**Multithreading (Java Concurrency)**

**Multitasking:**

* **Task** is a unit of work to be done. **e.g. some logic of a program of a project/application.**
* Running or executing multiple tasks simultaneously is called as **multitasking**.
* It is an ability of an operating system to execute **multiple tasks** at the same time. **The Single CPU switches rapidly between tasks, giving the illusion that they're running simultaneously.**

**Example:**

* ***Imagine you're listening to music using a media player ,while typing in MS Word and downloading a file in the browser. All three are running simultaneously – that's multitasking.***

**Multi-Processing:**

* A process is an independent program in execution.
* Multiprocessing means using **two or more** **CPUs** in a single computer system to perform **multiple tasks/processes** simultaneously**.**
* **It actually executes tasks in parallel, not just switching.**
* It is **faster compared to multi-tasking.**

**Example:**

* If you have a **quad-core processor**, it can run **4 processes** literally at the same time — **one on each core (mini-CPU/component of CPU).**
* **A web server (like Apache)** handling thousands of requests — each core handles a different request.

**Multithreading:**

* Thread is ***a smallest unit of a Process*** which can be referred as a program.
* Executing multiple threads(**programs**) simultaneously is called as **multithreading**.
* ***For each new thread, stack area will be created separately!! (one stack for one thread)***

**Real Time Example:**

* A **thread** is like a worker in a restaurant (**process**).
* All workers share the same kitchen **(memory/resources)** where all the works**(execution)** is done.
* Each *cook/Receptionist/Janitor* (**thread**) handles one **Task** — *cooking, billing, cleaning.*
* Let’s say your computer is a **restaurant**:
* **Multitasking:** One chef switch between preparing pizza, pasta, and salad very fast **(one at a time, but it feels like all at once).**
* **Multiprocessing:** multiple chefs each make a different dish at the same time **(true parallel cooking).**
* **Multithreading:** One chef **(process)** has multiple hands **(threads)** – one chopping, one stirring, one garnishing — all helping the recipe **(Multiple Tasks)** efficiently.

**Creating Threads in Java:**

**1. Using Runnable interface (java.lang.Runnable)**

**2. Using Thread class (java.lang.Thread)**

* **Thread is an implementing class(Child) of Runnable Interface!**
* ***Thread implements Runnable***
* **Runnable** is a **functional interface** i.e it has **only one abstract method run().**

**Methods of Thread Class**

* **run () :**

It is an abstract method present inside a **Runnable interface** and implemented in **Thread Class**.

🡪It contains the **code we want to execute in a thread**.

🡪This method is called by the thread when it executes.

🡪Calling this method will:

* This just **calls the method like a regular function**.
* It does **not** create a **new thread**.
* It runs on the **main thread**, not in parallel.

***i.e main thread will end only after executing these the other Threads executed by run()***

* **start ()** :

It is a non-static method present inside a Thread class.

🡪 Causes this(current) thread to begin execution; the Java Virtual Machine ***internally calls the run() method*** of this(current) thread.

🡪It starts a new thread. It internally calls the **run()** method, but in a **new thread**.

🡪run() runs in a new thread, ***in parallel with the main thread.***

**i.e** main thread and this thread execution are **independent** of each other

* ***All the Threads will run concurrently independent of each other!!***

| **Aspect** | **start()** | **run()** |
| --- | --- | --- |
| Thread creation | Starts a **new thread** | Runs in the **current thread** |
| Execution mode | **Asynchronous** (runs independently) | **Synchronous** (blocks current thread) |
| Thread scheduling | Managed by JVM thread scheduler | No scheduling— runs like a normal method |
| Typical use | To start a new thread | For custom behaviour or direct call (not recommended for threading) |
| Creates a new thread? | ✅ Yes | ❌ No |
| Runs run() method? | ✅ Yes (internally) | ✅ Yes |
| Runs concurrently? | ✅ Yes | ❌ No |
| Used for multithreading? | ✅ Yes | ❌ No |
| Performance | **Faster** | **Slower** |

* ***Each new thread will have a dedicated stack memory, where each and every thread will be executed independently.***
* **Thread.*activeCount*()**

➜ Returns the number of active threads in the current thread's thread group (including the current thread).

🡪A thread is considered **active/live** after **start ()** is called and until it terminates (i.e., completes execution or is stopped).

* **Thread.currentThread()**

🡪 Returns the ***Thread object*** for the current thread

e.g. **Thread[#22,Negative Number,5,main]**

* #22: ***Thread id***
* *Negative Number:* ***Thread Name***
* *5:* ***Priority***
* *main :* ***Thread group***
* **Thread.sleep()**

🡪 **pauses the execution of the current thread** for a specified duration (in **milliseconds**)

* **public final boolean isAlive()**

**🡪** non static method of Thread class, used to check whether a particular thread is alive or not!!

* **public final void join()**

**🡪** non static method of Thread class, used to make main thread waits for current thread to terminate.

* **Thread.yeild()**

**🡪** pauses the current thread temporarily and gives a chance for other threads of the same or higher priority to execute.

But it doesn’t guarantee that the current thread will stop — it’s just a hint/request to the Thread scheduler.

**Properties of a thread:**

* **Id :**
* The id of a thread is given by the **JVM**. We can fetch this property using a method called as **threadId(),**
* **threadId()** is non-static method present inside a Thread class. ***(getId() method is deprecated)***
* **Name :**
* Name property is modifiable i.e. we can change the name of a thread using **setName(String name).**
* We can fetch the name of the thread using **getName().**
* **Priority :**
* A newly created thread will have the priority as **5 by default**, the **minimum priority can be 1** and **maximum priority can be 10**.
* We can set the priority value using **setPriority(int priority)**, we can fetch the priority value using **getPriority().**

**Note:**

* Thread priorities are **hints** to the thread scheduler, not guarantees.
* On some systems (like Windows/Linux), the JVM might **ignore priorities** or treat them differently based on the OS scheduling policy.
* It cannot guarantee the preferences but may increase the chance.

**Thread life Cycle :**

* A programmer cannot control a thread life cycle, it is completely dependent on **thread schedular.**
* Following Factors which will be taken into consideration by the thread schedular to manage thread life cycle.

**1.Sometimes thread schedular follows first come first serve principle.**

**2.The priority of a thread.**

**3.System configuration ( Operating System, Processor )**

* The **Thread Scheduler** is a part of the **Java Virtual Machine (JVM)** that **decides which thread to run next** out of multiple threads.
* **Thread Scheduler is like a traffic controller** — it manages the flow of multiple threads trying to run on the **CPU** and picks one based on certain **scheduling policies**.
* **Thread schedular** will allocate the CPU to a thread for its execution for specific interval of time (not always).

A diagram of a process

AI-generated content may be incorrect.

**1. New Thread (New State)**

* A thread object is created using new Thread(), but **it hasn’t started yet**.
* **Transition:**
  + Call **start()** to move to the **Runnable** state.

**🔹 2. Runnable State**

* The thread is **ready to run** and is waiting for the **Thread Scheduler** to pick it up. It’s not running yet !.
* **Transition:**
  + **start()** → Runnable
  + **yield() or run()** → remains in Runnable

**🔹 3. Running State**

* The thread is **actively executing code** in its **run()** method.
* **How it gets here:**
  + Thread Scheduler picks it from the **Runnable queue**.
* **Transitions from Running:**
  + **yield()** → goes back to Runnable (gives chance to other threads)
  + **wait(), join(), park()** → Waiting
  + **sleep(ms), wait(ms), join(ms)** → Timed Waiting
  + **synchronized** block not available → Blocked

**🔹 4. Blocked State**

* The thread is **waiting to enter a synchronized block/method(Resource)** locked by another thread.
* **How it gets here:**
  + Tries to enter **synchronized** code that's already **locked**.
* **How it exits:**
  + When the lock is released, it returns to **Runnable**.
* **Special Case:**
  + May lead to **Deadlock** if threads are stuck waiting on each other forever.

**🔹 5. Waiting State**

* The thread is waiting **indefinitely** for another thread to perform some action like **notify() or join()** or waiting for **Thread Schedular** to pick them up again for execution.
* **How it gets here:**
  + **wait(), join(), park()**
* **How it exits:**
  + Another thread calls **notify(), notifyAll(), or unpark()** on it.
* **Note:**  
  No timeout here — the thread waits forever unless notified.

**🔹 6. Timed Waiting State**

* The thread is waiting for a **specific time** (not indefinitely).
* **How it gets here:**
  + **sleep(ms)**
  + **wait(ms)**
  + **join(ms)**
  + **parkNanos(), parkUntil()**
* **How it exits:**
  + Time runs out **or**
  + It is **notified/unparked** early

**🔹 7. Terminated (Dead) State**

* The thread has **completed execution** or **terminated abnormally**.
* **How it gets here:**
  + **run()** completes successfully → **Completed Execution**
  + An **exception** or **forced stop** → **Abrupt Termination**

**🔹 Abrupt Termination**

* Thread is killed or stopped due to **some problem or** an **unhandled exception**, or external **interrupt/kill**.(using **sleep()** for long time, interrupt(), stop())
* **Leads to:**  
  Terminated state.

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.No** | **State** | **Trigger/Method** | **Description** |
| 1️⃣ | New | new Thread() | Thread object created |
| 2️⃣ | Runnable | .start() | Ready to run |
| 3️⃣ | Running | Scheduler picks it | Now executing run() |
| 4️⃣ | Blocked | Enters locked synchronized block | Waits for lock release |
| 5️⃣ | Waiting | wait(), join(), park() | Waits until notified/picked again by Thread Schedular |
| 6️⃣ | Timed Waiting | sleep(), wait(ms), join(ms) | Waits for limited time |
| 7️⃣ | Terminated | run() ends or crash | Thread dies |
| 🔄 | Runnable ↔ Running | yield() | Ready to running and vice versa |
| 🚨 | Any → Terminated | Uncaught exception or kill signal | Abrupt stop due to some problem |

* Imagine a food delivery app like Swiggy:
* 🧾 Order created → New
* 🚴 Delivery person assigned → Runnable
* 🏍️ Out for delivery → Running
* 🚦 Stuck at red signal → Blocked
* ☕ Waiting at restaurant → Waiting
* ⏲️ Waiting for OTP → Timed Waiting
* ✅ Order delivered or canceled → Terminated

**Synchronization in Multithreading:**

* When multiple threads try to access a **shared resource(single resource)** simultaneously being unaware of each other’s operation on the shared resource, leads to **data inconsistency or undesired output.**
* This is because multiple threads will try to access a **single resource,** resulting in **race conditions, both executing together without waiting for each other!!!**
* The above problem in multithreading can be overcome using **synchronization** concept.

**Synchronization:**

* It is the process using which we can achieve **data consistency**(**accuracy**) in multithreaded environment. It can be achieved using **synchronized keyword**.
* **“synchronized” keyword** can be used only with the blocks.
* Whenever a thread gets an access to a **synchronized shared resource** then **the lock will be applied on that resource** so that no other thread can get access to same synchronized shared resource.
* This lock will not be released until and unless the currently executing thread (which has the access on the synchronized shared resource )completes its execution.

**There are two types of locks :**

**1. Class lock :** If the synchronized shared resource is **static** in nature, then the lock applied on it is called as **class lock.**

**2. Object lock :** If the synchronized shared resource is **non-static** in nature then lock applied on it is called as **object lock.**

**Method used for synchronization**

* **wait() :** It is a non-static method present inside Object class. Which is used to pause the execution of currently executing thread. This method must be used inside synchronized block.
* **notify() :** It is a non-static method present inside Object class. Which is used to resume the execution of paused thread on the same shared resource.
* **notifyAll() :** It is a non-static method present inside Object class. This method is used to resume the execution of multiple threads whose execution has been paused on the same shared resource.

**Deadlock in multithreading:**

A diagram of a process

AI-generated content may be incorrect.

* In the given diagram, we are dealing with **two threads** and **two shared resources**, both of which are synchronized. Let’s denote them as:
* Threads: T1 and T2
* Resources: R1 and R2

**Step 1: Thread T1 locks Resource R1**

* Thread T1 starts its execution.
* It successfully acquires a lock on Resource R1.
* This means ***no other thread can access or lock R1 until T1 releases it***.

**Step 2: Thread T2 locks Resource R2**

* At the same time, Thread T2 also starts executing.
* It successfully acquires a lock on Resource R2.
* Now, R2 is locked by T2, and ***any other thread (including T1) trying to access R2 will have to wait.***

**Step 3: Thread T1 tries to lock Resource R2**

* After locking R1, Thread T1 attempts to lock R2.
* However, R2 is already locked by Thread T2.
* So, T1 is **put on hold** — it will wait until R2 is released.

**Step 4: Thread T2 tries to lock Resource R1**

* Similarly, after locking R2, Thread T2 attempts to acquire a lock on R1.
* But R1 is currently held by Thread T1.
* As a result, T2 is **also put on hold**, waiting for R1 to be released.

**Step 5: The Deadlock**

* Now, both threads are **stuck**:
  + T1 is waiting for T2 to release R2.
  + T2 is waiting for T1 to release R1.
* **Neither thread can proceed** because they are each holding one resource and waiting for the other.
* This results in a **deadlock**, where both threads are **indefinitely waiting** for each other to release the required locks!!!.

**Daemon Threads :**

* Daemon Threads are **low priority threads** which will be started by the **JVM** itself after regular interval of time.
* A programmer cannot control the execution of daemon threads. Along with that a programmer cannot see the execution of daemon threads.
* Daemon threads will be executed in the background during the run time of an application.
* We have some inbuilt Daemon Threads as well as a user can also define customized Daemon threads.
* *The popular inbuilt daemon thread in java is* ***Garbage Collector.***
* ***Garbage collector*** *is a program* ***(Daemon Thread)*** *which will be executed over a regular interval of time by the JVM to search for* ***garbage (de-referred objects)*** *inside* ***the heap memory*** *and removing them from the heap memory* ***which leads to efficient utilization of the heap space.***