Learning Objectives in this Part of the Lesson

• Understand the key Java languages features & library classes for synchronizing Java threads in concurrent programs

```java
default 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. The framework permits much greater flexibility in the use of locks and conditions, at the expense of more awkward syntax. The **`Lock`** interface supports locking disciplines that differ in semantics (reentrant, fair, etc), and that can be used in non-block-structured contexts including hand-over-hand and lock reordering algorithms. The main implementation is **`ReentrantLock`**.
```

```java
default package java.util.concurrent

Utility classes commonly useful in concurrent programming. This package includes a few small standardized extensible frameworks, as well as some classes that provide useful functionality and are otherwise tedious or difficult to implement. Here are brief descriptions of the main components. See also the **`java.util.concurrent.locks`** and **`java.util.concurrent.atomic`** packages.
```
Capabilities Provided by Java Synchronizers
Capabilities Provided by Java Synchronizers

- Java’s synchronization mechanisms address inherent complexities of concurrent software

Inherent complexities are the “rocket science” of software development
Capabilities Provided by Java Synchronizers

- Java’s synchronization mechanisms address inherent complexities of concurrent software

Ensure interactions between computations
- don’t corrupt shared data &
- occur in the right order, at the right time, & under the right conditions

See en.wikipedia.org/wiki/Synchronization_(computer_science)
Capabilities Provided by Java Synchronizers

- Java’s synchronization mechanisms address inherent complexities of concurrent software, e.g.
  - Ensure **mutual exclusion** in critical sections to ensure key properties

See [en.wikipedia.org/wiki/Mutual_exclusion](en.wikipedia.org/wiki/Mutual_exclusion)
Capabilities Provided by Java Synchronizers

- Java’s synchronization mechanisms address inherent complexities of concurrent software, e.g.
  - Ensure **mutual exclusion** in critical sections to ensure key properties
  - *Atomic ordering (linearization)*

See [en.wikipedia.org/wiki/Linearizability](en.wikipedia.org/wiki/Linearizability)
Java’s synchronization mechanisms address inherent complexities of concurrent software, e.g.

- **Ensure **mutual exclusion** in critical sections to ensure key properties**
- **Atomic ordering (linearization)**
- Operations on a field in thread$_1$ occur all at once wrt operations on the field in thread$_2$..n

### Capabilities Provided by Java Synchronizers

<table>
<thead>
<tr>
<th></th>
<th>Thread$_1$</th>
<th>Thread$_2$</th>
<th>Long field</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialized</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>read field</td>
<td>←</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>increase field by 1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write back</td>
<td>→</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>read field</td>
<td>←</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>increase field by 1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write back</td>
<td>→</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

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

See [docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html](docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html)
Race conditions occur when a program depends on the sequence or timing of threads for it to operate properly.

Capabilities Provided by Java Synchronizers

- Java’s synchronization mechanisms address inherent complexities of concurrent software, e.g.
  - Ensure **mutual exclusion** in critical sections to ensure key properties
    - Atomic ordering (linearization)
  - **Avoid race conditions**

See [en.wikipedia.org/wiki/Race_condition#Software](en.wikipedia.org/wiki/Race_condition#Software)
Java’s synchronization mechanisms address inherent complexities of concurrent software, e.g.

- Ensure **mutual exclusion** in critical sections to ensure key properties
  - Atomic ordering (linearization)
- Avoid race conditions, e.g.
  - Read/write conflicts
  - If one thread reads while another thread writes concurrently, the field that’s read may be inconsistent

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

<table>
<thead>
<tr>
<th></th>
<th>Thread₁</th>
<th>Thread₂</th>
<th>Long field</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialized</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>read field</td>
<td></td>
<td>←</td>
<td>0</td>
</tr>
<tr>
<td>increase field by 1</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>write back</td>
<td></td>
<td>←—</td>
<td>0 or 1?</td>
</tr>
</tbody>
</table>
Java’s synchronization mechanisms address inherent complexities of concurrent software, e.g.:

- **Ensure mutual exclusion** in critical sections to ensure key properties
  - Atomic ordering (linearization)
  - Avoid race conditions, e.g.
    - Read/write conflicts
    - Write/write conflicts
    - If two threads try to write to same field concurrently, the result may be inconsistent

### Capabilities Provided by Java Synchronizers

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Long field</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialized</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>read field</td>
<td>← 0</td>
<td>read field</td>
<td>← 0</td>
</tr>
<tr>
<td>increase field by 2</td>
<td></td>
<td>increase field by 1</td>
<td>0</td>
</tr>
<tr>
<td>write back</td>
<td>write back</td>
<td>→ 1 or 2?</td>
<td></td>
</tr>
</tbody>
</table>
Java’s synchronization mechanisms address inherent complexities of concurrent software, e.g.

- Ensure **mutual exclusion** in critical sections to ensure key properties
  - Atomic ordering (linearization)
- Avoid race conditions, e.g.
  - 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](en.wikipedia.org/wiki/Memory_ordering)
Java’s synchronization mechanisms address inherent complexities of concurrent software, e.g.

- **Ensure mutual exclusion** in critical sections to ensure key properties
- **Coordinate** multiple threads to ensure computations run properly
Java’s synchronization mechanisms address inherent complexities of concurrent software, e.g.

- **Ensure mutual exclusion** in critical sections to ensure key properties

- **Coordinate** multiple threads to ensure computations run properly, e.g.
  - In the right order

---

```plaintext
% java PingPong
Ready...Set...Go!
Pong!(1)
Pong!(2)
Pong!(3)
Pong!(4)
Pong!(5)
Pong!(6)
Pong!(7)
Pong!(8)
Pong!(9)
Pong!(10)
Done!
```
Java’s synchronization mechanisms address inherent complexities of concurrent software, e.g.

- **Ensure** *mutual exclusion* in critical sections to ensure key properties

- **Coordinate** multiple threads to ensure computations run properly, e.g.
  - In the right order
  - At the right time

See [en.wikipedia.org/wiki/Real-time_computing](en.wikipedia.org/wiki/Real-time_computing)
Java’s synchronization mechanisms address inherent complexities of concurrent software, e.g.

- Ensure **mutual exclusion** in critical sections to ensure key properties
- **Coordinate** multiple threads to ensure computations run properly, e.g.
  - In the right order
  - At the right time
  - Under the right conditions
Capabilities Provided by Java Synchronizers

- When used properly, synchronizers avoid concurrency hazards.
When used properly, synchronizers avoid concurrency hazards, e.g.

- Ensure object/class fields have valid state even when accessed concurrently by multiple threads.
Capabilities Provided by Java Synchronizers

- When used properly, synchronizers avoid concurrency hazards, e.g.
  - Ensure object/class fields have *valid state* even when accessed concurrently by multiple threads
  - “Valid state” means that each method preserves key invariants
Capabilities Provided by Java Synchronizers

- When used properly, synchronizers avoid concurrency hazards, e.g.
  - Ensure object/class fields have valid state even when accessed concurrently by multiple threads
  - “Valid state” means that each method preserves key invariants
    - e.g., a priority queue ensures sorted order in a concurrent program

![Diagram of Java synchronizers]

- Critical Section
- Queue of waiting threads
- Running thread
- Sorted in ascending order

(priority 1) -> (priority 5) -> (priority 9)
When used properly, synchronizers avoid concurrency hazards, e.g.

- Ensure object/class fields have valid state even when accessed concurrently by multiple threads.

“Valid state” means that each method preserves key invariants.

- e.g., a priority queue ensures sorted order in a concurrent program.

Invariants need not be satisfied when access to an object is protected by a synchronizer.
Capabilities Provided by Java Synchronizers

- Java’s memory model defines semantics of multi-threaded access to shared memory

See www.cs.umd.edu/users/pugh/java/memoryModel/jsr-133-faq.html
Java’s memory model defines semantics of multi-threaded access to shared memory, e.g.

- Which instruction reorderings are allowed in memory

There are a number of potential sources of reordering, e.g., the Java compiler, the JIT, & processor caches, etc.
Java’s memory model defines semantics of multi-threaded access to shared memory, e.g.

- Which instruction reorderings are allowed in memory
- Should not be overly restrictive, to enable hardware optimizations

```
x = y = 0
```

```
x = 1
```

```
y = 1
```

```
j = y
```

```
i = x
```

```
start threads
```

```
It can end up that `i = 0 & j = 0` due to local caching effects in `Thread_1` & `Thread_2`
```

See [en.wikipedia.org/wiki/Memory_ordering](https://en.wikipedia.org/wiki/Memory_ordering)
Java’s memory model defines semantics of multi-threaded access to shared memory, e.g.

- Which instruction reorderings are allowed in memory
- Which program outputs may occur in a correct JVM implementation

See [docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.3](docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.3)
Java’s memory model defines semantics of multi-threaded access to shared memory, e.g.

- Which instruction reorderings are allowed in memory
- Which program outputs may occur in a correct JVM implementation
- Should not be too generous such that values appear randomly!

```
x = y = 0
```

```
start threads
```

```
Thread1
r1 = x
```

```
Thread2
r2 = y
```

```
y = r1
```

```
x = r2
```

```
Must not result in r1 = r2 = 42!
```
Capabilities Provided by Java Synchronizers

- Java’s memory model defines semantics of multi-threaded access to shared memory, e.g.
  - Which instruction reorderings are allowed in memory
  - Which program outputs may occur in a correct JVM implementation

Fortunately, you needn’t understand all these memory model details – you just need to know how to use Java synchronizers & monitor objects properly!!
Capabilities Provided by Java Synchronizers

- The rest of this lesson summarizes key Java language features & class libraries used to synchronize interactions between threads.

We only show a few Java code examples in this summary.
Capabilities Provided by Java Synchronizers

Java synchronization features & libraries can be split into several categories:

<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>Prevents simultaneous access to a shared resource</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 features & libraries can be split into several categories:

### Capabilities Provided by Java Synchronizers

<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>Prevents simultaneous access to a shared resource</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 features & libraries can be split into several categories.

<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>Prevents simultaneous access to a shared resource</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 Provided by Java Synchronizers

Java synchronization features & libraries can be split into several categories:

<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>Prevents simultaneous access to a shared resource</td>
</tr>
<tr>
<td><strong>Coordination</strong></td>
<td><strong>Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.</strong></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 Provided by Java Synchronizers

Java synchronization features & libraries can be split into several categories:

<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>Prevents simultaneous access to a shared resource</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><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 Provided by Java Synchronizers

- We cover Java language features & library classes for synchronization

<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>

We show how these features & classes are implemented & used in Java & in practice
These synchronizers are used extensively in Java applications & class libraries.
Capabilities Provided by Java Synchronizers

- We focus more on Java synchronization mechanisms than on Java threading mechanisms
• The complexity of synchronization arises largely from coordinating the interactions of multiple entities that run concurrently.
End of Overview of Java Synchronizers (Part 1)
1. Which of the following are inherent complexities of synchronization
   a. Programming with limited debugging tools & environments
   b. Ensuring mutual exclusion in critical sections to ensure key properties
   c. Coordinating multiple threads to ensure computations run properly
   d. Using low-level APIs written in C

2. Match following categories of synchronization mechanisms to their corresponding definitions
   a. Coordination
     1. An action that effectively happens all at once or not at all
   b. Atomic operations
     2. Prevents simultaneous access to a shared resource
   c. Barrier synchronization
     3. Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.
   d. Mutual exclusion
     4. Ensures that any thread(s) must stop at a certain point & cannot proceed until all other thread(s) reach this barrier