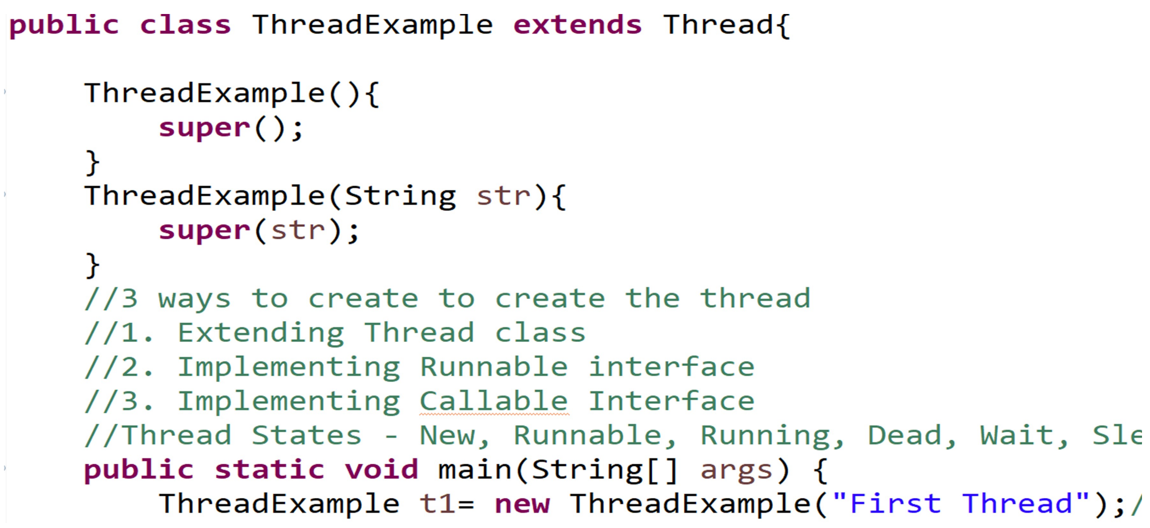
Threads

- **What is Thread** ? Thread is a independent flow of execution with in a same program
- **How many ways we can create thread object**
  3 ways(extending thread class, implementing runnable interface, implementing callable interface)
  - **Thread Object creation by extending the Thread class**

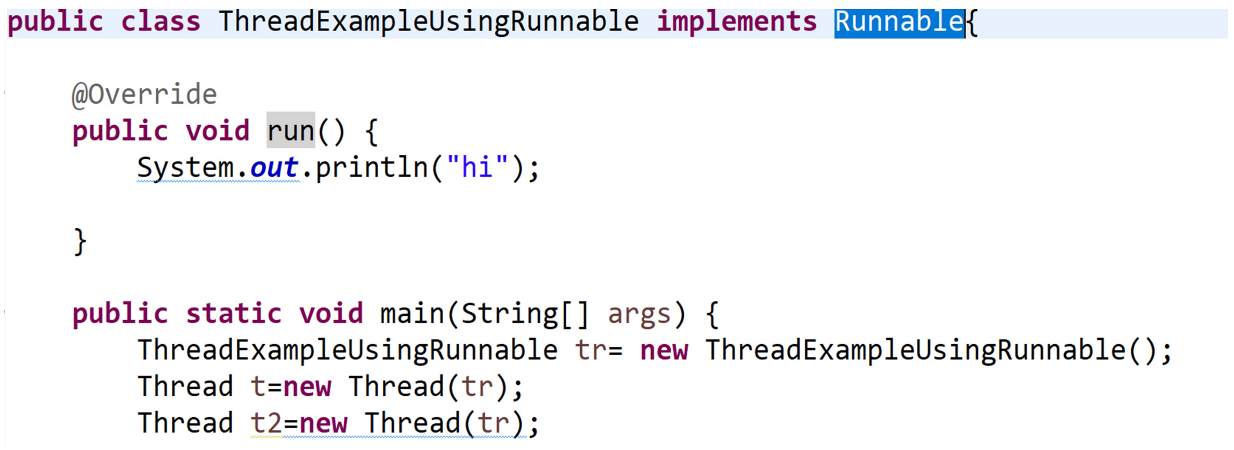


```
public class ThreadExample extends Thread{

    ThreadExample(){
        super();
    }
    ThreadExample(String str){
        super(str);
    }
    //3 ways to create to create the thread
    //1. Extending Thread class
    //2. Implementing Runnable interface
    //3. Implementing Callable Interface
    //Thread States - New, Runnable, Running, Dead, Wait, Sle
    public static void main(String[] args) {
        ThreadExample t1= new ThreadExample("First Thread");/
```

Whenever we are extending the thread class by creating the child class object we can have the thread object

- **Creating the Thread Object by implementing the Runnable Interface.**



```
public class ThreadExampleUsingRunnable implements Runnable{

    @Override
    public void run() {
        System.out.println("hi");

    }

    public static void main(String[] args) {
        ThreadExampleUsingRunnable tr= new ThreadExampleUsingRunnable();
        Thread t=new Thread(tr);
        Thread t2=new Thread(tr);
```

Start() method present in **thread class** is responsible to create the independent path of execution.
Thread flow of execution starts from run method.
Start() method internally calls run method.

- **Why to override the run method when we are creating the Thread by extending thread class?**
  Thread class internally implements runnable interface and it has overridden the run method. If we don't override the run method, once the start method is called Thread class run method will be

invoked but we cannot write any code in side thread class run method. There is no use of thread. Hence by overriding the run method, we can write our own code inside run method which gets executed for every thread.

- **What are the different Thread states**

  Thread states (New, Running, runnable, dead)

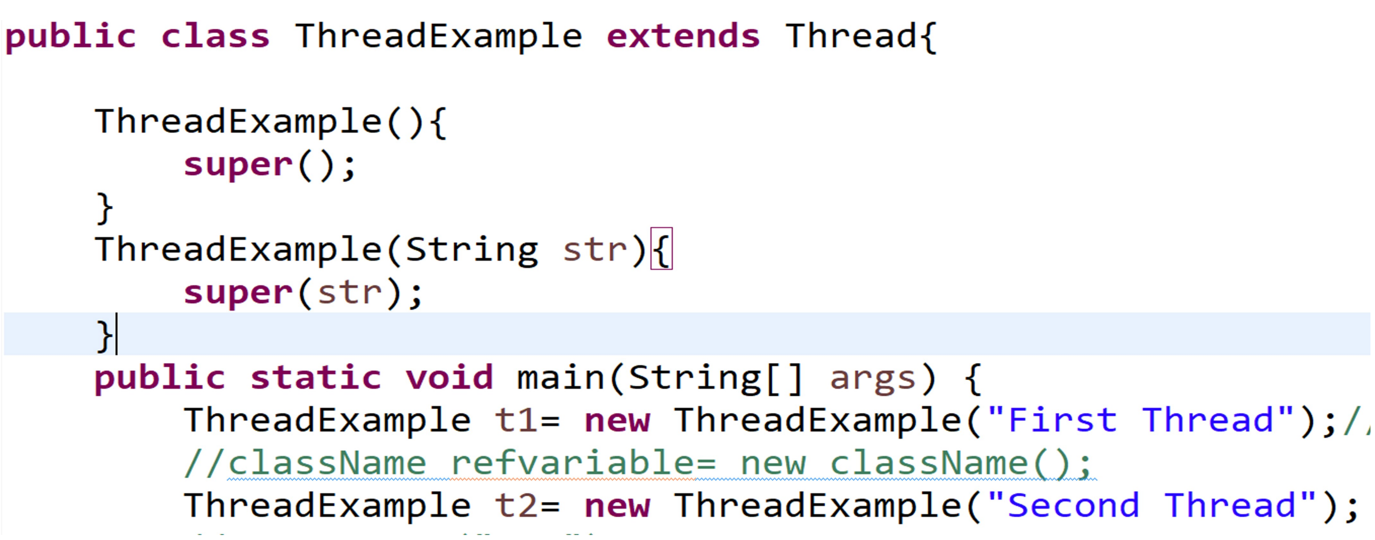
  New : This is the state when the thread object is created but start method is not invoked on it

  Runnable: after invoking the start method thread will go to runnable pool **Running**: from the runnable pool, JVM or scheduler will pick the thread based on the priority and may be some other factors and start the execution that means start executing the run method.

  Dead: after the completion of run method thread is considered as Dead.
- **When the thread flow of execution starts ?**

  Immediately after calling the start method
- **Why to override run method while creating the thread using the Runnable interface ?**

  because Runnable is interface and it has abstract method called run. Hence when we implement the runnable interface we have to override run method)
- **What is the difference between start and run method**

  Start will create the thread execution (independent path of execution) but run will be treated as normal method execution
- **What happens when start method is called on dead thread (or) What happens when we call start method twice ?**
  - Illegal Thread state exception will be thrown. When we call the start method, internally in the start method there is a variable called thread status. Initially it is zero when ever we call it and after the independent path of execution is created the value is set to 5 or any other non zero value. Later when ever we call start method on the same thread object it checks what is the thread status value if it is !=0 then illegal thread state exception will be thrown.
- **How to set the Name to a Thread ?**
  - By calling the Thread class constructor which takes String argument we can set the name of thread
  - 

```
public class ThreadExample extends Thread{

    ThreadExample(){
        super();
    }
    ThreadExample(String str){
        super(str);
    }
    public static void main(String[] args) {
        ThreadExample t1= new ThreadExample("First Thread");/
        //className refvariable= new className();
        ThreadExample t2= new ThreadExample("Second Thread");
```
  - Or by calling the setName method we can set name to thread
  - setName should be called before calling the start method

```java
ATMThread ATM= new ATMThread(karthikAccnt);
ATM.setName("ATM Thread");

GooglePay GooglePay= new GooglePay(ShankarAccnt);
GooglePay.setName("GooglePay Thread");

ATM.start();
GooglePay.start();
```

- **How to know which thread is executing ?**
  - By using the static method currentThread present in Thread class we can know.
  -
```java
Thread t=Thread.currentThread();
System.out.println(t.getName());

System.out.println(Thread.currentThread().getName());
```

- **What is synchronization**
  - Synchronization is needed when multiple threads are working on same object, there may be a possibility of getting the wrong output. But by making the methods as synchronized or adding the synchronized keyword to method signature, we can allow only one thread to execute one method at a time. All other threads executing any other method on same object will go on wait state
  - Example:

```java
synchronized void withdraw(int withdrawalAmount) {
    accountBalance=this.accountBalance-withdrawalAmount;
}
```

- **How synchronization is implemented**
  - By using the synchronized keyword
- **What is Object Lock ?**
  - It is the mechanism when we want to synchronize the non-static method or non-static code block such that only one thread will be able to execute the code block on the given instance of the class
  - In object lock if one thread acquires the lock on object then any other thread cannot access any other synchronized methods on the same object but they can access non synchronized methods
  - So the Synchronized methods in object lock are instance methods
  - If the threads are working on different object of a class then lock acquires on different objects
  - Assume on a bank account if you are performing different operations like ATM Withdrawal and Net banking deposit. Here Karthik Account is Object ATM is one Thread Net Banking is another Thread Methods are Deposit and Withdrawal
  - Like If ATM thread is working on Withdrawal method on Karthik Account then Lock will be

acquired on Karthik Account So in order to make the data consistency or thread safe any other thread like Net banking should not be able to access any other methods like Deposit or Send Money to other threads on the same Karthik bank account. This is called Object lock

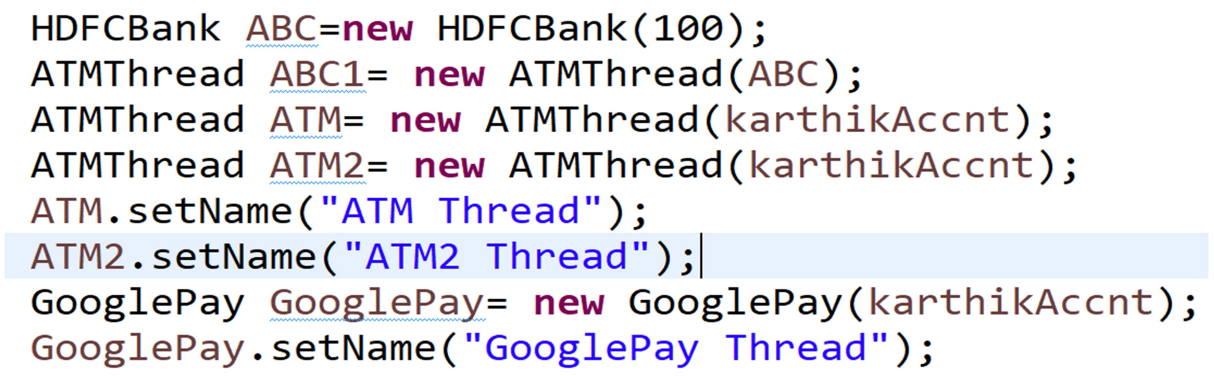
- **What is class Lock ?**
  - Whereas if we want to make the class level data means static methods thread safe then we should go for class lock
  - In the above example if the synchronized methods are static methods then we don't need the object to call the static methods.
  - When synchronized methods are static methods, then the lock that acquires is class Lock.
  - In class lock if one thread acquires the lock on class reference then any other thread cannot access any other synchronized static methods on the same class reference until the first lock is released but they can access non synchronized static methods
- **What is the difference between sleep and wait methods ?**

	Sleep	**Wait**
1.	This is Static method. Present in Thread class	This is instance method. Present in Object class
2.	Sleep accepts milli seconds. The Thread will sleep for the specified time and after that starts the execution if system resource is not busy	Thread will be in waiting status unless until notify or notify all is called
3.	Sleep doesn't release the lock	Wait will release the lock
4.	Sleep can be used with or with out synchronized block	Wait can be used only with synchronized block. If used in non synchronized block then it throws illegal monitor state exception
5.	Thread is used to pause the thread execution for some time	Wait and notify or notify all are used for inter thread communication

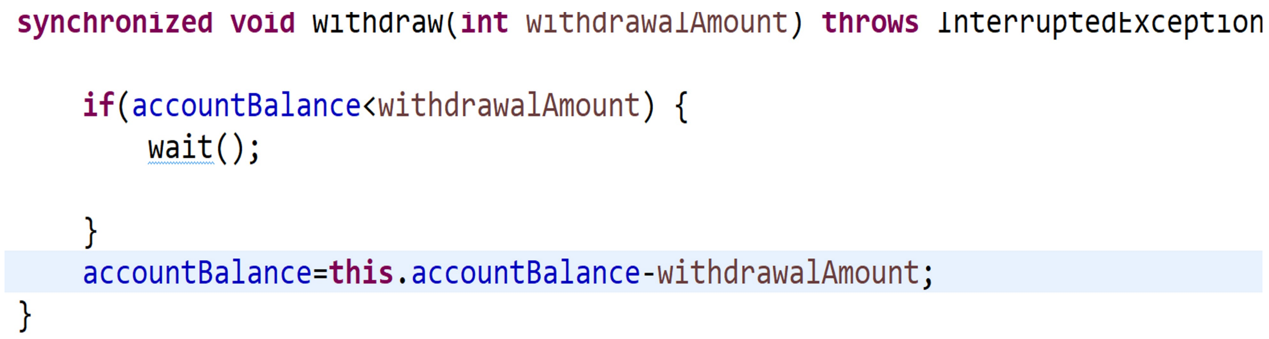
- **What is the disadvantage of Thread ?**
  - We cannot reuse the Thread. Once the thread has completed the start method we cannot call the start method again to bring back the thread to execution. That means in multi threading we should create multiple thread which is disadvantage. To over come this Thread pool concept is introduced.
- **What is Thread pool ?**
  - Thread Pool: It is mainly for re purpose of threads which are already created. Because on normal thread if you call start after completion on run method system throws illegal thread state exception
- **What are the different Thread pools are available in Java ?**
  - Single Thread pool
  - Fixed Thread pool

    - Cached Thread pool
    - Scheduled thread pool
- How thread pools can be implemented in Java ?
    - To implement the thread pool in Java, executor framework is used. This is introduced in Java 1.5 under java.util.concurrent package. Since its introduction in Java 5, the Executor Framework has become a standard tool for concurrent programming in Java
- **What are the maximum number of threads that we can create using thread pool ?**
    - The maximum number of threads in a thread pool can be calculated using the following formula:
    - maximumPoolSize = (processorCount * desiredThreadPoolUtilization) + overheadThreadCount
    - where processorCount is the number of available processors in the system, desiredThreadPoolUtilization is the target utilization percentage for the thread pool, and overheadThreadCount is an additional number of threads that may be required for handling blocking I/O or other tasks that may cause threads to be blocked.
      Here's an example:
    - Suppose we have a system with 8 processors, and we want to create a thread pool that is 80% utilized. We also estimate that we need an additional 2 threads for handling blocking I/O.
    - maximumPoolSize = (8 * 0.8) + 2 maximumPoolSize = 8
    - So in this example, the maximum number of threads in the thread pool should be set to 8.
    - It's important to note that the maximum pool size should be carefully chosen to avoid overloading the system with too many threads, which can lead to performance degradation and increased resource consumption
- **What is the difference between notify and notify All ?**
    - both notify() and notifyAll() will notify the object for which the lock is acquired on the same object.
    - When a thread calls the notify() method on an object, it wakes up one of the threads that are waiting on that object's monitor or lock. The thread that is awakened will then compete with other threads that are blocked on the same monitor for the object's lock.
    - When a thread calls the notifyAll() method on an object, it wakes up all the threads that are waiting on that particular object's monitor or lock. All of the awakened threads will then compete with other threads that are blocked on the same monitor for the object's lock.
    - In both cases, the threads that are waiting on the same object's monitor will be notified and will compete for the lock on that same object.
    - Notify All: Assume that A and B are threads and acquired lock on particular object e (which is an object of Employee class) on the same object (e) if any other thread calls notify all then all the threads which are waiting will come back to execution. Threads waiting on other object of the same class (e1 of Employee)will not come back to execution
    - Notify: Assume that A and B are threads and acquired lock on particular object e (which is an object of Employee class) on the same object (e) if any other thread calls notify then any one thread which is waiting will come back to execution, remaining threads will be in waiting mode only Threads waiting on other object of the same class (e1 of Employee)will not come back to execution

- **Why wait, notify and nootifyall are present in Object class and why not in Thread class ?**
  - because they are used to control the behavior of threads that are accessing objects.
  - In Java, threads execute code in the context of an object. When a thread accesses an object, it may need to wait for another thread to release the object's lock. This is where the wait(), notify(), and notifyAll() methods come into play.
  - The wait(), notify(), and notifyAll() methods allow threads to communicate and synchronize with each other when accessing shared objects. When a thread calls the wait() method on an object, it releases the object's lock and waits until another thread notifies it. When a thread calls the notify() method on an object, it wakes up one of the threads that are waiting on that object's monitor. When a thread calls the notifyAll() method on an object, it wakes up all the threads that are waiting on that object's monitor.
  - Since the wait(), notify(), and notifyAll() methods are used to control the behavior of threads that are accessing objects, it makes sense to include these methods in the Object class rather than the Thread class. This way, these methods can be used with any object that needs to be accessed by multiple threads, regardless of whether those threads are executing in the context of the same or different threads.
  - Look at the code for example



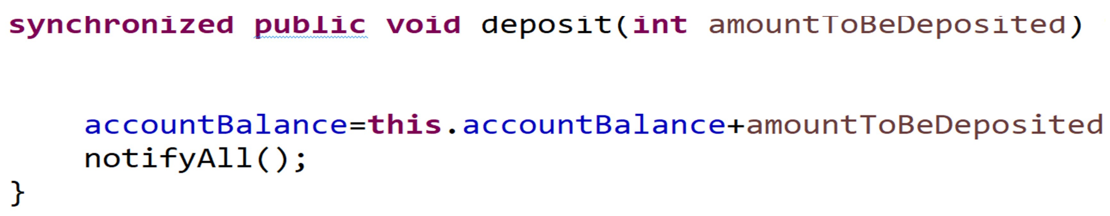
```
HDFCBank ABC=new HDFCBank(100);
ATMThread ABC1= new ATMThread(ABC);
ATMThread ATM= new ATMThread(karthikAccnt);
ATMThread ATM2= new ATMThread(karthikAccnt);
ATM.setName("ATM Thread");
ATM2.setName("ATM2 Thread");
GooglePay GooglePay= new GooglePay(karthikAccnt);
GooglePay.setName("GooglePay Thread");
```



```
synchronized void withdraw(int withdrawalAmount) throws InterruptedException

    if(accountBalance<withdrawalAmount) {
        wait();

    }
    accountBalance=this.accountBalance-withdrawalAmount;
}
```



```
synchronized public void deposit(int amountToBeDeposited)


    accountBalance=this.accountBalance+amountToBeDeposited
    notifyAll();
}
```

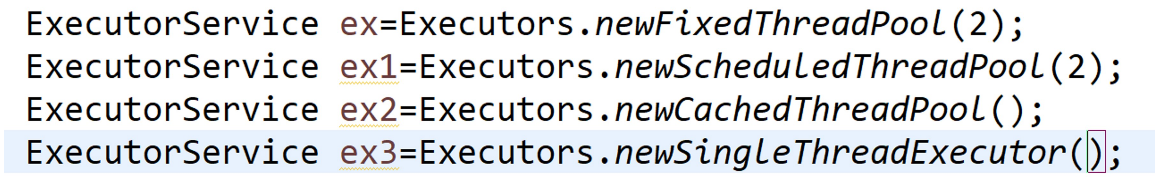
  - In the above example, ATM, ATM2, google pay are working on same object, and ABC thread is working on different object. so there are 2 threads went to waiting mode on same object another thread ABC went to waiting mode on different object of same class. When google pay does notifyAll, that means what ever the locks are present on the object on which ATM, ATM2 threads are working, these threads will come back to the execution not the ABC Thread because ABC thread has acquired the lock on different object.

- When google pay thread does the notify then only any one thread which are working on the same object, in this example any one thread wither ATM or ATM2 thread will come back to execution not ABC thread, as ABC thread has acquired the lock on different object.

- So here in the above example one thread is telling all other waiting threads on the same object to come back to the execution, indirectly we can say that one threads is communicating to other thread. Hence it is called as Inter thread communication. Wait, Notify and Notifyall should be in Object class because threads are talking based on the lock acquired on object and object class is the base class for all classes hence these methods should be object class.

- **What is the drawback of Thread ?**
  - The disadvantage of a thread is we cannot re use it. Once the thread completed its execution then we cannot call start method and again. If we call it throws illegal thread state exception.
- **What is Thread pool ?**
  - To overcome the disadvantage of thread we use thread pool, using thread pool we can re use the threads
  - Thread pool is a mechanism in Java for efficiently managing a group of threads to perform a set of tasks. It involves creating a fixed number of worker threads and submitting tasks to the pool for execution. Thread pool provides a number of benefits over creating new threads for each task:
  - Reusing threads: Creating a new thread is a relatively expensive operation, as it involves allocating memory and other resources. Thread pool allows the reuse of existing threads, which can significantly reduce the overhead of creating and destroying threads.
  - Managing thread concurrency: Thread pool provides a way to limit the number of threads that are actively executing at any given time. This helps to prevent resource contention and avoid thread starvation.
  - Simplifying thread management: Thread pool abstracts away the details of thread management, allowing developers to focus on the logic of their application rather than the low-level details of thread creation and destruction.
  - In Java, the java.util.concurrent package provides several classes for implementing thread pool:
- **How many thread pools are present in Java ?**
  - Fixed thread pool
  - Cached thread pool
  - Scheduled thread pool
  - Single thread pool
- **How the thread pool can be implemented in Java ?**
  - In order to implement the thread pool, executor framework is introduced in java 1.5

onwards

- In executor framework Executors is a class which has methods to create the thread pool. Please see the below screen on how to create the thread pool
- 

```
ExecutorService ex=Executors.newFixedThreadPool(2);
ExecutorService ex1=Executors.newScheduledThreadPool(2);
ExecutorService ex2=Executors.newCachedThreadPool();
ExecutorService ex3=Executors.newSingleThreadExecutor();
```

- **Executors**: This class provides static factory methods for creating thread pool implementations, such as newFixedThreadPool, newCachedThreadPool, and newSingleThreadExecutor.
- **ExecutorService**: This interface provides methods for submitting tasks to the thread pool, such as submit, invokeAll, and invokeAny.
- Start() method is responsible to create a thread when we implement by extending Thread class and calling start method on thread object
- But when the executor framework is used, submit method or execute method is responsible to create thread or independent path of execution

➢ **What are the difference between execute method and submit method ?**

Feature	execute Method	submit Method
Return Value	void	Future object that represents the task's result
Exceptions Handling	Throws RejectedExecutionException	Exceptions are caught and wrapped in Future object
Task Cancellation	Cannot be cancelled	Can be cancelled
Overloaded Methods	Only accepts Runnable objects	Accepts both Runnable and Callable objects
Usage	Use when you don't need the result	Use when you need the result of the task

➢ **What is the reason to introduce submit method in executor service interface or why do we use submit method in executor service interface ?**

- When the thread is running and in case if we want to return some data from thr thread that means if we want to return something from run method, we cannot return as return type of run method is void. So to over come this issue Callable interface is introduced in Java.

➢ **What is callable interface in Java ?**

- Using the callable interface we can return some value from thread
- Callable interface has call method so when we create the thread we should override the call method
- In multi threading when we implement callable interface, in order to start the thread we should use Submit method on executor service interface

➢ **What are the differences between Callable and Runnable**

	Runnable	Callable<V>
Interface Type	Functional Interface	Functional Interface
Method Signature	void run()	V call() throws Exception
Return Type	void	Generic Type V
Exception Handling	None	Throws Exception
Usage	Used to execute a task asynchronously	Used to execute a task and return a result

- Here are some notes on using thread pool in Java:
- Choosing the right thread pool implementation: The choice of thread pool implementation depends on the specific requirements of your application. For example, newFixedThreadPool is suitable for applications with a fixed number of tasks, while newCachedThreadPool is suitable for applications with a large number of short-lived tasks.
- Submitting tasks to the thread pool: Tasks can be submitted to the thread pool using the submit method of ExecutorService. This method returns a Future object that represents the result of the task.
- Handling task exceptions: If a task throws an exception, the exception is caught by the thread pool and logged. It is important to handle task exceptions appropriately to avoid leaving the thread pool in an inconsistent state.

➢ **Differences across thread pools in Java ?**

Thread Pool	Description	Creation Method
Fixed Thread Pool	Creates a thread pool with a fixed number of threads. Once created, the size of the thread pool cannot be changed. If all threads are busy when a new task is submitted, it will be queued until a thread becomes available.	Executors.newFixedThreadPool(int nThreads)
Cached Thread Pool	Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. If a thread is idle for 60 seconds, it will be terminated and removed from the pool.	Executors.newCachedThreadPool()
Single Thread Pool	Creates a thread pool with a single thread. All tasks submitted to the pool will be executed sequentially in the order they were submitted.	Executors.newSingleThreadExecutor()
Scheduled Thread Pool	Creates a thread pool that can schedule tasks to run at a specified time or after a specified delay.	Executors.newScheduledThreadPool(int corePoolSize)

➢ **How to Shutdown the thread pool or how to terminate the thread pool ?**

- Shutting down the thread pool: When the application is finished with the thread pool, it is important to shut down the thread pool using the shutdown method of ExecutorService. This ensures that all threads are terminated and all resources are released.

- **How Schedule thread pool is different from other thread pool ?**
  - Scheduled Thread pool: In order to provide the delay while doing the multi thread programming we use scheduled thread pool.
  - **ScheduledExecutorService ex1=Executors.newScheduledThreadPool(2);**
  - newScheduledThreadPool of Executors class will return ScheduledExecutorService which has internally the scheduled method through which delay can be provided to thread. So in this scenario schedule method is responsible to create independent path of execution or thread
  - Internally, when the **schedule**() method is called, the ScheduledThreadPoolExecutor creates a new thread to execute the task at the specified delay time. The thread is managed by the executor, and there is no need to call submit() or execute() methods explicitly to start the task
- **How to accept the object returned from thread in multi threading ?**
  - When we use callable, it returns the object that means which ever the thread that is executing the call method, that returns the object but we cannot predict when the thread will execute and when it will complete the call method. So We need some placeholder to accept that is coming in future. To do this work Future interface in introduced in Java
- **What is Future interface in Java ?**
  - The Future interface in Java represents a result of an asynchronous computation. It is used when a method needs to return a value that is not immediately available, but will be available at some point in the future after some processing is done.
  - The Future interface provides methods for checking whether the computation is complete, blocking until the computation completes, and retrieving the result of the computation
  - Here's an overview of the methods provided by the Future interface:
  - boolean **cancel**(boolean mayInterruptIfRunning): Attempts to cancel the computation. Returns true if the computation was cancelled successfully, false otherwise.
  - boolean **isCancelled**(): Returns true if the computation was cancelled, false otherwise.
  - boolean **isDone**(): Returns true if the computation is complete, either by finishing normally or by throwing an exception.
  - V **get**() throws InterruptedException, ExecutionException: Waits if necessary for the computation to complete, and then returns the result of the computation. If the computation threw an exception, this method will throw an ExecutionException
- **What is ScheduledFuture in Java ?**
  - When scheduled thread pool is used in order to hold or catch the output of thread scheduled future is used. It internally extends from Future interface.
  - The Future and ScheduledFuture interfaces provide ways to interact with and retrieve results from tasks that are executed asynchronously. The main difference between them is that Future represents a result that will be available in the future, while ScheduledFuture represents a result that will be available in the future at a specific time.
- **What are the drawback of Synchronized key word ?**
  - When a thread holds a lock on a synchronized block, other threads that require access to the same block must wait until the lock is released. Once the lock is released any

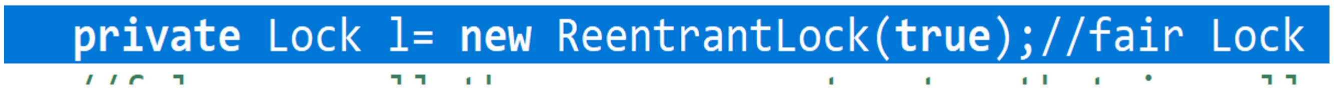
waiting thread can comeback to execution. There is no preference for a long waiting thread to execute
   - Using Synchronized keyword we cannot get the lock across methods.

➢ **What is Lock interface ?**
   - The Lock interface is an alternative mechanism to synchronization for providing mutual exclusion. It provides more flexibility and control over the locking behavior than synchronization. The Lock interface is part of the java.util.concurrent.locks package and provides methods for acquiring and releasing locks

➢ **What are the Differences between Lock and synchronized ?**

Feature	Lock	synchronized
Acquiring a lock	Use the lock() method of the Lock interface	Automatically acquired when entering a synchronized block or method
Releasing a lock	Use the unlock() method of the Lock interface	Automatically released when exiting a synchronized block or method
Waiting for a lock	Use the tryLock() method or lockInterruptibly() method of the Lock interface	Automatically waits until the lock is available
Fairness	Can be set to fair or unfair using the ReentrantLock(boolean fair) constructor By passing true to the constructor, fairness will be acheved	Always unfair
Scope	Can be applied across multiple methods or even across threads	Limited to a single method or block

➢ **What is Re-entrant Lock and what is fair lock and un fair lock in Java ?**
   - Reentrant lock is a class in java which implements Lock interface. This class has 2 constructors.



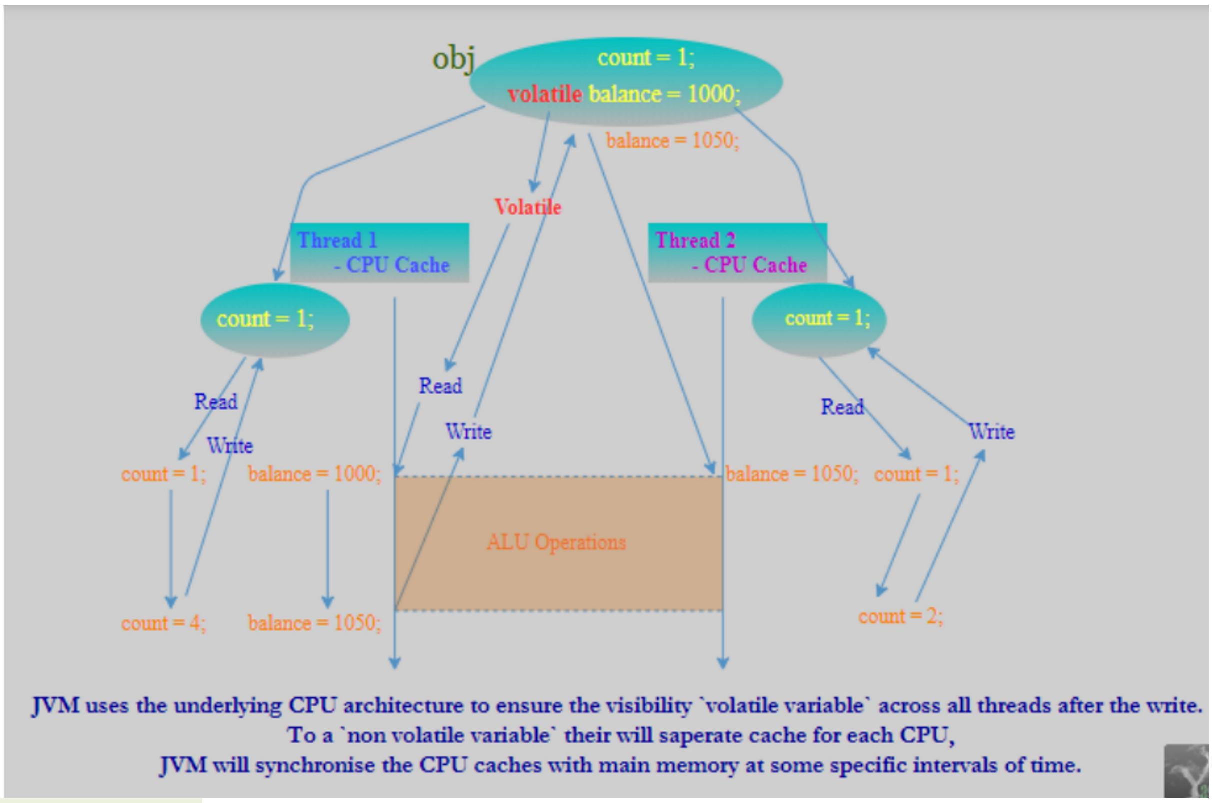
```
private Lock l= new ReentrantLock(true);//fair Lock
```

- Fair lock can be achieved by passing true to the Re entrant lock class.
- Fair lock means. On the same object, currently 1 thread is executing and multiple threads are waiting then once the current execution thread releases the lock, the thread that is waiting for longer time will be given the access. That means once the lock is released, another thread will be given access based on the waiting time.
- In a fair lock, threads are granted access to the lock in the order in which they requested it, i.e., the lock is granted to the thread that has been waiting the longest. This ensures that all threads get an equal chance to acquire the lock and prevents any thread from being blocked indefinitely.
- In unfairlock, pass false or call no arg constructor of ReentrantLock class then unfair lock is achieved. In unfairlock after the lock is released any thread can execute. Waiting time doesn’t matter.

    - In contrast, an unfair lock grants access to the lock to any thread that requests it, without regard for the order in which threads arrived. This can lead to some threads being blocked for a long time while other threads are repeatedly granted access to the lock.

- **What is read write lock in java ?**
    - In Java, the ReadWriteLock interface is implemented by the ReentrantReadWriteLock class. This class provides two locks, a read lock and a write lock, that can be acquired and released by threads. **The read lock can be acquired by multiple threads at the same time, as long as no thread holds the write lock.**
    - The write lock is exclusive, meaning that only one thread can hold it at a time, and it prevents any other thread from holding the read lock
    - In other wards, multiple read threads can work at a time and write lock thread will wait for the read threads to complete, but when write lock starts the execution then it acquires lock and none of the read thread will be able to execute.
    - To acquire the read or write lock, you need to call the readLock() or writeLock() method, respectively, on an instance of ReentrantReadWriteLock. Once you have acquired the lock, you should perform the necessary operations on the shared resource, and then release the lock by calling the unlock() method on the lock object
- **What is Join method in Threads ?**
    - join() is a method in Java that allows one thread to wait for the completion of another thread before continuing its execution. When a thread calls the join() method on another thread, the calling thread is blocked until the other thread finishes executing.
    - The syntax for the join() method is as follows.
    - In a real-time banking application, one possible scenario where we can use the join() method of threads is during the processing of a large batch of financial transactions.
    - Suppose that the application receives a batch of financial transactions from multiple sources, and each transaction needs to be processed and validated before it can be approved and committed to the database.
    - To improve performance and reduce the time required to process the entire batch of transactions, the application can use multiple threads to process the transactions in parallel. For example, one thread can be responsible for processing the transaction data, another thread can be responsible for validating the transaction data, and a third thread can be responsible for committing the approved transactions to the database.
- **What is the disadvantage of Lock interface in java ?**
    - Risk of forgetting to release the lock: Unlike synchronized, where the lock is automatically released when the synchronized block or method exits, Lock requires explicit locking and unlocking. This can increase the risk of forgetting to release the lock, which can cause deadlocks or other problems
    - Developer might forget to write lock.unlock which make the lock cannot be released for ever.
- **What is Volatile keyword in Java ?**
    - Volatile Keyword is applicable to variables. volatile keyword in Java guarantees that value of the volatile variable will always be read from main memory and not from Thread's local cache.
    - Volatile Field: An indication to the VM that multiple threads may try to access/update the field's value at the same time. To a special kind of instance variables which has to shared

among all the threads with Modified value. Similar to Static(Class) variable, Only one copy of volatile value is cached in main memory, So that before doing any ALU Operations each thread has to read the updated value from Main memory after ALU operation it has to write to main memory direclty. (A write to a volatile variable v synchronizes-with all subsequent reads of v by any thread) This means that changes to a volatile variable are always visible to other threads.

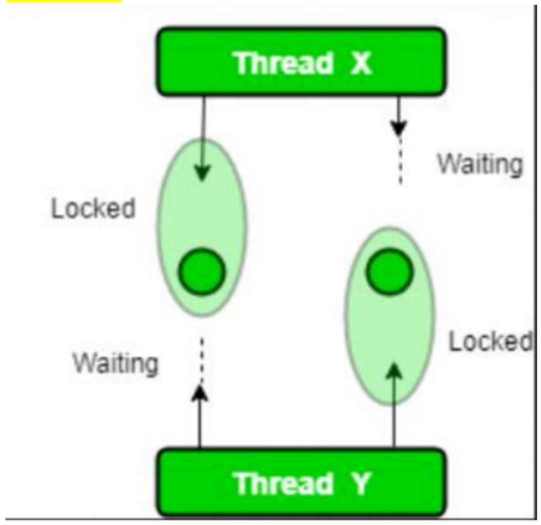

obj
count = 1;
volatile balance = 1000;
balance = 1050;
Volatile
Thread 1 - CPU Cache
Thread 2 - CPU Cache
count = 1;
count = 1;
Read
Write
Read
Write
Read
Write
count = 1;
balance = 1000;
balance = 1050;
count = 1;
ALU Operations
count = 4;
balance = 1050;
count = 2;
JVM uses the underlying CPU architecture to ensure the visibility `volatile variable` across all threads after the write.
To a `non volatile variable` their will saperate cache for each CPU,
JVM will synchronise the CPU caches with main memory at some specific intervals of time.

- **What is Yield method ?**
    - The yield() method is a method in the Thread class of Java, and it is used to pause the currently executing thread and give other threads an opportunity to run. When a thread calls the yield() method, it indicates that it is willing to give up the CPU for a while. The scheduler can then choose to resume any other thread that is waiting to run.

- **What is join method in Thread ?**
    - The join() method in Java is used to wait for the completion of a thread before the execution of the calling thread resumes. When a thread calls join() on another thread, it waits until that thread completes its execution before continuing. This is useful in situations where the main thread needs to wait for other threads to complete their tasks before continuing.

- **What is Dead Lock in Threads and how to avoid the deadlock ?**

Dead Lock

▪ 

- Deadlock in threads is a situation in which two or more threads are blocked waiting for each other to release a resource or a lock, resulting in a standstill in the execution of the program. This situation occurs when two threads are waiting for each other to complete a certain task, and neither thread can proceed until the other thread releases the resource it is holding
- To avoid deadlock in Java, there are a few strategies:
- Avoid acquiring multiple locks in different order across threads. Instead, try to acquire all locks in the same order across all threads.
- Use a timeout when acquiring locks. If a lock cannot be acquired within a certain time period, release the lock and try again later.
- Use the tryLock() method instead of the synchronized keyword to acquire locks. The tryLock() method allows a thread to attempt to acquire a lock, but immediately return if the lock is not available, so that the thread can try again later.
- Use higher-level concurrency constructs like Semaphore, CountDownLatch, and CyclicBarrier, which provide more advanced synchronization mechanisms that can help avoid deadlock.

➢ **What is SemaPhore in Java ?**

- Semaphore in Java is a synchronization mechanism that allows a fixed number of threads to access a shared resource at the same time. It maintains a count of the number of permits available, and each thread that wants to access the shared resource must acquire a permit from the semaphore. If no permits are available, the thread will block until a permit becomes available.
- Semaphore in Java internally maintains a counter that represents the number of available permits. When a thread tries to acquire a permit from the semaphore, the counter is decremented. If the counter is greater than zero, the thread is granted the permit and allowed to proceed. If the counter is zero, the thread is blocked until a permit becomes available.
- When a thread releases a permit back to the semaphore, the counter is incremented. If there are any threads waiting to acquire a permit, one of them is granted the permit and allowed to proceed.

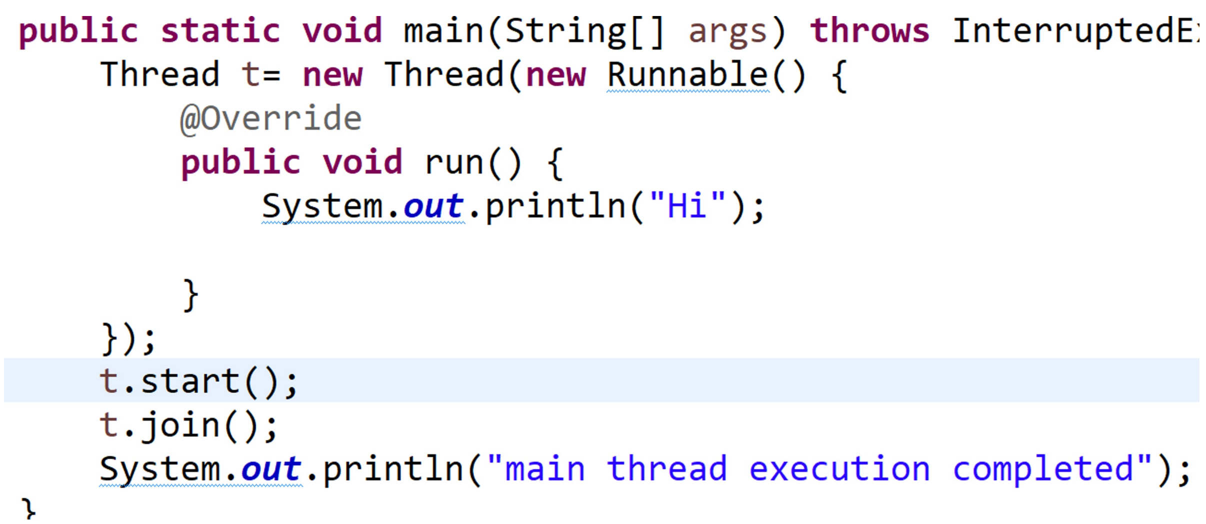
➢ **How does Semaphore avoids the dead lock ?**

- Semaphore can help avoid deadlock in multi-threaded programs by providing a way to control access to shared resources. Deadlock occurs when two or more threads are blocked waiting for each other to release resources that they need. By using a semaphore to limit the number of threads that can access a shared resource at any given time, we can reduce the likelihood of threads blocking each other and causing a deadlock.

- For example, suppose we have two threads that both need to access two shared resources, A and B. If both threads try to acquire A and B in the same order (i.e., Thread 1 acquires A first and Thread 2 acquires B first), they can become deadlocked if one thread holds A and is waiting for B, while the other thread holds B and is waiting for A.
- However, if we use a semaphore with a count of 1 for each shared resource, we can ensure that only one thread can access each resource at any given time. This means that if Thread 1 acquires A first, it will be able to access B while Thread 2 waits for A to become available. Once Thread 1 releases A, Thread 2 can acquire it and access B. By controlling the order in which threads acquire shared resources, we can prevent deadlocks from occurring.
- In addition to limiting the number of threads that can access a shared resource at any given time, semaphores can also be used in conjunction with other synchronization mechanisms like locks to provide more advanced concurrency control. By carefully managing the use of shared resources and coordinating the interactions between threads, we can avoid deadlocks and ensure that our multi-threaded programs are safe and efficient.

➢ **What is join method in Thread ?**

- join() is a method that can be called on a thread object in Java. It causes the calling thread (in this case, the main thread) to wait for the completion of the thread it is called on (in this case, the t thread). Once the t thread completes its execution, the main thread resumes its execution. Here is an example program that demonstrates the use of join()



```
public static void main(String[] args) throws InterruptedE
    Thread t= new Thread(new Runnable() {
        @Override
        public void run() {
            System.out.println("Hi");

        }
    });
    t.start();
    t.join();
    System.out.println("main thread execution completed");
}
```

➢ **What is countdown latch ?**

- CountDownLatch is a synchronization mechanism provided by the Java standard library that allows one or more threads to wait for a set of operations to complete before proceeding further.
- CountDownLatch is initialized with a count, which specifies the number of operations that need to be completed before the waiting threads are released. The count is decremented each time an operation completes, until it reaches zero. Once the count reaches zero, the waiting threads are released and allowed to proceed.

- CountDownLatch is often used in scenarios where a program needs to wait for multiple independent tasks to complete before proceeding. For example, in a parallel processing system, a program might spawn multiple threads to perform different parts of a computation, and then use CountDownLatch to wait for all threads to finish before combining their results
- join method will make main thread to wait or any other thread to wait until it completes the single thread . where as countdown latch makes sure that in executor framework all the threads have completed their work before the main gets completed.
- In summary, join() is used to wait for a single thread to complete, while CountDownLatch is used to wait for multiple threads to complete before proceeding

- **What is atomicity in Java ? What is the need for Atomic variables ?**
  - In multi-threaded programming, it is possible for multiple threads to access and modify the same variable concurrently. When multiple threads modify a variable concurrently, there is a possibility of data races, which can result in incorrect or unpredictable behavior of the program
  - To avoid data races, we need a way to ensure that a variable is accessed and modified atomically, i.e., by a single thread at a time. This is where atomic variables come in. Atomic variables provide a way to modify a variable atomically, without the need for locks or other synchronization mechanisms
  - An atomic variable is a variable that is updated atomically, i.e., in a way that ensures that no other thread can modify the variable at the same time. In Java, atomic variables are provided by the java.util.concurrent.atomic package. Some common examples of atomic variables are AtomicInteger, AtomicBoolean
  - when we use synchronized block, there is no issue to variables. because one thread will work at a time on same object. but when there is no synchronized block multiple threads can execute multiple methods on same object and the value of one variable is not visible to other thread. so in this scenario we can use atomic variable instead of normal variable. that means atomic can be more effective when there is no synchronized or lock mechanism
  - When we use a synchronized block, only one thread can execute the block at a time, which ensures that there are no data races on shared variables. So, if all accesses to shared variables are within synchronized blocks, then we don't need to use atomic variables

- **Difference between Volatile atomic variables ?**
  - Volatile and atomic variables are both used in Java for concurrent programming, but they differ in how they provide thread-safety.
  - Volatile variables provide a way to ensure that the variable's value is always read from and written to main memory, rather than from a thread's local cache. This ensures that changes made to the variable by one thread are visible to all other threads immediately. However, volatile variables do not ensure atomicity, i.e., they do not ensure that multiple threads accessing the variable do so atomically. If multiple threads modify a volatile variable concurrently, there is still a risk of data races and other concurrency issues.
  - On the other hand, atomic variables provide a way to ensure that operations on the variable are atomic, i.e., thread-safe. They ensure that multiple threads can modify the variable concurrently without causing data races. Atomic variables are implemented using low-level CPU instructions to ensure that the variable is accessed and modified atomically.
  - The difference between volatile and atomic variables is that volatile variables provide visibility guarantees, i.e., they ensure that changes made to the variable by one thread are immediately

visible to other threads. Atomic variables, on the other hand, provide both visibility and atomicity guarantees, i.e., they ensure that operations on the variable are thread-safe.

- In summary, while both volatile and atomic variables provide some form of thread-safety, atomic variables provide stronger guarantees of thread-safety by ensuring both visibility and atomicity, which helps to prevent data races and other concurrency issues.

- **What is Thread Local ?**
    - In Java, ThreadLocal is a class that provides thread-local variables. A thread-local variable is a variable that is local to a particular thread, i.e., each thread has its own copy of the variable. This means that changes made to the variable by one thread do not affect the value of the variable in other threads.
    - The ThreadLocal class provides a simple way to implement thread-local variables in Java. To use a ThreadLocal variable, you first create an instance of the ThreadLocal class, and then call its get() and set() methods to access and modify the value of the variable, respectively. Each thread that accesses the variable will see its own copy of the value

    - In Java, ThreadLocal is a class that provides thread-local variables. A thread-local variable is a variable that is local to a particular thread, i.e., each thread has its own copy of the variable. This means that changes made to the variable by one thread do not affect the value of the variable in other threads.
    - The ThreadLocal class provides a simple way to implement thread-local variables in Java. To use a ThreadLocal variable, you first create an instance of the ThreadLocal class, and then call its get() and set() methods to access and modify the value of the variable, respectively. Each thread that accesses the variable will see its own copy of the value.

    - thread-local variables are different from method-local variables in that they are local to a specific thread, whereas method-local variables are local to a specific method. Thread-local variables are useful for scenarios where you have shared objects that are not thread-safe, but you don't want to synchronize access to them

- **What is the need for Thread Local variables ?**
    - Thread-local variables are useful in scenarios where you have shared objects that are not thread-safe, but you don't want to synchronize access to them. By using thread-local variables, you can ensure that each thread has its own copy of the object, which eliminates the need for synchronization
- **What is Thread Group in Java ?**
    - A thread group in Java is a way of organizing a set of related threads into a single unit. Threads can be added to a thread group, and the group can be used to perform operations on all of the threads in the group, such as interrupting them or setting their priority
    - Advantage of Thread group is that we can call method on a set directly instead of calling individually.
    - Here are some of the methods that can be called on a ThreadGroup in Java:
    - getName(): Returns the name of the thread group.
    - getParent(): Returns the parent of the thread group.
    - activeCount(): Returns an estimate of the number of active threads in the thread group.
    - enumerate(Thread[] list): Copies into the specified array every active thread in the thread group and its subgroups.

- interrupt(): Interrupts all threads in the thread group.
- setDaemon(boolean daemon): Sets the daemon status of the thread group.
- setMaxPriority(int priority): Sets the maximum priority of the thread group.
- uncaughtException(Thread t, Throwable e): Called when a thread in the thread group throws an uncaught exception.
- list(): Prints information about the thread group to the standard output

➢ **What is race condition in Java ? How to prevent race condition ?**

- A race condition in Java is a concurrency bug that occurs when multiple threads access shared resources or variables simultaneously, leading to unpredictable and inconsistent behavior. It can happen when two or more threads execute concurrently and access shared data without proper synchronization, leading to data inconsistency, incorrect results, or even program crashes.
- Synchronization: Synchronization is a technique used to prevent race conditions by ensuring that only one thread can access a shared resource at a time. In Java, synchronization can be achieved using the synchronized keyword or by using Lock objects.
- Atomic Operations: Java provides the java.util.concurrent.atomic package that contains classes that provide atomic operations on variables. These classes ensure that operations on variables are performed atomically and are not interrupted by other threads.
- Thread-Safe Classes: Java provides several thread-safe classes in the java.util.concurrent package that can be used to prevent race conditions. These classes include ConcurrentHashMap, CopyOnWriteArrayList

➢ **What is cyclic barrier ?**

- A cyclic barrier is a synchronization mechanism in Java that allows multiple threads to wait for each other to reach a certain point of execution before continuing. In other words, it allows threads to wait for each other until they all reach a common barrier point, and then they can proceed to the next stage of execution.
- In the insurance domain, a common use case for a cyclic barrier is to simulate a scenario where multiple agents need to wait for all the necessary paperwork to be completed before processing a claim
- in summary, the run() method of the CyclicBarrier instance is invoked automatically when all parties have reached the barrier by calling the await() method on the barrier object
- In the insurance domain, a common use case for a cyclic barrier is to simulate a scenario where multiple agents need to wait for all the necessary paperwork to be completed before processing a claim.

➢ **What is Fork join pool ?**

- The Fork/Join framework is a feature of the Java Concurrency API that enables efficient parallel processing of recursive divide-and-conquer problems. It is designed to take advantage of multiple processors or cores available in a computer to perform parallel execution of tasks. The Fork/Join framework was introduced in Java 7 and is part of the java.util.concurrent package.
- The core component of the Fork/Join framework is the ForkJoinPool, which is a thread pool that manages worker threads and the tasks they execute. The ForkJoinPool uses a work-stealing algorithm, where worker threads that have completed their tasks can steal tasks from other worker threads that are still executing tasks. This allows for better load balancing and reduces the likelihood of idle threads.

- here's an example of using the Fork/Join framework in the insurance domain. The example calculates the total amount of claims paid by an insurance company for a given year, by summing up the amounts of individual claims.
- pool.invoke(task) method starts the execution of the task using the ForkJoinPool, which in turn invokes the compute() method of the ClaimProcessor class to perform the parallel processing of the subtasks.

➢ **What is optimistic lock and pessimistic lock in java.**

- Optimistic locking assumes that conflicts between transactions are rare, and so it allows multiple transactions to proceed concurrently.
- If a conflict is detected, the operation is rolled back and retried. This can be implemented using techniques such as versioning, timestamping, or using atomic variables
- Pessimistic locking, on the other hand, assumes that conflicts between transactions are likely, and so it locks the data being accessed by one transaction to prevent other transactions from modifying it. This guarantees that only one transaction can modify the data at any given time, but it can lead to performance issues if multiple transactions are waiting to access the same data