Multithreading in Java

What is a concurrent programming language?

A **concurrent programming language** is defined as one which uses the concept of simultaneously executing processes or threads of execution as a means of structuring a program. A **parallel language** is able to express programs that are executable on more than one processor.

In concurrent programming, there are two basic units of execution: *processes* and *threads*. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.

A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called time slicing.

**Multithreading in java** is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

### **Advantages of Java Multithreading**

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

## **Multitasking**

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

* Process-based Multitasking (Multiprocessing)
* Thread-based Multitasking (Multithreading)

### **Process-based Multitasking (Multiprocessing)**

* Each process has an address in memory. In other words, each process allocates a separate memory area.
* A process is heavyweight.
* Cost of communication between the process is high.
* Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

### **2) Thread-based Multitasking (Multithreading)**

* Threads share the same address space.
* A thread is lightweight.
* Cost of communication between the thread is low.

#### **Note: At least one process is required for each thread.**

## **What is Thread in java**

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

#### **Note: At a time one thread is executed only.**

# Multithreading in Java

1. [Multithreading](https://www.javatpoint.com/multithreading-in-java)
2. [Multitasking](https://www.javatpoint.com/multithreading-in-java#multitasing)
3. [Process-based multitasking](https://www.javatpoint.com/multithreading-in-java#multiprocessing)
4. [Thread-based multitasking](https://www.javatpoint.com/multithreading-in-java#multithreading)
5. [What is Thread](https://www.javatpoint.com/multithreading-in-java#thread)

**Multithreading in java** is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

### **Advantages of Java Multithreading**

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

**Note: At least one process is required for each thread.**



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

#### **Note: At a time one thread is executed only.**

## **Java Thread class**

Java provides **Thread class** to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

# Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated

|  |
| --- |
| **1) New** The thread is in new state if you create an instance of Thread class but before the invocation of start() method. |

### **2) Runnable**

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

### **3) Running**

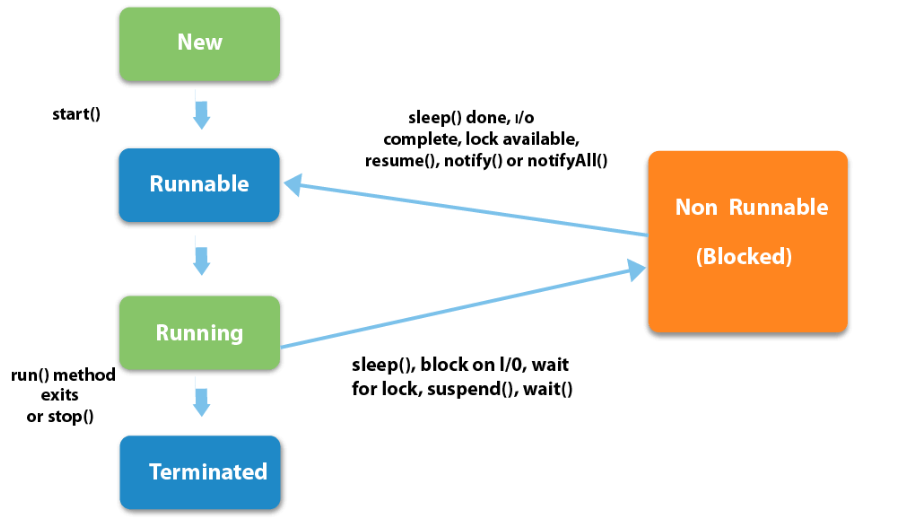
The thread is in running state if the thread scheduler has selected it.

### **4) Non-Runnable (Blocked)**

This is the state when the thread is still alive, but is currently not eligible to run.

### **5) Terminated**

A thread is in terminated or dead state when its run() method exits.



|  |  |  |
| --- | --- | --- |
| How to create thread :There are two ways to create a thread:  1. By extending Thread class 2. By implementing Runnable interface.  **Thread class:**  |  | | --- | | Thread class provide constructors and methods to create and perform operations on a  thread.Thread class extends Object class and implements Runnable interface. |  **Commonly used Constructors of Thread class:**  |  | | --- | | * Thread() * Thread(String name) * Thread(Runnable r) * Thread(Runnable r,String name) | |

### **Commonly used methods of Thread class:**

|  |
| --- |
|  |

### **Runnable interface:**

|  |
| --- |
| The Runnable interface should be implemented by any class whose instances are  intended to be executed by a thread. Runnable interface have only one method named run(). |

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. |

### **Starting a thread:**

|  |
| --- |
| **start() method** of Thread class is used to start a newly created thread. It performs following tasks:   * A new thread starts(with new callstack). * The thread moves from New state to the Runnable state. * When the thread gets a chance to execute, its target run() method will run. |

### **Some Important points to Remember**

1. When we extend Thread class, we cannot override **setName()** and **getName()** functions, because they are declared final in Thread class.
2. While using **sleep()**, always handle the exception it throws.

*static* void **sleep**(long *milliseconds*) throws **InterruptedException**

# Thread Scheduler in Java

**Thread scheduler** in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

### **Difference between preemptive scheduling and time slicing**

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

# Daemon thread in Java with example

Daemon thread is a low priority thread (in context of JVM) that runs in background to perform tasks such as garbage collection (gc) etc., they do not prevent the JVM from exiting (even if the daemon thread itself is running) when all the user threads (non-daemon threads) finish their execution. JVM terminates itself when all user threads (non-daemon threads) finish their execution, JVM does not care whether Daemon thread is running or not, if JVM finds running daemon thread (upon completion of user threads), it terminates the thread and after that shutdown itself.

## Properties of Daemon threads: