**What is Thread in Java**

A **thread in Java** simply represents a single independent path of execution of a group of statements. It is the flow of execution, from beginning to end, of a task.

When we write a group of statements in a program, these statements are executed by JVM one by one. This execution process is called thread in Java.

There is always at least one thread running internally in every program and this thread is used by JVM to execute statements in the program.

When a program contains a single flow of control, it is called single-threaded program. In a single thread program, there is a beginning, a body, and an end, and execute commands sequentially.

We can also create more than one execution thread in a program that can be used to perform multiple tasks simultaneously.

When we create more than one thread in a program, each thread has its own path of execution and all the threads share the same memory address space, data, and code in a program.

## What is Process in Java?

Thread in Java is the smallest unit of executable code in a program. It helps to divide a program into multiple parts to speed up the process. A process is a program that executes as a single thread. In other words, when an executable program is loaded into memory, it is called process.

**Points**

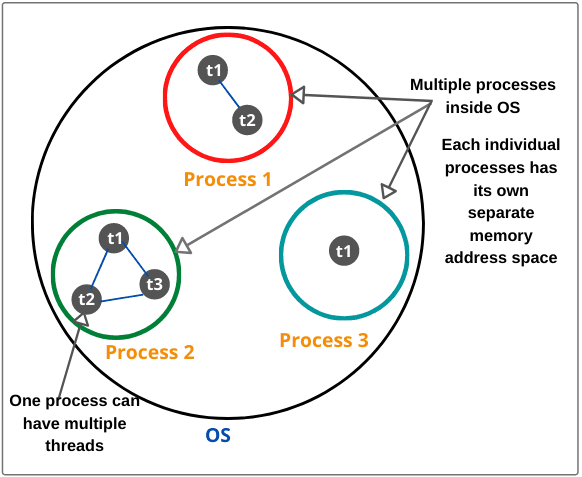
1. Every individual process has its own separate memory address space and can execute a different program.

2. Each process can have more than one thread.

3. Each process communicates through the operating system, files, and network.

When we will create a new thread in a program, it shares the same memory address space with other threads in a program whereas every individual process has its own separate memory address space.

Therefore, creating a thread takes fewer resources than creating a new process.



## Why Java threads are lightweight process?

Java Threads are also known as lightweight threads or light-weight processes. It means that they can be executed in the same memory space because all the threads in the main application program share the same address space in the memory so that they can easily communicate among themselves. Thus, they also take less space in memory and less processor time.

## Main Thread in Java

Every Java program has always at least one thread, even if you do not create any thread. This thread is called main thread. The main thread is also called parent thread and the rest of threads that are generated from it are called child threads of the program.

Main thread is the last thread to be executed in a program. When main thread finishes the execution, the program terminates immediately. Whenever Java program starts, main thread is created automatically.

This main thread is available in all programs. We can control the execution of the main thread by creating a Thread object and then using methods of Thread class.

To do so, we will have to create a Thread object by calling currentThread() method of class Thread. A Thread object can be created as follows:

Thread obj = Thread.currentThread();

Since currentThread() method of class Thread is a public static method, therefore, we can call it using class name.

The currentThread() method returns a reference of current thread on which it is called. “obj” is a reference variable that is assigned to store the return value of currentThread() method.

public class MainThread

{

public static void main(String[] args)

{

// Create a Thread object by calling currentThread() method of class Thread.

Thread obj = Thread.currentThread();

System.out.println("Current thread is " +obj);

System.out.println("Name of current thread is " +obj.getName());

}

}

**Explanation:**

1. In this program, currentThread() is a static method in a class Thread. Therefore, we called it as Thread.currentThread(). This method returns the reference of main thread because this is called inside the main thread. The reference of main thread will be stored in a variable obj of type Thread.

2. When line 8 will be executed by JVM, it will display an output “Thread[main,5,main]” on screen. In the square bracket, the first value, main represents the name of thread; second value 5 represents the priority of main thread.

Every thread will have a priority number associated with it that can be range from 1 to 10. 1 is the minimum priority and 10 is the maximum priority of a thread.

The third value main represents the name of group to which main thread belongs.

3. getName() method of Thread class returns the name of thread that is referred by object obj.

Let’s make a program where we will control main thread and also change name of main thread.

public class MainThreadDemo

{

public static void main(String[] args)

{

// Create a Thread object by calling currentThread() method of class Thread.

Thread obj = Thread.currentThread();

System.out.println("Current thread is " +obj);

System.out.println("Name of current thread is " +obj.getName());

obj.setName("New Thread"); // Changing name of main thread.

System.out.println("Name of current thread after changing name is " +obj);

System.out.println("Main thread existing");

}

}

In the preceding program, the name of main thread is changed by calling setName() method of Thread class and a new name of main thread is displayed. The new name of main thread is New Thread

## Use of Thread in Java

Threads can be used for multiple purposes. Some advantages of using threads are as follows:

1. Threads are mainly used in server-side programs where we need to handle multiple clients on network or internet simultaneously.

2. Another important use of threads is in creating games and animations. For example, threads can be used to show picture in motion. In many games, threads help to perform more than one task simultaneously.

For example, in fighter game, a fighter plane may be from left to right. A machine gun in fighter plane continuously shoots enemy by releasing bullets.

Here, two tasks are happening simultaneously. One thread is moving the fighter plane while another thread releasing bullets simultaneously.

3. Generally, threads can be used to perform more than one task simultaneously.

A program running on a single thread can cause problems when we want to perform two or more tasks simultaneously. For example, when you play online cricket game, you sometimes see such a situation where game does not show the updated score but still displays graphics properly.

Here, displaying graphics of game and updating scores are two different jobs that are to be handled at the same time. Therefore, it is important that the game programming must be able to handle these multiple tasks to make game fast.

To overcome these problems, Java supports multithreaded programming that enables a single program to perform multiple tasks simultaneously.

In multithreaded programming, each thread is assigned to perform a single task and executes independently. Multiple threads in a program share the same memory address space among themselves.

Hope that this tutorial has covered all the important points related to what is thread in java, main thread, and use of thread. I hope that you will have understood the basics of thread.

# Java Multithreading

We know that Java is a multithreaded programming language which means we can develop a multithreaded program using Java.

Multithreading programming approach is one of the most powerful features of Java. It makes our program more responsive and interactive as well as improves performance.

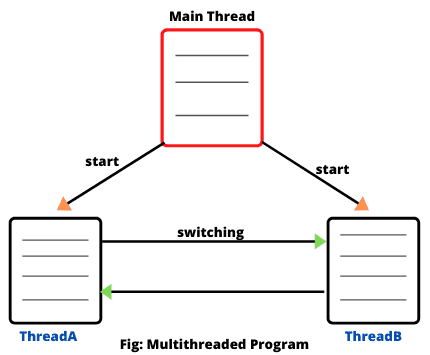
To grasp the concept of multithreading, one has to understand the meaning of multithreading.

## What is Multithreading in Java

Multithreading means multiple threads of execution concurrently. The process of executing multiple threads simultaneously (concurrently) is called **multithreading in Java**.

In other words, multithreading is a technique or programming concept in which a program (process) is divided into two or more subprograms (subprocesses), each of which can perform different tasks simultaneously (at the same time and in parallel manner). Each subprogram of a program is called thread in Java.

Look at the below figure where a Java program has three threads, one main and two others. The other two threads ThreadA and ThreadB are created and started from the main thread. When a program contains multiple flows of control, it is called multithreaded program.



Once initiated by the main thread, thread ThreadA and ThreadB run simultaneously and share the resources together.

When a program contains more than one thread, the CPU can switch between two threads to execute them at the same time. The switching between two threads is known as **context switch**.

The switching from one thread to another thread occurs so fast that it appears to the users that all threads are being executed at the same time. But in reality, only one thread is executed at a time.

This technique is useful for those applications which need multiple tasks to be done simultaneously. In a single processor system, multiple threads share CPU time that is known as **time-sharing**.

The operating system is responsible for allocating and scheduling resources to them. Thus, multithreading improves the performance of CPU by maximum utilization and keeping the idle time of CPU to minimum.

In a multithreaded program, each thread is assigned a single task to perform and executes independently. If an exception occurs in one thread, it does not affect other threads during the execution.

For example, one thread read data, another thread process it, and third thread write it, thus improves the overall performance of an application.

We can write a multithreaded program to display animations, to play music, and download a file from the network at the same time.

## Advantage of Multithreading in Java

1. In a multithreaded application program, different parts of the application are executed by different threads. The entire application does not stop even if an exception occurs in any of the threads. It does not affect other threads during the execution of the application.

2. Different threads are allotted to different processors and each thread is executed in different processors in parallel.

3. Multithreading helps to reduce computation time.  
4. Multithreading technique improves the performance of the application.  
5. Threads share the same memory address space. Hence, it saves memory.  
6. Multithreaded program makes maximum utilization of CPU and keeping the idle time of CPU to minimum.  
7. Context switching from one thread to another thread is less expensive than between processes.

### Drawbacks of Multithreading in Java

1. Increased complexity.  
2. Synchronization of shared resources.  
3. In the multithreading programming concept, debugging is difficult. At times, result is unpredictable.  
4. Potential deadlocks.  
5. Programming complications may occur.

## Multitasking in Java

The process of executing one or more tasks concurrently or at the same time is called **multitasking**. It is the ability of an operating system to execute multiple tasks at once. The main purpose of multitasking is to use the idle time of CPU.

Multitasking can be implemented in two ways:

1. Process-based multitasking (Multiprocessing)  
2. Thread-based multitasking (Multithreading)

## Process-based Multitasking (Multiprocessing)

The process of executing multiple programs or processes at the same time (concurrently) is called process-based multitasking or program-based multitasking. In process-based multitasking, several programs are executed at the same time by the microprocessor.

Therefore, it is also called multiprocessing in Java. It is a heavyweight. A process-based multitasking feature allows to execute two or more programs concurrently on the computer.

A good example is, running spreadsheet program while also working with word-processor.

Each program (process) has its own address space in the memory. In other words, each program is allocated in a separate memory area.

The operating system requires some CPU time to switch from one program to another program. The switching of CPU among programs is called context switching.

The switching from one program to another program is so fast that it appears to the user that multiple tasks are being done at the same time.

## Thread-based Multitasking (Multithreading)

A thread is a separate path of execution of code for each task within a program. In thread-based multitasking, a program uses multiple threads to perform one or more tasks at the same time by a processor.

That is, thread-based multitasking feature allows you to execute several parts of the same program at once. Each thread has a different path of execution.

## Advantage of Thread-based Multitasking over Process-Thread Multitasking

The main advantages of thread-based multitasking as compared to process-based tasking are

1. Threads share the same memory address space.

2. Context switching from one thread to another thread is less expensive than between processes.

3. The cost of communication between threads is relatively low.

4. Threads are lightweight as compared to processes (heavyweight). They utilize the minimum resources of the system. They take less memory and less processor time.

Java supports thread-based multitasking and provides a high quality of facilities for multithreading programming.

# Thread Class in Java | Thread Methods in Java

Thread class contains several constructors for creating threads for tasks and methods for controlling threads. It is a predefined class declared in java.lang default package.

Each thread  is created and controlled by a unique object of the Thread class. An object of thread controls a thread running under JVM.

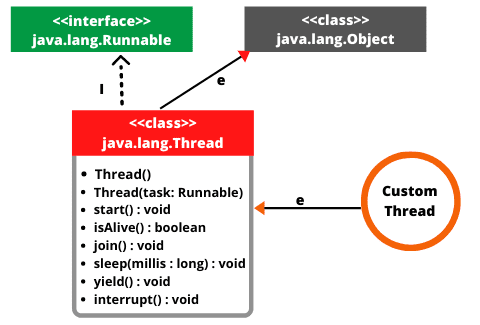
Thread class contains various methods that can be used to start, control, interrupt the execution of a thread, and for many other thread related activities in a program.

Thread class extends Object class and it implements Runnable interface. The declaration of thread class is as follows:

public class Thread

extends Object

implements Runnable



## Thread Class Constructor

The various constructors of thread class are defined in java.lang package that can be used to create an object of thread are as follows:

1. **Thread():** This is a basic and default constructor without parameters. It simply creates an object of Thread class.

2. **Thread(String name):** It creates a new thread object with specified name to a thread.

3. **Thread(Runnable r):** It creates a thread object by passing a parameter r as a reference to an object of the class that implements Runnable interface.

4. **Thread(Runnable r, String name):** This constructor creates a thread object by passing two arguments r and name. Here, variable r is a reference of an object of class that implements Runnable interface.

## Methods of Thread Class in Java

Thread class provides various static methods that are as follows:

1. **currentThread():** The currentThread() returns the reference of currently executing thread. Since this is a static method, so we can call it directly using the class name. The general syntax for currentThread() is as follows:

**Syntax:**

public static Thread currentThread()

2. **sleep():** The sleep() method puts currently executing thread to sleep for specified number of milliseconds. This method is used to pause the current thread for specified amount of time in milliseconds.

Since this method is static, so we can access it through Thread class name. The general syntax of this method is as follows:

**Syntax:**

public static void sleep(long milliseconds) throws InterruptedException

The general syntax for overloaded version of sleep method is as follows:

**Syntax:**

public static void sleep(long milliseconds, int nanoseconds ) throw InterruptedException

The overloaded version of sleep() method is used to pause specified period of time in milliseconds and nanoseconds. Both methods throw InterruptedException and must be used within Java try-catch block.

3. **yield():** The yield() method pauses the execution of current thread and allows another thread of equal or higher priority that are waiting to execute. Currently executing thread give up the control of the CPU.  The general form of yield() method is as follows:

**Syntax:**

public static void yield()

1. start(): The start() method is used to start the execution of a thread by calling its run() method. JVM calls the run() method on the thread. The general syntax for start() method is as follows:

Syntax:

public void start()

2. run(): The run() method moves the thread into running state. It is executed only after the start() method has been executed. The general syntax is as follows:

Syntax:

public void run()

3. getName(): This method returns the name of the thread. The return type of this method is String. The general form of this method is:

Syntax:

public final String getName()

4. setName(): This method is used to set the name of a thread. It takes an argument of type String. It returns nothing.

Syntax:

public final void setName(String name)

5. getPriority(): This method returns the priority of a thread. It returns priority in the form of an integer value ranging from 1 to 10. The maximum priority is 10, the minimum priority is 1, and normal priority is 5.

Syntax:

public final int getPriority() // Return type is an integer.

6. setPriority(): This method is used to set the priority of a thread. It accepts an integer value as an argument. The general syntax is given below:

Syntax:

public final void setPriority(int newPriority)

7. isAlive(): This method is used to check the thread is alive or not. It returns a boolean value (true or false) that indicates thread is running or not. The isAlive() method is final and native. The general syntax for isAlive() method is as follows:

Syntax:

public final native boolean isAlive()

8. join(): The join() method is used to make a thread wait for another thread to terminate its process. The general syntax is

Syntax:

public final void join() throw InterruptedException

This method throws InterruptedException and must be used within a try-catch block.

9. stop(): This method is used to stop the thread. The general form for this method is as follows:

Syntax:

public final void stop()

This method neither accepts anything nor returns anything.

10. suspend(): The suspend() method is used to suspend or pause a thread.

Syntax:

public final void suspend()

11. resume(): This method is used to resume the suspended thread. It neither accepts anything nor returns anything.

Syntax:

public final void resume()

# How to create Thread

**Creating threads in Java** | We know that every Java program has at least one thread called main thread. When a program starts, main thread starts running immediately.

Apart from this main thread, we can also create our own threads in a program that is called child thread. Every child threads create from its main thread known as parent thread.

There are two ways to create a new thread in Java. They are as follows:

1. One is by extending java.lang.Thread class  
2. Another is by implementing java.lang.Runnable interface

**Extending Thread Class in Java**

Extending Thread class is the easiest way to create a new thread in Java. The following steps can be followed to create your own thread in Java.

1. Create a class that extends the Thread class. In order to extend a thread, we will use a keyword extends. The Thread class is found in java.lang package.

The syntax for creating a new class that extends Thread class is as follows:

Class Myclass extends Thread

2. Now in this newly created class, define a method run(). Here, run() method acts as entry point of the new thread. The run() method contains the actual task that thread will perform. Thus, we override run() method of Thread class.

public void run()

{

// statements to be executed.

}

5. Run the thread. For this, we need to call to start() method of Thread class because we cannot call run() method directly. The syntax to call start() method is as follows:

t.start();

Now, the thread will start execution on the object of Myclass, and statements inside run() method will be executed while calling it. By following all the above steps, you can create a Thread in Java.

**Example program**

// Custom thread class.

public class MyThread extends Thread

{

// Override the run method in Runnable.

public void run()

{

System.out.println("New thread running ");

}

public static void main(String[] args)

{

System.out.println("Main thread running");

// Create an object of MyThread class.

MyThread th = new MyThread();

// Create an object of Thread class and pass the object reference variable th.

Thread t = new Thread(th);

// Now run thread on the object. For this, call start() method using reference variable t.

t.start(); // This thread will execute statements inside run() method of MyThread object.

}

}

**Explanation:**

1. In this example program, we create a class MyThread with run() method. The class MyThread extends Thread class so that we are overriding run() method of Thread class to perform a specific task.

2. We created an object of MyThread class with reference variable th so that run() is available for execution. Declaring an object of MyThread class creates a new thread (child thread) from the main thread whose execution will start in the run() method.

3. Now, we created an object of Thread class and passes the reference variable th as an argument to its constructor.

4. When JVM executes t.start(); then start() method is called on a new thread. Thread schedular places the new thread in the queue for execution.

Placing new thread in the queue does not mean that it will execute immediately. It will run when other running threads release the CPU time.

When CPU allocates time slots for a new thread for its execution, thread starts executing the code inside run() method. In run() method, we wrote code to print a statement on the console.

The most important thing to realize that the main thread and new thread are two separate processes and are not executed sequentially. This is because we created and started a new thread from main thread.

It does not mean that new thread will execute first and then return to main thread. Here, two different tasks are performing one by one by two independent threads under the same program environment.

………………………

public class NewThread extends Thread {

public void run()

{

Thread th1 = Thread.currentThread();

System.out.println(th1);

System.out.println("New thread strats running");

System.out.println("I am in run() method");

}

public static void main(String[] args)

{

System.out.println("Main thread starts running");

Thread ct1 = Thread.currentThread();

System.out.println(ct1);

int ac1 = Thread.activeCount();

System.out.println(ac1);

// Create an object of NewThread class.

NewThread nt = new NewThread();

int ac2 = Thread.activeCount();

System.out.println(ac2);

// Create an object of Thread class and pass the object reference variable nt.

Thread t = new Thread(nt);

int ac3 = Thread.activeCount();

System.out.println(ac3);

// Now run thread on the object. For this, call start() method using reference variable t.

t.start(); // This thread will execute statements inside run() method of NewThread object.

int ac4 = Thread.activeCount();

System.out.println(ac4);

t.setName("NewThread"); // Setting a new name of thread.

}

}

**Explanation:**

1. In this example program, you can observe output that when JVM executes t.start() method then the number of active threads becomes 2 that shows that when the start() method is called, a new thread goes into runnable state (ready for execution). Before it, the number of active thread is one.

2. When JVM execute Thread.currentThread(); in the run() method, it will show name of current thread running in the run() method. Therefore, the output will print “Thread[NewThread,5,main]”.

## Creating Threads in Java using Runnable Interface

Threads can also be created by implementing Runnable interface of java.land package. Creating a thread by implementing Runnable interface is very similar to creating a thread by extending Thread class.

Program source code 3:

public class MyThread implements Runnable

{

public void run()

{

System.out.println("New thread running ");

for(int i = 1; i <= 5; i++)

{

System.out.println(i);

}

System.out.println(Thread.currentThread());

}

public static void main(String[] args)

{

System.out.println("Main thread running");

// Create an object of MyThread class.

MyThread th = new MyThread();

// Create an object of Thread class and pass reference variable th to Thread class constructor.

Thread t = new Thread(th);

t.start(); // This thread will execute statements inside run() method of MyThread object.

}

}

## Multitasking with single Thread in Java

In all the previous example programs, you will have noticed that one thread executes only one task at a time. But we can also execute multiple tasks from a single thread.

Program source code 4:

public class MyThread implements Runnable

{

int a = 20, b = 10;

public void run()

{

addition(); // task1

subtraction(); // task2

multiplication(); // task3

}

void addition()

{

int sum = a + b;

System.out.println("Addition of two numbers: " +sum);

}

void subtraction()

{

int sub = a - b;

System.out.println("Subtraction of two numbers: " +sub);

}

void multiplication()

{

int multiply = a \* b;

System.out.println("Multiplication of two numbers: " +multiply);

}

public static void main(String[] args)

{

System.out.println("Main thread running");

MyThread th = new MyThread();

Thread t = new Thread(th);

t.start(); // This thread will execute statements inside run() method of MyThread object.

}

}

# Life Cycle of Thread in Java | Thread State

**Life Cycle of Thread in Java** is basically state transitions of a thread that starts from its birth and ends on its death.

When an instance of a thread is created and is executed by calling start() method of Thread class, the thread goes into runnable state.

When sleep() or wait() method is called by Thread class, the thread enters into non-runnable state.

From non-runnable state, thread comes back to runnable state and continues execution of statements. When the thread comes out of run() method, it dies. These state transitions of a thread are called **Thread life cycle in Java**.

To work with threads in a program, it is important to identify thread state. So. let’s understand how to identify thread states in Java thread life cycle.

## Thread States in Java

A thread is a path of execution in a program that enters in any one of the following five states during its life cycle. The five states are as follows:

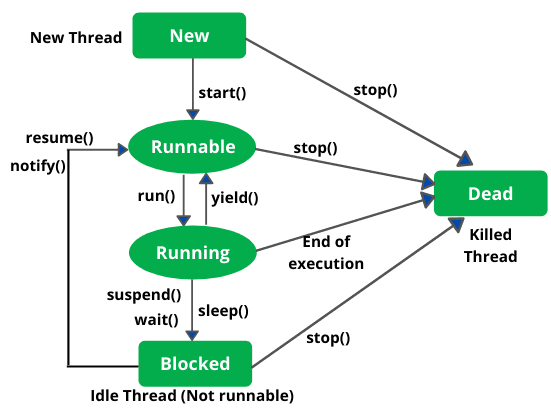
1. New

2. Runnable

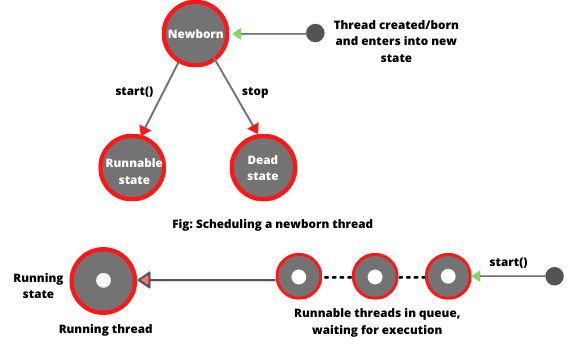
3. Running

4. Blocked (Non-runnable state)

5. Dead



1. **New (Newborn State):** When we create a thread object using Thread class, thread is born and is known to be in Newborn state. That is, when a thread is born, it enters into new state but the start() method has not been called yet on the instance.



1. **Runnable state:** Runnable state means a thread is ready for execution. When the start() method is called on a new thread, thread enters into a runnable state.

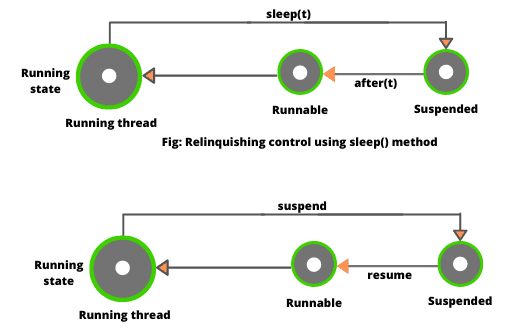
In runnable state, thread is ready for execution and is waiting for availability of the processor (CPU time). That is, thread has joined queue (line) of threads that are waiting for execution.

If all threads have equal priority, CPU allocates time slots for thread execution on the basis of first-come, first-serve manner. The process of allocating time to threads is known as **time slicing**. A thread can come into runnable state from running, waiting, or new states.

**3.** **Running state:** Running means Processor (CPU) has allocated time slot to thread for its execution. When thread scheduler selects a thread from the runnable state for execution, it goes into running state. Look at the above figure.

In running state, processor gives its time to the thread for execution and executes its run method. This is the state where thread performs its actual functions. A thread can come into running state only from runnable state.

A running thread may give up its control in any one of the following situations and can enter into the blocked state.



1. When sleep() method is invoked on a thread to sleep for specified time period, the thread is out of queue during this time period. The thread again reenters into the runnable state as soon as this time period is elapsed.

2. When a thread is suspended using suspend() method for some time in order to satisfy some conditions. A suspended thread can be revived by using resume() method.

3. When wait() method is called on a thread to wait for some time. The thread in wait state can be run again using notify() or notifyAll() method.

**4.** **Blocked state:** A thread is considered to be in the blocked state when it is suspended, sleeping, or waiting for some time in order to satisfy some condition.

**5.** **Dead state:** A thread dies or moves into dead state automatically when its run() method completes the execution of statements. That is, a thread is terminated or dead when a thread comes out of run() method. A thread can also be dead when the stop() method is called.

# Creating Multiple Threads in Java

Program source code 1:

// Two threads performing two tasks at a time.

public class MyThread extends Thread

{

// Declare a String variable to represent task.

String task;

MyThread(String task)

{

this.task = task;

}

public void run()

{

for(int i = 1; i <= 5; i++)

{

System.out.println(task+ " : " +i);

try

{

Thread.sleep(1000); // Pause the thread execution for 1000 milliseconds.

}

catch(InterruptedException ie) {

System.out.println(ie.getMessage());

}

} // end of for loop.

} // end of run() method.

public static void main(String[] args)

{

// Create two objects to represent two tasks.

MyThread th1 = new MyThread("Cut the ticket"); // Passing task as an argument to its constructor.

MyThread th2 = new MyThread("Show your seat number");

// Create two objects of Thread class and pass two objects as parameter to constructor of Thread class.

Thread t1 = new Thread(th1);

Thread t2 = new Thread(th2);

t1.start();

t2.start();

}

}

……………..Program source code 3:

public class MyThread1 implements Runnable

{

public void run() // Entry point of Thread1

{

for(int i = 0; i < 5; i++)

{

System.out.println("First Child Thread: " +i);

}

System.out.println("\t First child existed");

}

}

public class MyThread2 implements Runnable

{

public void run() // Entry point of Thread2

{

for(int i = 0; i < 5; i++)

{

System.out.println("Second Child Thread: " +i);

}

System.out.println("Second child existed");

}

}

public class MyClass {

public static void main(String[] args)

{

MyThread1 th1 = new MyThread1();

Thread t1 = new Thread(th1);

t1.start(); // Execution of first thread is started.

MyThread2 th2 = new MyThread2();

Thread t2 = new Thread(th2);

t2.start(); // Execution of second thread is started.

int j = 0;

while(j < 4)

{

System.out.println("Main Thread: " +j);

j = j + 1;

}

System.out.println("\t Main thread existing");

}

}

## Multiple Threads acting on Single object

It is also possible to create two or more threads on a single object. Let’s create a program where three threads will share the same object (same run() method).

public class MultipleThread implements Runnable

{

String task;

MultipleThread(String task)

{

this.task = task;

}

public void run()

{

for(int i = 1; i <= 5; i++)

{

System.out.println(task+ ":" +i);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public static void main(String[] args)

{

Thread nThread = Thread.currentThread();

System.out.println("Name of thread: " +nThread);

// Multiple child threads acting on single object.

MultipleThread mt = new MultipleThread("Hello Java");

Thread t1 = new Thread(mt);

Thread t2 = new Thread(mt);

Thread t3 = new Thread(mt);

t1.start();

t2.start();

t3.start();

int count = Thread.activeCount();

System.out.println("No of active threads: " +count);

}

}

# Thread Scheduler in Java | Time Slicing in Java

**Thread scheduler in Java** is the component of JVM that determines the execution order of multiple threads on a single processor (CPU).

It decides the order in which threads should run. This process is called **thread scheduling in Java**.

When a system with a single processor executes a program having multiple threads, CPU executes only a single thread at a particular time.

Other threads in the program wait in Runnable state for getting the chance of execution on CPU because at a time only one thread can get the chance to access the CPU.

The thread scheduler selects a thread for execution from runnable state. But there is no guarantee that which thread from runnable pool will be selected next to run by the thread scheduler.

Java runtime system mainly uses one of the following two strategies:

1. **Preemptive scheduling**  
2. **Time-sliced scheduling**

## Preemptive Scheduling

This scheduling is based on priority. Therefore, this scheduling is known as priority-based scheduling. In the priority-based scheduling algorithm, Thread scheduler uses the priority to decide which thread should be run.

If a thread with a higher priority exists in Runnable state (ready state), it will be scheduled to run immediately.

In case more than two threads have the same priority then CPU allocates time slots for thread execution on the basis of first-come, first-serve manner.

## Time-Sliced Scheduling

The process of allocating time to threads is known as **time slicing in Java**. Time-slicing is based on non-priority scheduling. Under this scheduling, every running thread is executed for a fixed time period.

A currently running thread goes to the Runnable state when its time slice is elapsed and another thread gets time slots by CPU for execution.

With time-slicing, threads having lower priorities or higher priorities gets the same time slots for execution.

Hope that this tutorial has covered the basic points of time slicing and **thread scheduler in Java**. I hope that you will have understood this topic. In the next tutorial, we will learn thread priority.

# Thread Priority in Java with Example

**Thread priority in Java** is a number assigned to a thread that is used by Thread scheduler to decide which thread should be allowed to execute.

In Java, each thread is assigned a different priority that will decide the order (preference) in which it is scheduled for running.

Thread priorities are represented by a number from 1 to 10 that specifies the relative priority of one thread to another. The thread with the highest priority is selected by the scheduler to be executed first.

The default priority of a thread is 5. Thread class in java also provides several priority constants to define the priority of a thread. These are:

1. MIN\_PRIORITY = 1  
2. NORM\_PRIORITY = 5  
3. MAX\_PRIORTY = 10

These constants are public, final, and static members of the Thread class.

In all the previous tutorials, threads we have discussed, are of the same priority (equal priority). The threads of the same priority are provided equal time by Java scheduler.

Thread scheduler selects the thread for execution on the first-come, first-serve basis. That is, the threads having equal priorities share the processor time on the first-come, first-serve basis.

When multiple threads are ready for execution, the highest priority thread is selected and executed by JVM. In case when a high priority thread stops, yields, or enters into the blocked state, a low priority thread starts executing.

If any high priority thread enters into the runnable state, it will preempt the currently running thread forcing it to move to the runnable state. Note that the highest priority thread always preempts any lower priority thread.

## How to get Priority of Current Thread in Java?

Thread class provides a method named getPriority() that is used to determine the priority of a thread. It returns the priority of a thread through which it is called.

public class A implements Runnable

{

public void run()

{

System.out.println(Thread.currentThread()); // This method is static.

}

public static void main(String[] args)

{

A a = new A();

Thread t = new Thread(a, "NewThread");

System.out.println("Priority of Thread: " +t.getPriority());

System.out.println("Name of Thread: " +t.getName());

t.start();

}

}

## How to set Priority of Thread in Java?

The setPriority() of Thread class is used to set the priority of a thread. This method accepts an integer value as an argument and sets that value as priority of a thread through which it is called. The syntax to set the priority of a thread is as follows:

Program source code 2:

public class A implements Runnable

{

public void run()

{

System.out.println(Thread.currentThread()); // This method is static.

}

public static void main(String[] args)

{

A a = new A();

Thread t = new Thread(a, "NewThread");

t.setPriority(2); // Setting the priority of thread.

System.out.println("Priority of Thread: " +t.getPriority());

System.out.println("Name of Thread: " +t.getName());

t.start();

}

}

Program source code 3:

public class A implements Runnable

{

public void run()

{

System.out.println(Thread.currentThread()); // This method is static.

}

public static void main(String[] args)

{

A a = new A();

Thread t1 = new Thread(a, "First Thread");

Thread t2 = new Thread(a, "Second Thread");

Thread t3 = new Thread(a, "Third Thread");

t1.setPriority(4); // Setting the priority of first thread.

t2.setPriority(2); // Setting the priority of second thread.

t3.setPriority(8); // Setting the priority of third thread.

t1.start();

t2.start();

t3.start();

}

}

Program source code 4:

public class X implements Runnable

{

public void run()

{

System.out.println("Thread X started");

for(int i = 1; i<=4; i++)

{

System.out.println("Thread X: " +i);

}

System.out.println("Exit from X");

}

}

public class Y implements Runnable

{

public void run()

{

System.out.println("Thread Y started");

for(int j = 0; j <= 4; j++)

{

System.out.println("Thread Y: " +j);

}

System.out.println("Exit from Y");

}

}

public class Z implements Runnable

{

public void run()

{

System.out.println("Thread Z started");

for(int k = 0; k <= 4; k++)

{

System.out.println("Thread Z: " +k);

}

System.out.println("Exit from Z");

}

}

public class ThreadPriority {

public static void main(String[] args)

{

X x = new X();

Y y = new Y();

Z z = new Z();

Thread t1 = new Thread(x);

Thread t2 = new Thread(y);

Thread t3 = new Thread(z);

t1.setPriority(Thread.MAX\_PRIORITY);

t2.setPriority(t2.getPriority() + 4);

t3.setPriority(Thread.MIN\_PRIORITY);

t1.start();

t2.start();

t3.start();

}

}

# Yield Method in Java | Yield in Java

**Yield() method in Java |** When a currently executing thread goes to the runnable state from running state, this process is called **yield in Java**.

When running state receives higher priority thread than the thread that is currently running, thread sheduler sends the currently running thread back to the runnable state and selects another thread of equal or higher priority to start its execution.

## Program based on yield() method in Java

public class A implements Runnable

{

public void run()

{

System.out.println(Thread.currentThread());

Thread.yield(); // Calling yield() method on current thread to move back into the runnable state from running state.

System.out.println(Thread.currentThread());

}

public static void main(String[] args)

{

A a1 = new A();

Thread t1 = new Thread(a1, "First Child Thread");

A a2 = new A();

Thread t2 = new Thread(a2, "Second Child Thread");

t1.start();

t2.start();

}

}

Program source code 2:

public class A implements Runnable

{

public void run()

{

System.out.println(Thread.currentThread());

Thread.yield(); // Calling yield() method on current thread to move back into the runnable state from running state.

System.out.println(Thread.currentThread());

}

public static void main(String[] args)

{

A a1 = new A();

Thread t1 = new Thread(a1, "First Child Thread");

A a2 = new A();

Thread t2 = new Thread(a2, "Second Child Thread");

t1.setPriority(4); // Setting the priority of thread

t2.setPriority(8);

t1.start();

t2.start();

}

}

**Key Points**

1. Do not use yield() method constantly in Java program.

2. When the yield() method is called on a thread object, it does not move the thread into sleeping, waiting, or blocking state. It sends thread into runnable state that can be resumed to the running state later.