Virtual Thread: Ushering in a New Era of Concurrency

Why and how virtual threads were implemented in Java

A N M Bazlur Rahman (@bazlur_rahman)

Part 1

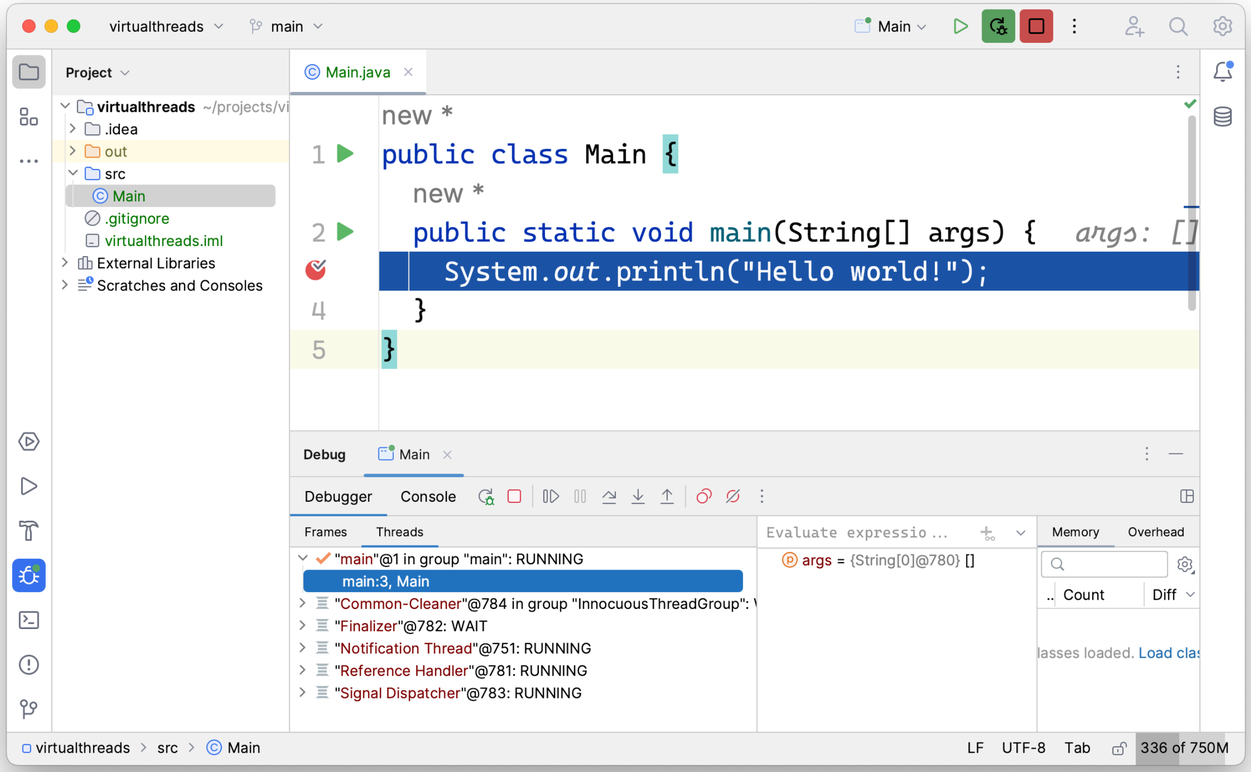
Background and the problem

Java Is Made of Threads

- From day zero, Java introduced Thread
- Threads are units of execution in Java
- More threads == More execution environment
- That also means we can do more things simultaneously

Threads Are All layer in Java Platform

- Exceptions
- ThreadLocal
- Debugger
- Profiler



How Do We Create Threads?

```
public class Main {
  public static void main(String[] args) {

    var thread = new Thread(() → {
      System.out.println("Hello world!");
    });

    thread.start();
  }
}
```

Let's Sleep For A While & Take ThreadDump

```
public static void main(String[] args) throws InterruptedException
{

  var thread = new Thread(() → {
    System.out.println("Hello world!");
    sleep();
  });

  thread.start();
  thread.join();
}
```

Java Threads == OS threads

```
"Thread-0" #29 [40451] prio=5 os_prio=31 cpu=0.57ms elapsed=27.69s
tid=0x00007fae8d009400 nid=40451 waiting on condition
[0x0000700003bc6000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
 at java.lang.Thread.sleep0(java.base@21-ea/Native Method)
 at java.lang.Thread.sleep(java.base@21-ea/Thread.java:509)
 at Main.sleep(Main.java:13)
 at Main.lambda$main$0(Main.java:5)
 at Main$$Lambda/0x0000000801003200.run(Unknown Source)
 at java.lang.Thread.runWith(java.base@21-ea/Thread.java:1605)
 at java.lang.Thread.run(java.base@21-ea/Thread.java:1592)
```

Threads Are Expensive

Threads Are Expensive Commodity

- Thread.start() consider inefficient
- 2 MB of memory (outside of heaps)
- A thin wrapper of OS threads
- Context Switching takes time: ~100 μs
- So 1 million threads would take two terabytes of memory?

Threads Are Limited

- In modern Java Web applications, **Throughput** is achieved by concurrent connection.
- Modern OS can handle millions of concurrent connections.
- More connections == Higher throughput
- But Threads are limited in number

Let's Count Threads

```
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.LockSupport;

public class ThreadCounter {
    public static void main(String[] args) {
        var counter = new AtomicInteger();

        while (true) {
            var thread = new Thread(() -> {
                counter.incrementAndGet();

                if (counter.get() % 100 == 0) {
                    System.out.printf("Number of threads created so far: %d%n", counter.get());
                }
                LockSupport.park();
            });
            thread.start();
        }
    }
}
```

Native Thread Count

Number of threads created so far: 3900

Number of threads created so far: 4000

[0.984s][warning][os,thread] Failed to start thread "Unknown thread" - pthread_create failed (EAGAIN) for attributes: stacksize: 1024k, guardsize: 4k, detached.

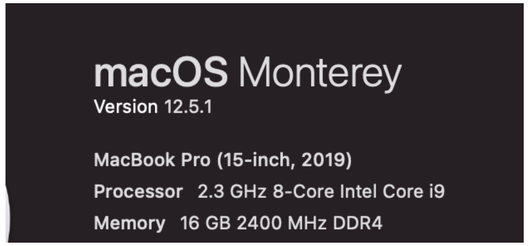
[0.984s][warning][os,thread] Failed to start the native thread for java.lang.Thread "Thread-4074"

Exception in thread "main" java.lang.OutOfMemoryError: unable to create native thread: possibly out of memory or process/resource limits reached

at java.base/java.lang.Thread.starto(Native Method)

at java.base/java.lang.Thread.start(Thread.java:1535)

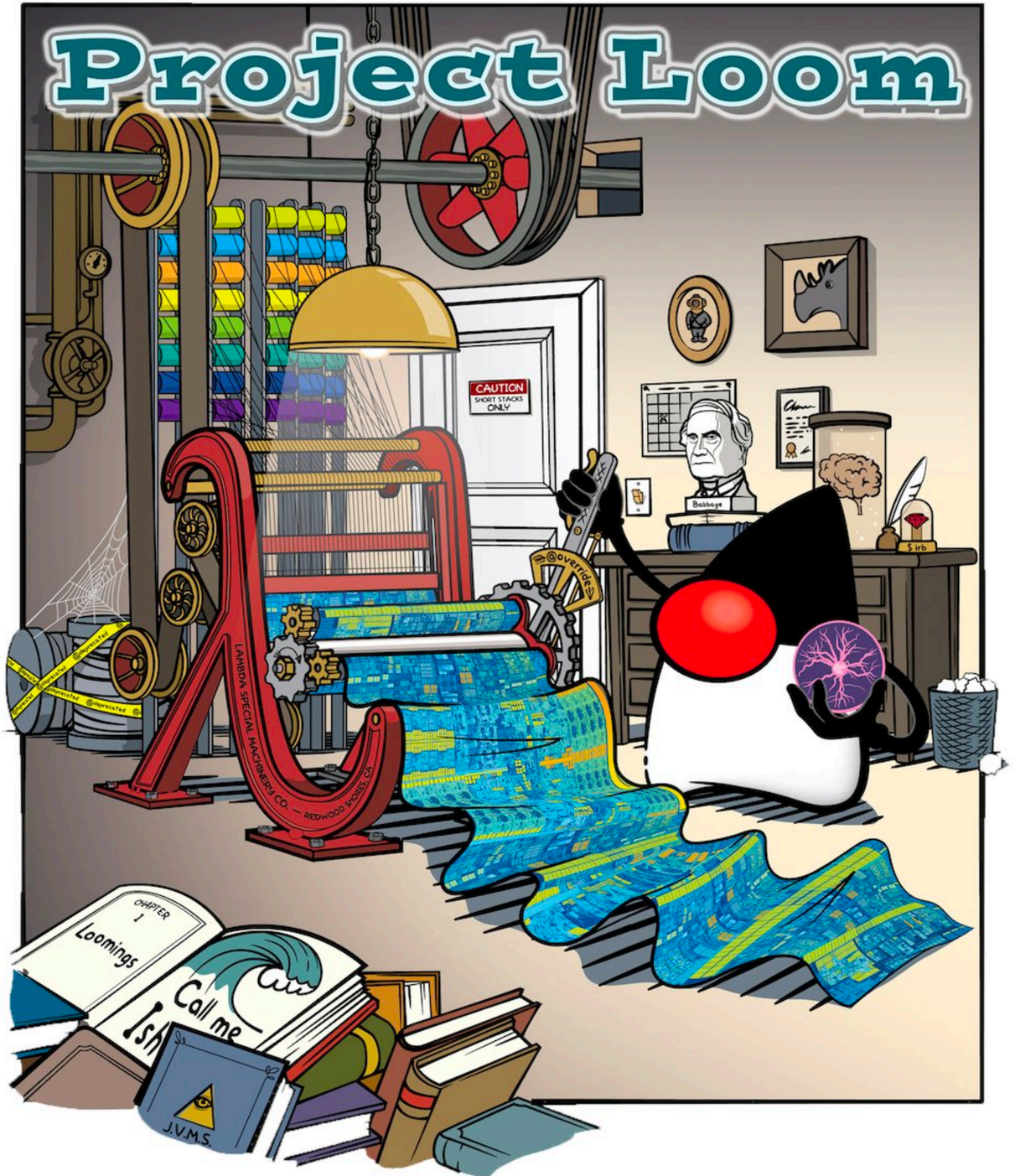
at ca.bazlur.ThreadCounter.main(ThreadCounter.java:19)



Demo

Part 2

Solution and Implementation



Introducing Virtual Threads

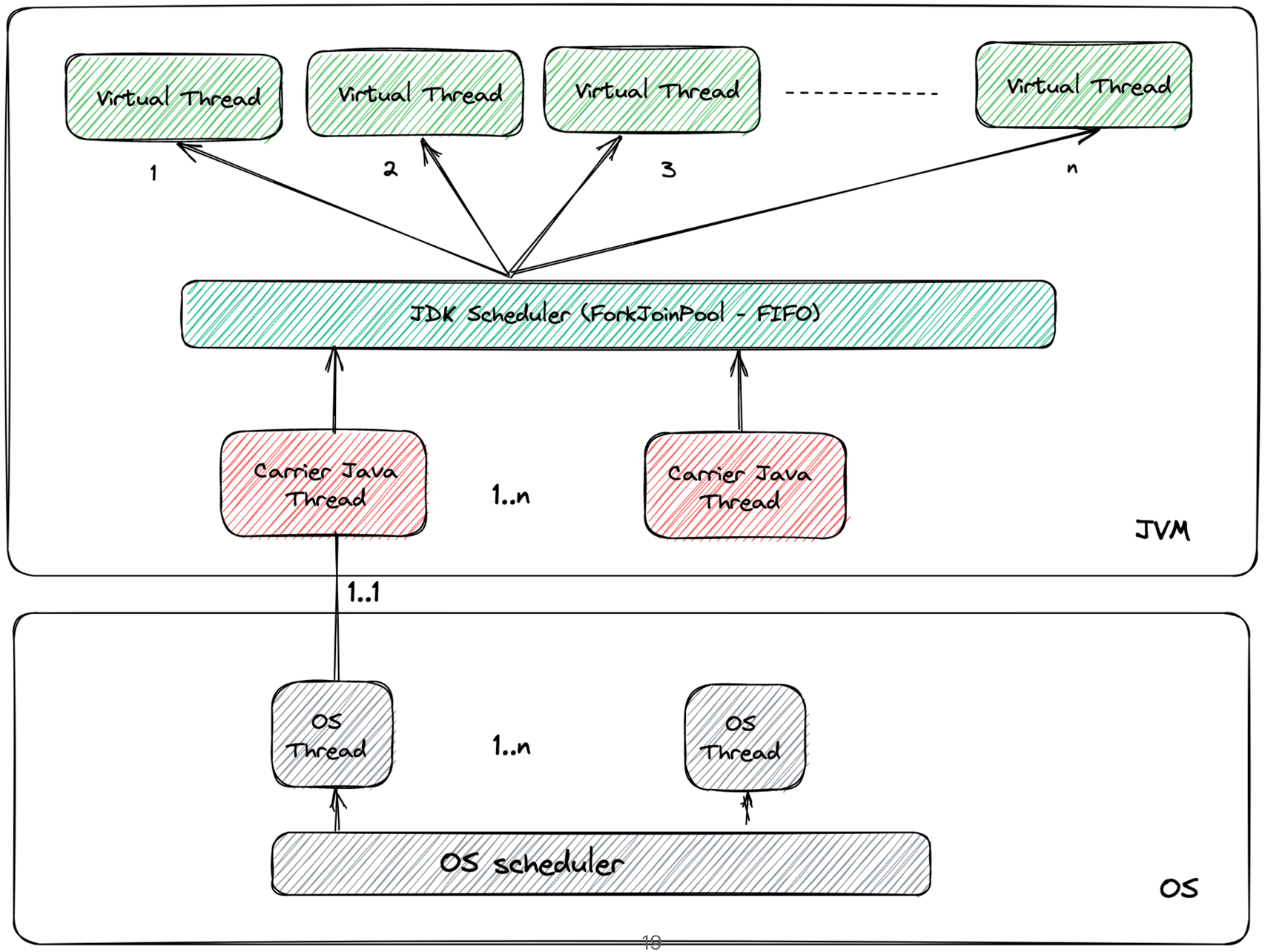
```
var thread = Thread.startVirtualThread(() → {

    System.out.println("Hello world!");
});


var thread2 = Thread.ofVirtual().start(() → {

    System.out.println("Hello world!");
});
```

Demo

How?

Implementation

- Fork/Join Pool
- Continuation


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

Continuation

Continuation

- Basic definition: a sequence of code that may suspend (itself) and can be resumed later
- May suspend or yield execution at some point
- When it suspends, control is passed outside of the continuation, and when it is resumed, control returns to the last yield point.

```java
public class ContinuationExample {
    public static void main(String[] args) throws InterruptedException {
        var scope = new ContinuationScope("JConf Demo");

        var continuation = new Continuation(scope, () -> {
            System.out.println("C1");

            Continuation.yield(scope);

            System.out.println("C2");
        });


        System.out.println("Start here");

        continuation.run();

        System.out.println("Coming back");

        continuation.run();

        System.out.println("Finish");
    }
}
```

Who else Uses The Similar Concept?

- Coroutine in Kotlin
- Goroutine Go
- Async/Await in JavaScript
- Python Generators
- Ruby Fibers
- Finally in Java As well

Benefits

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

Let's Download 10K Images

```
Flux.fromIterable(imageUrls)
        .flatMap(url → downloadImage(httpClient, url))
        .doOnNext(tuple → saveImage(tuple.getT1(), tuple.getT2()))
        .doOnComplete(() → System.out.println("Finished downloading and saving images."))
        .blockLast();
```

```
for (String imageUrl : imageUrls) {
    executor.submit(() → {
        byte[] imageBytes = downloadImage(httpClient, imageUrl);
        saveImage(imageUrl, imageBytes);
        System.out.println("Downloaded and saved: " + imageUrl);
    });
}
```

Framework Implementing Virtual Threads

- Spring Boot
- Helidon Nima
- Quarkus
- And many more

Think About a Bank with Fixed Number of Tellers

- In Java terms, a teller is like a Platform Thread.
- Generally, it would take time to process each customer, say an average of 5 minutes.
- Sometimes, a customer would block the process, such as the teller needing to make a phone call to get some information.
- No work is performed while the teller is blocked waiting, and consequently, the entire line is blocked.

Before Loom

In Java terms, a teller is still like a Platform Thread but can park a customer.

Generally, it still takes time to process each customer, say an average of 5 minutes.

Sometimes, a customer would block the process, such as the teller needing some information before proceeding…

- The teller sends a text message or email to get the necessary information
- The teller asks the customer to be seated, and as soon the information is available, they will be the next customer processed by the first available teller
- The teller starts processing the next customer in line
- This is analogous to a parked Virtual Thread, where the teller is like a Platform Thread, and the customer is like a Virtual Thread

Concurrency is increased by better policies and procedures in dealing with blocking operations

API Changes

API Differences Between Virtual and Platform Threads

1. Public Thread constructors cannot create virtual threads
2. Virtual threads are always daemon threads
3. Virtual threads have a fixed priority of Thread.NORM_PRIORITY
4. Virtual threads are not active members of thread groups
5. Virtual threads have no permissions when running with a SecurityManager set

Thread Local with Virtual Thread

- Virtual Thread supports:
  - ThreadLocal
  - InheritableThreadLocal
- A large number of virtual threads - use thread-local variables cautiously.
- JDK assistance for migration
    - System property: jdk.traceVirtualThreadLocals
    - Triggers stack trace when a virtual set a value of any ThreadLocal variable.

Java.util.concurrent Updates for Virtual Threads

1. Updated LockSupport API:
   - Gracefully parks and unparks virtual threads
   - Enables seamless functionality with Locks, Semaphores, and blocking queues
2. New ExecutorService methods:
   - Executors.newThreadPerTaskExecutor(ThreadFactory)
   - Executors.newVirtualThreadPerTaskExecutor()
   - Creates a new thread for each task
3. Migration and interoperability:
   - Facilitates integration with existing code using thread pools and ExecutorService

Networking APIs Update for Virtual Threads

1. Enhanced support in java.net and java.nio.channels packages:
   - Virtual threads improve efficiency in concurrent applications
2. Blocking operations on virtual threads:
   - Frees up the underlying platform thread
3. Interruptible I/O methods:
   - Socket, ServerSocket, and DatagramSocket classes
4. Benefits for Java developers:
   - Consistent behavior and improved performance in concurrent applications

java.io Package Updates for Virtual Threads

1. Goal: Avoid pinning in virtual threads
   - Pinning limits concurrency and flexibility due to blocking operations
2. Updated classes to use explicit lock instead of a monitor:
   - BufferedInputStream
   - BufferedOutputStream
   - BufferedReader
   - BufferedWriter
   - PrintStream
   - PrintWriter
3. Stream decoders and encoders:
   - InputStreamReader and OutputStreamWriter now use the same lock as their enclosing instances

Debugging Architecture Updates for Virtual Threads

1. Updated components to support virtual threads:
   - JVM Tool Interface (JVM TI)
   - Java Debug Wire Protocol (JDWP)
   - Java Debug Interface (JDI)
2. New capabilities and methods:
   - Handling thread start and end events
   - Bulk suspension and resumption of virtual threads
3. Enhanced debugging experience for Java developers working with virtual threads

JDK Flight Recorder (JFR) Support for Virtual Threads

1. JFR now supports virtual threads
2. New events to monitor virtual thread behavior:
   - jdk.VirtualThreadStart
   - jdk.VirtualThreadEnd
   - jdk.VirtualThreadPinned
   - jdk.VirtualThreadSubmitFailed
3. Provides valuable insights into virtual thread performance and usage in applications

JMX Support for Virtual Threads

1. JMX and platform threads:
   - ThreadMXBean interface continues to support only platform threads
2. New method for virtual threads support:
   - Added to the HotSpotDiagnosticsMXBean interface
3. New-style thread dump:
   - Supports virtual threads while maintaining compatibility with existing JMX infrastructure

Compatibility Risks with Virtual Threads

1. Performance improvements come with risks:
   - Changes to existing APIs and their implementations
2. Examples of compatibility risks:
   - Revisions to the internal locking protocol in the java.io package
   - Source and binary incompatible changes affecting code that extends the Thread class
3. Developers need to carefully consider these risks when adopting virtual threads in their applications

Resources

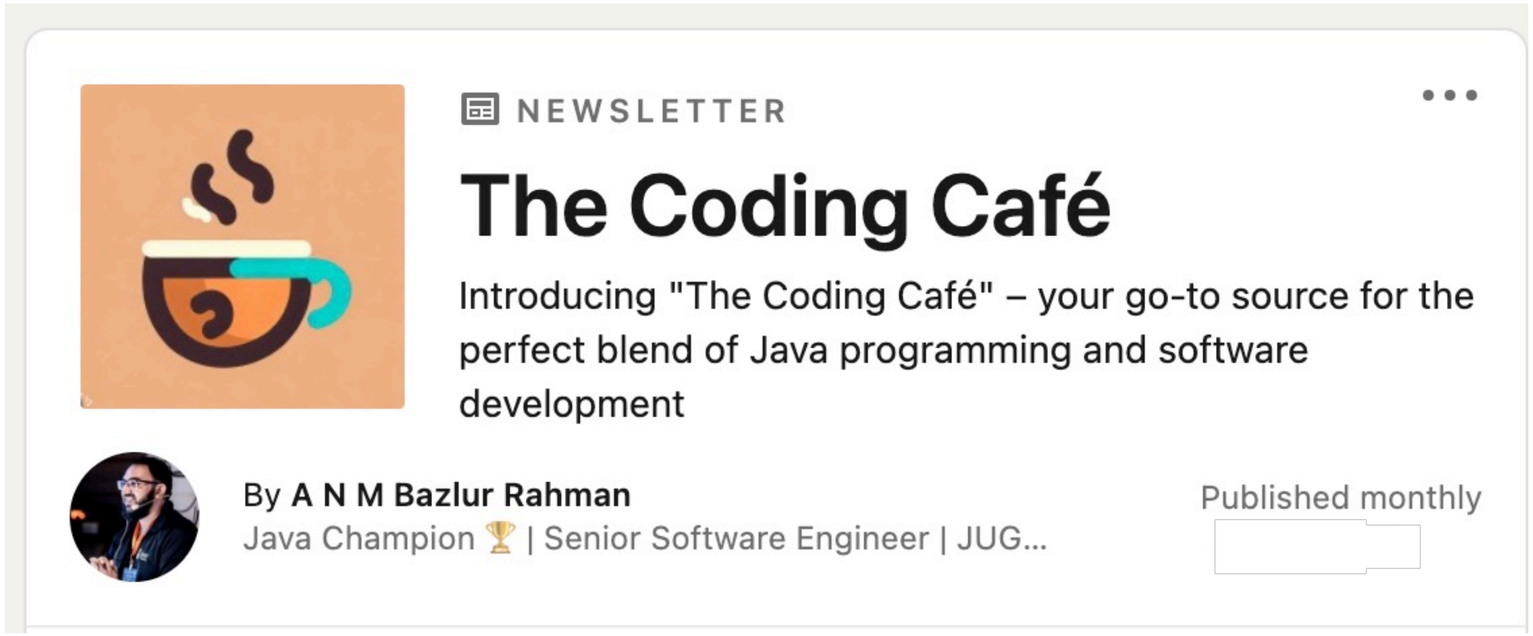
- JEP 425: Virtual Threads (Preview)
- JEP 436: Virtual Threads (Second Preview)
- JEP 444: Virtual Threads
- Virtual Threads: New Foundations for High-Scale Java Applications

“Virtual threads are cheap enough to have a single thread per task, and eliminate many of the common issues with writing concurrent code in Java.”

Ron Pressler

“There aren't any new APIs you have to learn, but you do need to unlearn many habits such as using thread pools to deal with resource contention.”

Ron Pressler


NEWSLETTER
The Coding Café
Introducing "The Coding Café" – your go-to source for the perfect blend of Java programming and software development
By A N M Bazlur Rahman
Java Champion 🏆 | Senior Software Engineer | JUG...
Published monthly

Thank You