1. **ThreadLocal class.**
2. **ThreadLocalRandom class.**
3. **Lock Interface**
4. **ReentrantLock**
5. **ReentrantReadWriteLock**
6. **StampedLock**
7. **ReadWriteLock Interface**

Java.lang. ThreadLocal :- Java ThreadLocal is used to create thread local variables which can only be read and written by the same thread. We know that all threads of an Object share its variables, so the variable is not thread safe. We can use synchronization for thread safety but if we want to avoid synchronization, we can use ThreadLocal variables.

For example, if two threads are accessing code having reference to same threadLocal variable then each thread will not see any modification to threadLocal variable done by other thread.

- Every thread has its own ThreadLocal variable and they can use it's **get ()** and **set (**) methods to get the default value or change its value local to Thread.
- ThreadLocal **instances are typically private static fields** in classes that wish to associate state with a thread.

ThreadLocal Methods

Method	Description
public T get()	Returns the value in the current thread's copy of this thread-local variable.
protected T initialValue()	Returns the current thread's "initial value" for this thread-local variable.
public void remove()	Removes the current thread's value for this thread-local variable.
public void set(T value)	Sets the current thread's copy of this thread-local variable to the specified value.
static<S>ThreadLocal<S>withInitial (Supplier<? extends S> supplier)	Creates a thread local variable.

Do not use ThreadLocal with ExecutorService:- If we want to use an ExecutorService and submit a Runnable to it, using ThreadLocal will yield non-deterministic results – because we do not have a guarantee that every Runnable action for a given userId will be handled by the same thread every time it is executed.

Because of that, our ThreadLocal will be shared among different userIds. That's why we should not use a TheadLocal together with ExecutorService. It should only be used when we have full control over which thread will pick which runnable action to execute.

Java.util.concurrent.ThreadLocalRandom :- Normally to generate Random numbers, we either do Create an instance of java.util.Random OR Math.random() - which internally creates an instance of java.util.Random on first invocation. However, in a concurrent applications usage of above leads to contention issues

Random is thread safe for use by multiple threads. But if multiple threads use the same instance of Random, the same seed is shared by multiple threads. It leads to contention between multiple threads and so to performance degradation.

ThreadLocalRandom is solution to above problem. ThreadLocalRandom has a Random instance per thread and safeguards against contention.

ThreadLocalRandom is more efficient in a highly concurrent environment.

ThreadLocalRandom over Random

ThreadLocalRandom is a combination of ThreadLocal and Random classes, which is isolated to the current thread. Thus, it achieves better performance in a multithreaded environment by simply avoiding any concurrent access to the Random objects.

The random number obtained by one thread is not affected by the other thread, whereas java.util.Random provides random numbers globally.

Also, unlike Random, ThreadLocalRandom doesn't support setting the seed explicitly. Instead, it overrides the setSeed(long seed) method inherited from Random to always throw an UnsupportedOperationException if called.

ThreadLocalRandom Methods :-

Method	Description
public static ThreadLocalRandom current()	Returns the current thread's ThreadLocalRandom.

protected int next(int bits)	Generates the next pseudorandom number.
public double nextDouble(double n)	Returns a pseudorandom, uniformly distributed double value between 0 (inclusive) and the specified value (exclusive).
public double nextDouble(double least, double bound)	Returns a pseudorandom, uniformly distributed value between the given least value (inclusive) and bound (exclusive).
public int nextInt(int least, int bound)	Returns a pseudorandom, uniformly distributed value between the given least value (inclusive) and bound (exclusive).
public long nextLong(long n)	Returns a pseudorandom, uniformly distributed value between 0 (inclusive) and the specified value (exclusive).
public long nextLong(long least, long bound)	Returns a pseudorandom, uniformly distributed value between the given least value (inclusive) and bound (exclusive).
public void setSeed(long seed)	Throws UnsupportedOperationException.

Java.util.concurrent.locks.Lock :- A java.util.concurrent.locks.Lock interface is used to as a thread synchronization mechanism similar to synchronized blocks. New Locking mechanism is more flexible and provides more options than a synchronized block. 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. However, some locks may allow concurrent access to a shared resource, such as the read lock of a ReadWriteLock.

The use of synchronized methods or statements provides access to the implicit monitor lock associated with every object, but forces all lock acquisition and release to occur in a block-structured way: when multiple locks are acquired they must be released in the opposite order, and all locks must be released in the same lexical scope in which they were acquired.

While the scoping mechanism for synchronized methods and statements makes it much easier to program with monitor locks, and helps avoid many common programming errors involving locks, there are occasions where you need to work with locks in a more flexible way. For example, some algorithms for traversing concurrently accessed data structures require the use of "hand-over-hand" or "chain locking": you acquire the lock of node A, then node B, then release A and acquire C, then release B and acquire D and so on. Implementations of the Lock interface enable the use of such techniques by allowing a lock to be acquired and released in different scopes, and allowing multiple locks to be acquired and released in any order.

With this increased flexibility comes additional responsibility. The absence of block-structured locking removes the automatic release of locks that occurs with synchronized methods and statements. When locking and unlocking occur in different scopes, care must be taken to ensure that all code that is executed while the lock is held is protected by try-finally or try-catch to ensure that the lock is released when necessary.

Lock implementations provide additional functionality over the use of synchronized methods and statements by providing a non-blocking attempt to acquire a lock (tryLock()), an attempt to acquire the lock that can be interrupted (lockInterruptibly(), and an attempt to acquire the lock that can timeout (tryLock(long, TimeUnit)).

A Lock class can also provide behavior and semantics that is quite different from that of the implicit monitor lock, such as guaranteed ordering, non-reentrant usage, or deadlock detection. If an implementation provides such specialized semantics then the implementation must document those semantics.

Note that Lock instances are just normal objects and can themselves be used as the target in a synchronized statement. Acquiring the monitor lock of a Lock instance has no specified relationship with invoking any of the lock() methods of that instance. It is recommended that to avoid confusion you never use Lock instances in this way, except within their own implementation.

Except where noted, passing a null value for any parameter will result in a NullPointerException being thrown.

Memory Synchronization

All Lock implementations must enforce the same memory synchronization semantics as provided by the built-in monitor lock :-

1. A successful lock operation has the same memory synchronization effects as a successful Lock action.

2. A successful unlock operation has the same memory synchronization effects as a successful Unlock action.

Unsuccessful locking and unlocking operations, and reentrant locking/unlocking operations, do not require any memory synchronization effects.

Implementation Considerations

The three forms of lock acquisition (interruptible, non-interruptible, and timed) may differ in their performance characteristics, ordering guarantees, or other implementation qualities. Further, the ability to interrupt the ongoing acquisition of a lock may not be available in a given Lock class. Consequently, an implementation is not required to define the same guarantees or semantics for all three forms of lock acquisition, nor is it required to support interruption of an ongoing lock acquisition. An implementation is required to clearly document the semantics and guarantees provided by each of the locking methods. It must also obey the interruption semantics as defined in this interface, to the extent that interruption of lock acquisition is supported: which is either totally, or only on method entry.

As interruption generally implies cancellation, and checks for interruption are often infrequent, an implementation can favor responding to an interrupt over normal method return. This is true even if it can be shown that the interrupt occurred after another action may have unblocked the thread. An implementation should document this behavior.

Main differences between a Lock and a synchronized block are following –

- **Guarantee of sequence** – Synchronized block does not provide any guarantee of sequence in which waiting thread will be given access. Lock interface handles it.
- **No timeout** – Synchronized block has no option of timeout if lock is not granted. Lock interface provides such option.
- **Single method** – Synchronized block must be fully contained within a single method whereas a lock interface's methods lock() and unlock() can be called in different methods.
- A s*ynchronized block* doesn't support the fairness, any thread can acquire the lock once released, no preference can be specified. **We can achieve fairness within the *Lock* APIs by specifying the *fairness* property**. It makes sure that longest waiting thread is given access to the lock
- A thread gets blocked if it can't get an access to the synchronized *block*. **The *Lock* API provides *tryLock()* method. The thread acquires lock only if**

it's available and not held by any other thread. This reduces blocking time of thread waiting for the lock

- A thread which is in "waiting" state to acquire the access to *synchronized block*, can't be interrupted. **The *Lock* API provides a method *lockInterruptibly()* which can be used to interrupt the thread when it's waiting for the lock**

Lock Methods

Following is the list of important methods available in the Lock class.

Method	Description
public void lock()	Acquires the lock.
public void lockInterruptibly()	Acquires the lock unless the current thread is interrupted.
public Condition newCondition()	Returns a new Condition instance that is bound to this Lock instance.
public boolean tryLock()	Acquires the lock only if it is free at the time of invocation.
public boolean tryLock()	Acquires the lock only if it is free at the time of invocation.
public boolean tryLock(long time, TimeUnit unit)	Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted.
public void unlock()	Releases the lock.

Lock implementations: -

1. **ReentrantLock**
2. **ReentrantReadWriteLock**
3. **StampedLock**

ReentrantLock :- ReentrantLock class implements the Lock interface. It offers the same concurrency and memory semantics, as the implicit monitor lock accessed using synchronized methods and statements, with extended capabilities.

ReentrantReadWriteLock :- ReentrantReadWriteLock class implements the ReadWriteLock interface. Let's see rules for acquiring the ReadLock or WriteLock by a thread:

- **Read Lock –** if no thread acquired the write lock or requested for it then multiple threads can acquire the read lock
- **Write Lock –** if no threads are reading or writing then only one thread can acquire the write lock

StampedLock :- StampedLock is introduced in Java 8. It also supports both read and write locks. However, lock acquisition methods return a stamp that is used to release a lock or to check if the lock is still valid. Another feature provided by StampedLock is optimistic locking. Most of the time read operations doesn't need to wait for write operation completion and as a result of this, the full-fledged read lock isn't required.

ReentrantReadWriteLock has some severe issues with starvation if not handled properly (using fairness may help, but it may be an overhead and compromise throughput). For example, a number of reads but very few writes can cause the writer thread to fall into starvation. Make sure to analyze your setup properly to know how many reads/writes are present before choosing ReadWriteLock.

But it's within this ReadWriteLock series that Java introduced StampedLock.

StampedLock is made of a stamp and mode, where your lock acquisition method returns a stamp, which is a long value used for unlocking within the finally block. If the stamp is ever zero, that means there's been a failure to acquire access. StampedLock is all about giving us a possibility to perform optimistic reads.

Keep one thing in mind: StampedLock is not reentrant, so each call to acquire the lock always returns a new stamp and blocks if there's no lock available, even if the same thread already holds a lock, which may lead to deadlock.

Another point to note is that ReadWriteLock has two modes for controlling the read/write access while StampedLock has three modes of access:

Reading: Method 'public long readLock()' actually acquires a non-exclusive lock, and it blocks, if necessary, until available. It returns a stamp that can be used to unlock or convert the mode.

Writing: Method 'public long writeLock()' acquires an exclusive lock, and it blocks, if necessary, until available. It returns a stamp that can be used to unlock or convert the mode.

Optimistic reading: Method 'public long tryOptimisticRead()' acquires a non-exclusive lock without blocking only when it returns a stamp that can be later validated. Otherwise the value is zero if it doesn't acquire a lock. This is to allow read operations. After calling the tryOptimisticRead() method, always check if the stamp is valid using the 'lock.validate(stamp)' method, as the optimistic read lock doesn't prevent another thread from getting a write lock, which will make the optimistic read lock stamp's invalid.

You might be wondering how to convert the mode and when. Well:

- There could be a situation when you acquired the write lock and written something and you wanted to read in the same critical section. So, as to not break the potential concurrent access, we can use the 'tryConvertToReadLock(long stamp)' method to acquire read access.
- Now suppose you acquired the read lock, and after a successful read, you wanted to change the value. To do so, you need a write lock, which you can acquire using the 'tryConvertToWriteLock(long stamp)' method.

Note: One thing to note is that the tryConvertToReadLock and tryConvertToWriteLock methods will not block and may return the stamp as zero, which means these methods' calls were not successful.

Condition class: - The Condition class provides the ability for a thread to wait for some condition to occur while executing the critical section.

This can occur when a thread acquires the access to the critical section but doesn't have the necessary condition to perform its operation. For example, a reader thread can get access to the lock of a shared queue, which still doesn't have any data to consume.

Traditionally Java provides wait (), notify () and notifyAll() methods for thread intercommunication. Conditions have similar mechanisms, but in addition, we can specify multiple conditions.

Condition Methods

Method	Description
public void await()	Causes the current thread to wait until it is signalled or interrupted.
public boolean await(long time, TimeUnit unit)	Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.
public long awaitNanos(long nanosTimeout)	Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.
public long awaitUninterruptibly()	Causes the current thread to wait until it is signalled.
public long awaitUntil()	Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses.
public void signal()	Wakes up one waiting thread.

public void signalAll()	Wakes up all waiting threads.

Benefits of ReentrantLock in Java: -

- Ability to lock interruptibly.
- Ability to timeout while waiting for lock.
- Power to create fair lock.
- API to get list of waiting thread for lock.
- Flexibility to try for lock without blocking.

Disadvantages of ReentrantLock in Java: - Major drawback of using ReentrantLock in Java is wrapping method body inside try-finally block, which makes code unreadable and hides business logic. It's really cluttered, and I hate it most, though IDE like Eclipse and Netbeans can add those try catch block for you. Another disadvantage is that, now programmer is responsible for acquiring and releasing lock, which is a power but also opens gate for new subtle bugs, when programmer forget to release the lock in finally block.

ReentrantReadWriteLock implementation guarantees the following behaviors: -

1. Multiple threads can read the data at the same time, as long as there's no thread is updating the data.
2. Only one thread can update the data at a time, causing other threads (both readers and writers) block until the write lock is released.
3. If a thread attempts to update the data while other threads are reading, the write thread also blocks until the read lock is released.

So ReadWriteLockcan be used to add concurrency features to a data structure, but it doesn't guarantee the performance because it depends on various factors: how the data structure is designed, the contention of reader and writer threads at real time, CPU architecture (single core or multicores), etc.

Similar to ReentrantLock, the ReentrantReadWriteLock allows a thread to acquire the read lock or write lock multiple times recursively, thus the word "Reentrant".

To conclude, remember the following key points:

- ReadWriteLockallows multiple concurrent readers abut only one exclusive writer.
- ReentrantReadWriteLock is an implementation of ReadWriteLock. In addition, it allows a reader/writer thread acquire a read lock/write lock multiple times recursively (reentrancy).

- Use the read lock to safeguard code that performs read operations and use the write lock to protect access to code that performs update operation.
- In practice, ReadWriteLock can be used to increase throughput for shared data structure like cache or dictionary-like data which the update is infrequent and read is more frequent.

java.util.concurrent.locks.ReadWriteLock interface: -

A java.util.concurrent.locks.ReadWriteLock interface allows multiple threads to read at a time but only one thread can write at a time.

- **Read Lock** – If no thread has locked the ReadWriteLock for writing then multiple thread can access the read lock.
- **Write Lock** – If no thread is reading or writing, then one thread can access the write lock.

Lock Methods

Method	Description
public Lock readLock()	Returns the lock used for reading.
public Lock writeLock()	Returns the lock used for writing.