Multithreading:

Multithreading is a process of executing two or more parts of a program simultaneously to maximum utilization of the CPU. Each part of such a program is called a **thread**.

Synchronization:

In Java, **synchronization** is the mechanism that controls the access of multiple threads to shared resources. Without it, you might encounter "**Race Conditions**," where two threads modify the same data simultaneously, leading to unpredictable results.

Think of it like a **single-person restroom**: the lock on the door ensures that only one person can enter at a time. In Java, this "lock" is known as a **Monitor**.

Race Condition:

Race Condition ek aisi situation hai jaha do ya do se zyada threads ek hi shared data par kaam karne ki koshish karte hain, aur final result is baat par depend karta hai ki kaunsa thread pehle execute hua.

Isse program mein unpredictable bugs aate hain kyunki result hamesha "race" (kaun jeeta) par nirbhar karta hai.

Ek Real-Life Example

Maano ek Bank Account mein ₹100 hain. Do log (Threads) ek saath ₹50 nikalne ki koshish karte hain:

1. **Thread A** check karta hai: Balance ₹100 hai? Haan.
2. **Thread B** bhi check karta hai: Balance ₹100 hai? Haan.
3. **Thread A** ₹50 nikal leta hai aur balance ₹50 update kar deta hai.
4. **Thread B** (jisne pehle hi ₹100 dekha tha) ₹50 nikalta hai aur balance ko phir se ₹50 update kar deta hai.

Problem: Total ₹100 nikal gaye, lekin balance abhi bhi ₹50 dikha raha hai! Yeh data inconsistency hi Race Condition hai.

Java mein Synchronization kyun zaroori hai?

Synchronization ka maqsad race conditions ko rokna hai. Java mein hum `synchronized` keyword ka use karte hain taaki ek waqt mein sirf **ek hi thread** critical section (shared data) ko access kar sake.

Method Level Synchronization:

A. Synchronized Methods

By adding the `synchronized` keyword to a method header, you lock the **entire method** for the current object (`this`).

Java

```java
public synchronized void deposit(int amount) {
    balance += amount;
}
```

Block Level Synchronization:

B. Synchronized Blocks

If you only need to protect a few lines of code rather than an entire method, synchronized blocks are more efficient. They allow you to specify exactly which object to lock on.

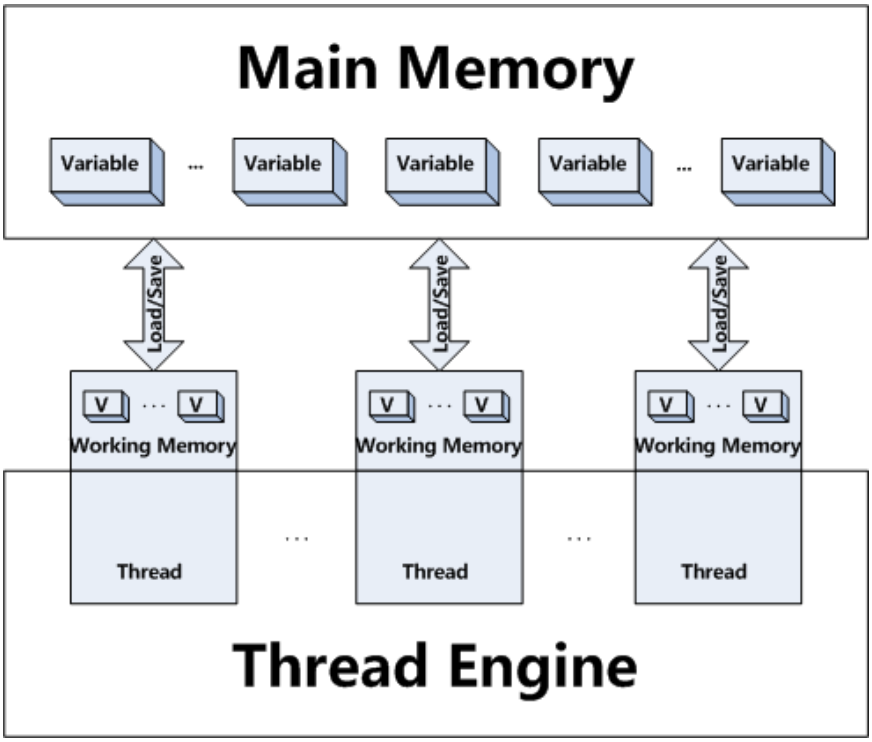
Java

```java
public void updateData() {
    // Non-critical code here (accessible by all threads)

    synchronized(this) {
        // Critical section (only one thread at a time)
        this.count++;
    }
}
```

Volatile keyword:

- Synchronization in Java is possible by using Java keywords **synchronized** and **volatile**.
- In Java, we can not have **synchronized** variable. Using synchronized keyword with a variable is illegal and will result in compilation error. Instead of using the synchronized variable in Java, you can use the java **volatile** variable, which will instruct JVM threads to read the value of volatile variable from main memory and don't cache it locally.
- If a variable is not shared between multiple threads then there is no need to use volatile keyword.
- Using volatile variables reduces the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable.
- Volatile Variables are light-weight synchronization. When visibility of latest data among all threads is requirement and atomicity can be compromised , in such situations Volatile Variables must be preferred. Read on volatile variables always return most recent write done by any thread since they are neither cached in registers nor in caches where other processors can not see. Volatile is Lock-Free.

Note: In volatile first priority is to always write after read operation.



Differences Between Synchronized and Volatile:

1. The volatile keyword in Java is a field modifier while synchronized modifies code blocks and methods.

2. Synchronized obtains and releases the lock on monitor's Java volatile keyword doesn't require that.

3. Threads in Java can be blocked for waiting for any monitor in case of synchronized, that is not the case with the volatile keyword in Java.

4. Synchronized method affects performance more than a volatile keyword in Java.

5. Since volatile keyword in Java only synchronizes the value of one variable between Thread memory and "`main`" memory while synchronized synchronizes the value of all variable between thread memory and "`main`" memory and locks and releases a monitor to boot. Due to this reason synchronized keyword in Java is likely to have more overhead than volatile.

6. You can not synchronize on the `null` object but your volatile variable in Java could be null.

7. From Java 5 writing into a volatile field has the same memory effect as a monitor release, and reading from a volatile field has the same memory effect as a monitor acquire

Example Singleton Class in case of Multithreading:

```
public class Singleton {
    private volatile static Singleton singleton;
    private Singleton() {
    }
    public static Singleton getInstance() {
        if (singleton == null) {
            // To make thread safe
            synchronized (Singleton.class) {
                // check again as multiple threads can reach above step
                if (singleton == null)
                    singleton = new Singleton();
            }
        }
        return singleton;
    }
}
```

```
public class Test {

    public static void main(String[] args) {
        ExecutorService executorService = null;
        try {
            executorService = Executors.newFixedThreadPool(5);

            Runnable t1 = new Runnable() {

                @Override
                public void run() {
                    for (int i = 1; i <=100; i++) {
                        Singleton singleton = Singleton.getInstance();
                        System.out.println(singleton.hashCode());
                    }
                }
            };


            Runnable t2 = new Runnable() {

                @Override
                public void run() {
                    for (int i = 1; i <=100; i++) {
                        Singleton singleton = Singleton.getInstance();
                        System.out.println(singleton.hashCode());
                    }
                }
            };

            executorService.submit(t1);
            executorService.submit(t2);
        } catch (Exception e) {
            e.printStackTrace();
        }finally{
            if(executorService != null)
                executorService.shutdown();
        }
    }

}
```

Problems @ Javadoc Declaration Console

<terminated> Test (1) [Java Application] C:\Program Files\Java\jre1.8.0_65\bin\javaw.exe (Feb 2, 2018 10:05:36 AM)

```
1951881630
1951881630
1951881630
1951881630
1951881630
1951881630
1951881630
1951881630
1951881630
1951881630
1951881630
1951881630
1951881630
```

The hashcode will be the same because instance create only once.

Note: Volatile gives a visibility but not the atomicity.

Atomic Variable:

Atomic variables are classes provided in the java.util.concurrent.atomic package that allow you to perform thread-safe operations on single variables without using locks (like synchronized or ReentrantLock).

Example-#1

Problem:

Problem: Normal Variable ke saath (Race Condition)

Sochiye hamare paas ek variable hai `int count = 0;`. Do threads (`Thread 1` aur `Thread 2`) isko increment (`count++`) karna chahti hain.

Asal mein `count++` ek step nahi, balki **3 steps** hota hai:

1. **Read:** Memory se value uthao (maan lo `0`).
2. **Modify:** Usme `+1` karo (ab value `1` ho gayi).
3. **Write:** Wapas memory mein save karo.

Solution:

Solution: AtomicInteger ke saath (CAS Mechanism)

Ab wahi kaam `AtomicInteger` se karte hain. Ye **Compare-And-Swap (CAS)** ka use karta hai. Iska logic hota hai: *"Value tabhi update karo agar purani value wahi hai jo maine read ki thi."*

Example with Atomic:

1. **Thread 1** ne read kiya: `0` .
2. **Thread 2** ne read kiya: `0` .
3. **Thread 1** ne update karne ki koshish ki. Usne pucha: *"Kya memory mein abhi bhi 0 hai?"*
   - System ne kaha: *"Haan"*.
   - **Thread 1** ne use `1` kar diya. (Memory = 1).
4. **Thread 2** ne update karne ki koshish ki. Usne pucha: *"Kya memory mein abhi bhi 0 hai?"* (Kyuki usne 0 read kiya tha).
   - System ne kaha: **"Nahi, ab wahan 1 ho chuka hai!"**
5. **Thread 2** haar nahi maanta. Wo phir se nayi value (`1`) read karta hai aur dobara koshish karta hai. Is baar wo successfully use `2` kar deta hai.

AtomicInteger

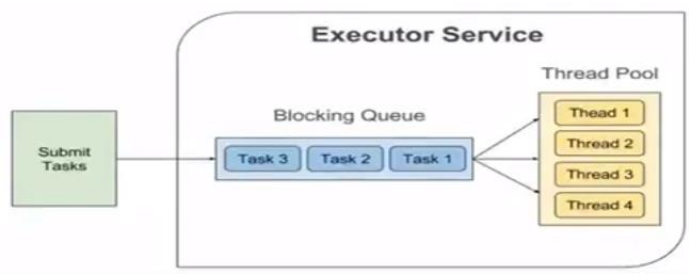
```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicExample {
    // Atomic variable
    AtomicInteger count = new AtomicInteger(0);

    public void doWork() {
        // Thread 1
        Thread t1 = new Thread(() -> {
            for(int i=0; i<1000; i++) count.incrementAndGet();
        });

        // Thread 2
        Thread t2 = new Thread(() -> {
            for(int i=0; i<1000; i++) count.incrementAndGet();
        });

        t1.start();
        t2.start();

        // Wait for both to finish...
        // Result will ALWAYS be 2000
    }
}
```

Thread Pool:

Most of the executor implementations use thread pools to execute tasks, A thread pool is nothing but a bunch of worker threads that exist separately from the runnable or Callable tasks and is managed by the executor.

Creating a thread is an expensive operation and it should be minimized. Having worker threads minimizes the overhead due to thread creation because executor service has to create the thread pool only once and then it can reuse the threads for executing any tasks.

Tasks are submitted to a thread pool via an internal queue called the Blocking queue. If there are more tasks then for waiting until any thread becomes available. If the blocking queue is full than new tasks are rejected



java.util.concurrent:

The java.util.concurrent (J.U.C) package is organized into several distinct sub-hierarchies. Rather than one single tree, it is a collection of frameworks designed to handle specific threading problems like task scheduling, synchronization, and thread-safe data storage.

It was introduced in JDK 1.5 release as an alternative of synchronized keyword. As I told you Lock is an interface, so we cannot use it directly, instead we need to use its implementation class

Future and Callable Concurrency:

java.util.concurrent.Callable object can return the computed result done by a thread in contrast to runnable interface which can only run the thread. The Callable object returns Future object which provides methods to monitor the progress of a task being executed by a thread. Future object can be used to check the status of a Callable and then retrieve the result from the Callable once the thread is done. It also provides timeout functionality.

Callable vs Runnable interface in Java

As I explained major differences between a Callable and Runnable interface in the last section. Sometimes this question is also asked as the difference between call() and run() method in Java. All the points discussed here is equally related to that question as well. Let's see them in point format for better understanding :

1) The Runnable interface is older than Callable, there from JDK 1.0, while Callable is added on Java 5.0.

2) Runnable interface has run() method to define task while Callable interface uses call() method for task definition.

3) run() method does not return any value, it's return type is void while call method returns value. The Callable interface is a generic parameterized interface and Type of value is provided when an instance of Callable implementation is created.

4) Another difference on run and call method is that run method can not throw checked exception while call method can throw checked exception in Java.

Here is a nice summary of all the **differences between Callable and Runnable in Java**:

```
public Object call() throws Exception;
```

Both callable and runnable interface are the Functional Interface(annotation of the java 7(functionalinteface)).

Runnable interfaces belong to java.lang package.

1. In this runnable run method don't throws any exception.
2. Does not return any value.

```
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```

Callable interface belong to java.util.concurrent. package.

1. In this callable call() method throws Exeption.
2. It return future value means return object.
3. Return type "v" represent Future Interace.

```
public interface Future<V> {

    /**
     * Attempts to cancel execution of this t
```

```
@FunctionalInterface
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

Example:

Output: Future Result : My Callable Interface Task

1. The Executor Framework (Task Execution):

Managing thread pools and asynchronous tasks. It separates task submission from task execution.

Executors, A framework for creating and managing threads. Executors framework helps you with –

- **Thread Creation**: It provides various methods for creating threads, more specifically a pool of threads, that your application can use to run tasks concurrently.
- **Thread Management**: It manages the life cycle of the threads in the thread pool. You don't need to worry about whether the threads in the thread pool are active or busy or dead before submitting a task for execution.
- **Task submission and execution**: Executors framework provides methods for submitting tasks for execution in the thread pool, and also gives you the power to decide when the tasks will be executed. For example, You can submit a task to be executed now or schedule them to be executed later or make them execute periodically.

Java Concurrency API defines the following three executor interfaces that covers everything that is needed for creating and managing threads –

- **Executor** – A simple interface that contains a method called execute(Runnable command) to launch a task specified by a Runnable object.
- **ExecutorService** – A sub-interface of Executor that adds functionality to manage the lifecycle of the tasks. It also provides a submit() method whose overloaded versions can accept a Runnable as well as a Callable object. Callable objects are similar to Runnable except that the task specified by a Callable object can also return a value.
- **ScheduledExecutorService** – A sub-interface of ExecutorService. It adds functionality to schedule the execution of the tasks.

ExecutorService provides two methods for shutting down an executor –

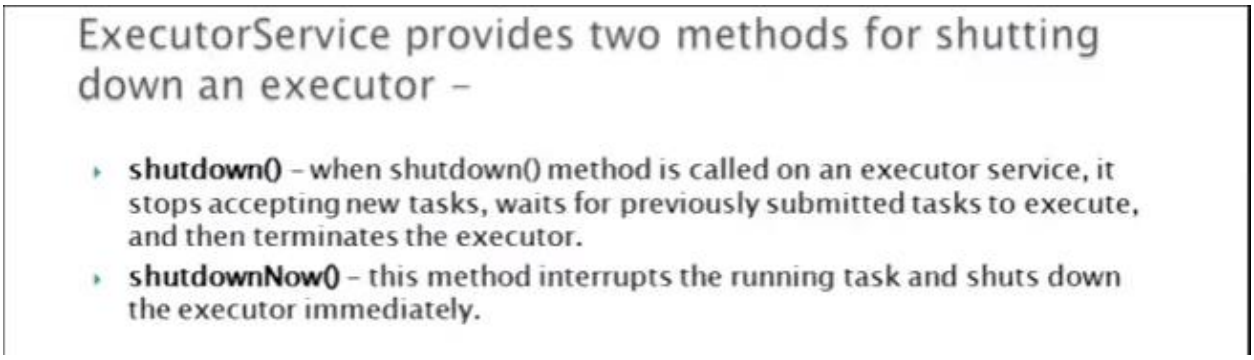
- **shutdown()** – when shutdown() method is called on an executor service, it stops accepting new tasks, waits for previously submitted tasks to execute, and then terminates the executor.
- **shutdownNow()** – this method interrupts the running task and shuts down the executor immediately.

2.Executor Interface:

```
public interface Executor {

    /**
     * Executes the given command at some time in the future.  The command
     * may execute in a new thread, in a pooled thread, or in the calling
     * thread, at the discretion of the {@code Executor} implementation.
     *
     * @param command the runnable task
     * @throws RejectedExecutionException if this task cannot be
     * accepted for execution
     * @throws NullPointerException if command is null
     */
    void execute(Runnable command);
}
```

3.ExecutorService Interface:

```
public abstract interface ExecutorService
  extends Executor
{
  public abstract void shutdown();

  public abstract List<Runnable> shutdownNow();

  public abstract boolean isShutdown();

  public abstract boolean isTerminated();

  public abstract boolean awaitTermination(long paramLong, TimeUnit paramTimeUnit)
    throws InterruptedException;

  public abstract <T> Future<T> submit(Callable<T> paramCallable);

  public abstract <T> Future<T> submit(Runnable paramRunnable, T paramT);

  public abstract Future<?> submit(Runnable paramRunnable);

  public abstract <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> paramCollection)
    throws InterruptedException;

  public abstract <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> paramCollection, long paramLong, TimeUnit paramTimeUnit)
    throws InterruptedException;

  public abstract <T> T invokeAny(Collection<? extends Callable<T>> paramCollection)
    throws InterruptedException, ExecutionException;

  public abstract <T> T invokeAny(Collection<? extends Callable<T>> paramCollection, long paramLong, TimeUnit paramTimeUnit)
    throws InterruptedException, ExecutionException, TimeoutException;
}
```



Here are examples to creates ExecutorService and ScheduledExecutorService instances:

```
// Creates a single thread ExecutorService
ExecutorService singleExecutorService = Executors.newSingleThreadExecutor();

// Creates a single thread ScheduledExecutorService
ScheduledExecutorService singleScheduledExecutorService = Executors.newSingleThreadScheduledExecutor();

// Creates an ExecutorService that use a pool of 10 threads
ExecutorService fixedExecutorService = Executors.newFixedThreadPool(10);

// Creates an ExecutorService that use a pool that creates threads on demand
// And that kill them after 60 seconds if they are not used
ExecutorService onDemandExecutorService = Executors.newCachedThreadPool();

// Creates a ScheduledExecutorService that use a pool of 5 threads
ScheduledExecutorService fixedScheduledExecutorService = Executors.newScheduledThreadPool(5);
```

4.ScheduledExecutorService Interface:

```
public abstract interface ScheduledExecutorService
  extends ExecutorService
{
  public abstract ScheduledFuture<?> schedule(Runnable paramRunnable, long paramLong, TimeUnit paramTimeUnit);

  public abstract <V> ScheduledFuture<V> schedule(Callable<V> paramCallable, long paramLong, TimeUnit paramTimeUnit);

  public abstract ScheduledFuture<?> scheduleAtFixedRate(Runnable paramRunnable, long paramLong1, long paramLong2, TimeUnit paramTimeUnit);

  public abstract ScheduledFuture<?> scheduleWithFixedDelay(Runnable paramRunnable, long paramLong1, long paramLong2, TimeUnit paramTimeUnit);
}
```

Executors Utility Class:

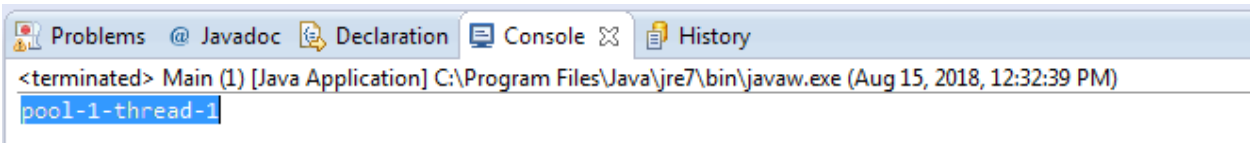
Apart from the above three interfaces, the API also provides an Executors class that contains factory methods for creating different kinds of executor services.

Example #1.

Store all threads into the thread pool.

Here we are using single thread only. So it will create single thread only into the thread pool.

```java
package com.executorsevice.framwork;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Main {
public static void main(String[] args) {
    ExecutorService executorService= Executors.newSingleThreadExecutor();
     Runnable task1=new Runnable() {

        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName());
        }
    };
    executorService.submit(task1);
    executorService.shutdown();
}
}
```



Example #2:

Java ExecutorService with a pool of threads for executing multiple tasks:

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class Main {
public static void main(String[] args) {
    ExecutorService executorService= Executors.newFixedThreadPool(3);
     Runnable task1=new Runnable() {

        @Override
        public void run() {
            System.out.println("First Task Started : "+Thread.currentThread().getName());
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("First Task Ended : "+Thread.currentThread().getName());
        }
    };
    Runnable task2=new Runnable() {

        @Override
        public void run() {
            System.out.println("Second Task Started : "+Thread.currentThread().getName());
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Second Task Ended : "+Thread.currentThread().getName());
        }
    };
```

```
    Runnable task3=new Runnable() {

        @Override
        public void run() {
            System.out.println("Third Task Started : "+Thread.currentThread().getName());
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Third Task Ended : "+Thread.currentThread().getName());
        }
    };
    executorService.submit(task1);
    executorService.submit(task2);
    executorService.submit(task3);
    executorService.shutdown();
}
}
```

Problems @ Javadoc Declaration Console History

<terminated> Main (1) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Aug 15, 2018, 12:42:54 PM)

```
First Task Started : pool-1-thread-1
Second Task Started : pool-1-thread-2
Third Task Started : pool-1-thread-3
First Task Ended : pool-1-thread-1
Third Task Ended : pool-1-thread-3
Second Task Ended : pool-1-thread-2
```

Java Concurrency - Lock Interface:

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. 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.

1. Key Methods of the `Lock` Interface

Unlike `synchronized`, which is a language-level keyword, `Lock` is an interface that requires explicit calls to acquire and release locks.

Method	Description
`void lock()`	Acquires the lock. Blocks the thread indefinitely until the lock is available.
`boolean tryLock()`	**Non-blocking.** Attempts to acquire the lock and returns `true` immediately if successful, `false` otherwise.
`boolean tryLock(long time, TimeUnit unit)`	Attempts to acquire the lock within a specific timeframe before giving up.
`void lockInterruptibly()`	Acquires the lock unless the thread is interrupted, allowing it to respond to `Thread.interrupt()`.
`void unlock()`	Releases the lock. **Must** be called in a `finally` block to avoid deadlocks.
`Condition newCondition()`	Returns a `Condition` instance for advanced inter-thread communication (replaces `wait/notify`).

ReentrantLock

ReentrantLock is the most common implementation. "Reentrant" means a thread can re-acquire the same lock it already holds without deadlocking itself (it simply increments a "hold count").

A **ReentrantLock** is a concrete implementation of the Lock interface in the java.util.concurrent.locks package. It is called "reentrant" because it allows a thread to acquire the same lock multiple times without deadlocking itself.

```
Lock lock = new ReentrantLock();

public void performTask() {
    lock.lock(); // Explicitly acquire
    try {
        // Critical Section (shared resource access)
    } finally {
        lock.unlock(); // Always release in finally to ensure safety
    }
}
```

1. How Reentrancy Works

Every time a thread acquires the lock, a **hold count** is incremented by 1. Every time the thread releases the lock, the count is decremented. The lock is only truly released for other threads when the hold count reaches zero.

Example Scenario: If Method A (synchronized on a lock) calls Method B (also synchronized on the same lock), a reentrant lock allows the thread to enter Method B because it already owns the lock.

```
import java.util.concurrent.locks.ReentrantLock;

public class Counter {
    private final ReentrantLock lock = new ReentrantLock();
    private int count = 0;

    public void increment() {
        lock.lock(); // Block until acquired
        try {
            count++;
        } finally {
            lock.unlock(); // Ensure release even if exception occurs
        }
    }

    public int getCount() {
        return count;
    }
}
```

Blocking Queues (Data Transfer):

Before BlockingQueue, you had to manually use synchronized, wait(), and notifyAll() to manage buffers, which is extremely error-prone and leads to bugs like **Lost Wakeups** or **Spurious Wakeups**. BlockingQueue encapsulates all that complexity into a single, thread-safe class.

Blocking Queues are the primary tool for solving the **Producer-Consumer** problem. They act as a thread-safe "buffer" that handles the coordination of data transfer between threads automatically.

What makes them "Blocking" is their behavior when the queue is full or empty:

- **When Empty:** A thread trying to **take**() an element will block (wait) until a producer adds something.
- **When Full:** A thread trying to **put**() an element will block (wait) until a consumer removes something (for bounded queues).

2. Blocking Queues (Data Transfer)

These queues are used for the **Producer-Consumer pattern**. They automatically "block" a thread if it tries to take from an empty queue or put into a full one.

- `BlockingQueue` **(Interface):** The core interface for thread-safe queues.
  - `ArrayBlockingQueue` : Bounded (fixed size), backed by an array.
  - `LinkedBlockingQueue` : Optionally bounded, backed by linked nodes (higher throughput).
  - `PriorityBlockingQueue` : Unbounded, elements are ordered by priority.
  - `SynchronousQueue` : A queue where each insert must wait for a corresponding remove (direct hand-off).

Synchronizers:

Synchronizers are high-level coordination utilities that facilitate common interaction patterns between threads. They allow threads to wait for each other, communicate state changes, or manage access to shared resources without the complexity of low-level wait() and notify() calls.

Here is the breakdown of the primary synchronizers found in java.util.concurrent.

1.CountDownLatch:

Real-life example aisa samajhiye: **Ek race tabhi start hogi jab saare (N) runners starting line par pahunch jayenge.**

CountDownLatch in Java is a kind of synchronizer which allows one Thread to wait for one or more Threads before starts processing. This is very crucial requirement and often needed in server side core Java application and having this functionality built-in as CountDownLatch greatly simplifies the development. CountDownLatch in Java is introduced on Java 5.

When should we use CountDownLatch in Java :

Use CountDownLatch when one of Thread like main thread, require to wait for one or more thread to complete, before its start doing processing. Classical example of using CountDownLatch in Java is any server side core Java application which uses services architecture, where multiple services is provided by multiple threads and application cannot start processing until all services have started successfully as shown in our CountDownLatch example.**countDownLatch in Java – Things to remember:**

Few points about Java CountDownLatch which is worth remembering:

1. You can not reuse CountDownLatch once count is reaches to zero, this is the main difference between CountDownLatch and CyclicBarrier, which is frequently asked in core Java interviews and multi-threading interviews.

2. Main Thread wait on Latch by calling CountDownLatch.await() method while other thread calls CountDownLatch.countDown() to inform that they have completed.

Example: IT and BA task.

```java
package com.infotech.client;

import java.util.concurrent.CountDownLatch;

public class AssignTaskManagerTest {
    public static void main(String[] args) throws InterruptedException {
        //Created CountDownLatch for 2 threads
        CountDownLatch countDownLatch = new CountDownLatch(2);

        //Created and started two threads
        DevTeam teamDevA = new DevTeam(countDownLatch, "dev-A");
        DevTeam teamDevB = new DevTeam(countDownLatch, "dev-B");

        teamDevA.start();
        teamDevB.start();

        //When two threads(dev-A and dev-B)completed tasks are returned
        countDownLatch.await();

        //Now execution of thread(QA team) started..
        QATeam qaTeam = new QATeam("QA team");
        qaTeam.start();
    }
}
```

```java
package com.infotech.tasks;

public class QATeam extends Thread {

    public QATeam(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println("Task assigned to "+Thread.currentThread().getName());
        try {
            Thread.sleep(2000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        System.out.println("Task finished by "+Thread.currentThread().getName());
    }
}
```

2.CyclicBarrier:

CyclicBarrier in Java is a synchronizer introduced in JDK 5 on java.util.Concurrent package along with other concurrent utility like Counting Semaphore, BlockingQueue, ConcurrentHashMap etc. CyclicBarrier is similar to CountDownLatch and allows multiple threads to wait for each other (barrier) before proceeding. The difference between CountDownLatch . CyclicBarrier is a natural requirement for a concurrent program because it can be used to perform final part of the task once individual tasks are completed. All threads which wait for each other to reach barrier are called parties, CyclicBarrier is initialized with a number of parties to wait and threads wait for each other by calling CyclicBarrier.await() method which is a blocking method in Java and blocks until all Thread or parties call await(). In general calling await() is shout out that Thread is waiting on the barrier. await() is a blocking call but can be timed out or Interrupted by other thread.

Difference between CountDownLatch and CyclicBarrier in Java:

If you look at CyclicBarrier it also the does the same thing but there is different you can not reuse CountDownLatch once the count reaches zero while you can reuse CyclicBarrier by calling reset() method which resets Barrier to its initial State. What it implies that CountDownLatch is a good for one-time events like application start-up time and CyclicBarrier can be used to in case of the recurrent event e.g. concurrently calculating a solution of the big problem etc.

Important point of CyclicBarrier in Java:

1. CyclicBarrier can perform a completion task once all thread reaches to the barrier, This can be provided while creating CyclicBarrier.

2. If CyclicBarrier is initialized with 3 parties means 3 thread needs to call await method to break the barrier.

3. The thread will block on await() until all parties reach to the barrier, another thread interrupt or await timed out.

4. If another thread interrupts the thread which is waiting on barrier it will throw BrokernBarrierException as shown below:

java.util.concurrent.BrokenBarrierException

at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:172)

at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:327)

5.CyclicBarrier.reset() put Barrier on its initial state, other thread which is waiting or not yet reached barrier will terminate with java.util.concurrent.BrokenBarrierException.

Example:

```
public class PassengerThread extends Thread {
    private int duration;
    private CyclicBarrier barrier;

    public PassengerThread(int duration, CyclicBarrier barrier, String pname) {
        super(pname);
        this.duration = duration;
        this.barrier = barrier;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(duration);
            System.out.println(Thread.currentThread().getName() + " is arrived.");

            int await = barrier.await();
            if(await == 0) {
                System.out.println("Four passengers have arrived so cab is going to start.
            }

        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class ClientTest {
    public static void main(String args[]) throws InterruptedException, BrokenBarrierExcep

        System.out.println(Thread.currentThread().getName() + " has started");

        CyclicBarrier barrier = new CyclicBarrier(4);

        PassengerThread p1 = new PassengerThread(1000, barrier, "John");
        PassengerThread p2 = new PassengerThread(2000, barrier, "Martin");
        PassengerThread p3 = new PassengerThread(3000, barrier, "Joya");
        PassengerThread p4 = new PassengerThread(4000, barrier, "Sam");

        PassengerThread p5 = new PassengerThread(1000, barrier, "Pipa");
        PassengerThread p6 = new PassengerThread(2000, barrier, "Dolly");
        PassengerThread p7 = new PassengerThread(3000, barrier, "Harman");
        PassengerThread p8 = new PassengerThread(4000, barrier, "Brad");

        p1.start();
        p2.start();
```

```
31          p3.start();
32          p4.start();
33
34          p5.start();
35          p6.start();
36          p7.start();
37          p8.start();
```

```
    }   }
```

```
<terminated> ClientTest (1) [Java Application] C:\Program Files\Java\jre1.8.0_65\bin\javaw.exe (Feb 7, 2018, 7:04:38 AM)
main has started
main has finished
John is arrived.
Pipa is arrived.
Martin is arrived.
Dolly is arrived.
Four passengers have arrived so cab is going to start..
Joya is arrived.
Harman is arrived.
Brad is arrived.
Sam is arrived.
Four passengers have arrived so cab is going to start..
```

3. Semaphore:

Semaphore multithreading mein ek aisi utility hai jo **"Permits" (Ijazat/Tokens)** ke concept par kaam karti hai. Iska asali maqsad ye control karna hota hai ki ek waqt mein kitne threads kisi specific resource (jaise Database connection, File, ya Network socket) ko access kar sakte hain.

Ise aap ek **Parking Lot** ki tarah samajh sakte hain:

- Agar parking mein 10 jagah (permits) hain, toh 10 gaadiyan (threads) andar ja sakti hain.
- 11वीं gaadi ko tab tak bahar wait karna padega jab tak koi ek gaadi nikal na jaye.

1. Semaphore kaise kaam karta hai?

Semaphore do main methods use karta hai:

1. `acquire()` : Ye method ek permit mangta hai. Agar permit available hai, toh thread aage badh jata hai aur count 1 kam ho jata hai. Agar permit nahi hai, toh thread wahi ruk (block) jata hai.
2. `release()` : Jab thread ka kaam khatam ho jata hai, wo permit wapas kar deta hai, jisse count 1 badh jata hai aur waiting threads mein se kisi ek ko mauka milta hai.

- **Semaphore** is the new class introduced as a part of concurrency package in **JDK 1.5** .It maintains a set of permits which gets acquired by thread before using some functionality.
- It is a technique to protect your critical resource from being used by more than 'N' threads simultaneously. Semaphore maintains number of available permits. Whenever a thread wants to use some shared resource, maintained by semaphore. It asks semaphore for the permit. If permit is available thread can use the shared resource, otherwise it will wait till some other thread releases the permit or come out without using the shared resource.

Example:

```
import java.util.concurrent.Semaphore;

public class Connection {
public int number_of_connetion=0;
private static Connection INSTANCE = new Connection();
private Semaphore semaphore=new Semaphore(5, true);
public static Connection getConnection() {
    return INSTANCE;
}

public void connection(){
    try {
        semaphore.acquire();
        synchronized (this) {
            number_of_connetion++;
            System.out.println("number of conntions :"+number_of_connetion);
        }
        Thread.sleep(2000);
        synchronized (this) {
            number_of_connetion--;
            System.out.println("number of conntions :"+number_of_connetion);
        }

    } catch (Exception e) {

    }finally{
        if(semaphore!=null)
        semaphore.release();
    }
}
}
```

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Semaphore {
    static ExecutorService executorService=null;

    public static void main(String[] args) {
        try {
        executorService=Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++) {
        executorService.submit(new Runnable() {

            @Override
            public void run() {
                Connection.getConnection().connection();
            }
        });
        }
        } catch (Exception e) {
            // TODO: handle exception
        }finally{
            if(executorService!=null)
            executorService.shutdown();
        }
    }
}
```

Before Semaphore implement it will create number of connections that we are using in loop(50)

```
<terminated> Semaphore [Java Application] C:\Program Files\Java\jre7\bin\javaw.
number of conntions :1
number of conntions :2
number of conntions :3
number of conntions :4
number of conntions :5
number of conntions :6
number of conntions :7
number of conntions :8
number of conntions :9
number of conntions :10
number of conntions :9
number of conntions :8
number of conntions :7
number of conntions :6
number of conntions :5
number of conntions :4
number of conntions :3
number of conntions :2
number of conntions :1
number of conntions :0
```

After Semaphore Implement: it will create only connection pool.

```
<terminated> Semaphore [Java Application] C:\Program Files\Ja
number of conntions :1
number of conntions :2
number of conntions :3
number of conntions :4
number of conntions :5
number of conntions :4
number of conntions :3
number of conntions :4
number of conntions :3
number of conntions :4
number of conntions :3
number of conntions :2
number of conntions :3
number of conntions :4
number of conntions :5
number of conntions :4
number of conntions :3
number of conntions :2
number of conntions :1
number of conntions :0
```