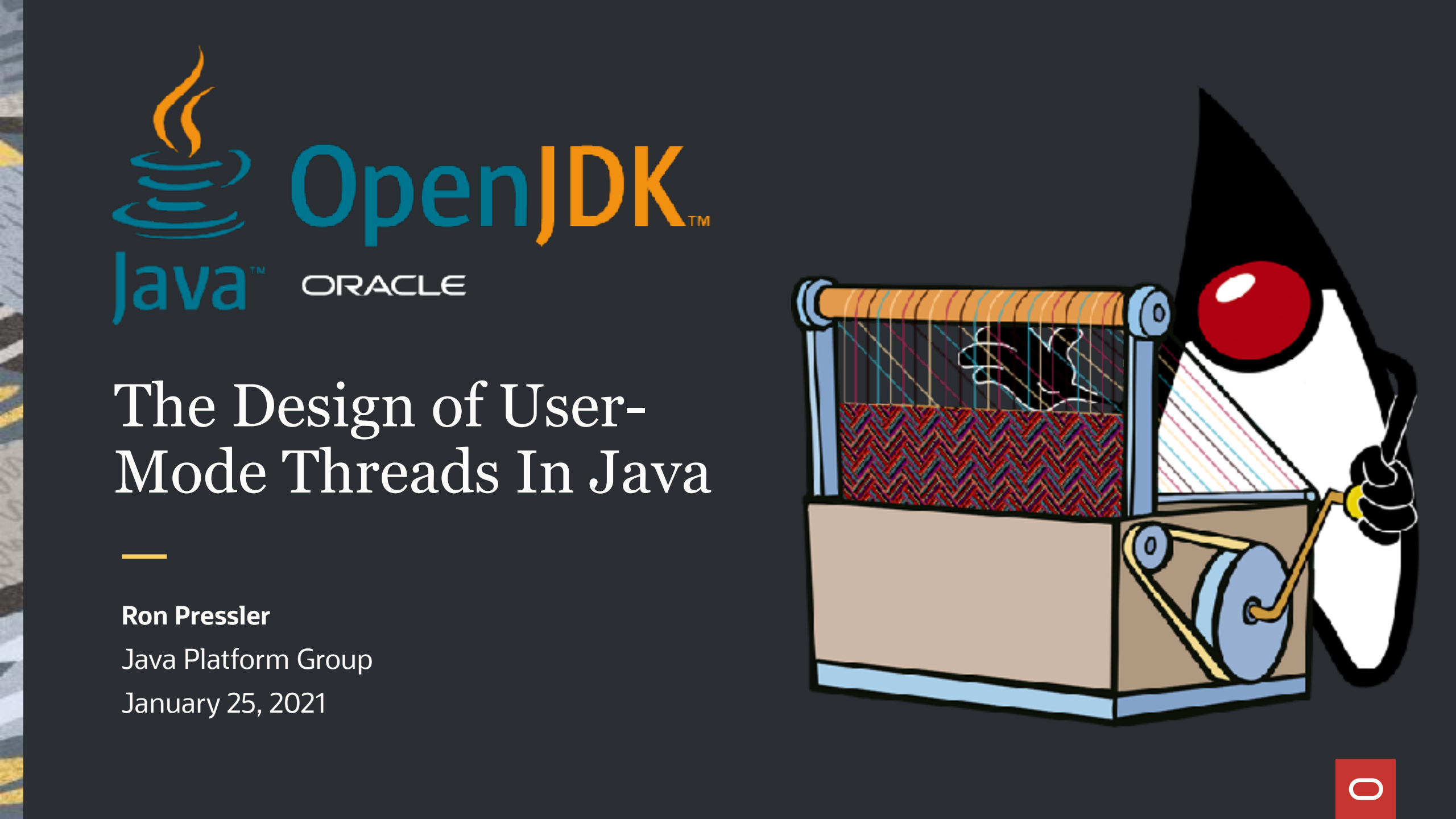


The Design of User-Mode Threads In Java

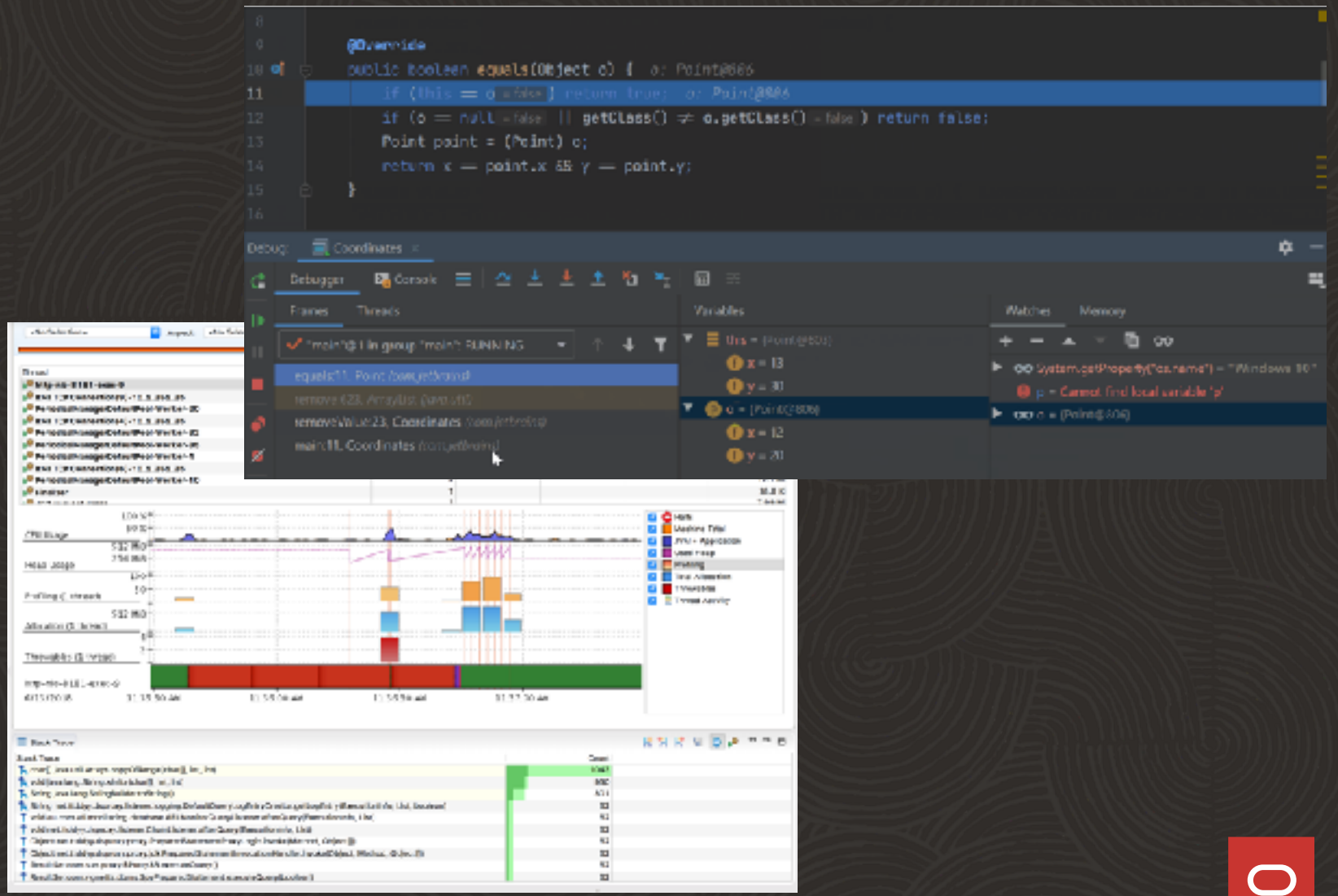
Ron Pressler

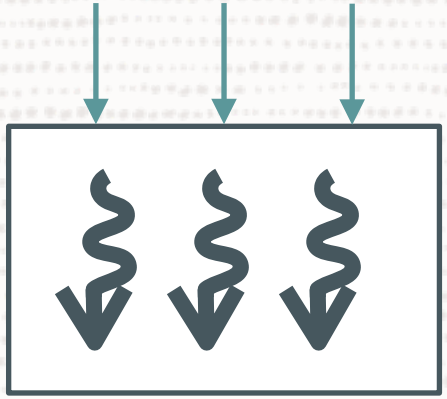
Java Platform Group

January 25, 2021

Java Is Made of Threads

- Exceptions
- Thread Locals
- Debugger
- Profiler (JFR)





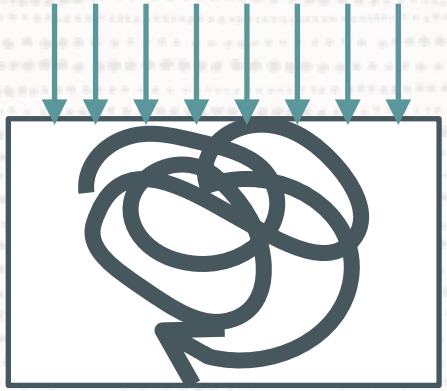
simple
less scalable

SYNC

Programmer 😀

OS / Hardware ☹️

OR



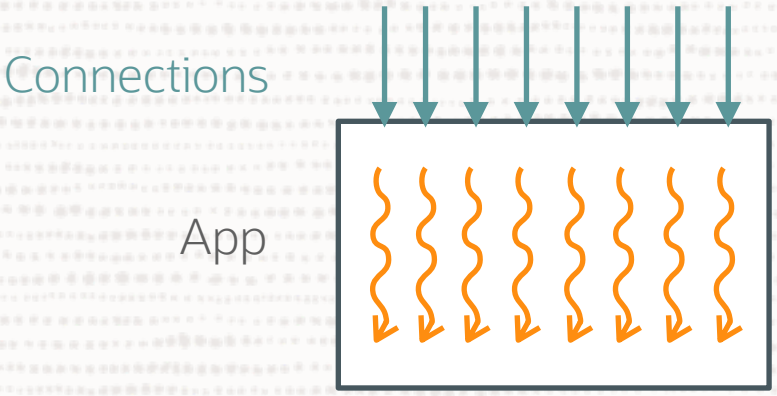
scalable,
complex,
non-interoperable,
hard to debug/profile

ASYNC

Programmer ☹️

OS / Hardware 😀

Codes Like Sync, Scales Like Async



Programmer 😀

OS / Hardware 😀

"We must carefully balance **conservation** and **innovation**"

— Mark Reinhold

- **Forward Compatibility**: we want existing code to enjoy new functionality
- We want to **correct past mistakes** and **start afresh**

"The solutions of **yesterday** are the problems of **today**"

— Brian Goetz

Threads *in Java*

- The use of `Thread.currentThread()` and `ThreadLocal` is pervasive. Without support, or with changed behaviour, little existing code would run.
- Other parts are superseded by new APIs since Java 5 so their datedness/clunkiness is mostly hidden/ignored.

Threads *in Java*

- `java.lang.Thread`
- The Java runtime is well positioned to implement threads.
- Resizable stacks (possible b/c we only need to support Java).
- Context-switching in user-mode.
- Pluggable schedulers, default optimised for transaction processing.

Threads *in Java*

When code in a virtual thread calls an I/O method in the JDK,
suspend the virtual thread,
start a non-blocking I/O operation in the OS,
the scheduler schedules another virtual thread,
when I/O completes re-submit waiting thread to scheduler.

Module java.base
Package java.util.concurrent

Class ConcurrentHashMap<K,V>

java.lang.Object
  java.util.AbstractMap<K,V>
    java.util.concurrent.ConcurrentHashMap<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Implemented Interfaces:

Serializable, ConcurrentMap<K,V>, Map<K,V>

```
public class ConcurrentHashMap<K,V>
extends AbstractMap<K,V>
implements ConcurrentMap<K,V>, Serializable
```

A hash table supporting full concurrency of retrievals and high expected concurrency for updates. This class obeys the same functional specification as Hashtable, and includes versions of methods corresponding to each method of Hashtable. However, even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access. This class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization

Module java.base
Package java.util.concurrent.locks

Class ReentrantLock

java.lang.Object
  java.util.concurrent.locks.ReentrantLock

All Implemented Interfaces:

Serializable, Lock

```
public class ReentrantLock
extends Object
implements Lock, Serializable
```

A reentrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.

A ReentrantLock is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking lock will return, successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock. This can be checked using methods isHeldByCurrentThread(), and getHoldCount().

The constructor for this class accepts an optional fairness parameter. When set true, under contention, locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order. Programs using fair locks accessed by many threads may display lower overall throughput (i.e., are slower;

Module java.base
Package java.nio.channels

Class SocketChannel

java.lang.Object
  java.nio.channels.spi.AbstractInterruptibleChannel
    java.nio.channels.SelectableChannel
      java.nio.channels.spi.AbstractSelectableChannel
        java.nio.channels.SocketChannel

All Implemented Interfaces:

Closeable, AutoCloseable, ByteChannel, Channel, GatheringByteChannel, InterruptibleChannel, NetworkChannel, ReadableByteChannel, ScatteringByteChannel, WritableByteChannel

---

```
public abstract class SocketChannel
extends AbstractSelectableChannel
implements ByteChannel, ScatteringByteChannel, GatheringByteChannel, NetworkChannel
```

A selectable channel for stream-oriented connecting sockets.

A socket channel is created by invoking one of the open methods of this class. It is not possible to create a channel for an arbitrary, pre-existing socket. A newly-created socket channel is open but not yet connected. An attempt to invoke an I/O operation upon an unconnected channel will cause a NotYetConnectedException to be thrown. A socket channel can be connected by invoking its connect method; once connected, a socket channel remains connected until it is closed. Whether or not a socket channel is connected may be determined by invoking its

Module java.base
Package java.io

Class InputStream

java.lang.Object
  java.io.InputStream

All Implemented Interfaces:

Closeable, AutoCloseable

Direct Known Subclasses:

AudioInputStream, ByteArrayInputStream, FileInputStream, FilterInputStream, ObjectInputStream, PipedInputStream, SequenceInputStream, StringBufferInputStream

---

```
public abstract class InputStream
extends Object
implements Closeable
```

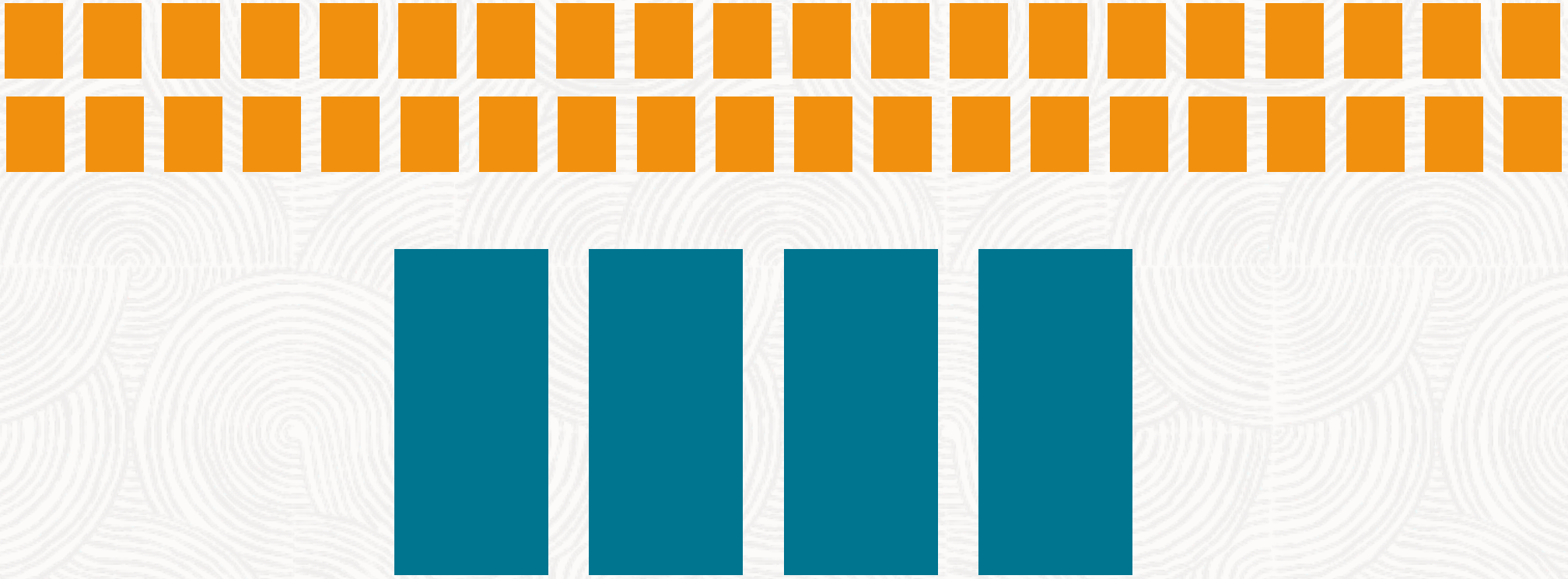
This abstract class is the superclass of all classes representing an input stream of bytes.

Applications that need to define a subclass of InputStream must always provide a method that returns the next byte of input.

Since:

1.0

virtual threads



“carrier” platform threads managed by a scheduler

async/await

C#
JavaScript
Kotlin
C++/Rust

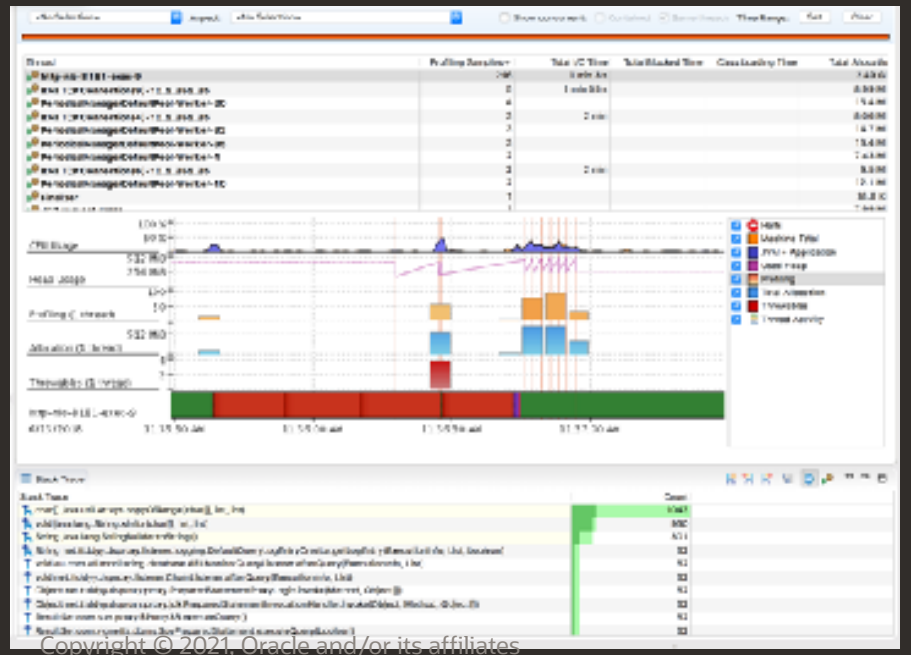
User-Mode Threads

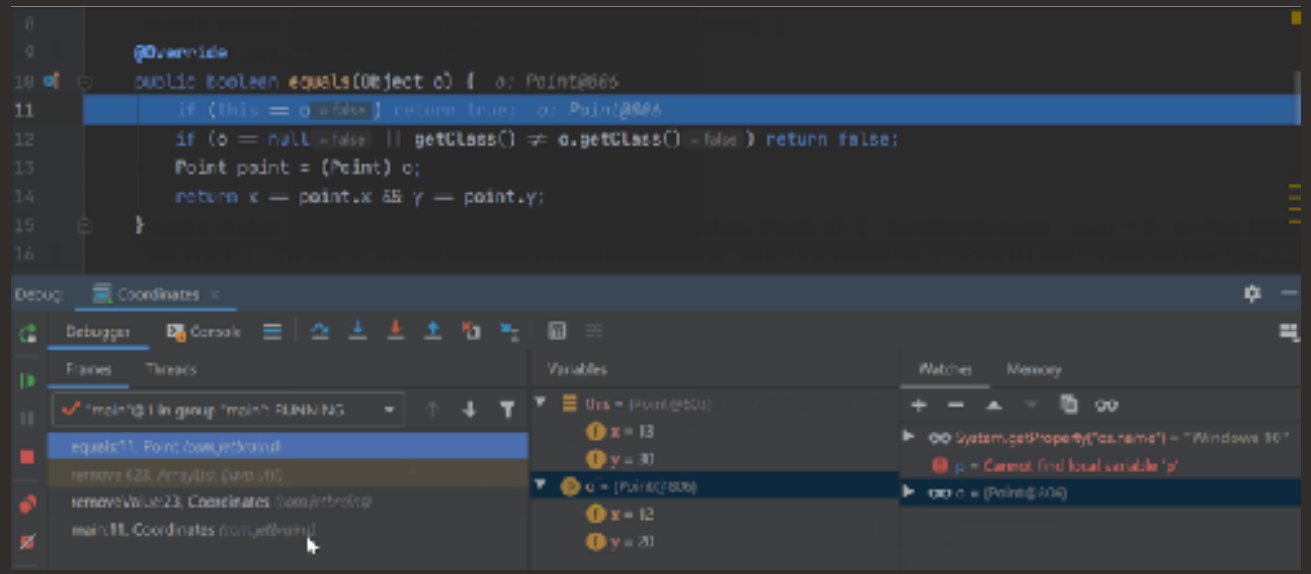
Erlang
Go
Java

Process: Unit of Concurrency

E.g. a transaction

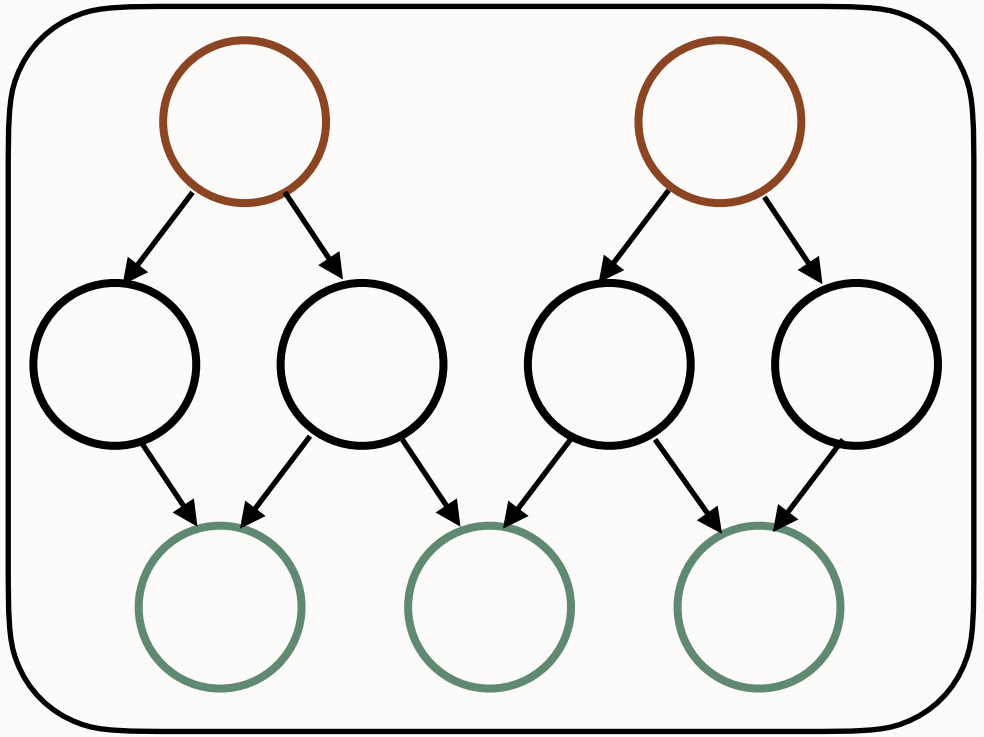
- Code (writing/reading)
- Troubleshooting: stack traces, debugger single-stepping
- Profiling





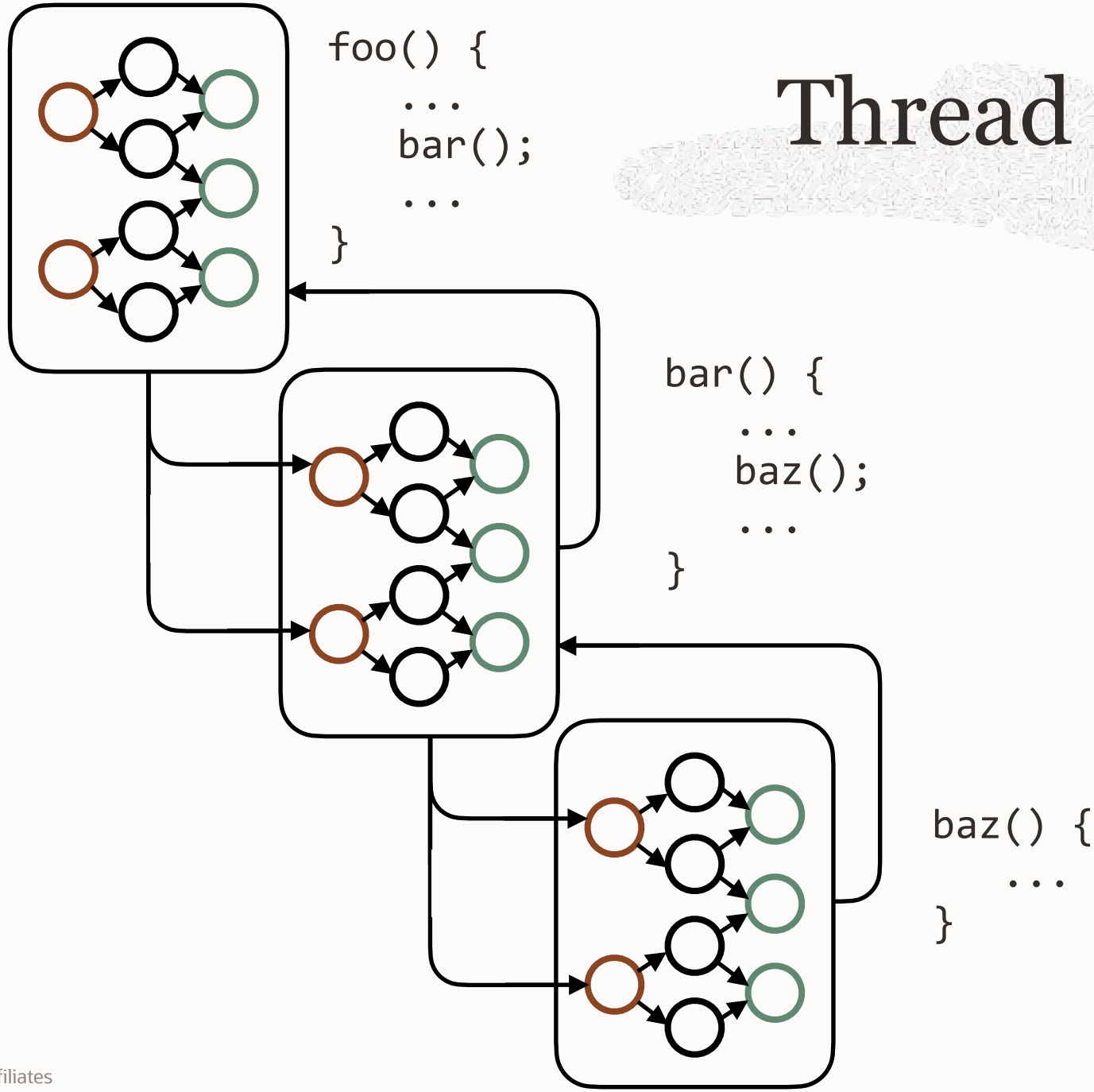
Process

```
a;b;c;d = (a;b);(c;d)
```



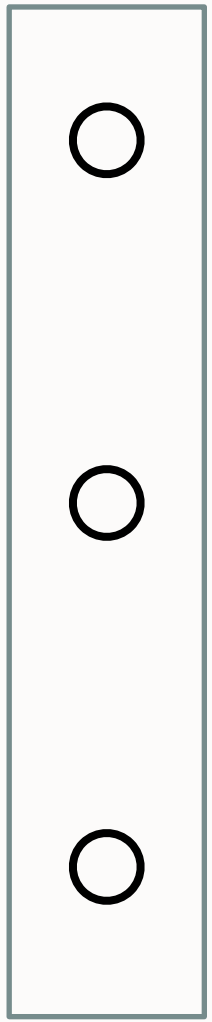
(Nondeterminism https://youtu.be/9vupFNsND6o)

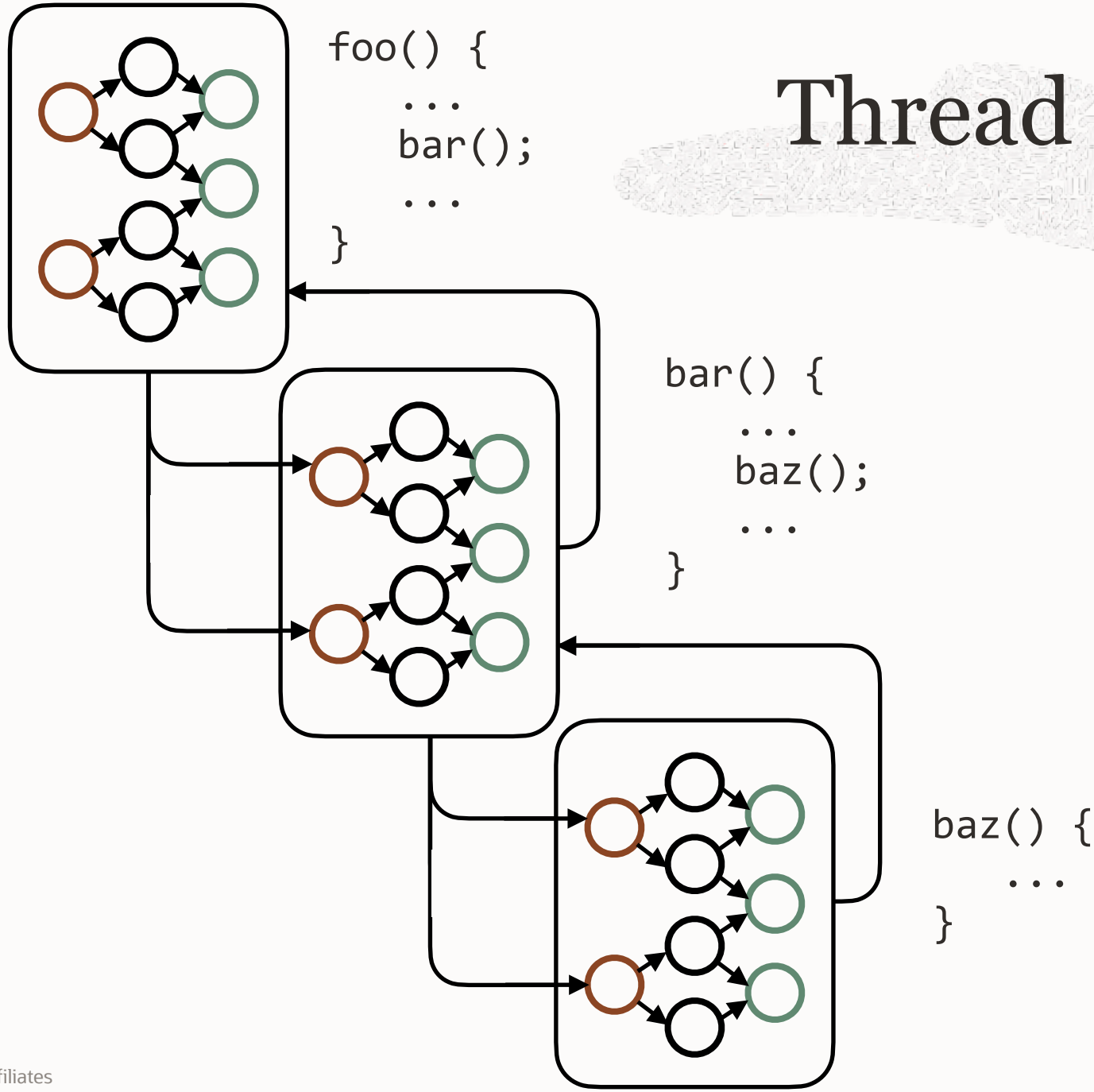
Thread



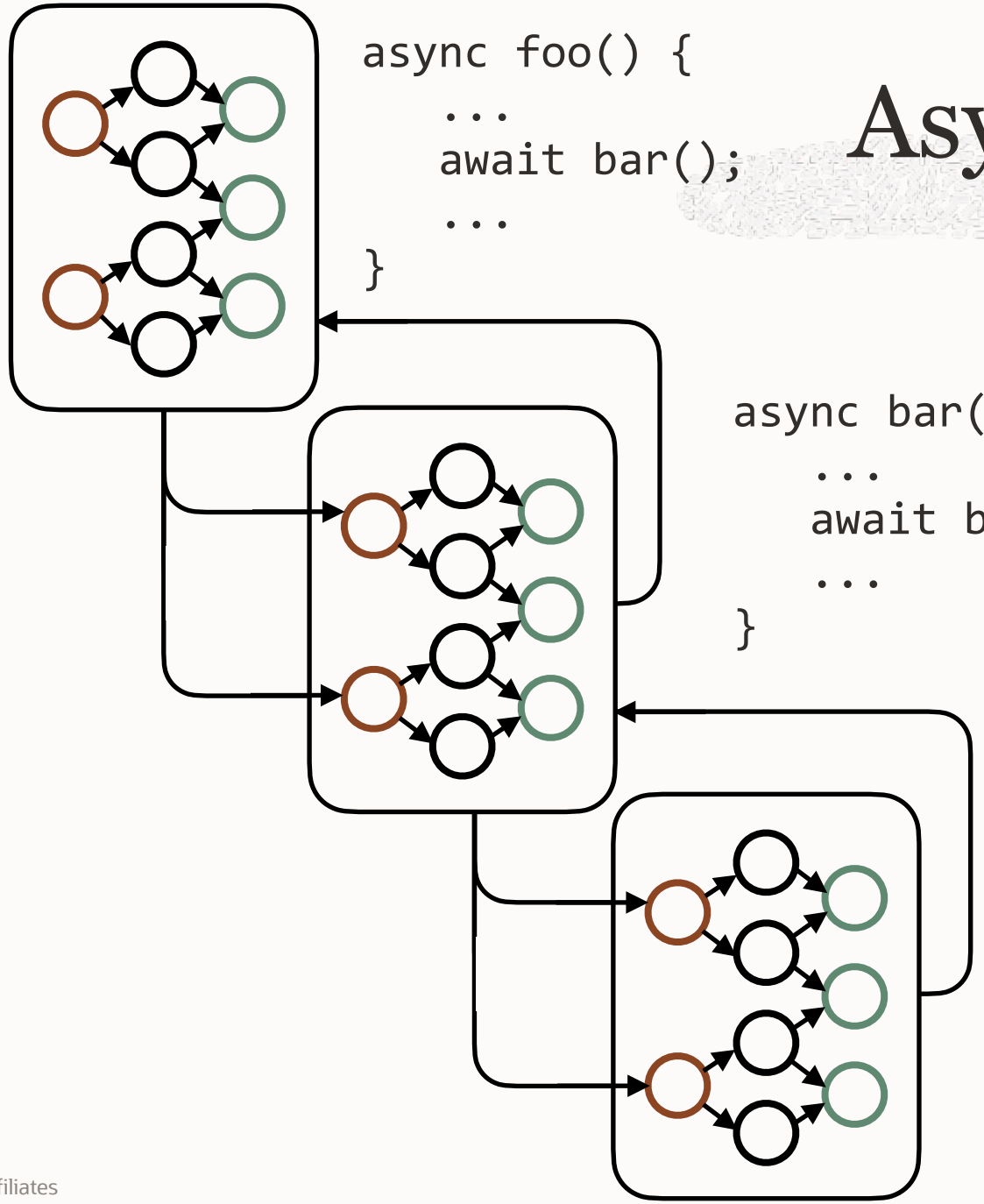
Thread

Call Stack





Async/Await



Thread vs. Async/Await

Scheduling/interleaving points

Thread: Everywhere *except* where explicitly forbidden (with a CS)

async/await: Nowhere *except* where explicitly allowed (with `await`)

Thread vs. Async/Await

Scheduling/interleaving points

Thread: Everywhere *except* where explicitly forbidden (with a CS)

async/await: Nowhere *except* where explicitly allowed (with `await`)

JavaScript

Thread vs. Async/Await

Implementation

Thread: Requires integrating with the implementation of subroutines (control over backend)

async/await: Can be implemented in the compiler frontend

Thread vs. Async/Await

Implementation

Thread: Requires integrating with the implementation of subroutines (control over backend)

async/await: Can be implemented in the compiler frontend

Kotlin

Thread vs. Async/Await

Recursion & virtual calls

Thread: Yes (requires ~~large~~/resizable stacks)

async/await: Can be excluded

Thread vs. Async/Await

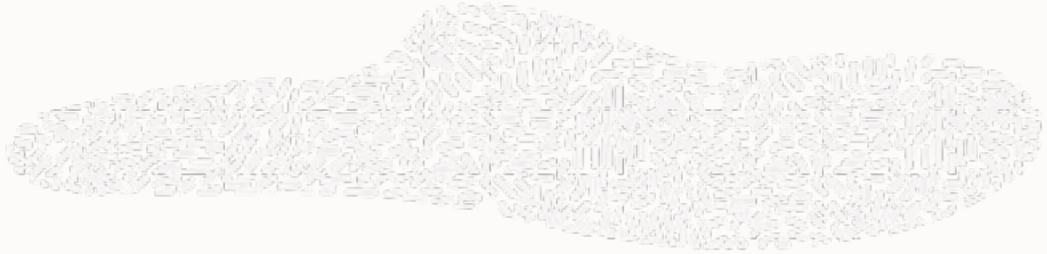
Recursion & virtual calls

Thread: Yes (requires ~~large~~/resizable stacks)

async/await: Can be excluded

C++/Rust

Resizable Stack



- Transparent allocation
- Efficient allocation
- No internal pointers/tracked pointers (no FFI)

Performance

Latency — How long an operation takes (*s*)

Throughput — How many operations complete per time unit (*ops/s*)

Impact — How much a metric would improve with full optimisation (%)

Syntactic Concurrency: Generators et al.

- Updating simulation entities in a frame
- Generators (two processes with an unbuffered channel)

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1, -1, -1):
        yield my_str[i]

for char in rev_str("hello"):
    print(char)
```


Context-Switching Impact: Generators

- Impact: 100%
- Best case latency: ~0ns (monomorphic, fits in cache)

Concurrency: Transactions

$$L = \lambda W$$

https://inside.java/2020/08/07/loom-performance/

Throughput: $\lambda = L/W$

Context-switch impact on throughput: t/μ

t — Mean context-switch latency

μ — Mean wait (I/O) latency

Context-Switching Impact: Transactions

- Impact: low if blocking for external events
- Best case latency: 60ns (polymorphic, doesn't fit in cache) (1.5% impact)
- Target latency for ≤5% impact: ≤200ns

29

Conclusion

- Control over backend
- Rare I/O in FFI
- No internal pointers/pointers tracked
- Efficient and transparent stack resizing
- Threads already in the platform, libraries and tooling
- async/await can be implemented on top by languages that want it

async/await

C#
JavaScript
Kotlin
C++/Rust

User-Mode Threads

Erlang
Go
Java

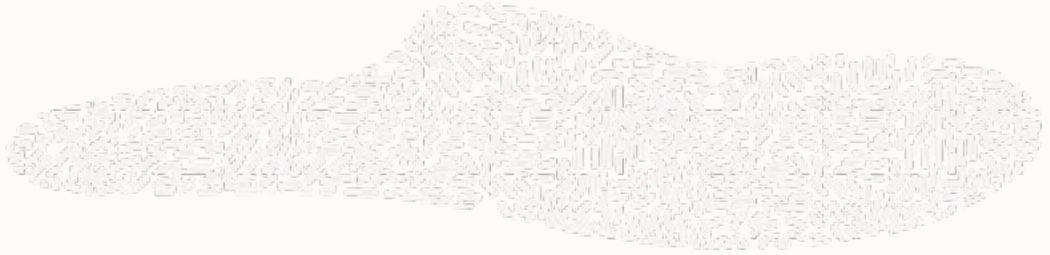
Zig

Continuations

Continuation

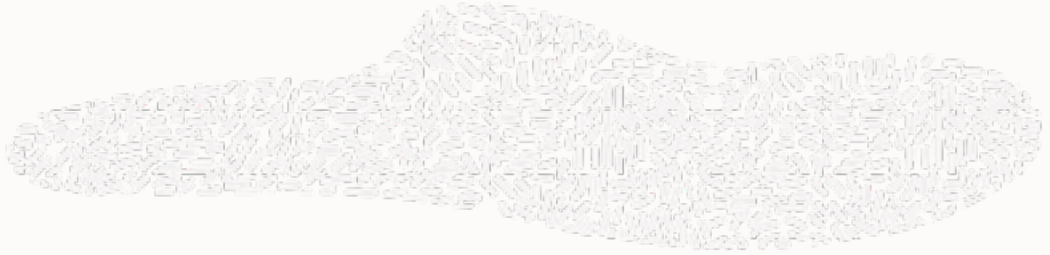
- Delimited
- Stackful
- Asymmetric
- Type `void` (but other types can be trivially implemented on top)
- Scoped (aka multi-prompt, aka “effect handlers”)
- One-shot/non-reentrant (future: cloneable & serializable)

Continuations



```
class Continuation implements Runnable {
    public Continuation(ContinuationScope scope, Runnable body);
    public void run();
    public boolean isDone();
    public static void yield(ContinuationScope scope);
}
```


Continuations



```
Continuation cont = new Continuation(SCOPE, () -> {
    System.out.println("before");
    Continuation.yield(SCOPE);
    System.out.println("after");
});

cont.run(); // prints before
cont.isDone(); // false
cont.run(); // prints after
cont.isDone(); // true
```


thread =

continuation

\+

scheduler

36

Thread = Continuation + Scheduler

```
package java.util.concurrent.locks;
public class LockSupport {

    public static void park(...) {
        var t = Thread.currentThread();
        if (t.isVirtual())
            Continuation.yield(FIBER_SCOPE);
        else
            Unsafe.park(false, 0L);
    }

}
```

```
public static void unpark(Thread thread) {
    if (thread.isVirtual()) {
        t.scheduler.submit(t.continuation);
    } else {
        Unsafe.unpark(thread)
    }
}
```

```
var readLock = new java.util.concurrent.locks.ReentrantLock();

readLock.lock();
```

```
java.base/java.lang.Continuation.yield(Continuation.java:396)
java.base/java.lang.Fiber.maybePark(Fiber.java:597)
java.base/java.lang.Fiber.park(Fiber.java:525)
java.base/java.lang.System$2.parkFiber(System.java:2333)
java.base/jdk.internal.misc.Strands.parkFiber(Strands.java:89)
java.base/java.util.concurrent.locks.LockSupport.park(LockSupport.java:235)
java.base/java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:887)
java.base/java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireQueued(AbstractQueuedSynchronizer.java:919)
java.base/java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(AbstractQueuedSynchronizer.java:1242)
java.base/java.util.concurrent.locks.ReentrantLock.lock(ReentrantLock.java:269)
Main.lambda$main$0(Main.java:72)
java.base/java.lang.Fiber.lambda$new$0(Fiber.java:198)
java.base/java.lang.Continuation.enter(Continuation.java:372)
java.base/java.lang.Continuation.run(Continuation.java:328)
java.base/java.lang.Fiber.runContinuation(Fiber.java:366)
java.base/java.util.concurrent.ForkJoinTask$RunnableExecuteAction.exec(ForkJoinTask.java:1425)
java.base/java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:290)
java.base/java.util.concurrent.ForkJoinPool$WorkQueue.topLevelExec(ForkJoinPool.java:1017)
java.base/java.util.concurrent.ForkJoinPool.scan(ForkJoinPool.java:1666)
java.base/java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1599)
java.base/java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:189)
```


```
var readLock = new java.util.concurrent.locks.ReentrantLock();

readLock.lock();
```

```
java.base/java.lang.Continuation.run(Continuation.java:328)
java.base/java.lang.Fiber.runContinuation(Fiber.java:366)
java.base/java.util.concurrent.ForkJoinTask$RunnableExecuteAction.exec(ForkJoinTask.java:1425)
java.base/java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:290)
java.base/java.util.concurrent.ForkJoinPool$WorkQueue.topLevelExec(ForkJoinPool.java:1017)
java.base/java.util.concurrent.ForkJoinPool.scan(ForkJoinPool.java:1666)
java.base/java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1599)
java.base/java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:189)
```

Thread = Continuation + Scheduler

```
package java.util.concurrent.locks;
public class LockSupport {

    public static void park(...) {
        var t = Thread.currentThread();
        if (t.isVirtual())
            Continuation.yield(FIBER_SCOPE);
        else
            Unsafe.park(false, 0L);
    }

}
```

```
public static void unpark(Thread thread) {
    if (thread.isVirtual()) {
        t.scheduler.submit(t.continuation);
    } else {
        Unsafe.unpark(thread)
    }
}
```

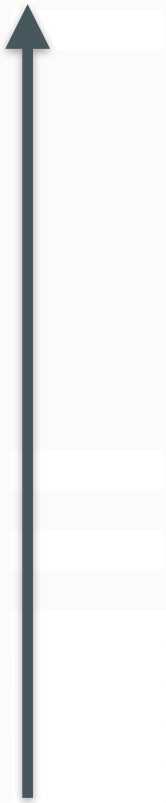
```
var readLock = new java.util.concurrent.locks.ReentrantLock();
readLock.lock();
…
```

```
java.base/java.lang.Continuation.yield(Continuation.java:396)
java.base/java.lang.Fiber.maybePark(Fiber.java:597)
java.base/java.lang.Fiber.park(Fiber.java:525)
java.base/java.lang.System$2.parkFiber(System.java:2333)
java.base/jdk.internal.misc.Strands.parkFiber(Strands.java:89)
java.base/java.util.concurrent.locks.LockSupport.park(LockSupport.java:235)
java.base/java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:887)
java.base/java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireQueued(AbstractQueuedSynchronizer.java:919)
java.base/java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(AbstractQueuedSynchronizer.java:1242)
java.base/java.util.concurrent.locks.ReentrantLock.lock(ReentrantLock.java:269)
Main.lambda$main$0(Main.java:72)
java.base/java.lang.Fiber.lambda$new$0(Fiber.java:198)
java.base/java.lang.Continuation.enter(Continuation.java:372)
java.base/java.lang.Continuation.run(Continuation.java:328)
java.base/java.lang.Fiber.runContinuation(Fiber.java:366)
java.base/java.util.concurrent.ForkJoinTask$RunnableExecuteAction.exec(ForkJoinTask.java:1425)
java.base/java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:290)
java.base/java.util.concurrent.ForkJoinPool$WorkQueue.topLevelExec(ForkJoinPool.java:1017)
java.base/java.util.concurrent.ForkJoinPool.scan(ForkJoinPool.java:1666)
java.base/java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1599)
java.base/java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:189)
```


```
var listener = new java.net.ServerSocket(..);
listener.accept();
```

```
java.base/java.lang.Continuation.yield(Continuation.java:396)
java.base/java.lang.Fiber.maybePark(Fiber.java:597)
java.base/java.lang.Fiber.park(Fiber.java:525)
java.base/java.lang.System$2.parkFiber(System.java:2333)
java.base/jdk.internal.misc.Strands.parkFiber(Strands.java:89)
java.base/sun.nio.ch.NioSocketImpl.park(NioSocketImpl.java:183)
java.base/sun.nio.ch.NioSocketImpl.park(NioSocketImpl.java:212)
java.base/sun.nio.ch.NioSocketImpl.accept(NioSocketImpl.java:752)
java.base/java.net.ServerSocket.implAccept(ServerSocket.java:649)
java.base/java.net.ServerSocket.platformImplAccept(ServerSocket.java:615)
java.base/java.net.ServerSocket.implAccept(ServerSocket.java:591)
java.base/java.net.ServerSocket.implAccept(ServerSocket.java:548)
java.base/java.net.ServerSocket.accept(ServerSocket.java:505)
Main.lambda$main$0(Main.java:80)
java.base/java.lang.Fiber.lambda$new$0(Fiber.java:195)
java.base/java.lang.Continuation.enter(Continuation.java:372)
java.base/java.lang.Continuation.run(Continuation.java:328)
java.base/java.lang.Fiber.runContinuation(Fiber.java:366)
java.base/java.util.concurrent.ForkJoinTask$RunnableExecuteAction.exec(ForkJoinTask.java:1425)
java.base/java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:290)
java.base/java.util.concurrent.ForkJoinPool$WorkQueue.topLevelExec(ForkJoinPool.java:1017)
java.base/java.util.concurrent.ForkJoinPool.scan(ForkJoinPool.java:1666)
java.base/java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1599)
java.base/java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:189)
```

```
var listener = new java.net.ServerSocket(..);
listener.accept();
```



```
java.base/java.lang.Continuation.run(Continuation.java:328)
java.base/java.lang.Fiber.runContinuation(Fiber.java:366)
java.base/java.util.concurrent.ForkJoinTask$RunnableExecuteAction.exec(ForkJoinTask.java:1425)
java.base/java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:290)
java.base/java.util.concurrent.ForkJoinPool$WorkQueue.topLevelExec(ForkJoinPool.java:1017)
java.base/java.util.concurrent.ForkJoinPool.scan(ForkJoinPool.java:1666)
java.base/java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1599)
java.base/java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:189)
```

Continuation: Limitations

- Can’t yield with native frames on continuation stack (rare)
- Can’t yield while holding a monitor (temporary)

- Virtual threads block their carrier when their continuation is pinned to preserve thread semantics.

```
class Continuation implements Runnable {
  public Continuation(ContinuationScope scope, Runnable body);

  public final void run();
  public final boolean isDone();
  public static void yield(ContinuationScope scope);

  protected static Continuation currentContinuation(ContinuationScope
  scope);

  public StackWalker stackWalker();

  protected void onPinned(Reason reason)
       { throw new IllegalStateException("Pinned: " + reason); }
}
```

```
new Continuation (A, () -> {
    System.out.println(“A begin");
    new Continuation(B, () -> {
      System.out.println("B begin");
      new Continuation(C, () -> {
          System.out.println("C begin");
          Continuation.yield(B);
          System.out.println("C end");
        }).run();
      System.out.println("B end");
    }).run();
    System.out.println("A end");
}).run();
```

```
A begin
B begin
C begin
A end
```

Implementation

- Continuation stacks are stored on the heap
- Not GC roots
- Copying to/from stack is *very* cheap (same no. of cache misses as an sp switch)
- Lazy-copying to/from stack with a return barrier (freeze/thaw)
- Finding references is *very* expensive

Implementation

- Stack portions are copied into “stack chunk” objects without parsing.
- GC parses stack and finds oopmaps
- Chunks are mutable while they’re in a non-barrier region (young-gen and other special cases, depending on GC)
- Once a chunk object is promoted, frames can be thawed (removed) but not frozen (added). A new chunk is allocated on next freeze.
- Read-only chunks are trimmed.

Thank you


ORACLE