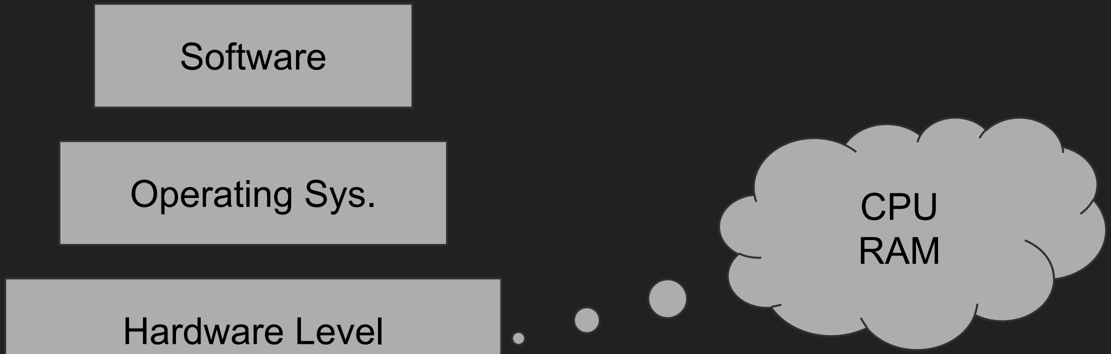
Threads In Java

ITS 2018

By Udara San
Bsc (Hons) In Computer Science in
Software Engineering

How Computer Run Programs,Softwares



Multitasking

Process-Based Multitasking

- In process-based multitasking, the operating system allocates separate memory space and resources for each process.
- Each process has its own address space, file descriptors, and system resources.
- Example : Web Server

Thread-Based Multitasking

- In thread-based multitasking, tasks are executed within the same address space and share the same resources within a process.
- Threads within a process share memory space, file descriptors, and other resources, making communication between threads more efficient compared to processes.
- Example : Chat Application

Thread Based Multitasking

Threads In Java

A thread in Java refers to a lightweight subprocess, a separate path of execution within a program. In simple terms, it's a single sequential flow of control within a program. Threads are used to perform concurrent operations, allowing multiple tasks to be executed simultaneously.

Here are some key points about threads:

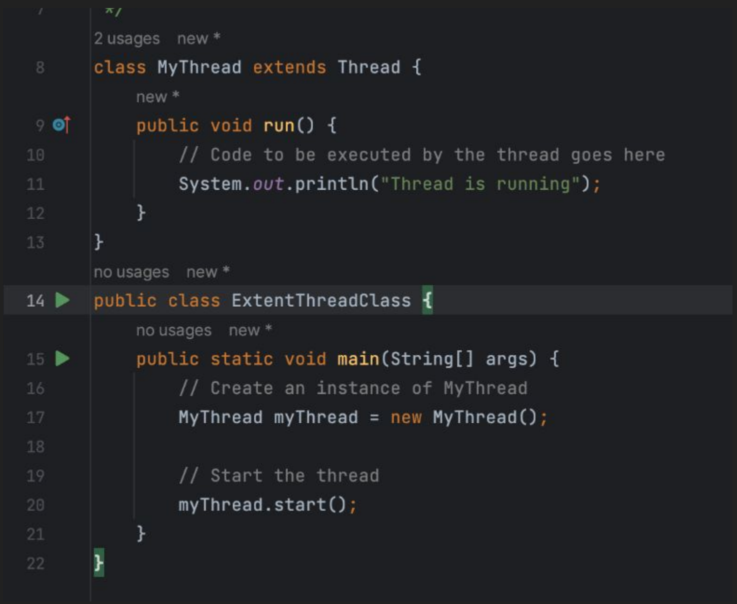
- **Lightweight**: Threads are lightweight compared to processes. They share the same memory space and resources within a process and can be created and managed efficiently.
- **Concurrent Execution**: Threads enable concurrent execution of tasks within a program. Multiple threads can run concurrently, allowing tasks to be executed simultaneously and making efficient use of CPU resources.
- **Independence and Cooperation**: Each thread within a program can execute independently, performing its own tasks. Threads can also cooperate and communicate with each other by sharing data and resources.
- **Multithreading**: Multithreading is the practice of using multiple threads within a program to achieve concurrent execution. It allows applications to perform multiple tasks concurrently, improving responsiveness and performance.

Threads are commonly used in various scenarios, such as:

- GUI applications to handle user interactions and background tasks simultaneously.
- Server applications to handle multiple client requests concurrently.
- Parallel processing to improve performance by dividing tasks among multiple threads.
- Asynchronous programming to perform non-blocking I/O operations and improve responsiveness

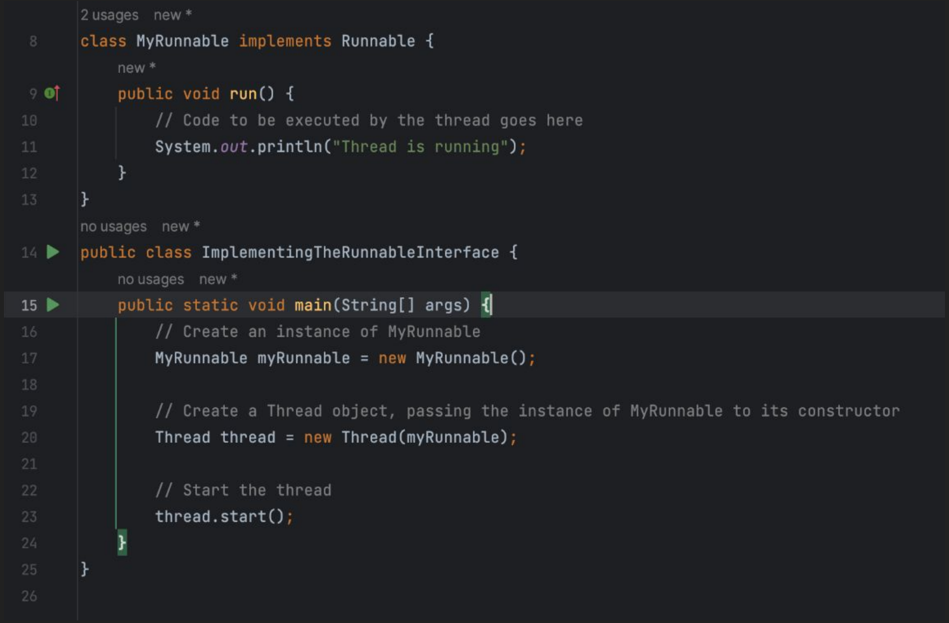
How to Create a Thread In Java

In Java, you can create a thread by extending the `Thread` class or by implementing the `Runnable` interface.



```java
 */
2 usages   new *
class MyThread extends Thread {
    new *
    public void run() {
        // Code to be executed by the thread goes here
        System.out.println("Thread is running");
    }
}
no usages   new *
public class ExtentThreadClass {
    no usages   new *
    public static void main(String[] args) {
        // Create an instance of MyThread
        MyThread myThread = new MyThread();

        // Start the thread
        myThread.start();
    }
}
```



```java
2 usages   new *
class MyRunnable implements Runnable {
    new *
    public void run() {
        // Code to be executed by the thread goes here
        System.out.println("Thread is running");
    }
}
no usages   new *
public class ImplementingTheRunnableInterface {
    no usages   new *
    public static void main(String[] args) {
        // Create an instance of MyRunnable
        MyRunnable myRunnable = new MyRunnable();

        // Create a Thread object, passing the instance of MyRunnable to its constructor
        Thread thread = new Thread(myRunnable);

        // Start the thread
        thread.start();
    }
}
```

Why are there two types?

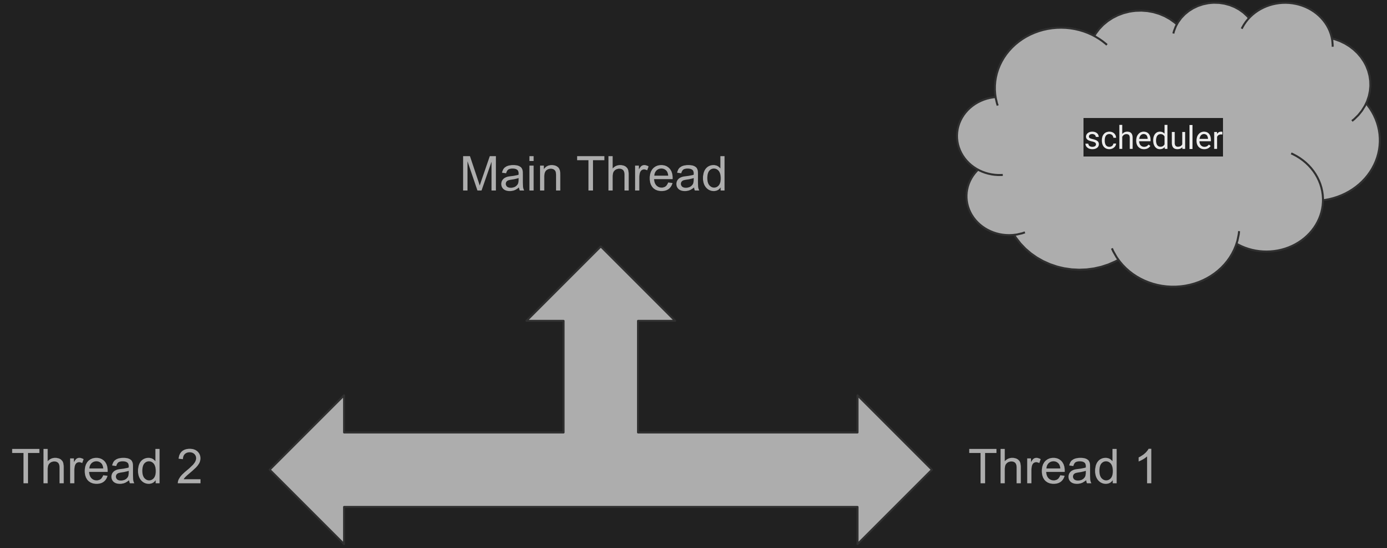
Extending the `Thread` Class:

- This approach is straightforward and easy to understand, especially for beginners. You simply extend the `Thread` class and override its `run()` method to define the task that the thread will execute.
- It's suitable for scenarios where you're creating a thread to encapsulate a specific task, and the thread itself is the primary unit of work.

Implementing the `Runnable` Interface:

- Implementing `Runnable` separates the task (the `run()` method) from the thread itself. This promotes better separation of concerns and is considered a cleaner design approach.
- It allows better reusability of the task logic since the same `Runnable` instance can be used to create multiple threads.
- It supports better object-oriented design principles, as it allows your class to extend other classes if needed (Java doesn't support multiple inheritance, so extending `Thread` class might limit other inheritance options).

Threads In Real World

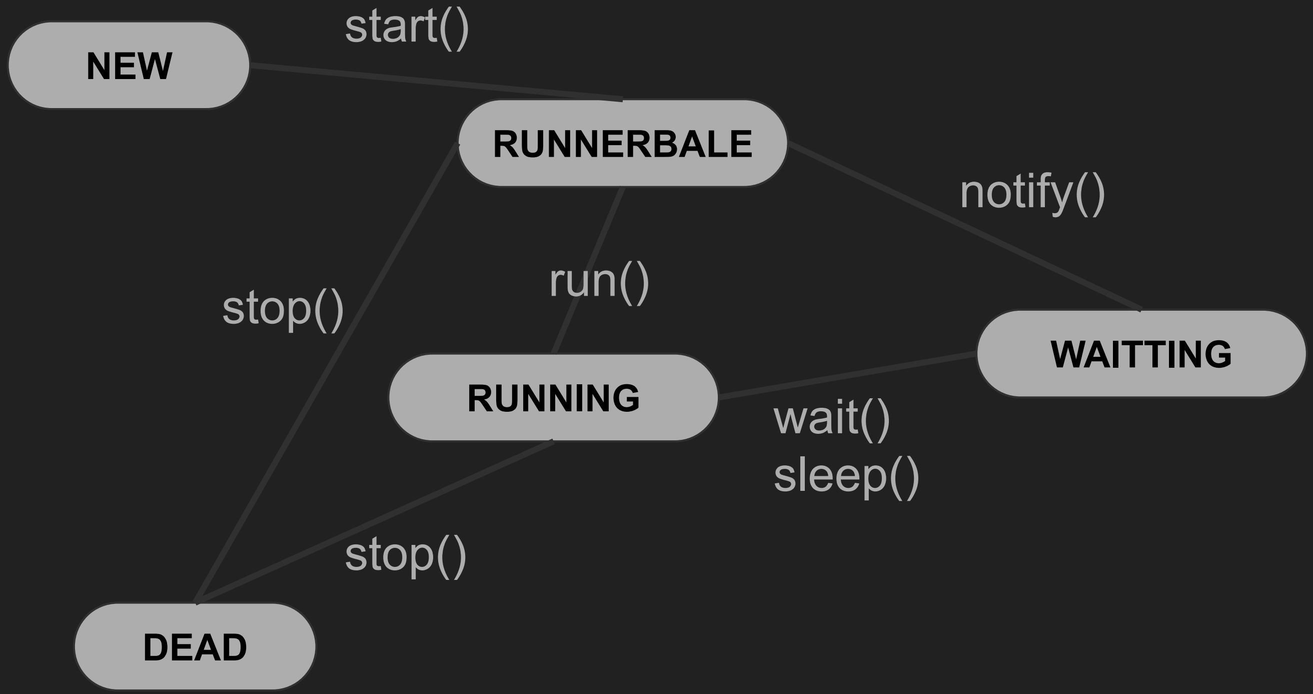


a scheduler is responsible for managing the execution of multiple tasks (such as processes or threads) on a single CPU core. The scheduler decides which task to execute next and for how long, based on certain scheduling policies and algorithms.

Main responsibilities and functions of a scheduler:

- **Task Management**: The scheduler manages the lifecycle of tasks, including task creation, scheduling, suspension, resumption, and termination.
- **CPU Resource Allocation**: The scheduler decides which task should be allocated CPU time next. It determines the order and priority of tasks in the ready queue.
- **Concurrency Control**: The scheduler ensures that multiple tasks can execute concurrently on a single CPU core. It coordinates the execution of tasks to maximize CPU utilization and throughput.
- **Scheduling Policies**: The scheduler implements various scheduling policies and algorithms to determine the order in which tasks are executed. Common scheduling policies include First-Come-First-Served (FCFS), Round Robin, Shortest Job Next (SJN), Priority Scheduling, and Multilevel Queue Scheduling.
- **Context Switching**: The scheduler performs context switching, which involves saving the state of the currently running task and loading the state of the next task to be executed. Context switches incur overhead, so the scheduler aims to minimize them to improve system performance.
- **Fairness and Responsiveness**: The scheduler ensures fairness in resource allocation by giving each task a fair share of CPU time. It also aims to provide responsive performance to interactive tasks such as user interfaces.
- **Adaptability**: Modern schedulers may dynamically adjust scheduling decisions based on system load, task priorities, and other factors to optimize system performance and responsiveness.

Thread States



- **New state** = if we create a new thread
- **start()** - thread goes to the runnable state
- **Runnable State** - when ur thread is exciting and waiting for the scheduler its runnable state
- **run()** - thread goes to the running state
- **Running State** - when ur thread actually running on the CPU its running state
- **sleep() / wait()** - thread goes to the waiting state
- **notify()** - thread goes to the runnable state
- **stop()** - goes to dead state (DEPRECATED) when work end thread automatically going to dead state

- **New Thread**: When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.
- **Runnable State**: A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run. A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.
- **Blocked**: The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.
- **Waiting state**: The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.
- **Timed Waiting**: A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.
- **Terminated State**: 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 a segmentation fault or an unhandled exception.

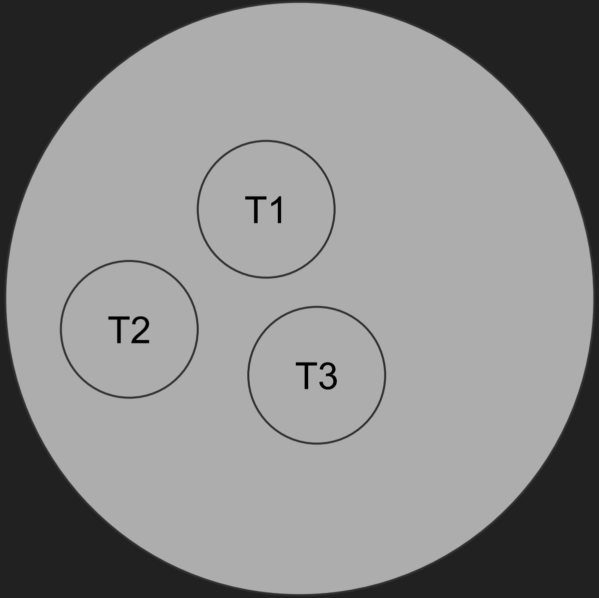
Thread Pool

WHY ?

- ❖ Resource Management
- ❖ Performance

Process ?

- ❖ Thread Pool 50 — 100 (50 Queue)



What is Daemon Thread

- It is a Service Thread.
- Providing Service to Non Daemon Thread.
- Non Daemon Thread - Reading Informations and Display Outputs.
- Daemon Thread - Thread Which are running Behind the Application.In the Background running threads,provide service to non daemon threads.
- JVM will stop all Daemon Threads if there is no running Non Daemon Threads.

Thank You

BYE!