Overview of Java Synchronizers

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems

Vanderbilt University

Nashville, Tennessee, USA

Learning Objectives in this Lesson

- Understand the categories of capabilities provided by Java synchronizers

Category	Definition
Atomic operations	An action that effectively happens all at once or not at all
Mutual exclusion	Allows concurrent access & updates to shared data without race conditions
Coordination	Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.
Barrier synchronization	Ensures that any thread(s) must stop at a certain point & cannot proceed until all other thread(s) reach this barrier

Learning Objectives in this Lesson

- Understand the categories of capabilities provided by Java synchronizers

Category	Definition
Atomic operations	An action that effectively happens all at once or not at all
Mutual exclusion	Allows concurrent access & updates to shared mutable data without race conditions
Coordination	Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.
Barrier synchronization	Ensures that any thread(s) must stop at a certain point & cannot proceed until all other thread(s) reach this barrier

Learning Objectives in this Lesson

- Understand the categories of capabilities provided by Java synchronizers

Category	Definition
Atomic operations	An action that effectively happens all at once or not at all
Mutual exclusion	Allows concurrent access & updates to shared mutable data without race conditions
Coordination	Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.
Barrier synchronization	Ensures that any thread(s) must stop at a certain point & cannot proceed until all other thread(s) reach this barrier

Learning Objectives in this Lesson

- Understand the categories of capabilities provided by Java synchronizers

Category	Definition
Atomic operations	An action that effectively happens all at once or not at all
Mutual exclusion	Allows concurrent access & updates to shared mutable data without race conditions
Coordination	Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.
Barrier synchronization	Ensures that any thread(s) must stop at a certain point & cannot proceed until all other thread(s) reach this barrier

Learning Objectives in this Lesson

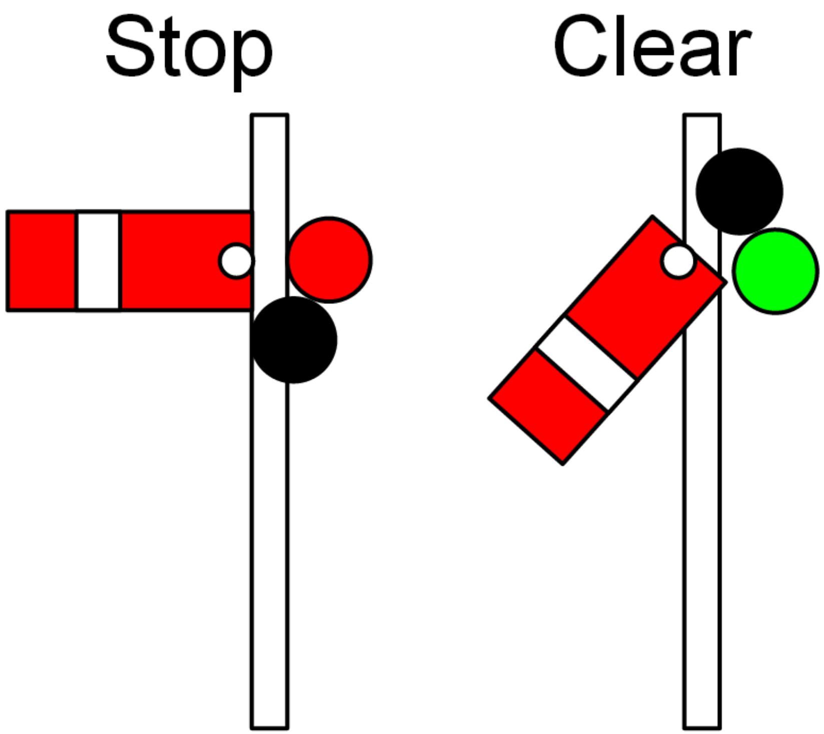
- Understand the categories of capabilities provided by Java synchronizers

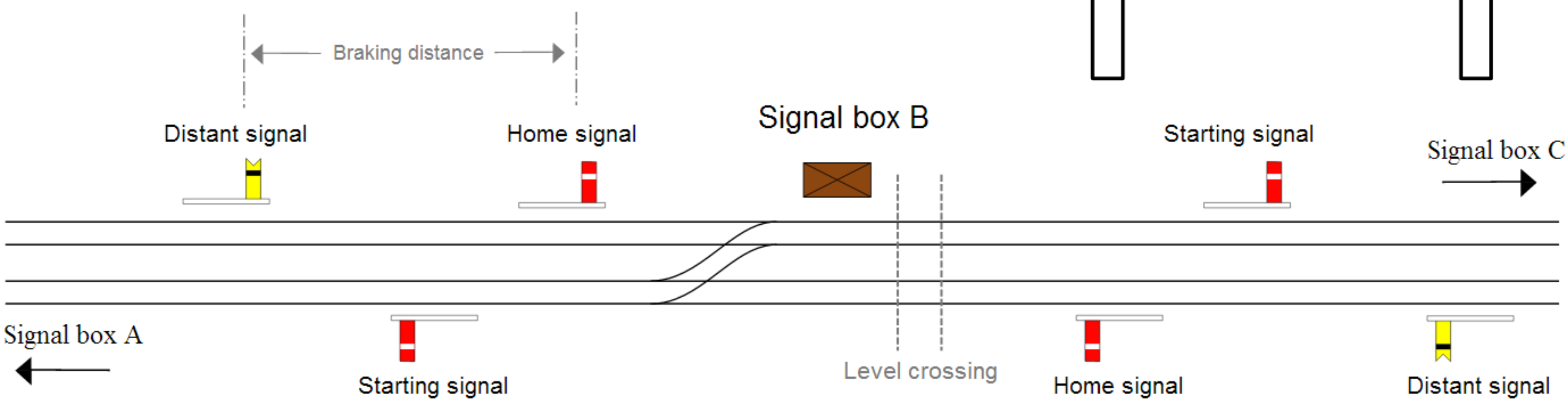
Category	Definition
Atomic operations	An action that effectively happens all at once or not at all
Mutual exclusion	Allows concurrent access & updates to shared mutable data without race conditions
Coordination	Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.
Barrier synchronization	Ensures that any thread(s) must stop at a certain point & cannot proceed until all other thread(s) reach this barrier

Overview of Java Synchronizers

Overview of Java Synchronizers

- A Java synchronizer is an object use to control the flow of cooperating threads based on its state





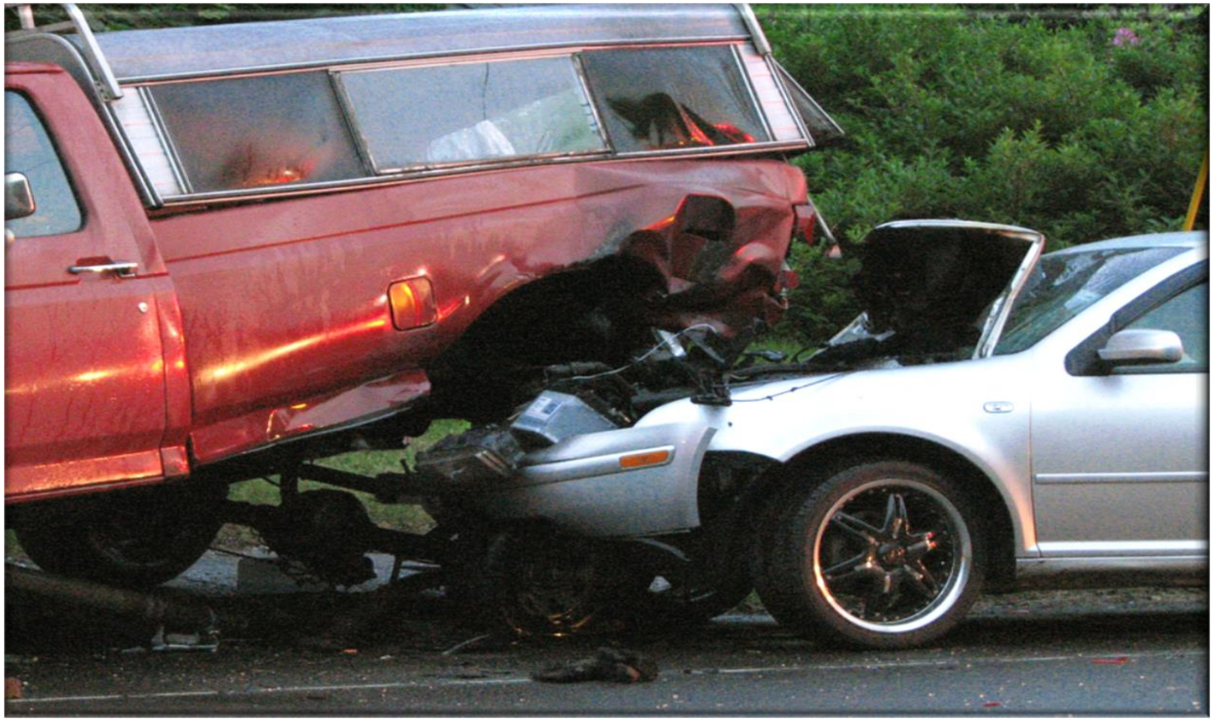
See en.wikipedia.org/wiki/Synchronization_(computer_science)

Overview of Java Synchronizers

- Java synchronizers ensure interactions between threads obey certain properties

Overview of Java Synchronizers

- Java synchronizers ensure interactions between threads obey certain properties, e.g.
  - Don’t corrupt shared mutable data



Overview of Java Synchronizers

- Java synchronizers ensure interactions between threads obey certain properties, e.g.
  - Don't corrupt shared mutable data

```
class AtomicOps {
  long mCounter = 0;

  void increment() {
    // Thread T1
    for (;;) mCounter++;
  }

  void decrement() {
    // Thread T2
    for (;;) mCounter--;
  }
  ...
}
```

Running increment() & decrement() concurrently yields undefined behavior since mCounter is shared mutable data

Overview of Java Synchronizers

- Java synchronizers ensure interactions between threads obey certain properties, e.g.
  - Don't corrupt shared mutable data

```
class AtomicOps {
  long mCounter = 0;

  synchronized void increment() {
    // Thread T1
    for (;;) mCounter++;
  }

  synchronized void decrement() {
    // Thread T2
    for (;;) mCounter--;
  }
  ...
}
```

Running increment() & decrement() concurrently yields correct behavior since mCounter is synchronized shared mutable data

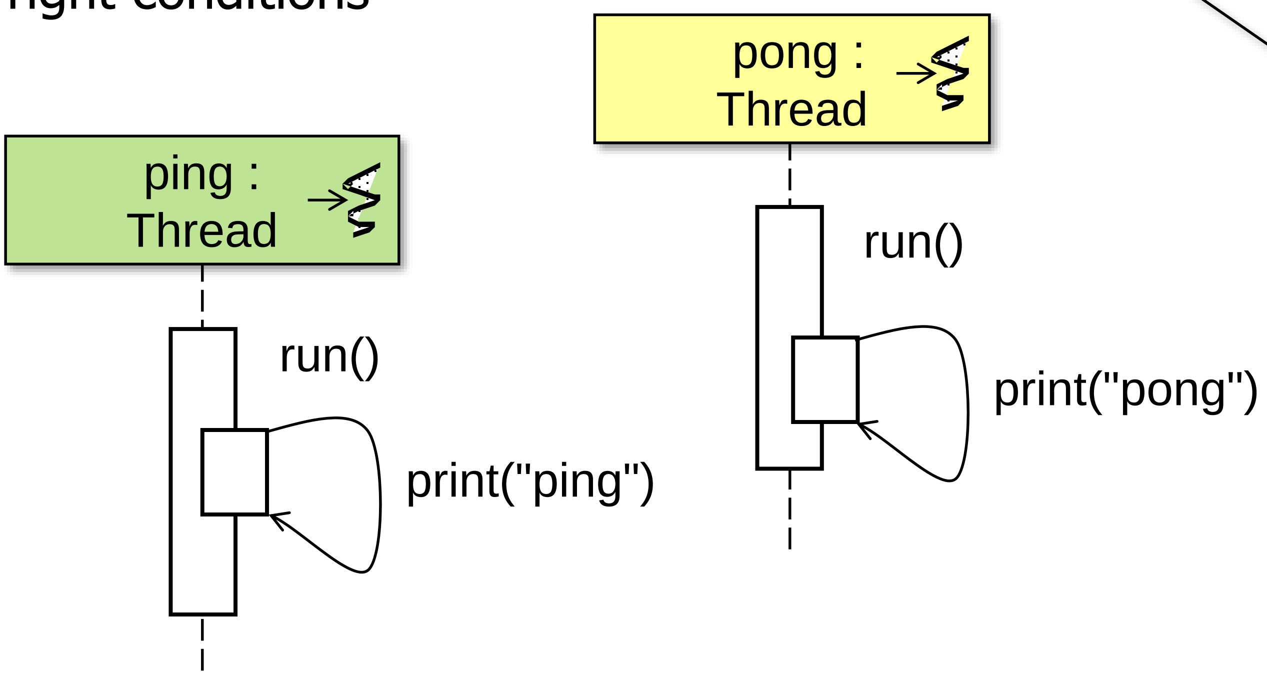
Overview of Java Synchronizers

- Java synchronizers ensure interactions between threads obey certain properties, e.g.
  - Don't corrupt shared mutable data
  - Occur in the right order, at the right time, & under the right conditions

Overview of Java Synchronizers

- Java synchronizers ensure interactions between threads obey certain properties, e.g.
  - Don't corrupt shared mutable data
  - Occur in the right order, at the right time, & under the right conditions

The unsynchronized version is buggy

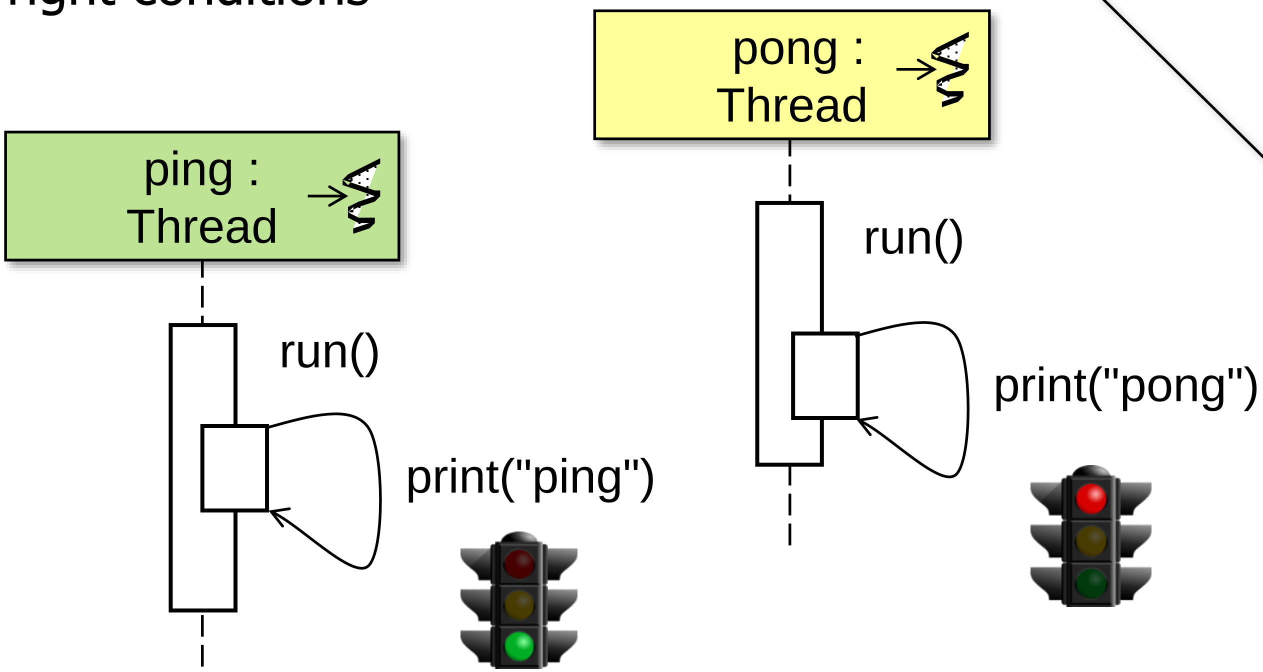



```
% java PingPongWrong
Ready...Set...Go!
Ping!(1)
Ping!(2)
Ping!(3)
Ping!(4)
Ping!(5)
Ping!(6)
Ping!(7)
Ping!(8)
Ping!(9)
Ping!(10)
Pong!(1)
Pong!(2)
Pong!(3)
Pong!(4)
Pong!(5)
Pong!(6)
Pong!(7)
Pong!(8)
Pong!(9)
Pong!(10)
Done!
```

Overview of Java Synchronizers

- Java synchronizers ensure interactions between threads obey certain properties, e.g.
  - Don't corrupt shared mutable data
  - Occur in the right order, at the right time, & under the right conditions

The synchronized version coordinates the threads properly




```
% java PlayPingPong
Ready...Set...Go!
Ping!(1)
Pong!(1)
Ping!(2)
Pong!(2)
Ping!(3)
Pong!(3)
Ping!(4)
Pong!(4)
Ping!(5)
Pong!(5)
Ping!(6)
Pong!(6)
Ping!(7)
Pong!(7)
Ping!(8)
Pong!(8)
Ping!(9)
Pong!(9)
Ping!(10)
Pong!(10)
Done!
```

Java Synchronizers Address Inherent Complexities

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency

Inherent complexities are the "rocket science" of software development

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - **Atomic ordering**
    - Ensures an action happens all at once or not at all

See en.wikipedia.org/wiki/Linearizability

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - **Atomic ordering**
    - Ensures an action happens all at once or not at all
    - Operations on a field in thread_1 occur all at once wrt operations on the field in $\text{thread}_{2..n}$

time

Thread_1	Thread_2		Long field
initialized			0
read field		←	0
increase field by 1			0
write back		→	1
	read field	←	1
	increase field by 1		1
	write back	→	2

Atomicity does not occur on primitive Java data types without using synchronizers

See docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - **Atomic ordering**
    - Ensures an action happens all at once or not at all
    - Operations on a field in thread_1 occur all at once wrt operations on the field in $\text{thread}_{2..n}$
    - Atomic ordering is supported by the Java atomic package

Package java.util.concurrent.atomic

A small toolkit of classes that support lock-free thread-safe programming on single variables.

See: Description

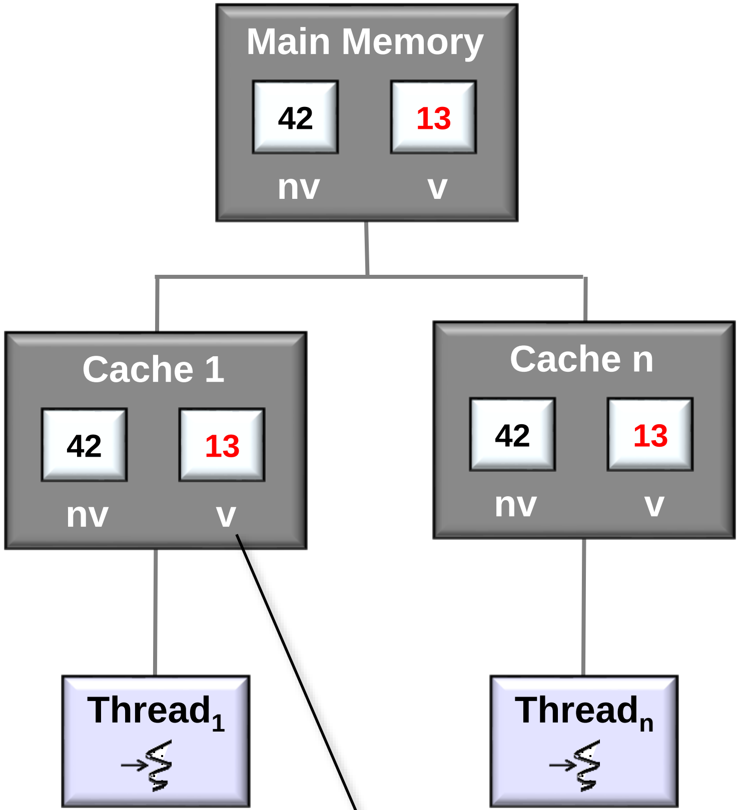
Class Summary

Class	Description
AtomicBoolean	A `boolean` value that may be updated atomically.
AtomicInteger	An `int` value that may be updated atomically.
AtomicIntegerArray	An `int` array in which elements may be updated atomically.
AtomicIntegerFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated `volatile int` fields of designated classes.
AtomicLong	A `long` value that may be updated atomically.
AtomicLongArray	A `long` array in which elements may be updated atomically.
AtomicLongFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated `volatile long` fields of designated classes.
AtomicMarkableReference<V>	An `AtomicMarkableReference` maintains an object reference along with a mark bit, that can be updated atomically.
AtomicReference<V>	An object reference that may be updated atomically.
AtomicReferenceArray<E>	An array of object references in which elements may be updated atomically.
AtomicReferenceFieldUpdater<T,V>	A reflection-based utility that enables atomic updates to designated `volatile` reference fields of designated classes.
AtomicStampedReference<V>	An `AtomicStampedReference` maintains an object reference along with an integer "stamp", that can be updated atomically.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - **Atomic ordering**
    - Ensures an action happens all at once or not at all
    - Operations on a field in thread_1 occur all at once wrt operations on the field in $\text{thread}_{2..n}$
    - Atomic ordering is supported by the Java atomic package
    - Atomic ordering is also supported by the Java volatile type qualifier



The volatile type qualifier ensures a variable is read from & written to main memory & not cached

See en.wikipedia.org/wiki/Volatile_variable#In_Java

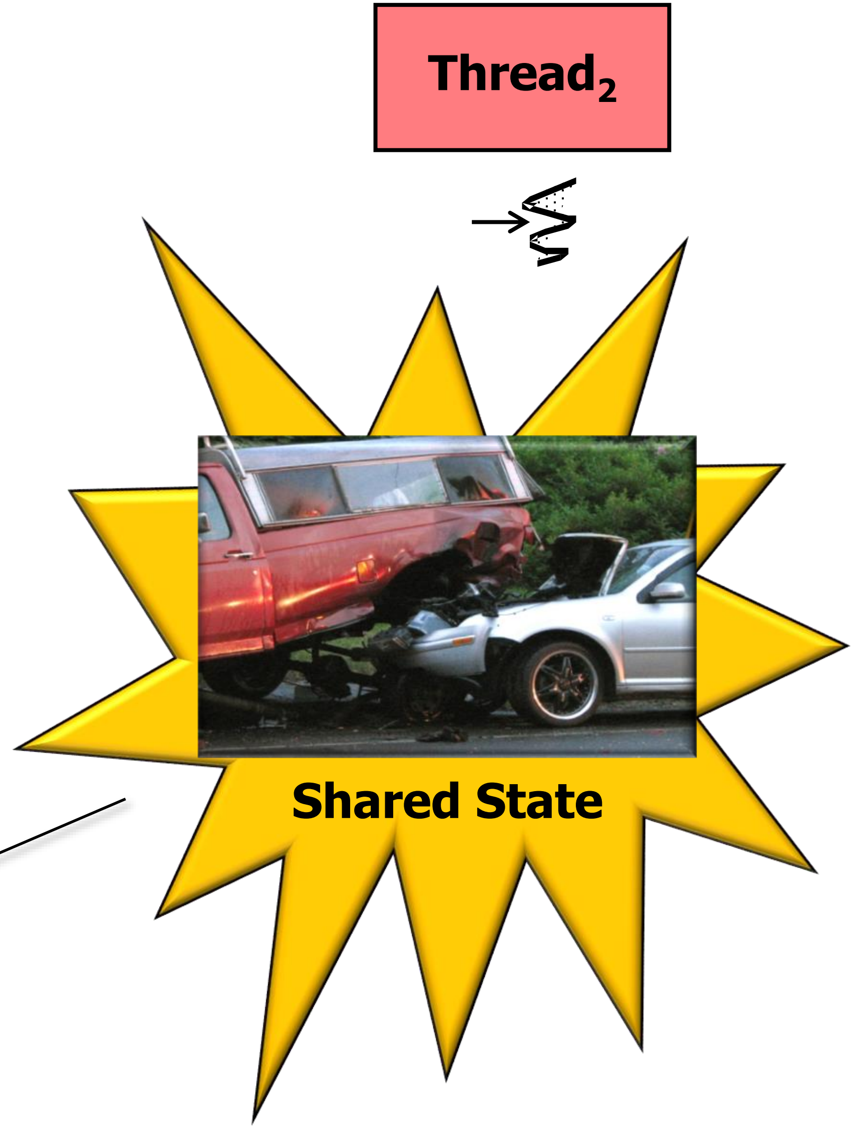
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - **Mutual exclusion**
    - Prevents simultaneous access to a shared resource in a critical section

See en.wikipedia.org/wiki/Mutual_exclusion

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - **Atomic ordering**
  - **Mutual exclusion**
    - Prevents simultaneous access to a shared resource in a critical section



Race conditions occur when a program depends on the sequence or timing of threads for it to operate properly

See en.wikipedia.org/wiki/Race_condition#Software

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - **Atomic ordering**
  - **Mutual exclusion**
    - Prevents simultaneous access to a shared resource in a critical section
    - Read/write conflicts
      - If one thread reads while another thread writes concurrently, the field that's read may be inconsistent

time

$Thread_1$	$Thread_2$		Long field
initialized			0
read field		←	0
increase field by 1			0
write back	read field	← →	0 or 1?

Two operations *conflict* if at least one is a write

Java Synchronizers Address Inherent Complexities

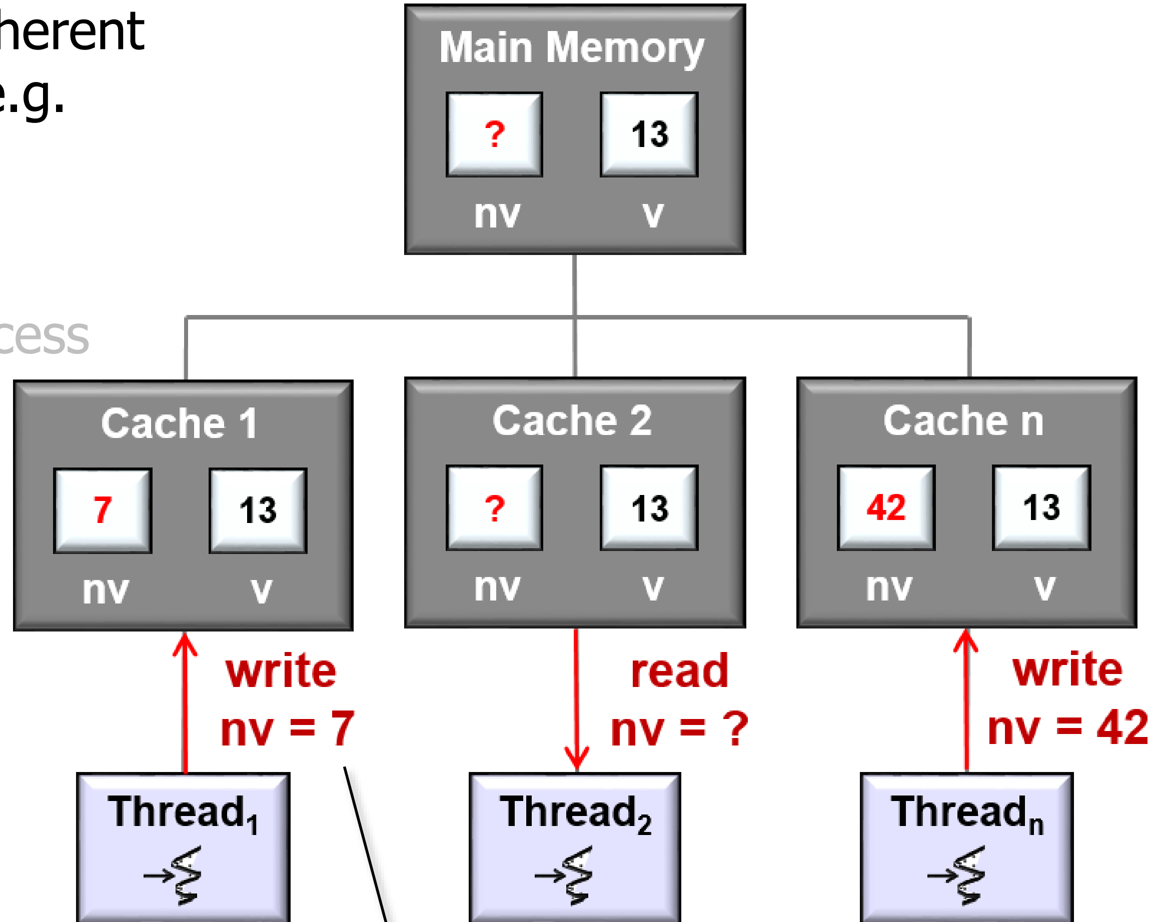
- Java synchronizers address inherent complexities of concurrency, e.g.
  - **Atomic ordering**
  - **Mutual exclusion**
    - Prevents simultaneous access to a shared resource in a critical section
    - Read/write conflicts
    - Write/write conflicts
      - If two threads try to write to same field concurrently, the result may be inconsistent

time

Thread$_1$	Thread$_2$		Long field
initialized			0
read field		←	0
	read field	←	0
increase field by 2			0
	increase field by 1		0
write back	write back	→	1 or 2?

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - **Mutual exclusion**
    - Prevents simultaneous access to a shared resource in a critical section
    - Read/write conflicts
    - Write/write conflicts



These problems often occur in multi-core processors with "weak" memory ordering due to core caches that allow "out-of-order" load & store operations

See en.wikipedia.org/wiki/Memory_ordering

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - **Mutual exclusion**
    - Prevents simultaneous access to a shared resource in a critical section
    - Read/write conflicts
    - Write/write conflicts
    - Mutual exclusion is supported by the Java locks package
      - e.g., ReentrantLock, Reentrant ReadWriteLock, StampedLock, etc.

Package java.util.concurrent.locks

Interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors.

See: Description

Interface Summary

Interface	Description
Condition	Condition factors out the Object monitor methods (**wait**, **notify** and **notifyAll**) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary **Lock** implementations.
Lock	Lock implementations provide more extensive locking operations than can be obtained using synchronized methods and statements.
ReadWriteLock	A ReadWriteLock maintains a pair of associated **locks**, one for read-only operations and one for writing.

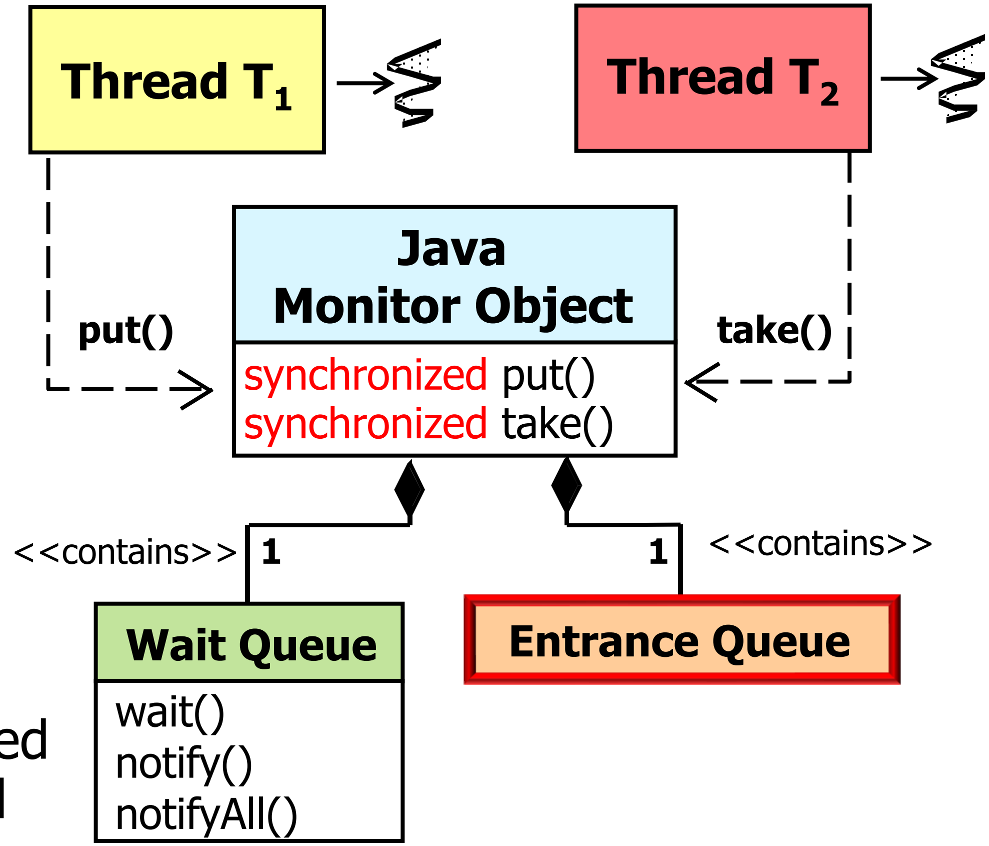
Class Summary

Class	Description
AbstractOwnableSynchronizer	A synchronizer that may be exclusively owned by a thread.
AbstractQueuedLongSynchronizer	A version of **AbstractQueuedSynchronizer** in which synchronization state is maintained as a long.
AbstractQueuedSynchronizer	Provides a framework for implementing blocking locks and related synchronizers (semaphores, events, etc) that rely on first-in-first-out (FIFO) wait queues.
LockSupport	Basic thread blocking primitives for creating locks and other synchronization classes.
ReentrantLock	A reentrant mutual exclusion **Lock** with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.
ReentrantReadWriteLock	An implementation of **ReadWriteLock** supporting similar semantics to **ReentrantLock**.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/package-summary.html

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - **Mutual exclusion**
    - Prevents simultaneous access to a shared resource in a critical section
    - Read/write conflicts
    - Write/write conflicts
    - Mutual exclusion is supported by the Java locks package
    - Mutual exclusion is also supported by the **`synchronized`** keyword in Java built-in monitor objects



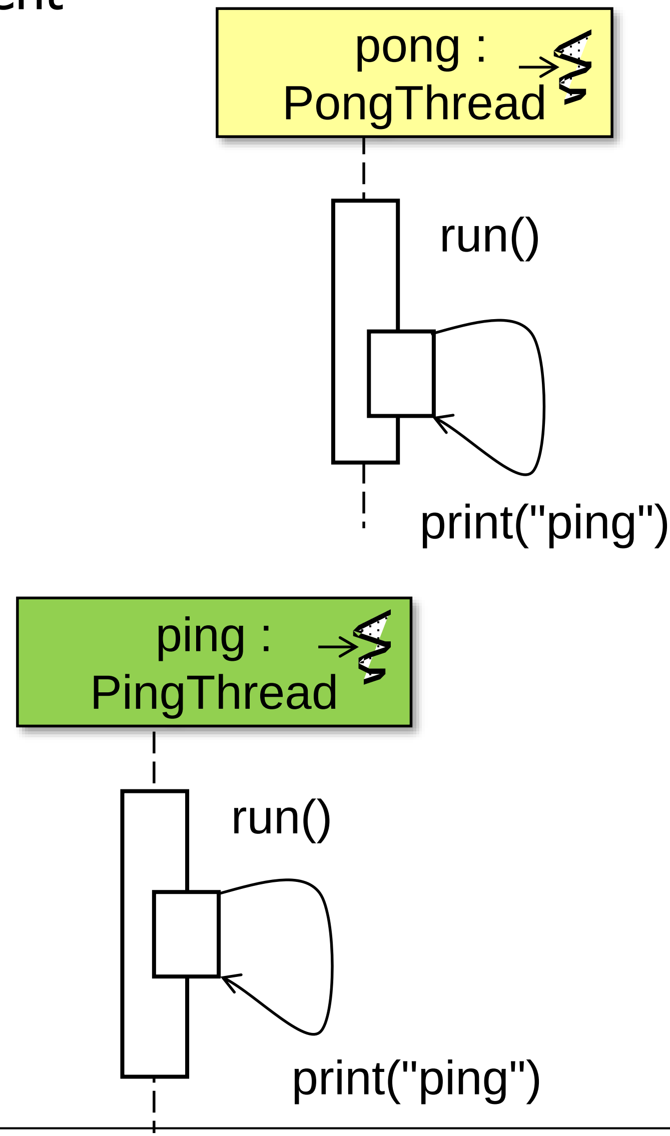
See www.artima.com/insidejvm/ed2/threadsynch.html

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - Mutual exclusion
  - **Coordination**
    - Ensures computations run properly

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - Mutual exclusion
  - **Coordination**
    - Ensures computations run properly, e.g.
      - In the right order

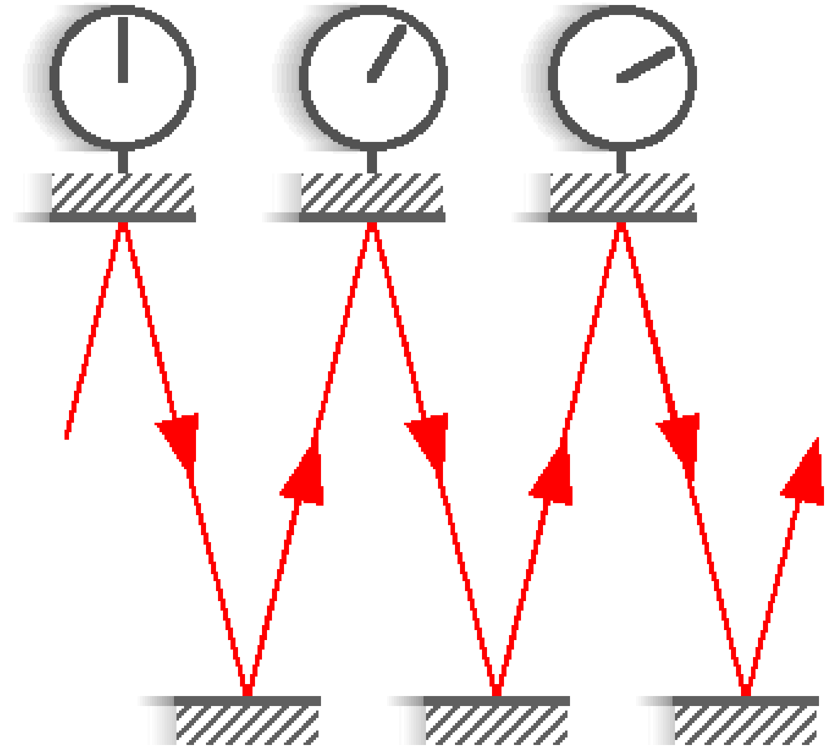



```
% java PingPong
Ready...Set...Go!
Ping!(1)
Pong!(1)
Ping!(2)
Pong!(2)
Ping!(3)
Pong!(3)
Ping!(4)
Pong!(4)
Ping!(5)
Pong!(5)
Ping!(6)
Pong!(6)
Ping!(7)
Pong!(7)
Ping!(8)
Pong!(8)
Ping!(9)
Pong!(9)
Ping!(10)
Pong!(10)
Done!
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/PingPongApplication

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - **Atomic ordering**
  - **Mutual exclusion**
  - **Coordination**
    - Ensures computations run properly, e.g.
      - In the right order
      - At the right time



See en.wikipedia.org/wiki/Real-time_computing

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - **Atomic ordering**
  - **Mutual exclusion**
  - **Coordination**
    - Ensures computations run properly, e.g.
      - In the right order
      - At the right time
      - Under the right conditions

See github.com/douglascraigschmidt/LiveLessons/tree/master/PalantiriManagerApplication

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - **Atomic ordering**
  - **Mutual exclusion**
  - **Coordination**
    - Ensures computations run properly
    - Coordination is supported by the Java concurrent & locks packages
      - e.g., ConditionObject, Semaphore, etc.

Package java.util.concurrent

Utility classes commonly useful in concurrent programming.

See: Description

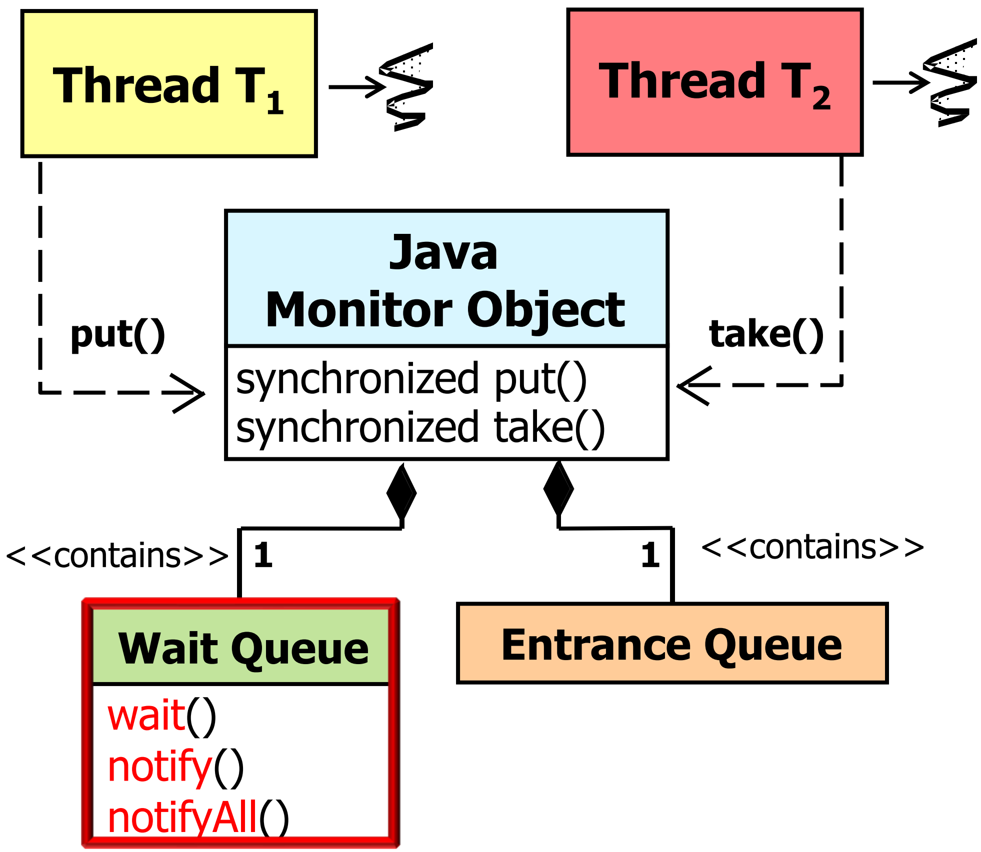
Interface Summary

Interface	Description
BlockingDeque<E>	A **Deque** that additionally supports blocking operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element.
BlockingQueue<E>	A **Queue** that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
Callable<V>	A task that returns a result and may throw an exception.
CompletableFuture.AsynchronousCompletionTask	A marker interface identifying asynchronous tasks produced by `async` methods.
CompletionService<V>	A service that decouples the production of new asynchronous tasks from the consumption of the results of completed tasks.
CompletionStage<T>	A stage of a possibly asynchronous computation, that performs an action or computes a value when another CompletionStage completes.
ConcurrentMap<K,V>	A **Map** providing thread safety and atomicity guarantees.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - Mutual exclusion
  - **Coordination**
    - Ensures computations run properly
    - Coordination is supported by the Java concurrent & locks packages
    - Coordination is also supported by Java built-in monitor objects



See www.artima.com/insidejvm/ed2/threadsynch.html

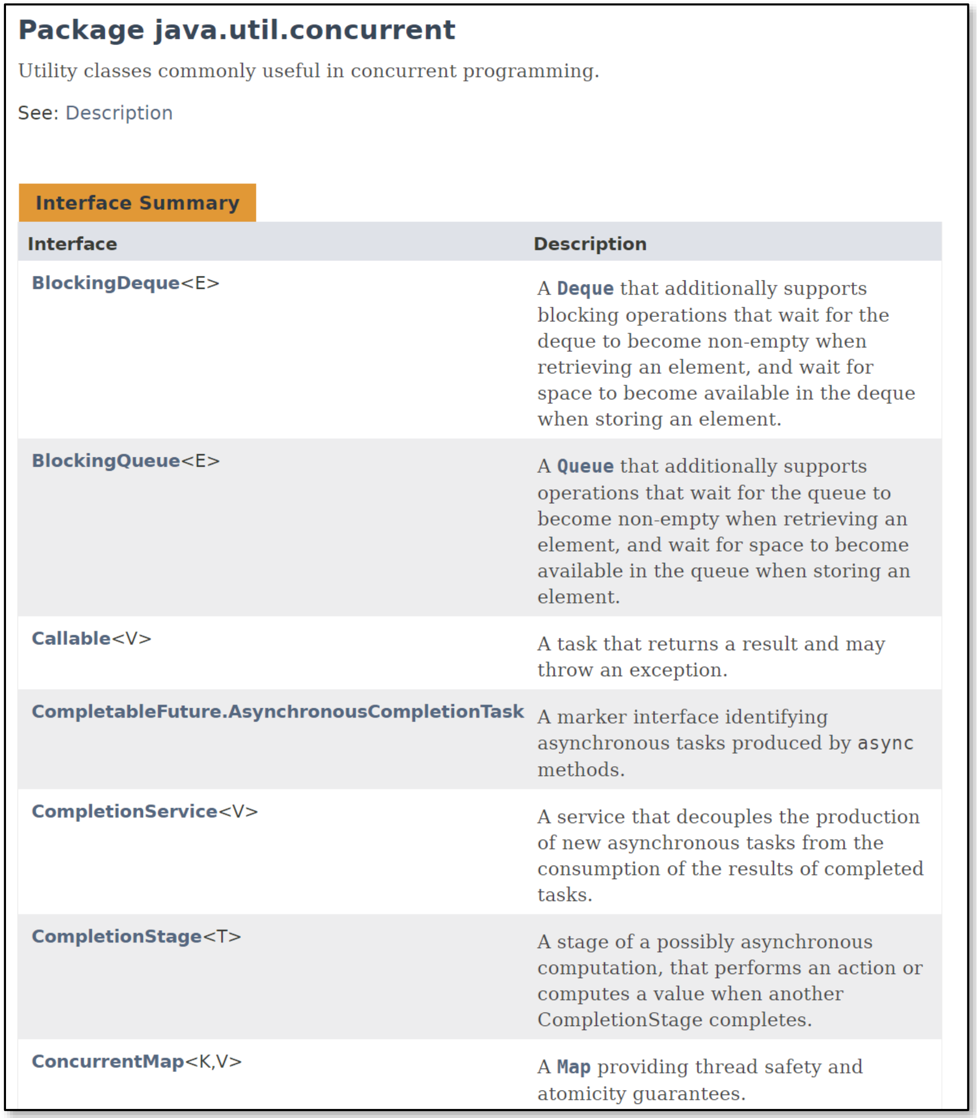
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - Mutual exclusion
  - Coordination
  - **Barrier synchronization**
    - Ensures that any thread(s) must stop at a certain point & cannot proceed until all thread(s) reach the barrier

Barrier synchronization is a variant of coordination

Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - Mutual exclusion
  - Coordination
  - **Barrier synchronization**
    - Ensures that any thread(s) must stop at a certain point & cannot proceed until all thread(s) reach the barrier
    - Barrier synchronization is supported by the Java concurrent package
      - e.g., CountDownLatch, CyclicBarrier, Phaser, etc.



Package java.util.concurrent

Utility classes commonly useful in concurrent programming.

See: Description

Interface Summary

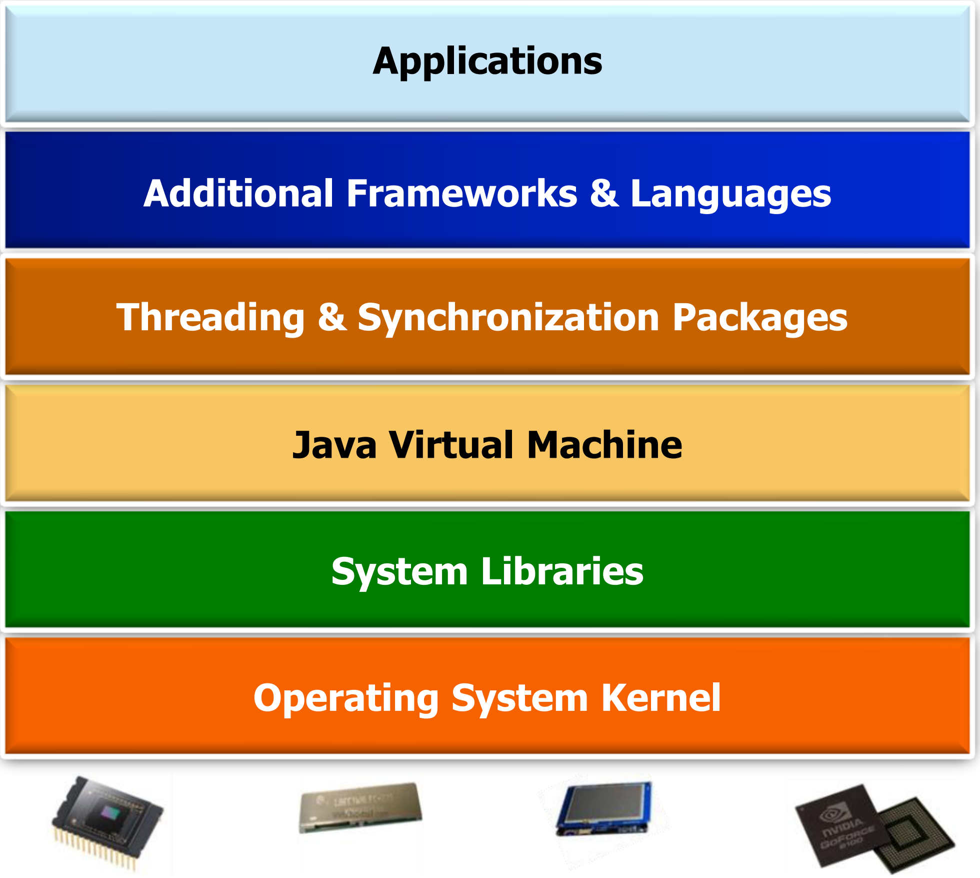
Interface	Description
BlockingDeque<E>	A Deque that additionally supports blocking operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element.
BlockingQueue<E>	A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
Callable<V>	A task that returns a result and may throw an exception.
CompletableFuture.AsynchronousCompletionTask	A marker interface identifying asynchronous tasks produced by async methods.
CompletionService<V>	A service that decouples the production of new asynchronous tasks from the consumption of the results of completed tasks.
CompletionStage<T>	A stage of a possibly asynchronous computation, that performs an action or computes a value when another CompletionStage completes.
ConcurrentMap<K,V>	A Map providing thread safety and atomicity guarantees.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html

Pervasiveness of Synchronizers in Java

Pervasiveness of Java Synchronizer Classes

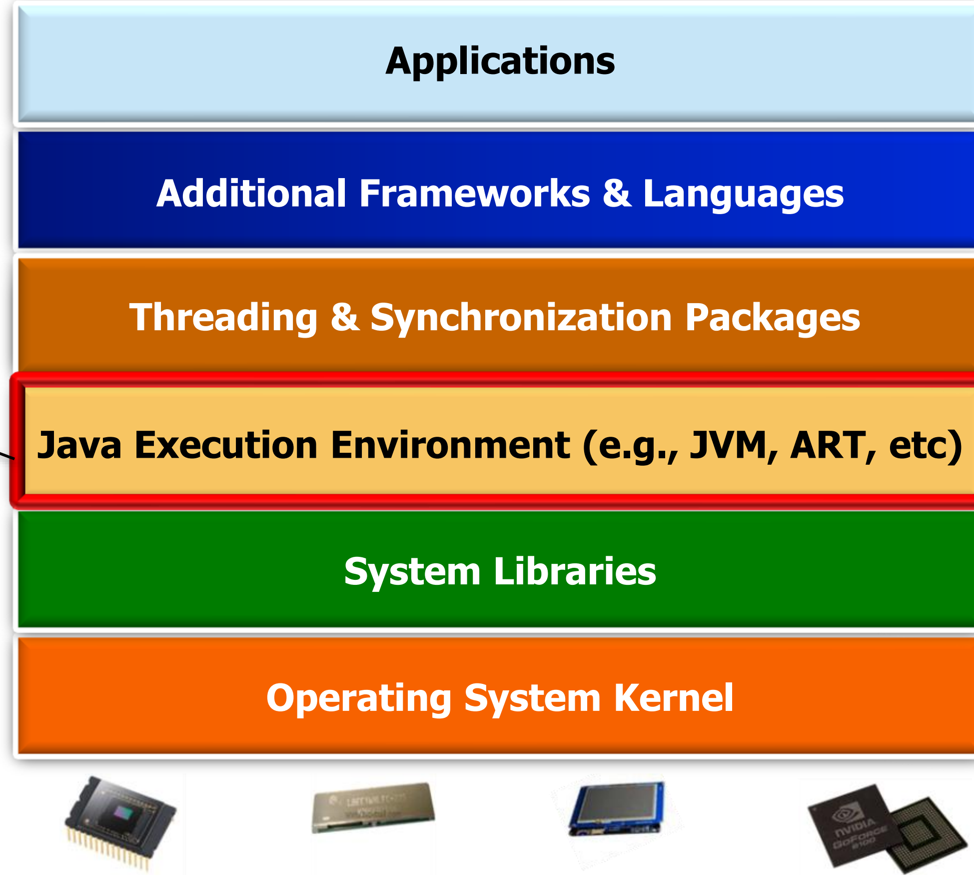
- Multiple layers of synchronizers are provided on the Java platform



Pervasiveness of Java Synchronizer Classes

- Multiple layers of synchronizers are provided on the Java platform, e.g.
  - The Java language contains some features that synchronize threads

e.g., volatile variables & built-in monitor objects

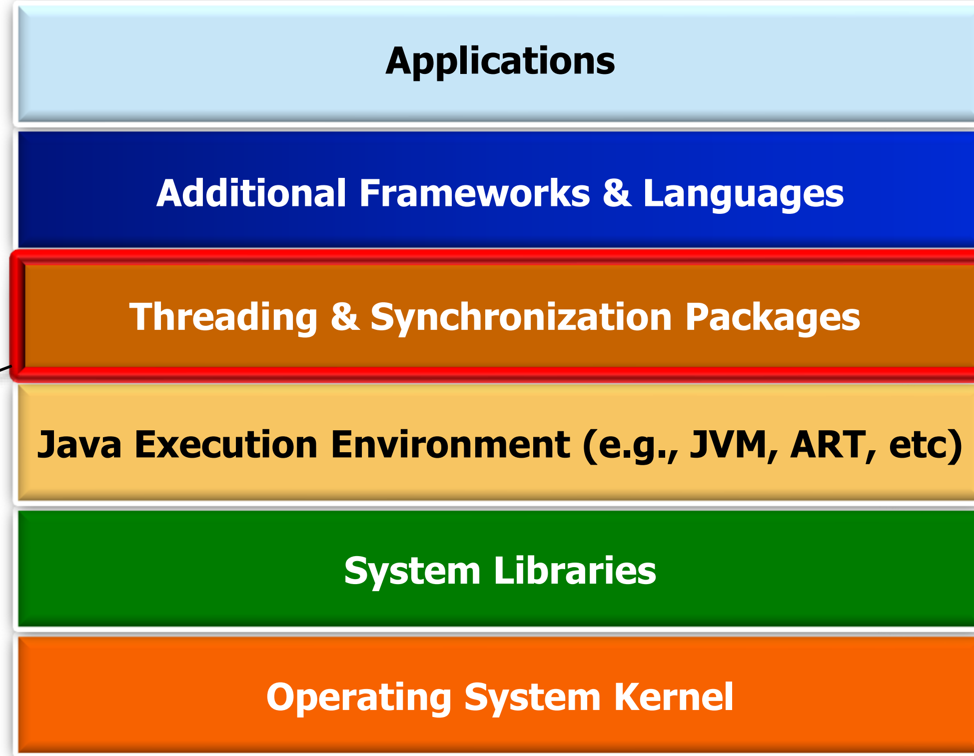


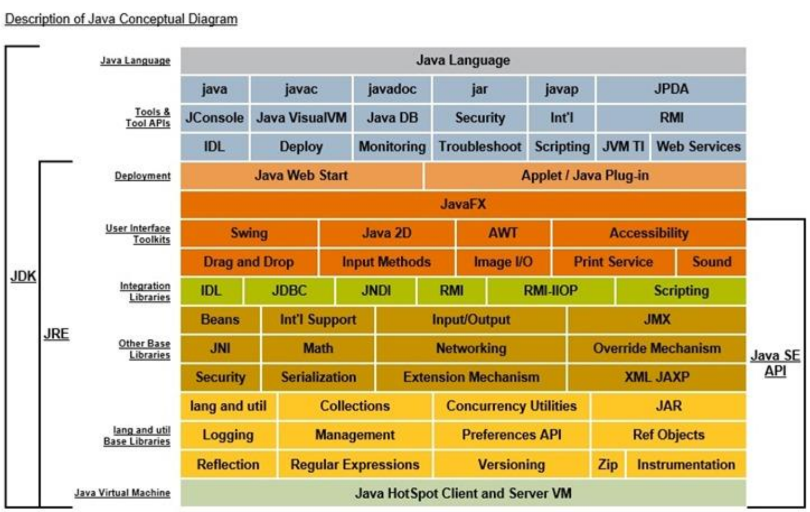
See en.wikipedia.org/wiki/Java_(programming_language)

Pervasiveness of Java Synchronizer Classes

- Multiple layers of synchronizers are provided on the Java platform, e.g.
  - The Java language contains some features that synchronize threads
  - Other synchronizers are provided by the Java Class Library

e.g., Java atomics, locks, & other synchronizers





See en.wikipedia.org/wiki/Java_Class_Library

Pervasiveness of Java Synchronizer Classes

- We focus more on Java synchronization mechanisms than on Java threading mechanisms

Threading coverage

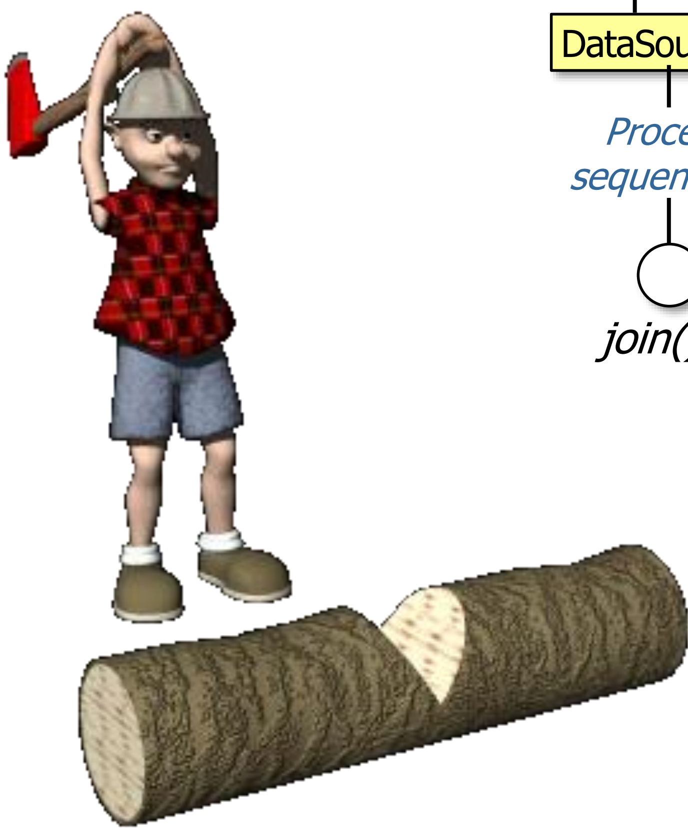
Synchronization coverage

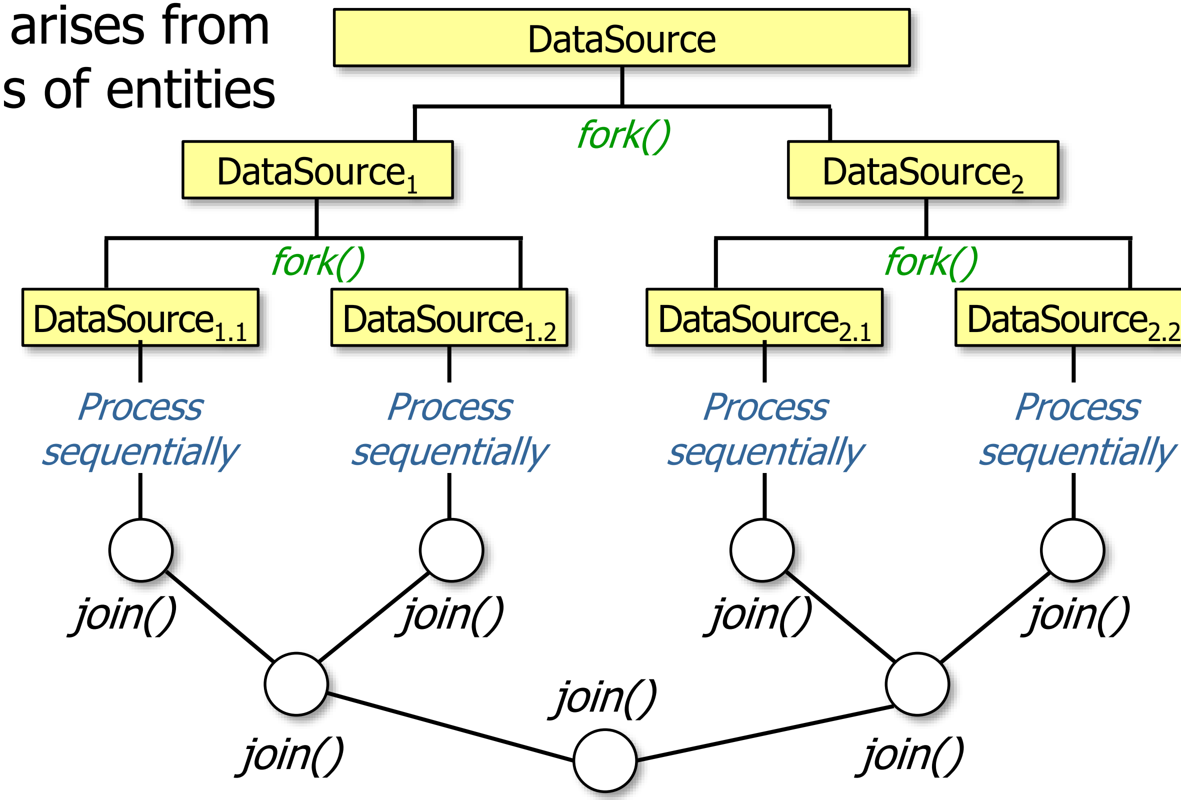
Pervasiveness of Java Synchronizer Classes

- Synchronization complexity arises from coordinating the interactions of entities that run concurrently

Pervasiveness of Java Synchronizer Classes

- Synchronization complexity arises from coordinating the interactions of entities that run concurrently





Java 8 parallelism frameworks may eliminate some of this complexity via "divide and conquer"

End of Overview of Java Synchronizers