Scaling Concurrent Applications on the Java Platform

Ron Pressler
Java Platform Group, Oracle

OpenJDK™

Safe Harbor

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

Statements in this presentation relating to Oracle's future plans, expectations, beliefs, intentions and prospects are "forward-looking statements" and are subject to material risks and uncertainties. A detailed discussion of these factors and other risks that affect our business is contained in Oracle's Securities and Exchange Commission (SEC) filings, including our most recent reports on Form 10-K and Form 10-Q under the heading "Risk Factors." These filings are available on the SEC's website or on Oracle's website at http://www.oracle.com/investor. All information in this presentation is current as of September 2019 and Oracle undertakes no duty to update any statement in light of new information or future events.

Concurrency

$$L = \lambda W$$

Threads in Java

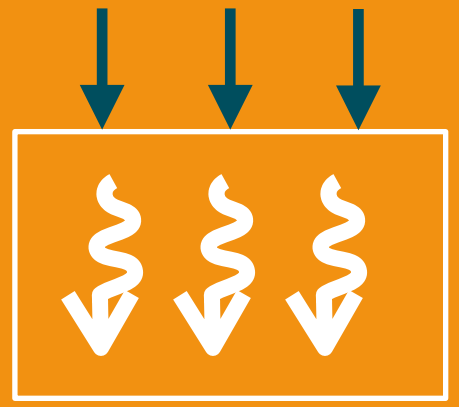
- Exceptions
- Debugger
- Profiler (JFR)

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

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

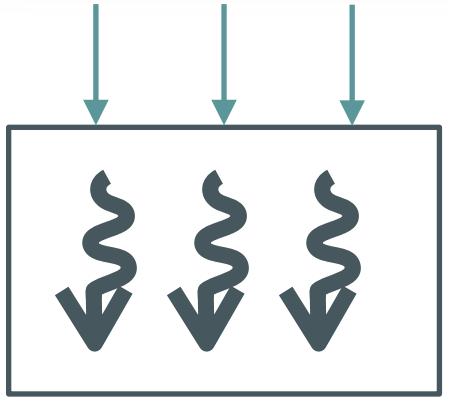
- Scalable

But

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

Programmer ☹️

OS / Hardware 😀



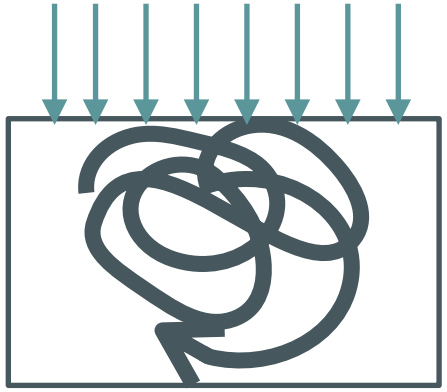
simple
less scalable

SYNC

Programmer 😀

OS / Hardware ☹️

OR



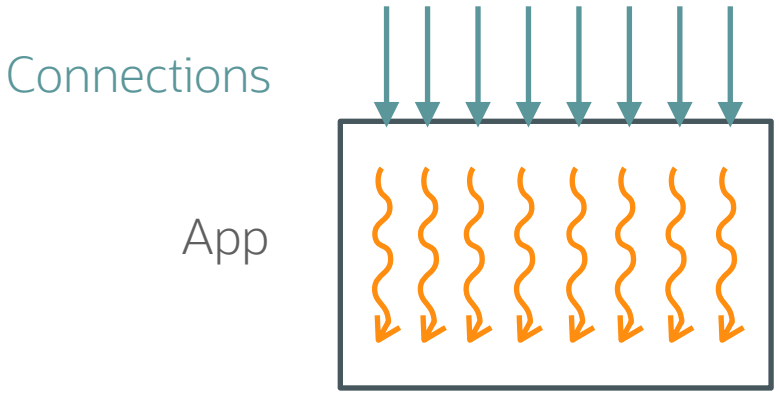
scalable,
complex,
non-interoperable,
hard to debug/profile

ASYNC

Programmer ☹️

OS / Hardware 😀

Codes Like Sync, Works Like Async



Programmer 😀

OS / Hardware 😀

“Rethink threads.”

– The Architects

"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

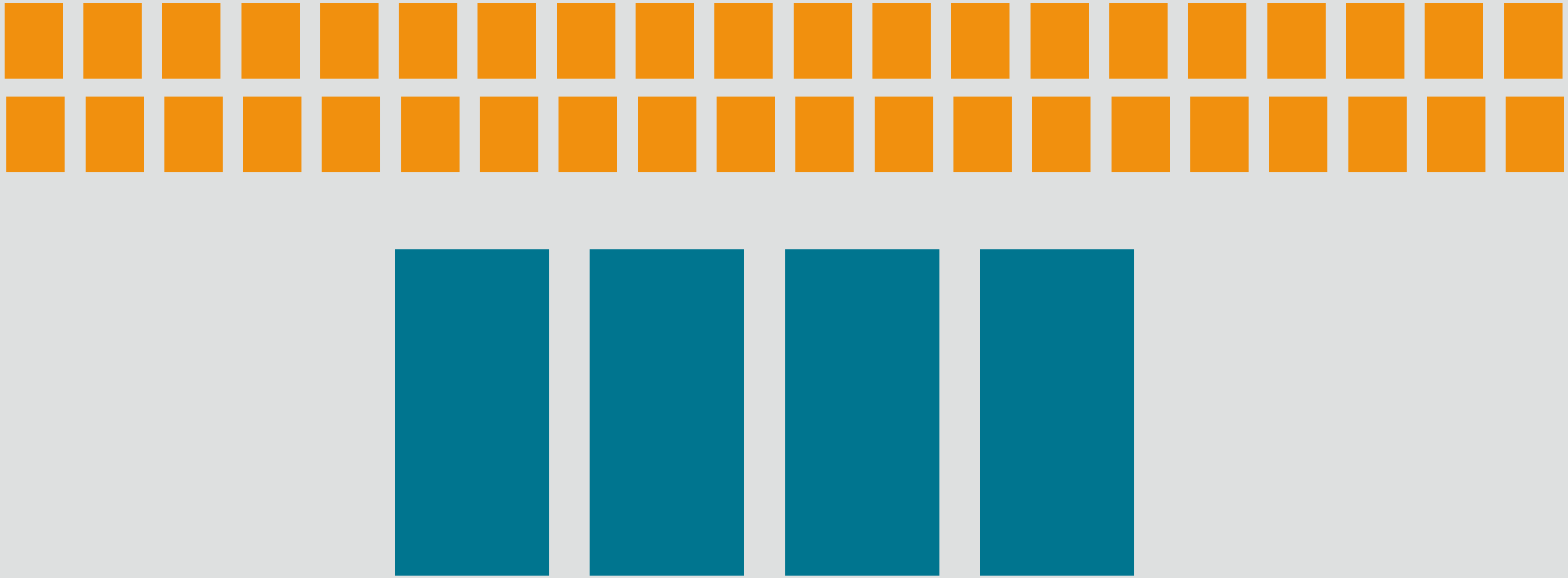
Threads *in Java*

- The use of `Thread.currentThread()` and `ThreadLocal` is pervasive. Without support, or with changed behaviour, little existing code would run
- Ever since Java 5 we've encouraged people not to use the Thread API directly anyway. People use `Executor` and `Future`, so the baggage and past API mistakes are largely inconspicuous.
- `Thread` could be cleaned up by removing long-deprecated methods.
- Realised we could drastically reduce the footprint of `Thread`.

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).
- Task-switching in user-mode, w/ VM support (continuations).
- Pluggable schedulers, default optimised for transactions.
- Can’t support blocking from native code.

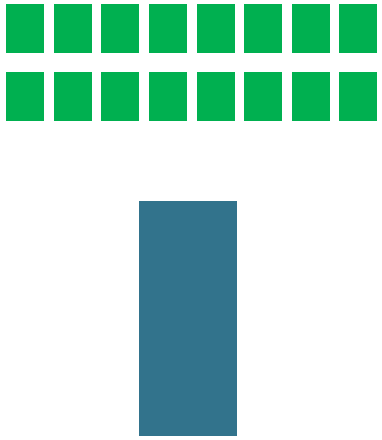
virtual threads



“carrier” heavyweight/kernel threads managed by scheduler

Java Concurrency, Then and Now

Green



Platform

Virtual

>2KB metadata 1MB stack	200-300B metadata Pay-as-you-go stack
1-10μs	~200ns

Virtual Threads

```
Thread t = Thread.startVirtualThread(() -> {
    System.out.println("Hello, Loom!");
});
```

```
Thread t = Thread.builder().virtual().task(() -> { ... }).build();
```

```
Thread t = Thread.builder().virtual().task(() -> { ... }).start();
```

```
ThreadFactory factory = Thread.builder().virtual().factory();
```

We’re “just” adding another, more scalable, implementation of threads, but this has big consequences on the code we can write.

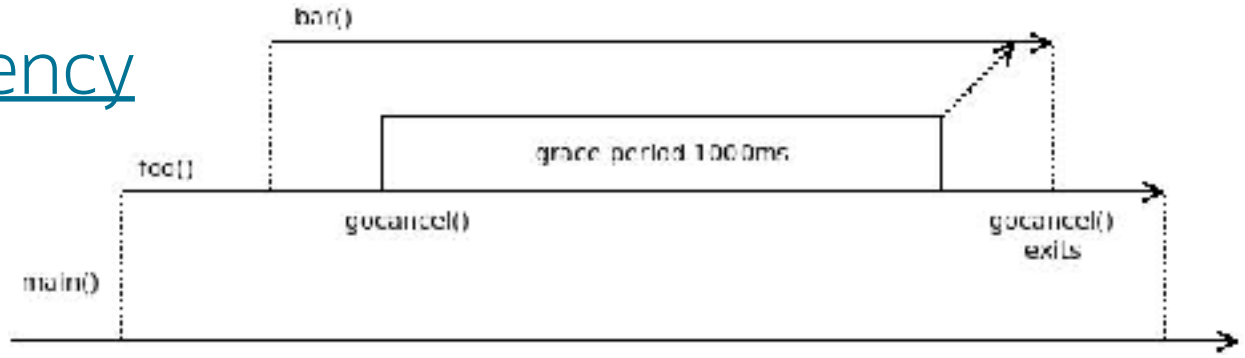
Why not language-level coroutines?

- Coroutines are **syntactic**; threads are **dynamic** — a code element is either a *coroutine* or a regular method.
- The coroutine designation is viral, just like async code.
- Coroutines require that every API is marked for use by coroutines or for use by ordinary methods. Existing APIs can't be migrated.

Structured Concurrency

Martin Sústrik (libdill, C)

- Structured Concurrency
- Update on Structured Concurrency

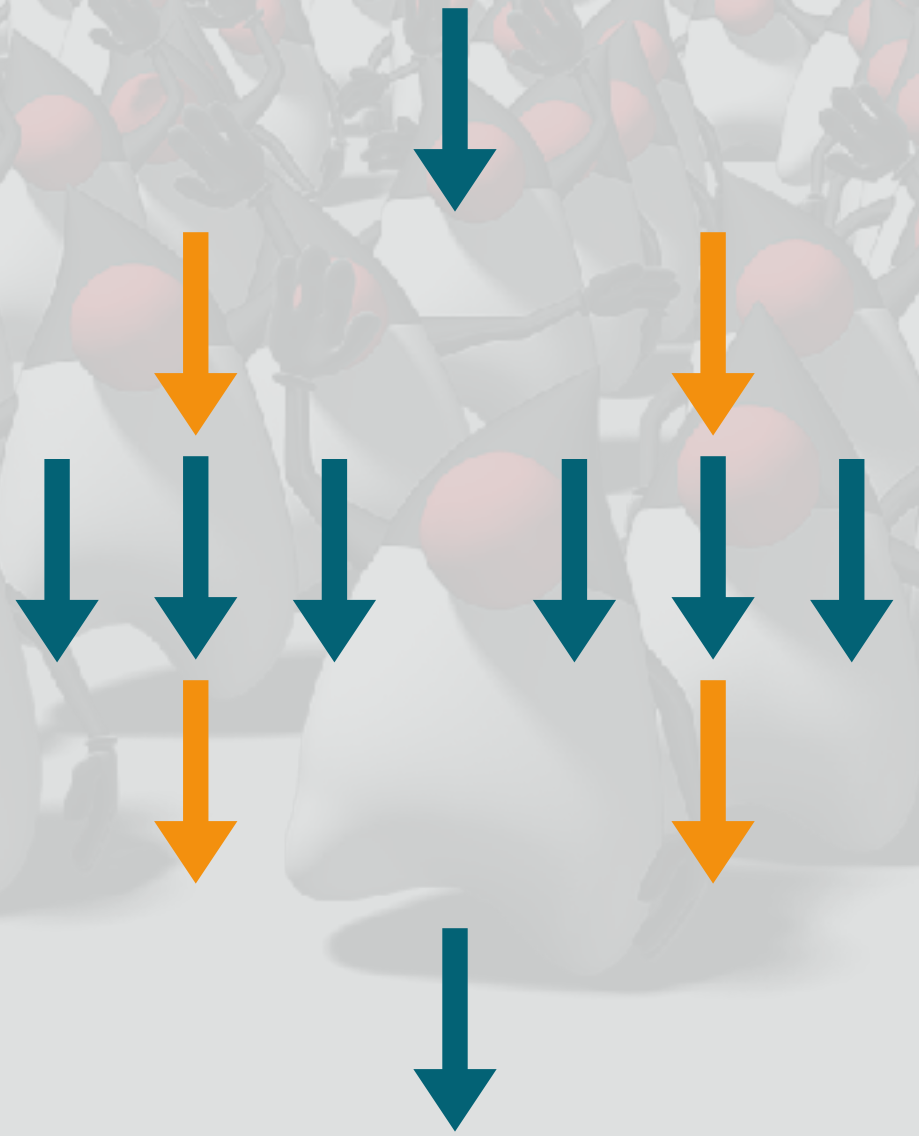


© Martin Sústrik

Nathaniel J. Smith (Trio, Python)

- Timeouts and cancellation for humans
- Notes on structured concurrency, or: Go statement considered harmful

Structured Concurrency



Structured — the runtime behaviour mirrors the structure of the code, arranged in **blocks**.

The structure of the code shows where control starts and where it ends.

Structured Concurrency

```
ThreadFactory factory = Thread.builder().virtual().factory();
try (var executor = Executors.newThreadExecutor(factory)) {
    executor.submit(task1);
    executor.submit(task2);
}
```

Structured Concurrency: Deadlines

```
ThreadFactory factory = Thread.builder().virtual().factory();
try (var executor = Executors.newThreadExecutor(factory)
                              .withDeadline(Instant.now().plusSeconds(30))) {
   executor.submit(task1);
   executor.submit(task2);
}
```

Structured Concurrency

```
try (var e = Executors.newVirtualThreadExecutor()) {
    String first = e.invokeAny(List.of(
        () -> "a",
        () -> { throw new IOException("too lazy for work"); },
        () -> "b"
     ));
     System.out.println("one result: " + first);
} catch (ExecutionException ee) {
    System.out.println("¯\\_(ツ)_/¯");
}
```

Thread Locals

- `ThreadLocal` variables, context `ClassLoader`, `InheritableThreadLocal`, `AccessControlContext`
- Don’t play well with thread pools
- **Mutable**, and **unstructured**
- **Dynamic** — Not that fast

Scope Variables (speculated)

```
static final Scoped<Integer> s1 = Scoped.forType(Integer.class);
static final Scoped<String>  s2 = Scoped.forType(String.class);


try (s1.bind(1);
     s2.bind("hello")) {
   System.out.println(foo()); // prints hello1
}



String foo() {
   return s2.get() + s1.get();
}
```

Scope Variables (speculated)

```
try (s1.bind(1); s2.bind("hello")) {
   System.out.println(foo()); // prints hello1

   try (s1.bind(2); s2.bind("goodbye")) {
      System.out.println(foo()); // prints goodbye2
   }

   System.out.println(foo()); // prints hello1
}
```

Scope Variables (speculated)

```
static final Scoped<Integer> s1 = Scoped.forType(Integer.class);
static final Scoped<String>  s2 = Scoped.forType(String.class);

try (s1.bind(99);
     s2.bind(“hello")) {

   try (var scope = Executors.newVirtualThreadExecutor()) {
      scope.submit(task1);
      scope.submit(task2);
   }

}
```

- Scope locals expected to supplant many uses of `ThreadLocal`
- Other require “processor locals”

Status

- `java.util.concurrent` works, but requires re-tuning.
- `Thread.sleep`
- `java.net.Socket/ServerSocket` (JDK 13)
- `java.nio.channels.SocketChannel` and friends (JDK 11)
- JSSE implementation of TLS
- `AccessController.doPrivileged` w/o native frame (JDK 12)
- `java.lang.reflect.Method.invoke` requires more work
- Monitors/`Object.wait()` pin thread (temporary)
- Native frames pin thread
- Debugger support
- Initial JFR support

Further work

- `java.net.InetAddress`
- Console I/O
- File I/O ?
- Thread dumps
- Channels ?

Q & A

Mailing list: loom-dev@openjdk.java.net

Repo: https://github.com/openjdk/loom

Early Access: http://jdk.java.net/loom/