JAVA SESSIONS

THREADING AND SYNCHRONIZERS DEEP DIVE

Daniel Hinojosa

THREADS

THREADS

- An independent path of execution with code.
- Multiple threads executing within the same program is a *multithreaded application*
- All Threaded code is performed using `java.lang.Thread`
- In every Java application there is a non-daemon (non-background thread)
- All threads will be executed until:
  - `Runtime.exit()` has been called
  - All non-daemon threads have been terminated

CREATING A BASIC `Thread`

- Two different philosophies
  - extending `Thread`
  - using a `Runnable` and plugging it into a `Thread`

EXTENDING Thread

```
class MyThread extends Thread {
   private boolean done = false;

   public void finish() {
      this.done = true;
   }

   public void run() {
       while (!done) {
         try {
           Thread.sleep(1000);
         } catch (InterruptedException ie) {
           //ignore
         }
         System.out.print(String.format("In Run: [%s] %s\r\n",
             Thread.currentThread().getName(), LocalDateTime.now()));
```

THREADS WITH `Runnable`

- A Thread can be created with instances of the `Runnable` interface
- `Runnable` interface has a `run` method and what is used in the interface is what is run.
- Perfect to have plug the same behavior into multiple `Thread`

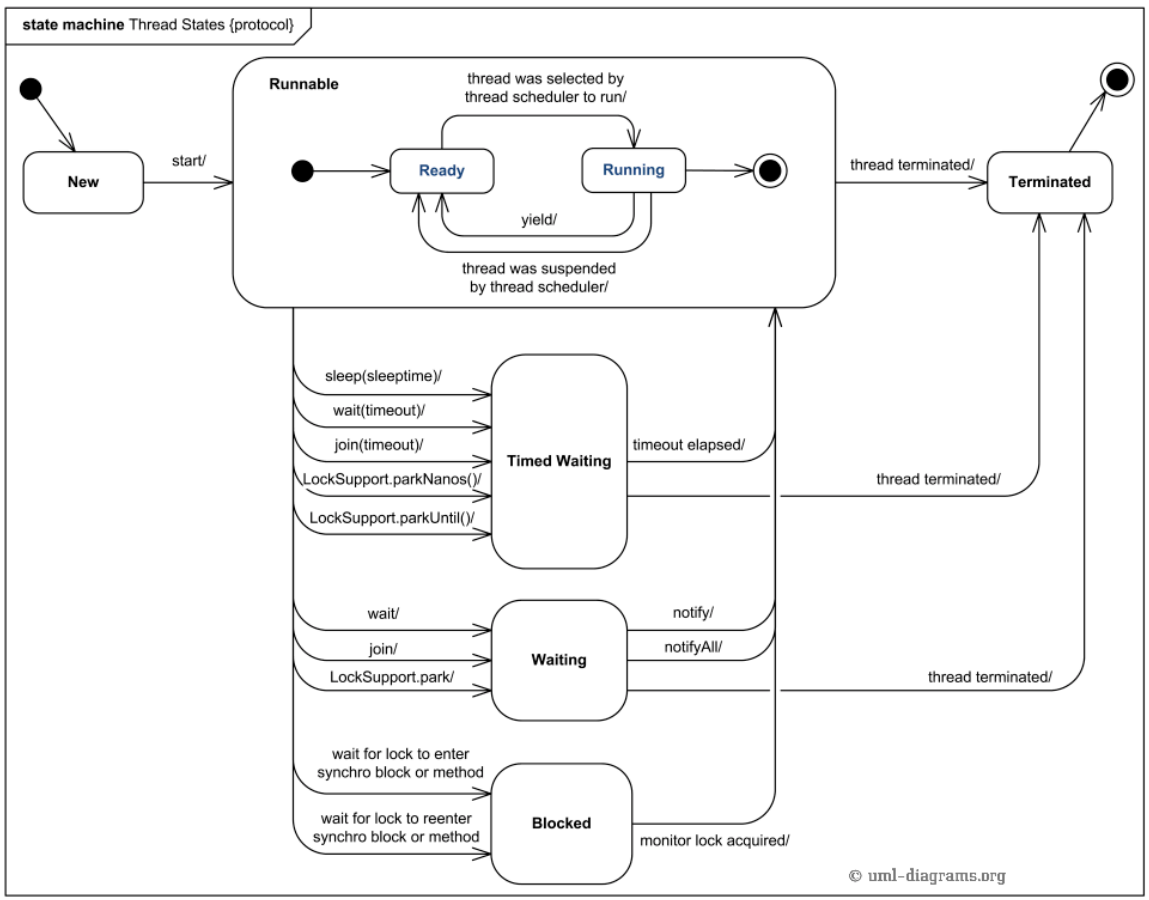
CREATE A Thread WITH Runnable

```
class MyRunnable implements Runnable {
   private boolean done = false;

   public void finish() {
      this.done = true;
   }

   public void run() {
       while (!done) {
         try {
           Thread.sleep(1000);
         } catch (InterruptedException ie) {
           //ignore
         }
         System.out.print(String.format("In Run: [%s] %s\r\n",
             Thread.currentThread().getName(), LocalDateTime.now()));
```

COMMON `Thread` METHODS

- `void interrupt()` sends an interrupt signal to a `Thread`
- `static boolean interrupted()` tests if the current `Thread` is interrupted
- `isInterrupted` tests whether a `Thread` is interrupted
- `currentThread` retrieves the current `Thread` in the current scope

THREAD STATES



THREAD PRIORITIES

- Each thread have a priority.
- Priorities are represented by a number between `1` and `10`.
- Thread Schedulers schedules the threads according to their priority (known as preemptive scheduling).
- Indeterminate because it depends on JVM specification that which scheduling it chooses.
- Predefined constants are available:
  - `MIN_PRIORITY`
  - `MAX_PRIORITY`
  - `NORM_PRIORITY`

join

- Join allows one thread to wait for another thread to complete.
- If Thread `t` is running, then the following will cause the current running `Thread` to wait until `t` is done.

```
t.join() //Wait for Thread t to finish and block
```

join THREADS

```
Thread thread1 = new Thread() {
    @Override
    public void run() {
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.format("Did two seconds on Thread %s\n", Thread.currentThrea
    }
};

thread1.start();
thread1.join(); //block
System.out.println("Thread test done");
```

DAEMON THREADS

- A daemon thread is a thread that doesn't prevent the JVM from exiting when the thread finishes
- An example of a daemon thread is the garbage collection thread
- Use `setDaemon` to set the `Thread` to a daemon `Thread`.

DAEMON THREAD EXAMPLE

```
public class DaemonRunner {
    public static void main(String[] args) {
        Thread t = new Thread() {
            @Override
            public void run() {
                while(true) {
                    try {
                        Thread.sleep(2000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("Going...");

                }
            }
        };
```

DEMO: DAEMON THREADS

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.evolutionnext.concurrent.daemon` package and open `DaemonRunner`

IMMUTABILITY

- Immutability is not having the capability of changing an object
- Any change to an object provides a copy
- Processor caching does not need to exchange state.
- Desirable in modern applications particularly in streaming

```
public class Person {
    private final String firstName;
    private final String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
```

EFFECTIVE JAVA 3RD EDITION ITEM 80

Item 80: Prefer executors, tasks, and streams to threads

The first edition of this book contained code for a simple *work queue* [Bloch01, Item 49]. This class allowed clients to enqueue work for asynchronous processing by a background thread. When the work queue was no longer needed, the client could invoke a method to ask the background thread to terminate itself gracefully after completing any work that was already on the queue. The implementation was little more than a toy, but even so, it required a full page of subtle, delicate code, of the sort that is prone to safety and liveness failures if you don't get it just right. Luckily, there is no reason to write this sort of code anymore.

By the time the second edition of this book came out, `java.util.concurrent` had been added to Java. This package contains an *Executor Framework*, which is a flexible interface-based task execution facility. Creating a work queue that is better in every way than the one in the first edition of this book requires but a single line of code:

FUTURES

FUTURE DEFINITIONS

Future def. - Future represents the lifecycle of a task and provides methods to test whether the task has completed or has been cancelled.

Future can only move forwards and once complete it stays in that state forever.

THREAD POOLS

Before setting up a future, a thread pool is required to perform an asynchronous computation. Each pool with return an `ExecutorService`.

There are a few thread pools to choose from:

- `FixedThreadPool`
- `CachedThreadPool`
- `SingleThreadExecutor`
- `ScheduledThreadPool`
- `ForkJoinThreadPool`

FIXED THREAD POOL

- "Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue."
- Keeps threads constant and uses the queue to manage tasks waiting to be run
- If a thread fails, a new one is created in its stead
- If all threads are taken up, it will wait on an unbounded queue for the next available thread

CACHED THREAD POOL

- Flexible thread pool implementation that will reuse previously constructed threads if they are available
- If no existing thread is available, a new thread is created and added to the pool
- Threads that have not been used for sixty seconds are terminated and removed from the cache

SINGLE THREAD EXECUTOR

- Creates an Executor that uses a single worker thread operating off an unbounded queue
- If a thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.

SCHEDULED THREAD POOL

- Can run your tasks after a delay or periodically
- This method does not return an `ExecutorService`, but a `ScheduledExecutorService`
- Runs periodically until `canceled()` is called.

FORK JOIN THREAD POOL

- An `ExecutorService`, that participates in *work-stealing*
- By default when a task creates other tasks (`ForkJoinTasks`) they are placed on the same on queue as the main task.
- *Work-stealing* is when a processor runs out of work, it looks at the queues of other processors and "steals" their work items.
- Not a member of Executors. Created by instantiation
- Brought up since this will be in many cases the "default" thread pool on the JVM

BASIC FUTURE BLOCKING (JDK 5)

- `get` will always block
- Would be best to ensure that there is an result ready before querying

DEMO: BASIC FUTURES

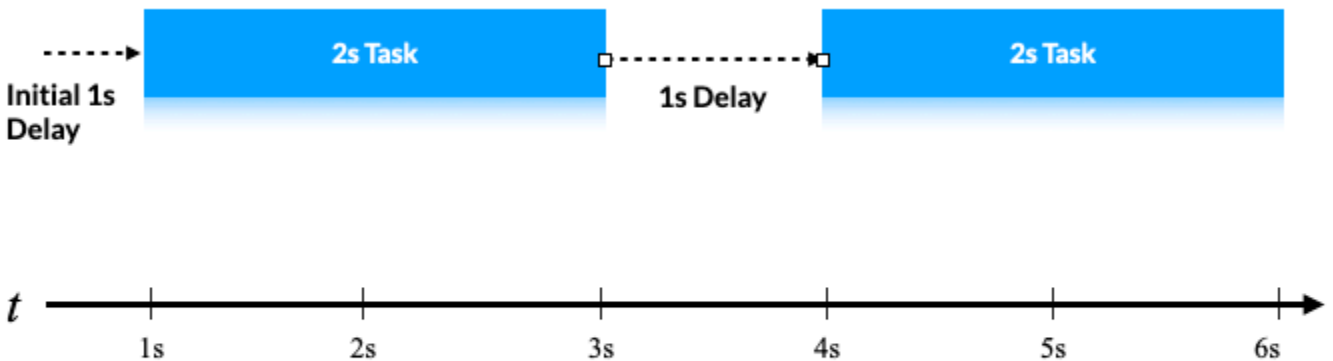
In the *java-sessions-threading-synchronizers-deep-dive/src/test/java* directory, navigate to the `com.evolutionnext.concurrency.futures` package and open *FutureBasicsTest.java*

SCHEDULED DELAY

- Creates a `ScheduledFuture<T>` that will be enabled after a given delay

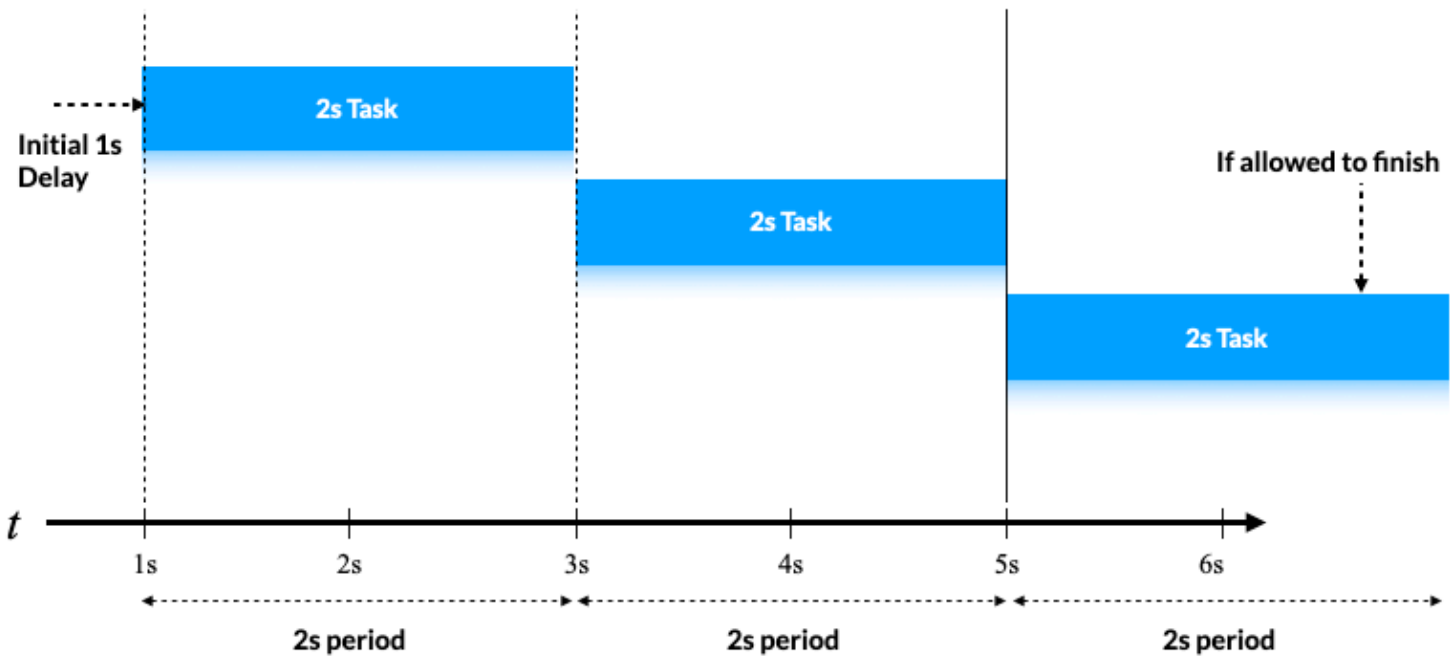
DELAY VS. RATE

DELAY [2S TASK]

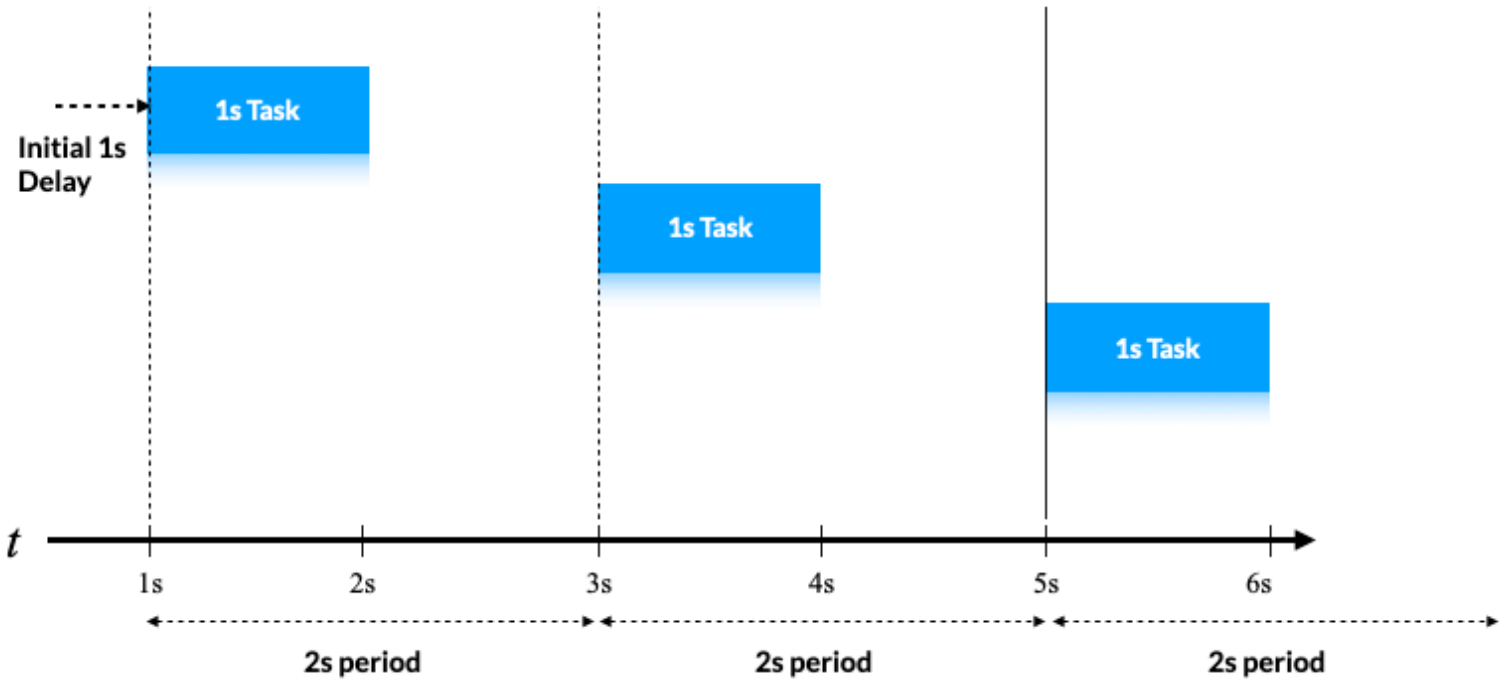


Warning: Any task can be starved by the previous task.

RATE [2S TASK]


2s Task
Initial 1s
Delay
2s Task
If allowed to finish
2s Task
t
1s
2s
3s
4s
5s
6s
2s period
2s period
2s period

RATE [1S TASK]


1s Task
Initial 1s
Delay
1s Task
1s Task
t
1s
2s
3s
4s
5s
6s
2s period
2s period
2s period

DEMO: BASIC FUTURES

In the *java-sessions-threading-synchronizers-deep-dive/src/test/java* directory, navigate to the `com.evolutionnext.concurrency.completablefutures` package and open *ScheduledFuturesTest.java*

COMPLETABLE FUTURE

- Staged Completions of Interface `java.util.concurrent.CompletionStage<T>`
- Ability to chain functions to `Future<V>`
- Analogies
  - `thenApply(…)` = map
  - `thenCompose(…)` = flatMap
  - `thenCombine(…)` = independent combination
  - `thenAccept(…)` = final processing

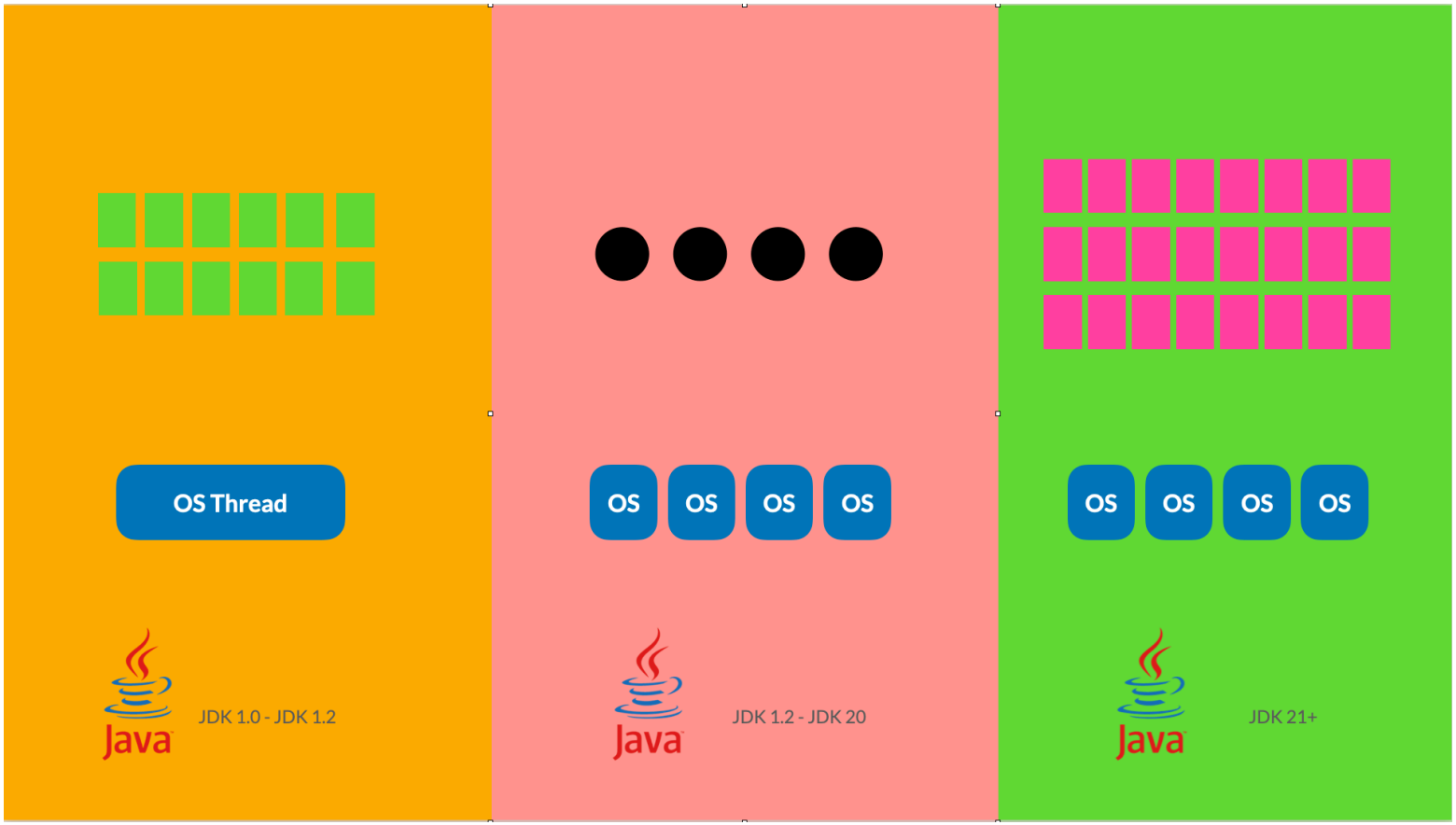
DEMO: COMPLETABLE FUTURES

In the *java-sessions-threading-synchronizers-deep-dive/src/test/java* directory, navigate to the `com.evolutionnext.concurrency.completablefutures` package and open *CompletableFuturesTest.java*

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
OS
OS
OS
OS
OS
OS
OS
OS
Java
JDK 1.0 - JDK 1.2
Java
JDK 1.2 - JDK 20
Java
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

- 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`

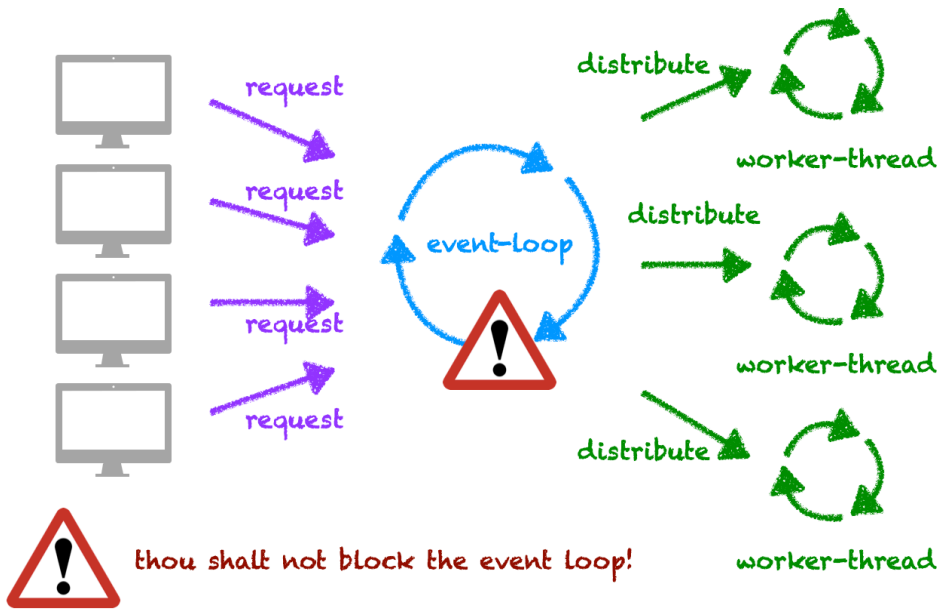
THREAD WEIGHT AND EXPENSE

- 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)
- Coroutines, Suspendable Functions (Kotlin)

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

```
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:
  - 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, 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
- **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

- 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.

DEMO: VIRTUAL THREADS

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.xyzcorp.virtualthread` package and open all the classes.

HAPPENS BEFORE GUARANTEE

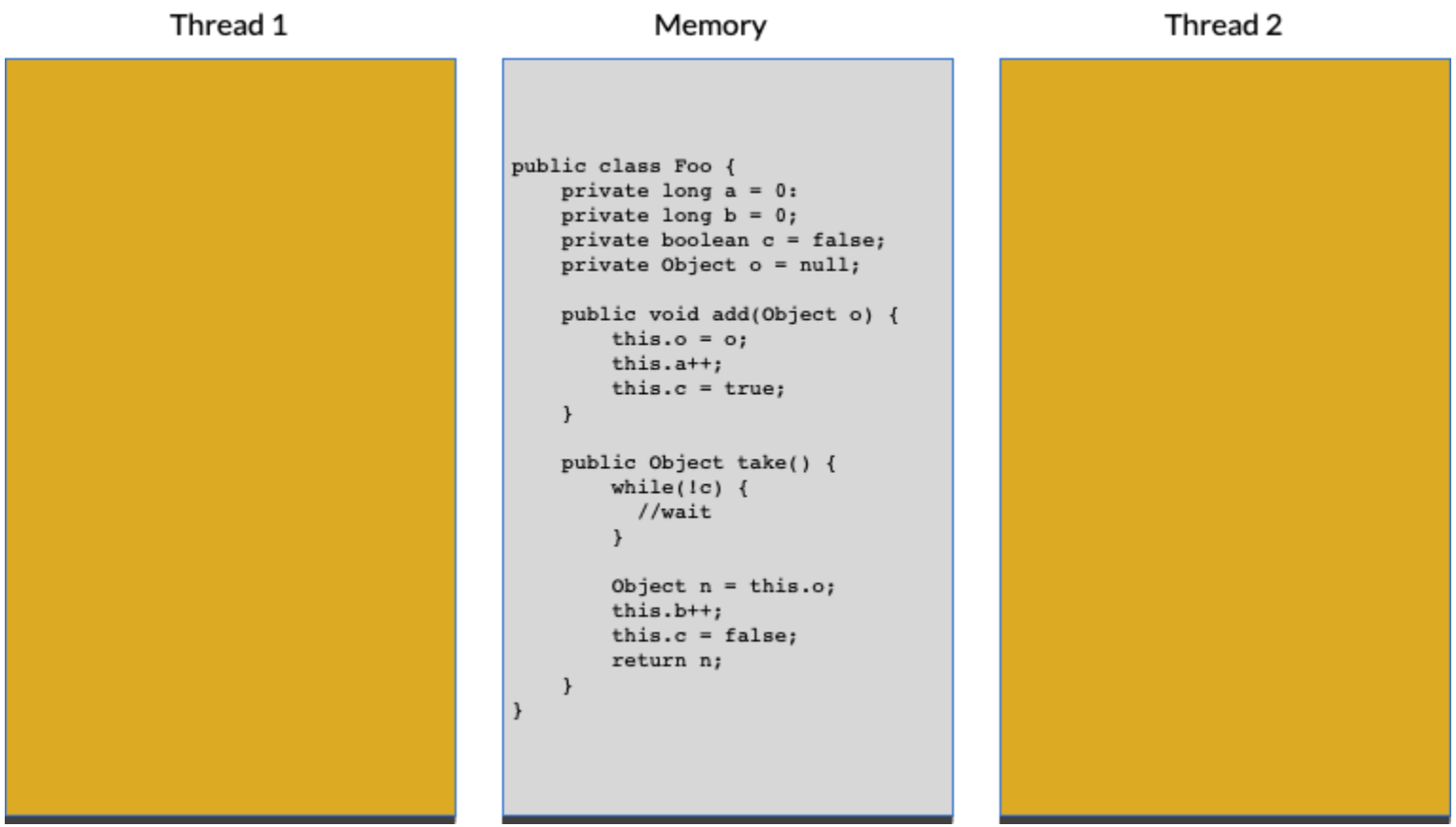
TO UNDERSTAND HAPPENS BEFORE

- The CPU can determine what can split in parallel by the CPU
- CPU can decide how to parallelize called *instruction reordering*
- CPU may process a look-ahead for, in other words, `a` and `f` can be evaluated first

```
a = b + c
d = a + e

f = g + h
i = f + j
```

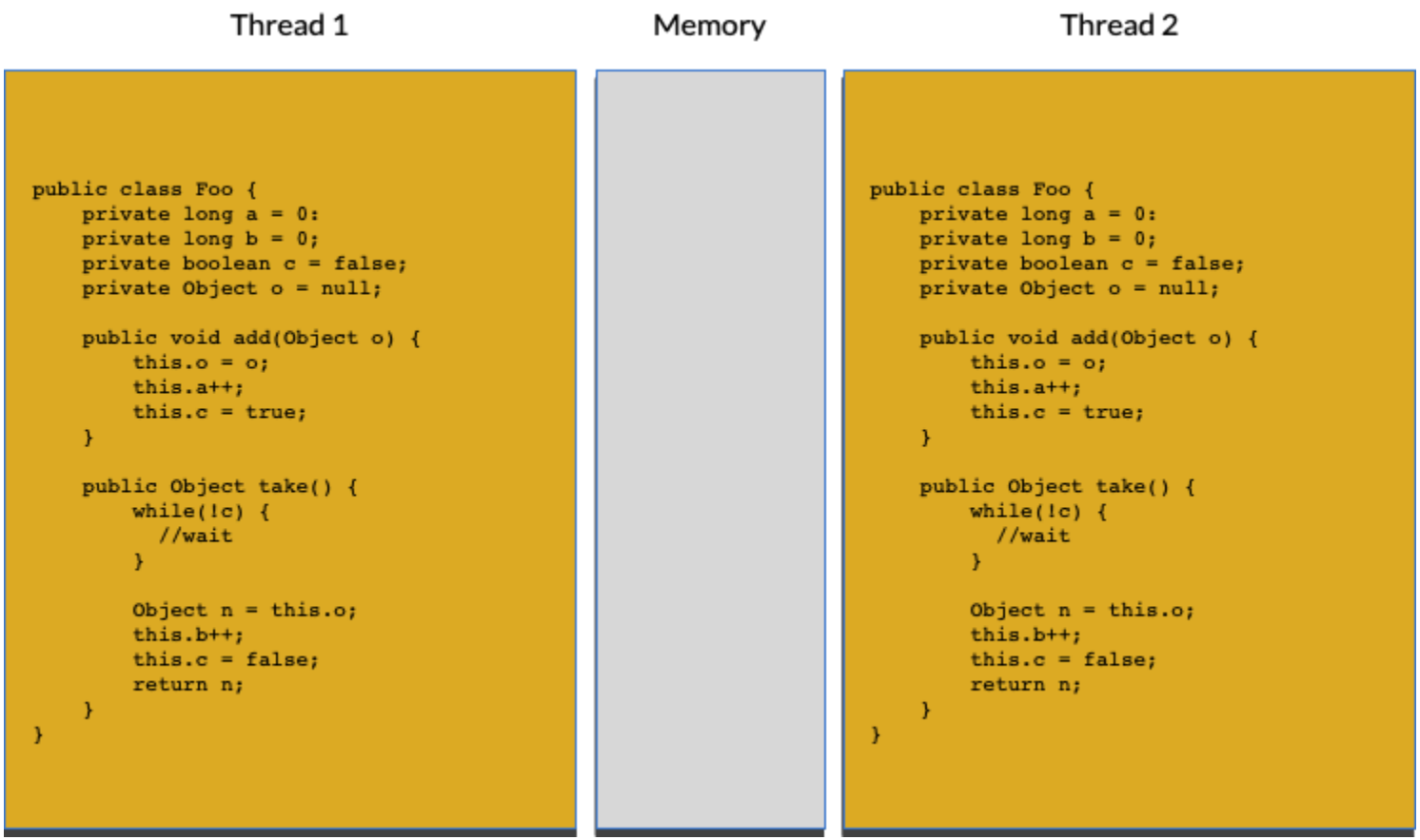
HAPPENS BEFORE GUARANTEE

The member variables are stored in memory



CPU CACHE ISSUE

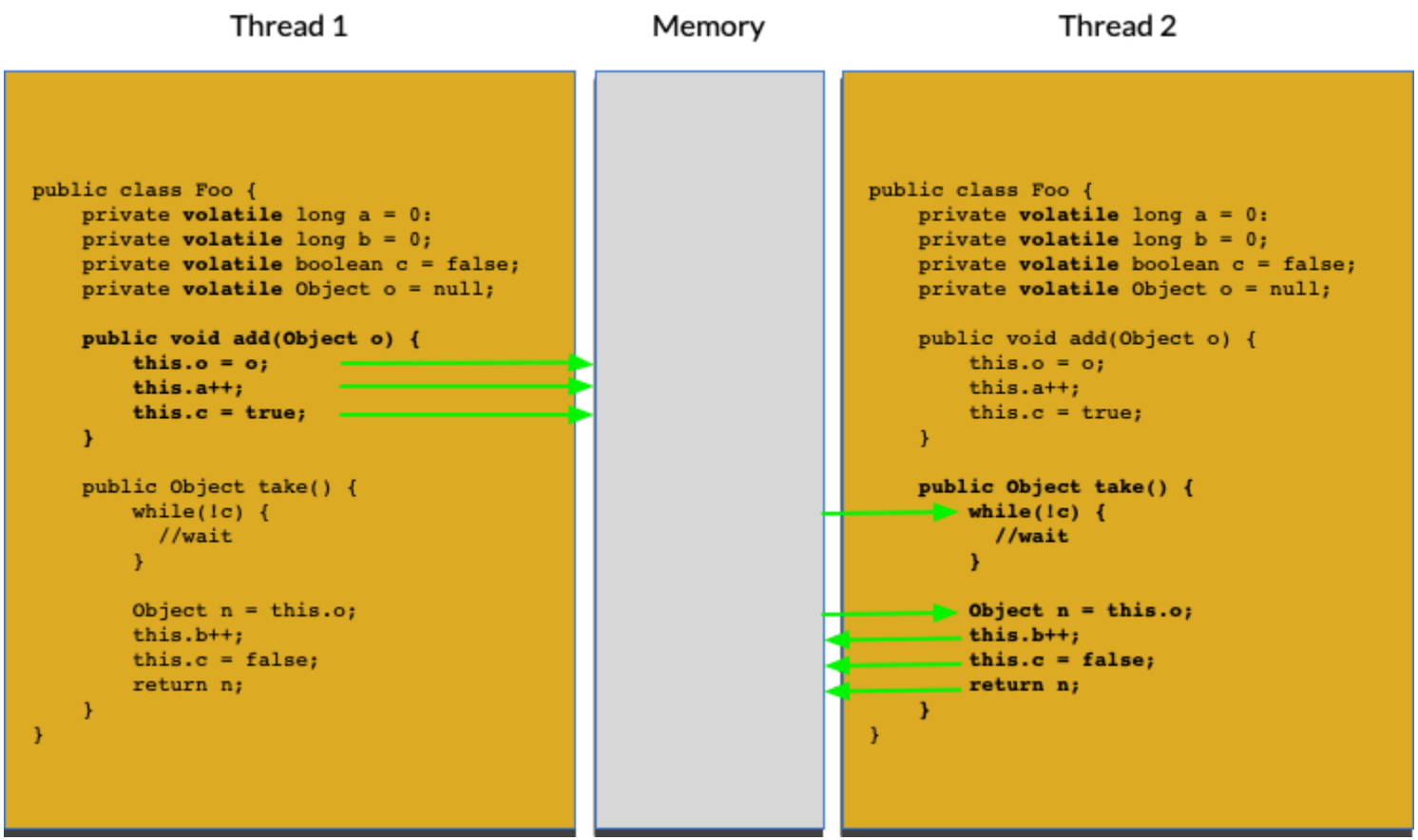
Each thread may have their own copies in either registers or cache



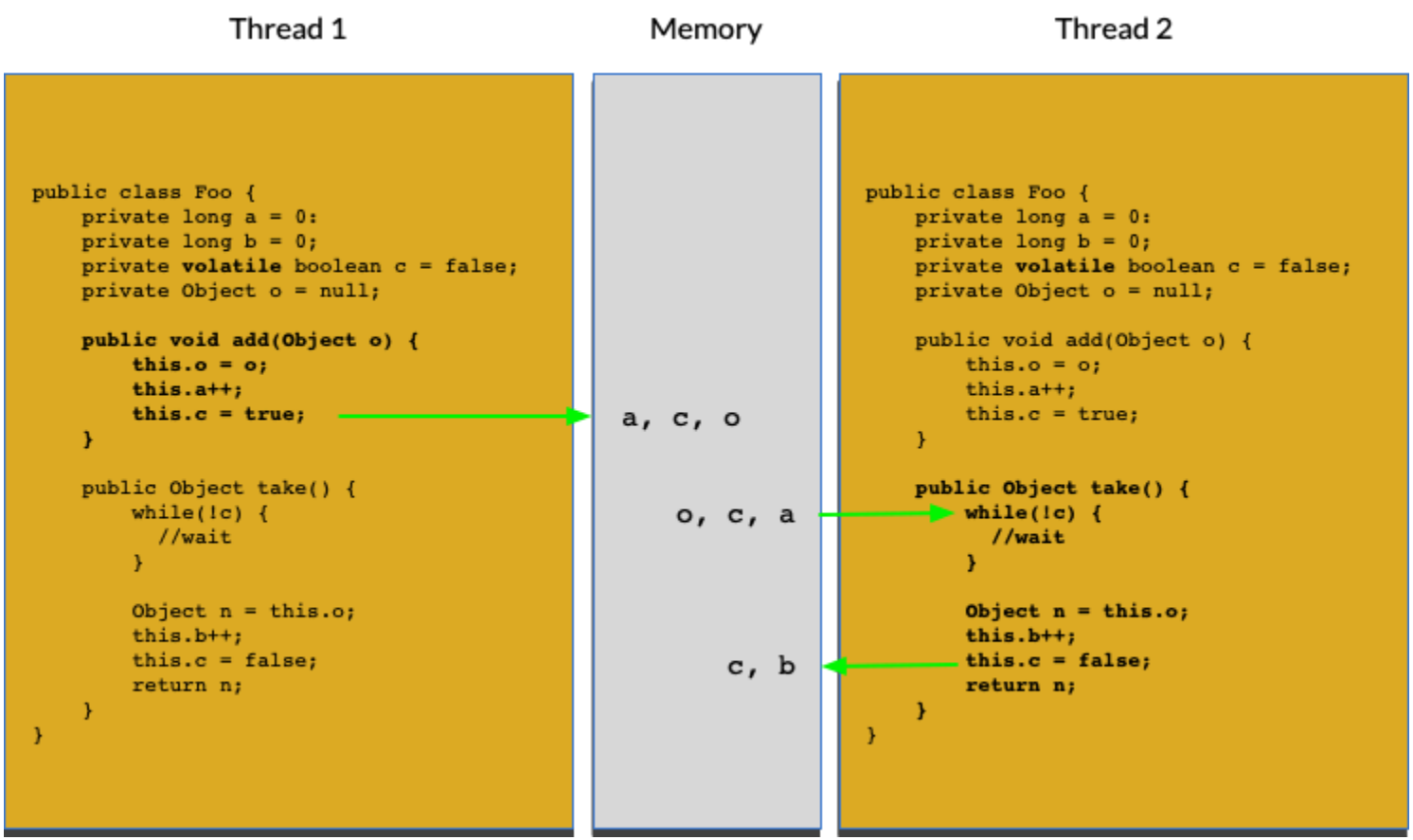
USING THE `volatile` KEYWORD

- We can denote each variable as `volatile` but that is unnecessary
- We can select one that will be `volatile`
- Here we will select all of the member variables as `volatile`
- This ensures that for multiple threads that these variable will *be visible to other threads*

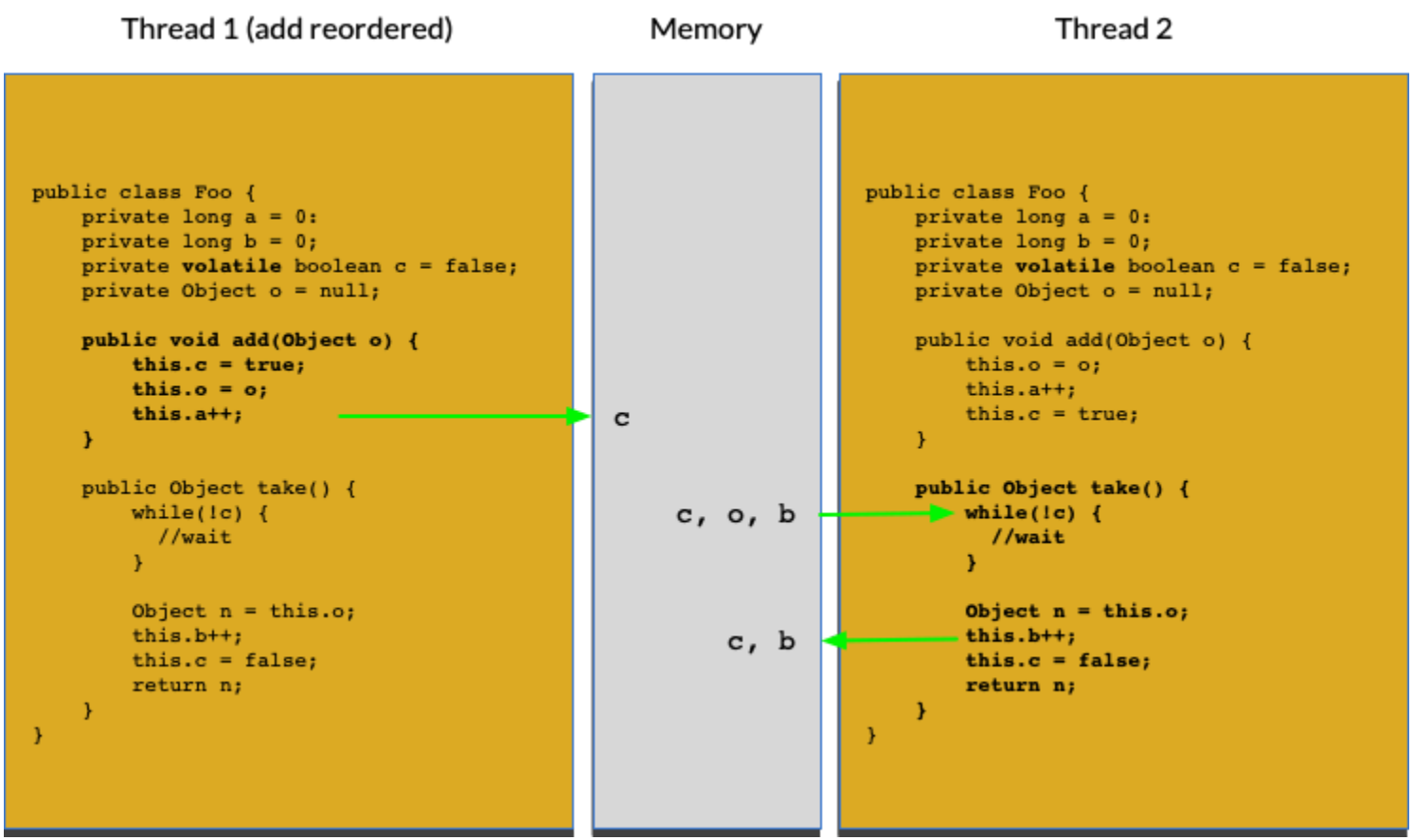
USING THE `volatile` KEYWORD



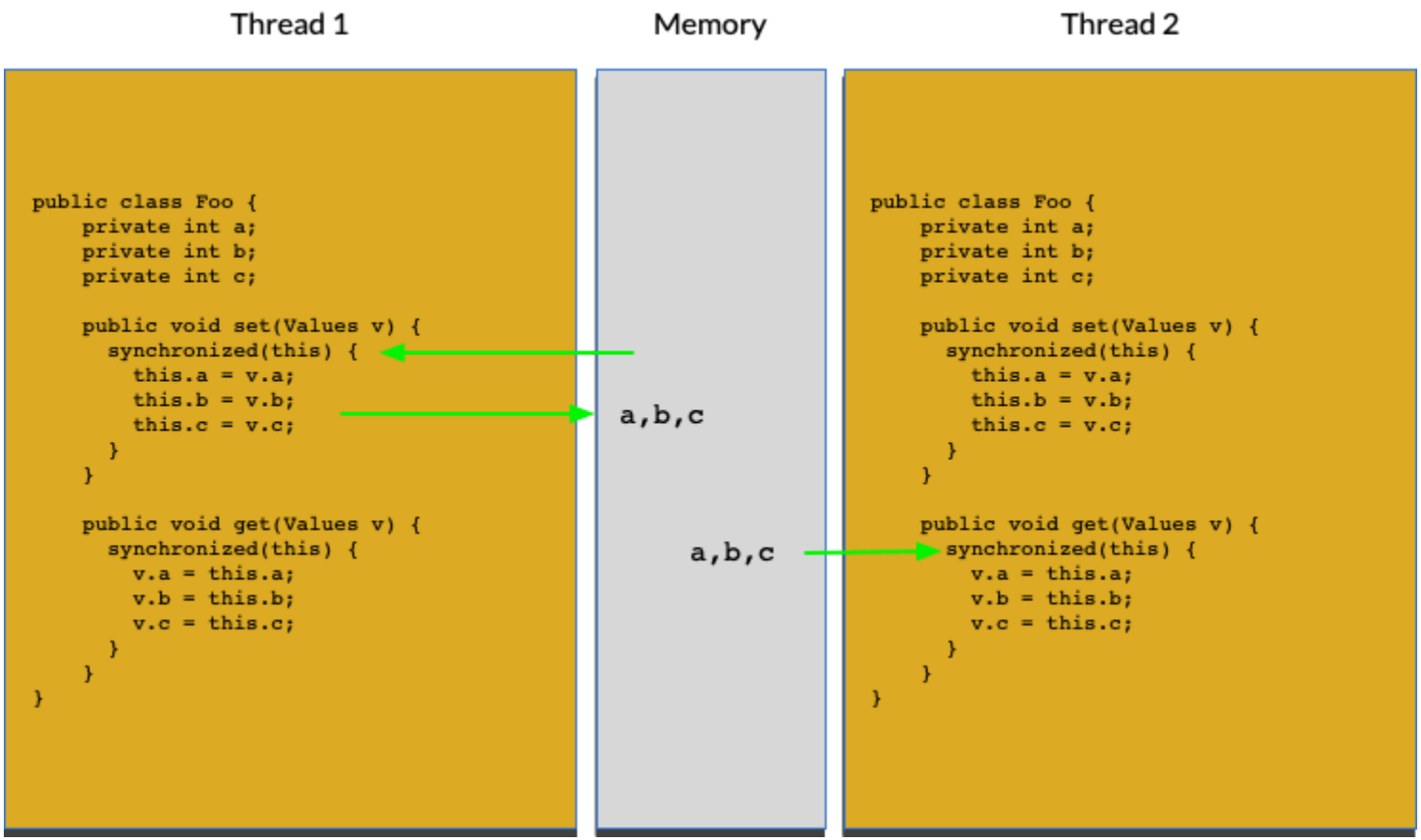
HAPPENS BEFORE GUARANTEE



ORDERING MATTERS

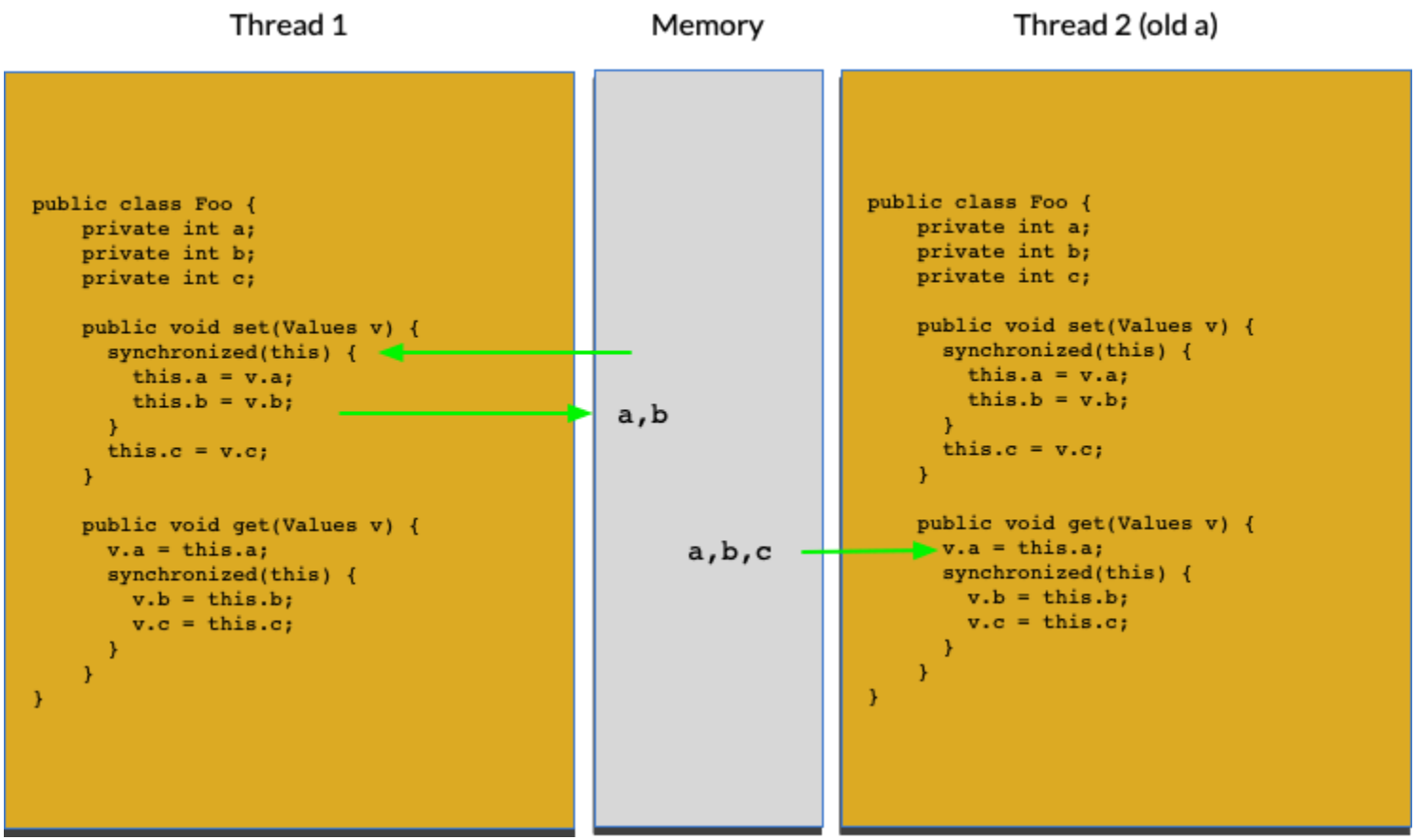


SYNCHRONIZED BLOCKS HAVE AN ORDERING GUARANTEE



IF WE LEAVE VARIABLES OUT OF THE BLOCK

Variables left out of the block are not guaranteed to be flushed into memory



volatile

`volatile`

- The Java `volatile` keyword is intended to address variable visibility problems
- By declaring a variable `volatile` all writes to that variable will be written back to main memory immediately.
- Also, all reads of the counter variable will be read directly from main memory
- Declaring a variable volatile thus guarantees the visibility for other threads of writes to that variable

FULL VISIBILITY GUARANTEES

```
public class MyClass {
    private int years;
    private int months
    private volatile int days;


    public void update(int years, int months, int days){
        this.years  = years;
        this.months = months;
        this.days   = days;
    }
}
```

Source: https://jenkov.com/tutorials/java-concurrency/volatile.html

READING WITH FULL VISIBILITY GUARANTEES

```
public class MyClass {
    private int years;
    private int months
    private volatile int days;

    public int totalDays() {
        int total = this.days; (1)
        total += months * 30;
        total += years * 365;
        return total;
    }

    public void update(int years, int months, int days){
        this.years  = years;
        this.months = months;
        this.days   = days;
```

(1) This ensures that we get all the member variables from another thread, including `months` and `years`

Source: https://jenkov.com/tutorials/java-concurrency/volatile.html

synchronized

INTRINSIC LOCKS

- An intrinsic lock is a lock that is innate within the language and provided depending on where it is used
- Often called a "monitor lock"
- Intrinsic locks can either be established on a method using the `synchronized` keyword on the method
- Intrinsic locks can also be established on a your selected object
- All threads must establish an "intrinsic lock" on the object.
- Constructors cannot be synchronized since one thread creates objects

`synchronized`

- A `synchronized` block has two parts:
  - A reference to an object that will serve as the lock
  - A block of code to be guarded by that lock.
- A `synchronized` method is a shorthand for:
  - A `synchronized` block that spans an entire method body
  - Whose lock is the object on which the method is being invoked.
  - Static synchronized methods use the Class object for the lock

WHERE CAN `synchronized` BE PLACED?

- The synchronized keyword can be used to mark four different types of blocks:
  - Instance methods
  - Static methods
  - Code blocks inside instance methods
  - Code blocks inside static methods

INTRINSIC LOCK ON A METHOD

- The following example shows an intrinsic lock that locks on the `Account` instance that is created

Applying a `synchronized` keyword on the method `deposit`

```
class Account {
  private int amount;

  public synchronized void deposit(int amount) {
     this.amount = amount;
  }
}
```

INTRINSIC LOCK ON `this`

```
class Account {
  private int amount;

  public void deposit(int amount) {
     synchronized(this) { // Synchronized on the account object
        this.amount = amount;
     }
  }
}
```

INTRINSIC LOCK ON AN EXTERNAL OBJECT

```
class Account {
  private Object lock;
  private int amount;
  public Account(Object lock) {
     this.lock = lock;
  }
  public void deposit(int amount) {
     synchronized(lock) { // Synchronized on the account object
        this.amount = amount;
     }
  }
}
```

INTRINSIC LOCK ON A `class`

```
class Account {
  private Object lock;
  private int amount;
  public Account(Object lock) {
     this.lock = lock;
  }
  //The static makes the class become the lock
  public static synchronized void deposit(int amount) {
     this.amount = amount;
  }
}
```

LIMITING THREADS ON THE SYNCHRONIZATION

```
public static MyStaticCounter{

  private static int count = 0;

  public static synchronized void add(int value){ (1)
    count += value;
  }

  public static synchronized void subtract(int value){ (2)
    count -= value;
  }
}
```

(1) The lock is the class

(2) The lock is the class

Source: https://jenkov.com/tutorials/java-concurrency/synchronized.html

DEMO: STATIC LOCKS

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.evolutionnext.concurrency.staticlock` package and open the two classes

`synchronized` ON AN INSTANCE, TWO WAYS

These two methods are essentially the same

```
public class MyClass {

    public synchronized void log1(String msg1, String msg2){
       log.writeln(msg1);
       log.writeln(msg2);
    }


    public void log2(String msg1, String msg2){
       synchronized(this){
          log.writeln(msg1);
          log.writeln(msg2);
       }
    }
  }
```

Source: https://jenkov.com/tutorials/java-concurrency/synchronized.html

synchronized ON A STATIC METHOD TWO WAYS

```
public class MyClass {

  public static synchronized void log1(String msg1, String msg2){
     log.writeln(msg1);
     log.writeln(msg2);
  }

  public static void log2(String msg1, String msg2){
     synchronized(MyClass.class){
        log.writeln(msg1);
        log.writeln(msg2);
     }
  }
}
```

Source: https://jenkov.com/tutorials/java-concurrency/synchronized.html

`synchronized` BLOCK REENTRANCE

- Once a thread has entered a `synchronized` block the thread is said to "hold the lock" on the monitoring object the synchronized block is synchronized on.
- If the thread calls another method which calls back to the first method with the `synchronized` block inside, the thread holding the lock can reenter the `synchronized` block
- It is not blocked just because a thread (itself) is holding the lock.
- Only if a different thread is holding the lock. Look at this example:

synchronized BLOCK REENTRANCE EXAMPLE

```
public class MyClass {

  List<String> elements = new ArrayList<String>();

  public void count() {
    if(elements.size() == 0) {
        return 0;
    }
    synchronized(this) {
       elements.remove();
       return 1 + count();
    }
  }
}
```

Source: https://jenkov.com/tutorials/java-concurrency/synchronized.html

wait, notify,
notifyAll

`wait`, `notify`, `notifyAll`

- `wait()` - Causes the current thread to block in the given object until awakened by a `notify()` or `notifyAll()`.
- `notify()`
  - Causes a randomly selected thread waiting on this object to be awakened.
  - It must then try to regain the intrinsic lock.
  - If the “wrong” thread is awakened, your program can deadlock.
- `notifyAll()`
  - Causes all threads waiting on the object to be awakened
  - Each will then try to regain the monitor lock. Hopefully one will succeed.

DEMO: GUARDED BLOCKS

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.evolutionnext.concurrency.guardedblocks` package and open the two classes

DEMO: GUARDED BLOCKS ADDITIONAL

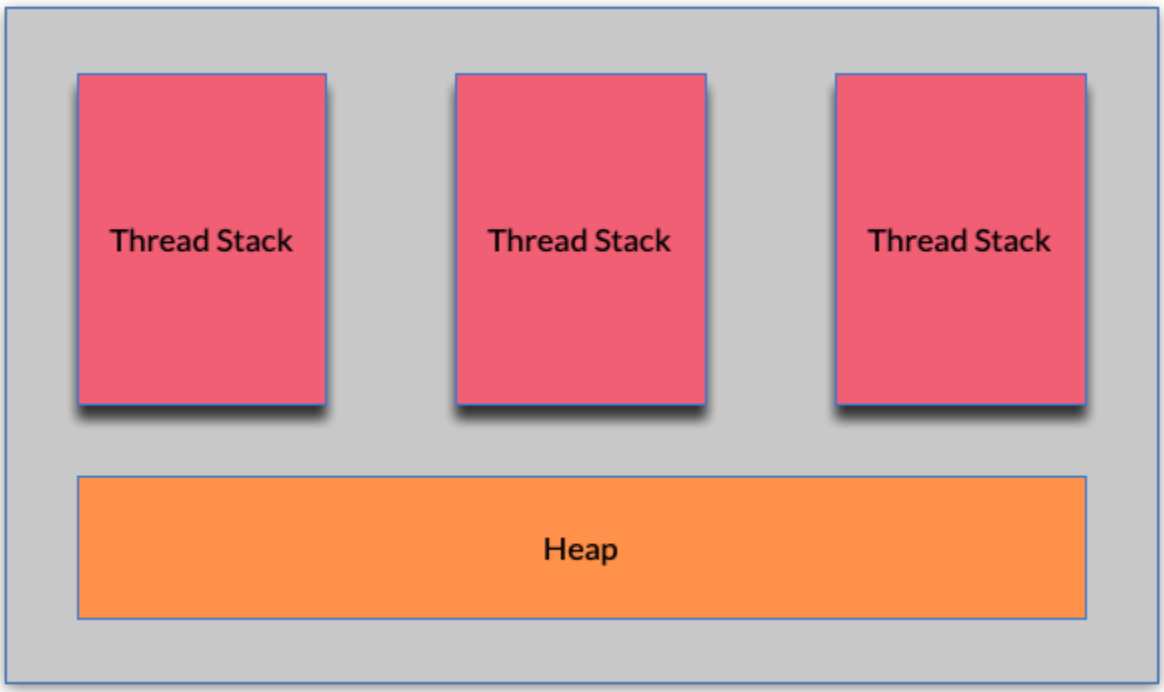
In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.evolutionnext.concurrency.primitives` package and open the `BankAccount`

JAVA MEMORY MODEL

ABOUT THE MEMORY MODEL

- Still the same Memory Model since Java 5
- Comes in two components:
  - Stack (or Thread Stack)
  - Heap

MEMORY DIAGRAM



THREAD STACK

- Every thread in the JVM has its own thread stack.
- Thread stacks contains information about what methods the thread has called to reach the current point of execution.
- Referred as the "call stack", which if called repeatedly results in `StackOverflowError`
- As the thread executes its code, the call stack changes.

THREAD STACK INTERNALS

- Thread stack also contains all local variables for each method being executed internal to the stack
- A Thread can only access the local variables in it’s stack
- Local variables in a Thread are not visible to another Thread
- Each thread will only have access to it’s copy, but copies can be sent to other threads

JVM HEAP

- Contains the objects that are used by the Threads in the Stack
- Objects also include wrappers `Integer`, `Long`, etc.

JVM PRIMITIVES AND OBJECTS

- If the variable is a primitive type, then it is stored in the Thread stack
- Object References
  - If the variable is a object reference, the variable is stored in the Thread Stack
  - The object that the object refers to is stored in the Heap and undergoes heap lifecycle
  - The object's member variable are also stored on the heap
  - `static` class variables are stored on the heap with the class definition

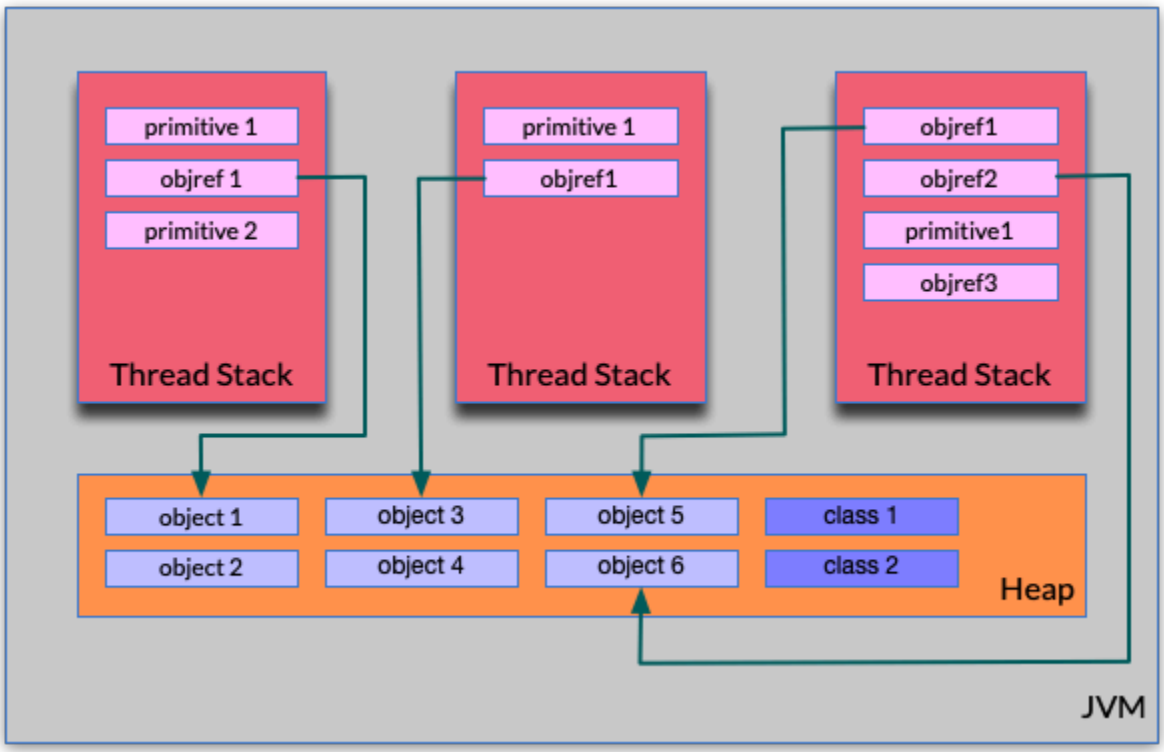
HEAP OBJECTS

- Objects on the Heap can be accessed by any Thread Stack
- Two references on the Thread Stack can access:
  - Object reference
  - Object's class reference
  - Object's member variables
  - Object's class `static` members

MEMORY MODEL AND PHYSICAL MEMORY

- Hardware memory architecture does not distinguish between thread stacks and heap
- Both the thread stack and the heap are located in main memory
- Parts of the thread stacks and heap may sometimes be present in CPU caches and in internal CPU registers

JAVA MEMORY DETAILED



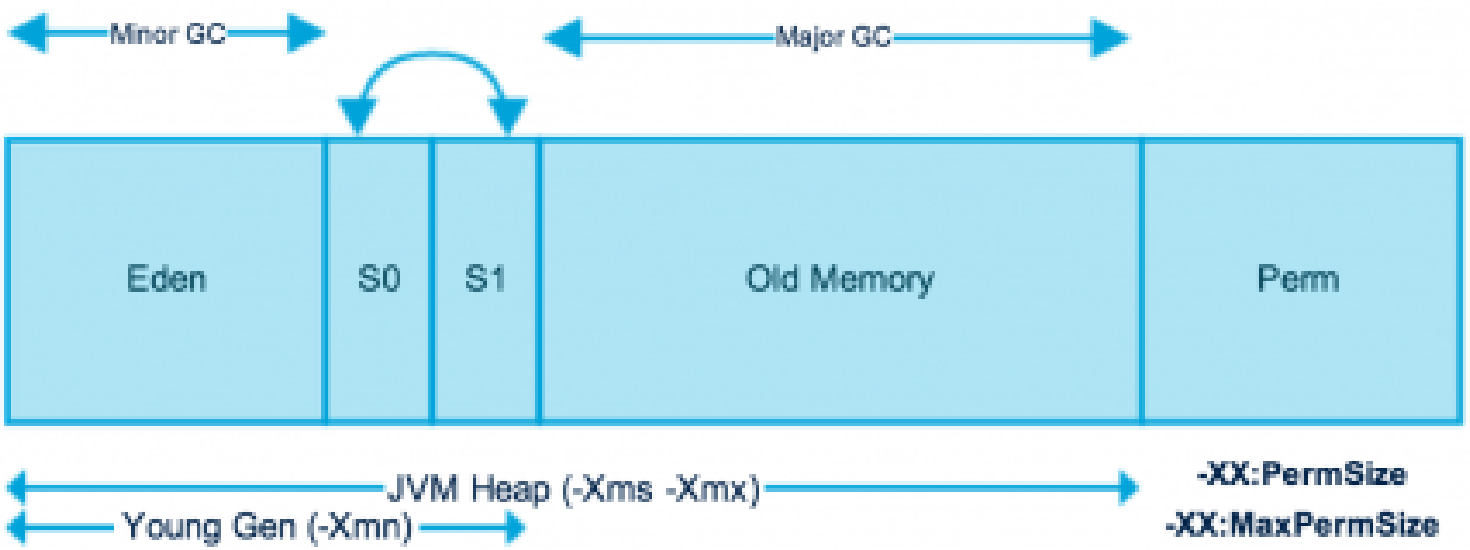
OBJECTS AND GARBAGE COLLECTION

- Once objects are never used in the Heap they are *Garbage Collected*
- Occasionally, from the JVM
  - Memory is reclaimed by deleting garbage
  - Memory is compacted by *defragmenting*

HEAPS ARE BROKEN INTO SUBHEAPS

Subheap	Description	Age in GC Cycles
Eden	Newly Created Objects	0
Young	New Objects that live beyond the first GC cycle	1
Survivor	Objects that survive a few GC Cycles, Gets moved between survivor areas: S1 → S2, S2 → S1	< 15
Tenured	Objects that survive longer	> 15
PermGen	Method definitions & `static` declarations	No age allocation

SUBHEAP DIAGRAM



VIRTUAL STACKS AND HEAP AND PHYSICAL RAM

ISSUES WITH THE JVM

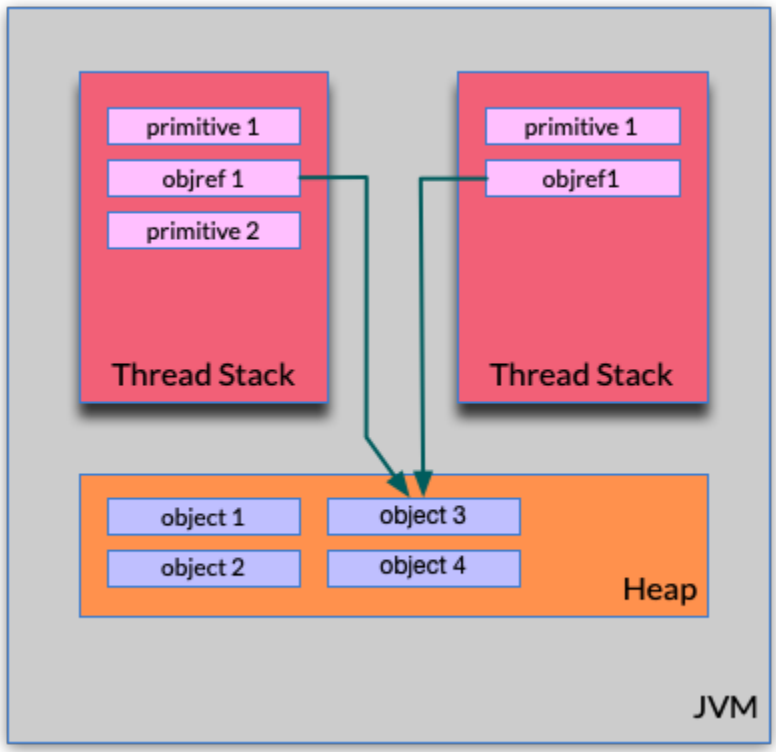
There are two issues in regard to physical memory when running the JVM

- Visibility of thread updates (writes) to shared variables.
- Race conditions when reading, checking and writing shared variables.

VISIBILITY OF SHARED OBJECTS

- If two or more threads share an object, without proper mitigation, updates to the shared object made by one thread may not be visible to others
- Proper mitigation techniques:
  - `volatile`
  - `synchronized`

SHARED OBJECTS



THE `Lock` `interface`

THE `Lock interface` DEFINED

- `Lock` implementations provide more extensive locking operations than can be obtained using `synchronized` methods and statements.
- They allow more flexible structuring, may have quite different properties, and may support multiple associated `Condition` objects.
- A `lock` is a tool for controlling access to a shared resource by multiple threads
- Commonly, a lock provides exclusive access to a shared resource: only one thread at a time can acquire the lock and all access to the shared resource requires that the lock be acquired first.
- Some locks may allow concurrent access to a shared resource, such as the read lock of a `ReadWriteLock`

GENERAL USE OF A Lock

Example of using an explicit `Lock`

```
Lock l = ...; (1)
l.lock(); (2)
try {
  // access the resource protected by this lock (3)
} finally {
  l.unlock(); (4)
}
```

1. Create or instantiate the `Lock`
2. Obtain the `Lock` using `lock()`
3. Perform what you require
4. `unlock()` the `Lock` so that others can use the `Lock`

ReentrantLock

`ReentrantLock` DEFINITION

- A `RentrantLock`:
  - Mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using `synchronized` methods and statements
  - But has more extended capabilities
  - 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()`

CREATING A `ReentrantLock`

- 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; often much slower) than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation. Note however, that fairness of locks does not guarantee fairness of thread scheduling. Thus, one of many threads using a fair lock may obtain it multiple times in succession while other active threads are not progressing and not currently holding the lock. Also note that the untimed tryLock method does not honor the fairness setting. It will succeed if the lock is available even if other threads are waiting.

EXAMPLE OF A `ReentrantLock`

- It is recommended practice to always immediately follow a call to lock with a `try` block most typically in a before/after construction
- For example:

```
class X {
  private final ReentrantLock lock = new ReentrantLock();
  // ...

  public void m() {
    lock.lock();  // block until condition holds
    try {
      // ... method body
    } finally {
      lock.unlock();
    }
  }
}
```

DEMO: `ReentrantLock`

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.evolutionnext.concurrency.reentrantlocks` package and open the `BankAccountReentrantLock`

ReentrantReadWriteLock

`ReentrantReadWriteLock` DEFINITION

- A Read/Write Lock, separation of concerns
- Supports a fairness policy

FAIRNESS POLICIES

Non-fair mode (default)

The order of entry to the read and write lock is unspecified, subject to reentrancy constraints. A nonfair lock that is continuously contended may indefinitely postpone one or more reader or writer threads, but will normally have higher throughput than a fair lock

Fair mode

Threads contend for entry using an approximately arrival-order policy. When the currently held lock is released either the longest-waiting single writer thread will be assigned the write lock, or if there is a group of reader threads waiting longer than all waiting writer threads, that group will be assigned the read lock

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html

DEMO: `ReentrantReadWriteLock`

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.evolutionnext.concurrency.reentrantlocks` package and open the class `BankAccountReentrantReadWriteLock`

StampedLock

`StampedLock`

- A capability-based lock with three modes for controlling read/write access
- The state of a `StampedLock` consists of a version and mode
- Lock acquisition methods return *a stamp* that represents and controls access with respect to a lock state
- "try" versions of these methods may instead return the special value zero to represent failure to acquire access
- Lock release and conversion methods require stamps as arguments, and fail if they do not match the state of the lock

THE THREE MODES OF `StampedLock`

- **Writing.** Method `writeLock()` possibly blocks waiting for exclusive access, returning a stamp that can be used in method `unlockWrite(long)` to release the lock. Untimed and timed versions of `tryWriteLock` are also provided. When the lock is held in write mode, no read locks may be obtained, and all optimistic read validations will fail.
- **Reading.** Method `readLock()` possibly blocks waiting for non-exclusive access, returning a stamp that can be used in method `unlockRead(long)` to release the lock. Untimed and timed versions of `tryReadLock` are also provided.
- **Optimistic Reading.** Method `tryOptimisticRead()` returns a non-zero stamp only if the lock is not currently held in write mode. Method `validate(long)` returns true if the lock has not been acquired in write mode since obtaining a given stamp.

FURTHER NOTES ON `StampedLock` OPTIMISTIC READING

- This mode can be thought of as an extremely weak version of a read-lock, that can be broken by a writer at any time.
- The use of optimistic mode for short read-only code segments often reduces contention and improves throughput.
- However, its use is inherently fragile.
- Optimistic read sections should only read fields and hold them in local variables for later use after validation.

DEMO: `StampedLock`

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.evolutionnext.concurrency.reentrantlocks` package and open the `BankAccountStampedLock`

DEADLOCKS

DEADLOCKS

- Deadlock describes a situation where two or more threads are blocked forever, waiting for each other
- This is a common occurrence when multiple locks or mutexes are waiting for one another

I Am Devloper
@iamdevloper

interviewer: if you can explain what deadlocks are, I'll hire you

me: hire me, and I'll explain deadlocks

5:42 PM · Jun 21, 2022 · Twitter Web App

DEADLOCK SITUATIONS

1. Mutual exclusion: At least one resource must be held in a non-shareable mode; that is, only one process at a time can use the resource. Otherwise, the processes would not be prevented from using the resource when necessary. Only one process can use the resource at any given instant of time.

2. Hold and wait or resource holding: a process is currently holding at least one resource and requesting additional resources which are being held by other processes.

3. No preemption: a resource can be released only voluntarily by the process holding it.

4. Circular wait: each process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource. In general, there is a set of waiting processes, P = {P1, P2, …, PN}, such that P1 is waiting for a resource held by P2, P2 is waiting for a resource held by P3 and so on until PN is waiting for a resource held by P1.

https://en.wikipedia.org/wiki/Deadlock

DEMO: DEADLOCKS

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.evolutionnext.concurrency.deadlock` package and open the `Friend` class.

LIVELOCKS

LIVELOCKS

- A thread often acts in response to the action of another thread.
- If the other thread's action is also a response to the action of another thread, then livelock may result.
- As with deadlock, livelocked threads are unable to make further progress.
- However, the threads are not blocked — they are simply too busy responding to each other to resume work.
- This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass.
- Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so…

DEMO: LIVELOCKS

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.evolutionnext.concurrency.livelock` package and open the classes.

RACE CONDITIONS

RACE CONDITIONS

- A race condition is a concurrency problem where two threads vie for the same resource or set of resources
- This leads to unexpected behaviors, this is perhaps one of the most common issues with concurrency
- A race condition can be difficult to reproduce and debug because the end result is nondeterministic and depends on the relative timing between interfering threads
- Problems of this nature can therefore disappear when running in debug mode, adding extra logging, or attaching a debugger.
- A bug that disappears like this during debugging attempts is often referred to as a "Heisenbug".
- It is therefore better to avoid race conditions by careful software design.

Some elements from source: https://en.wikipedia.org/wiki/Race_condition

RACE CONDITION ANALOGY

- You are planning to go to a movie at 5 pm.
- You inquire about the availability of the tickets at 4 pm.
- The representative says that they are available.
- You relax and reach the ticket window 5 minutes before the show.
- I’m sure you can guess what happens: it’s a full house.

Source: https://stackoverflow.com/questions/34510/what-is-a-race-condition

IDEAL CONDITION EXAMPLE

- Assume that two threads each increment the value of a global integer variable by 1.
- Ideally, the following sequence of operations would take place

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Source: https://en.wikipedia.org/wiki/Race_condition

RACE CONDITION EXAMPLE

- In this case, the final value is 1 instead of the expected result of 2.
- This occurs because here the increment operations are not *mutually exclusive*.
- Mutually exclusive operations are those that cannot be interrupted while accessing some resource such as a memory location

Source: https://en.wikipedia.org/wiki/Race_condition

RACE CONDITION DIAGRAM

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Source: https://en.wikipedia.org/wiki/Race_condition

DEMO: RACE CONDITIONS

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.evolutionnext.concurrency.racecondition` package and open all the classes.

STARVATION

STARVATION

- Starvation and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.
- Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress.
- This happens when shared resources are made unavailable for long periods by "greedy" threads.
- For example, suppose an object provides a `synchronized` method that often takes a long time to return.
- If one thread invokes this method frequently, other threads that also need frequent `synchronized` access to the same object will often be blocked

https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html

DEMO: STARVATION

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.evolutionnext.concurrency.starvation` package and open all the classes.

SEMAPHORE

SEMAPHORE

- A Semaphore manages a set of virtual permits; the initial number of permits is passed to the Semaphore constructor
- Activities can acquire permits (as long as some remain) and release permits when they are done with them
- If no permit is available, acquire blocks until one is (or until interrupted or the operation times out)
- The release method returns a permit to the semaphore.

Source: Java Concurrency in Practice Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea

BINARY SEMAPHORES ARE MUTEXES

- A degenerate case of a counting semaphore is a binary semaphore, a Semaphore with an initial count of one
- A binary semaphore can be used as a mutex with nonreentrant locking semantics; whoever holds the sole permit holds the mutex

Source: Java Concurrency in Practice Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea

COUNTING SEMAPHORES

- Counting semaphores are used to control the number of activities that can access a certain resource or perform a given action at the same time [CPJ 3.4.1]
- Counting semaphores can be used to implement resource pools (like database connections) or to impose a bound on a collection

Source: Java Concurrency in Practice Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea

DEMO: SEMAPHORES

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.evolutionnext.concurrency.semaphore` package and open the `BoundedHashSet` class

ATOMIC

ATOMIC VARIABLES

- Based on a branch of research focused on creating non-blocking algorithms for concurrent environments
- Algorithms exploit low-level atomic machine instructions such as compare-and-swap (CAS)

COMPARE AND SWAP

A typical CAS works on three operands

1. The memory location on which to operate (M)
2. The existing expected value (A) of the variable
3. The new value (B) which needs to be set

Compare and Swap Definition

Atomically set the value in **M** to **B**, but only if the existing value in **M** matches **A**, otherwise no action is taken - All as a single operation

WHAT IS DIFFERENT?

- No Threads are suspended
- They are informed that they have not completed the update
- You may have to handle the situation where a CAS operation did not succeed.
  - Retry
  - Do Nothing

COMMON ATOMIC VARIABLES

- `AtomicInteger`
- `AtomicLong`
- `AtomicBoolean`
- `AtomicReference`

COMMON METHODS IN ATOMIC VARIABLES

- `get()` - gets the value from the memory, like `volatile`
- `set()` - write a value to memory, like `volatile`
- `lazySet()` - eventually writes the value to memory, maybe reordered with subsequent relevant memory operations
- `compareAndSet()` - compare value, if maintained, write new value. returns `true` when it succeeds, else `false`

DEMO: ATOMIC VARIABLES

In the *java-sessions-threading-synchronizers-deep-dive/src/test/java* directory, navigate to the `com.evolutionnext.concurrency.atomic` package and open the tests included.

DEMO: CONCURRENTSTACK

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.evolutionnext.concurrency.atomic` package and open *ConcurrentStack.java*

ACCUMULATORS

LongAdder

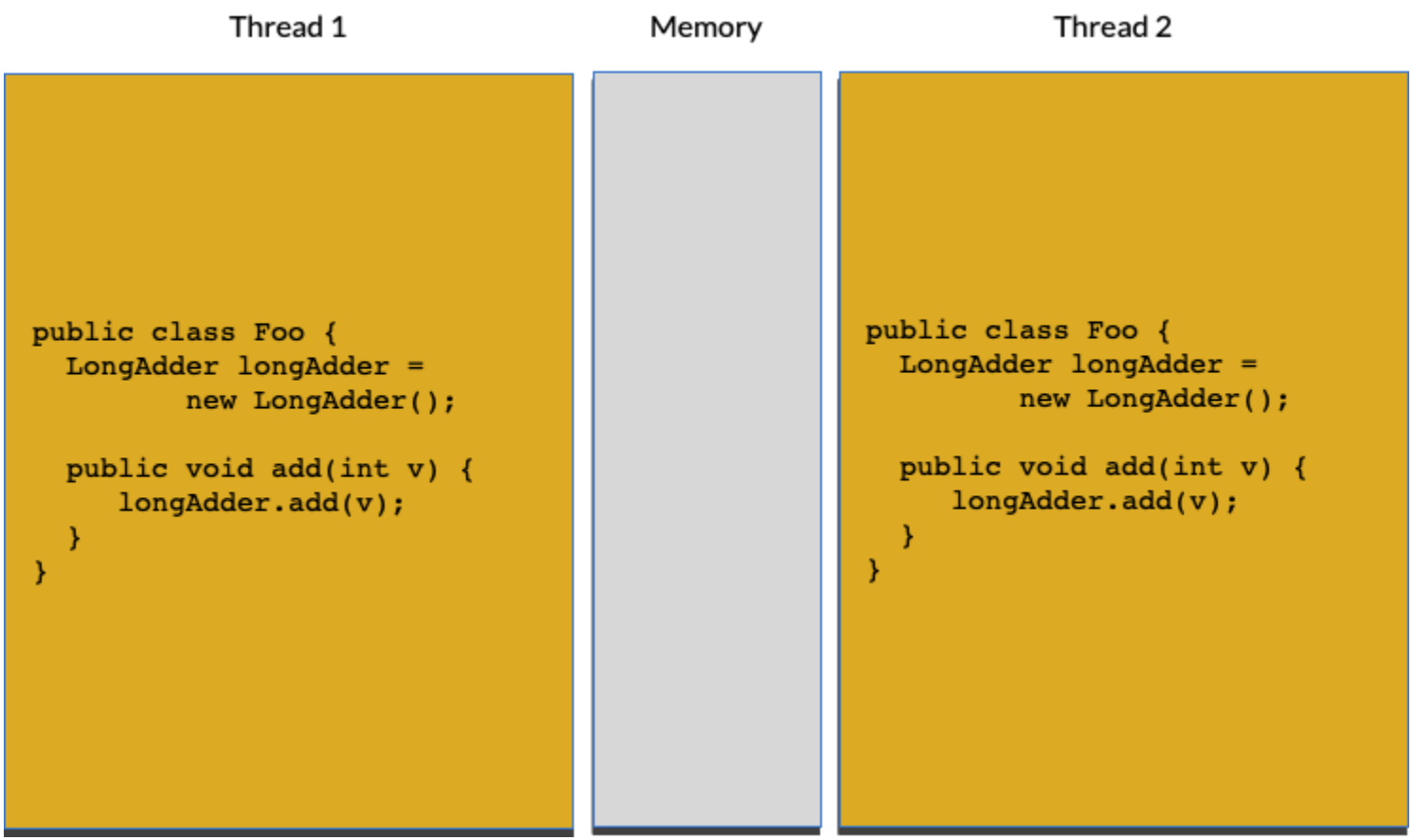
- One or more variables that together maintain an initially zero long sum.
- When updates like method `add(long)` are contended across threads, the set of variables may grow dynamically to reduce contention.
- Method `sum()` (or, equivalently, `longValue()`) returns the current total combined across the variables maintaining the sum
- Preferrable to `AtomicLong` for accumulation
- Extends `Number`

PERFORMANCE OF LongAdder

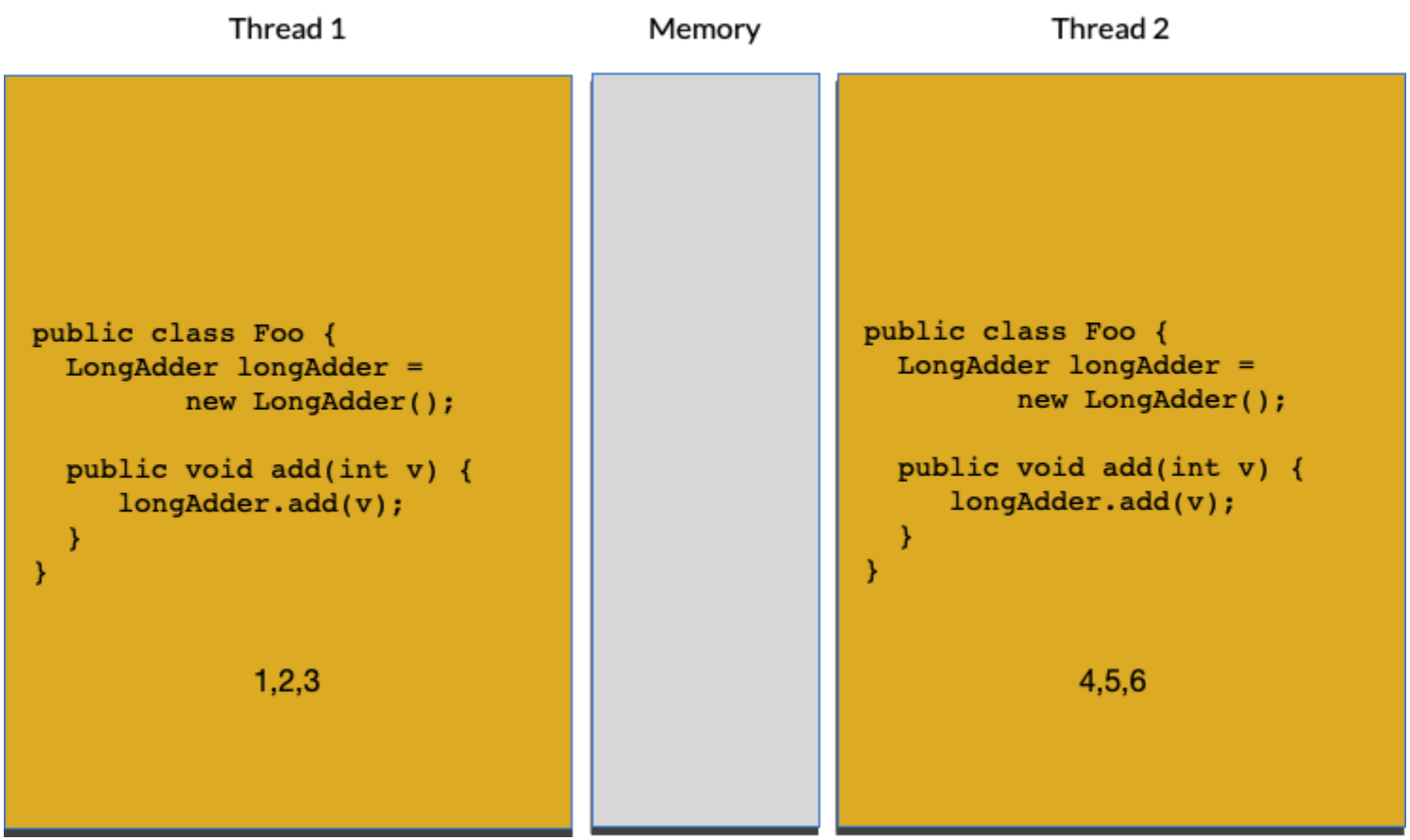
- Under low update contention, the two classes have similar characteristics
- But under high contention, expected throughput of this class is significantly higher
  - At the expense of higher space consumption

LONG ADDER BY EXAMPLE

In `LongAdder` each thread is maintaining in it's own count

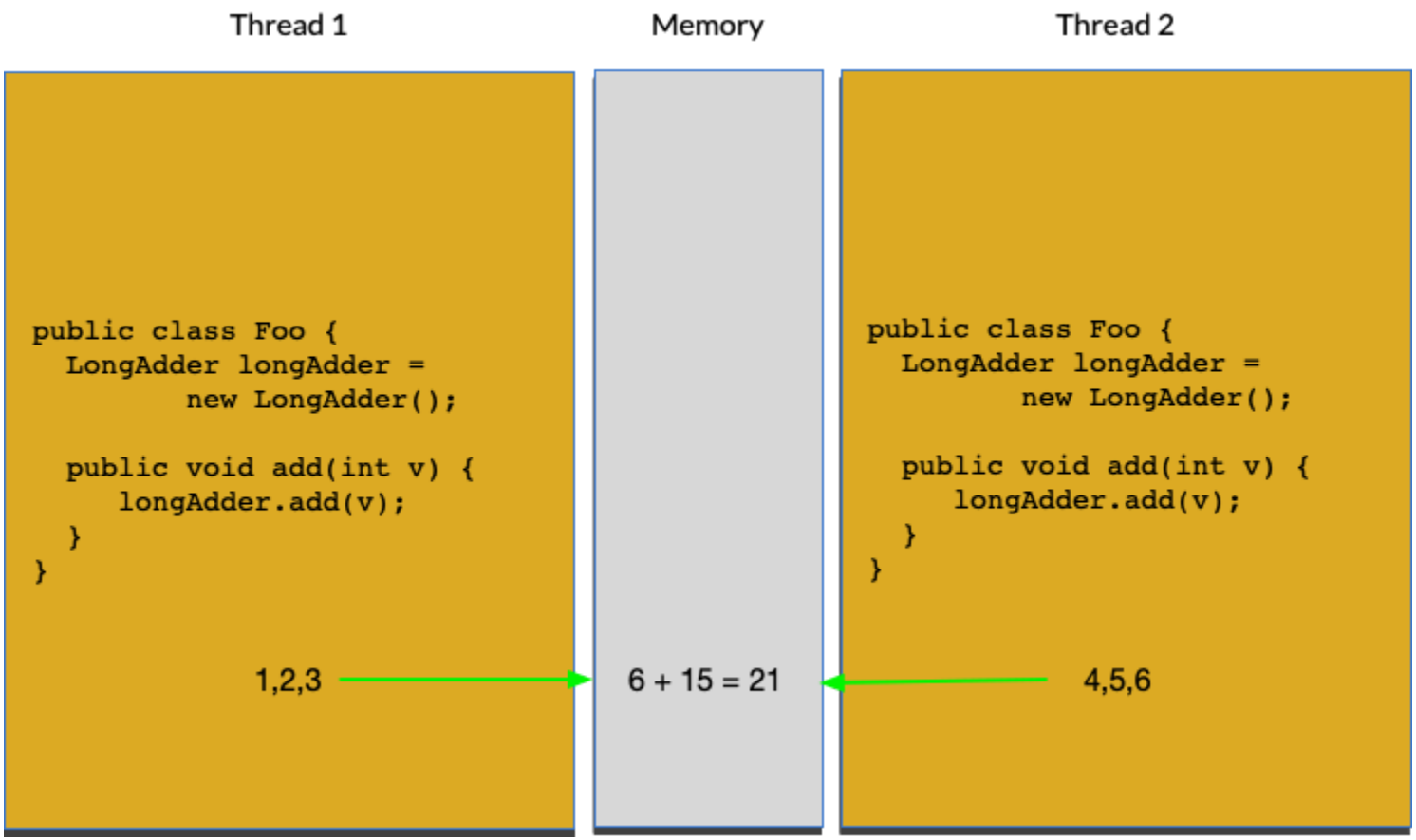


LOCALLY ADDING THE NUMBERS



COMBINING THE NUMBER INTO MAIN MEMORY

Since Adding is Commutative, we can bring the values in together when done



`DoubleAdder`

- Same as `LongAdder` but for `Double` floating numbers

`LongAccumulator`

- `LongAccumulator` is a General Form of the `LongAdder`, thread safe, where you can apply your own function
- Similar to `reduce` in functional programming
- Pass each number that is required to accumulate with the `accumulate` method

`DoubleAccumulator`

- Same as `LongAccumulator` but for `Double` floating numbers

DEMO: ADDERS AND ACCUMULATORS

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.evolutionnext.concurrency.accumulators` package and the files in the package

COUNTDOWN LATCH

COUNTDOWN LATCH DESCRIBED

- You want a thread to wait until one or more events have occurred
- Initially created with a count of the number of events that must occur before the latch is release
- Each time an event happens, the count is decremented
- A form of `Abstract-QueuedSynchronizer`

HOW IT IS USED?

1. Instantiate the `CountDownLatch` with the same value for the counter as a number of threads we want to work across.
2. Call `await()` on the `CountDownLatch` instance to hold
3. Call `countdown()` on the `CountDownLatch` instance to `countdown()`
4. At zero, all `Thread` that are waiting will now continue.

DEMO: COUNTDOWN LATCH

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.evolutionnext.concurrency.countdownlatch` package and open all three files in the package

CYCLIC BARRIER

WHAT IS IT?

- Allows a fixed number of parties to rendezvous repeatedly at a barrier point
- Useful in parallel iterative algorithms that break down a problem into a fixed number of independent subproblems
- Threads call `await` when they reach the barrier point
- `await` blocks until all the threads have reached the barrier point
- If all threads meet at the barrier point, the barrier has been successfully passed
  - All Threads are released
  - The Barrier is reset

WHAT HAPPENS IF THERE ARE PROBLEMS?

- If a call to `await` times out or a thread blocked in await is interrupted, then the barrier is considered broken and all outstanding calls to await terminate with `BrokenBarrierException`
- If the barrier is successfully passed, await returns a unique arrival `index` for each thread, which can be used to “elect” a leader that takes some special action in the next iteration.

ALTERNATE CALLS TO `CyclicBarrier`

- `CyclicBarrier` also lets you pass a barrier "action" to the constructor
- Action is a `Runnable` that is executed as a subthread before the blocked threads are released.

DEMO: CYCLIC BARRIERS

Go to the java-sessions-threading-synchronizers-deep-dive project. In the */src/main/java* directory, navigate to the `com.evolutionnext.concurrency.cyclicbarrier` package and open the two classes.

PHASER

WHAT IS IT?

- A reusable synchronization barrier, that moves in phases
- Similar in functionality to `CyclicBarrier` and `CountDownLatch` but supporting more flexible usage.
- Tasks may be registered at any time with `register` and `bulkRegister`
- Arrivals are when a thread has reached the barrier and waits until others have reached that same barrier

REGISTRATION

- Adds a new unarrived party to this phaser.
- If an ongoing invocation of `onAdvance` is in progress, this method may await its completion before returning
- Returns the arrival phase number to which this registration applied.
- If the return value is negative, then this phaser has terminated, in which case registration has no effect.

ARRIVAL

- Thread signals it has arrived with `arriveAndAwaitAdvance()` and block until others in the party have reached the barrier
- When the number of arrived parties is equal to the number of registered parties, the execution of the program will continue
- Phase number will increase, and we can obtain that with `getPhase()`

DEREGISTERING

- When done, we can call `arriveAndDeregister()` signaling that we are no longer a member of the party

DEMO: PHASERS

In the java-sessions-threading-synchronizers-deep-dive project, go to the /src/main/java directory and navigate to the com.evolutionnext.concurrency.phaser package and open the two classes.

CONCURRENT COLLECTIONS

`BlockingQueue`

- `Queue` that additionally supports operations that wait for the queue to become non-empty when retrieving an element
- Wait for space to become available in the queue when storing an element
- Subtypes include:
  - `ArrayBlockingQueue`
  - `DelayQueue`
  - `LinkedBlockingDeque`
  - `LinkedBlockingQueue`
  - `LinkedTransferQueue`
  - `PriorityBlockingQueue`
  - `SynchronousQueue`

WHAT IF THE OPERATION CANNOT BE SATISFIED?

	Throws Exception	Special Value (`null` or `false`)	Blocks	Times out
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(e, time, unit)`
Examine	`element()`	`peek()`	*n/a*	*n/a*

`ConcurrentMap`

- A `Map` providing thread safety and atomicity guarantees, since regular `Map` are not Thread-safe
- Actions in a thread prior to placing an object into a `ConcurrentMap` as a key or value happen-before actions subsequent to the access or removal of that object from the `ConcurrentMap` in another thread.
- Is an extension of `Map`
- Subtypes include:
  - `ConcurrentHashMap`
  - `ConcurrentSkipListMap`

`ConcurrentNavigableMap`

- An interface that has navigable methods for the `Map`
- Backed by `SortedMap` where keys are sorted
- Includes
  - `subMap` - Returns a portion of the `Map` given by keys
  - `headMap` - Returns a view of the portion of this map whose keys are strictly less than a key
  - `tailMap` - Returns a view of the portion of this map whose keys are strictly greater than a key
  - `descendingMap` - Returns a reverse order view of the mappings contained in this map.
  - `descendingKeySet()` - Returns a reverse order NavigableSet view of the keys contained in this map.
- Subtypes include: `ConcurrentSkipListMap`

STRUCTURED CONCURRENCY

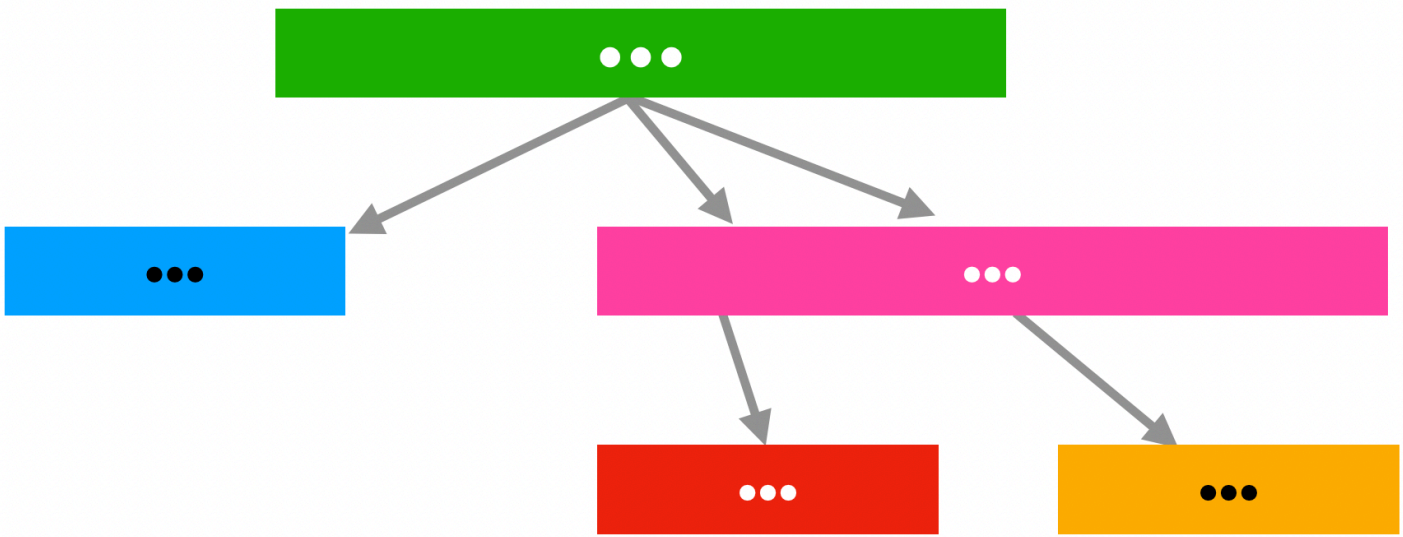
STRUCTURED CONCURRENCY

- **"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?

```
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

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

        scope.join()            // Join both subtasks
             .throwIfFailed();  // ... and propagate errors

        // 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
- Two subclasses of `StructuredTaskScope`:
  - `ShutdownOnFailure` - (invoke all)
  - `ShutdownOnSuccess` - (invoke any)

As soon as one subtask succeeds this scope automatically shuts down, cancelling unfinished subtasks.

DEMO: STRUCTURED CONCURRENCY

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.xyzcorp.concurrency.structuredconcurrency` package and open all the classes.

SCOPED VALUES

SCOPED VALUES

- Enable a method to share immutable data both with its callees within a thread, and with 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

```
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.

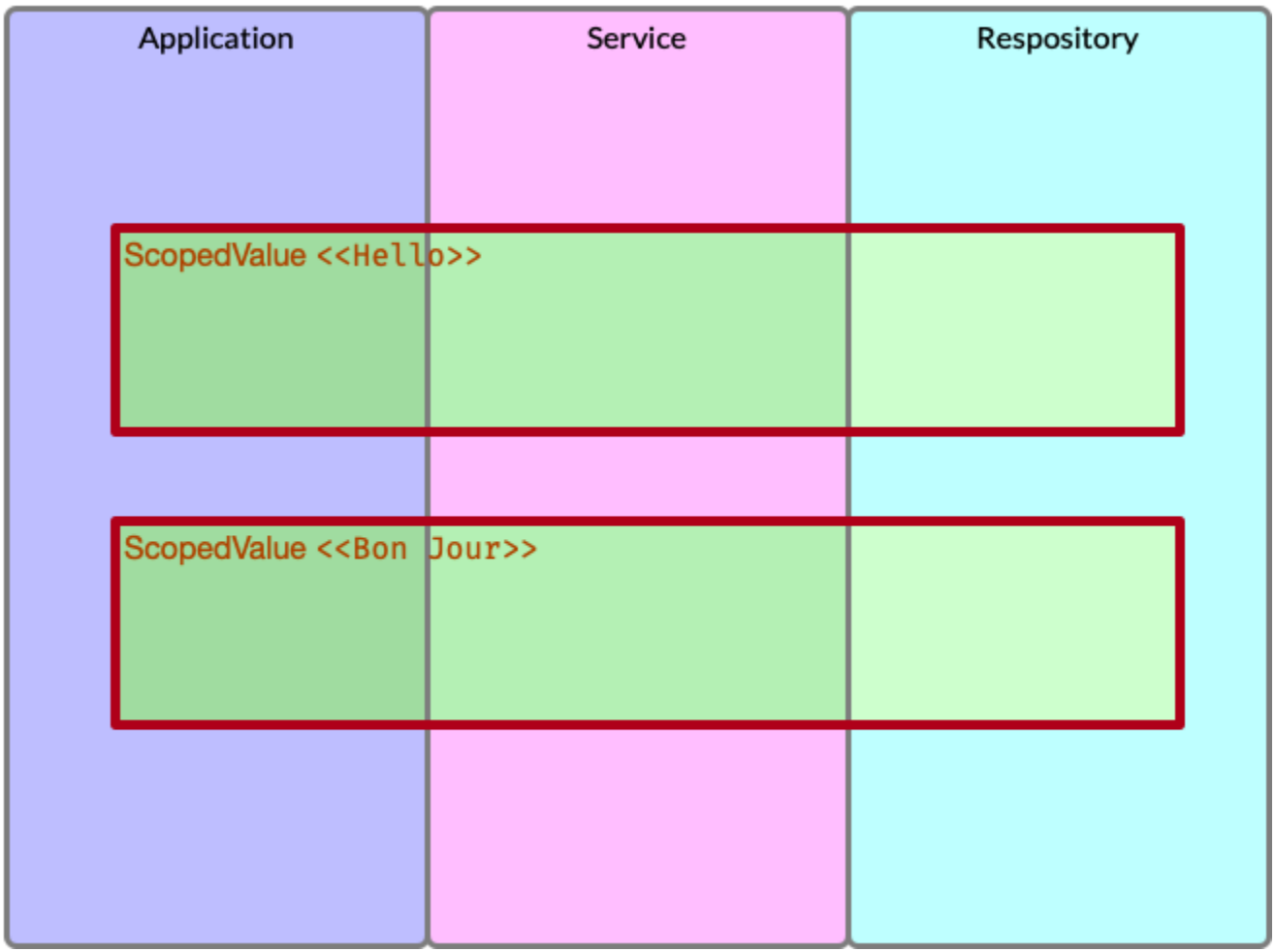
Source: JEP 481

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.

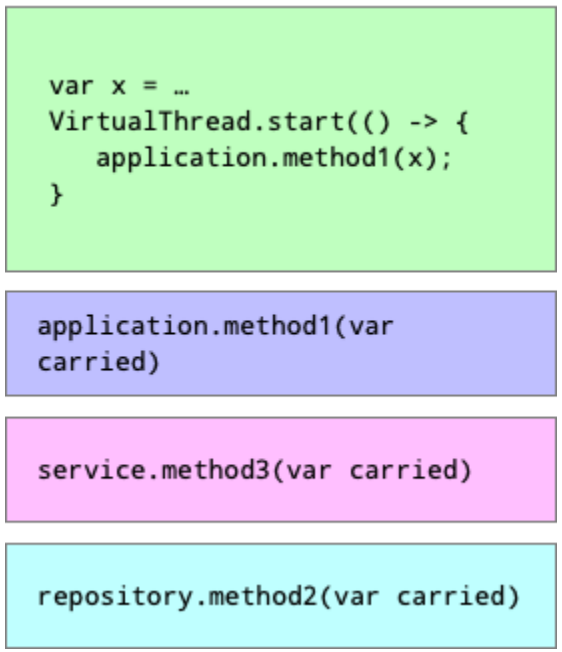
Source: JEP 481

SCOPED VALUES



WITHOUT SCOPED VALUES

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



WITH SCOPED VALUES

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

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

VirtualThread.start(() -> {
  application.method1(scoped);
}
```

```
application.method1()
```

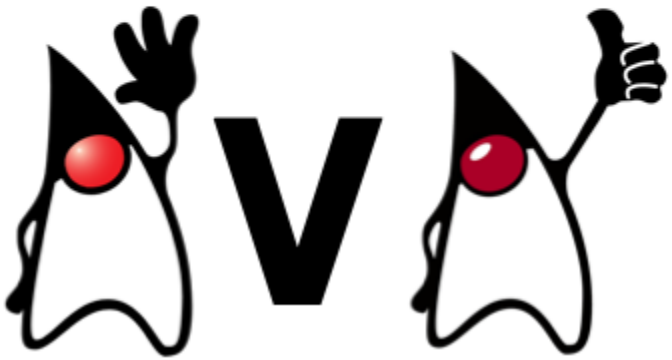
```
service.method3()
```

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

DEMO: SCOPED VALUES

In the *java-sessions-threading-synchronizers-deep-dive/src/main/java* directory, navigate to the `com.xyzcorp.scopedvalues` package and open all the classes.

WIN WITH JAVA!



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