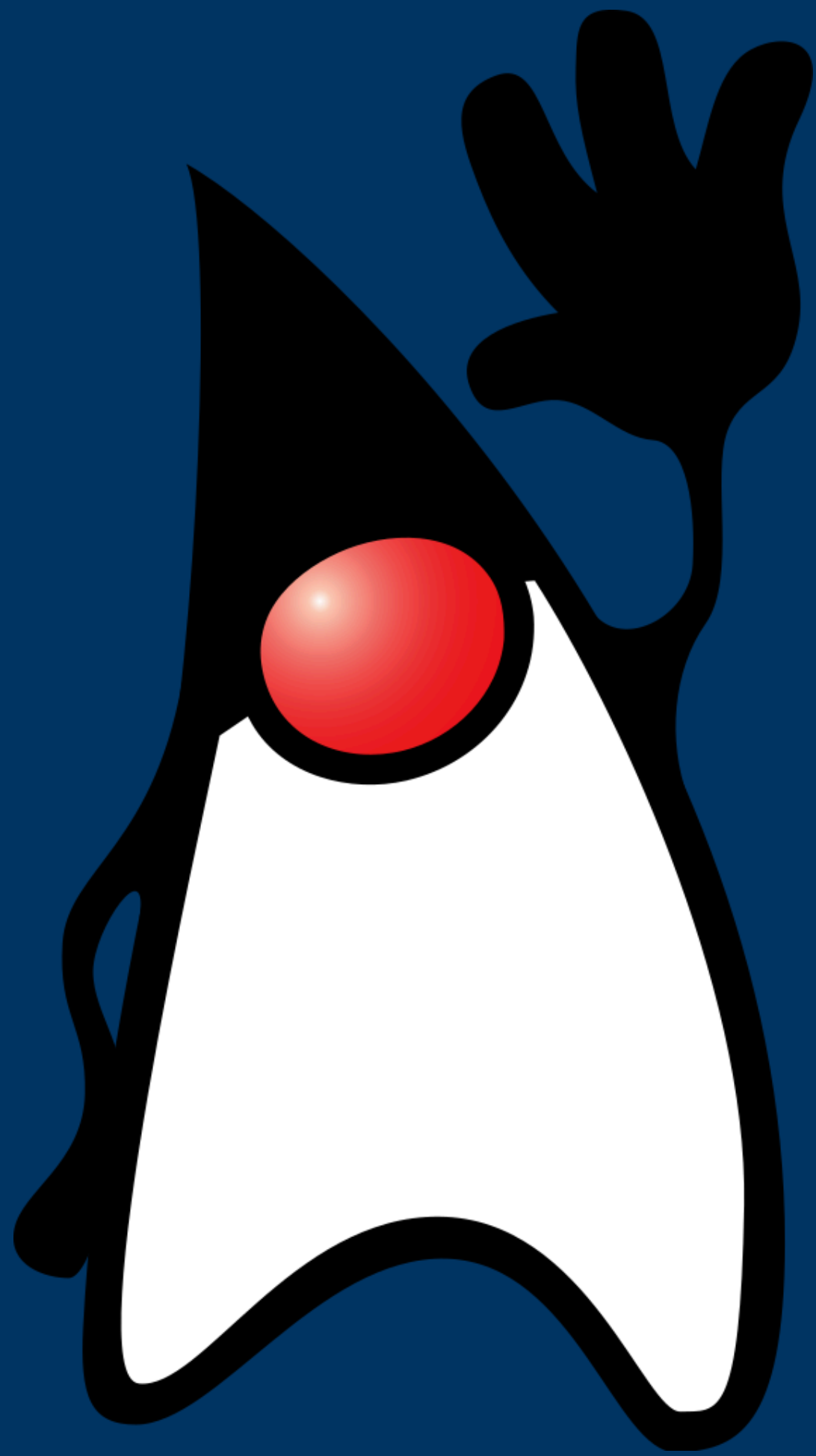


# Project Loom

Fibers and Continuations



**Slides and Repository Located at:**  
**[https://github.com/dhinojosa/loom-](https://github.com/dhinojosa/loom-workshop)**  
**[workshop](https://github.com/dhinojosa/loom-workshop)**



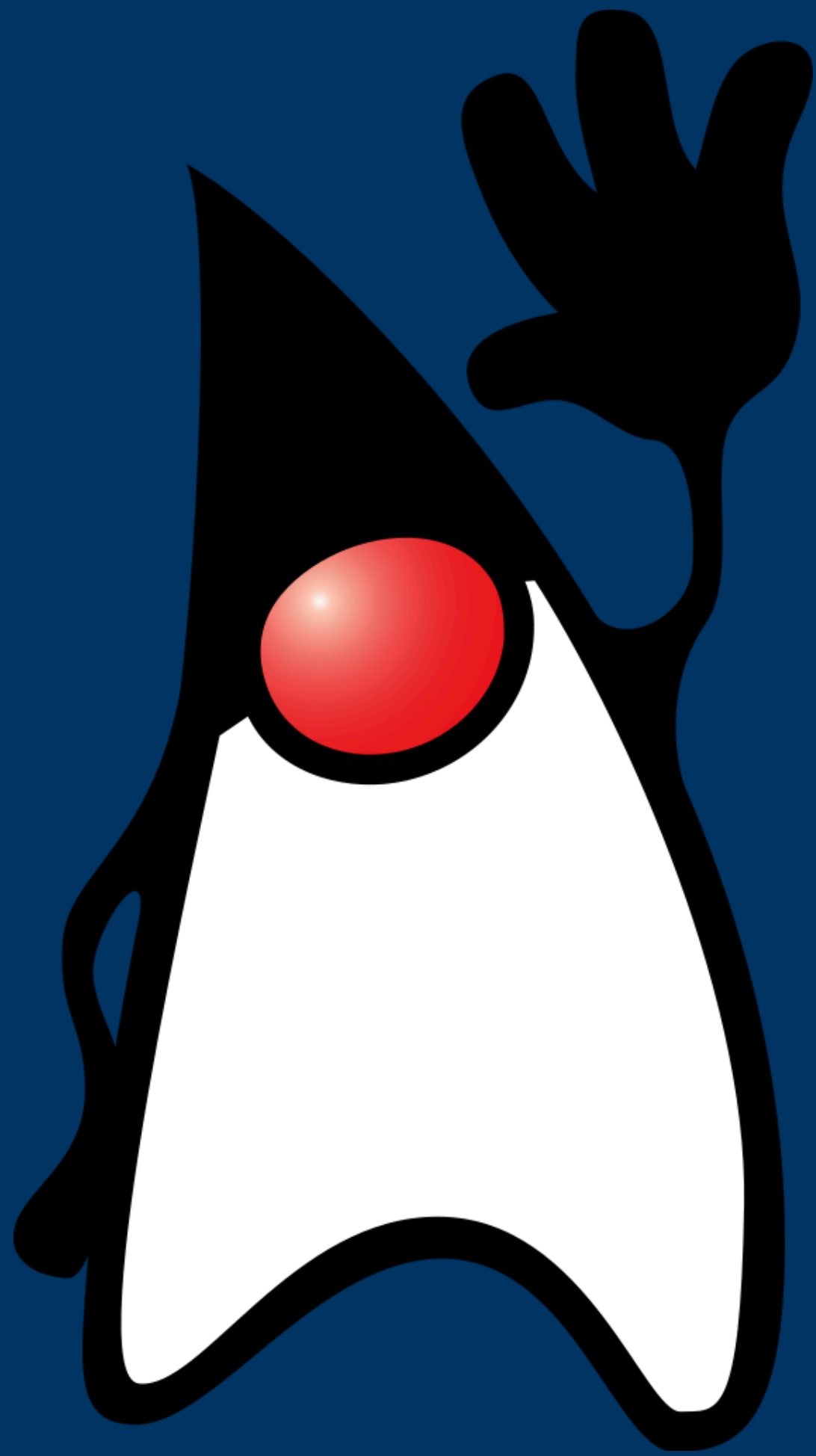
# What is Project Loom?

# Project Loom

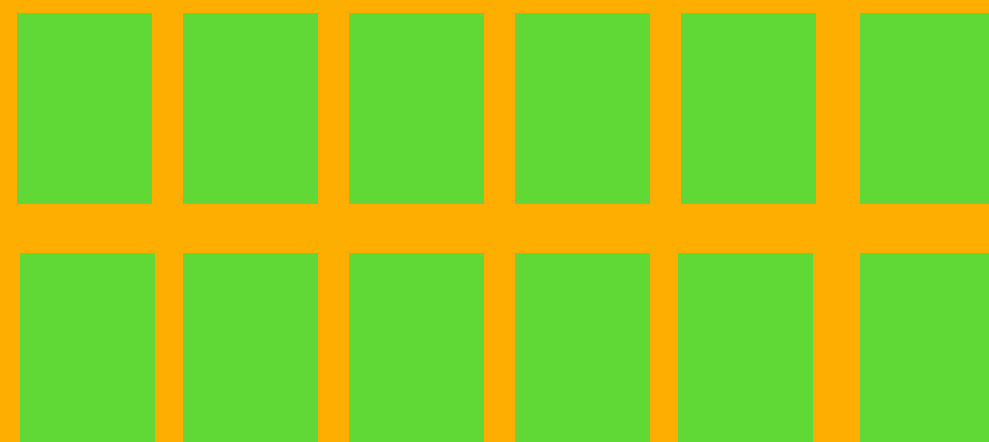
“easy-to-use, high-throughput lightweight concurrency and new programming models on the Java platform.”

## ● Key Elements

- A virtual thread is a Thread — in code, at runtime, in the debugger and in the profiler.
- A virtual thread is not a wrapper around an OS thread, but a Java entity.
- Creating a virtual thread is cheap — have millions, and **don't pool them!**
- Blocking a virtual thread is cheap — **be synchronous!**
- No language changes are needed.
- Pluggable schedulers offer the flexibility of asynchronous programming.



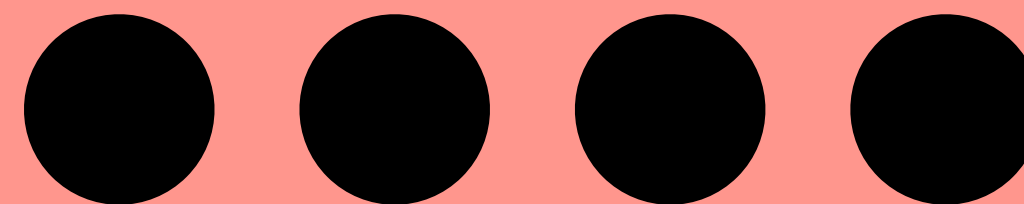
# History



OS Thread



JDK 1.0 - JDK 1.2



OS

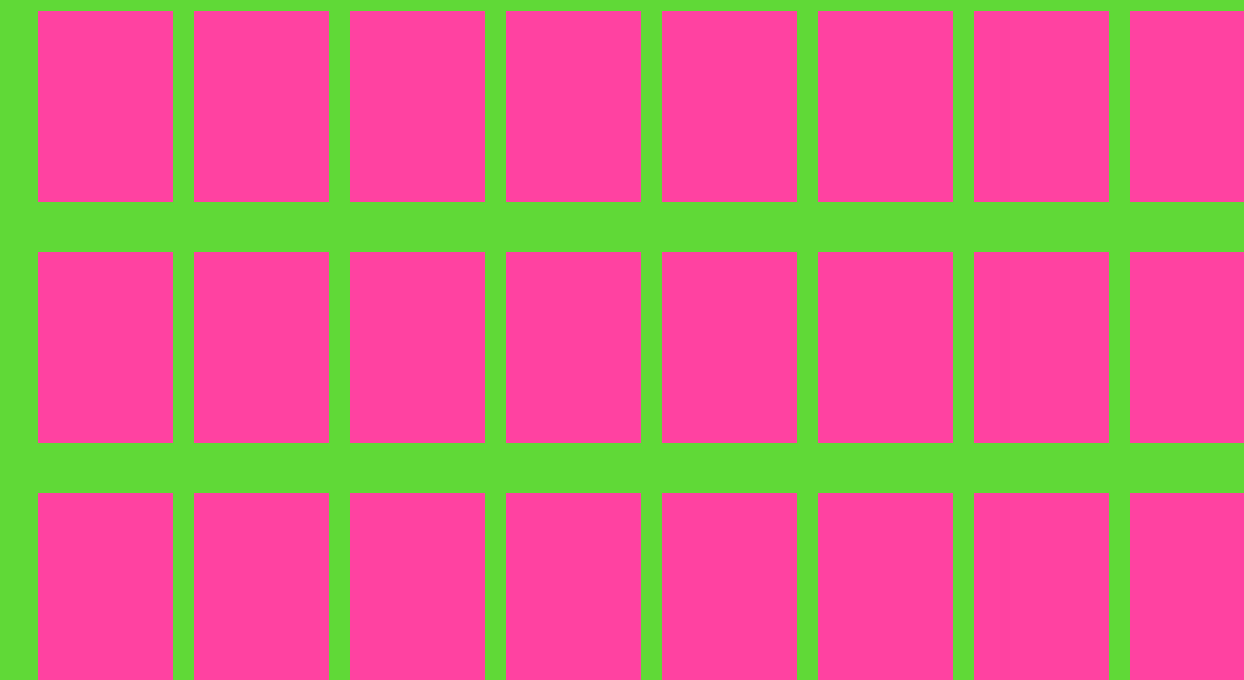
OS

OS

OS



JDK 1.2 - JDK (Current)



OS

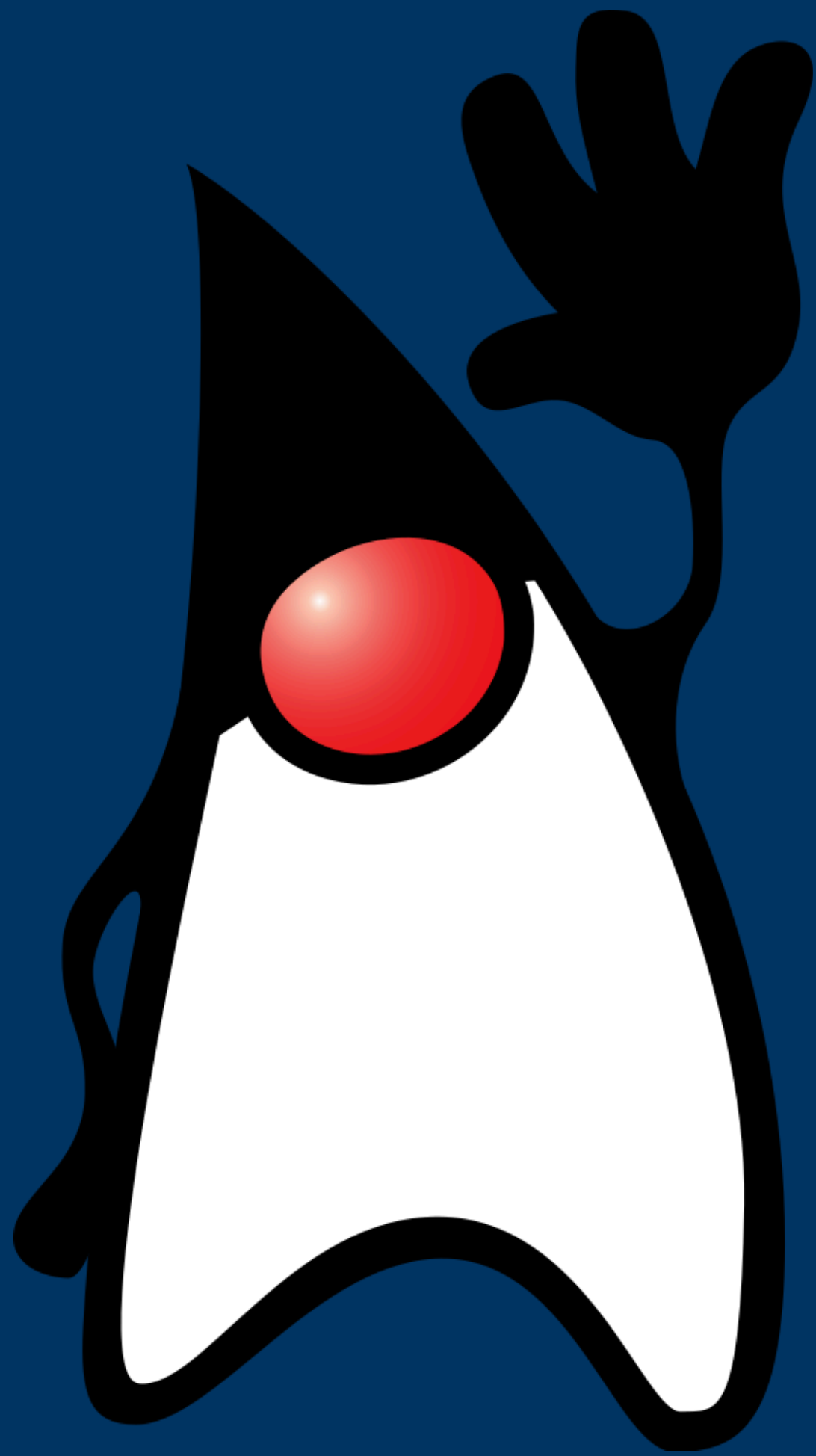
OS

OS

OS



JDK 17 ? + Loom

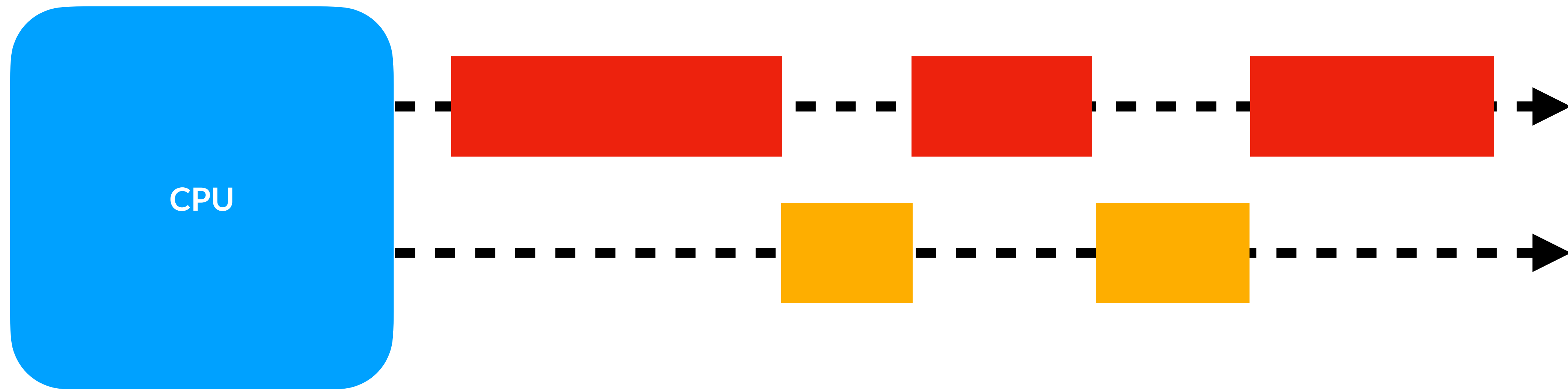


# Concepts

# Concurrency v. Parallelism



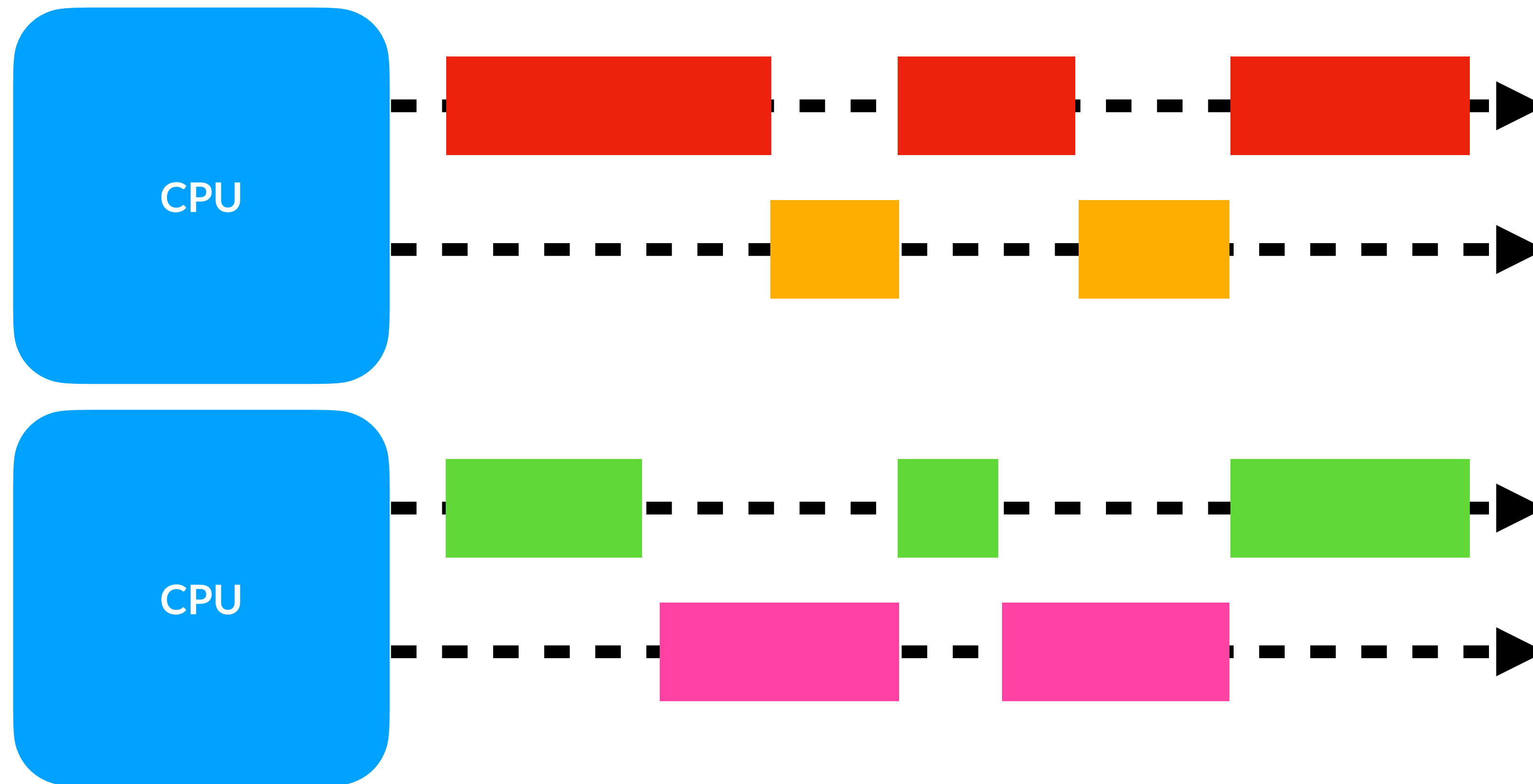
# About Concurrency



# About Parallelism



# About Concurrency & Parallelism



# Context Switching

- If there are more runnable threads than CPUs, eventually **the OS** will *preempt* one thread so that another can use the CPU.
- This causes a context switch, which requires saving the execution context of the currently running thread and restoring the execution context of the newly scheduled thread.

# Schedulers

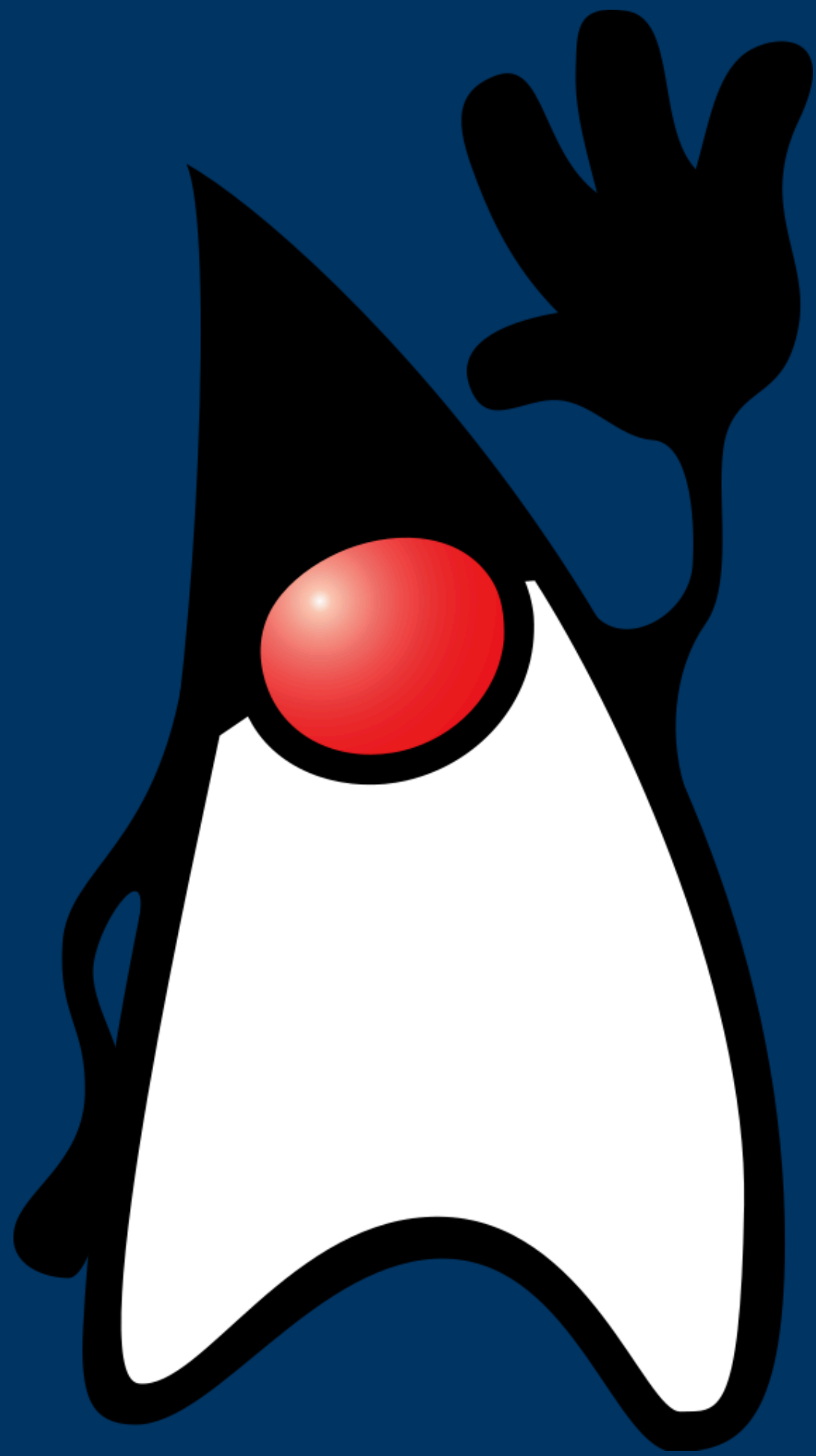
- In our dream world, it would be great if we had one to one thread to cpu correspondence
- Either the JVM or the underlying platform's operating system deciphers how to share the processor resource among threads, this is *Thread Scheduling*
- The JVM or OS that performs the scheduling of the threads is the *Thread Scheduler*

# User vs. Kernel

- User Level is `java.lang.Thread`
  - Runs within the JVM
  - Specific Handling which is up to the language, in this case Java
- Kernel Threads are more general
  - Each user-level thread created by the application is known to the kernel

# Blocking Operations

- Operations that will hold the Thread until complete
- You cannot avoid it entirely
- Examples:
  - `Thread.sleep`, `Thread.join`
  - `Socket.connect`, `Socket.read`, `Socket.write`
  - `Object.wait`
  - and more <https://wiki.openjdk.java.net/display/loom/Blocking+Operations>



What are some problems  
we face today?



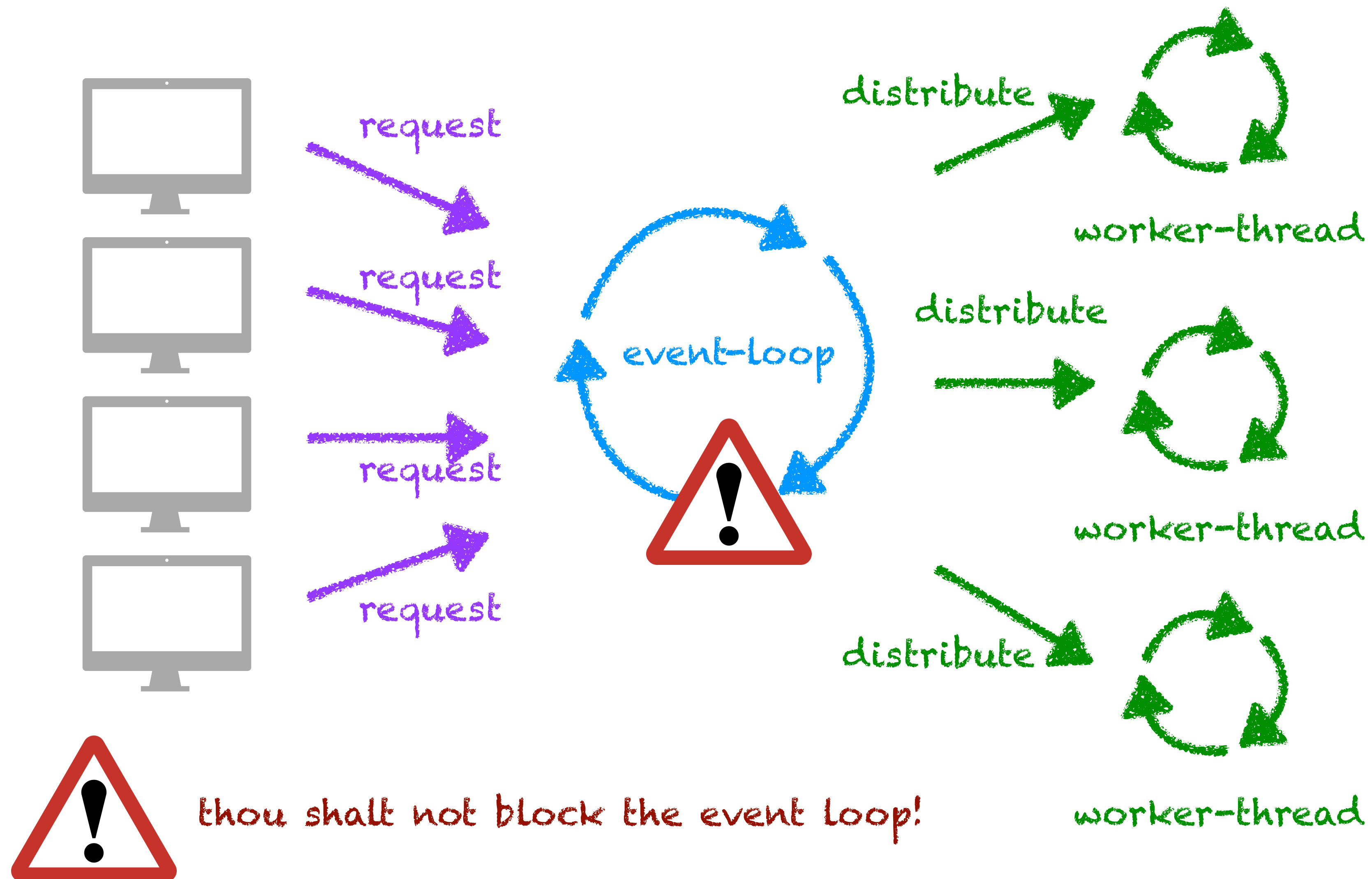
# Thread Weight and Expense

- Threads by themselves are expensive to create
- We cannot have millions of Threads
- Threads are mostly idle, we are not maximizing its use
- Threads are also non-scalable

# How do we compensate?

- `java.util.Future<T>`
- Callbacks
- Async/Await
- Promises or Incomplete Futures
- Reactive Programming (RXJava, Reactor)
- Coroutines, Suspendable Functions

# Sample from my Vert.X presentation



# RXJava/Project Reactor

```
Observable.just(1, 3, 4, 5, 10, 100)    // io
    .doOnNext(i -> debug("L1", i))      // io
    .observeOn(Schedulers.computation()) // Scheduler change
    .doOnNext(i -> debug("L2", i))      // computation
    .map(i -> i * 10)                  // computation
    .observeOn(Schedulers.io())          // Scheduler change
    .doOnNext(i -> debug("L3", i))      // io
    .subscribeOn(schedulers.io())     // Dictate at beginning to use io
    .subscribe(System.out::println);    // Print
```

# ThreadLocal

- If you have a data structure that isn't safe for concurrent access, you can sometimes use an instance per thread, hence ThreadLocal.
- ThreadLocals have been used for Thread Locality
- ThreadLocal has some shortcomings. They're unstructured, they're mutable
- Once a ThreadLocal value is set, it is in effect throughout the thread's lifetime or until it is set to some other value
- ThreadLocal is shared among multiple tasks
- ThreadLocals can leak into one another

# Example of Thread Local

Thread local may already be set

```
var oldValue = myTL.get();  
myTL.set(newValue);  
try {  
    ...  
} finally {  
    myTL.set(oldValue);  
}
```

# Complicated Tracing

- Using ThreadLocals in the way we do threading now, it is complicated to appropriately to do spans
- Spans require inheritance, where a ThreadLocal or Inheritance Thread Local is inherited *by copy* to a subthread.
- ThreadLocals are mutable and cannot be shared hence the copy
- If ThreadLocals were immutable, then it would be efficient to handle
- Be aware that setting the same ThreadLocal would cause an IllegalStateException

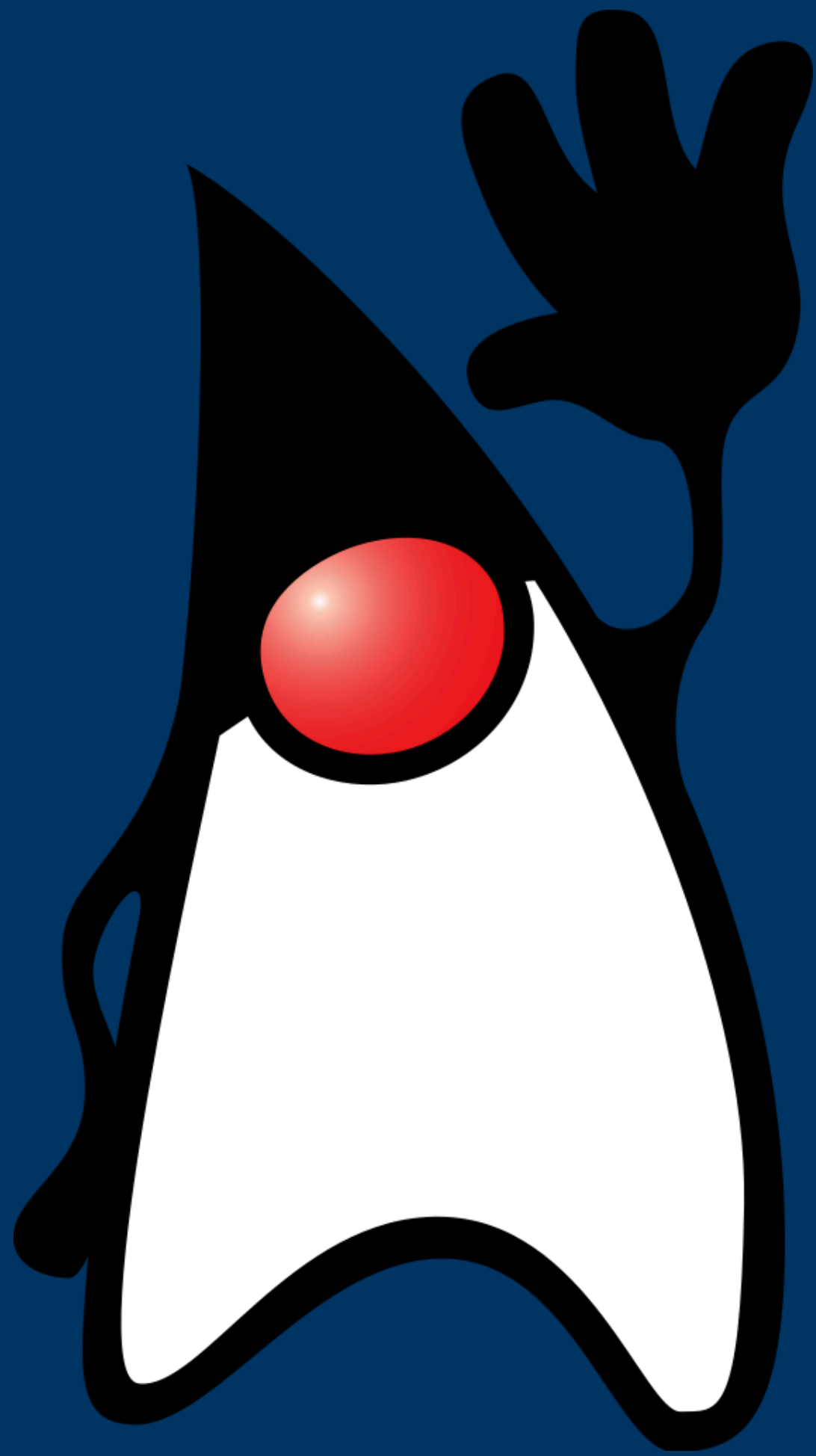
# Complex Cancellation

- Threads support a cooperative interruption mechanism comprised of:
  - `interrupt()`
  - `interrupted()`
  - `isInterrupted()`
  - `InterruptedException`
- When a thread is interrupted it has to clear and reset the status because it has to go back into a pool



# Losing Stack Traces

- Stack Traces are also applicable to the live thread
- If the thread is recycled back into a pool, then the stack trace is lost
- This can prove to be valuable information



# Virtual Threading

# What is Virtual Threading

- **Virtual threads** are just threads, except lightweight 1kb and fast to create
- Creating and blocking virtual thread is cheap and encouraged. 23 million virtual threads in 16GB of memory
- They are managed by the Java runtime and, unlike the existing platform threads, are not one-to-one wrappers of OS threads, rather, they are implemented in userspace in the JDK.
- Whereas the OS can support up to a few thousand active threads, the Java runtime can support millions of virtual threads
- Every unit of concurrency in the application domain can be represented by its own thread, making programming concurrent applications easier

# Continuation

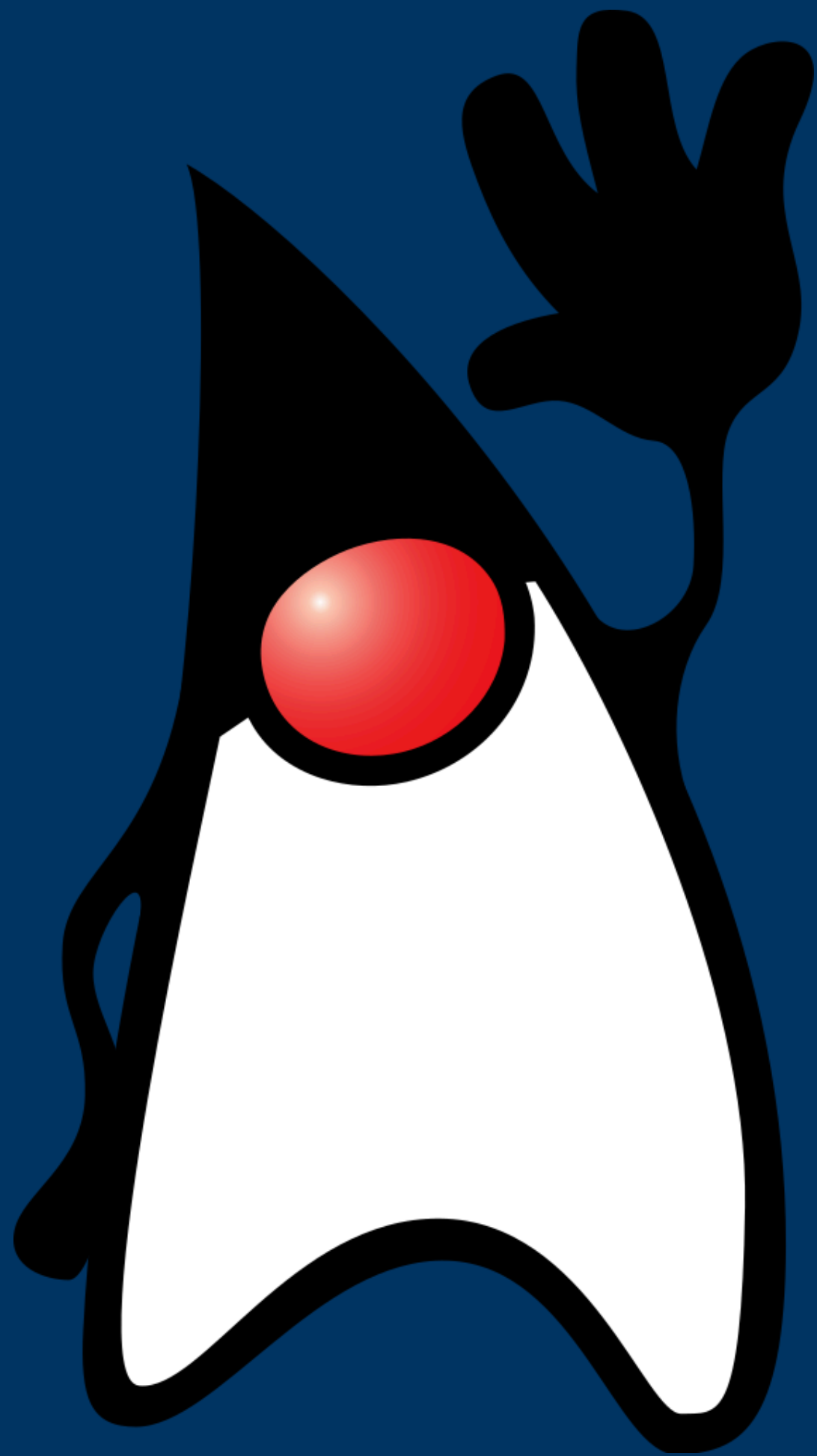
- Object Representing a Computation that may be suspended, resumed, cloned or serialized
- When it reaches a call that blocks it will yield allowing **the JVM** to use another virtual thread to process.

```
public class Main {  
    public static void main(String[] args)  
    {  
        int num = 29;  
        boolean flag = false;  
        for (int i = 2; i <= num / 2; ++i) {  
            // condition for nonprime number  
            if (num % i == 0) {  
                flag = true;  
                break;  
            }  
        }  
    }  
}
```

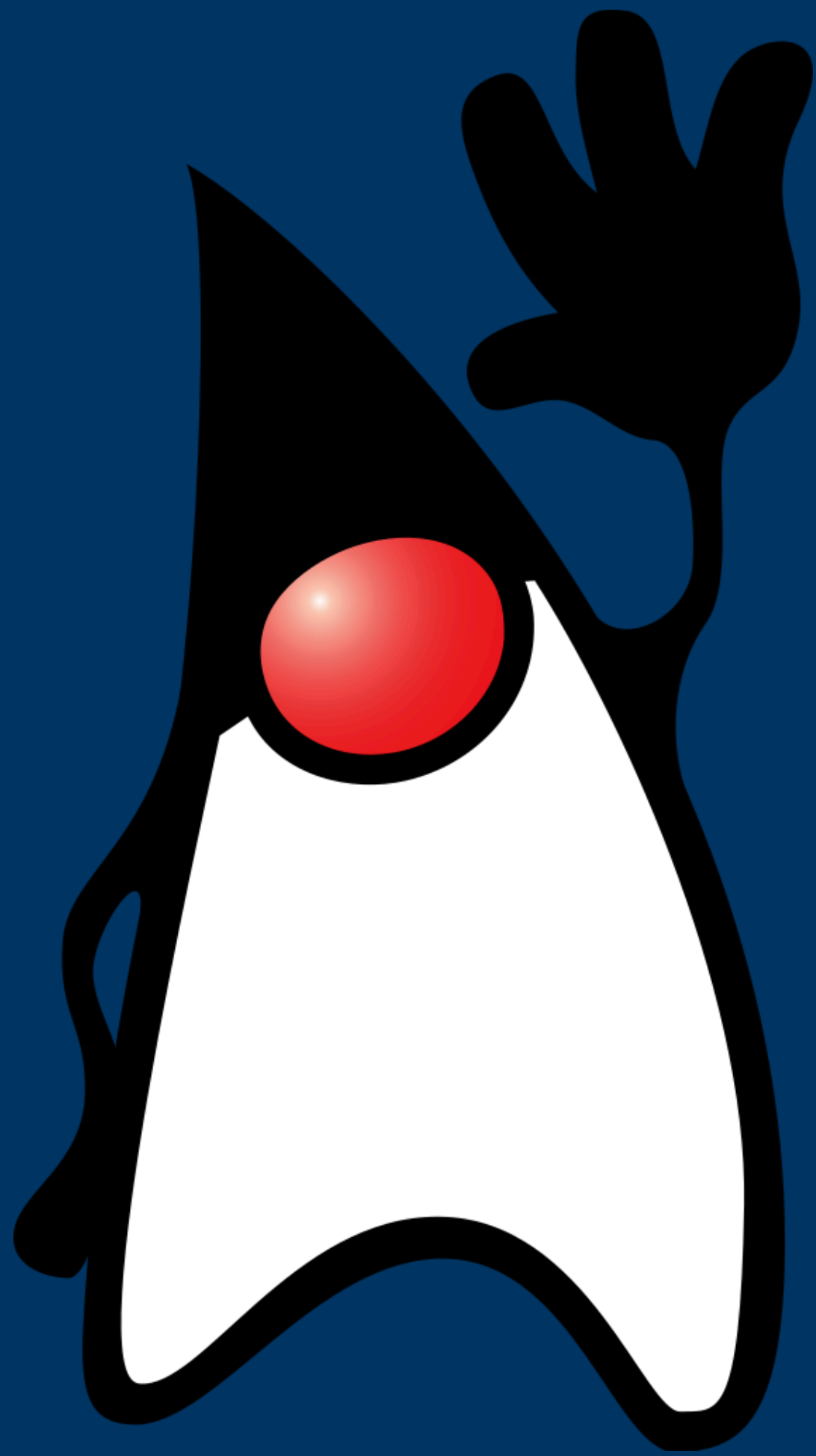
Virtual Thread

java.lang.Thread





# Demo: Continuations



# How Scheduling Works

# Scheduling

- The Kernel Scheduler is very general, and makes assumptions about what the user language requires
- Virtual Threads are tailored and specific for the task and are on the JVM
- Virtual Threads Scheduling with the Kernel Space is abstracted, and typically you don't need to know much unless you feel pedantic or want to create your own Scheduler
- Your Virtual Threads that you create will be running on a Worker Thread called a "Carrier Thread"

# Executing by Carrier Threads

- Scheduler is implemented with a ForkJoinPool
- Carrier Thread are daemon threads
- The number of initial threads is `Runtime.getRuntime().availableProcessors()`
- New Threads can be initialized with a Managed Blocker - If a thread is blocked, new threads are created





- Parking (blocking) a virtual thread results in yielding its continuation
- Unparking it results in the continuation being resubmitted to the scheduler.
- The scheduler worker thread executing a virtual thread (while its continuation is mounted) is called a carrier thread.

# Virtual Thread on a Carrier



```
public class Main {  
    public static void main(String[] args)  
    { int num = 29;  
      boolean flag = false;  
      Thread.sleep(30);  
      for (int i = 2; i <= num / 2; ++i) {  
          // condition for nonprime number  
          if (num % i == 0) {  
              flag = true;  
              break;  
          }  
      }  
    }
```

Virtual Thread

java.lang.Thread

# Virtual Thread on a Carrier



```
public class Main {  
    public static void main(String[] args)  
    {  
        int num = 29;  
        boolean flag = false;  
        Thread.sleep(20);  
        for (int i = 2; i <= num / 2; ++i) {  
            // condition for nonprime number  
            if (num % i == 0) {  
                flag = true;  
                break;  
            }  
        }  
    }  
}
```

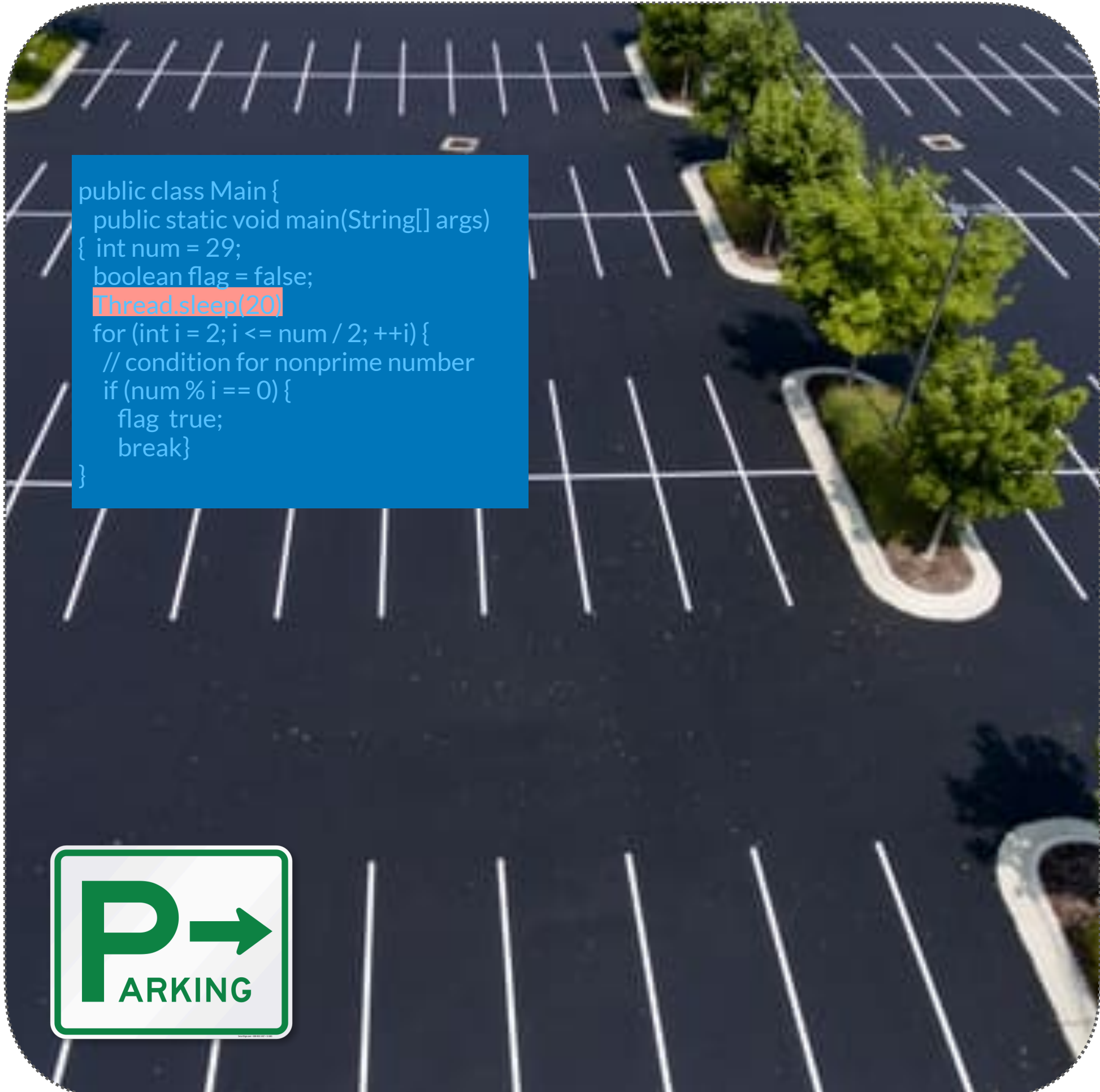
Block!

Virtual Thread


java.lang.Thread



# Virtual Thread on a Carrier



```
public class Main {  
    public static void main(String[] args)  
    {  
        int num = 29;  
        boolean flag = false;  
        Thread.sleep(20);  
        for (int i = 2; i <= num / 2; ++i) {  
            // condition for nonprime number  
            if (num % i == 0) {  
                flag = true;  
                break;  
            }  
        }  
    }  
}
```



```
public class Main {  
    public static void main(String[] args)  
    {  
        int num = 29;  
        boolean flag = false;  
        for (int i = 2; i <= num / 2; ++i) {  
            // condition for nonprime number  
            if (num % i == 0) {  
                flag = true;  
                break;  
            }  
        }  
    }  
}
```

Virtual Thread

java.lang.Thread

# Virtual Thread on a Carrier



```
public class Main {  
    public static void main(String[] args)  
    {  
        int num = 29;  
        boolean flag = false;  
        Thread.sleep(20)  
        for (int i = 2; i <= num / 2; ++i) {  
            // condition for nonprime number  
            if (num % i == 0) {  
                flag = true;  
                break;  
            }  
        }  
    }  
}
```

Virtual Thread

java.lang.Thread



# Virtual Thread on a Carrier

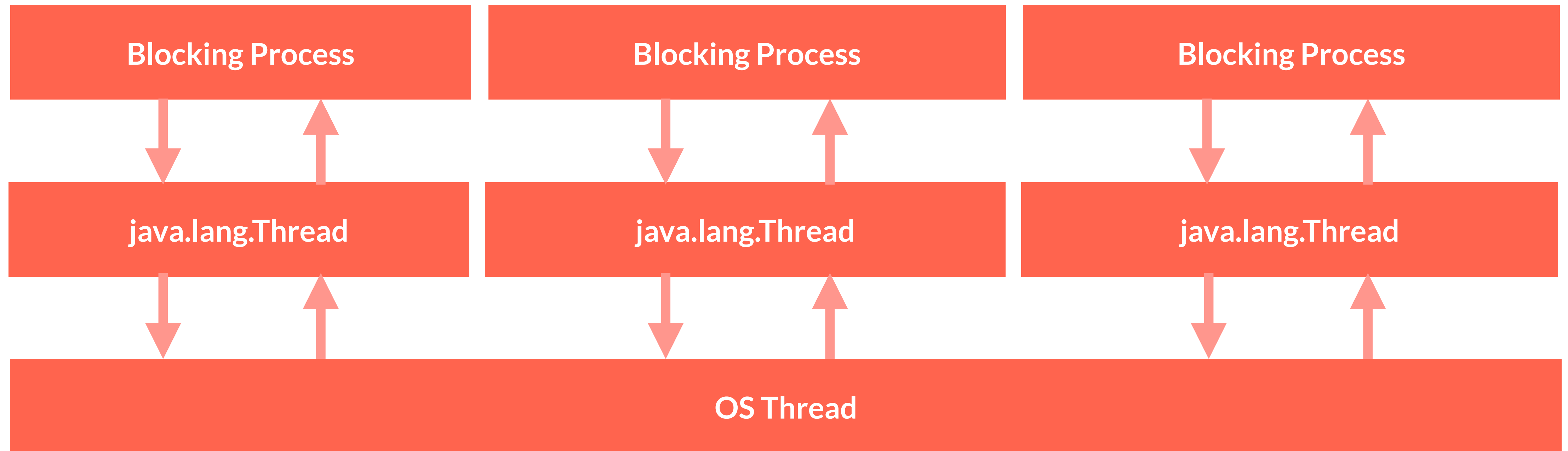


```
public class Main {  
    public static void main(String[] args)  
    {  
        int num = 29;  
        boolean flag = false;  
        Thread.sleep(20)  
        for (int i = 2; i <= num / 2; ++i) {  
            // condition for nonprime number  
            if (num % i == 0) {  
                flag = true;  
                break;  
            }  
        }  
    }  
}
```

Virtual Thread

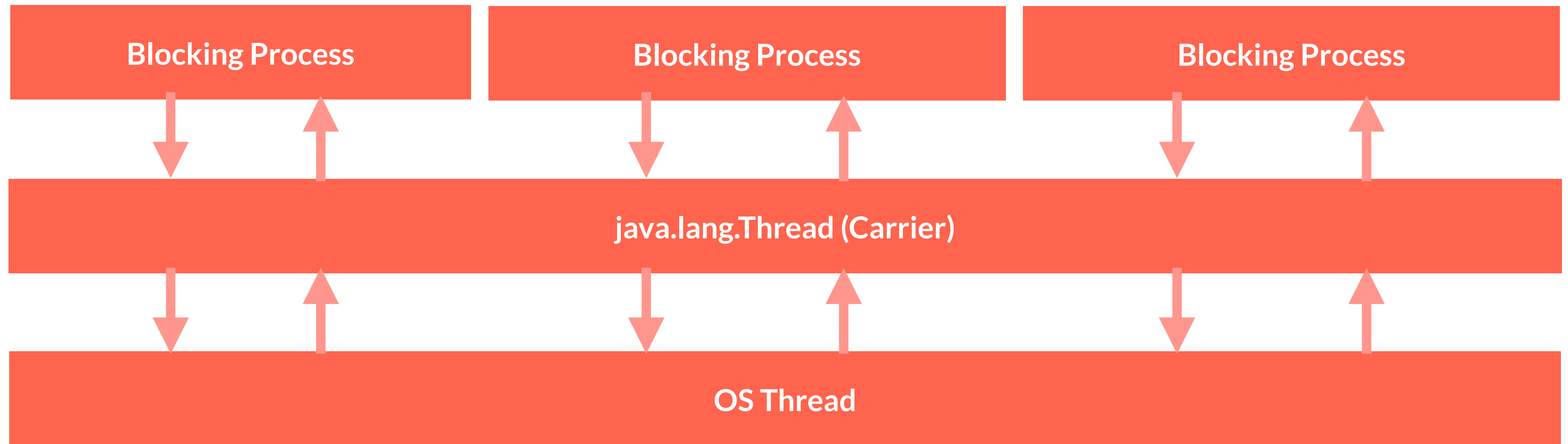
java.lang.Thread

# The way it works now (Pre-Loom)



Each Thread is typically thrown back into the pool

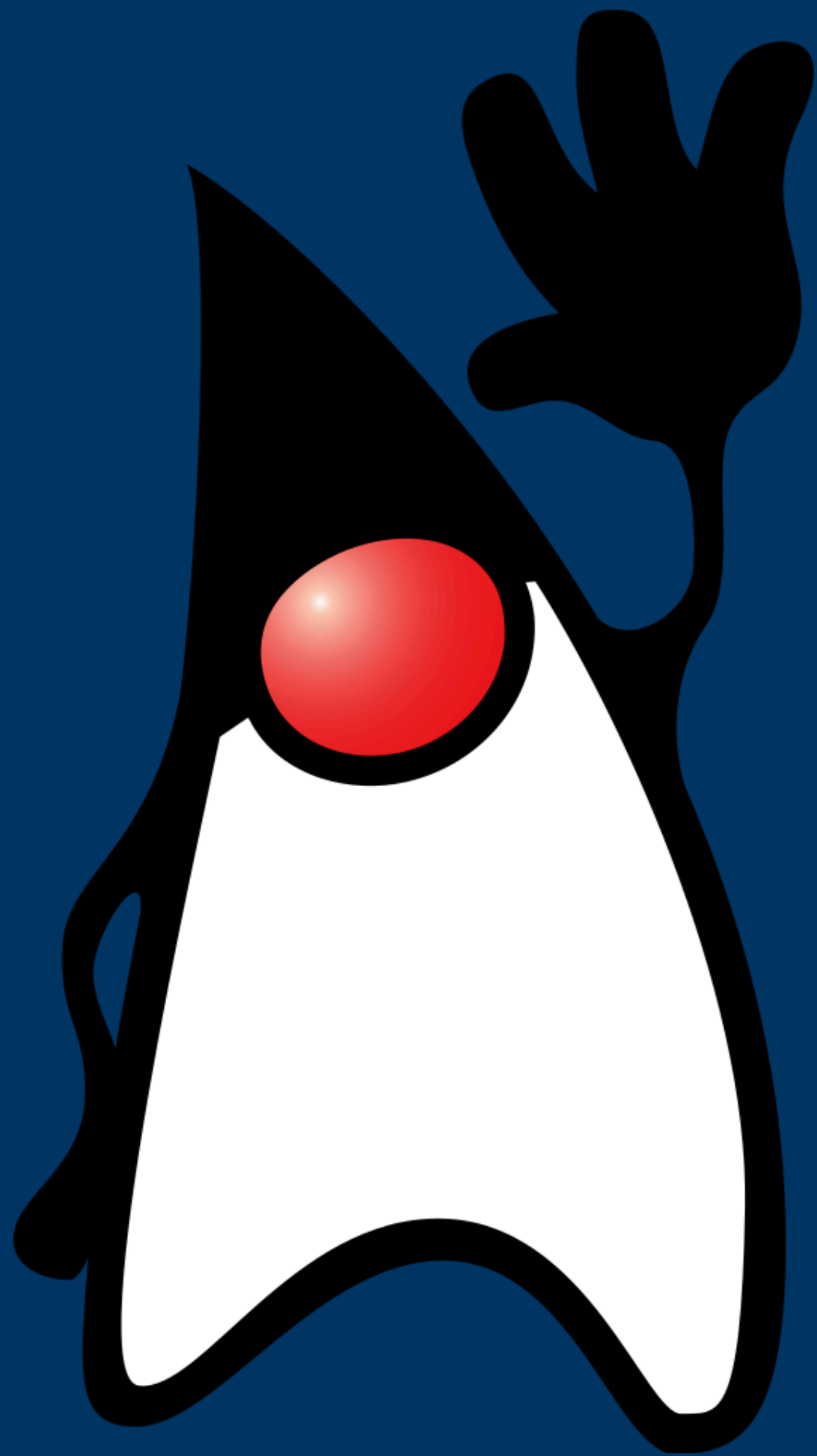
# With Virtual Threading



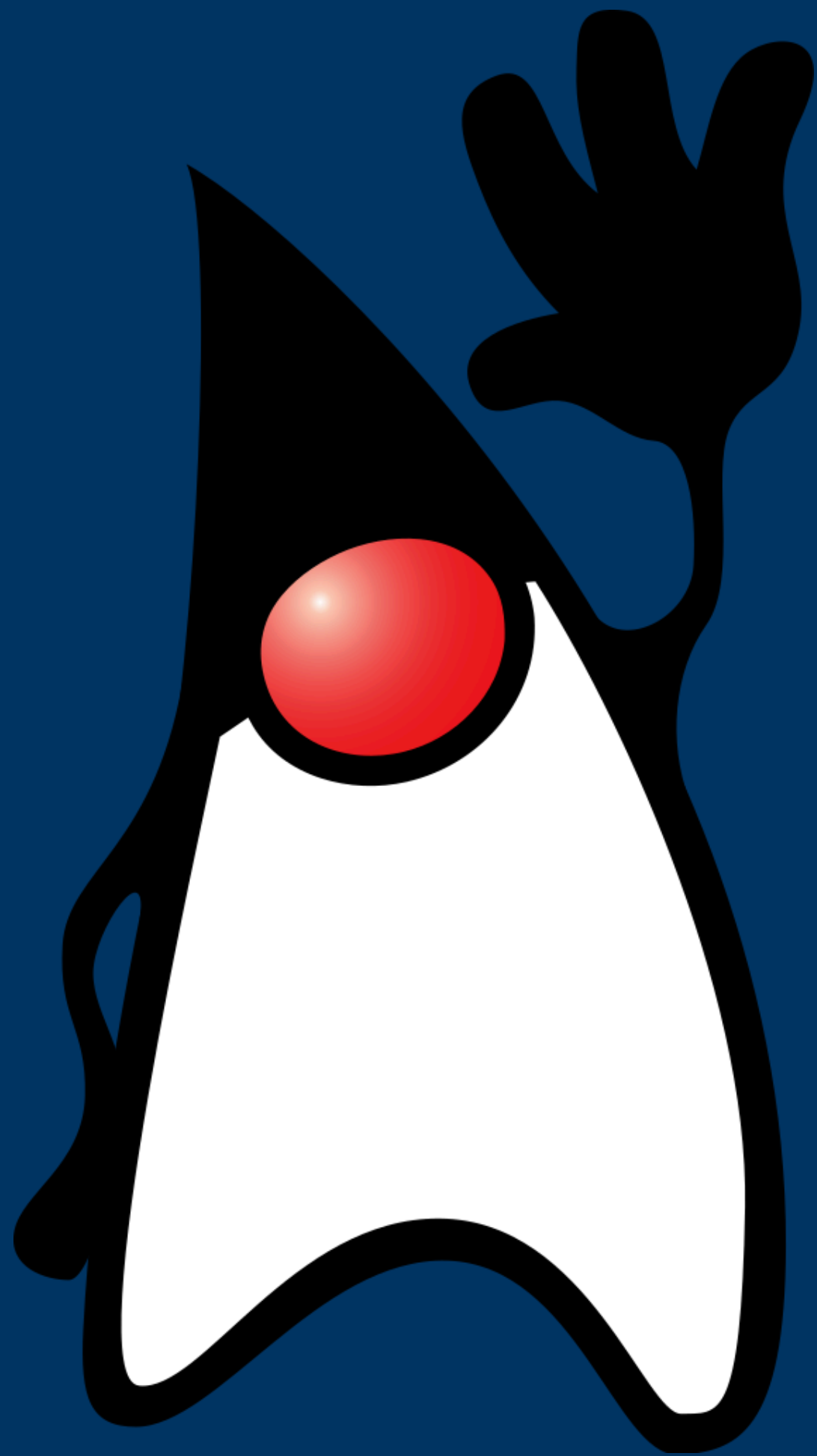




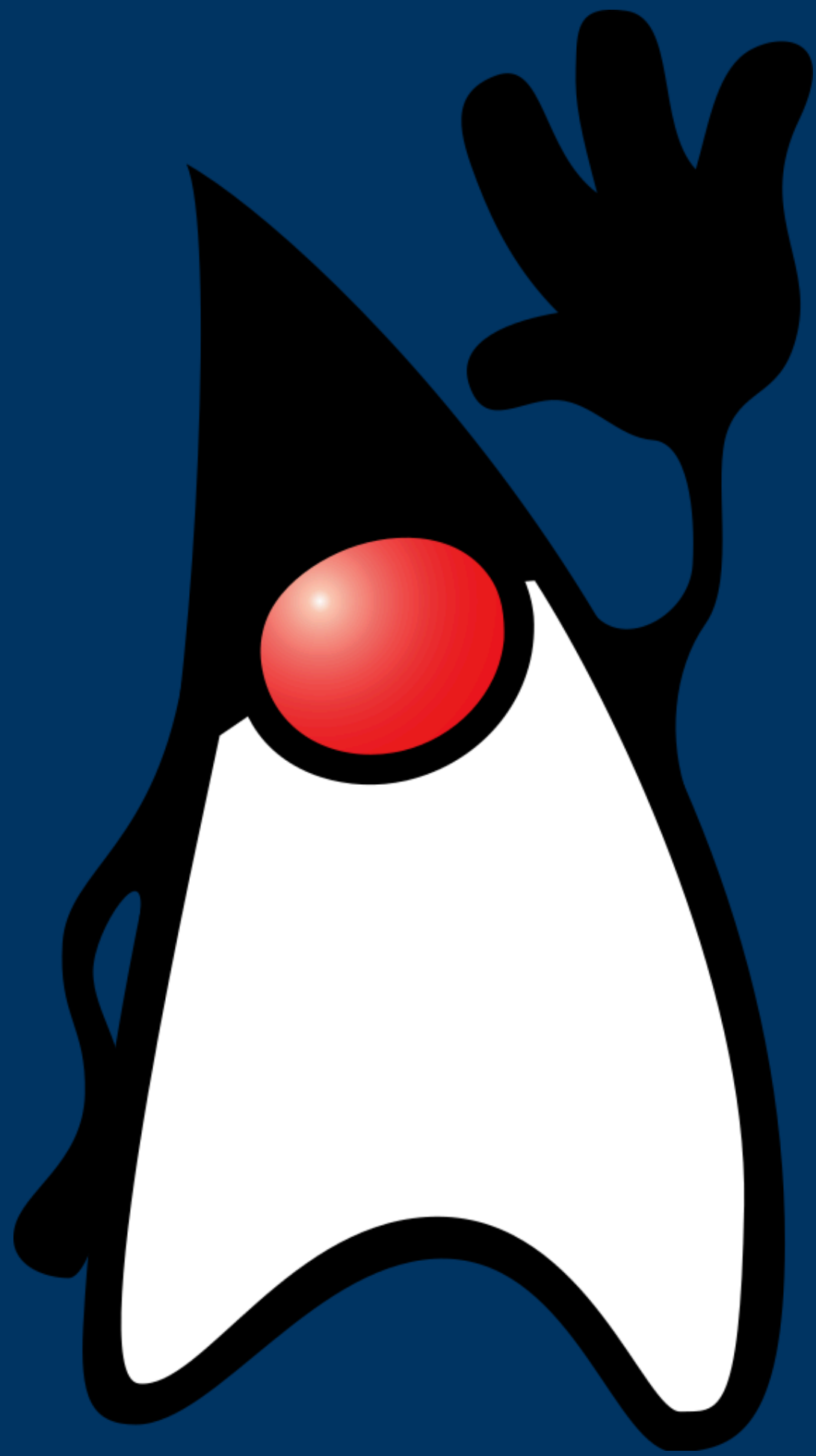
- Create your full task represented by your own Thread
- Add Blocking Code
- Forget about:
  - Thread Pools
  - Reactive Frameworks
- If you want to do more, then make more Threads!
- It is OK to block, just program without consideration to blocking, Loom will handle it for you!



# Demo: Creating Virtual Threads



# Lab: Creating Virtual Threads



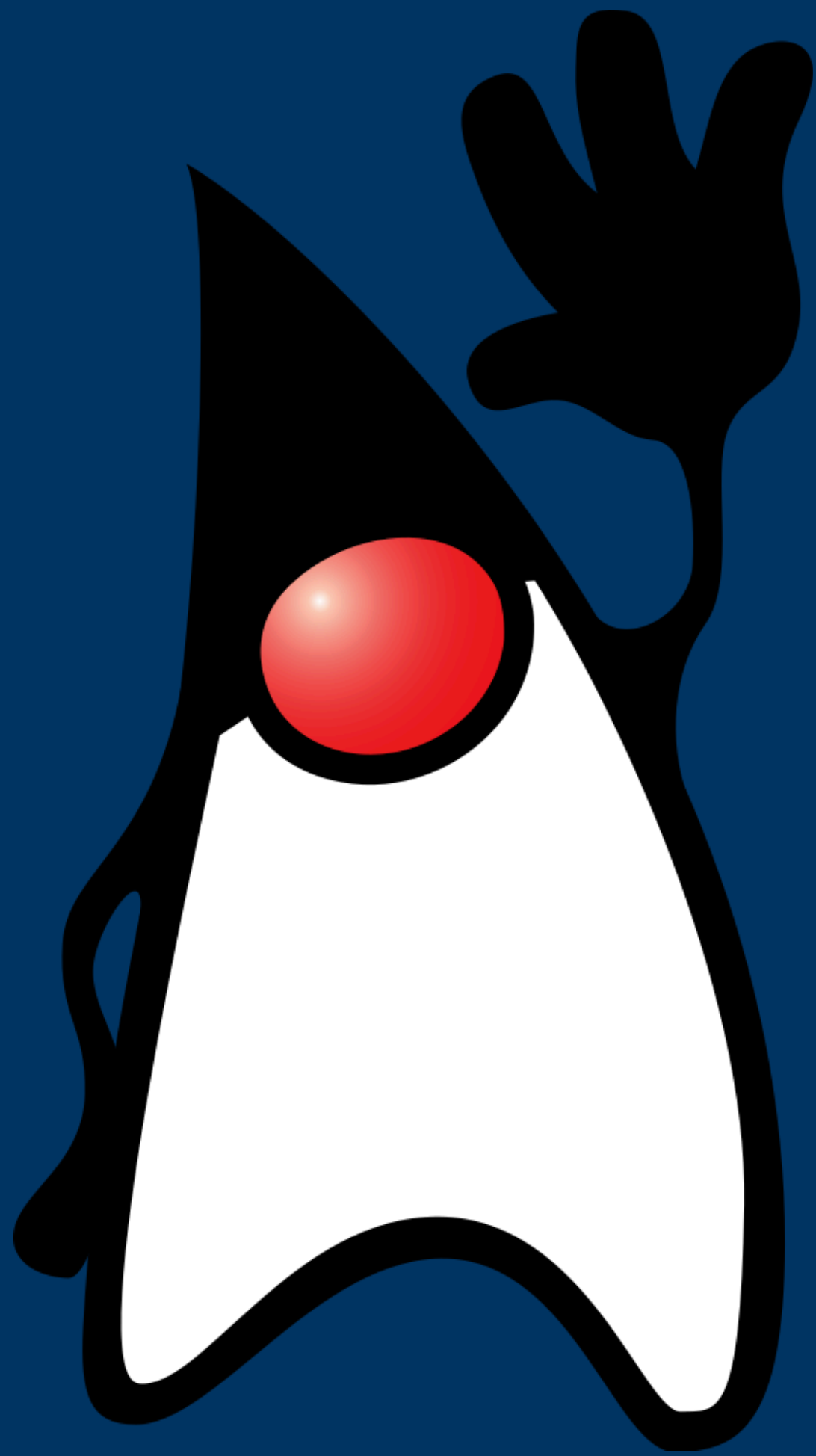
Pinning

# Pinning Defined

- Virtual thread is pinned to its carrier if it is mounted but is in a state in which it cannot be unmounted
- If a virtual thread blocks while pinned, it blocks its carrier.
- This behavior is still correct, but it holds on to a worker thread for the duration that the virtual thread is blocked, making it unavailable for other virtual threads.
- Occasional pinning is not harmful
- Very frequent pinning, however, will harm throughput

# Pinning Situations

- **Synchronized Block**
  - If you have a Thread blocked with synchronized opt for ReentrantLock or StampedLock, which is preferred anyway for better performance
- **JNI (Java Native Interface)**
  - When there is a native frame on the stack — when Java code calls into native code (JNI) that then calls back into Java
  - JNI Pinning will **never** go away, it may also be given up on in the GA



# Structured Concurrency



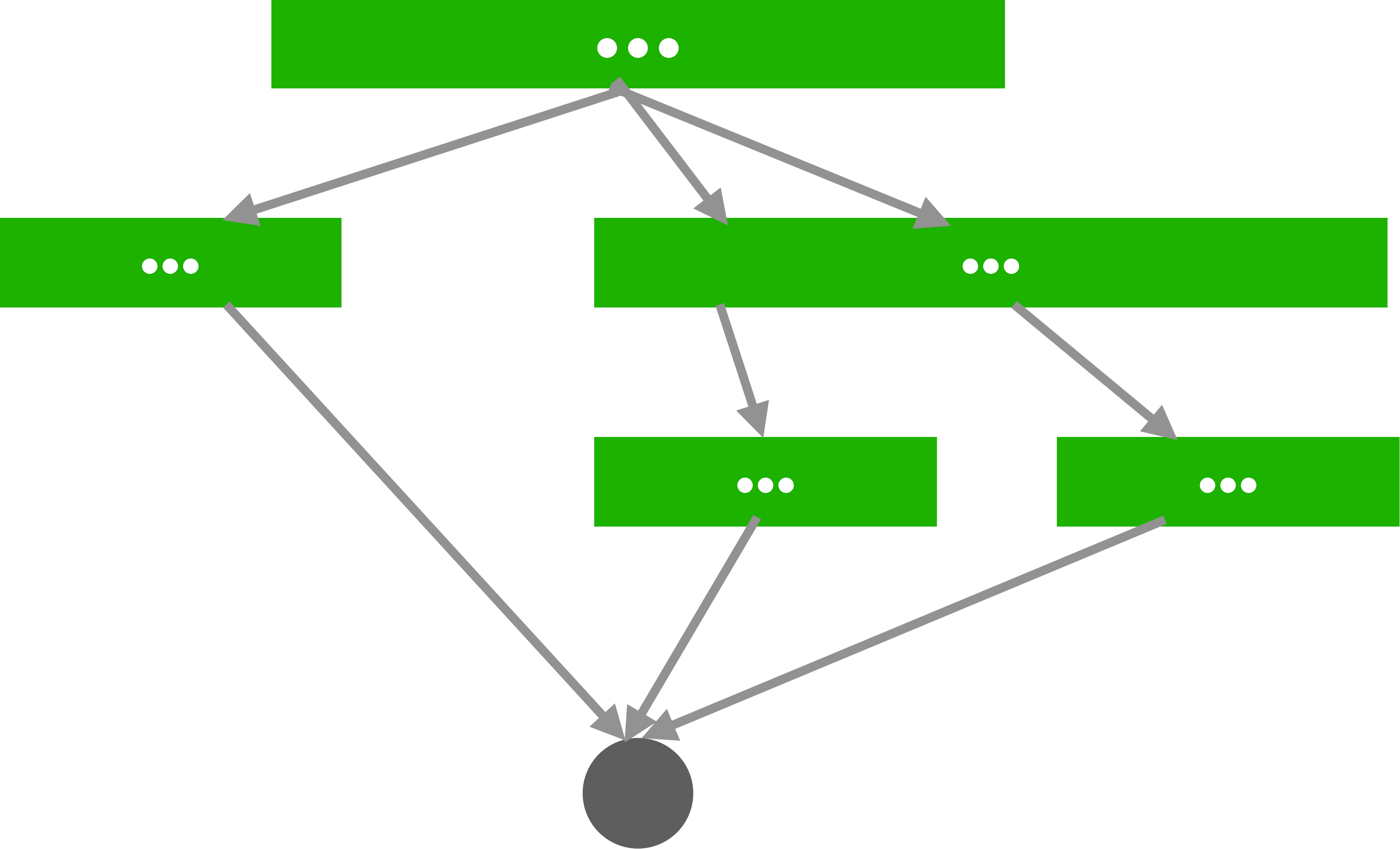
# Structured Concurrency

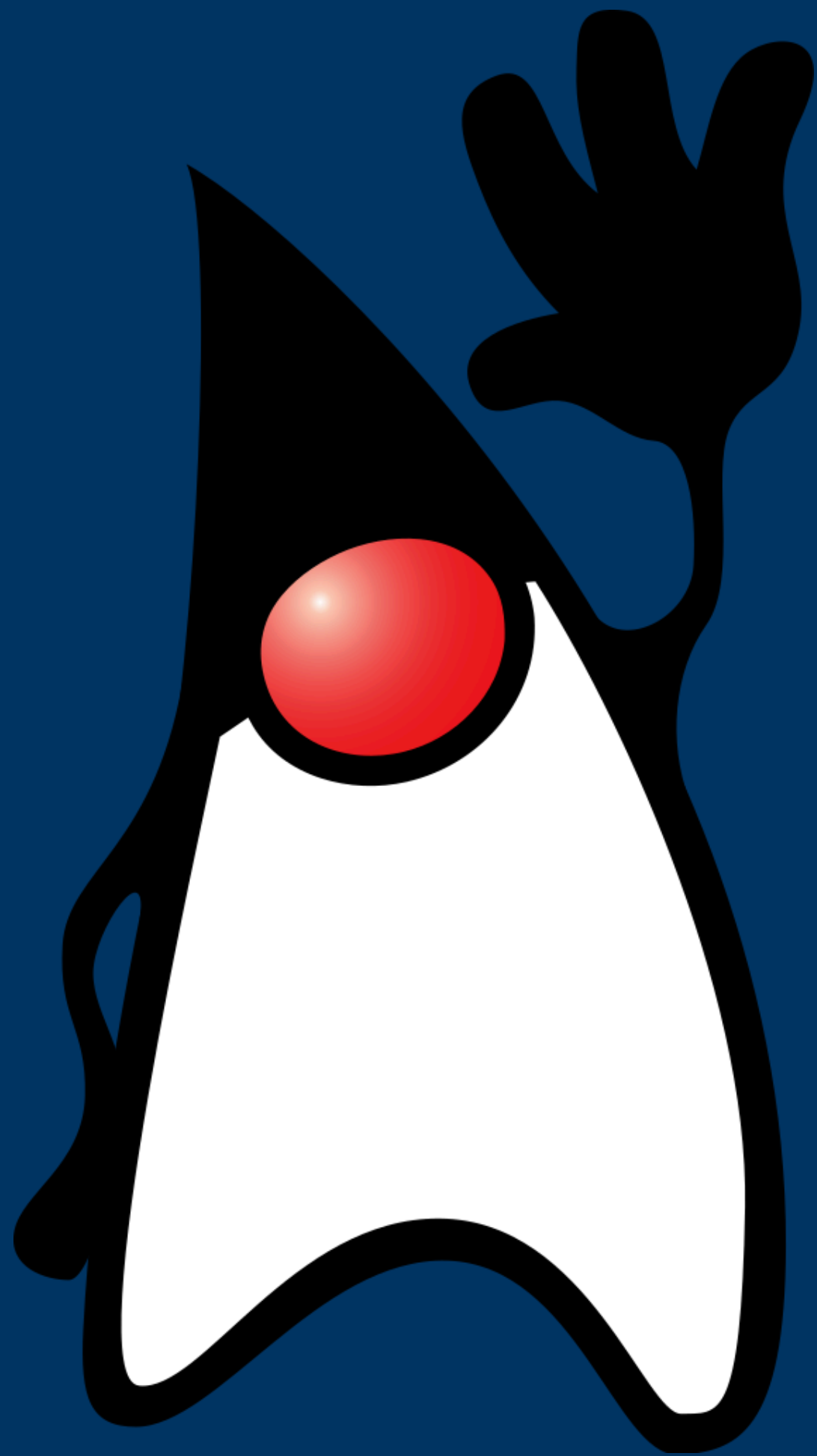
- "Launching a task in a new thread is really no better than programming with GOTO"
- Structured concurrency corrals thread lifetimes into code blocks.
- Similar to how structured programming confines control flow of sequential execution into a well-defined code block, structured concurrency does the same with concurrent control flow.
- Principle:
  - Threads that are created in some code unit must all terminate by the time we exit that code unit
  - If execution splits to multiple thread inside some scope, it must join before exiting the scope.



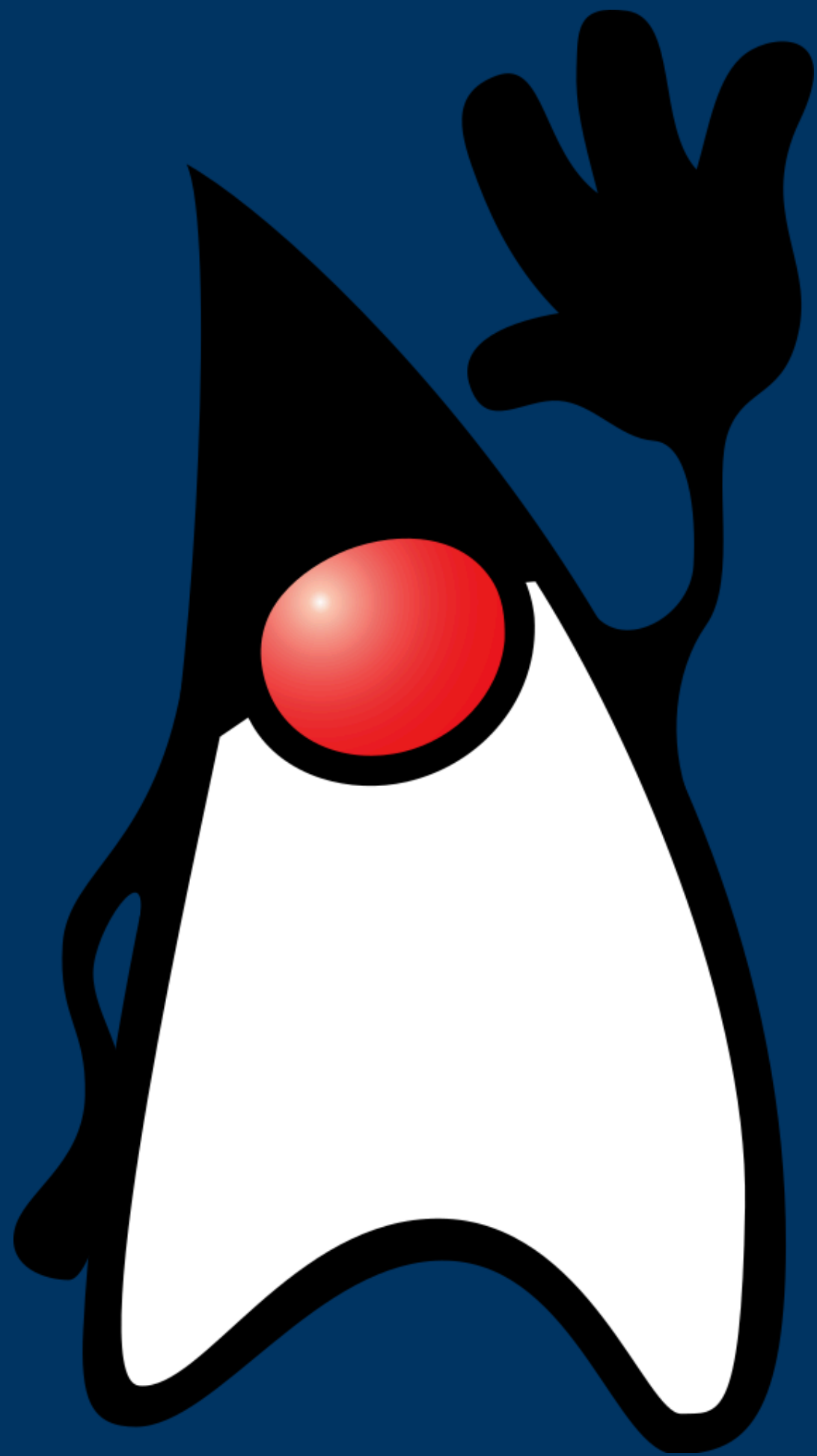
# Executors is now AutoCloseable

- `java.util.concurrent.ExecutorService` is an `AutoCloseable`, with `close` shutting down the service and awaiting termination.
- Therefore you can place it within a try with resources (TWR) block
- This guarantees that all tasks submitted to the service will have terminated by the time we exit the try-with-resources (TWR) block
- This also means we can confine their lifetime to the code structure
- Before we exit the TWR block, the current thread will block, waiting for all tasks — and their threads — to finish.
- Once outside it, we are guaranteed that the tasks have terminated.

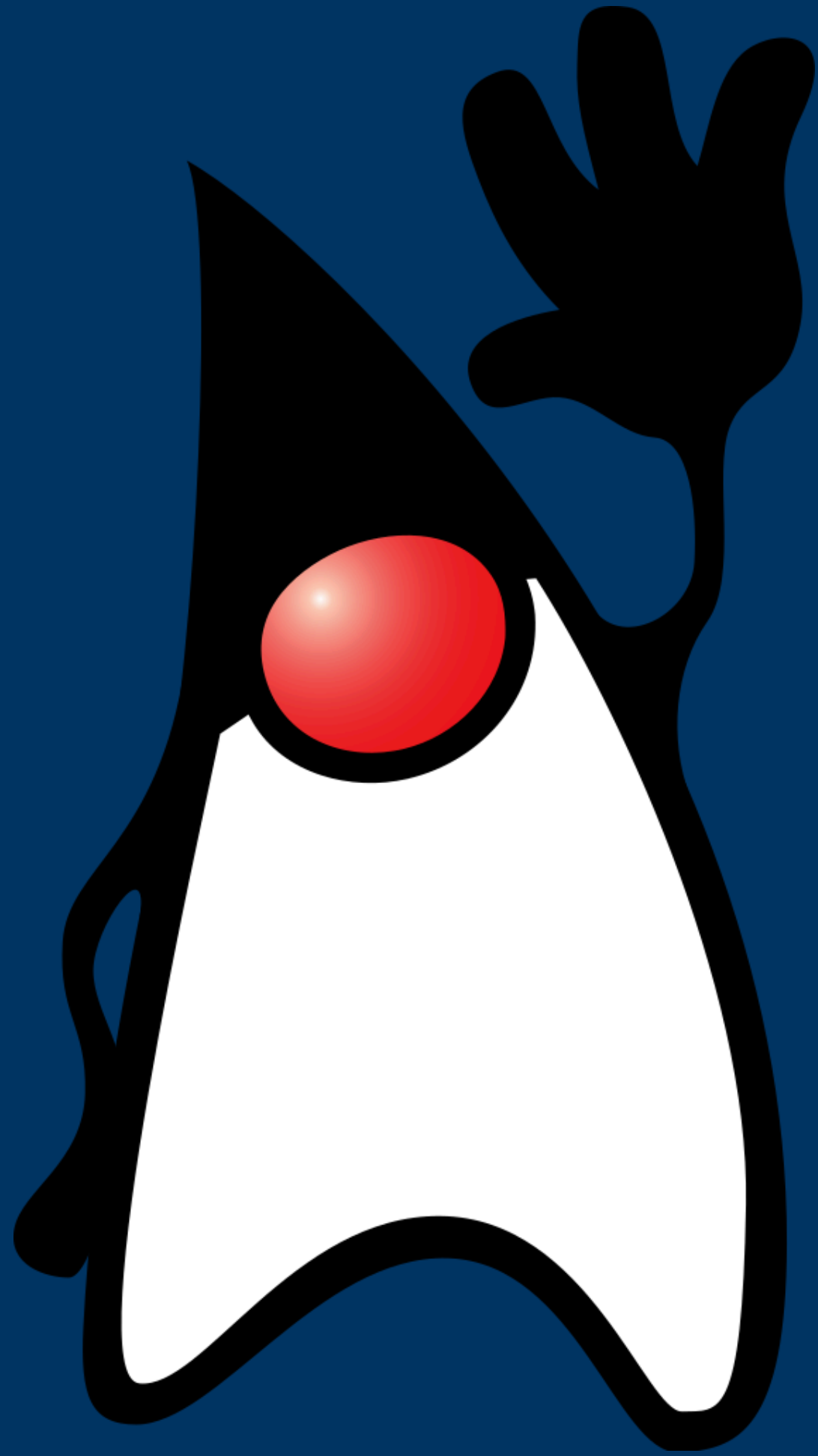




# Demo: Structured Concurrency



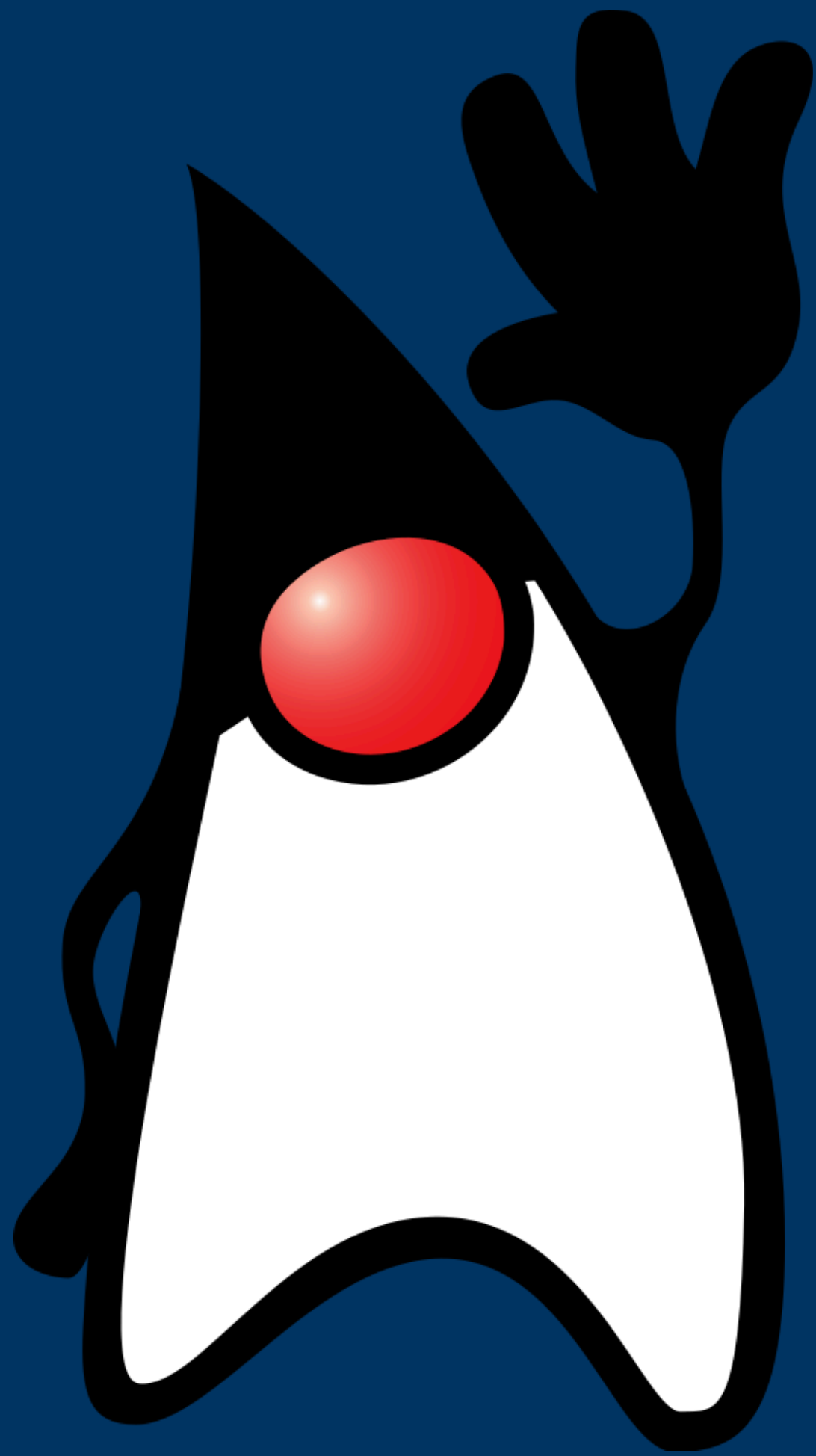
# Lab: Structured Concurrency



# Retrofitted Sockets

# Reimplemented Java IO

- JEP353 Reimplement Legacy Socket API
  - PlainSocketImpl replaced by NIOSocketImpl
  - <https://openjdk.java.net/jeps/353>
- JEP373 Reimplement Legacy Datagram Socket API
  - <https://openjdk.java.net/jeps/373>



# Debugging and Profiling

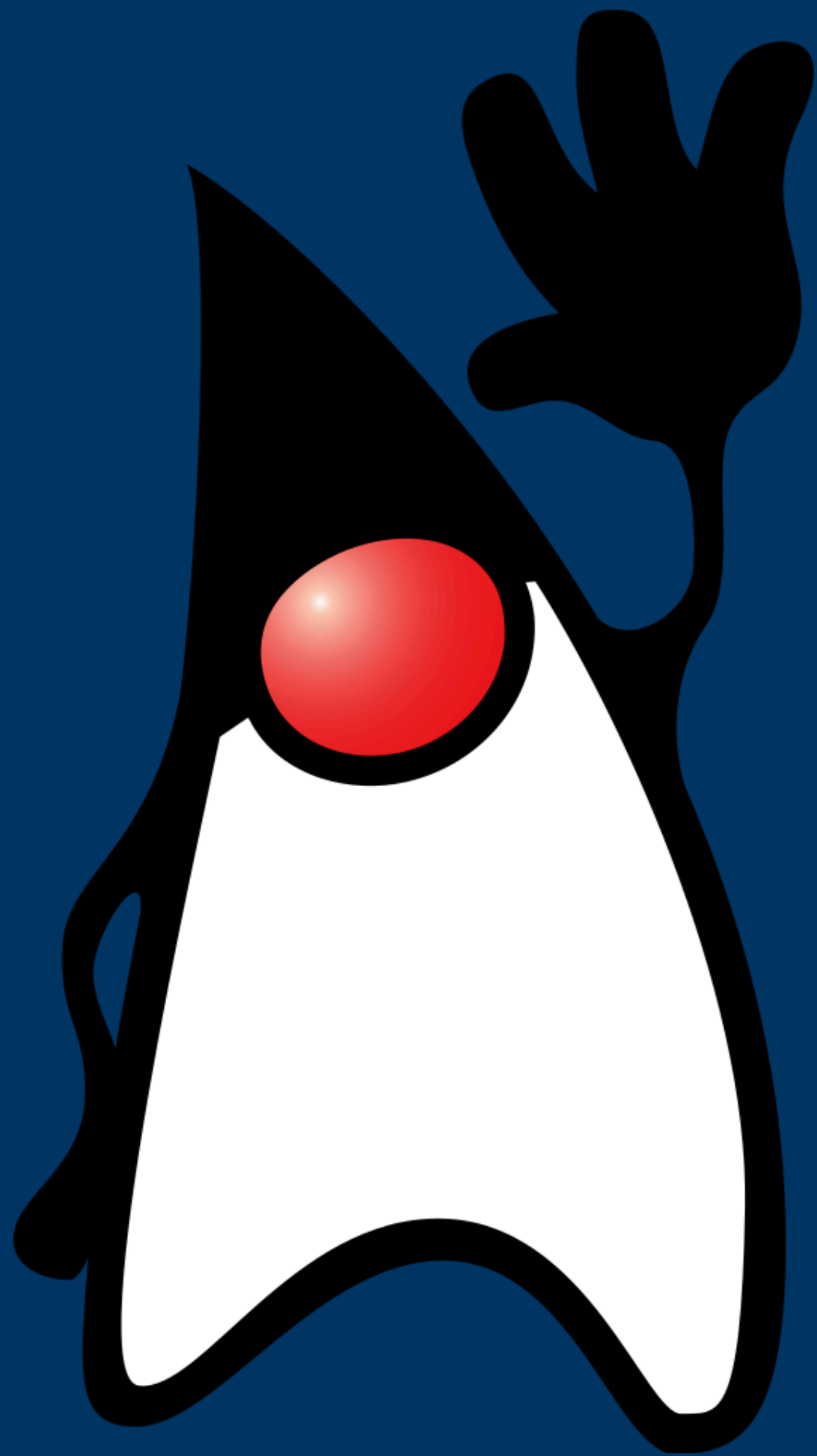
# Debugging

- Java Debugger Wire Protocol (JDWP) and the Java Debugger Interface (JDI) used by Java debuggers and supports ordinary debugging operations such as breakpoints, single stepping, variable inspection etc., works for virtual threads as it does for classical threads
- EA, not all debugger operations are supported for virtual threads. Some operations pose special challenges, since there can be a million virtual threads, it would take special consideration how to handle that



# Profiling

- It has been proven difficult to profile asynchronous code, particularly since a Thread can be phased out.
- Now that all code in a continuation and a continuations can be members of an overarching context, we can collate all subtasks and maintain information better
- Java Flight Recorder the foundation of profiling and structured logging in the JDK — to support virtual threads.
- Blocked virtual threads can be shown in the profiler and time spent on I/O measured and accounted for.



Any Questions?