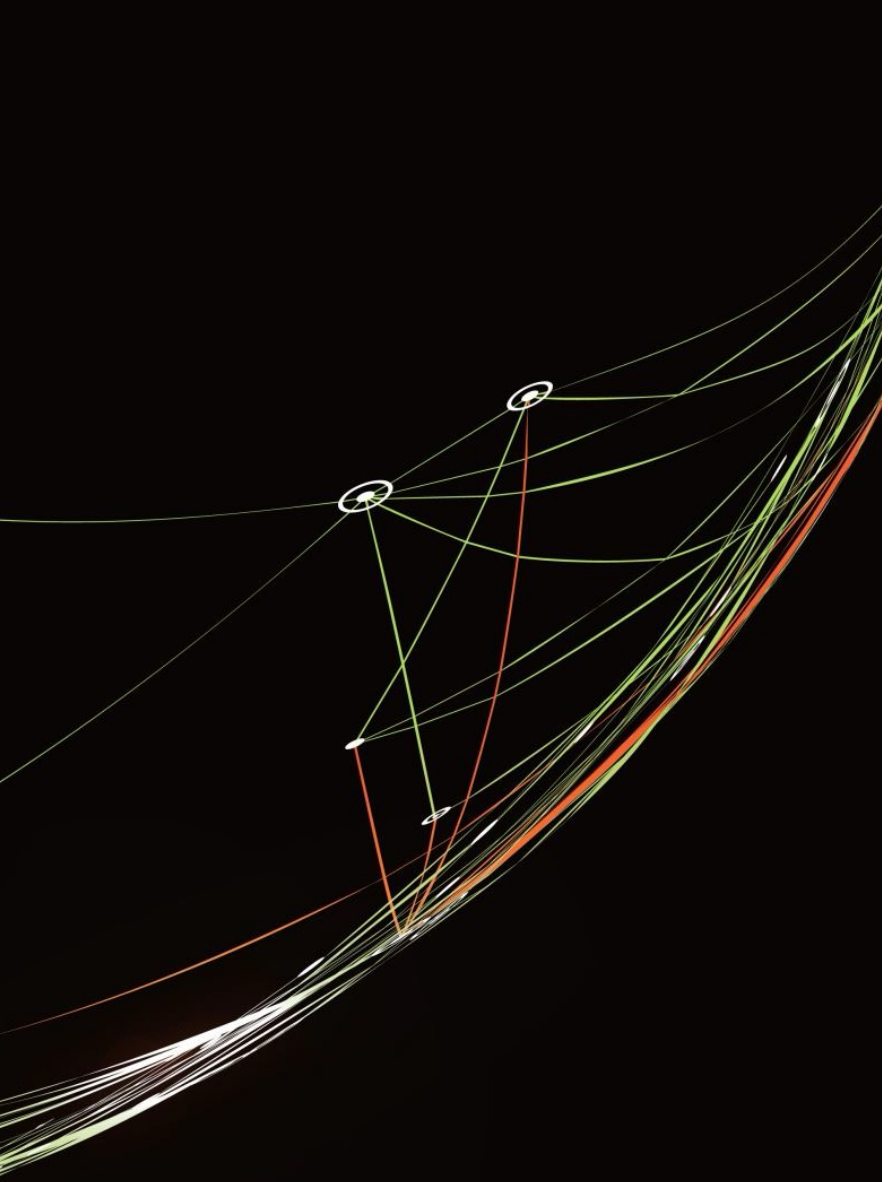


Threads—The Foundation of Java Concurrency

- Java is born with threads! When a Java program starts, the "main" thread is the birthplace of execution.
- Threads give birth to threads - Method calls spawn execution within the caller's thread
- Benefits we take for granted:
  - Structured control flow (think about how 'easy' writing sequential if/then/else logic is...)
  - Local variables for methods
  - Stack traces to solve problems
  - Ability to schedule work across CPUs


tale-of-two-cities
Version control
ca.bazlur.jcon2023.Main
tale-of-two-cities > src > ca > bazlur > jcon2023 > Main
Project
tale-of-two-cities ~/projec
.idea
out
src
ca.bazlur.jcon2023
CountThreads
Main
Person.java
Playground
.gitignore
tale-of-two-cities.iml
threadump.txt
External Libraries
Scratches and Consoles
CountThreads.java
Main.java
Playground.java
1 package ca.bazlur.jcon2023;
2
3 public class Main {
4 public static void main(String[] args) { args: []
5 System.out.println("Hello world!");
6 }
7 }
Debug
ca.bazlur.jcon2023.Main
Threads & Variables
Console
"main"@... RUNNING
Evaluate expression (⏎) or add a watch (⇧⌘⏎)
"main"@1 in group "main": RUNNING
"Common-Cleaner"@770 in group "InnocuousThreadGroup": WAIT
"Finalizer"@768: WAIT
"Notification Thread"@732: RUNNING
"Reference Handler"@767: RUNNING
"Signal Dispatcher"@769: RUNNING
Switch frames from anywhere in th...
5:1
LF
UTF-8
4 spaces

```
void main() throws InterruptedException {

    var thread = Thread.ofPlatform().start(() ->

            System.out.println("""
                    Hello from the brand new thread!
                    I was just born, but I could already go for a cup of coffee!
                    """));

    System.out.println("Main thread: Time to fetch some coffee!");

    thread.join();
}
```



Java Threads: A Wrapper Around OS Threads

- Java threads rely on operating system threads.
- OS-managed threads are often called "**Platform threads**."
- Pre-JDK 21 had a one-to-one correspondence between Java threads and OS threads.
- This approach could limit performance.

The Cost of Platform Threads

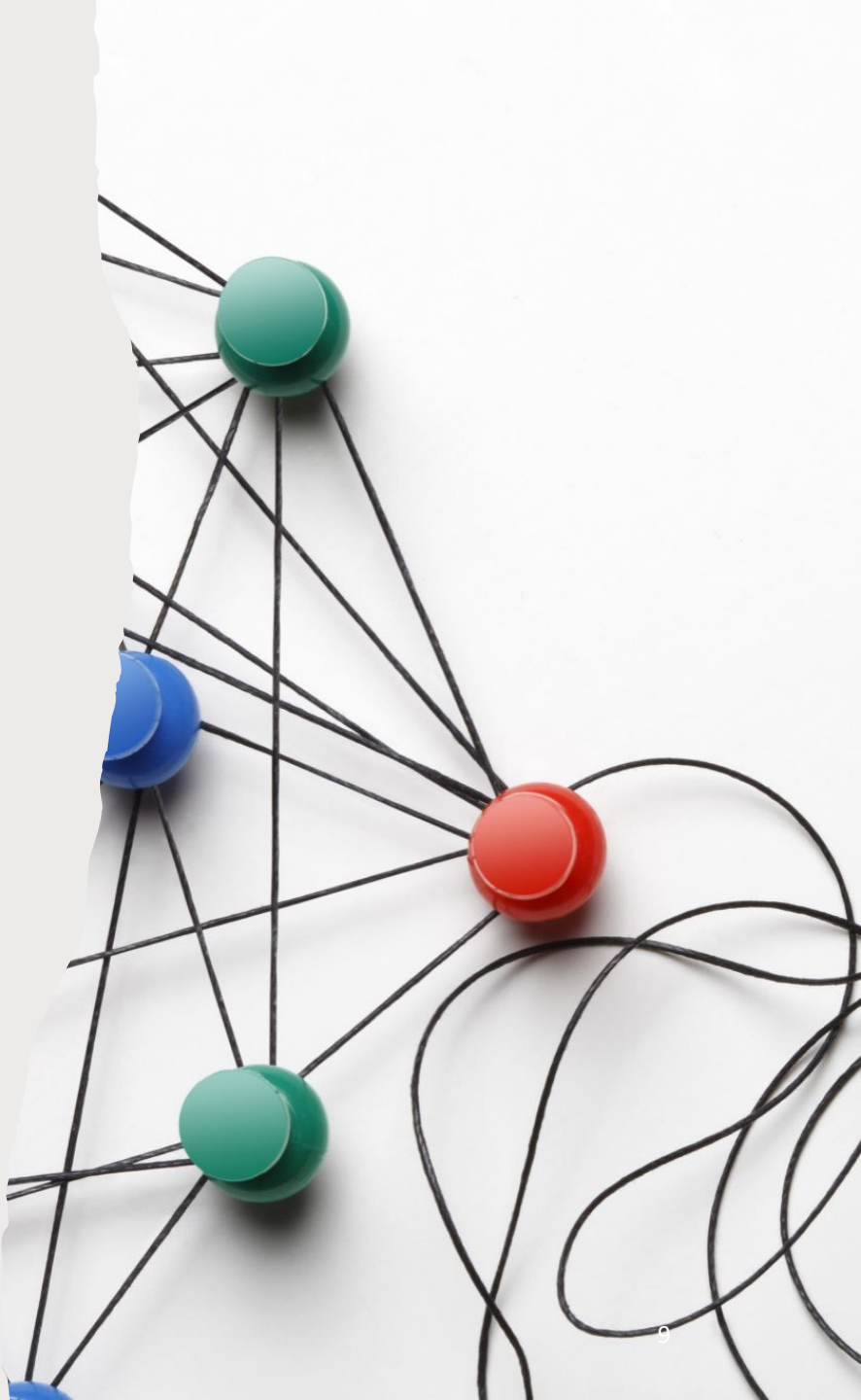
- OS thread creation takes time and significant resources.
- Large "pre-commitment" needed for stacks - often megabytes in size (can sometimes be tuned, but with risks).
- This introduces a practical cap on the number of concurrent threads in an application.

How Many Platform Threads?

```
void main() {
    var counter = new AtomicInteger();
    System.out.println("Welcome to the Thread Race Marathon!");

    while (true) {
        Thread.ofPlatform().start(() -> {
            int count = counter.incrementAndGet();
            if (count % 1000 == 0) {
                System.out.println(STR."We have created \{count} threads so far. Keep
going!");
            }
            LockSupport.park(); // Suspend the thread, simulating the racer taking a break
        });
    }
}
```

The Challenges of Shared-State Concurrency

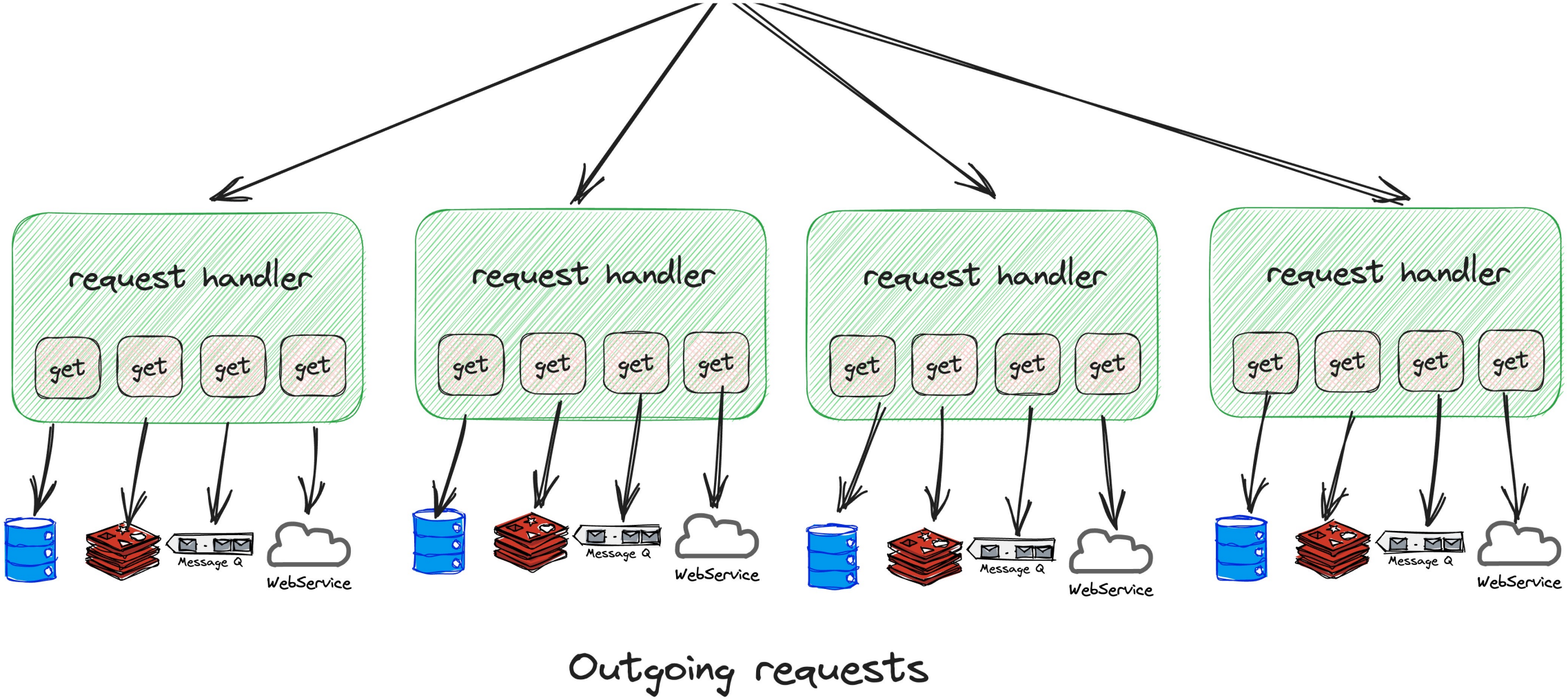
- While invaluable, threads become challenging when we manipulate shared data across threads.
- Issues to keep in mind:
  - Memory visibility (ensuring one thread sees another's changes)
  - Safeguarding against race conditions and data corruption
  - Mastering locks and synchronization for protection.

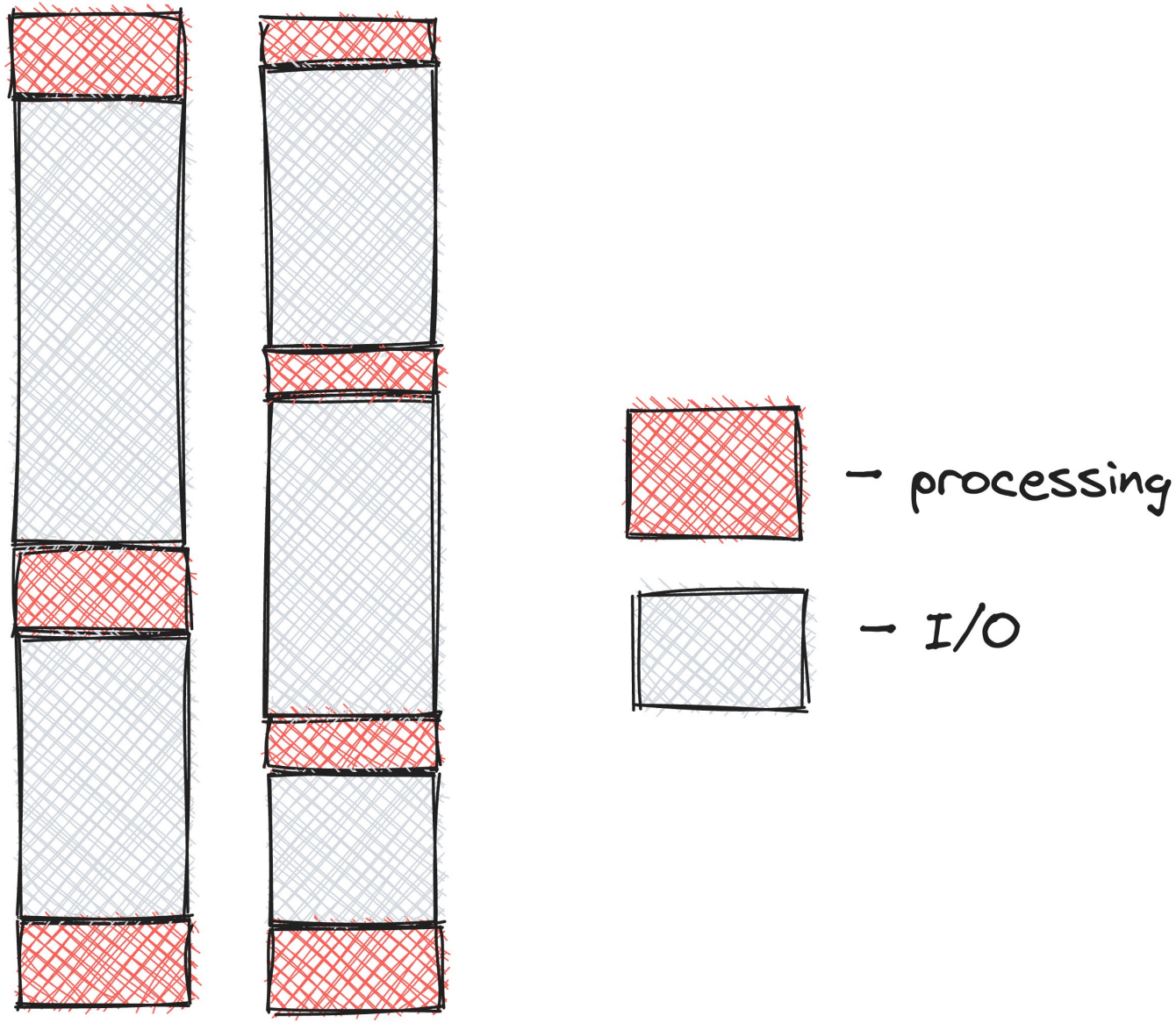


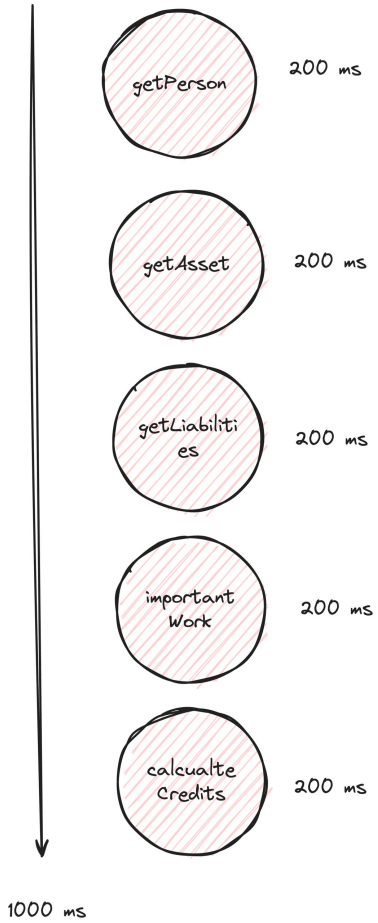
Beyond the Basics

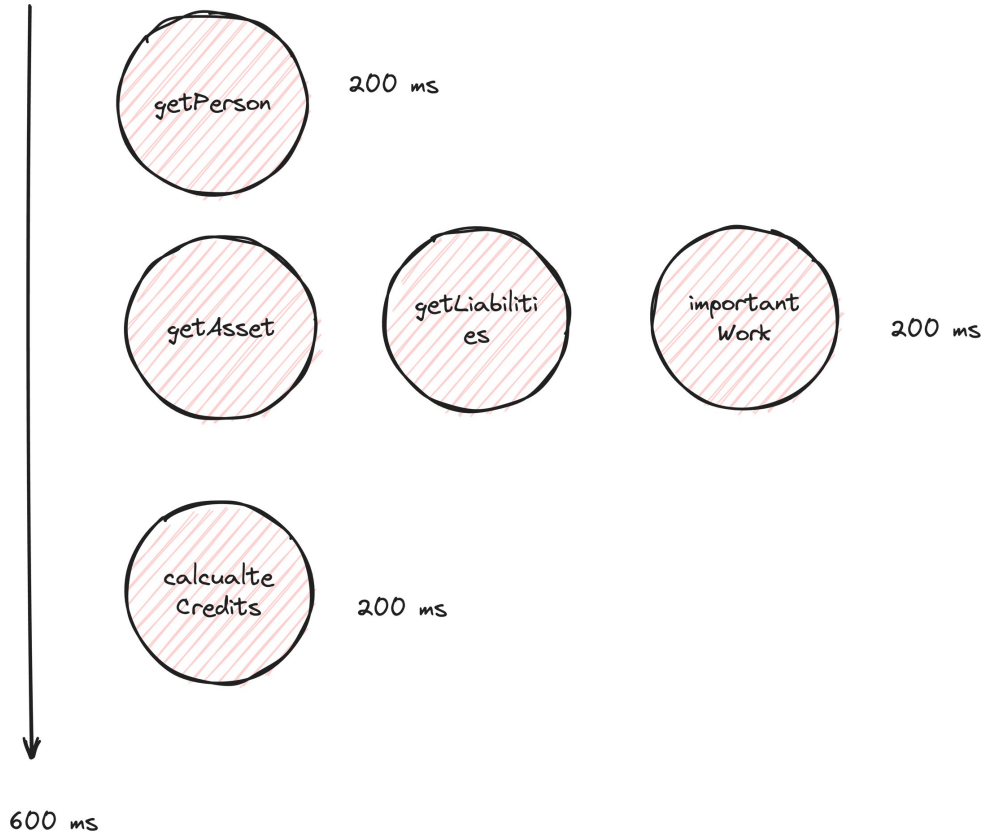
- Despite thread complexities, a huge amount of what we accomplish day-to-day relies on them working silently...
- Java concurrency is rich - we only highlight a part of the landscape today.
- Our focus today... how do we manage tasks that might take a while without tying up our precious threads?

Common Server Fanout




- processing
- I/O

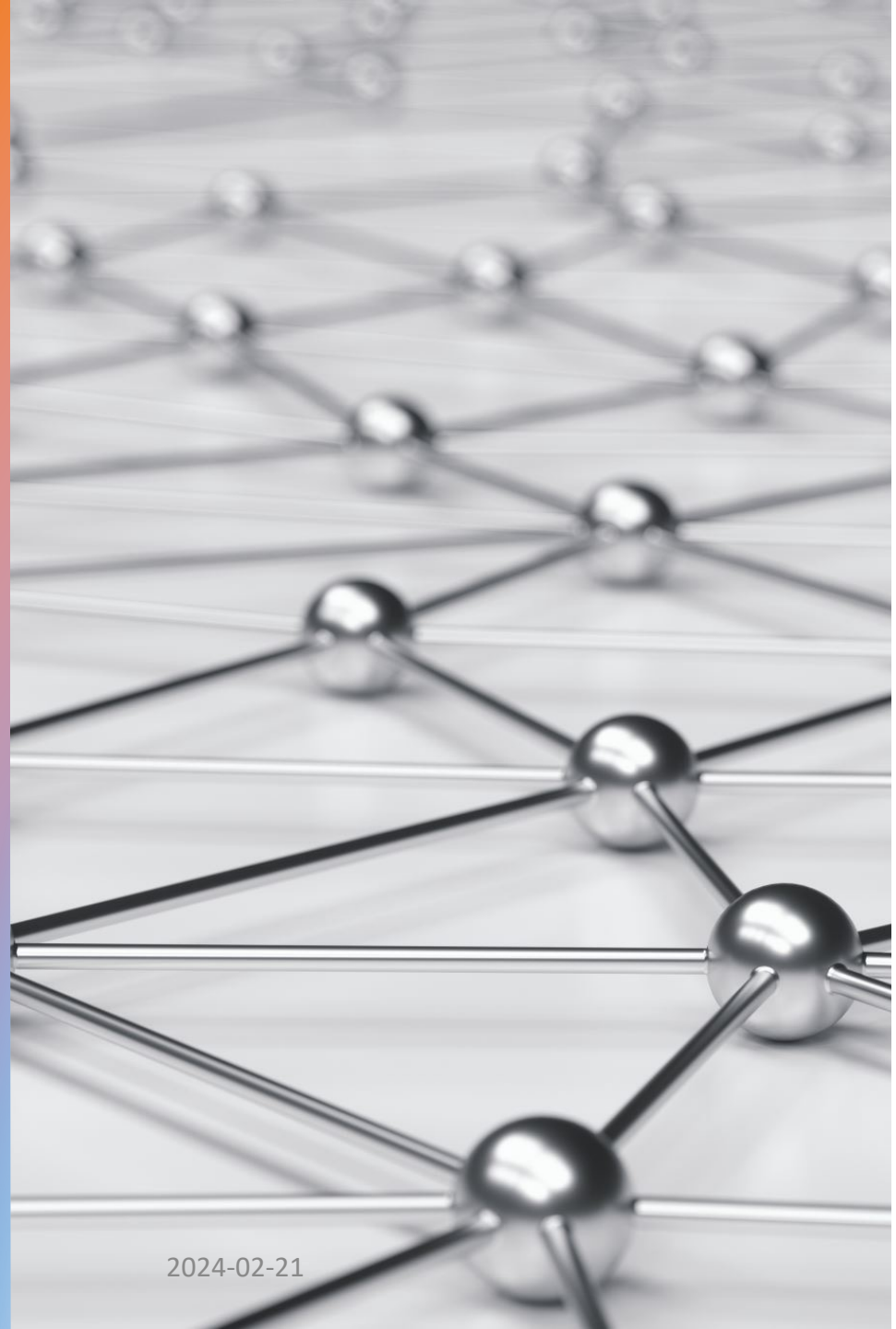

getPerson
200 ms
getAsset
200 ms
getLiabiliti
es
200 ms
important
Work
200 ms
calcualte
Credits
200 ms
1000 ms


getPerson
200 ms
getAsset
getLiabiliti
es
important
Work
200 ms
calcualte
Credits
200 ms
600 ms

Classical Implementation

```
var atomicReference = new AtomicReference<Person>();


Thread.ofPlatform().start(() -> {
    var person = getPerson(1L);
    atomicReference.set(person);
});
```

```java
Credit calculateCreditClassicalThreading(Long personId) throws InterruptedException {
    var person : Person = getPerson(personId);

    var assetsWrapper = new AtomicReference<List<Asset>>();
    var assetsThread : Thread = Thread.ofPlatform().unstarted(() -> {
        List<Asset> assets = getAssets(person);
        assetsWrapper.set(assets);
    });

    var liabilitiesWrapper = new AtomicReference<List<Liability>>();
    var liabilitiesThread : Thread = Thread.ofPlatform().unstarted(() -> {
        List<Liability> liabilities = getLiabilities(person);
        liabilitiesWrapper.set(liabilities);
    });

    var importantWorkThread : Thread = Thread.ofPlatform().unstarted(this::importantWork);

    assetsThread.start();
    liabilitiesThread.start();
    importantWorkThread.start();

    assetsThread.join();
    liabilitiesThread.join();

    var credit : Credit = calculateCredits(assetsWrapper.get(), liabilitiesWrapper.get());

    importantWorkThread.join();
    return credit;
}
```

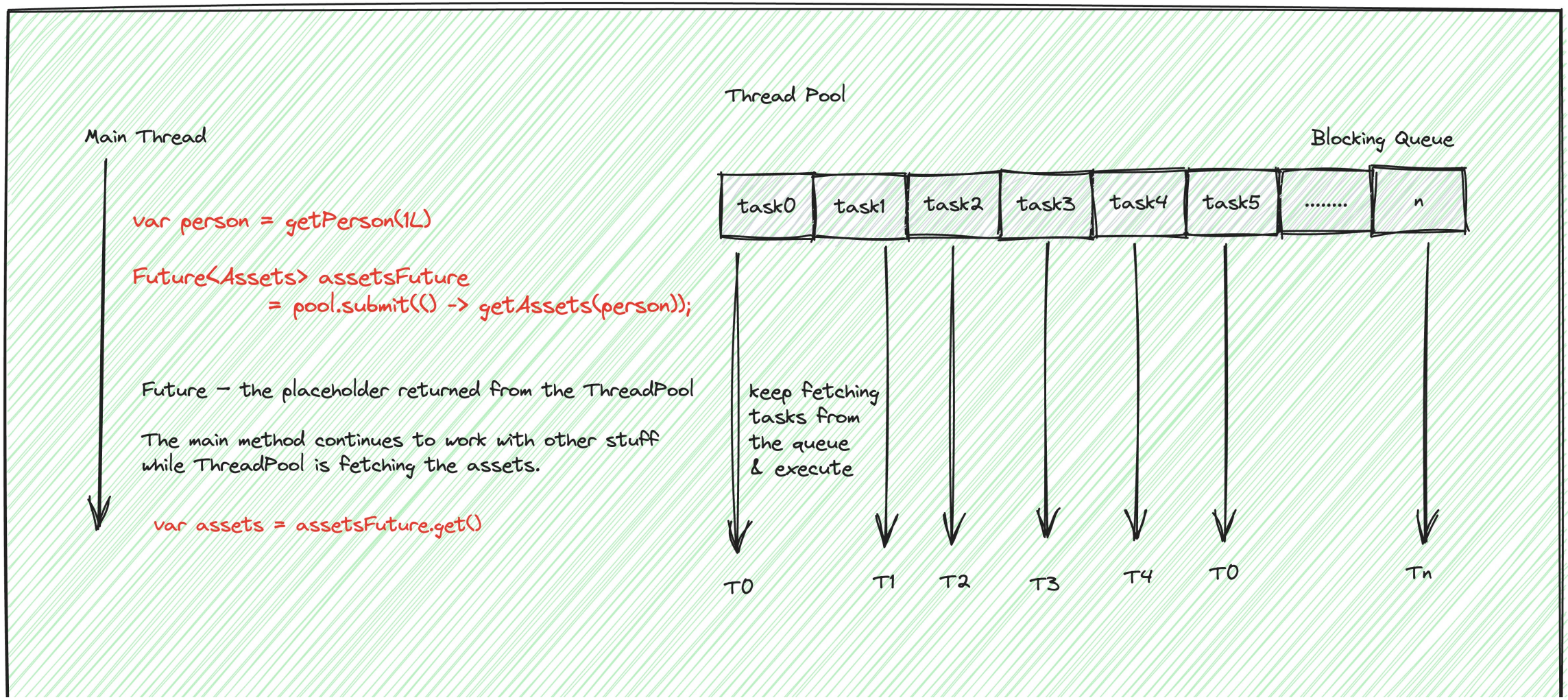


The Inception – Java's Early Concurrency Tools

- Future/Callable
  - This means to execute tasks asynchronously
  - The Future object represents the eventual result of the computation
  - The Callable interface provides a way to define the work to be done

```
Credit calculateCreditWithExecutor(Long personId) throws ExecutionException, InterruptedException {
    var person : Person = getPerson(personId);
    var assetsFuture : Future<List<...>> = executor.submit(() → getAssets(person));
    var liabilitiesFuture : Future<List<...>> = executor.submit(() → getLiabilities(person));
    executor.submit(this::importantWork);

    return calculateCredits(assetsFuture.get(), liabilitiesFuture.get());
}
```

How Traditional Thread Pool Works



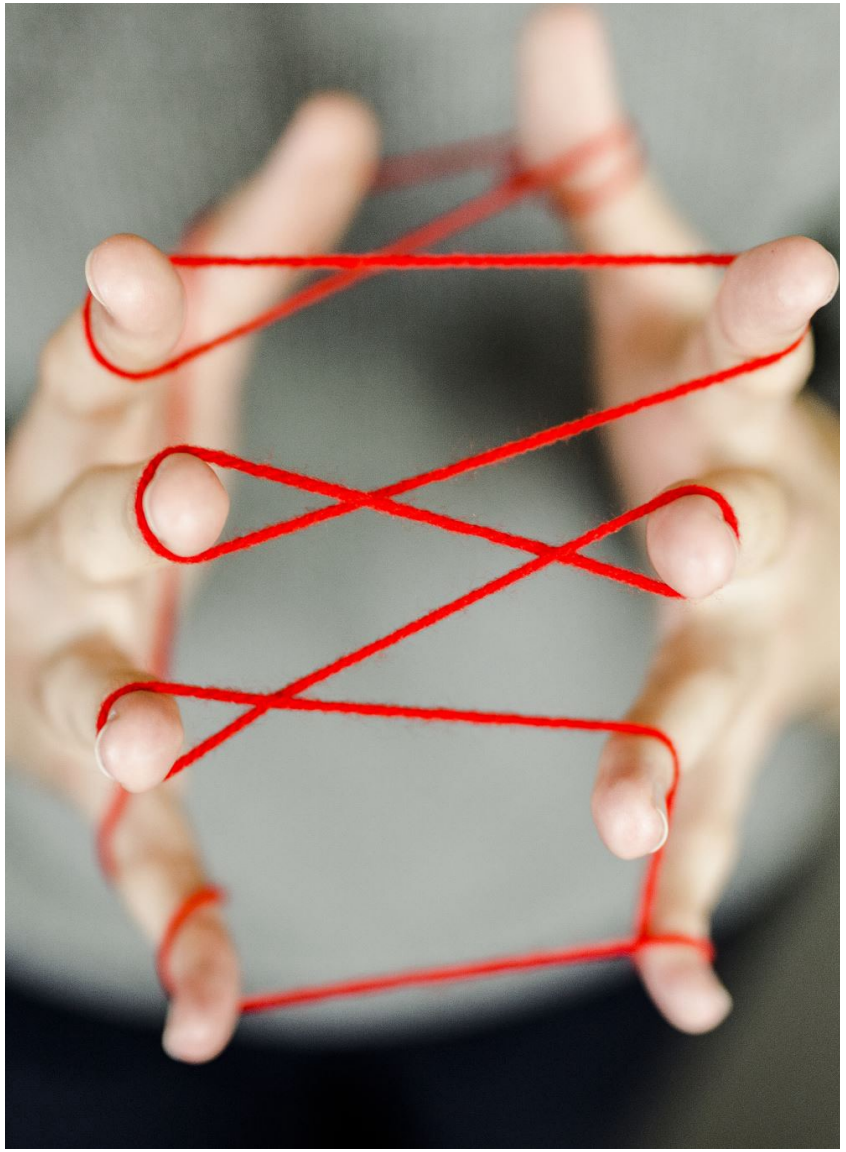
Challenges with Executors and Future/Callable

- **Awkward Composability:** Chaining dependent asynchronous operations leads to cumbersome nesting.
- **Blocking Future.get():** Retrieving results can potentially block threads, negating some non-blocking gains.
- **Limited Error Handling:** Propagating exceptions across asynchronous stages can be tricky to manage.
- **Executor Management:** Tuning thread pools for optimal performance requires careful consideration.

CompletableFuture to the Rescue

- Methods like **`thenApply`**, **`thenCombine`**, and handle enable fluent chaining of asynchronous operations without awkward nesting.
- The chaining pattern remains non-blocking; subsequent dependent tasks are asynchronously queued.
- Fine-grained control with exceptionally and similar methods provides more robust exception management across asynchronous tasks.
- While granular control is possible, many common operations have defaults, making it less of a required upfront burden.
- The chaining style starts to bridge non-blocking execution with syntax that feels closer to sequential code for many developers.

```
Void calculateCreditWithCompletableFuture(Long personId) {
    return CompletableFuture.supplyAsync(() -> getPerson(personId)).thenComposeAsync(person ->
                    CompletableFuture.allOf(
                            CompletableFuture.runAsync(this::importantWork),
                            CompletableFuture.supplyAsync(() -> getAssets(person))
                                    .thenAcceptBothAsync(CompletableFuture.supplyAsync(() -> getLiabilities(person)),
                                            this::calculateCredits)))
            .join();
}
```

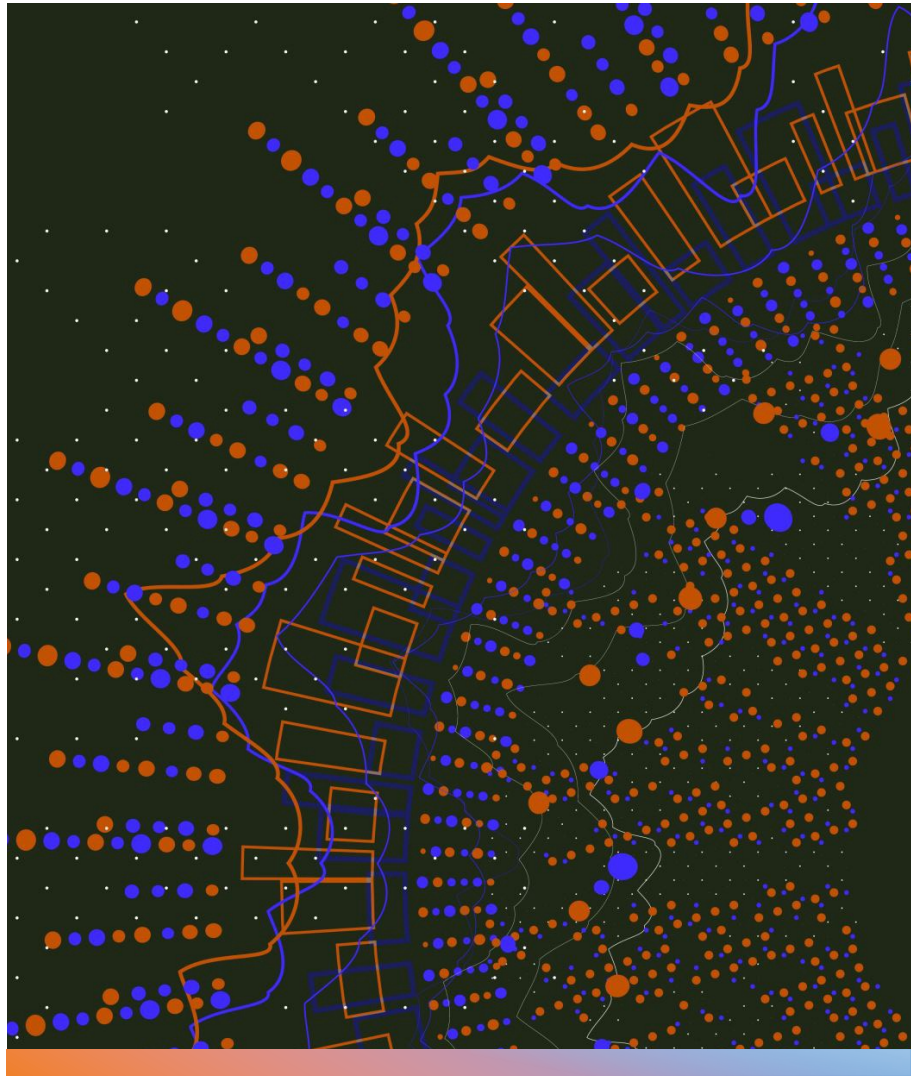


The Limitations of CompletableFuture

- **Complexity Can Creep Back In:** Intricate logic leads to dense chains, readability declines.
- **Control-Flow Limits:** Non-linear logic or complex task coordination becomes awkward.
- **Imperative Style Persists:** We still 'dictate' how tasks connect, not their data dependencies.
- **Debugging Challenges:** Async stacks remain tricky; traditional tools become less effective.

The Reactive Stack – Benefits

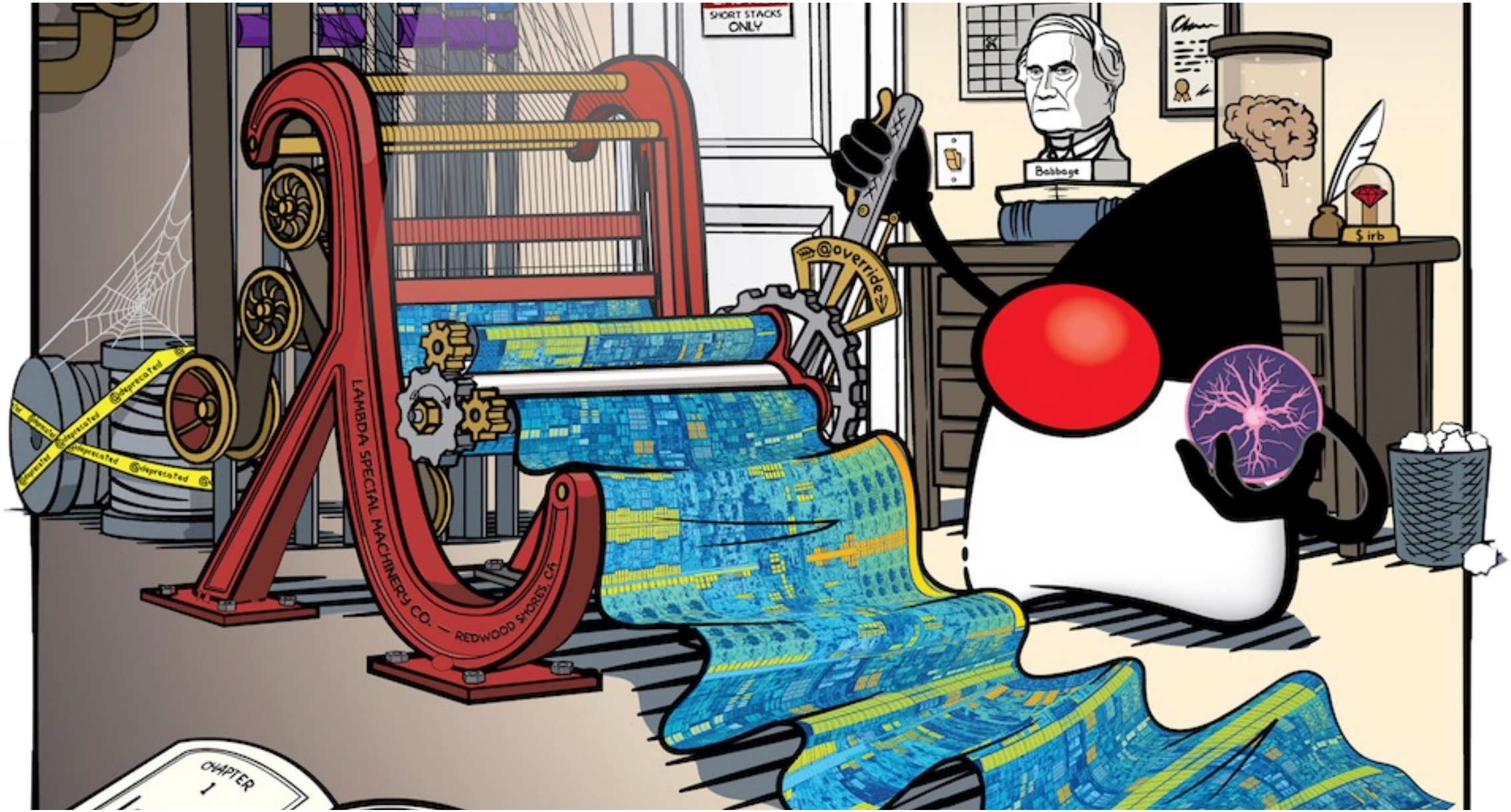
- **Flow Control (Backpressure):** Publishers and subscribers negotiate data rates, preventing components from being overwhelmed.
- **Declarative Composability:** Build complex stream transformations through chaining expressive operators (e.g. filter, map, combine).
- **Asynchronous by Design:** Optimized for concurrency and handling events that come at unpredictable intervals.
- **Resilience:** Gracefully handles errors and inconsistencies within data streams.
- **Popular Framework:** RxJava, Akka, Eclipse Vert.x , Spring WebFlux, Slick



The Reactive Stack – Considerations

- **Learning Curve:** The shift to data-flow thinking involves mastering new concepts and patterns.
- **Debugging:** Asynchronous data-flows can pose unique debugging challenges.
- **Code Readability:** Complex transformations may benefit from careful documentation for maintainability.
- **When to Choose:** Best suited for highly asynchronous, event-driven scenarios or problems that map naturally to a data-stream processing model.

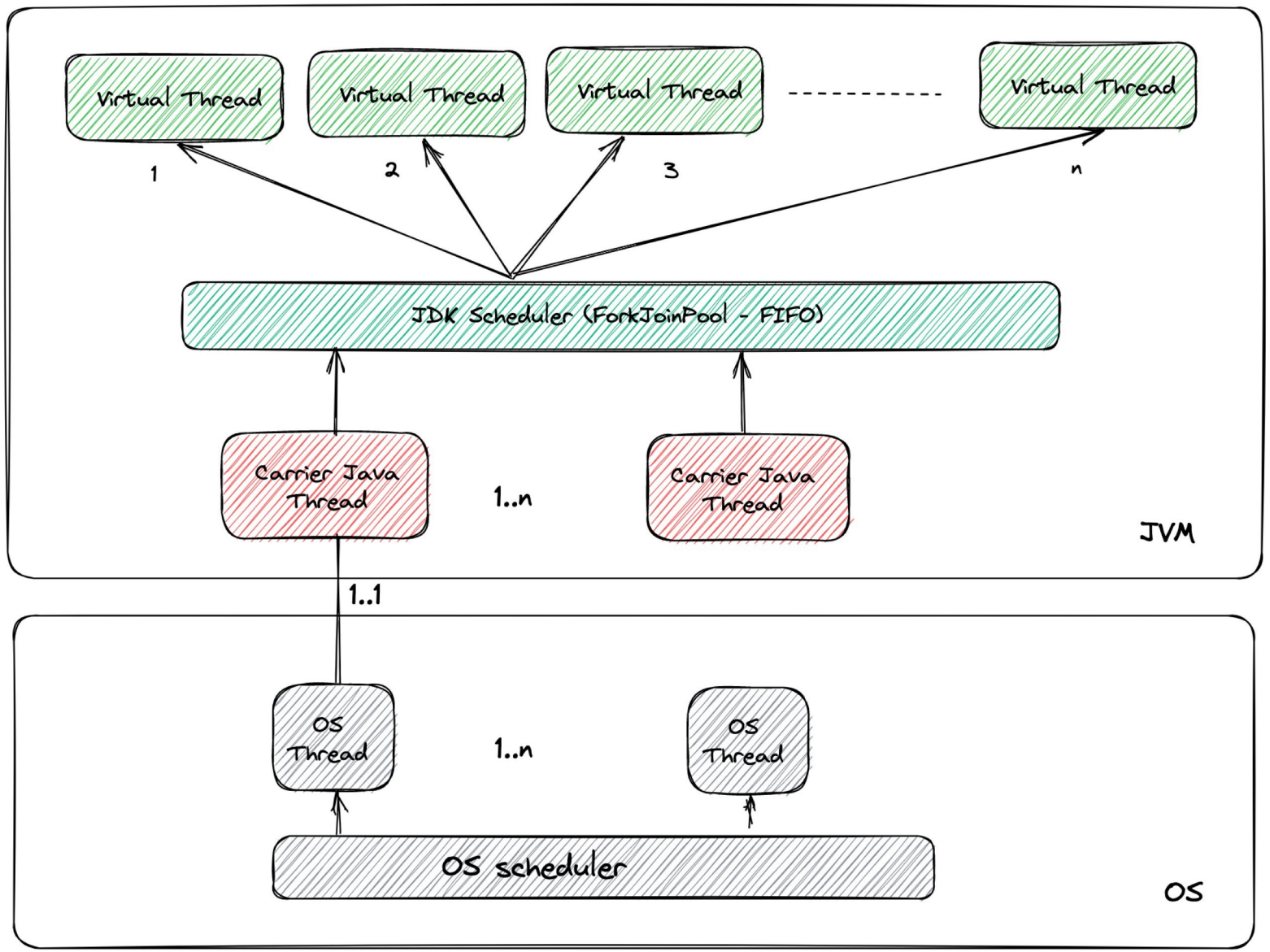
```
import io.reactivex.rxjava3.core.Single;

Single<Credit> calculateCredit (Long personId) { return
Single.fromCallable(this::importantWork)
        .flatMap(aVoid -> Single.fromCallable(() -> getPerson(personId))
            .flatMap(person -> Single.zip(
                Single.fromCallable(() -> getAssets(person)), Single.fromCallable(() ->
                getLiabilities(person)),
                this::calculateCredits
            )
        )
    );
}
```


SHORT STACKS ONLY
Babbage
$ irb
@override
@deprecated
LAMBDA SPECIAL MACHINERY CO. — REDWOOD SHORES, CA
CHAPTER 1

Demo

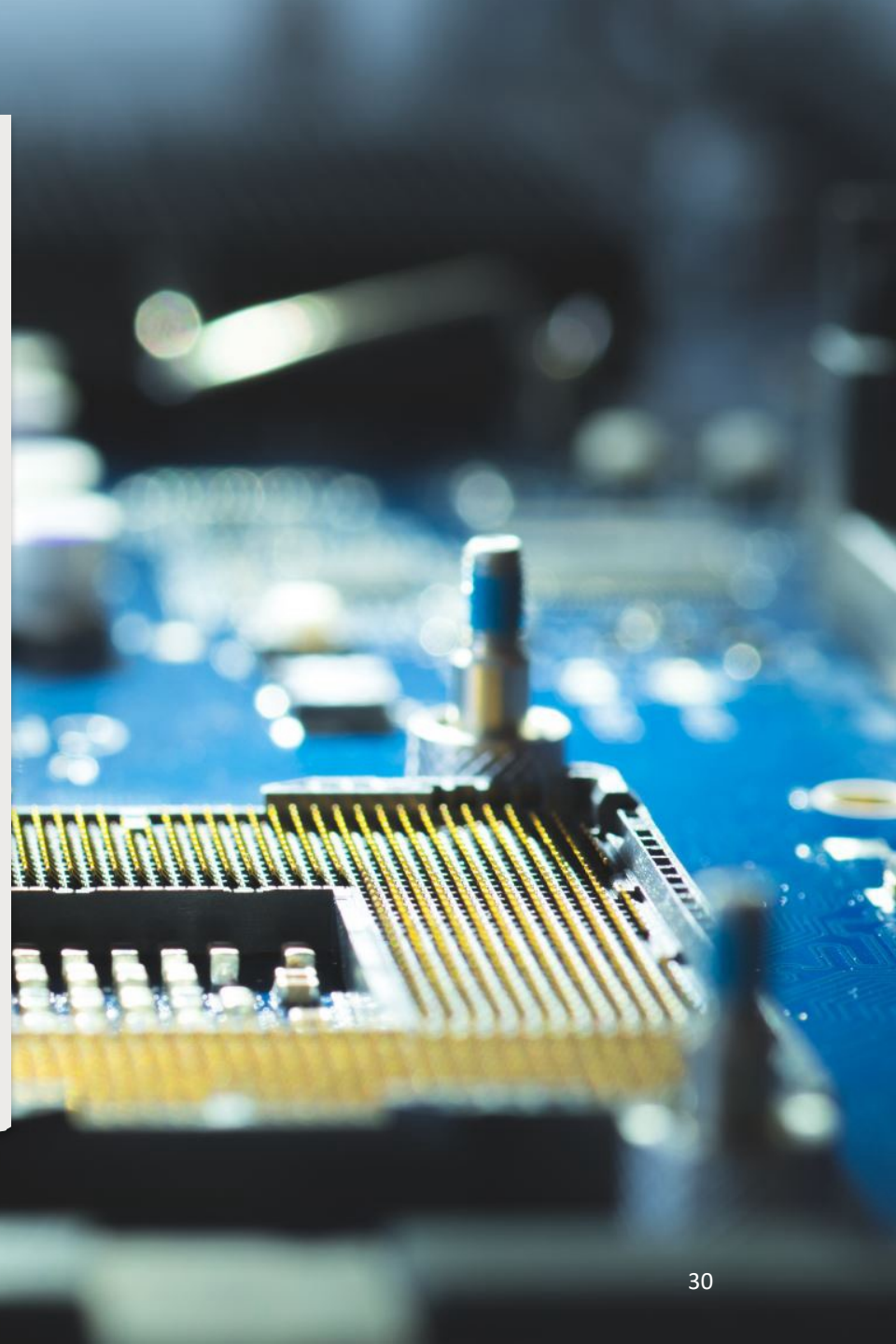
Thread Evolution - Virtual Threads vs. Platform Threads

Feature	OS-Managed Threads	Virtual Threads
Management	OS-managed, heavier resource footprint	JVM-managed, lightweight with dynamic memory
Quantity	Limited in number by memory constraints	Potential for millions of virtual threads
Context Switching Overhead	Context switching can be more overhead	Seamless mounting/unmounting reduces overhead


Virtual Thread
Virtual Thread
Virtual Thread
Virtual Thread
1
2
3
n
JDK Scheduler (ForkJoinPool - FIFO)
Carrier Java Thread
1..n
Carrier Java Thread
JVM
1..1
OS Thread
1..n
OS Thread
OS scheduler
OS

Benefits

- Virtual threads don’t consume CPU while they are waiting/sleeping
- Technically, we can have millions of virtual threads
- Threads are cheap
- The request-per-thread style of server-side programming becomes as manageable again as before.
- Improved throughput
- On Blocking I/O, virtual threads get automatically suspended
- No longer need to maintain ThreadPool



It's About Scalability!

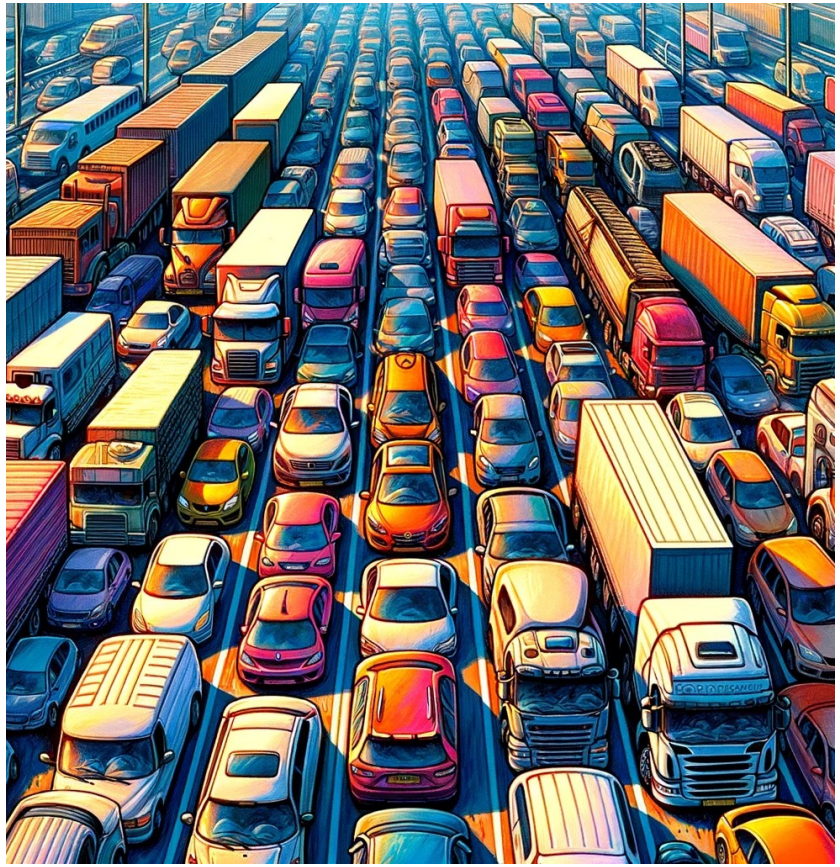
- The Bottleneck: Traditional Threads and I/O
  - Many server tasks involve waiting (network responses, database queries).
  - Platform threads become idle during waits, wasting a precious resource.
  - More platform threads = bigger memory overhead, even if they just sit inactive!
- Virtual Threads: Breaking the Constraint
  - Minimal footprint allows MANY virtual threads, most not consuming a platform thread at any given moment.
  - During I/O waits, virtual threads 'unmount' from their carriers, staying efficient with heap footprint instead.
  - Carriers dynamically pick up other ready-to-run virtual threads – maximizing work done.
- Hardware's True Power Realized
  - Before, we couldn't fully use modern CPUs due to the artificial limit on threads.
  - Virtual threads let applications squeeze the maximum work out of available cores.
  - This doesn't make INDIVIDUAL tasks finish faster, it keeps everything flowing at top potential

Use Semaphores to Limit Concurrency

```
Semaphore sem = new Semaphore(10);
...
Result foo() {
    sem.acquire();
    try {
        return callLimitedService();
    } finally {
        sem.release();
    }
}
```

The Challenge of Pinning

- Virtual threads map to carrier threads (platform threads) for real work to happen.
- Blocking operations (old-school sync, some native code) "pin" a virtual thread to its carrier.
- A pinned thread stalls the carrier, holding up other virtual threads waiting in line.
- Excessive pinning in high-concurrency systems negates some scalability gains.



Solution: Catch 'em in the Act.

```
-Djdk.tracePinnedThreads=full
```

Stacktrace

```
….
….
….
java.base/sun.nio.cs.StreamDecoder.close(StreamDecoder.java:249)
    java.base/java.io.InputStreamReader.close(InputStreamReader.java:203)
    okhttp3.ResponseBody$BomAwareReader.close(ResponseBody.kt:217)
    java.base/java.io.BufferedReader.implClose(BufferedReader.java:636)
    java.base/java.io.BufferedReader.close(BufferedReader.java:627) <== monitors:1
    com.fasterxml.jackson.core.json.ReaderBasedJsonParser._closeInput(ReaderBasedJsonParser.java:235)
    com.fasterxml.jackson.core.base.ParserBase.close(ParserBase.java:392)
    com.fasterxml.jackson.databind.ObjectMapper._readMapAndClose(ObjectMapper.java:4833)
```

How Long Was It Stuck?

- Find your application's PID with the **`jps`** command.
- Record some data:

```
jcmd <PID> JFR.start duration=200s filename=myrecording.jfr
```

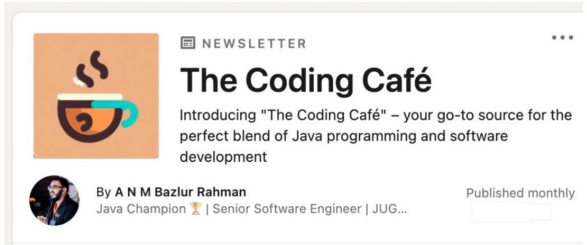
```
jfr print --events jdk.VirtualThreadPinned myrecording.jfr
```

```
jdk.VirtualThreadPinned {
  startTime = 13:21:28.226 (2024-02-12)
  duration = 108 ms
  eventThread = "tomcat-handler-8" (javaThreadId = 76, virtual)
  stackTrace = [
    java.lang.VirtualThread.parkOnCarrierThread(boolean, long) line: 677
    java.lang.VirtualThread.parkNanos(long) line: 636
    java.lang.System$2.parkVirtualThread(long) line: 2648
    jdk.internal.misc.VirtualThreads.park(long) line: 67
    java.util.concurrent.locks.LockSupport.parkNanos(long) line: 408
    ...
  ]
}

jdk.VirtualThreadPinned {
  startTime = 13:21:40.117 (2024-02-12)
  duration = 93.2 ms
  eventThread = "tomcat-handler-26" (javaThreadId = 94, virtual)
  stackTrace = [
    java.lang.VirtualThread.parkOnCarrierThread(boolean, long) line: 677
    java.lang.VirtualThread.parkNanos(long) line: 636
    java.lang.System$2.parkVirtualThread(long) line: 2648
    jdk.internal.misc.VirtualThreads.park(long) line: 67
    java.util.concurrent.locks.LockSupport.parkNanos(long) line: 408
    ...
  ]
}
```

About Me

- Staff Software Developer
- Java Champion
- Jakarta EE Ambassador
- JUG Leader
- Published Author
- InfoQ Editor of Java Queue
- Editor of Foojay.io

DNASTACK



https://twitter.com/bazlur_rahman
https://www.linkedin.com/in/bazlur/
https://foojay.io/today/author/bazlur-rahman/
https://bazlur.ca