Prepare for what *Loom*s ahead

Dr Heinz M. Kabutz

Last updated 2023-05-04

Javaspecialists.eu
java training

Heinz Kabutz

- **The Java Specialists' Newsletter**
  - **308 editions, published since 2000**
  - **www.javaspecialists.eu**

- **Please say "hi" after talk and during conference :-)**
  - **And we are doing a workshop tomorrow on Java Reflection**

JavaSpecialists.eu

When is Loom Coming?

- **Virtual threads to be fully released in Java 21**

- **Structured concurrency in an incubator module**
  - **Will take a bit longer to finish**
  - **Add with --add-modules jdk.incubator.concurrent**

Why do we need Virtual Threads?

- **Asynchronous code can be hard to debug**

- **1-to-1 Java thread to platform thread does not scale**

- **Welcome to Project Loom**

  – **Millions of virtual threads in a single JVM**

  – **Supported by networking, java.util.concurrent, etc.**

    • **(Almost) Anywhere you would block a thread**

JavaSpecialists.eu

Parallel Computing

- **Solving a problem on many CPUs in parallel**
  - **Large problem is broken into smaller ones, solved in parallel**
  - **Focus is on solving problems faster**
  - **Communication overhead reduces speedup possibilities**

- **Typically used on large number of cores**

- **Java: ForkJoin or parallel streams**

- **Examples: Weather prediction, financial trend analysis**

JavaSpecialists.eu

Concurrent Computing

- **Interacting tasks may execute concurrently**
  - **Independent tasks simplify architecture**

  - **Do something useful during wait time (IO, Locks, etc.)**

  - **Focus on task interaction (memory integrity, progress)**

  - **Does not always scale well with native threads**

- **Can be used on any number of cores**

- **Java: Structured concurrency, virtual threads**

- **Examples: Blocking IO, background tasks**

JavaSpecialists.eu

Best Deal Search

- **Our webpage server requires 4 steps**

  1. **Scan request for search terms**

  2. **Search partner websites**

  3. **Create advertising links**

  4. **Collate results from partner websites**

- **We can reorder some steps without affecting result**

JavaSpecialists.eu

Sequential Best Deal Search

4.3 seconds

- **Sequential processing is the simplest**

```java
public void renderPage(HttpRequest request) {
  List<SearchTerm> terms = scanForSearchTerms(request); // 1
  List<SearchResult> results = terms.stream()
      .map(SearchTerm::searchOnPartnerSite) // 2
      .toList();
  createAdvertisingLinks(request); // 3
  results.forEach(this::collateResult); // 4
}
```

JavaSpecialists.eu

Page Renderer with Future

- **Search partner sites in the background with Callable**
  - **We might get better performance this way**
  - **If we are lucky, search results are ready when we need them**

Searching in Background Thread

```java
public class FutureRenderer extends BasicRenderer {
  private final ExecutorService executor;
  public FutureRenderer(ExecutorService executor) {
    this.executor = executor;
  }
  public void renderPage(HttpRequest request)
      throws ExecutionException, InterruptedException {
    List<SearchTerm> terms = scanForSearchTerms(request); // 1
    Callable<List<SearchResult>> task = () ->
        terms.stream()
            .map(SearchTerm::searchOnPartnerSite) // 2
            .toList();
    Future<List<SearchResult>> results = executor.submit(task);
    createAdvertisingLinks(request); // 3
    results.get().forEach(this::collateResult); // 4
  }
}
```
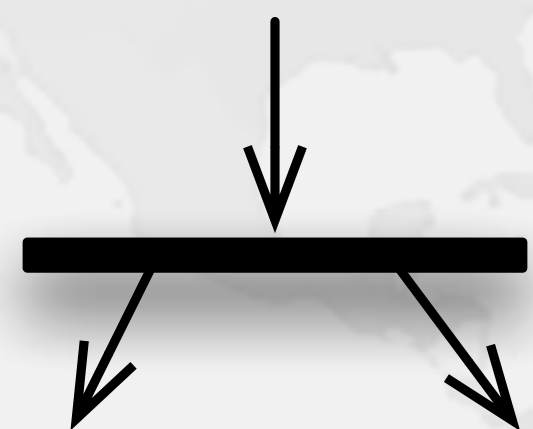
4.1 seconds

JavaSpecialists.eu

CompletableFuture

- **Convert each step into a CompletableFuture**
  - **Then combine these using *allOf()***
  - **Code is slightly faster, but a whole lot more complicated**
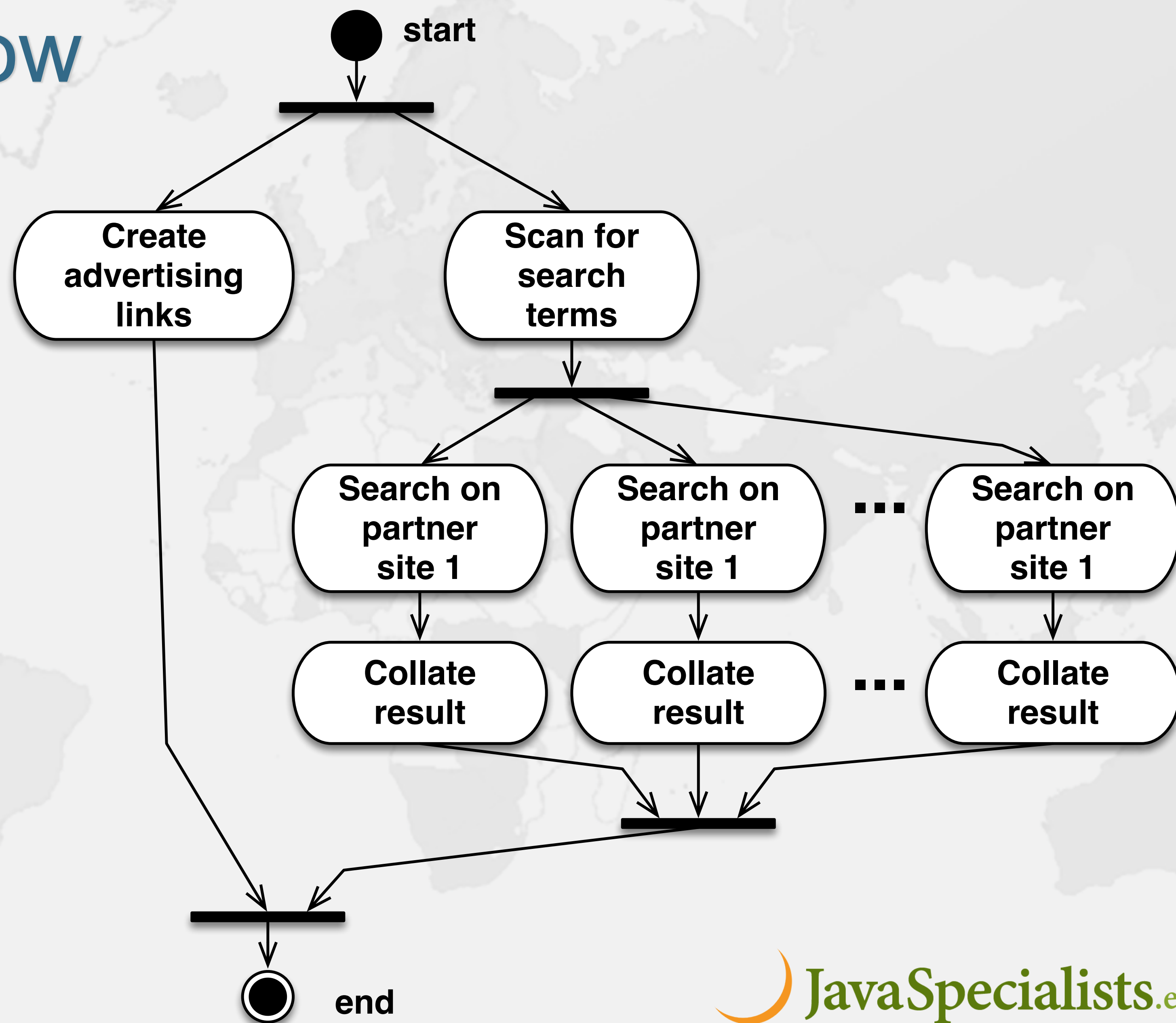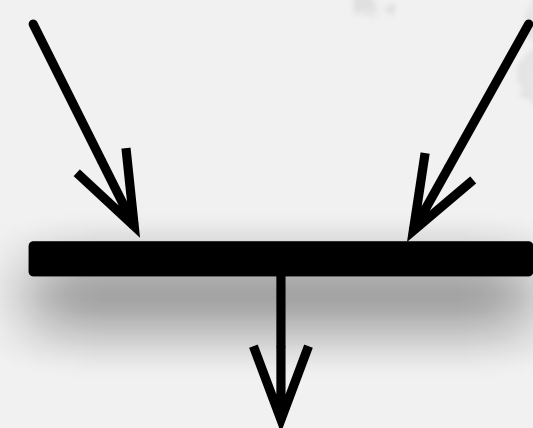   - **Need separate pools for CPU and IO bound tasks**

JavaSpecialists.eu

Modeling Control Flow

- **Renderer example as UML Activity Diagram**

  - **Fork**

  - **Join**

start

Create advertising links

Scan for search terms

Search on partner site 1

Search on partner site 1

... Search on partner site 1

Collate result

Collate result

... Collate result

end

JavaSpecialists.eu

renderPage() with CompletableFuture

```java
public class RendererCF extends BasicRenderer {
  private final ExecutorService cpuPool, ioPool;
  public RendererCF(ExecutorService cpuPool, ExecutorService ioPool) {
    this.cpuPool = cpuPool;
    this.ioPool = ioPool;
  }
  public void renderPage(HttpRequest request) {
    renderPageCF(request).join();
  }


  public CompletableFuture<Void> renderPageCF(HttpRequest request) {
    return CompletableFuture.allOf(createAdvertisingLinksCF(request),
        scanSearchTermsCF(request).thenCompose(this::searchAndCollateResults));
  }


  private CompletableFuture<Void> createAdvertisingLinksCF(HttpRequest request) {
    return CompletableFuture.runAsync(
        () -> createAdvertisingLinks(request), cpuPool);
  }
```

JavaSpecialists.eu

searchAndCollateResults()

```java
private CompletableFuture<List<SearchTerm>> scanSearchTermsCF(
    HttpRequest request) {
  return CompletableFuture.supplyAsync(
      () -> scanForSearchTerms(request), cpuPool);
}


private CompletableFuture<Void> searchAndCollateResults(
    List<SearchTerm> list) {
  return CompletableFuture.allOf(
      list.stream()
          .map(this::searchAndCollate)
          .toArray(CompletableFuture<?>[]::new)
  );
}


private CompletableFuture<Void> searchAndCollate(SearchTerm term) {
  return searchOnPartnerSiteCF(term).thenCompose(this::collateResultCF);
}
```

JavaSpecialists.eu

Tasks Wrapped in CompletableFutures

```java
private CompletableFuture<SearchResult> searchOnPartnerSiteCF(
    SearchTerm term) {
  return CompletableFuture.supplyAsync(
      term::searchOnPartnerSite, ioPool);
}

private CompletableFuture<Void> collateResultCF(SearchResult data) {
  return CompletableFuture.runAsync(
      () -> collateResult(data), cpuPool);
}
}
```

0.9 seconds

JavaSpecialists.eu

What about plain Thread?

- **Could we simply create one thread per task?**
  - **Code would be simpler than with the CompletableFuture**

renderPage() with platform threads

```java
public void renderPage(HttpRequest request)
    throws InterruptedException {
  Thread createAdvertisingThread =
      new Thread(() -> createAdvertisingLinks(request)); // 3
  createAdvertisingThread.start();
  Collection<Thread> searchAndCollateThreads =
      scanForSearchTerms(request).stream() // 1
          .map(term -> {
            Thread thread = new Thread(// 2 & 4
              () -> collateResult(term.searchOnPartnerSite()));
            thread.start();
            return thread;
          })
          .toList();
  createAdvertisingThread.join();
  for (Thread searchAndCollateThread : searchAndCollateThreads)
    searchAndCollateThread.join();
}
```

0.5 seconds

Started 11 threads

JavaSpecialists.eu

Not scalable

- **Even one thread per client connection is too many**
  - **In our example we could be launching dozens of threads**

Virtual Threads

- **Lightweight, less than 1 kilobyte**
  - **Fast to create**
  - **Over 23 million virtual threads in 16 GB of memory**

- **Executed by carrier threads**
  - **Scheduler is a ForkJoinPool**
    - **Carriers are always daemon threads**

Configuring Carrier Threads

- **# threads is Runtime.getRuntime().availableProcessors()**
  - **Configure size with `jdk.virtualThreadScheduler.parallelism`**
  - **Can temporarily increase due to ManagedBlocker**
    - **Configure with `jdk.virtualThreadScheduler.maxPoolSize`**

- **Moved off carrier threads when blocking on IO**
  - **Also with waiting on synchronizers from java.util.concurrent**

Before we continue ...

- **A small gift for you (no, it's not tsikoudia - expires at 12:45)**

```
tinyurl.com/devoxx-el
```


JavaSpecialists.eu

Let's go back to SingleThreadedRenderer

- **If threads are unlimited and free, why not create a new virtual thread for every task?**

- **This is how our single-threaded renderer looked**

```java
public void renderPage(HttpRequest request) {
  List<SearchTerm> terms = scanForSearchTerms(request); // 1
  List<SearchResult> results = terms.stream()
      .map(SearchTerm::searchOnPartnerSite) // 2
      .toList();
  createAdvertisingLinks(request); // 3
  results.forEach(this::collateResult); // 4
}
```

tinyurl.com/devoxx-el

Virtual threads galore

```java
public void renderPage(HttpRequest request)
    throws InterruptedException {
  Thread createAdvertisingThread =
      Thread.startVirtualThread(
        () -> createAdvertisingLinks(request)); // 3
  Collection<Thread> searchAndCollateThreads =
      scanForSearchTerms(request).stream() // 1
        .map(term -> Thread.startVirtualThread( // 2 & 4
          () -> collateResult(term.searchOnPartnerSite())))
        .toList();
  createAdvertisingThread.join();
  for (Thread searchThread : searchAndCollateThreads)
    searchThread.join();
}
```

0.5 seconds

How to create virtual threads

- **Individual threads**

  – **Thread.startVirtualThread(Runnable)**

  – **Thread.ofVirtual().start(Runnable)**

- **ExecutorService**

  – **Executors.newVirtualThreadPerTaskExecutor()**

  – **ExecutorService is now AutoCloseable**

    • **close() calls shutdown() and awaitTermination()**

`tinyurl.com/devoxx-el`

JavaSpecialists.eu

Using ExecutorService - Unstructured

```java
public void renderPage(HttpRequest request) {
  try (ExecutorService mainPool =
          Executors.newVirtualThreadPerTaskExecutor()) {
    mainPool.submit(() -> createAdvertisingLinks(request)); // 3
    mainPool.submit(() -> {
      List<SearchTerm> terms = scanForSearchTerms(request); // 1
      try (ExecutorService searchAndCollatePool =
                Executors.newVirtualThreadPerTaskExecutor()) {
        terms.forEach(term -> searchAndCollatePool.submit( // 2 & 4
          () -> collateResult(term.searchOnPartnerSite())));
      }
    });
  }
}
```

0.5 seconds

JavaSpecialists.eu

Structured Concurrency (Incubator)

- **Better approach for describing concurrent flows**

  – **https://openjdk.org/jeps/428**

- **Idioms are still being developed, e.g.**

```java
public void renderPage(HttpRequest request)
    throws InterruptedException, ExecutionException {
  try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    scope.fork(() -> createAdvertisingLinks(request)); // 3
    List<SearchTerm> terms = scanForSearchTerms(request); // 1
    terms.forEach(term -> scope.fork(
        () -> collateResult(term.searchOnPartnerSite()))); // 2 & 4
    scope.join();            // Join all forks
    scope.throwIfFailed();   // ... and propagate errors
  }
}
```

0.5 seconds

JavaSpecialists.eu

InterruptedException

- **Long-running methods should check if "interrupted"**
  - **And avoid "swallowing" InterruptedException**

- **StructuredDemo**

ManagedBlocker

- **ForkJoinPool makes more threads when blocked**
  - **ForkJoinPool is configured with desired parallelism**

- **Uses in the JDK**
  - **Java 7: Phaser, Java 8: CompletableFuture**
  - **Java 9: Process, SubmissionPublisher**
  - **Java 14: AbstractQueuedSynchronizer**
    - **ReentrantLock, CountDownLatch, Semaphore, etc.**
  - **Java 17: LinkedTransferQueue, SynchronousQueue**
  - **Loom: SelectorImpl, Object.wait(), old I/O**

JavaSpecialists.eu

ManagedBlocker

- **Might need to update our code base**
  - **Ideally we should never block a thread with native methods**
  - **If we cannot avoid it, wrap the code in a ManagedBlocker**

Java IO Implementation Rewritten

- **JEP353 Reimplement Legacy Socket API**
  - **PlainSocketImpl replaced by NioSocketImpl**

  - **https://openjdk.java.net/jeps/353**

- **JEP373 Reimplement Legacy DatagramSocket API**
  - **https://openjdk.java.net/jeps/373**

Things to know about Virtual Threads

- **Future now has: resultNow(), exceptionNow(), state()**
  - **See demo TimeoutInPool**

- **Java networking IO is interruptible on virtual threads**
  - **In the past, we had to close the socket, stop() didn't work**

- **Reflection rewritten to avoid native code (JEP 416)**

- **Thread dump does not pause application**
  - **$ jcmd <pid> Thread.dump_to_file -format=json <file>**

- **Stack depth same as the JVM native threads**

JavaSpecialists.eu

Things to know about Virtual Threads

- **Unlike old threads, virtual threads aren't GC roots**

- **Since Java 20, Thread.stop() throws an UnsupportedOperationException**

- **ThreadGroup does not enumerate virtual threads**

- **Virtual Thread that cannot be unblocked are GCed**
  - **Caused dumb demos to appear to create billions of threads**

Synchronized ⇒ ReentrantLock

- **synchronized/wait is not fully compatible with Loom**
  - **Virtual thread will stall the underlying carrier thread**
    - **It will create additional threads through ManagedBlocker**

```java
Object monitor = new Object();
for (int i = 0; i < 10_000; i++) {
  Thread.startVirtualThread(() -> {
    synchronized (monitor) {
      try { monitor.wait(); } catch (InterruptedException ignore) {}
    }
  });
}
Thread.startVirtualThread(() -> System.out.println("done")).join();
```

no output

![JavaSpecialists.eu logo]

Object.wait() calles wait(0L)

```java
public final void wait(long timeoutMillis) throws InterruptedException {
    long comp = Blocker.begin();
    try {
        wait0(timeoutMillis);
    } catch (InterruptedException e) {
        Thread thread = Thread.currentThread();
        if (thread.isVirtual())
            thread.getAndClearInterrupt();
        throw e;
    } finally {
        Blocker.end(comp);
    }
}
```

Synchronized ⇒ ReentrantLock

- **We might need to migrate our synchronized code to**
  - **ReentrantLock**
  - **StampedLock**

- **In both cases, idioms are more complicated**
  - **But fully compatible with virtual threads**

- **Synchronized is fine for guarding memory**
  - **Debug with jdk.tracePinnedThreads=full or short**

Biased Locking Turned Off

- **ConcurrentHashMap uses synchronized**
  - **Earlier versions used ReentrantLock**

- **Uncontended ConcurrentHashMap in Java 15 is measurably slower on some old hardware**
  - **-XX:+UseBiasedLocking to enable it again**
 - **Please report if turning it on makes a big difference**

Rather do not use ThreadLocal

- **Virtual threads do support ThreadLocal**

  – **However, it is costly**

  – **Virtual threads not reused**

    • **ThreadLocals often do not make sense**

- **Rather use Scoped Values (Incubator)**
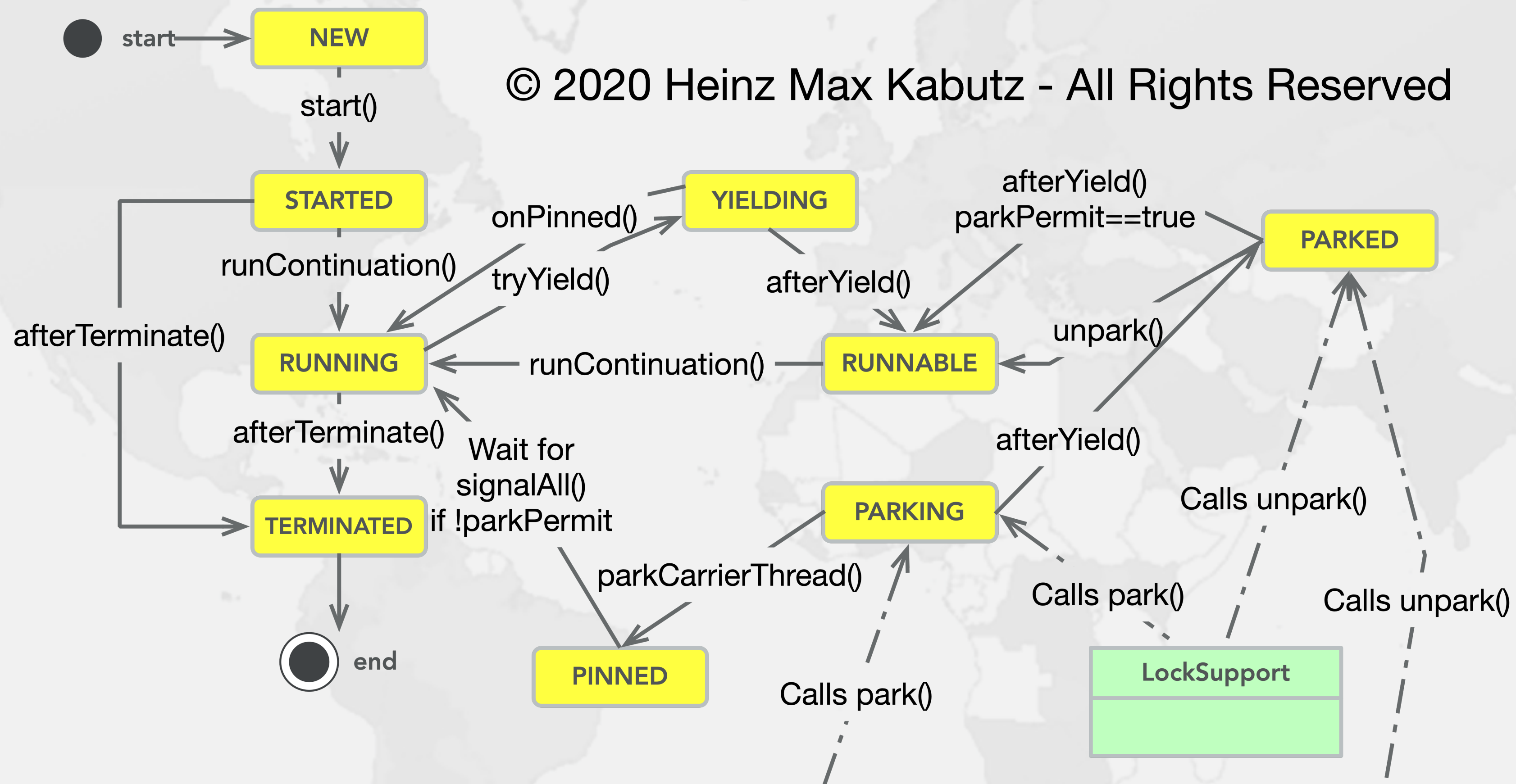
  – **https://openjdk.org/jeps/429**

Naming

- **Virtual threads do not have a name**

  – **Most of the time, sufficient to generate own with threadId()**

    • **Unlike getId(), this threadId() guarantees a unique final value**

java.lang.Thread States
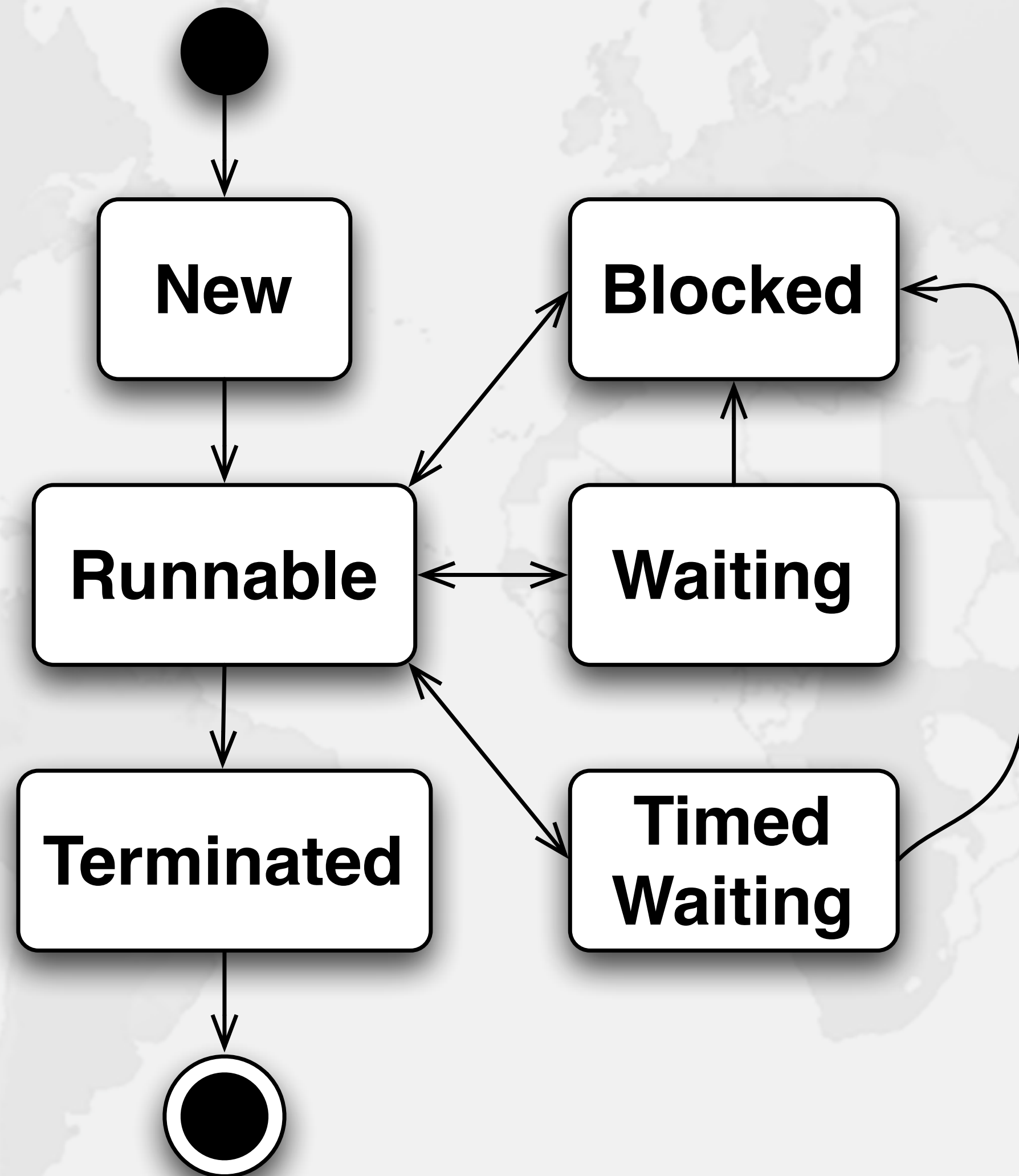
java.lang.VirtualThread States

© 2020 Heinz Max Kabutz - All Rights Reserved

VirtualThread.getState()

VirtualThread State	Thread State
NEW	NEW
STARTED, RUNNABLE	RUNNABLE
RUNNING	if mounted, carrier thread state else RUNNABLE
PARKING, YIELDING	RUNNABLE
PINNED, PARKED, PARKED_SUSPENDED	WAITING
TERMINATED	TERMINATED

JavaSpecialists.eu

Cost of old IO Streams

- **Benefit of Virtual Threads, is we can use the old java.io.InputStream and java.io.Reader**
  - **As opposed to java.nio Channel and Buffer**
- **But, they actually use a lot of memory**

Memory overhead of IO Streams

	OutputStream	InputStream	Writer	Reader
Print	**17400**		80	
Buffered	8312	8296	16488	**16496**
Data	80	328		
File	176	176	936	8552
GZIP	768	1456		
Object	2264	2256		
Adapter			808	**8424**

JavaSpecialists.eu

Used to be slightly worse

	OutputStream	InputStream	Writer	Reader
Print	**25064**		80	
Buffered	8312	8296	16480	**16496**
Data	80	328		
File	176	176	8608	8552
GZIP	768	1456		
Object	2264	2256		
Adapter			8480	**8424**

Deadlocks in Virtual Threads

- ## **Deadlocks with a virtual thread not in thread dump**

  - ### **https://www.javaspecialists.eu/archive/Issue302.html**

```
"platform" #30 cpu=1.75ms elapsed=4.42s waiting for monitor entry
  java.lang.Thread.State: BLOCKED (on object monitor)
    at SimpleLockOrderingDeadlockMixedThreads.lambda$main$0
    - waiting to lock <0x000000043fce3d90> (a java.lang.Object)
    - locked <0x000000043fce3d80> (a java.lang.Object)
    at SimpleLockOrderingDeadlockMixedThreads$$Lambda$14
    at java.lang.Thread.run
"ForkJoinPool-1-worker-1" #32 daemon cpu=0.70ms elapsed=4.41s
  Carrying virtual thread #31
    at jdk.internal.vm.Continuation.run
    at java.lang.VirtualThread.runContinuation
    at java.lang.VirtualThread$$Lambda$22
    at java.util.concurrent.ForkJoinTask$RunnableExecuteAction.exec
    at java.util.concurrent.ForkJoinTask.doExec
    at java.util.concurrent.ForkJoinPool$WorkQueue.topLevelExec
    at java.util.concurrent.ForkJoinPool.scan
    at java.util.concurrent.ForkJoinPool.runWorker
    at java.util.concurrent.ForkJoinWorkerThread.run
```

JavaSpecialists.eu

How to find out what thread #31 is doing?

- **Run the JVM with -Djdk.trackAllThreads=true**

- **Once deadlock occurs**
  - **jcmd pid Thread.dump_to_file some_file**

```
#31 "virtual" virtual
  SimpleLockOrderingDeadlockMixedThreads.lambda$main$1\
    (SimpleLockOrderingDeadlockMixedThreads.java:22)
  java.base/java.lang.VirtualThread.run
  java.base/java.lang.VirtualThread$VThreadContinuation.lambda$new$0
  java.base/jdk.internal.vm.Continuation.enter0
  java.base/jdk.internal.vm.Continuation.enter
```

JavaSpecialists.eu

Deadlocks with ReentrantLock

- **Does not pin the carrier thread**
  - **Much harder to find these**
  - **Good luck!**

Parallel Programming with Loom?

- **Loom for concurrent programming, not parallelism**
  - **Best not to do CPU intensive work in virtual threads**
    - **Use platform threads and ForkJoin or parallel streams**

JavaSpecialists.eu

Trick Question

● **How long will this take to execute?**

```java
public class ParallelismPuzzle {
  public static void main(String... args) {
    long time = System.nanoTime();
    try {
      ForkJoinPool.commonPool().submit(() -> {
        long until = System.currentTimeMillis() + 1000;
        while (System.currentTimeMillis() <= until) ;
      }).join();
    } finally {
      time = System.nanoTime() - time;
      System.out.printf("time = %dms%n", (time / 1_000_000));
    }
  }
}
```

JavaSpecialists.eu

How long?

- **Either about one second**

  – **This is the expected answer**

- **Or forever**

  – **If -Djava.util.concurrent.ForkJoinPool.common.parallelism=0**

How long will this take?

●Obviously depends on common pool parallelism

```java
public class ParallelStreamPuzzle {
  public static void main(String... args) {
    long time = System.nanoTime();
    try {
      IntStream.range(0, Runtime.getRuntime().availableProcessors())
          .parallel()
          .forEach(i -> {
            System.out.println(Thread.currentThread());
            long until = System.currentTimeMillis() + 1000;
            while (System.currentTimeMillis() <= until) ;
          });
    } finally {
      time = System.nanoTime() - time;
      System.out.printf("time = %dms%n", (time / 1_000_000));
    }
  }
}
```

JavaSpecialists.eu

How long?

- **Either about one second**

  – **This is the expected answer**

- **Or longer, but not forever, even with**

  – **-Djava.util.concurrent.ForkJoinPool.common.parallelism=0**

Retrofitting to Asynchronous Code

- **If your system works fine asynchronously, leave it**
  - **Virtual threads help to alleviate some of the pain**
  - **But are not necessarily faster**
  - **And retrofitting them is probably more trouble than worth**

- **Backpressure**
  - **With virtual thread model, use Semaphore or BlockingQueue**
    - **Be careful though, Semaphore is a rather primitive construct**
      - **Example from Spar during lockdown**
      - **If a permit is lost due to an exception, parallelism is reduced**

JavaSpecialists.eu

Don't forget ... almost at the end ...

tinyurl.com/devoxx-el

JavaSpecialists.eu

When will Loom be ready?

- **Virtual threads in Java 21, JEP 444**
  - **Some parts still in incubator stage:**
    - **Structured concurrency (JEP 437) and Scoped Values (JEP 429)**

- **Java has different levels of readiness**
  - **Part of the JDK**
  - **Preview feature - mostly done, supported by all JVMs**
  - **Experimental feature - not necessarily in all Java runtimes**
  - **Incubator**

JavaSpecialists.eu