Prepare for what "Loom"s ahead

Dr Heinz M. Kabutz

Last updated 2020-10-20

Javaspecialists.eu java training

Copyright Notice

whois Heinz?

- **Not ketchup**
- **10+ years teaching remotely from Crete**
  - **learning.javaspecialists.eu**
  - **Threading and concurrency, patterns, advanced topics, etc.**
  - **Contact: heinz@javaspecialists.eu**
- **Java Champion, Speaker, bla bla**

Questions

- Please please please please ask questions in chat!
- Interrupt me at any time
  - My colleague John Green will text them to me
- There are some stupid questions
  - They are the ones you didn’t ask
  - Once you’ve asked them, they are not stupid anymore
- The more you ask, the more we all learn

Why do we need Virtual Threads?

- **We need to find units of work that we can parallelize**
- **A single client request is a natural unit of work**
  - **However, sometimes we can parallelize parts of the request**
  - **For example, some of the data could be retrieved in parallel**

Best Deal Search

- **Let's say our webpage server requires 4 steps**
  1. **We scan the request for search terms**
  2. **We send the request to our partner websites**
  3. **We create our advertising links**
  4. **We collate the results from our partner websites**
- **We can reorder some steps without affecting result**

Example: Sequential Best Deal Search

- **Sequential processing is the simplest**
  - **Simply use the server thread to do all the work**

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

42.5 seconds

Example: Page Renderer with Future

- **We divide the page rendering into two tasks**
  - Create advertising links (CPU bound)
  - Searching partner sites (I/O bound)
- **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

```
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> infos = scanForSearchTerms(request); // 1
    Callable<List<SearchResult>> task = () ->
        infos.stream()
            .map(SearchTerm::searchOnPartnerSite) // 2
            .collect(Collectors.toList());
    Future<List<SearchResult>> results = executor.submit(task);
    createAdvertisingLinks(request); // 3
    results.get().forEach(this::collateResult); // 4
  }
}
```

40.5 seconds

Assigning Heterogenous Tasks

- Dividing work heterogeneously has its limitations
  - Our FutureRenderer not much more efficient than before
  - And code is more complicated

Our system scales best when we can split the work into lots of equal chunks that can run in parallel.

Renderer with CompletionService

- **CompletionService returns tasks as they are done**
  - **We can then collate results in the order they are returned**

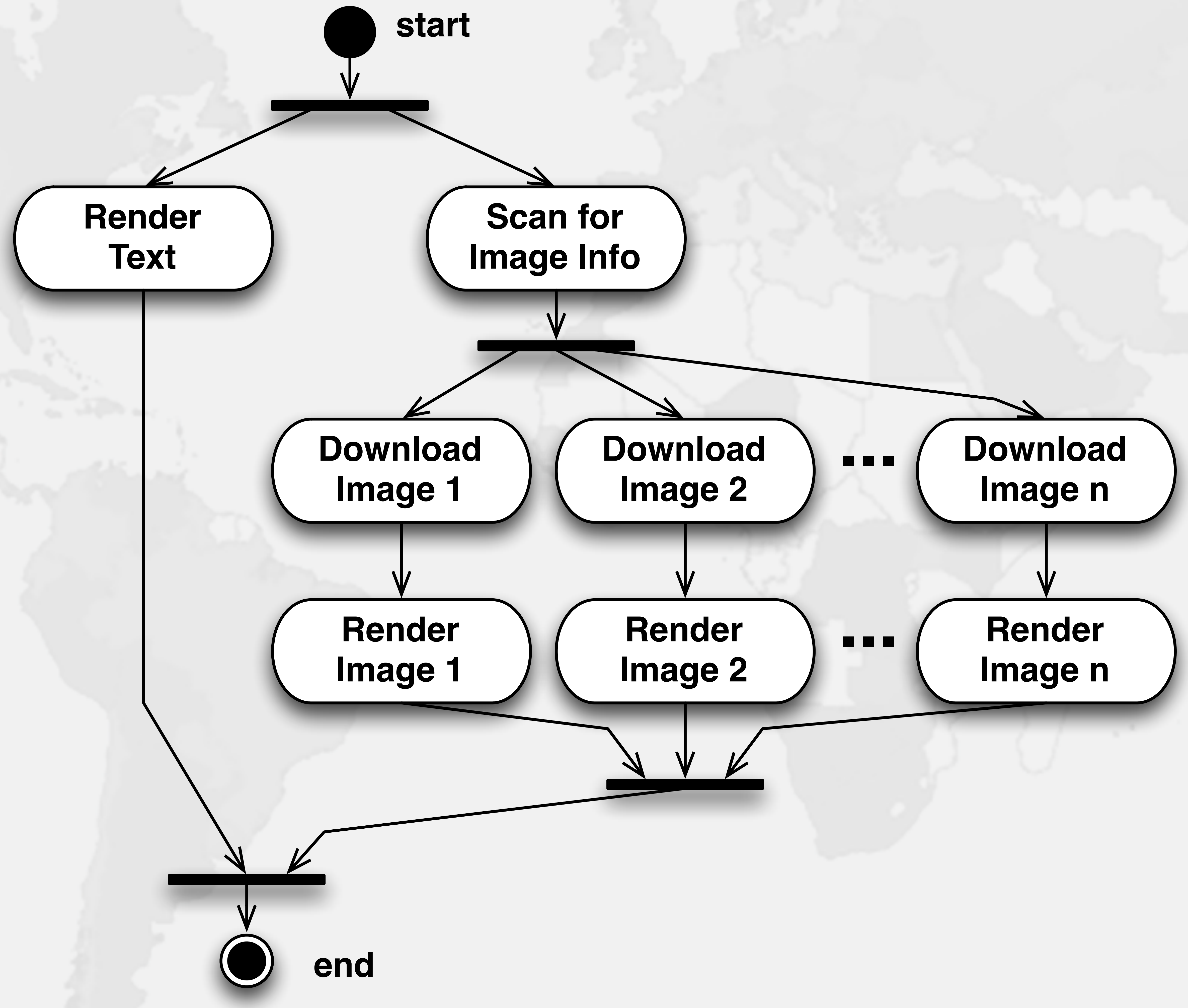
Renderer with CompletionService

22.5 seconds

```java
public class Renderer extends BasicRenderer {
  private final ExecutorService executor;

  public Renderer(ExecutorService executor) {
    this.executor = executor;
  }

  public void renderPage(HttpRequest request)
      throws ExecutionException, InterruptedException {
    List<SearchTerm> terms = scanForSearchTerms(request); // 1
    CompletionService<SearchResult> completionService =
        new ExecutorCompletionService<>(executor);
    terms.forEach(term ->
        completionService.submit(term::searchOnPartnerSite) // 2
    );
    createAdvertisingLinks(request); // 3
    for (int i = 0; i < terms.size(); i++) {
      collateResult(completionService.take().get()); // 4
    }
  }
}
```

CompletableFuture

- **Our Renderer example as an Activity Diagram**



renderPage() with CompletableFuture

```
public class RendererCF extends BasicRenderer {
  private final ExecutorService renderPool;
  private final ExecutorService downloadPool;

  public RendererCF(ExecutorService renderPool,
                    ExecutorService downloadPool) {
    this.renderPool = renderPool;
    this.downloadPool = downloadPool;
  }

  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), renderPool);
  }
```

searchAndCollateResults()

```
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);
}
```

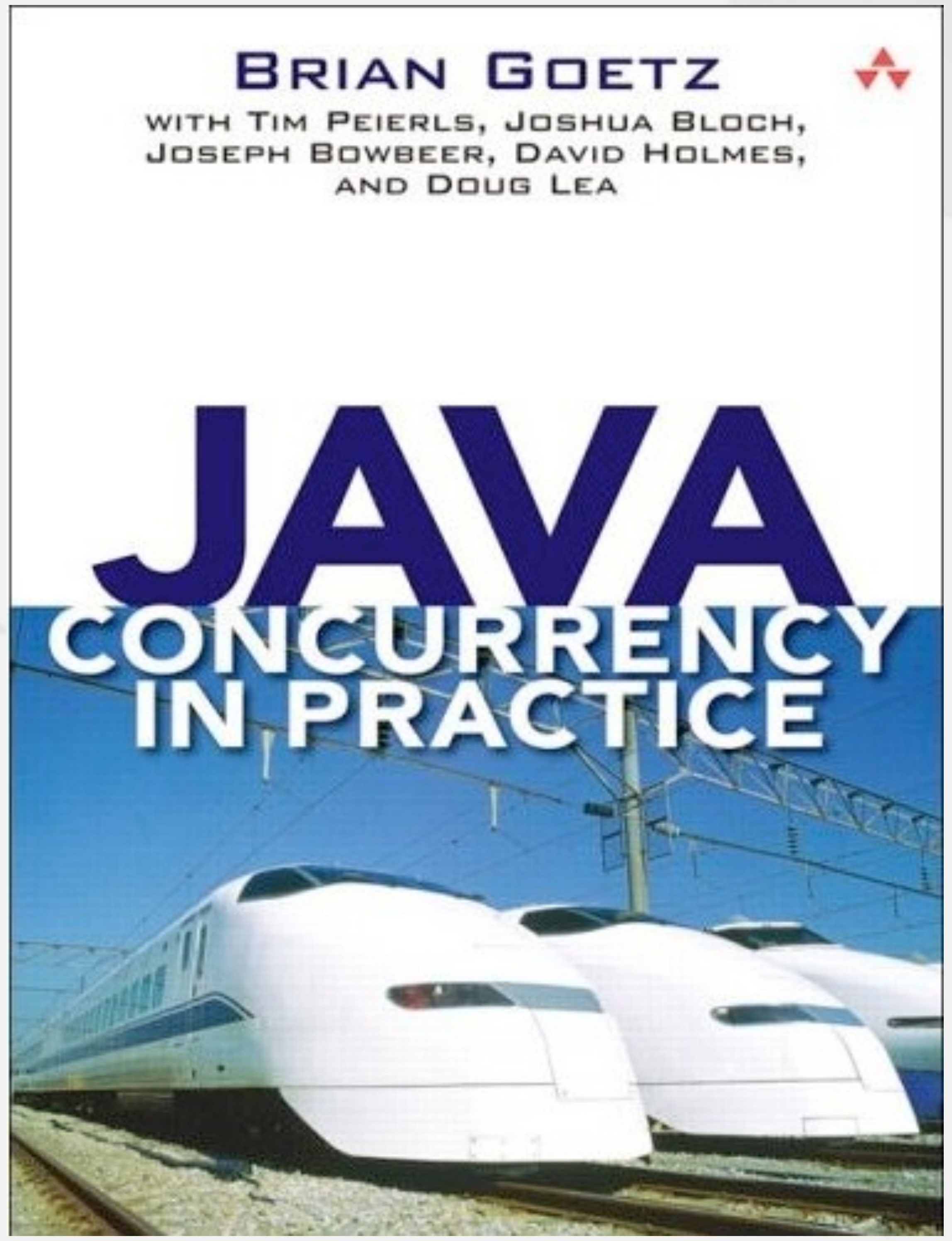
Tasks Wrapped in CompletableFutures

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

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

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

8.5 seconds

Brian Overload?

- **Concurrency Specialist Course**
  - https://www.javaspecialists.eu/courses/concurrency/
- **Only Java concurrency course officially endorsed by Brian Goetz, author of Java Concurrency in Practice**
- **Taught remotely anywhere in the world**
- **Includes all the latest Java concurrency constructs**
  - Virtual threads and Project Loom if so desired



Java Memory Model (JSR 133)

- Describes how Java memory behaves with threads
- Gives a minimum requirement of what must happen
- Allows JVM implementors some freedom
- A correctly written multi-threaded Java application will be correct on every available Java VM
  - A correctly *running* Java application on one JVM could still be incorrect if it breaks JMM laws

Virtual Threads

- Lightweight, about 500 bytes
- Fast to create
- We created 32 million in 16 GB of memory
- Are executed by carrier threads
  - Scheduler is currently a ForkJoinPool
    - Carriers are by default daemon threads
    - # threads is Runtime.getRuntime().availableProcessors()
      - Can temporarily increase due to ManagedBlocker
  - Moved off carrier threads when blocking on IO
    - Also with waiting on synchronizers from java.util.concurrent

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**

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

Virtual threads galore

4.5 seconds

```
public class RendererLoom extends BasicRenderer {
  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())))
        .collect(Collectors.toList());
    createAdvertisingThread.join();
    for (Thread searchThread : searchAndCollateThreads) {
      searchThread.join();
    }
  }
}
```

How to create virtual threads

- **Individual threads**
  - Thread.startVirtualThread(Runnable)
  - Thread.builder().task(Runnable).virtual().start()
    - Thread.Builder can also be used for native threads
- **ExecutorService**
  - Executors.newVirtualThreadExecutor()
  - Executors.newThreadExecutor(ThreadFactory)
  - ExecutorService is now AutoCloseable
    - close() calls shutdown() and awaitTermination()
      - *Not* shutdownNow()

Structured Concurrency

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

4.5 seconds

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, ReentrantReadWriteLock, CountDownLatch, Semaphore
  - Loom: LinkedTransferQueue, SynchronousQueue, SelectorImpl

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 Completely Rewritten

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

AbstractInterruptibleChannel.java

- **ReentrantLock instead of synchronized**
  - github.com/openjdk/loom/commit/5f62bc54f8ff13492af5ffc3e393943a5629da93

```
synchronized (stateLock) {
    ensureOpenAndConnected();
    // record thread so it can be signalled if needed
    readerThread = NativeThread.current();
}
```

```
stateLock.lock();
try {
    ensureOpenAndConnected();
    // record thread so it can be signalled if needed
    readerThread = NativeThread.current();
} finally {
    stateLock.unlock();
}
```

Synchronized ⇒ ReentrantLock

- **Idioms with synchronized are easier to get right**

```
lock.lock();
try {
  // do operation
} finally {
  lock.unlock();
}
```

```
synchronized(monitor) {
  // do operation
}
```

- **Performance of uncontended synchronized is better**
  - **Biased locking assigns the lock to the first thread**
- **Debugging synchronized is easier**
  - **More tools for finding contention**
  - **ReentrantLock.lock() goes into WAITING state**

Synchronized ⇒ ReentrantLock

- **Classes in JDK began moving back to synchronized**
  - **ConcurrentHashMap in Java 8**
  - **CopyOnWriteArrayList in Java 9**
- **But, synchronized/wait is not compatible with Loom**
  - **Virtual thread will stall the underlying carrier thread**

no output

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

Synchronized ⇒ ReentrantLock

- **We need to migrate our synchronized code to**
  - **ReentrantLock**
  - **StampedLock**
- **In both cases, idioms are more complicated**
  - **But compatible with virtual threads**

Biased Locking Turned Off

- ConcurrentHashMap uses synchronized
  - Earlier versions used ReentrantLock
- Uncontended ConcurrentHashMap in Java 15 is measurably slower
  - -XX:+UseBiasedLocking to enable it again
  - Please report if turning it on makes a big difference

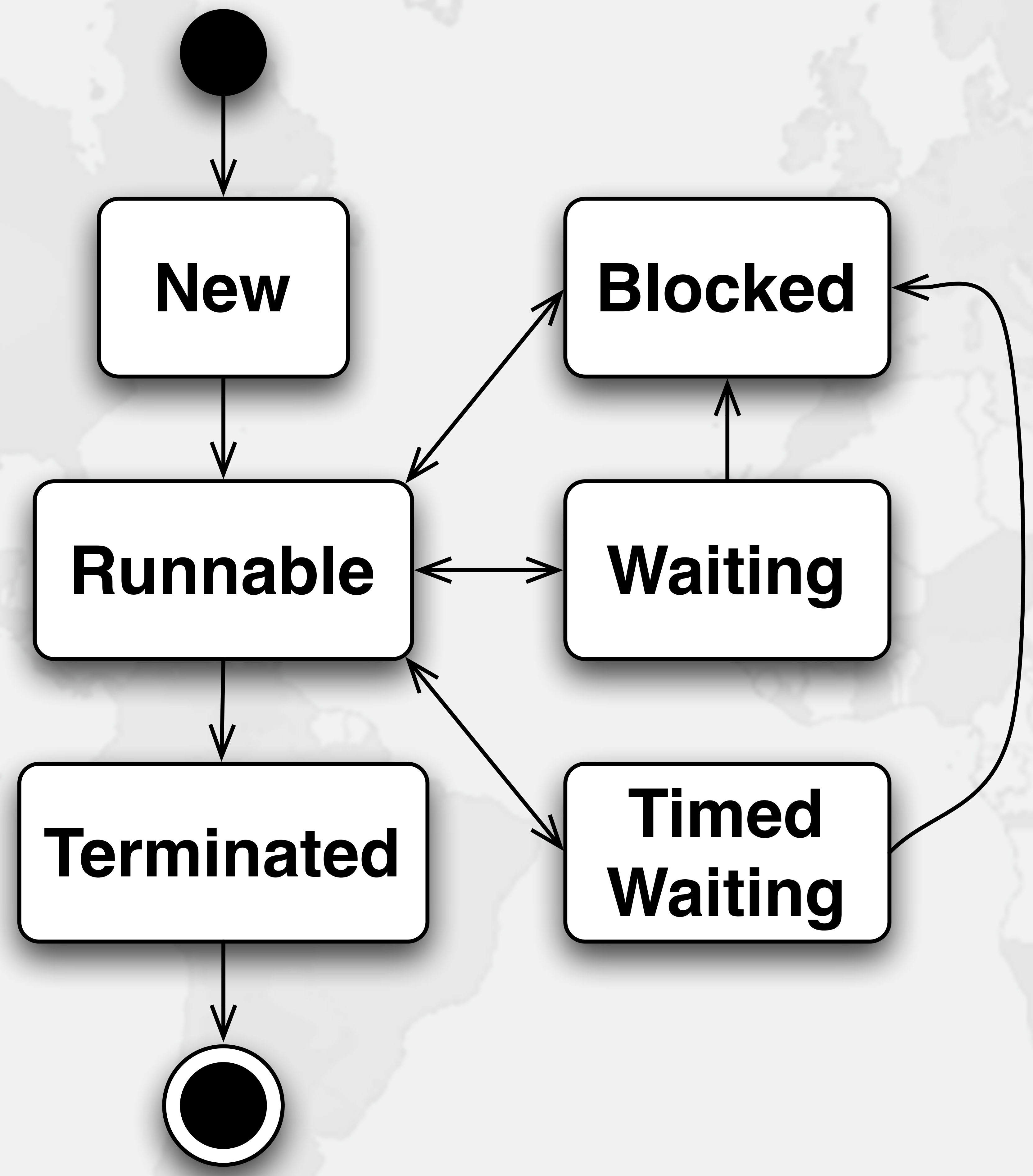
ThreadLocal

- Virtual threads support ThreadLocal by default
  - However, it is costly to support
  - Plus virtual threads are not reused, so ThreadLocals do not make sense
- Better to use either ScopedVariables or shared immutable objects

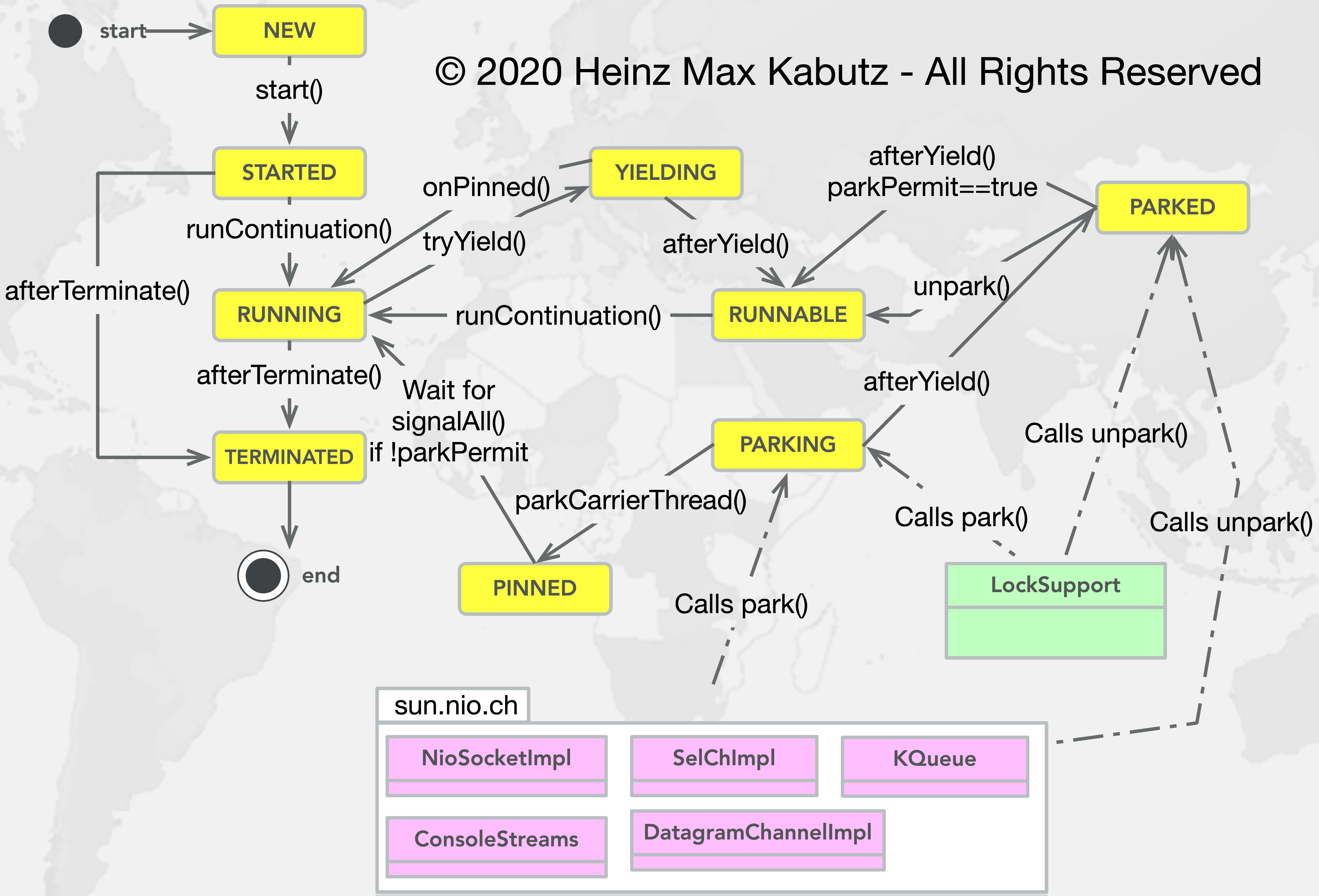
State Machine Disconnect

- java.lang.Thread has six states
  - NEW, RUNNABLE, TERMINATED
  - BLOCKED, WAITING, TIMED_WAITING
- java.lang.VirtualThread has 11 states
  - NEW, STARTED, RUNNING, TERMINATED
  - RUNNABLE, PARKING, PARKED, PINNED, YIELDING
  - SUSPENDED, PARKED_SUSPENDED

java.lang.Thread States



java.lang.VirtualThread States



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

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	25064		80	
Buffer	8312	8296	16480	16496
Data	80	328		
File	176	176	8608	8552
GZIP	768	1456		
Object	2264	2256		
Adapter			8480	8424

Questions?

Twitter: @heinzkabutz