

Prepare for what \*Loom\*s ahead

1

# Prepare for what \*Loom\*s ahead

**Dr Heinz M. Kabutz**

Last updated 2021-10-05

**© 2021 Heinz Kabutz – All Rights Reserved**



**Javaspecialists.eu**  
java training

## Why do we need Virtual Threads?

- **Asynchronous code is hard to debug**
- **1-to-1 Java thread to native thread does not scale**
- **Welcome to Project Loom**
  - **Millions of virtual threads in a single JVM**
  - **Supported by networking, `java.util.concurrent`, etc.**
    - **Anywhere you would block a thread**

## 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**

## Sequential Best Deal Search

- Sequential processing is the simplest

```
public void renderPage(HttpServletRequest 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  
}
```

**4.3 seconds**

## 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

```
public class FutureRenderer extends BasicRenderer {
    private final ExecutorService executor;

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

    public void renderPage(HttpServletRequest request)
        throws ExecutionException, InterruptedException {
        List<SearchTerm> terms = scanForSearchTerms(request); // 1
        Callable<List<SearchResult>> task = () ->
            terms.stream()
                .map(SearchTerm::searchOnPartnerSite) // 2
                .collect(Collectors.toList());
        Future<List<SearchResult>> results = executor.submit(task);
        createAdvertisingLinks(request); // 3
        results.get().forEach(this::collateResult); // 4
    }
}
```

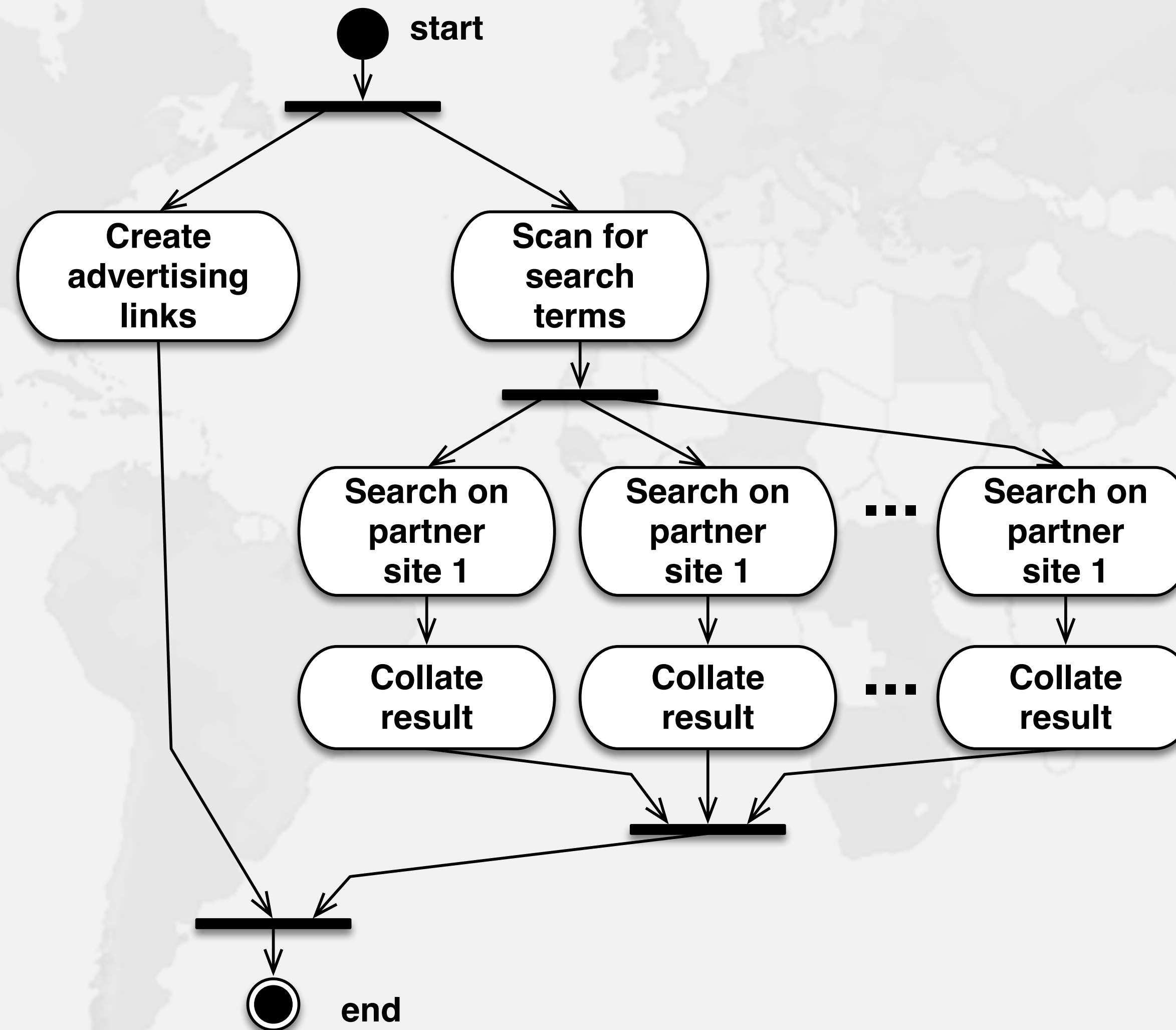
4.1 seconds

## CompletableFuture

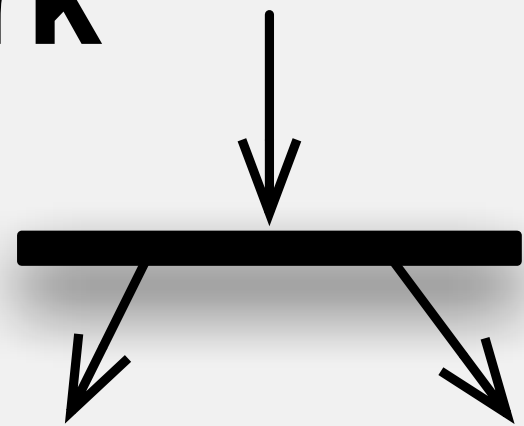
- **Convert each step into a CompletableFuture**
  - Then combine these using *allOf()*
  - Code is slightly faster, but a whole lot more complicated
    - Need separate pools for CPU and IO bound tasks

# Modeling Control Flow

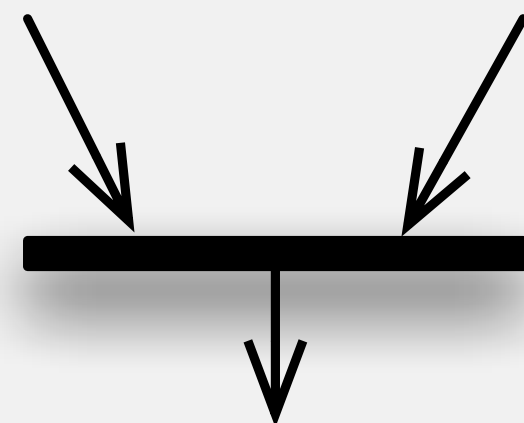
- Our Renderer example as a UML Activity Diagram



- Fork



- Join





## renderPage() with CompletableFuture

```
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(HttpServletRequest request) {
        renderPageCF(request).join();
    }

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

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

# searchAndCollateResults()

```
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);
}
```

# Tasks Wrapped in CompletableFutures

```
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**

## What about plain Thread?

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

# renderPage() with native threads

```
public void renderPage(HttpServletRequest request) throws InterruptedException {
    ThreadMXBean tmb = ManagementFactory.getThreadMXBean();
    long threads = tmb.getTotalStartedThreadCount();

    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;
            })
            .collect(Collectors.toList());
    createAdvertisingThread.join();
    for (Thread searchAndCollateThread : searchAndCollateThreads) {
        searchAndCollateThread.join();
    }

    threads = tmb.getTotalStartedThreadCount() - threads;
    System.out.println("threads = " + threads);
}
```

**threads = 11**  
**0.5 seconds**

## 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 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`**

## Before we continue ...

- **Get our Data Structures in Java Course here**
  - <https://tinyurl.com/jhug21>
  - **Coupon expires at 8pm Athens Time**





## 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(HttpServletRequest 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

```
public void renderPage(HttpServletRequest 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();
}
```

**0.5 seconds**

## How to create virtual threads

- **Individual threads**

- `Thread.startVirtualThread(Runnable)`
- `Thread.ofVirtual().start(Runnable)`

- **ExecutorService**

- `Executors.newVirtualThreadExecutor()`
- **ExecutorService is now AutoCloseable**
  - `close()` calls `shutdown()` and `awaitTermination()`

# Structured Concurrency

```
public void renderPage(HttpServletRequest 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(term -> searchAndCollatePool.submit( // 2 & 4  
                    () -> collateResult(term.searchOnPartnerSite())));  
            }  
        });  
    }  
}
```

**0.5 seconds**

## Deadlines for ExecutorService

- **We can create virtual thread pool with deadline**

```
ExecutorService pool = Executors.newVirtualThreadExecutor(  
    Instant.now().plusSeconds(1));
```

- **Our virtual threads are interrupted on timeout**
  - **We need to regularly check our interrupted status**
  - **A good approach is to then throw a CancellationException**

```
private void sleepSilently(int millis) {  
    try {  
        Thread.sleep(millis);  
    } catch (InterruptedException e) {  
        Thread.currentThread().interrupt();  
        throw new CancellationException("interrupted");  
    }  
}
```

- **No easy way currently to know which task timed out**

## Structured Concurrency with Deadline

```
Collection<Future<?>> futures = new ConcurrentLinkedQueue<>();
try (ExecutorService mainPool = Executors.newVirtualThreadExecutor(
    Instant.now().plusMillis(300))) {
    Stream.of(
        mainPool.submit(() -> createAdvertisingLinks(request)), // 3
        mainPool.submit(() -> {
            List<SearchTerm> terms = scanForSearchTerms(request); // 1
            try (ExecutorService searchAndCollatePool =
                Executors.newVirtualThreadExecutor()) {
                terms.stream()
                    .map(term -> searchAndCollatePool.submit( // 2 & 4
                        () -> collateResult(term.searchOnPartnerSite())))
                    .forEach(futures::add);
            }
        })
    ).forEach(futures::add);
}
if (futures.stream().anyMatch(Future::isCancelled))
    throw new TimeoutException("Timed out");
```

**0.3 seconds**

***TimeoutException***

**Not in latest Loom  
Version**

## 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
  - Java 17: LinkedTransferQueue, SynchronousQueue
  - Loom: SelectorImpl, Object.wait(), old I/O

## 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>

## Synchronized $\Rightarrow$ ReentrantLock

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

```
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**

## Object.wait()

```
public final void wait(long timeoutMillis)
    throws InterruptedException {
    Thread thread = Thread.currentThread();
    if (thread.isVirtual()) {
        try {
            Blocker.managedBlock(() -> wait0(timeoutMillis));
        } catch (Exception e) {
            if (e instanceof InterruptedException)
                thread.getAndClearInterrupt();
            throw e;
        }
    } else {
        wait0(timeoutMillis);
    }
}
```

## Synchronized $\Rightarrow$ 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

## Biased Locking Turned Off

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

## Rather do not use ThreadLocal

- **Virtual threads support ThreadLocal by default**
  - However, it is costly
  - Virtual threads not reused
    - ThreadLocals often do not make sense
- **Disallow with `Builder.allowSetThreadLocals(false)`**

```

public class ThreadLocalTest {
    private static final ThreadLocal<DateFormat> df =
        ThreadLocal.withInitial(() ->
            new SimpleDateFormat("yyyy-MM-dd") {
                {
                    System.out.println("Making SimpleDateFormat");
                }
            }
        );

    public static void main(String... args) throws Exception {
        Runnable task = () -> {
            try {
                for (int i = 0; i < 3; i++) {
                    System.out.println(df.get().parse("2020-05-04"));
                }
            } catch (ParseException e) { e.printStackTrace(); }
        };
        System.out.println("Standard Virtual Thread");
        Thread.startVirtualThread(task).join();

        System.out.println();

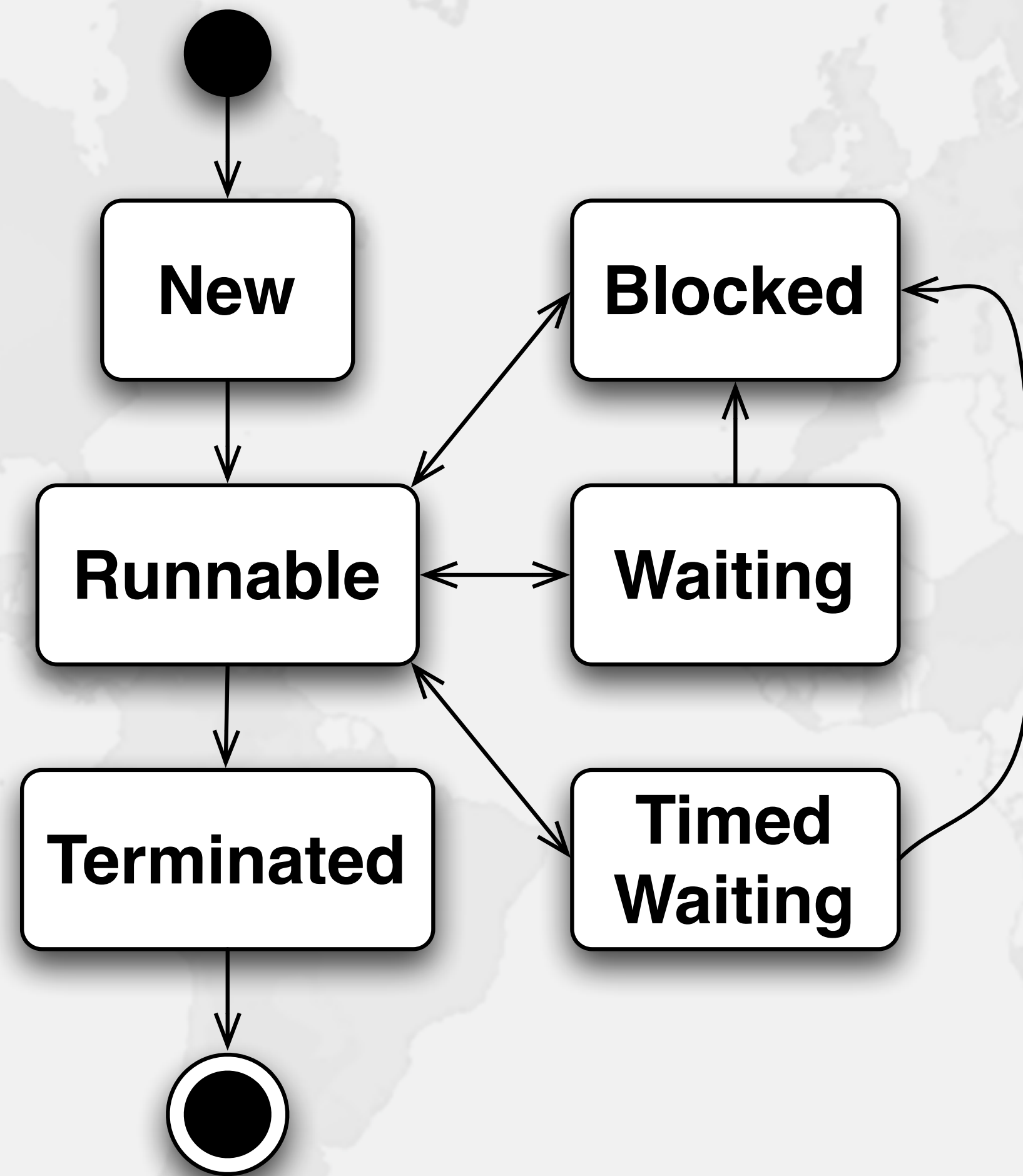
        System.out.println("Disallowing Thread Locals");
        Thread.ofVirtual().allowSetThreadLocals(false)
            .start(task).join();
    }
}

```

Standard Virtual Thread  
 Making SimpleDateFormat  
 Mon May 04 00:00:00 EEST 2020  
 Mon May 04 00:00:00 EEST 2020  
 Mon May 04 00:00:00 EEST 2020

Disallowing Thread Locals  
 Making SimpleDateFormat  
 Mon May 04 00:00:00 EEST 2020  
 Making SimpleDateFormat  
 Mon May 04 00:00:00 EEST 2020  
 Making SimpleDateFormat  
 Mon May 04 00:00:00 EEST 2020

# java.lang.Thread States







# VirtualThread.getState()

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

## 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	<b>17400</b>		80	
Buffered	8312	8296	16488	<b>16496</b>
Data	80	328		
File	176	176	936	8552
GZIP	768	1456		
Object	2264	2256		
Adapter			808	<b>8424</b>

# Used to be slightly worse

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

## Education is Key

- **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 on request
- **Don't forget gift: <https://tinyurl.com/jhug21>**

