Unit-4: Concepts of Thread

Sonawane Khushbu K.

Basics of Thread

- In Java, a thread represents a single path of execution within a program. It's a lightweight subprocess that allows multiple tasks to run concurrently, making your program more efficient and responsive.

- Key Points:
- **Lightweight:**
- Threads share the same memory space of the process they belong to, making them more efficient than creating separate processes.
- **Concurrency:**
- Multiple threads can execute independently, allowing for simultaneous operations like handling user input, performing calculations, and updating the UI.

- Multitasking:
- Threads enable multitasking within a single application. For example, a web browser can download multiple files while still allowing you to browse other pages.
- Improved Performance:
- Threads can help improve performance by utilizing multiple CPU cores, especially for tasks that can be parallelized.

- The thread life cycle in Java represents the various states a thread can be in during its execution. Understanding the life cycle helps in managing thread states and handling concurrency issues effectively.
- **Thread Life Cycle States**
- **New (Born)**
  - **State:** The thread is created but not yet started.
  - **How to reach:** When you create a thread object using new Thread() but haven't called the start() method yet.

- **Example:**
- Thread t = **new** Thread(); // t is in New state

- **Runnable**
- **State:** The thread is ready to run and waiting for CPU time.
- **How to reach:** When the start() method is called, the thread moves from New to Runnable state.
- **Note:** A thread in the Runnable state is either actively running or waiting for CPU time, depending on the JVM's thread scheduler.

- Example:
- t.start(); // t is now in Runnable state

- **Blocked**
- **State:** The thread is blocked and waiting for a monitor lock to enter or re-enter a synchronized block/method.
- **How to reach:** When a thread tries to access a synchronized block/method that is locked by another thread, it enters the Blocked state.
- **Example:**
- synchronized (someObject) {
- // thread t enters Blocked state if another thread holds the lock on someObject
- }

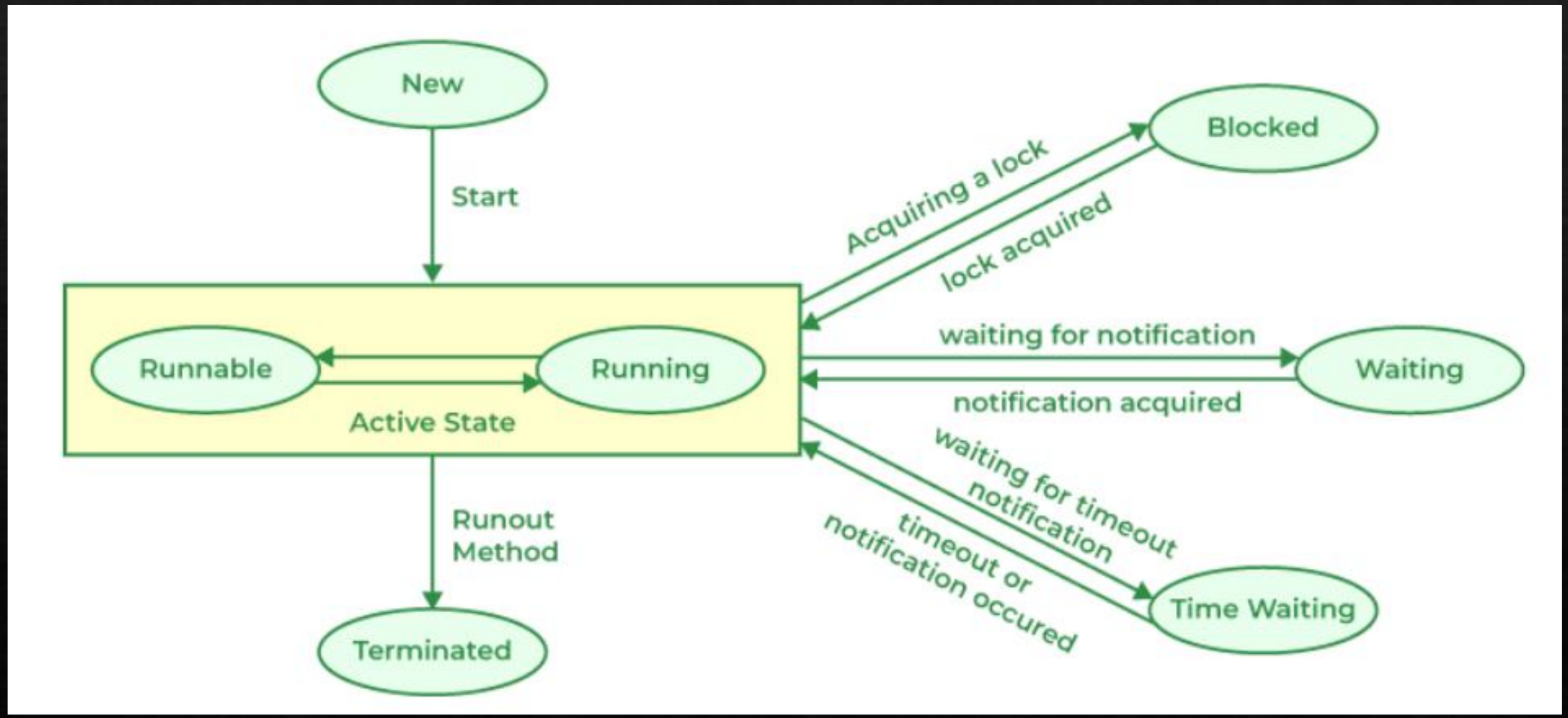
- **Waiting**
- **State:** The thread is waiting indefinitely for another thread to perform a particular action (e.g., notify).
- **How to reach:** The thread enters the Waiting state when wait(), join(), or park() is called.
- **Note:** The thread will remain in this state until another thread notifies it or interrupts it.
- Example:

- synchronized (someObject) {
- someObject.wait(); // t enters Waiting state
- }

- **Timed Waiting**
- **State:** The thread is waiting for another thread to perform an action for a specified waiting time.
- **How to reach:** The thread enters the Timed Waiting state when methods like sleep(long millis), wait(long millis), join(long millis), or parkNanos() are called.
- **Example**
- Thread.sleep(1000); // t enters Timed Waiting state for 1 second

- **Terminated (Dead)**
- **State:** The thread has finished executing, either by completing its execution or being terminated due to an exception.
- **How to reach:** When the run() method completes, the thread enters the Terminated state.
- **Example:**
- public void run() {
- // Thread execution logic
- }
- // After run() completes, the thread enters Terminated state

- **Thread Life Cycle Diagram**
- Here's a simplified flow of the thread states:
- **New** → (start()) → **Runnable**
- **Runnable** → (Running) → **Blocked** (if waiting for a lock) → **Runnable**
- **Runnable** → **Waiting** (if wait(), join(), etc.) → **Runnable**
- **Runnable** → **Timed Waiting** (if sleep(), etc.) → **Runnable**
- **Runnable** → **Terminated** (when execution completes or an exception occurs)


New
Start
Runnable
Running
Active State
Runout
Method
Terminated
Blocked
Acquiring a lock
lock acquired
waiting for notification
notification acquired
Waiting
waiting for timeout
notification
timeout or
notification occured
Time Waiting

- **Key Points**
- **Blocked** state occurs due to waiting for a monitor lock.
- **Waiting** and **Timed Waiting** states occur due to thread synchronization mechanisms like wait(), sleep(), or join().
- A thread in **Runnable** state is not necessarily running; it depends on the JVM thread scheduler.
- **Terminated** state is final, and the thread cannot be restarted.

Multithreading :

- How many way to create /define a Thread?
- 1. By extending Thread class
- 2. By implementing runnable interface

1. By extending Thread class:

Class a extends Thread

```
{
    Public void run()
    {
        // code;
    }
}
```

```
Class b
{
        public static void main(Strin[g] args)
        {
                a t = new a();
                t.start();
                //code;
        }
}
```

- 2. By implementing runnable interface
- Class a extends Runnable

```
{
    Public void run()
    {
        // code;  //Thread job
    }
}
```

```
Class b
{
        public static void main(Strin[g] args)
        {
                a r = new a();
                Thread t = new Thread(r); //ref
                 t.start();
                //code;
        }
}
```

- In Java, there are two main ways to create and run threads: using the Thread class directly or implementing the Runnable interface. Let's look at examples for each method.

- **1. Creating a Thread by Extending the Thread Class**
- When you extend the Thread class, you need to override its run() method to define the code that constitutes the new thread's task.

```java
// Creating a thread by extending the Thread class
class MyThread extends Thread {
    @Override
    public void run() {
        // Code that will run in a new thread
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread thread1 = new MyThread();  // Creating a new thread object
        MyThread thread2 = new MyThread();  // Creating another new thread object

        thread1.start();  // Starting the first thread
        thread2.start();  // Starting the second thread
    }
}
```

- **2. Creating a Thread by Implementing the Runnable Interface**
- When you implement the Runnable interface, you define the thread's task in the run() method, and then you pass an instance of your class to a Thread object.

- **Key Differences Between the Two Methods**
- **Extending Thread Class:**
  - The class that extends Thread cannot extend any other class (since Java does not support multiple inheritance).
  - It's more straightforward but less flexible than implementing Runnable.
- **Implementing Runnable Interface:**
  - Allows the class to implement other interfaces or extend another class, providing more flexibility.
  - Often preferred in practice because it separates the task from the thread mechanism, promoting better object-oriented design.

```java
// Creating a thread by implementing the Runnable interface
class MyRunnable implements Runnable {
    @Override
    public void run() {
        // Code that will run in a new thread
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();  // Creating a Runnable object
        Thread thread1 = new Thread(myRunnable);   // Passing the Runnable object to a new
        Thread thread2 = new Thread(myRunnable);   // Passing the same Runnable object to

        thread1.start();  // Starting the first thread
        thread2.start();  // Starting the second thread
    }
}
```

Stop Thread in java

- A thread is automatically destroyed when the run() method has completed. But it might be required to kill/stop a thread before it has completed its life cycle. Previously,
  methods **suspend**(), **resume**() and **stop**() were used to manage the execution of threads. But these methods were deprecated by Java 2 because they could result in system failures. Modern ways to suspend/stop a thread are by using a boolean flag and Thread.interrupt() method.

Concepts of Daemon Thread

- In Java, a daemon thread is a low-priority thread that runs in the background, providing services to user threads. Here are the key concepts:
- Nature of Daemon Threads:
- **Background Tasks:**
- Daemon threads are designed to perform tasks that support the main application logic, such as garbage collection, finalization, and monitoring.
- **Low Priority:**
- They typically have lower priority than user threads, ensuring that user threads get sufficient CPU time.
- **JVM Termination:**
- The JVM terminates daemon threads automatically when all non-daemon (user) threads have finished execution. Daemon threads don't prevent the JVM from exiting.

- Creating a Daemon Thread:

- setDaemon(true): You can create a daemon thread by setting the daemon property of a thread object to true before starting it.

- **Thread myThread = new Thread(new MyRunnable());**
  myThread.setDaemon(true);
  myThread.start();

- Use Cases:
- Garbage Collection:
- The Java garbage collector is a prime example of a daemon thread that runs in the background, reclaiming memory from unused objects.
- Background Services:
- Tasks like monitoring system resources, logging, or performing periodic cleanup operations are often implemented using daemon threads.

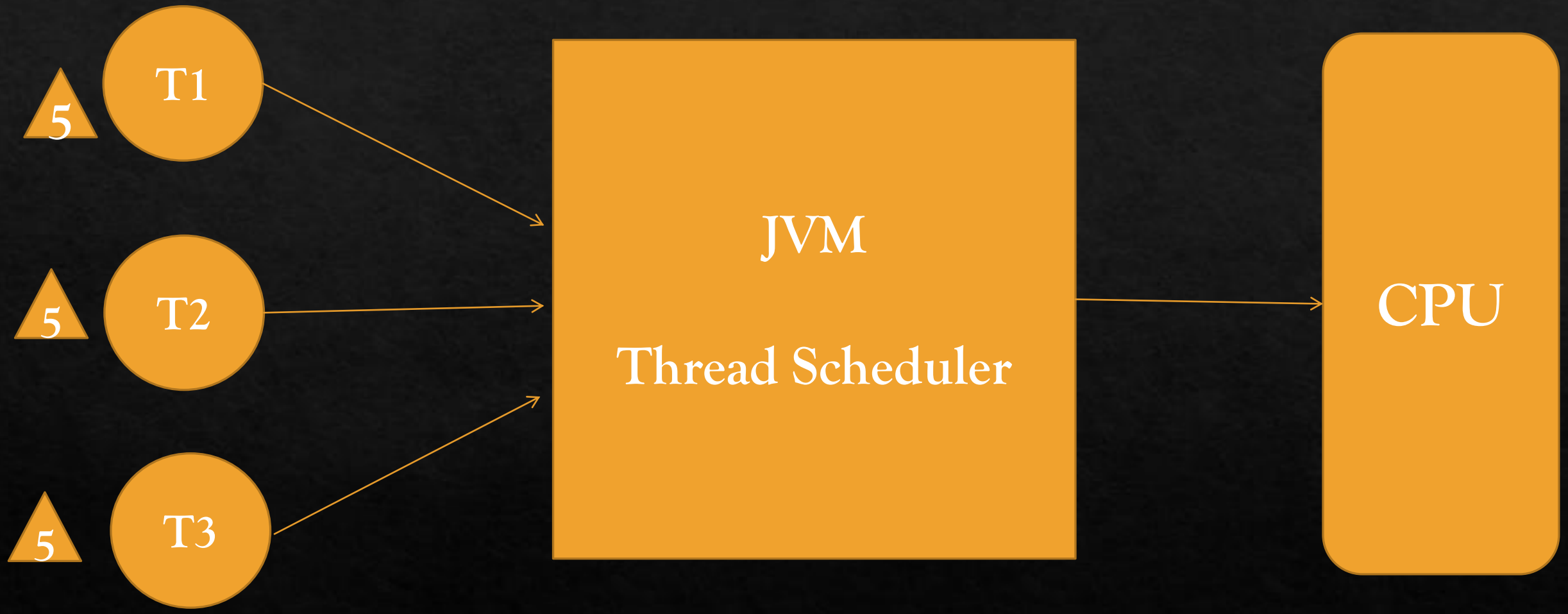
- Important Considerations:
- Termination:
- Don't rely on daemon threads to perform critical tasks that need to complete before the application exits. The JVM can terminate them abruptly.
- Synchronization:
- Use proper synchronization mechanisms (e.g., locks) to avoid race conditions and other concurrency issues when daemon threads interact with shared resources.
- I/O Operations:
- Avoid performing blocking I/O operations in daemon threads, as they might prevent the JVM from exiting.

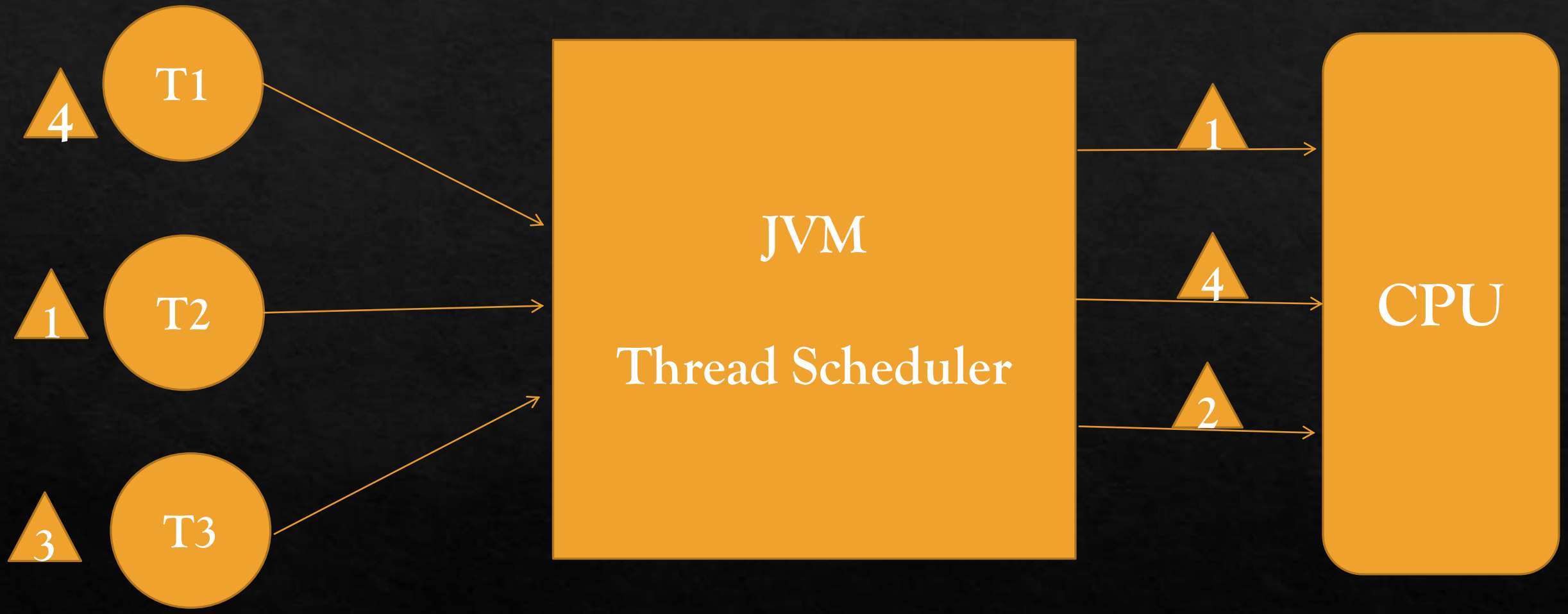
Priority of Thread and Thread scheduling

- In Java, the thread scheduler is a part of the JVM (Java Virtual Machine) responsible for managing the execution of multiple threads. It decides which thread to run next, ensuring fair and efficient utilization of CPU resources.

- Here's how it works:
- **Priority-based Scheduling:**
- Each Java thread has a priority, ranging from Thread.MIN_PRIORITY (1) to Thread.MAX_PRIORITY (10). The scheduler generally selects the thread with the highest priority to run. If multiple threads have the same priority, they are executed in a FIFO (First-In-First-Out) manner.
- **Preemptive Scheduling:**
- Java uses preemptive scheduling, meaning that a thread can be interrupted in the middle of its execution if a higher-priority thread becomes ready to run. This ensures that higher-priority tasks get immediate attention.(short job first)
- **Time-Slicing (Optional):**
- While not strictly guaranteed, some JVM implementations might employ time-slicing, where each thread gets a fixed amount of CPU time before another thread gets a chance. This can further improve responsiveness in multi-threaded environments.(round robin)

- Important Points:
- Thread Priority:
- You can set the priority of a thread using the setPriority() method. However, the actual priority assigned to a thread depends on the underlying operating system and JVM implementation.
- Yielding:
- A thread can voluntarily yield its execution time to other threads of the same priority using the Thread.yield() method.
- Sleeping:
- The Thread.sleep() method pauses a thread's execution for a specified period, allowing other threads to run.


T1
5
T2
5
T3
5
JVM
Thread Scheduler
CPU


T1
4
T2
1
T3
3
JVM
Thread Scheduler
1
4
2
CPU

setPriority() and getPriority() method

- In java it is possible to assign the priority of thread.
- To set these priority Java method class has provided two predefine methods.
- 1) setPriority()
- 2) getPriority()
- The thread class has also provided three pre-defined final static variable and it's value lie between 1 to 10.
- Thread.MIN_PRIORITY → 1
- Thread.NORM_PRIORITY → 5
- Thread.MAX_PRIORITY → 10

- EXAMPLE:
- Class A

```
{
    Public static void main(string[] args)
    {
        A   t1 = new A();
        A   t2 = new A();
        A   t3 = new A();
    }
}
```

What is Synchronous in java

- Synchronous is a technique through which we can control multiple threads or among the num of threads only one thread will enter inside the Synchronous area.
- Note :- The main purpose of synchronous is to overcome the problem of multithreading when multiple threads are trying to access same resource at same time on that situation it may provides some wrong result.
- Synchronous id brodly classified into 2 categories:
- 1. method level Synchronous
- 2. Block level Synchronous

- **1. Method level** Synchronous :
- **In method level** Synchronous the entire method get. Synchronous so, only one thread will enter inside the Synchronous area and remaining all the threads will wait at method level.

- **Example:**
- Synchronized void print table()
- {
  - //code;
- }
- **NOTE:**
- **Every object have a lock in java and this lock can be given to only one thread at all the time.**

2. **Block level** Synchronous :

In block level synch the entire method is not get Synchronous only the part of the method get Synchronous , we have to enclosed those few lines of the code put inside Synchronous block

Example:

```
Public void show ()
{
    synchronized(this)
{
   //code
}
}
```

what is an asynchronous thread in Java?

- In Java, an **asynchronous thread** refers to a thread that operates independently of the main thread and does not block the main thread's execution. It is typically used to perform tasks in the background, allowing the main thread to continue executing other operations without waiting for the background task to complete.
- In Java, there's no such thing as an "asynchronous thread" specifically. However, you can achieve asynchronous behavior using threads in a few ways:
- **1. Creating and Starting Threads:**
- Create a class that implements the Runnable interface or extends the Thread class.
- Implement the run() method, which contains the code you want to execute asynchronously.
- Create an instance of your thread class and start it using the start() method.

- **2. Using the Executor Framework:**
- The Executor framework provides a higher-level abstraction for managing threads.
- You can submit tasks to an executor, which will execute them asynchronously in a thread pool.
- **3. Using CompletableFuture (Java 8 and above):**
- CompletableFuture provides a way to manage asynchronous operations and their results.
- You can chain multiple asynchronous operations and handle their results using callbacks.

- **Runnable:**
- The Runnable interface defines a single method run(), which contains the code to be executed in the thread.
- **Thread:**
- The Thread class represents a thread of execution. You can create a new thread by passing a Runnable object to its constructor.
- **ExecutorService:**
- The ExecutorService interface provides a higher-level abstraction for managing threads. It provides methods to submit tasks for execution and manage the thread pool.
- **CompletableFuture:**
- The CompletableFuture class represents the result of an asynchronous computation. It provides methods to chain asynchronous operations and handle their results.

- In Java, newFixedThreadPool(2) is a method provided by the Executors class that creates a **fixed-size thread pool** with exactly 2 threads. This means that at any given time, the thread pool will have only two threads available to execute tasks, regardless of how many tasks are submitted.
- **Key Uses of newFixedThreadPool(2):**
- **Thread Reuse**: The threads in the pool are reused for executing multiple tasks. Once a thread finishes executing a task, it is returned to the pool and becomes available for another task, reducing the overhead of thread creation and destruction.
- **Task Management**: If more than two tasks are submitted to the pool, the additional tasks will be queued and will only be executed once one of the two threads becomes available.
- **Concurrency Control**: It allows you to limit the number of concurrent tasks. By setting the pool size to 2, you ensure that only two tasks are executed at any time, which is useful for managing resource contention or controlling the load on a system.

- **Behavior of newFixedThreadPool(2) in the Example:**
- Only two tasks will run simultaneously. The remaining tasks will wait in a queue.
- Once a thread completes a task, it will start executing the next queued task.
- The thread pool remains fixed at two threads, which helps limit the system's resource usage.

- In Java, the executor.submit() method is used to submit tasks for execution to an **executor service**. This method is part of the ExecutorService interface, which provides methods to manage thread pools and asynchronous task execution.
- The submit() method allows you to submit a **Callable** or **Runnable** task for execution. Unlike execute() (another method of ExecutorService), submit() returns a **Future** object that represents the result of the task, allowing you to:
- Track the progress of the task.
- Get the result once the task is completed.
- Handle exceptions.

- **Key Points:**
- **Returns a Future**: submit() returns a Future<T> object, where T is the result type of the task. If the task does not return any result (i.e., it's a Runnable), the Future can still be used to check if the task is complete or was cancelled.
- **Non-blocking**: The method is non-blocking, meaning it will return immediately after submitting the task, and the task will execute asynchronously in one of the threads in the thread pool.
- **Usage:**
- **Submitting a Runnable Task**: When you submit a Runnable task, the task does not return any result.

java Copy code

```java
ExecutorService executor = Executors.newFixedThreadPool(2);

executor.submit(() -> {
    // Runnable task
    System.out.println("Task is running");
});

executor.shutdown();
```

- **CompletableFuture.runAsync** is a method in Java that allows you to run a task asynchronously without blocking the current thread. It's part of the CompletableFuture class, introduced in Java 8 under the java.util.concurrent package.
- **Usage**
- The method takes a Runnable and runs it asynchronously. There are two common overloads for runAsync:
- **CompletableFuture.runAsync(Runnable runnable)**: This runs the task asynchronously using the common ForkJoinPool.
- **CompletableFuture.runAsync(Runnable runnable, Executor executor)**: This runs the task asynchronously using a provided Executor.

```java
import java.util.concurrent.CompletableFuture;

public class Example {
    public static void main(String[] args) {
        CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
            System.out.println("Task is running asynchronously in " + Thread.currentThread().getName());
        });

        // Ensure the task has time to finish before the main thread exits
        future.join();
    }
}
```

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

public class ExampleWithExecutor {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
            System.out.println("Task is running asynchronously in " + Thread.currentThread().getName());
        }, executor);

        // Ensure the task has time to finish before the main thread exits
        future.join();

        // Shutdown the executor
        executor.shutdown();
    }
}
```

THE END