



OpenJDK™

ORACLE

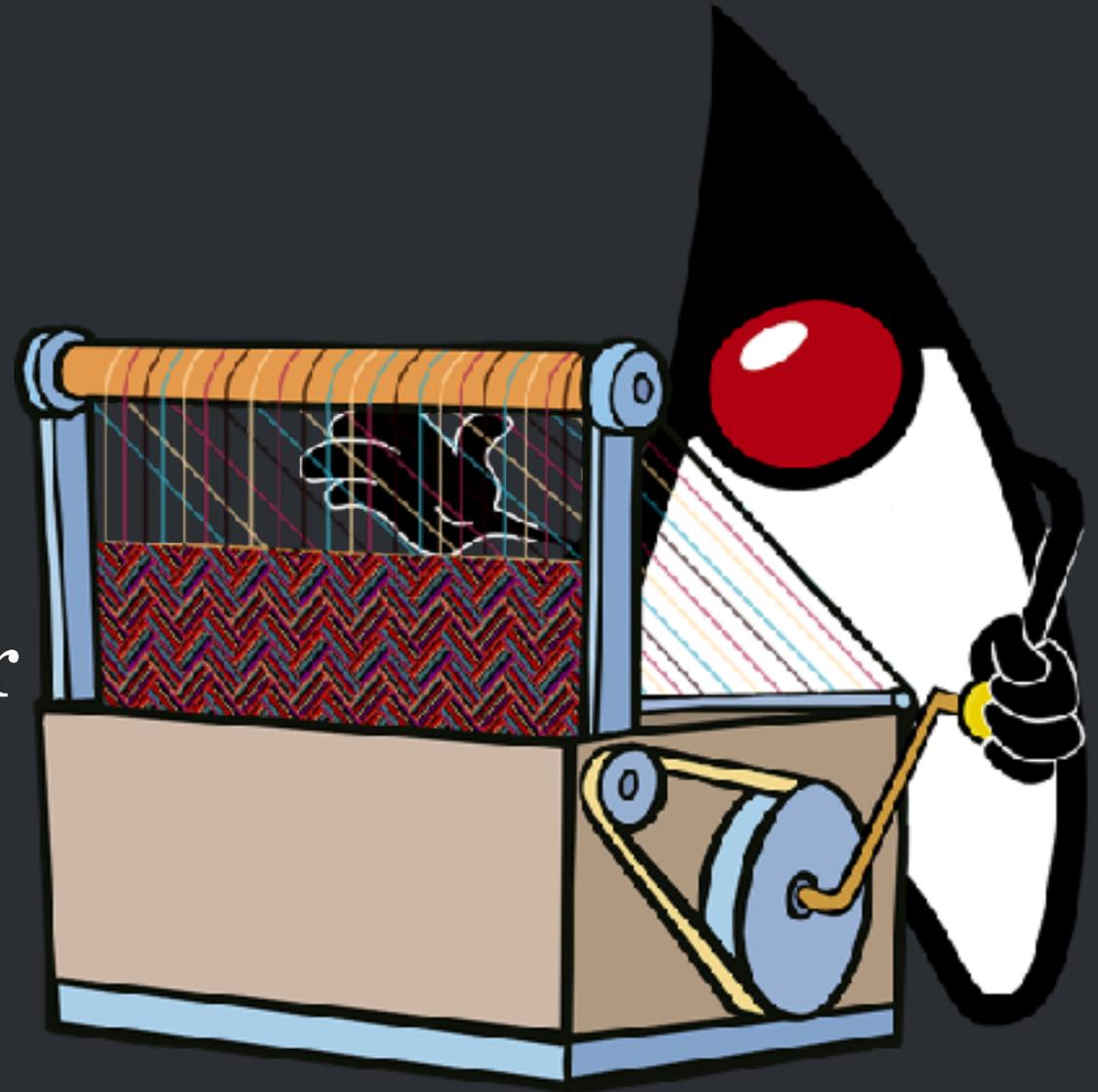
# Why User-Mode Threads Are Often the Right Answer

---

**Ron Pressler**

Java Platform Group

14 April 2021



# Java Is Made of Threads

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

The image displays three overlapping windows from an IDE, illustrating Java's multi-threaded nature and debugging tools.

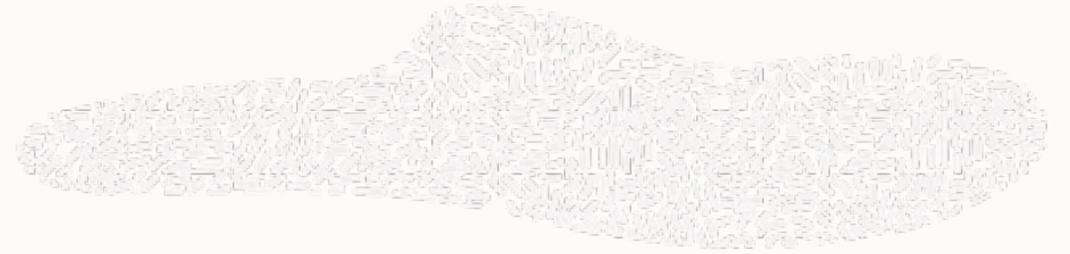
**Top Window (Code Editor):** Shows a Java class with an overridden `equals` method. The code is as follows:

```
8  
9  
10 @Override  
11 public boolean equals(Object o) {  
12     if (this == o) return true; // Point@886  
13     if (o == null || getClass() != o.getClass()) return false;  
14     Point point = (Point) o;  
15     return x == point.x && y == point.y;  
16 }
```

**Middle Window (Debugger):** Shows the "Coordinates" class being debugged. The "Frames" pane shows the current frame is `equals@1: Point.equals@886`. The "Variables" pane shows the state of variables: `res = (Point@886)` with `x = 13` and `y = 31`; `o = (Point@886)` with `x = 12` and `y = 21`. The "Watches" pane shows `p = Current (find local variable 'p')` and `o = (Point@886)`.

**Bottom Window (Profiler):** Shows a performance profile with various metrics over time. The "CPU Usage" graph shows a peak of 100% CPU usage. The "Heap Usage" graph shows memory usage around 750 MB. The "Profiling (C) (ms)" graph shows a significant spike in CPU time. The "Throughput (C) (ops/s)" graph shows a peak in throughput. The "Stack Trace" pane shows the call stack for the current thread, including `main`, `main$1`, `main$2`, and `main$3`.

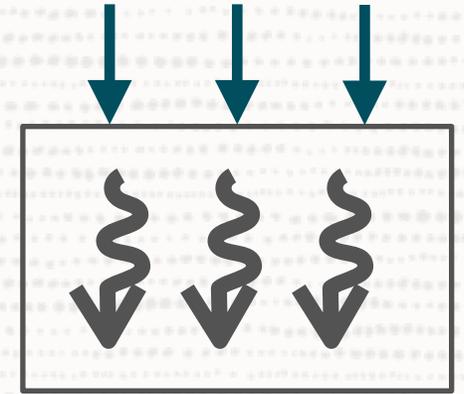
# Threads in Java



- `java.lang.Thread`
- One implementation: OS threads
- OS threads support all languages.
- RAM-heavy — megabyte-scale; page granularity; can't uncommit.
- Task-switching requires switch to kernel.
- Scheduling is a compromise for all usages. Bad cache locality.

# Synchronous

- Easy to read
- Fits well with language (control flow, exceptions)
- Fits well with tooling (debuggers, profilers)



Programmer 😊

OS / Hardware 😞

## But

- A costly resource

# Concurrency



$$L = \lambda W$$



# Reuse with Thread Pools

# Reuse with Thread Pools

- Return at end
  - Leaking ThreadLocals
  - Complex cancellation (interruption)

# Reuse with Thread Pools

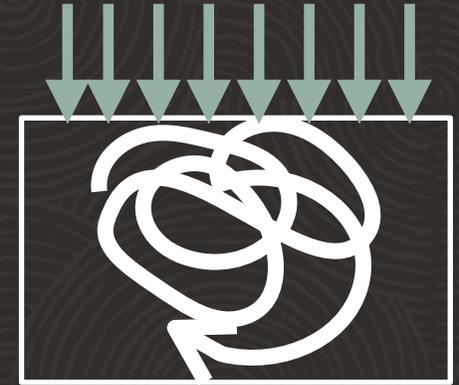
- Return at end
  - Leaking ThreadLocals
  - Complex cancellation (interruption)
- Return at wait
  - Incompatible APIs
  - Lost context

## Asynchronous

- Scalable

## But

- Hard to read
- Lost context: Very hard to debug and profile
- Intrusive; nearly impossible to migrate

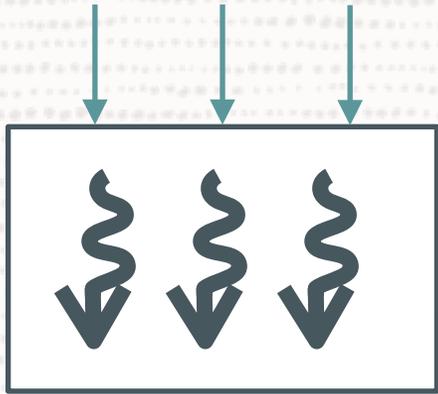


Programmer



OS / Hardware



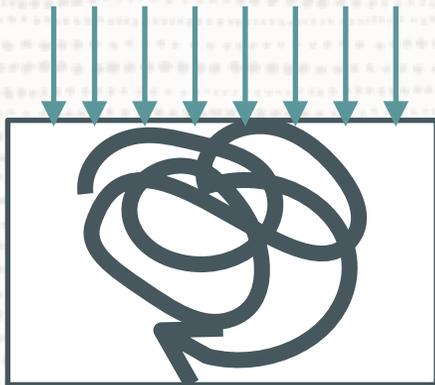


simple  
less scalable

SYNC

Programmer 😊  
OS / Hardware 😞

OR



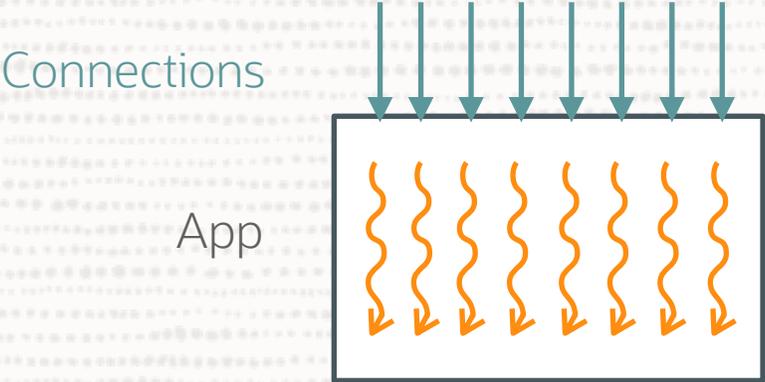
scalable,  
complex,  
non-interoperable,  
hard to debug/profile

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.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.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.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.

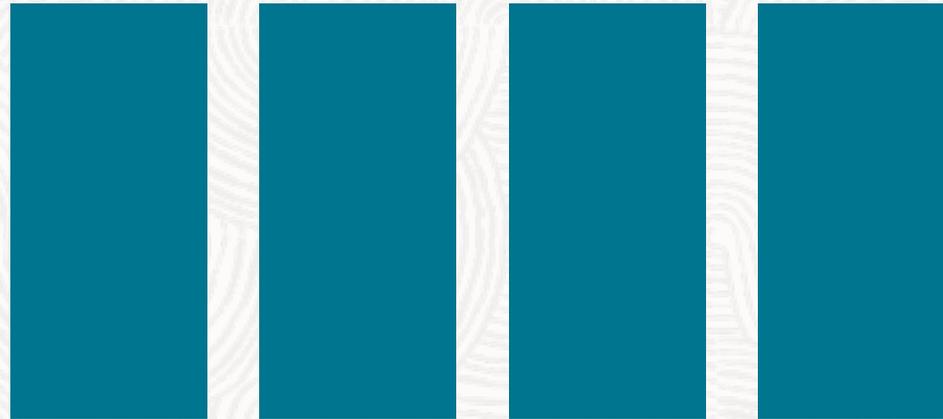
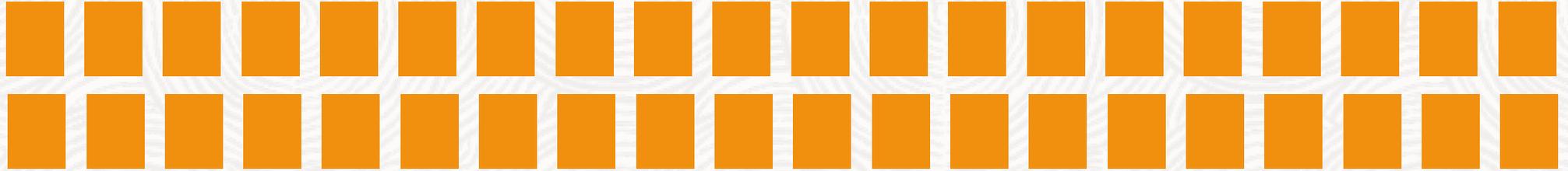
Applications that need to define a subclass of `InputStream` must always provide a method that returns the next byte of input.

Since:

1.0



## virtual threads



“carrier” platform threads managed by a scheduler



async/await

User-Mode Threads

C#

JavaScript

Kotlin

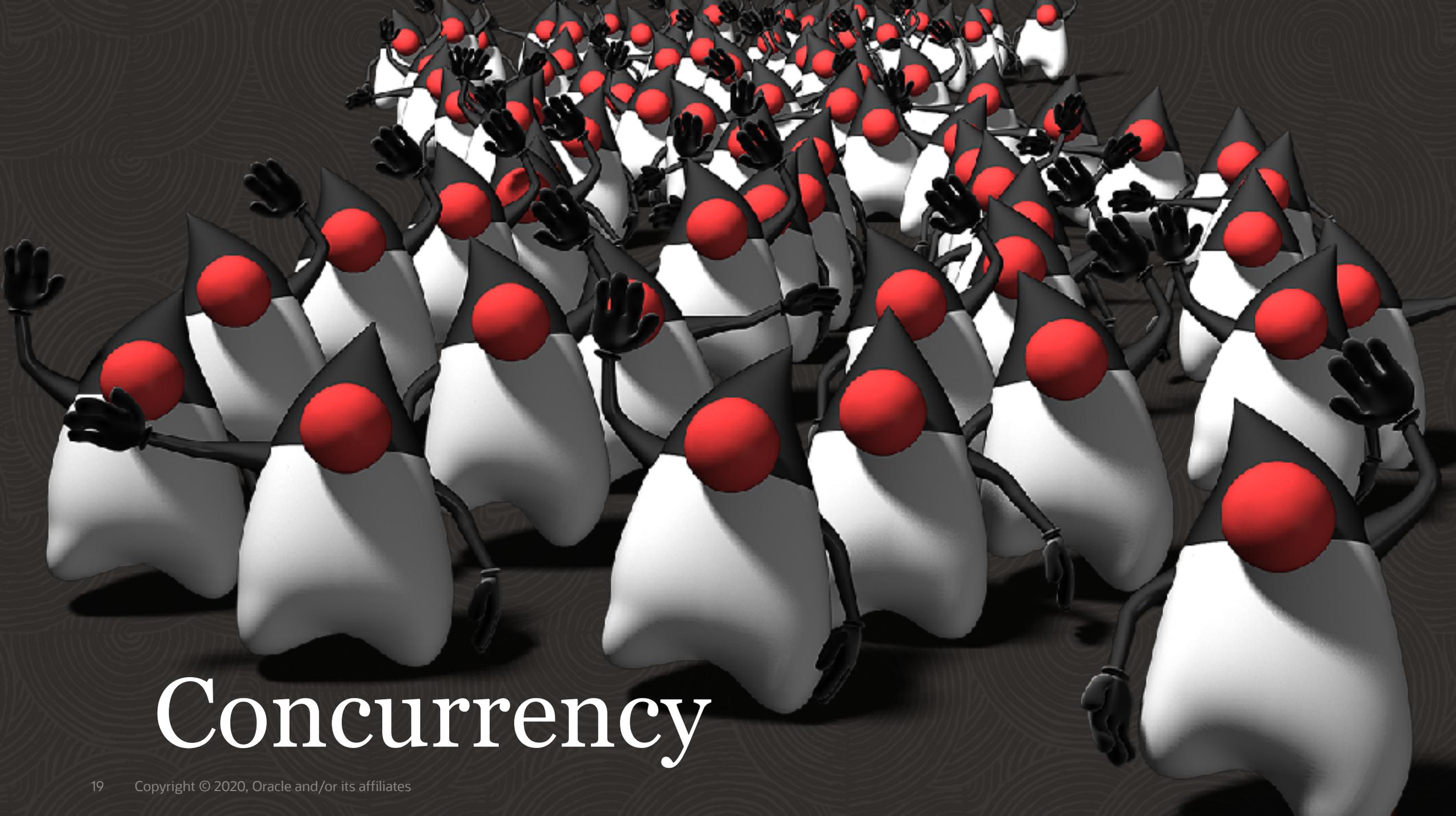
C++/Rust

Erlang

Go

Java

Zig



# Concurrency

# Algorithm (semantic)

(an abstract description of) *What* the computer does

# Expression (syntactic)

*How* the algorithm is written (in a specific programming language/paradigm)

# Concurrency: The Algorithmic View



Schedule multiple largely independent tasks to a set of computational resources

Performance: throughput (tasks/time unit)

COMPETITION

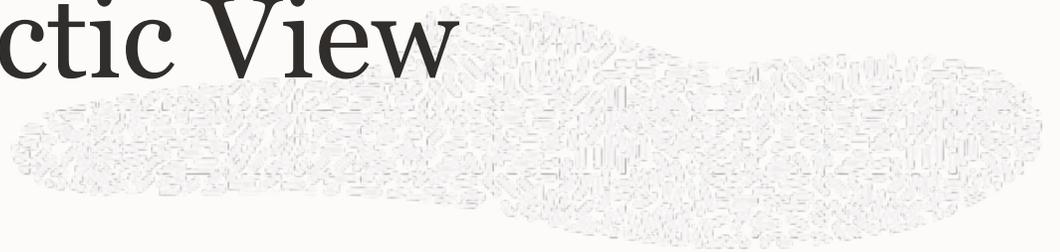
## Parallelism: The Algorithmic View

Speed up a task by splitting it to sub-tasks and exploiting multiple processing units

Performance: latency (time unit)

COOPERATION

# Concurrency: The Syntactic View



- `;` — Sequential composition  
E.g. `X;Y`, `await X;Y`, `X.andThen(Y)`
- `|` — Parallel composition  
E.g. `Thread.start(X)`, `Promise.submit(X)`
- `(a|b);c` — join  
E.g. `Thread.join`, `Future.get`
- assignment/channels/locks/IO

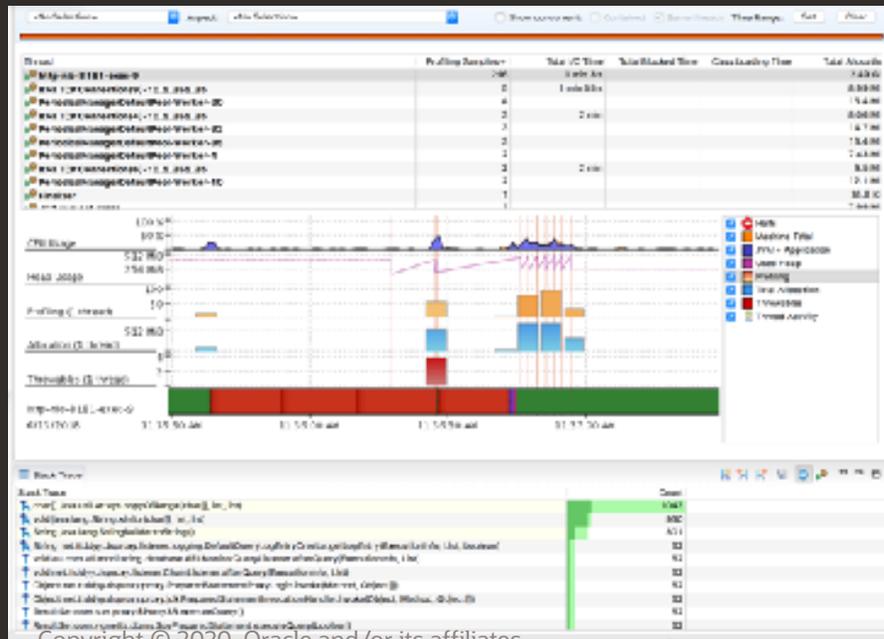
`a;((b;c)|(d|(e;f));g));h`



# Process: Unit of Concurrency

E.g. a transaction

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

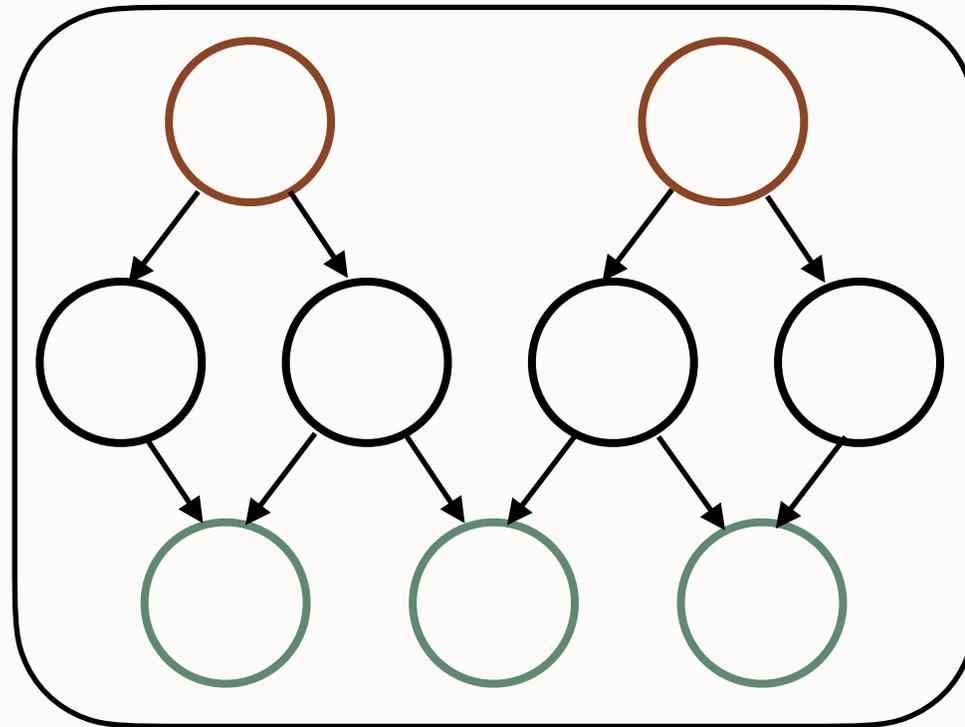


```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000
```



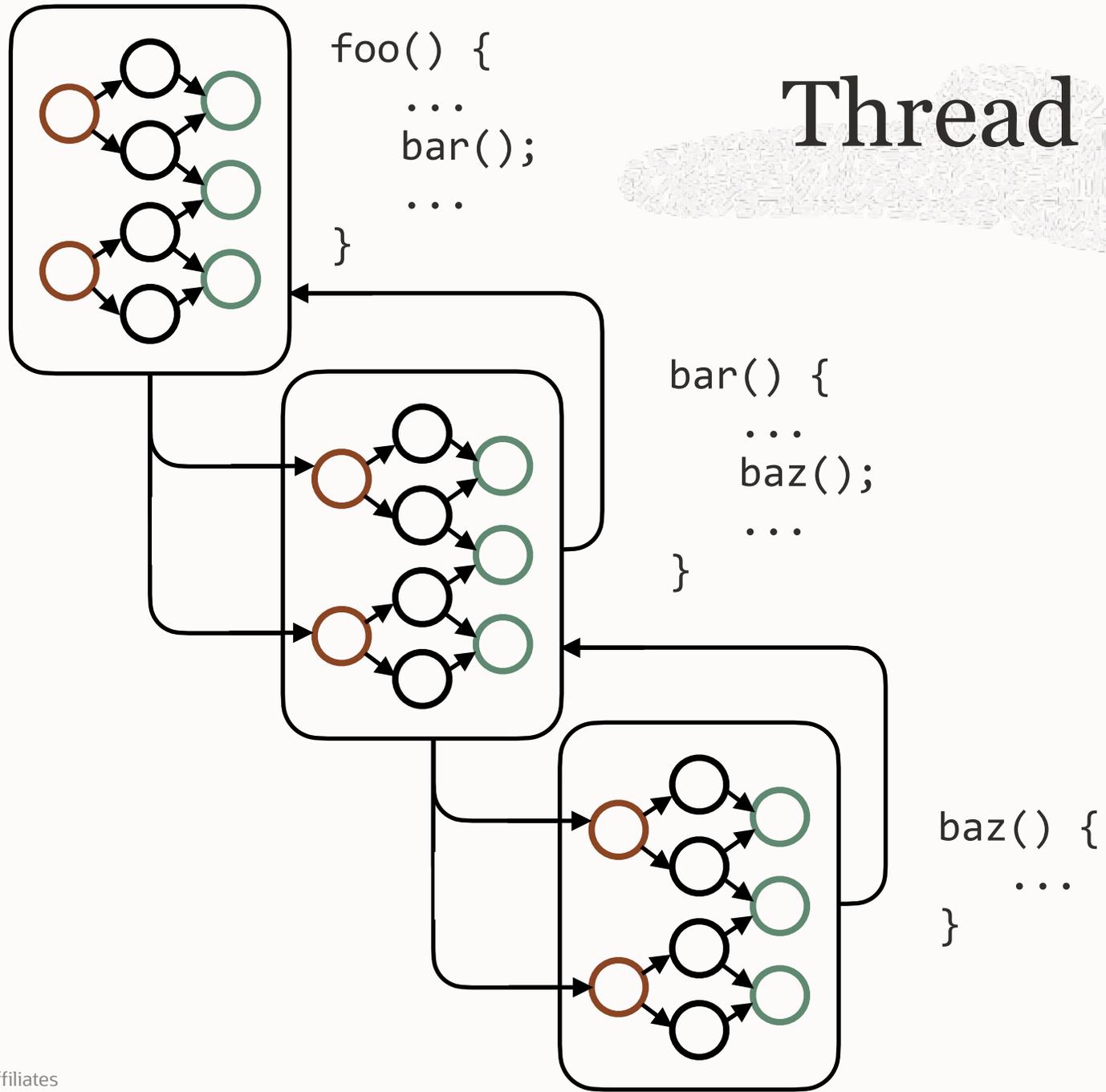
# Process

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

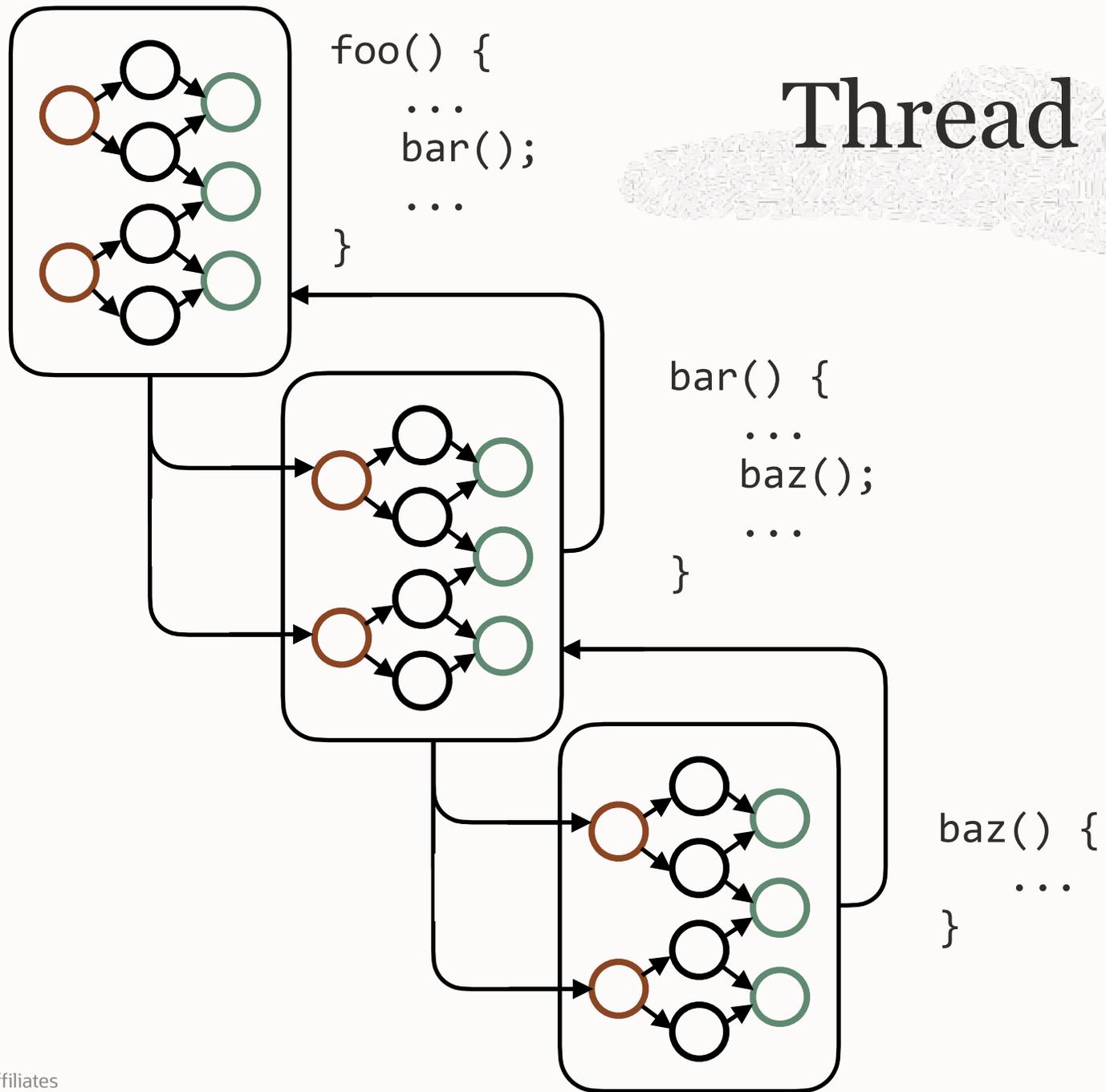
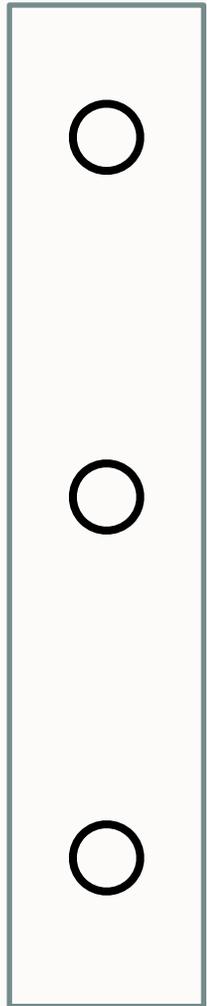


(Nondeterminism <https://youtu.be/9vupFNsND6o>)

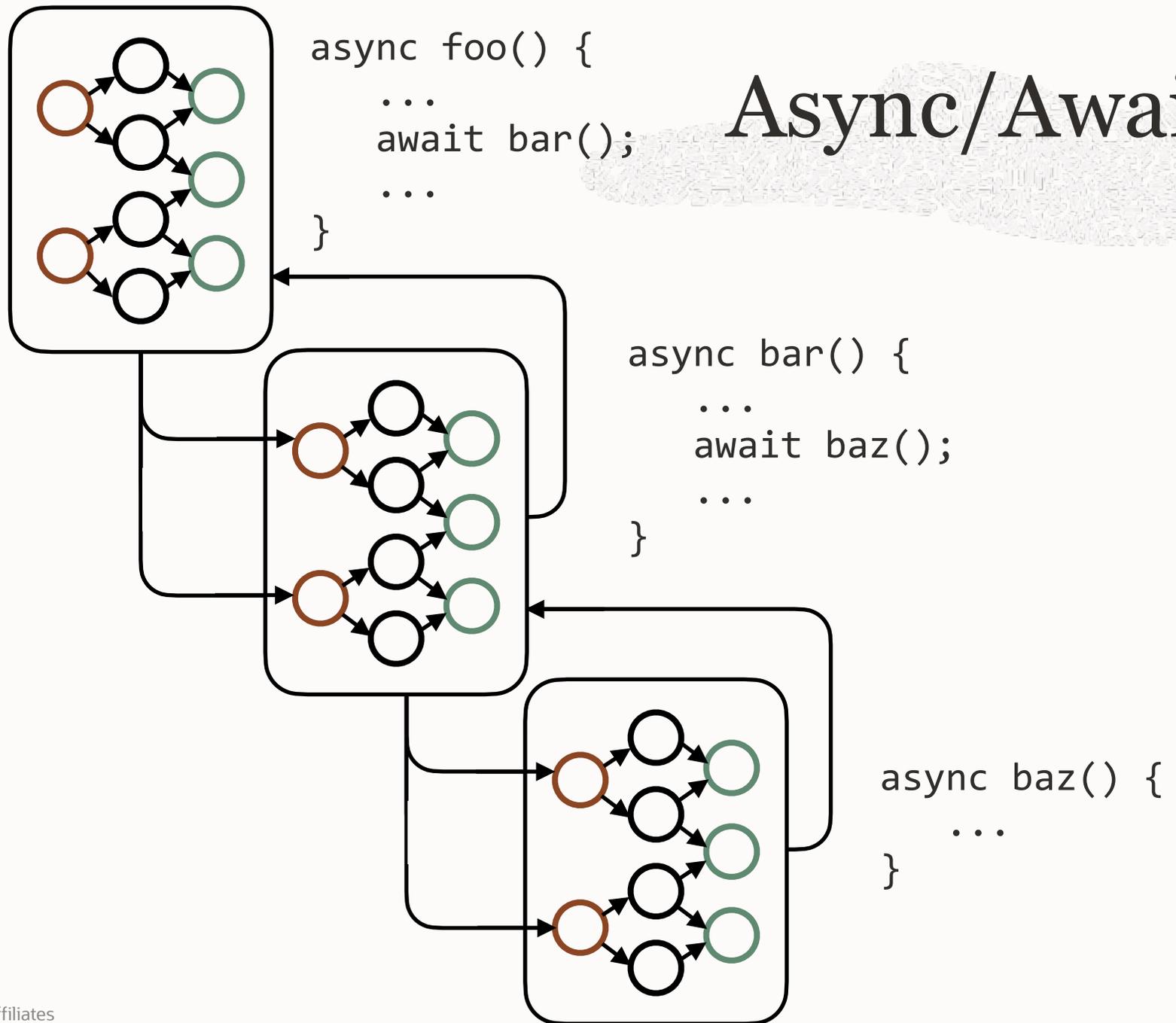
# Thread



# 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

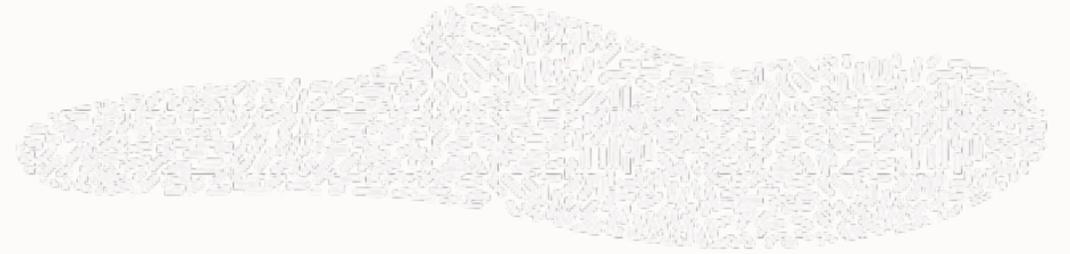
Recursion & virtual calls

**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:  $t/\mu$

$t$  — Mean context-switch latency

$\mu$  — 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  $\leq 5\%$  impact:  $\leq 200$ ns

# 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

User-Mode Threads

C#

JavaScript

Kotlin

C++/Rust

Erlang

Go

Java

Zig

# Thank you

---



ORACLE