Unit 7:

Threads

Thread :

- A thread is a lightweight sub-process, the smallest unit of processing and also has separate paths of execution.
- All Java programs have at least one thread, known as the main thread, which is created by the Java Virtual Machine (JVM) at the program's start, when the main() method is invoked.
- A single-threaded application has only one Java thread and can handle only one task at a time. To handle multiple tasks in parallel, multi-threading is used.

Multithreading :

- It is a process of executing two or more threads simultaneously to maximum utilization of CPU.
- Multithreaded applications execute two or more threads run concurrently. Hence, it is also known as Concurrency in Java. Each thread runs parallel to each other.
- Multiple threads don't allocate separate memory area. Hence they save memory. Also, context switching between threads takes less time.

Life Cycle of a Thread

- A thread goes through various stages in its life cycle.
- For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.

1. New :

- A new thread begins its life cycle in the new state.
- It remains in this state until the program starts the thread.
- It is also referred to as a **born thread**.

2. Runnable :

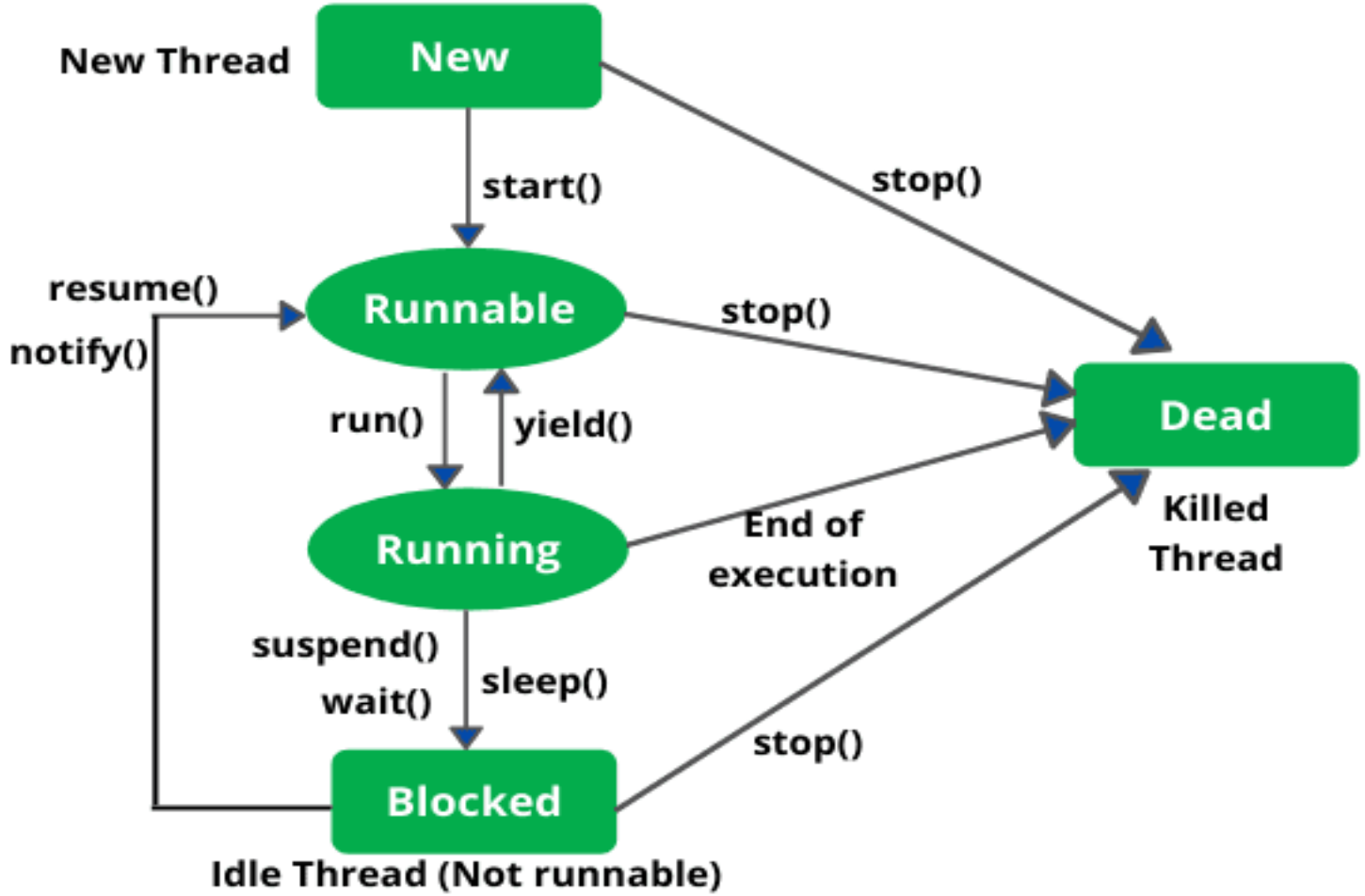
- After a newly born thread is started, the thread becomes runnable.
- A thread in this state is considered to be executing its task.

3. Running :

- When the thread scheduler selects the thread then, that thread would be in a running state.

4. Non-Runnable (Blocked)

The thread is still alive in this state, but currently, it is not eligible to run.



5. Terminated/Dead :

A thread terminates because of either of the following reasons:

- Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
- Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

Create/ Instantiate/ Start New Threads :

We can create Threads in java using two ways, namely :

- By extending Thread Class
- By Implementing a Runnable interface

In both the cases run() method should be implemented.

1. By extending Thread Class

Syntax:

```
class DemoThread extends Thread{
    public void run() {
        //block  of code
    }
}
```

2. By Implementing a Runnable interface

Syntax:

```
class DemoThread implements Runnable{

   public void run() {
        //block of code
    }
}
```

Thread Priorities :

- A component of Java that decides which thread to run or execute and which thread to wait is called a **thread scheduler in Java**.
- In Java, a thread is only chosen by a thread scheduler if it is in the runnable state. However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones.
- There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. **Priority** and **Time of arrival**.

Priority :

- Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.

- In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.
- Java programmer can also assign the priorities of a thread explicitly in a Java program.

- The default priority is set to 5 as excepted.
- Minimum priority is set to 1.
- Maximum priority is set to 10.

Here 3 constants are defined in it namely as follows:

- public static int NORM_PRIORITY
- public static int MIN_PRIORITY
- public static int MAX_PRIORITY

Setter & Getter Method of Thread Priority :

public final int getPriority():

- The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority):

- The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority.
- The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

Synchronization :

- Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results. So, it needs to be made sure by some synchronization method that only one thread can access the resource at a given point in time.

- In General, when two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called **synchronization**.

Why use Synchronization?

The synchronization is mainly used to :

1. To prevent thread interference
2. To prevent consistency problem

Types:
There are two types of thread synchronization :

1. Mutual Exclusive

2. Cooperation (Inter-thread communication in java)

Mutual Exclusive :

- Mutual Exclusive helps keep threads from interfering with one another while sharing data.
- It can be achieved by using the following three ways:

a. By Using Synchronized Method
b. By Using Synchronized Block
c. By Using Static Synchronization

a. By Using Synchronized Method :

1. If you declare any method as synchronized, it is known as synchronized method.
2. Synchronized method is used to lock an object for any shared resource.

3. When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

4. If the object lock is not available, the calling thread is blocked, and it has to wait until the lock becomes available.

- Once a thread completes the execution of code inside the synchronized method, it releases object lock and allows other thread waiting for this lock to proceed.

- That is once a thread completes its work using synchronized method, it will hand over to the next thread that is ready to use the same resource.

Syntax :

```
synchronized Access_Modifiers Return_Type MehthodName(Parameters) {

}
```

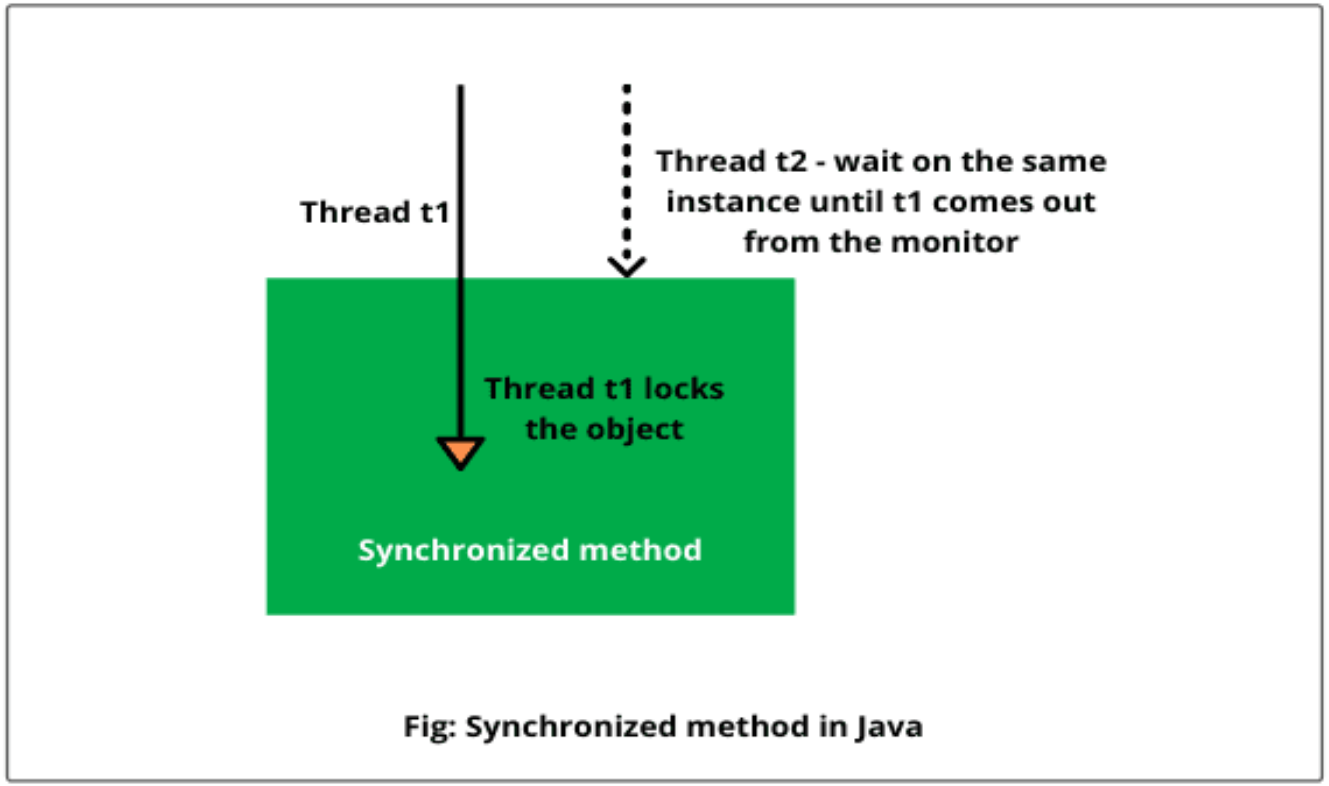


Fig: Synchronized method in Java

b. By Using Synchronized Block

- Synchronized block can be used to perform synchronization on any specific resource of the method.
- Suppose we have 100 lines of code in our method, but we want to synchronize only 10 lines, in such cases, we can use synchronized block.
- If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

- **Syntax :**

```
synchronized (object reference expression) {

}
```

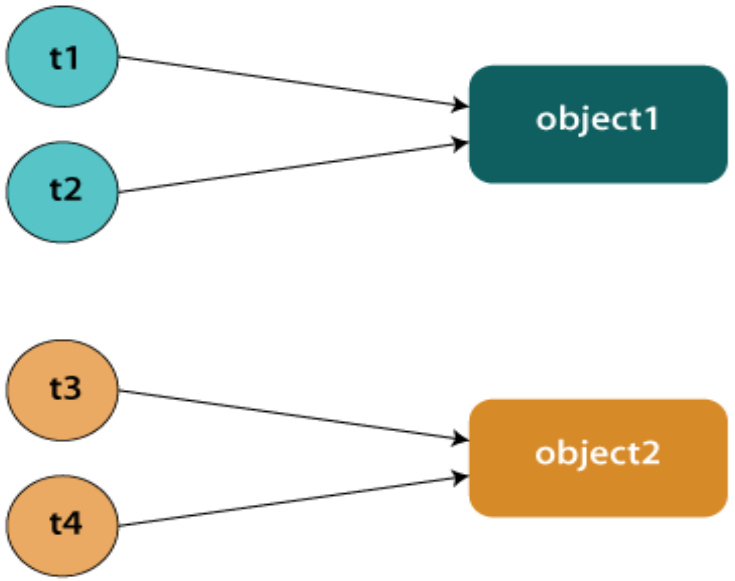
c. By Using Static Synchronization

- Suppose there are two objects of a shared class named object1 and object2.
- In case of synchronized method and synchronized block there cannot be interference between **t1 and t2** or **t3 and t4** because t1 and t2 both refers to a common object that have a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.
- We don't want interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.
- If you make any static method as synchronized, the lock will be on the class not on object.

Syntax :

```
static synchronized Access_Modifiers Return_Type MehthodName(Parameters) {

}
```

2. Co-operation (Inter-thread communication in java)

- Inter-thread communication is a mechanism in which a thread releases the lock and enter into paused state and another thread acquires the lock and continue to executed.

- It is implemented by following methods of **Object class**:

1. wait() :

- The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

2. notify() :

- This method is used to wake up a single thread and releases the object lock.

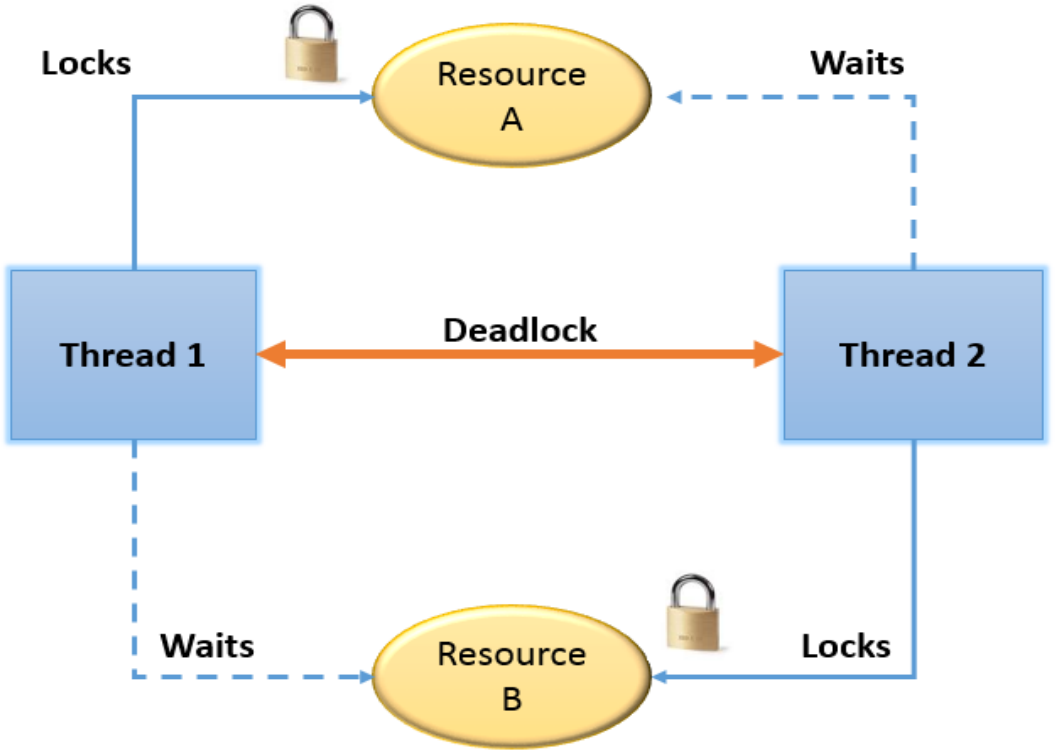
3. notifyAll() :

- This method is used to wake up all threads that are in waiting state.

Deadlock:

- Deadlock in Java is a part of multithreading.
- Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called **deadlock**.

- The thread thread1 holds the lock for the resource A and waits for resource B that is acquired by thread thread2.
- At the same time, thread2 holds the lock for the resource B and waits for A resource that is acquired by thread thread1.
- But thread2 cannot release the lock for resource B until it gets hold of resource A. Similarly, thread1 cannot release the lock for resource A until it gets hold of resource B.
- Since both threads are waiting for each other to unlock resources A and B, therefore, these mutually exclusive conditions are called deadlock in Java.



How to Avoid Deadlock in Java?

1. **Avoid Nested Locks**:
   - ❖ We must avoid giving locks to multiple threads, this is the main reason for a deadlock condition. It normally happens when you give locks to multiple threads.

2. **Avoid Unnecessary Locks**:
   - ❖ The locks should be given to the important threads. Giving locks to the unnecessary threads that cause the deadlock condition.

3. **Using Thread Join**:
   - ❖ A deadlock usually happens when one thread is waiting for the other to finish. In this case, we can use **join** with a maximum time that a thread will take.