JAVA SESSIONS

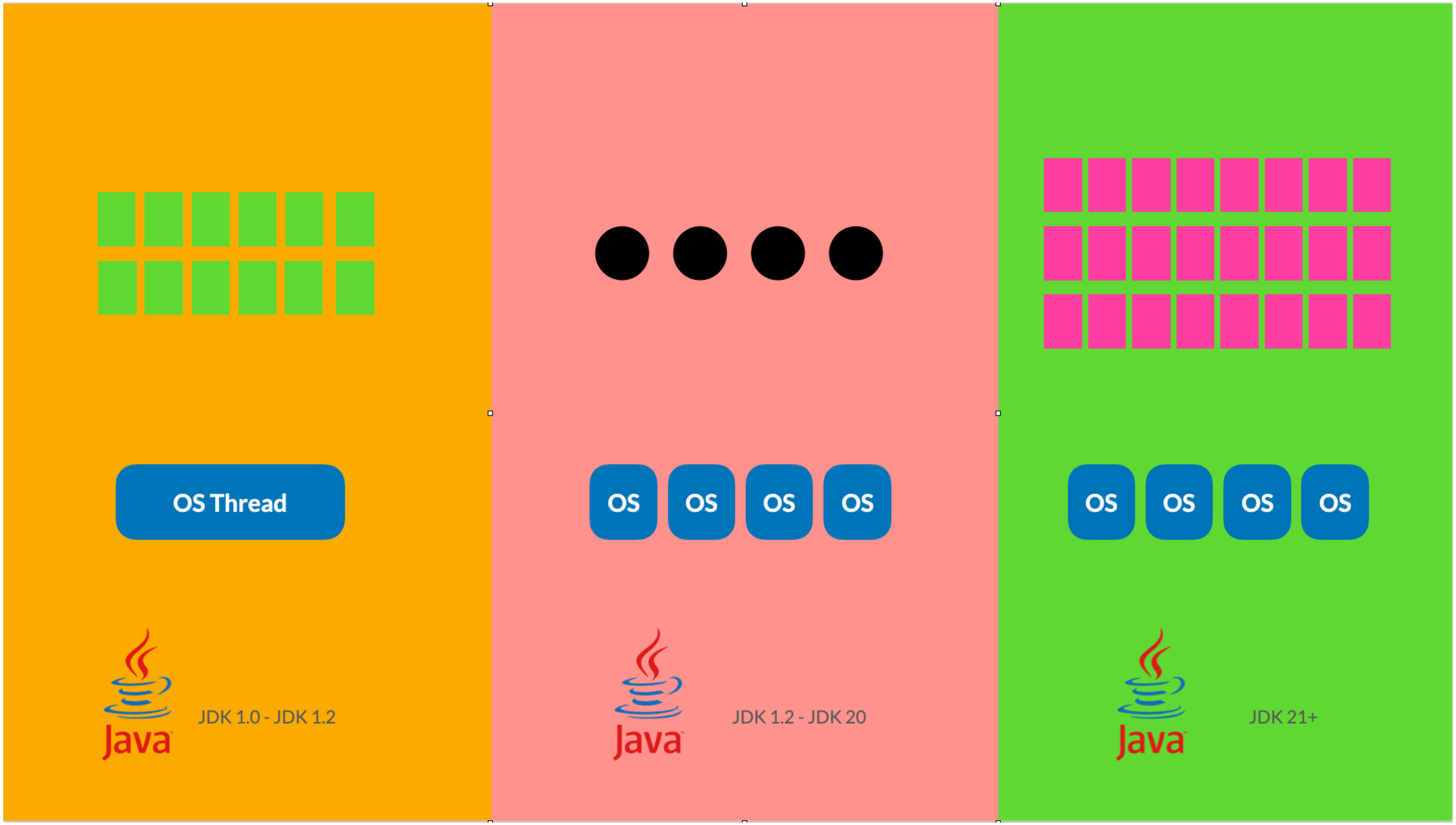
VIRTUAL THREADS AND STRUCTURED CONCURRENCY

Daniel Hinojosa

VIRTUAL THREADS

VIRTUAL THREADS

- 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.



OS Thread

JDK 1.0 - JDK 1.2

OS OS OS OS

JDK 1.2 - JDK 20

OS OS OS OS

JDK 21+

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

- Kernel threads
  - Managed by the operating system kernel.
  - The kernel schedules them onto CPU cores.
  - Every kernel thread is visible to the OS.
- User threads
  - Managed by a runtime or language library (e.g., JVM, Go runtime, green threads in old Python).
  - The OS only sees the underlying kernel threads, not the user-level abstractions.
  - The mapping between user threads and kernel threads can be:
    - 1:1 (Java platform threads, pthreads, C# threads, etc.)
    - M:N (virtual threads in Java, goroutines in Go, Haskell lightweight threads, etc.).

USER VS. KERNEL JVM EDITION

- `java.lang.Thread` (before Project Loom)
  - Each Thread object was mapped 1:1 to a kernel thread.
  - “User thread” == JVM object, but backed directly by a kernel-managed thread.
- Virtual Threads (Loom, Thread.ofVirtual())
  - Still java.lang.Thread objects, but now user-mode scheduled by the JVM.
  - Many virtual threads can run on a much smaller pool of kernel threads (“carrier threads”).
  - The kernel only sees and manages the carriers, not the thousands of virtual threads.

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`

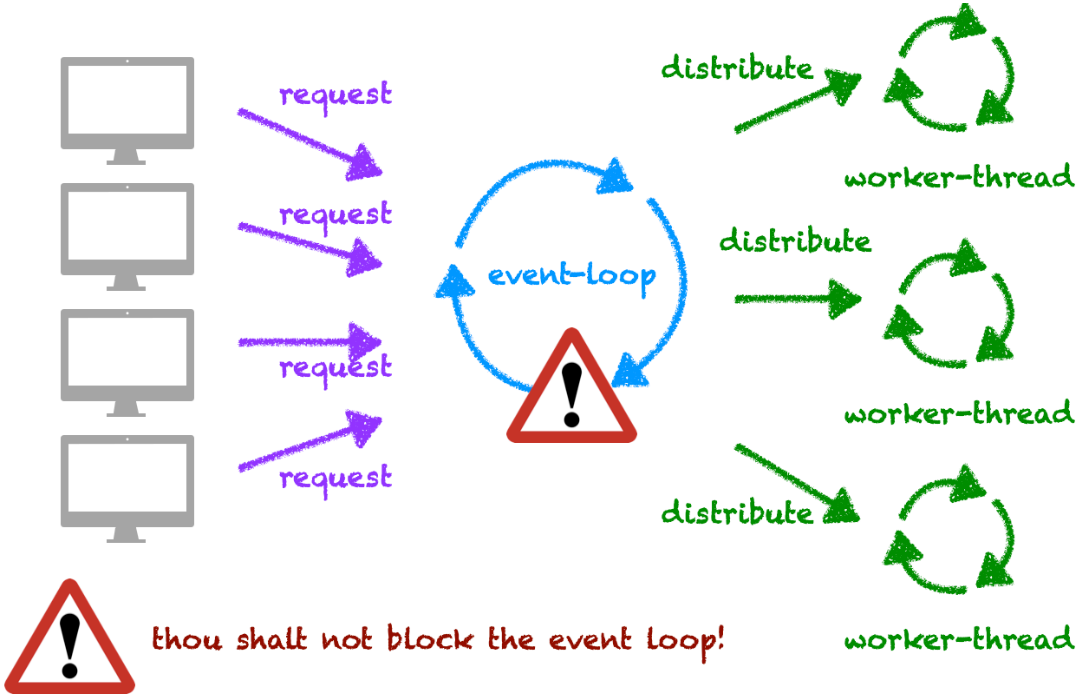
THREAD WEIGHT AND EXPENSE

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

HOW DO WE OR HOW DID WE COMPENSATE?

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

EXAMPLE FROM VERT.X



🛈 Vert.x is a concurrency framework build on Futures and Thread Pools. You do have to follow the rules.

RXJAVA/PROJECT REACTOR

```java
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
   .subscribe(System.out::println);     // Print
```

COMPLEX CANCELLATION

- `Thread`s 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

- *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

VIRTUAL THREADING USES CONTINUATIONS

- 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

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 THREAD

- Schedulers are implemented with a `ForkJoinPool`
- Carrier Threads 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

- 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.

RELEARN THE OLD WAYS

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

PINNING

- 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.
  - This is no longer of a concern with JEP 491
- **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
  - JNI will likely be replaced by the *Foreign Function and Memory API*

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
- 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

- Profiling asynchronous code has traditionally been difficult, since threads may be short-lived or reused, making attribution unclear.
- With continuations, all code executed within a virtual thread belongs to a well-defined logical context, so profilers can collate subtasks more accurately.
- Java Flight Recorder (JFR) has been enhanced to fully support virtual threads. It can attribute events (CPU, allocations, blocking, etc.) to the logical virtual thread.
- Blocked virtual threads appear clearly in the profiler, and time spent waiting (e.g., on I/O) is measured and accounted for separately from active CPU time.

DEMO: VIRTUAL THREADS

In the *java_sessions_virtual_threads_structured_concurrency/src/main/java* directory, navigate to the `com.evolutionnext.demo.concurrency.virtualthreads` package and open all the classes.

DEMO: VIRTUAL THREADS

In the *java_sessions_virtual_threads_structured_concurrency* project, go to the */src/main/java* directory and navigate to the `com.evolutionnext.concurrency.virtualthreads` package and open the two classes.

STRUCTURED CONCURRENCY

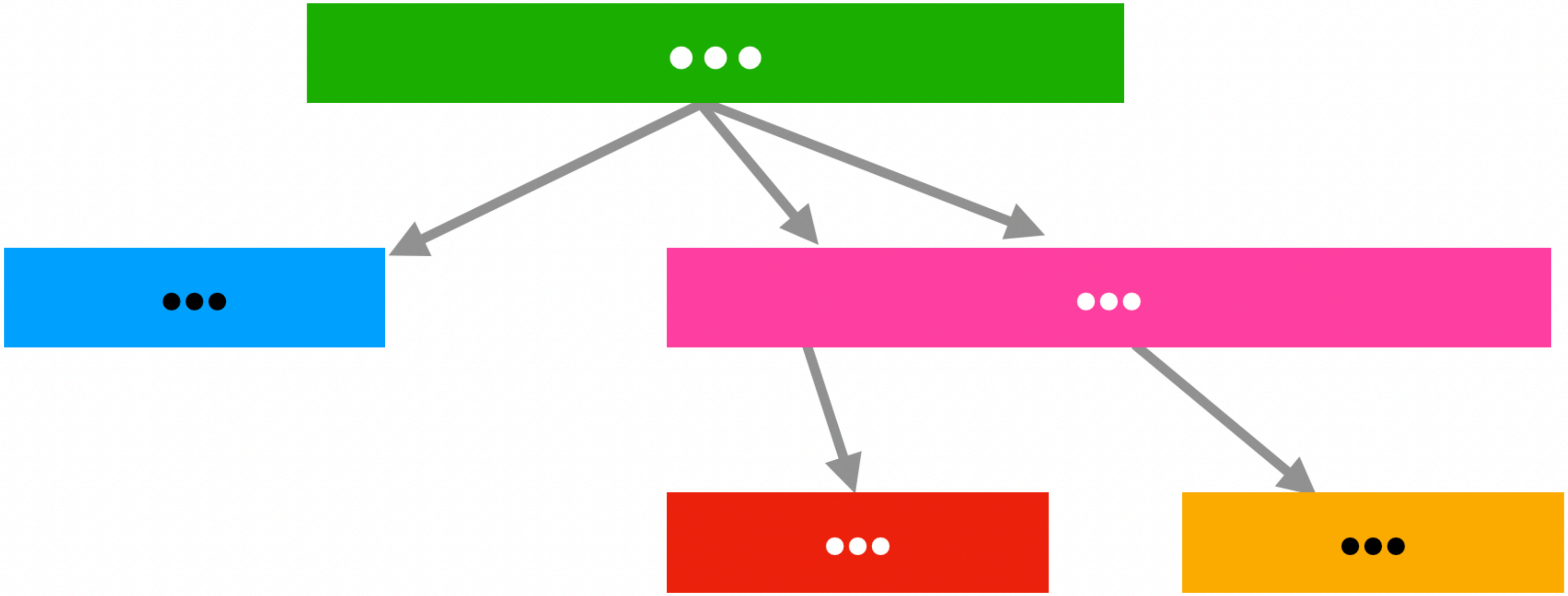
STRUCTURED CONCURRENCY

- **"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

STRUCTURED CONCURRENCY 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.
- Eliminate common risks arising from cancellation and shutdown, such as thread leaks and cancellation delays
- Improve Observability

STRUCTURED CONCURRENCY DIAGRAM



ISSUE WITH `ExecutorService` WITH STRUCTURED CONCURRENCY

- `ExecutorService` can perform tasks concurrently, but can fail *independently*
- `ExecutorService` and `Future<T>` allow unrestricted patterns of concurrency
- `ExecutorService` does not enforce or even track relationships among tasks and subtasks, even though such relationships are common and useful

UNSTRUCTURED CONCURRENCY

- What happens if each of the `Future`'s will fail?
- What happens if any of the `Future`'s will take a long time to complete?

```java
Response handle() throws ExecutionException, InterruptedException {
    Future<String>  user  = esvc.submit(() -> findUser());
    Future<Integer> order = esvc.submit(() -> fetchOrder());
    String theUser  = user.get();   // Join findUser
    int    theOrder = order.get();  // Join fetchOrder
    return new Response(theUser, theOrder);
}
```

STRUCTURED CONCURRENCY

Structured concurrency derives from the simple principle that:

If a task splits into concurrent subtasks then they all return to the same place, namely the task's code block.

NOTE THE DIFFERENCE

java

```java
Response handle() throws ExecutionException, InterruptedException {
    try (var scope = StructuredTaskScope.open(StructuredTaskScope.Joiner.anySuccessfulResul
        Supplier<String>  user  = scope.fork(() -> findUser());
        Supplier<Integer> order = scope.fork(() -> fetchOrder());

        scope.join();

        // Here, both subtasks have succeeded, so compose their results
        return new Response(user.get(), order.get());
    }
}
```

SHUTDOWN POLICIES

- A ShutdownPolicy describes what should be done when there is an error
- Shutdown Policies as of 25, are defined with `Joiner`
- `Joiner` have the following:
  - `Joiner.allSuccessfulOrThrow` - Join all and must be all successful. Returns result
  - `Joiner.anySuccessfulOrThrow` - Join any and either should be successul. Returns result
  - `Joiner.awaitAllSuccessfulOrThrow()` - Join all and must be all successful. Returns `null`
  - `Joiner.awaitAll()` - Waits fow all subtasks. Permissive. Does not cancel the scope if a subtask fails.
  - `Joiner.allUntil()` - Returns a new Joiner object that yields a stream of all subtasks when all subtasks complete or a predicate returns `true` to cancel the scope.

DEMO: STRUCTURED CONCURRENCY

In the *java_sessions_virtual_threads_structured_concurrency/src/main/java* directory, navigate to the `com.xyzcorp.concurrency.structuredconcurrency` package and open all the classes.

DEMO: STRUCTURED CONCURRENCY

In the *java_sessions_virtual_threads_structured_concurrency* project, go to the */src/main/java* directory and navigate to the `com.evolutionnext.demo.structuredconcurrency` package and open the two classes.

SCOPED VALUES

SCOPED VALUES

- Enable a method to share immutable data both with its callees within a thread, and with a child thread
- Scoped values are easier to reason about than thread-local variables.
- They also have lower space and time costs, especially when used together with virtual threads (JEP 444 and structured concurrency (JEP 480).

Source: JEP 481

GOALS

- *Ease of use* — It should be easy to reason about dataflow.
- *Comprehensibility* — The lifetime of shared data should be apparent from the syntactic structure of code.
- *Robustness* — Data shared by a caller should be retrievable only by legitimate callees.
- *Performance* — Data should be efficiently sharable across a large number of threads.

Source: JEP 481

NON-GOALS

- It is not a goal to change the Java programming language.
- It is not a goal to require migration away from thread-local variables or to deprecate the existing ThreadLocal API.

Source: JEP 481

WHAT WAS THE PROBLEM WITH THREAD LOCAL?

- *Unconstrained mutability* — Every thread-local variable is mutable: Any code that can call the get method of a thread-local variable can call the set method of that variable at any time.
- *Unbounded lifetime* — Once a thread's copy of a thread-local variable is set via the set method, the value to which it was set is retained for the lifetime of the thread, or until code in the thread calls the remove method.
- *Expensive inheritance* — The overhead of thread-local variables may be worse when using large numbers of threads, because thread-local variables of a parent thread can be inherited by child threads. (A thread-local variable is not, in fact, local to one thread.)

Source: JEP 481

ThreadLocal

- If you have a data structure that isn’t safe for concurrent access, you can sometimes use an instance per thread, hence `ThreadLocal`.
- `ThreadLocal`s 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
- `ThreadLocal`s can leak into one another

EXAMPLE OF `ThreadLocal`

`ThreadLocal` may already be set

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

COMPLICATED TRACING

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

HOW IS THIS TIED TO VIRTUAL THREADS

- The problems of thread-local variables have become more pressing with the availability of virtual threads (JEP 444)
- Can be used for both Virtual Threads, and classic threads.
- The Java Platform should provide a way to maintain inheritable per-thread data for thousands or millions of virtual threads.

Source: JEP 481

WHAT IS THE `ScopedValue` RECIPE?

1. A scoped value is a container object that allows a data value to be safely and efficiently shared by a method with its direct and indirect callees within the same thread, and with child threads, without resorting to method parameters.

2. It is a variable of type `ScopedValue`.

3. It is typically declared as a `final static field`

4. Its accessibility is set to `private` so that it cannot be directly accessed by code in other classes.

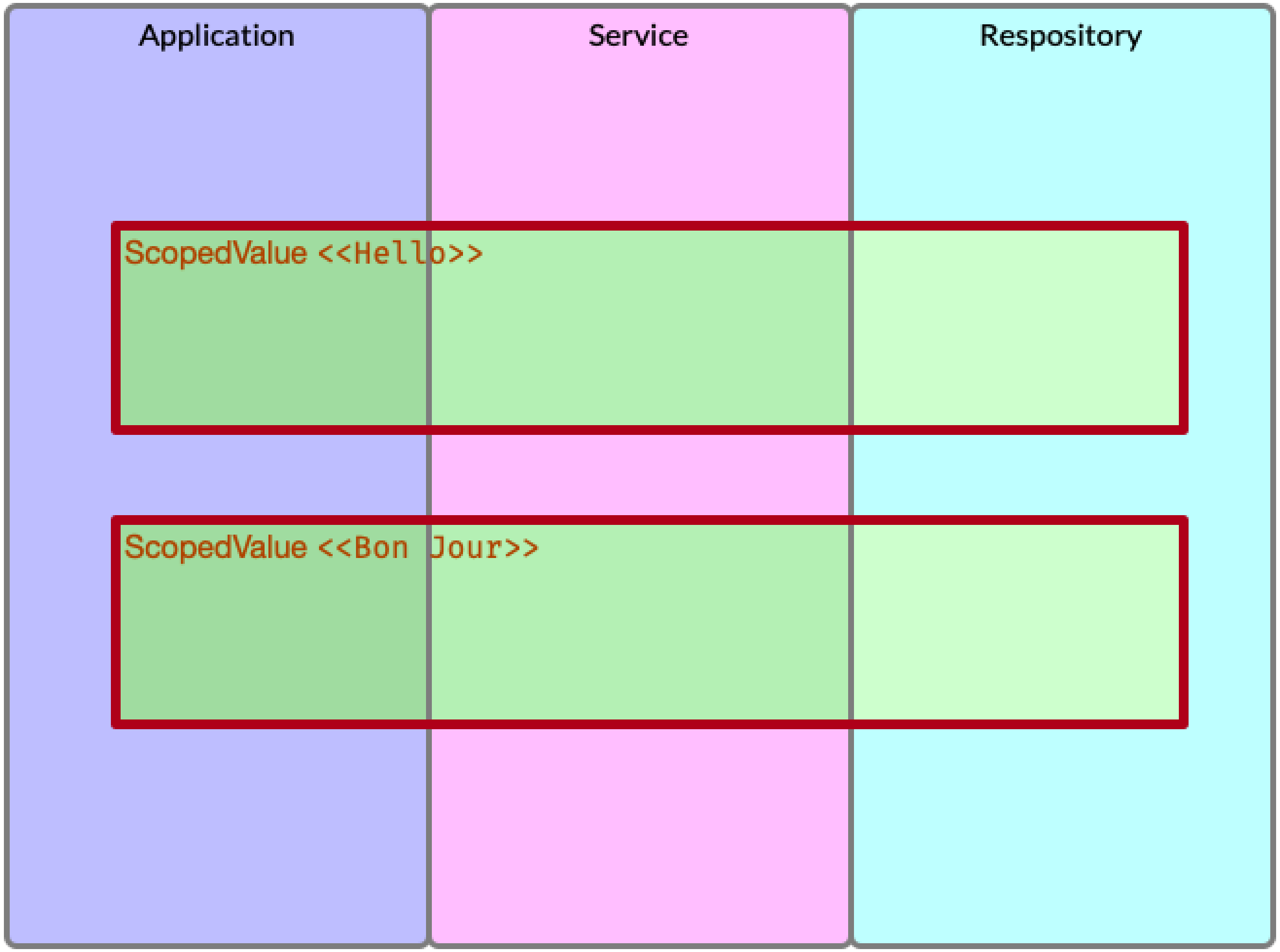
Source: JEP 481

DIFFERENCE BETWEEN `ScopedValue` AND `ThreadLocal`

- Like a thread-local variable, a `ScopedValue` has multiple values associated with it, one per thread.
- The particular value that is used depends on which thread calls its methods.
- Unlike a thread-local variable, a `ScopedValue` is written once, and is available only for a bounded period during execution of the thread.

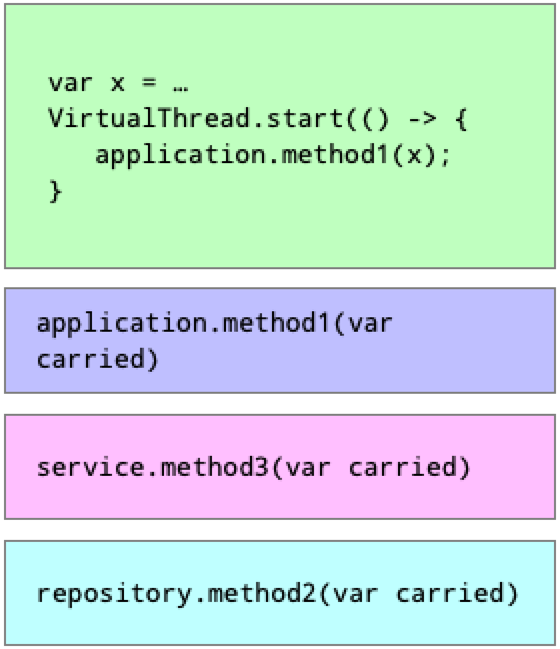
Source: JEP 481

SCOPED VALUES



WITHOUT SCOPED VALUES

Without Scoped Values, we are left to carry variables, stacks down



```
var x = …
VirtualThread.start(() -> {
   application.method1(x);
}
```

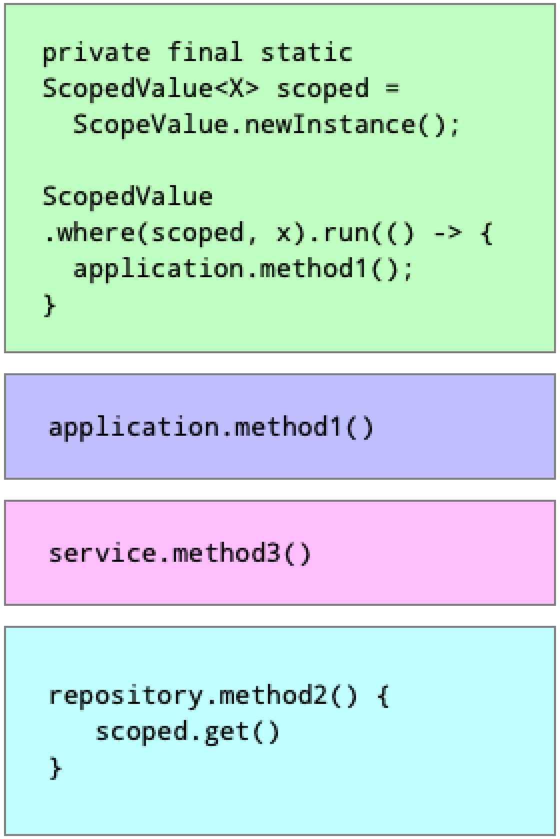
```
application.method1(var
carried)
```

```
service.method3(var carried)
```

```
repository.method2(var carried)
```

WITH SCOPED VALUES

With Scoped Values we can establish a context and recall it stacks down



```
private final static
ScopedValue<X> scoped =
  ScopeValue.newInstance();

ScopedValue
.where(scoped, x).run(() -> {
  application.method1();
}
```

```
application.method1()
```

```
service.method3()
```

```
repository.method2() {
   scoped.get()
}
```

DEMO: SCOPED VALUES

In the *java_sessions_virtual_threads_structured_concurrency/src/main/java* directory, navigate to the `com.xyzcorp.scopedvalues` package and open all the classes.

DEMO: SCOPED VALUES

In the *java_sessions_virtual_threads_structured_concurrency* project, go to the */src/main/java* directory and navigate to the `com.evolutionnext.demo.scopedvalues` package and open the two classes.

THANK YOU

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Mastodon: https://mastodon.social/@dhinojosa
- Linked In: https://www.linkedin.com/in/dhevolutionnext