**Advanced Multi-Threading Concepts**

**1. Using @Async for Asynchronous Execution**

* Spring provides the @Async annotation to execute methods asynchronously using a thread pool**.**
* **✅ Use Case:** Useful for executing long-running tasks like file processing, external API calls, or database operations.

@Service

public class AsyncService {

@Async

public CompletableFuture<String> asyncMethod() {

return CompletableFuture.supplyAsync(() -> "Task Executed");

}

}

**2. Using CompletableFuture for Non-Blocking Execution**

* **CompletableFuture** provides better control over async tasks and supports parallel execution.
* ✅ **Use Case:** Ideal for **chaining async operations** and handling parallel executions efficiently.

public CompletableFuture<String> processTask() {

return CompletableFuture.supplyAsync(() -> "Processing...")

.thenApply(result -> result + " Completed!");

}

**3. Using ForkJoinPool for Parallel Processing**

* The ForkJoinPool framework is useful for **divide-and-conquer tasks**, efficiently splitting workloads into multiple threads.
* ✅ **Use Case:** Best for **batch processing, processing large datasets, recursive tasks, and stream-based parallelism**.

ForkJoinPool forkJoinPool = new ForkJoinPool();

forkJoinPool.submit(() -> IntStream.range(1, 10).parallel().forEach(System.out::println));

**4. Using ExecutorService for Thread Management**

* **ExecutorService** provides better control over thread pools and can execute tasks asynchronously.
* ✅ **Use Case:** Used for **thread pool management, background processing, and concurrent tasks**.

ExecutorService executorService = Executors.newFixedThreadPool(5);

executorService.submit(() -> System.out.println("Executing Task"));

**5. Using ThreadPoolTaskExecutor for Custom Thread Pools**

* Spring provides **ThreadPoolTaskExecutor** for fine-grained thread pool configurations.
* ✅ **Use Case:** Ideal for **handling high-load applications with custom thread pools**.

@Bean

public Executor taskExecutor() {

ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();

executor.setCorePoolSize(5);

executor.setMaxPoolSize(10);

executor.setQueueCapacity(100);

executor.initialize();

return executor; }

**6. Using Parallel Streams for Concurrent Data Processing**

* Java Streams support parallel execution for **faster data processing**.
* ✅ **Use Case:** Best for **large data processing and concurrent operations on collections**.

List<String> list = Arrays.asList("A", "B", "C", "D");

list.parallelStream().forEach(System.out::println);

**7. Using ReentrantLock for Explicit Locking**

* **ReentrantLock** is an advanced locking mechanism that provides better control than synchronized.
* **Reentrant** means that locks are bound to the current thread
* ✅ **Use Case:** Ideal for **critical sections requiring manual lock handling**.

private final ReentrantLock lock = new ReentrantLock();

int count = 0;

public void process() {

lock.lock();

try {

count++;

} finally {

lock.unlock();

}

}

**8. Using Semaphore for Controlling Concurrent Access**

* Semaphore helps limit the number of concurrent threads executing a section of code.
* A semaphore is a data structure that maintains a set of permits that have to be acquired by competing threads.
* **Semaphores can be used to control how many threads access a critical resource simultaneously**.
* ✅ **Use Case:** Useful for **limiting concurrent database connections or API rate limiting**.

Semaphore semaphore = new Semaphore(2);

public void accessResource() throws InterruptedException {

semaphore.acquire();

try {

System.out.println("Accessing resource...");

} finally {

semaphore.release();

}

}

**9. Using ReadWriteLock**

* Better performance than ReentrantLock in **read-heavy scenarios** due to concurrent read access
* Maintains a pair of locks, **one for reading** (shared) and **one for writing** (exclusive).
* Multiple threads can acquire the read lock simultaneously if no thread holds the write lock. Only one thread can acquire the write lock at a time, and no other threads can hold either read or write locks during this period.

private final ReadWriteLock lock = new ReentrantReadWriteLock();

public void resource(){}

lock.readLock().lock();

try {

return map.get(key);

} finally {

lock.readLock().unlock();

}

}

public void resource(){}

lock.writeLock().lock();

try {

return map.get(key);

} finally {

lock.writeLock().unlock();

}

}

**Using Callable and Future for Concurrent Tasks**

**Explanation:**

* Callable is similar to Runnable, but it can return a result or throw an exception.
* Future is used to represent the result of an asynchronous computation (i.e., a task being executed in a different thread).

**Steps:**

1. Implement Callable and override the call() method.
2. Submit the Callable to an ExecutorService.
3. Retrieve the result using Future.get().

class MyCallable implements Callable<String> {

@Override

public String call() throws Exception {

return "Callable task result.";

}

public static void main(String[] args) throws Exception {

ExecutorService executor = Executors.newCachedThreadPool();

MyCallable task = new MyCallable();

Future<String> future = executor.submit(task); // Submit the task

System.out.println("Result: " + future.get()); // Get the result from the task

executor.shutdown();

}

}

**Cyclic Barrier**

* CyclicBarrier is used to make sure all threads wait for each other before proceeding.
* When all threads are ready, the barrier action is triggered.

import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierExample {

public static void main(String[] args) throws InterruptedException {

CyclicBarrier barrier = new CyclicBarrier(3, () -> System.out.println("All threads are ready. Let's start!"));

Runnable task = () -> {

try {

System.out.println(Thread.currentThread().getName() + " is ready.");

barrier.await(); // Wait for other threads to be ready

} catch (Exception e) {

e.printStackTrace();

}

};

// Start 3 threads

new Thread(task).start();

new Thread(task).start();

new Thread(task).start();

}

}

**CountDownLatch**

* **CountDownLatch** is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.
* It maintains an internal counter that is decremented each time a thread calls countDown(). When the counter reaches zero, the waiting threads can proceed.
* It is useful when you want to block some threads until a certain condition is met (e.g., a number of other threads finish execution).
* Once the count reaches zero, all waiting threads are released.
* It cannot be reset once its count reaches zero.

import java.util.concurrent.CountDownLatch;

public class CountDownLatchExample {

public static void main(String[] args) throws InterruptedException {

int numThreads = 3;

CountDownLatch latch = new CountDownLatch(numThreads);

// Worker thread that will decrement the count on latch

Runnable worker = () -> {

try {

System.out.println(Thread.currentThread().getName() + " is working.");

Thread.sleep(1000); // Simulating some work

latch.countDown(); // Decrement the latch count

System.out.println(Thread.currentThread().getName() + " is done.");

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

};

// Starting worker threads

for (int i = 0; i < numThreads; i++) {

new Thread(worker).start();

}

// Main thread waits for the latch to reach zero

latch.await(); // Blocks until latch count is zero

System.out.println("All worker threads have finished.");

}

}

**ReentrantReadWriteLock and ReadWriteLock**

**Explanation:**

* **ReadWriteLock** is an interface that defines a lock that allows multiple threads to read a resource concurrently but provides exclusive access to a thread for writing.
* **ReentrantReadWriteLock** is a concrete implementation of the ReadWriteLock interface, providing a more sophisticated and thread-friendly locking mechanism.

**Key Points of ReadWriteLock:**

* It provides two locks: **read lock** and **write lock**.
  + **Read lock** allows multiple threads to read the resource concurrently.
  + **Write lock** is exclusive, meaning only one thread can write, and no threads can read while a write is in progress.

**Key Points of ReentrantReadWriteLock:**

* It is **reentrant**, meaning a thread that holds a read or write lock can re-enter and acquire the same lock without deadlocking.
* It allows **fairness**, which means the lock grants access to threads in the order they requested it (optional).

public class ReadWriteLockExample {

private static final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

private static int sharedResource = 0;

// Method to perform a read operation

public static void read() {

lock.readLock().lock(); // Acquiring read lock

try {

System.out.println("Reading shared resource: " + sharedResource);

} finally {

lock.readLock().unlock(); // Releasing read lock

}

}

// Method to perform a write operation

public static void write(int value) {

lock.writeLock().lock(); // Acquiring write lock

try {

sharedResource = value;

System.out.println("Writing shared resource: " + sharedResource);

} finally {

lock.writeLock().unlock(); // Releasing write lock

}

}

public static void main(String[] args) throws InterruptedException {

// Multiple read threads

Thread readThread1 = new Thread(ReadWriteLockExample::read);

Thread readThread2 = new Thread(ReadWriteLockExample::read);

Thread writeThread = new Thread(() -> write(42));

readThread1.start();

readThread2.start();

Thread.sleep(100); // Ensure both readers start before writer

writeThread.start();

readThread1.join();

readThread2.join(); writeThread.join(); }}

**Concurrency** is the ability of a program to execute several computations simultaneously.

The **ExecutorService** interface is an asynchronous execution mechanism which is capable of executing tasks in the background. It is similar to **threadPool** concept.

**Java Thread pool** represents a group of worker threads that are waiting for the job and reuse many times.

In case of thread pool, a group of fixed size threads are created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, thread is contained in the thread pool again.

**Advantage: Better performance** It saves time because there is no need to create new thread.

**ExecutorService** is created using the **newFixedThreadPool**() factory method. This creates a thread pool with 10 threads executing tasks.

Second, an anonymous implementation of the **Runnable** interface is passed to the execute() method. This causes the **Runnable** to be executed by one of the threads in the **ExecutorService**.

How you create an ExecutorService depends on the implementation you use.

ExecutorService executorService1 = Executors.newSingleThreadExecutor();

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

ExecutorService executorService3 = Executors.newScheduledThreadPool(10);

| Sr. No. | Key | Runnable | Callable |
| --- | --- | --- | --- |
| 1 | Package | It belongs to Java.lang | It belongs to java.util.concurrent |
| 2 | Thread Creation | We can create thread by passing runnable as a parameter. | We can’t create thread by passing callable as parameter |
| 3 | Return Type | Ruunable does not return anything | Callable can return results |
| 4. | Method | It has run() method | It has call()method |
| 5 | Bulk Execution | It can’t be used for bulk execution of task | It can be used for bulk execution of task by invoking invokeAll(). |

Runnable cannot return a result to the caller.

In case you expect your threads to return a computed result you can use java.util.concurrent.Callable. The Callable object allows to return values after completion.

The Callable object uses generics to define the type of object which is returned.

If you submit a Callable object to an Executor the framework returns an object of typejava.util.concurrent.Future. Future exposes methods allowing a client to monitor the progress of a task being executed by a different thread. Therefore a Future object can be used to check the status of a Callable and to retrieve the result from the Callable.

On the Executor you can use the method submit to submit a Callable and to get a future. To retrieve the result of the future use the get() method.

There are a few different ways to delegate tasks for execution to an ExecutorService

* execute(Runnable)
* submit(Runnable)
* submit(Callable)
* invokeAny(...)
* invokeAll(...)

**submit(Runnable)**

The **submit(Runnable)**method also takes a **Runnable** implementation, but returns a **Future** object. This **Future** object can be used to check if the **Runnable** as finished executing.

Future future = executorService.submit(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

future.get(); //returns null if the task has finished correctly.

**execute(Runnable)**

The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously.

ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

executorService.shutdown();

**submit(Callable)**

The submit(Callable) method is similar to the submit(Runnable) method except for the type of parameter it takes. The Callable instance is very similar to a Runnable except that its call() method can return a result. The Runnable.run() method cannot return a result.

Future future = executorService.submit(new Callable(){

public Object call() throws Exception {

System.out.println("Asynchronous Callable");

return "Callable Result";

}

});

System.out.println("future.get() = " + future.get());

**invokeAny**

The invokeAny() method takes a collection of Callable objects, or sub interfaces of Callable. Invoking this method does not return a Future, but returns the result of one of the Callable objects. You have no guarantee about which of the Callable's results you get. Just one of the ones that finish.

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

String result = executorService.invokeAny(callables);

System.out.println("result = " + result);

executorService.shutdown();

**invokeAll()**

The invokeAll() method invokes all of the Callable objects you pass to it in the collection passed as parameter. The invokeAll() returns a list of Future objects via which you can obtain the results of the executions of each Callable.

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

List<Future<String>> futures = executorService.invokeAll(callables);

for(Future<String> future : futures){

System.out.println("future.get = " + future.get());

}

executorService.shutdown();

**What is the difference between yield() and join()**

**Yield:**

* Yield() method causes the current executing thread to pause execution and give chance for waiting thread or same priority.
* If there is no waiting thread or all the remaining waiting threads have low priority then the same thread will get chance once again for execution.

**Join:**

* If a Thread wants to wait until some other Thread completion then we should go for join() method
* Example if a Thread t1 execute **t2.join();** then t1 will entered into waiting state until t2 Thread completion

**Race condition:**

* A **race condition** is a situation in which two or more threads are reading/writing some shared data, and the final result depends on the timing of how the threads are scheduled. **Race conditions** can lead to **unpredictable** results and subtle program bugs. A thread can prevent this from happening by locking an object. When an object is locked by one thread and another thread tries to call a synchronized method on the same object, the second thread will block until the object is unlocked.
* Race condition occurs in a multi-threaded environment when more than one thread try to access a shared resource (modify, write) at the same time. Note that it is safe if multiple threads are trying to read a shared resource as long as they are not trying to change it. Since multiple threads try to race each other to finish executing a method thus the name race condition

**What is DeadLock? Is it possible to resolve DeadLock situation?**If two Threads are waiting for each other forever such type of situation is called DeadLock. For the DeadLock, there are no resolution techniques but prevention techniques are available.                              
Synchronized keyword is the thing to causes of DeadLock. If we are not using properly synchronized keyword the program will entered into DeadLock situation.

**Deadlock prevention methods**

* Don't use locks.
* If you must, keep your locks local. Global locks can be really tricky.
* Do as little as possible when you hold the lock.
* Use [stripes](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/util/concurrent/Striped.html) to only lock segments of your data
* Prefer Immutable types. Many times this means copying data instead of sharing data.
* don't hold several locks at once. If you do, always acquire the locks in the same order.

**Q30 What is the purpose of sleep() method?**Ans.  If a Thread don’t want to perform any operation for a particular amount of time then we should go for sleep() method. Whenever we are using sleep() method compulsory we should handle InterruptedException either by using try-catch or by using throws keyword otherwise we will get compile time error.  
  
**Q31. What is synchronized keyword? Explain its advantages and disadvantages.**Ans.       Synchronized keyword is applicable for method and blocks only. We can’t use for variables and classes.  
                If a method declared as a synchronized then at a time only one Thread is allow to execute that method on the given object.  
                The main advantages of synchronized keyword are, we can prevent data inconsistency problems and we can provide Thread safety.  
                But the main limitation of synchronized keyword is it increases waiting time of Threads and effect performance of the system. Hence if there is no specific requirement it is not recommended to use synchronized keyword.

### What is a CountDownLatch and how does it work?

**Answer:** A CountDownLatch is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes. It works by having a counter initialized to the number of operations. Threads waiting on the latch call await(), while threads performing the operations call countDown() when they finish. When the counter reaches zero, the waiting threads are released.

CountDownLatch latch = new CountDownLatch(3);

// Worker threads

new Thread(() -> {

// perform task

latch.countDown();

}).start();

new Thread(() -> {

// perform task

latch.countDown();

}).start();

// Main thread

latch.await(); // Wait until the latch count reaches zero

**Explain the difference between CyclicBarrier and CountDownLatch.**

* **CountDownLatch:**
  + One-time use.
  + Threads wait until the count reaches zero.
  + Can be used to wait for a set of operations to complete.
* **CyclicBarrier:**
  + Reusable after the barrier is broken.
  + A set of threads wait for each other to reach a common barrier point.
  + Useful for coordinating phases of computation among threads.

### What is ThreadLocal in Java?

ThreadLocal is a mechanism to create variables that can only be read and written by the same thread. Each thread accessing such a variable has its own, independently initialized copy of the variable. This is useful for maintaining thread confinement and avoiding synchronization for variables that are specific to a thread.

private static final ThreadLocal<Integer> threadLocalValue = ThreadLocal.withInitial(() -> 0);

public void someMethod() {

int value = threadLocalValue.get();

threadLocalValue.set(value + 1);

}

### Explain the concept of ForkJoinPool.

### It is an efficient parallel processing mechanism designed for divide-and-conquer tasks

### Used for splitting large tasks into smaller subtasks and executing them in parallel.

### Useful for processing large datasets (e.g., batch processing, aggregation tasks).

The ForkJoinPool serves to execute tasks in parallel, leveraging the fork/join framework. It is designed to enhance performance by dividing large tasks into smaller subtasks that can be processed concurrently, especially beneficial for multi-core processors.

ForkJoinPool is a specialized implementation of ExecutorService designed for work-stealing, suitable for parallelism. It is used for tasks that can be recursively split into smaller tasks, leveraging multiple cores for computation.

ForkJoinPool pool = new ForkJoinPool();

ForkJoinTask<Integer> task = pool.submit(new RecursiveTask<Integer>() {

@Override

protected Integer compute() {

// Recursive computation

}

});

Integer result = task.join(); // Get the result

**What are the differences between ExecutorService and ForkJoinPool?**

* **ExecutorService:**
  + General-purpose thread pool.
  + Suitable for tasks that do not need to be split.
* **ForkJoinPool:**
  + Designed for parallelism and recursive task splitting.
  + Uses work-stealing algorithm.
  + Suitable for divide-and-conquer algorithms.

### How does the Semaphore work in Java?

A Semaphore controls access to a shared resource through permits. It maintains a set of permits, and threads can acquire and release permits.

Semaphore semaphore = new Semaphore(3); // Allow up to 3 concurrent accesses

// Acquiring a permit

semaphore.acquire();

try {

// Access shared resource

} finally {

semaphore.release(); // Release the permit

}

**Deadlock**  
  
A deadlock occurs when two or more threads wait on each other, forming a cycle and preventing all of them from making any forward progress. Deadlocks are usually introduced by developers trying to solve the race conditions.

**Starvation**  
  
Starvation is an indefinite delay or permanent blocking of one or more runnable threads in a multithreaded application. Threads that are not being scheduled to run even though they are not blocking or waiting on anything else are said to be starving.  
  
**Livelock**  
  
Livelocks occur when threads are scheduled but are not making forward progress because they are continuously reacting to each other's state changes. High CPU utilization with no sign of real work being done is a classic warning sign of a livelock. Livelocks are incredibly difficult to detect and diagnose.

* Threads and Executors
* [Synchronization and Locks](http://winterbe.com/posts/2015/04/30/java8-concurrency-tutorial-synchronized-locks-examples/)
* [Atomic Variables and ConcurrentMap](http://winterbe.com/posts/2015/05/22/java8-concurrency-tutorial-atomic-concurrent-map-examples/)

**Synchronized**

When writing such multi-threaded code you have to pay particular attention when accessing shared mutable variables concurrently from multiple threads. Let's just say we want to increment an integer which is accessible simultaneously from multiple threads.

synchronized void incrementSync() {

count = count + 1;

}

When using incrementSync() concurrently we get the desired result count of 10000. No race conditions occur any longer and the result is stable with every execution of the code:

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 10000)

.forEach(i -> executor.submit(this::incrementSync));

stop(executor);

System.out.println(count); // 10000

**Locks**

Instead of using implicit locking via the synchronized keyword the Concurrency API supports various explicit locks specified by the Lock interface. Locks support various methods for finer grained lock control thus are more expressive than implicit monitors.

**Reentrant Locks**

The class ReentrantLock is a mutual exclusion lock with the same basic behavior as the implicit monitors accessed via the synchronized keyword but with extended capabilities. As the name suggests this lock implements reentrant characteristics just as implicit monitors.

**Reentrant means that locks are bound to the current thread.** A thread can safely acquire the same lock multiple times without running into deadlocks (e.g. a synchronized method calls another synchronized method on the same object).

ReentrantLock lock = new ReentrantLock();

int count = 0;

void increment() {

lock.lock();

try {

count++;

} finally {

lock.unlock();

}

}

A lock is acquired via lock() and released via unlock(). It's important to wrap your code into a try/finally block to ensure unlocking in case of exceptions. This method is thread-safe just like the synchronized counterpart. If another thread has already acquired the lock subsequent calls to lock() pause the current thread until the lock has been unlocked. Only one thread can hold the lock at any given time.

**ReadWriteLock**

The interface ReadWriteLock specifies another type of lock maintaining a pair of locks for read and write access. The idea behind read-write locks is that it's usually safe to read mutable variables concurrently as long as nobody is writing to this variable.

read tasks have to wait the whole second until the write task has finished. After the write lock has been released both read tasks are executed in parallel and print the result simultaneously to the console. They don't have to wait for each other to finish because read-locks can safely be acquired concurrently as long as no write-lock is held by another thread.

ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

executorService.shutdown();

ExecutorService executor = Executors.newFixedThreadPool(2);

Map<String, String> map = new HashMap<>();

ReadWriteLock lock = new ReentrantReadWriteLock();

executor.submit(() -> {

lock.writeLock().lock();

try {

sleep(1);

map.put("foo", "bar");

} finally {

lock.writeLock().unlock();

}

});

Runnable readTask = () -> {

lock.readLock().lock();

try {

System.out.println(map.get("foo"));

sleep(1);

} finally {

lock.readLock().unlock();

}

};

executor.submit(readTask);

executor.submit(readTask);

**StampedLock**

In contrast to ReadWriteLock the locking methods of a StampedLock return a stamp represented by a long value. You can use these stamps to either release a lock or to check if the lock is still valid. Additionally stamped locks support another lock mode called *optimistic locking*.

ExecutorService executor = Executors.newFixedThreadPool(2);

Map<String, String> map = new HashMap<>();

StampedLock lock = new StampedLock();

executor.submit(() -> {

long stamp = lock.writeLock();

try {

sleep(1);

map.put("foo", "bar");

} finally {

lock.unlockWrite(stamp);

}

});

Runnable readTask = () -> {

long stamp = lock.readLock();

try {

System.out.println(map.get("foo"));

sleep(1);

} finally {

lock.unlockRead(stamp);

}

};

executor.submit(readTask);

executor.submit(readTask);

stop(executor);

Obtaining a read or write lock via readLock() or writeLock() returns a stamp which is later used for unlocking within the finally block. Keep in mind that stamped locks don't implement reentrant characteristics. Each call to lock returns a new stamp and blocks if no lock is available even if the same thread already holds a lock. So you have to pay particular attention not to run into deadlocks.

An optimistic read lock is acquired by calling tryOptimisticRead() which always returns a stamp without blocking the current thread, no matter if the lock is actually available. If there's already a write lock active the returned stamp equals zero. You can always check if a stamp is valid by calling lock.validate(stamp).

**Semaphores**:

Whereas locks usually grant exclusive access to variables or resources, a semaphore is capable of maintaining whole sets of permits. This is useful in different scenarios **where you have to limit the amount concurrent access to certain parts of your application.**

It maintains set of permits and it uses acquire() and release() methods.

to release a permit based on the count they permits.

A semaphore can be atomically incremented and decremented to control access to a shared resource.

The beauty of the semaphore is that it hides all the complexity of managing access control, counting permits and, of course, getting the thread-safety right.

A semaphore is a data structure that maintains a set of permits that have to be acquired by competing threads. **Semaphores can therefore be used to control how many threads access a critical section or resource simultaneously**. Hence the constructor of java.util.concurrent.Semaphore takes as first parameter the number of permits the threads compete about. Each invocation of its acquire() methods tries to obtain one of the available permits. The method acquire() without any parameter blocks until the next permit gets available. Later on, when the thread has finished its work on the critical resource, it can release the permit by invoking the method release() on an instance of Semaphore.

**ExecutorService executor = Executors.newFixedThreadPool(10);**

**Semaphore semaphore = new Semaphore(5);**

**Runnable longRunningTask = () -> {**

**boolean permit = false;**

**try {**

**permit = semaphore.tryAcquire(1, TimeUnit.SECONDS);**

**if (permit) {**

**System.out.println("Semaphore acquired");**

**sleep(5);**

**} else {**

**System.out.println("Could not acquire semaphore");**

**}**

**} catch (InterruptedException e) {**

**throw new IllegalStateException(e);**

**} finally {**

**if (permit) {**

**semaphore.release();**

**}**

**}**

**}**

**IntStream.range(0, 10)**

**.forEach(i -> executor.submit(longRunningTask));**

**stop(executor);**

The executor can potentially run 10 tasks concurrently but we use a semaphore of size 5, thus limiting concurrent access to 5. It's important to use a try/finally block to properly release the semaphore even in case of exceptions.

**Livestock**

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, Alphonse moves to his right, while Gaston moves to his left. They're still blocking each other, and so on...

The main difference between **livelock** and **deadlock** is that threads are not going to be blocked, instead they will try to respond to each other continuously.

**Wait() Vs Sleep()**

**sleep()** is a method which is used to hold the process for few seconds or the time you wanted but in case of **wait()** method thread goes in waiting state and it won’t come back automatically until we call the notify() or notifyAll().

The ***major difference*** is that wait() releases the lock or monitor while sleep() doesn’t releases any lock or monitor while waiting. Wait is used for inter-thread communication while sleep is used to introduce pause on execution, generally.

**AtomicInteger**

The package java.concurrent.atomic contains many useful classes to perform atomic operations. An operation is atomic when you can safely perform the operation in parallel on multiple threads without using the synchronized keyword or locks.

Internally, the atomic classes make heavy use of [compare-and-swap](http://en.wikipedia.org/wiki/Compare-and-swap) (CAS).

These usually are much faster than synchronizing via locks. So my advice is to prefer atomic classes over locks in case you just have to change a single mutable variable concurrently.

**AtomicInteger atomicInt = new AtomicInteger(0);**

**ExecutorService executor = Executors.newFixedThreadPool(2);**

**IntStream.range(0, 1000)**

**.forEach(i -> executor.submit(atomicInt::incrementAndGet));**

**stop(executor);**

**System.out.println(atomicInt.get());**

By using AtomicInteger as a replacement for Integer we're able to increment the number concurrently in a thread-safe manor without synchronizing the access to the variable. The method incrementAndGet() is an atomic operation so we can safely call this method from multiple threads.

The method accumulateAndGet() use this method to sum up all values from 0 to 1000 concurrently.

AtomicInteger supports various kinds of atomic operations. The method updateAndGet()accepts a lambda expression in order to perform arbitrary arithmetic operations upon the integer.

**CountDownLatch:**

It is used when we want to wait for more than one thread to complete its task. It is similar to join in threads.

Where we can use CountDownLatch

Consider a scenario where we have requirement where we have three threads "A", "B" and "C" and we want to start thread "C" only when "A" and "B" threads completes or partially completes their task.

Here **await()** method waits for countdownlatch flag to become 0, and **countDown()** method decrements countdownlatch flag by 1.

In the BLOCKED state, a thread is about to enter a synchronized block, but there is another thread currently running inside a synchronized block on the same object. The first thread must then wait for the second thread to exit its block.

In the WAITING state, a thread is waiting for a signal from another thread. This happens typically by calling Object.wait(), or Thread.join(). The thread will then remain in this state until another thread calls Object.notify(), or dies.