# Learning Objectives in this Lesson

- Know the key synchronizers defined in the Java class library

<table>
<thead>
<tr>
<th>Java Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReentrantLock</td>
<td>A reentrant mutual exclusion lock that extends the built-in monitor lock capabilities</td>
</tr>
<tr>
<td>ReentrantReadWriteLock</td>
<td>Improves performance when resources are read much more often than written</td>
</tr>
<tr>
<td>StampedLock</td>
<td>A readers-writer lock that’s more efficient than ReentrantReadWriteLock</td>
</tr>
<tr>
<td>Semaphore</td>
<td>Maintains permits that controls thread access to limited # of shared resources</td>
</tr>
<tr>
<td>ConditionObject</td>
<td>Allows Thread to block until a condition becomes true</td>
</tr>
<tr>
<td>CountDownLatch</td>
<td>Allows one or more threads to wait until a set of operations being performed in other threads complete</td>
</tr>
<tr>
<td>CyclicBarrier</td>
<td>Allows a set of threads to all wait for each other to reach a common barrier point</td>
</tr>
<tr>
<td>Phaser</td>
<td>A more flexible reusable synchronization barrier</td>
</tr>
</tbody>
</table>
Learning Objectives in this Lesson

• Know the key synchronizers defined in the Java class library

• Recognize synchronizer usage considerations
Capabilities of Java Synchronizer Classes
### Capabilities of Java Synchronizer Classes

- Java synchronization classes can be split into several categories of capabilities

<table>
<thead>
<tr>
<th>Category</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic operations</td>
<td>An action that effectively happens all at once or not at all</td>
</tr>
<tr>
<td>Mutual exclusion</td>
<td>Allows concurrent access &amp; updates to shared data without race conditions</td>
</tr>
<tr>
<td>Coordination</td>
<td>Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.</td>
</tr>
<tr>
<td>Barrier synchronization</td>
<td>Ensures that any thread(s) must stop at a certain point &amp; cannot proceed until all other thread(s) reach this barrier</td>
</tr>
</tbody>
</table>
Java synchronization classes can be split into several categories of capabilities:

<table>
<thead>
<tr>
<th>Category</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic operations</td>
<td>An action that effectively happens all at once or not at all</td>
</tr>
<tr>
<td>Mutual exclusion</td>
<td>Allows concurrent access &amp; updates to shared data without race conditions</td>
</tr>
<tr>
<td>Coordination</td>
<td>Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.</td>
</tr>
<tr>
<td>Barrier synchronization</td>
<td>Ensures that any thread(s) must stop at a certain point &amp; cannot proceed until all other thread(s) reach this barrier</td>
</tr>
</tbody>
</table>
## Capabilities of Java Synchronizer Classes

- Java synchronization classes can be split into several categories of capabilities

<table>
<thead>
<tr>
<th>Category</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic operations</td>
<td>An action that effectively happens all at once or not at all</td>
</tr>
<tr>
<td><strong>Mutual exclusion</strong></td>
<td>Allows concurrent access &amp; updates to shared data without race conditions</td>
</tr>
<tr>
<td>Coordination</td>
<td>Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.</td>
</tr>
<tr>
<td>Barrier synchronization</td>
<td>Ensures that any thread(s) must stop at a certain point &amp; cannot proceed until all other thread(s) reach this barrier</td>
</tr>
</tbody>
</table>
**Capabilities of Java Synchronizer Classes**

- Java synchronization classes can be split into several categories of capabilities

<table>
<thead>
<tr>
<th>Category</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Atomic operations</strong></td>
<td>An action that effectively happens all at once or not at all</td>
</tr>
<tr>
<td><strong>Mutual exclusion</strong></td>
<td>Allows concurrent access &amp; updates to shared data without race conditions</td>
</tr>
<tr>
<td><strong>Coordination</strong></td>
<td>Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.</td>
</tr>
<tr>
<td><strong>Barrier synchronization</strong></td>
<td>Ensures that any thread(s) must stop at a certain point &amp; cannot proceed until all other thread(s) reach this barrier</td>
</tr>
</tbody>
</table>
Capabilities of Java Synchronizer Classes

- Java synchronization classes can be split into several categories of capabilities

<table>
<thead>
<tr>
<th>Category</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic operations</td>
<td>An action that effectively happens all at once or not at all</td>
</tr>
<tr>
<td>Mutual exclusion</td>
<td>Allows concurrent access &amp; updates to shared data without race conditions</td>
</tr>
<tr>
<td>Coordination</td>
<td>Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.</td>
</tr>
<tr>
<td>Barrier synchronization</td>
<td>Ensures that any thread(s) must stop at a certain point &amp; cannot proceed until all other thread(s) reach this barrier</td>
</tr>
</tbody>
</table>
We focus more on Java synchronization mechanisms than on Java threading mechanisms.
Capabilities of Java Synchronizer Classes

- Synchronization complexity arises from coordinating the interactions of entities that run concurrently.
Capabilities 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"
Overview of Java Atomic Classes
Overview of Java Atomic Classes

- The `java.util.concurrent.atomic` package several types of atomic actions on objects.

### Package `java.util.concurrent.atomic`

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

See: Description

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AtomicBoolean</td>
<td>A boolean value that may be updated atomically.</td>
</tr>
<tr>
<td>AtomicInteger</td>
<td>An int value that may be updated atomically.</td>
</tr>
<tr>
<td>AtomicIntegerArray</td>
<td>An int array in which elements may be updated atomically.</td>
</tr>
<tr>
<td>AtomicIntegerFieldUpdater&lt;T&gt;</td>
<td>A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes.</td>
</tr>
<tr>
<td>AtomicLong</td>
<td>A long value that may be updated atomically.</td>
</tr>
<tr>
<td>AtomicLongArray</td>
<td>A long array in which elements may be updated atomically.</td>
</tr>
<tr>
<td>AtomicLongFieldUpdater&lt;T&gt;</td>
<td>A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes.</td>
</tr>
</tbody>
</table>

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html)
Overview of Java Atomic Classes

- The java.util.concurrent.atomic package several types of atomic actions on objects
  - Atomic variables
    - Provide lock-free thread-safe operations on single variables

See [docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html](docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html)
Overview of Java Atomic Classes

- The java.util.concurrent.atomic package several types of atomic actions on objects
  - **Atomic variables**
    - Provide lock-free thread-safe operations on single variables
    - e.g., AtomicLong supports atomic “compare-and-swap” operations

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicLong.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicLong.html)
Overview of Java Atomic Classes

- The java.util.concurrent.atomic package contains several types of atomic actions on objects
  - Atomic variables
  - `LongAdder`
    - Allows multiple threads to update a common sum efficiently under high contention

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html)
Overview of Java Synchronizer Classes
Overview of Java Synchronizer Classes

- The java.util.concurrent & java.util.concurrent.locks packages define many synchronizers
- e.g., java.util.concurrent & java.util.concurrent.locks

# Overview of Java Synchronizer Classes

- We cover Java language features & library classes for synchronization

<table>
<thead>
<tr>
<th>Java Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ReentrantLock</strong></td>
<td>A reentrant mutual exclusion lock that extends the built-in monitor lock capabilities</td>
</tr>
<tr>
<td><strong>ReentrantReadWriteLock</strong></td>
<td>Improves performance when resources are read much more often than written</td>
</tr>
<tr>
<td><strong>StampedLock</strong></td>
<td>A readers-writer lock that’s more efficient than ReentrantReadWriteLock</td>
</tr>
<tr>
<td><strong>Semaphore</strong></td>
<td>Maintains permits that control thread access to limited # of shared resources</td>
</tr>
<tr>
<td><strong>ConditionObject</strong></td>
<td>Allows Thread to block until a condition becomes true</td>
</tr>
<tr>
<td><strong>CountDownLatch</strong></td>
<td>Allows one or more Threads to wait until a set of operations being performed in other Threads complete</td>
</tr>
<tr>
<td><strong>CyclicBarrier</strong></td>
<td>Allows a set of Threads to all wait for each other to reach a common barrier point</td>
</tr>
<tr>
<td><strong>Phaser</strong></td>
<td>A more flexible reusable synchronization barrier</td>
</tr>
</tbody>
</table>

We show how these features & classes are implemented & used in Java & in practice
• These synchronizers are used extensively in Java applications & class libraries
Overview of Java Synchronizer Classes

- **ReentrantLock**
  - A mutual exclusion lock that extends built-in monitor lock capabilities

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html)
Overview of Java Synchronizer Classes

- **ReentrantLock**
  - A mutual exclusion lock that extends built-in monitor lock capabilities
  - “Reentrant” means that the thread holding the lock can reacquire it without deadlock

See en.wikipedia.org/wiki/Reentrancy_(computing)
Overview of Java Synchronizer Classes

- **ReentrantLock**
  - A mutual exclusion lock that extends built-in monitor lock capabilities
  - “Reentrant” means that the thread holding the lock can reacquire it without deadlock
  - Must be “fully bracketed”
    - A thread that acquires a lock must be the one to release it

See [jasleendailydiary.blogspot.com/2014/06/java-reentrant-lock.html](jasleendailydiary.blogspot.com/2014/06/java-reentrant-lock.html)
Overview of Java Synchronizer Classes

- **ReentrantReadWriteLock**
- Improves performance when resources read more often than written

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html)
ReentrantReadWriteLock

- Improves performance when resources read more often than written
- Has many features
- Both a blessing & a curse.

Reentranct

This lock allows both readers and writers to reacquire read or write locks in the style of a ReentrantLock. Non-reentrant readers are not allowed until all write locks held by the writing thread have been released.

Additionally, a writer can acquire the read lock, but not vice-versa. Among other applications, reentrancy can be useful when write locks are held during calls or callbacks to methods that perform reads under read locks. If a reader tries to acquire the write lock it will never succeed.

Lock downgrading

Reentrancy also allows downgrading from the write lock to a read lock, by acquiring the write lock, then the read lock and then releasing the write lock. However, upgrading from a read lock to the write lock is not possible.

Interuption of lock acquisition

The read lock and write lock both support interruption during lock acquisition.

Condition support

The write lock provides a Condition implementation that behaves in the same way, with respect to the write lock, as the Condition implementation provided by newCondition() does for ReentrantLock. This Condition can, of course, only be used with the write lock.

The read lock does not support a Condition and readLock().newCondition() throws UnsupportedOperationExcepusion.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html
Overview of Java Synchronizer Classes

- **StampedLock**
  - A readers-writer lock that's more efficient than a ReentrantReadWriteLock

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html)
Overview of Java Synchronizer Classes

- **StampedLock**
  - A readers-writer lock that’s more efficient than a ReentrantReadWriteLock
  - Supports “optimistic” reads

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html
Overview of Java Synchronizer Classes

- **StampedLock**
  - A readers-writer lock that’s more efficient than a ReentrantReadWriteLock
  - Supports “optimistic” reads
  - Also supports “lock upgrading”

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html)
Overview of Java Synchronizer Classes

- **Semaphore**
  - Maintains permits that control thread access to limited # of shared resources

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html)
Overview of Java Synchronizer Classes

• **Semaphore**
  - Maintains permits that control thread access to limited # of shared resources
  - Operations need not be fully bracketed..

---

### Semaphore

- `Semaphore(int)`
- `Semaphore(int, boolean)`
- `acquire(): void`
- `acquireUninterruptibly(): void`
- `tryAcquire(): boolean`
- `tryAcquire(long, TimeUnit): boolean`
- `release(): void`
- `acquire(int): void`
- `acquireUninterruptibly(int): void`
- `tryAcquire(int): boolean`
- `tryAcquire(int, long, TimeUnit): boolean`
- `release(int): void`
- `availablePermits(): int`
- `drainPermits(): int`
- `isFair(): boolean`
- `hasQueuedThreads(): boolean`
- `getQueueLength(): int`
- `toString()`

---

**PingPongThread**

- `ping: Semaphore`
- `pong: Semaphore`
- `run()`
- `print("ping")`
- `print("pong")`
- **1 Semaphores**
- **0 Semaphores**
Overview of Java Synchronizer Classes

- **ConditionObject**
  - Allows a thread to wait until some condition become true

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/AbstractQueuedSynchronizer.ConditionObject.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/AbstractQueuedSynchronizer.ConditionObject.html)
Overview of Java Synchronizer Classes

- **ConditionObject**
  - Allows a thread to wait until some condition become true
  - Always used in conjunction with a ReentrantLock

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/AbstractQueuedSynchronizer.ConditionObject.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/AbstractQueuedSynchronizer.ConditionObject.html)
Overview of Java Synchronizer Classes

- **CountDownLatch**
- Allows one or more threads to wait on the completion of operations in other threads

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountDownLatch.html
Overview of Java Synchronizer Classes

- **CyclicBarrier**
  - Allows a set of threads to all wait for each other to reach a common barrier point

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html)
Overview of Java Synchronizer Classes

- **Phaser**
  - A synchronization barrier that’s more flexible & reusable than CyclicBarrier & CountDownLatch

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/Phaser.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/Phaser.html)
Java Synchronizer Class
Usage Considerations
Java Synchronizer Class Usage Considerations

- Choosing between these synchronizers involve understanding various tradeoffs between *performance* & *productivity*.
Java Synchronizer Class Usage Considerations

- Choosing between these synchronizers involve understanding various tradeoffs between *performance & productivity*
- Some synchronizers (or synchronizer methods) have more overhead
  - e.g., spin locks vs. sleep locks vs. hybrid locks

Choosing between these synchronizers involve understanding various tradeoffs between performance & productivity.

- Some synchronizers (or synchronizer methods) have more overhead.
- Some synchronizers are harder to program correctly than others.
  - e.g., risk of deadlock from non-reentrant locking semantics.

Deadlocks are problematic in object-oriented frameworks due to callbacks & complex control flows.

See [en.wikipedia.org/wiki/Deadlock](en.wikipedia.org/wiki/Deadlock)
Java Synchronizer Class Usage Considerations

- Java synchronizers differ from Java built-in monitor objects
Java Synchronizer Class Usage Considerations

- Java synchronizers differ from Java built-in monitor objects, e.g.
  - They are largely written in Java rather than C/C++
Java Synchronizer Class Usage Considerations

- Java synchronizers differ from Java built-in monitor objects, e.g.
  - They are largely written in Java rather than C/C++
  - Some low-level methods written in native C/C++
    - e.g., `compareAndSwapInt()`, `park()`, `unpark()`, etc.

Concurrence

And few words about concurrency with `Unsafe`. `compareAndSwap` methods are atomic and can be used to implement high-performance lock-free data structures.

For example, consider the problem to increment value in the shared object using lot of threads.

First we define simple interface `Counter`:

```java
interface Counter {
    void increment();
    long getCounter();
}
```

Then we define worker thread `CounterClient`, that uses `Counter`:

```java
class CounterClient implements Runnable {
    private Counter c;
    private int num;

    public CounterClient(Counter c, int num) {
        this.c = c;
        this.num = num;
    }

    @Override
    public void run() {
        for (int i = 0; i < num; i++) {
            c.increment();
        }
    }
}
```
Java Synchronizer Class Usage Considerations

- Java synchronizers differ from Java built-in monitor objects, e.g.
  - They are largely written in Java rather than C/C++
  - They provide many more features & have more powerful semantics
End of Overview of Java Synchronizer Classes