Multithreaded Programming

Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking. 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:

1. Process-based Multitasking (Multiprocessing)
2. 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.

Thread-based Multitasking (Multithreading)

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

The Java Thread Model

The Java thread model is integral to the Java runtime system, which relies on multithreading to maintain an asynchronous environment, thereby optimizing CPU usage and reducing inefficiencies. Unlike single-threaded systems that utilize a polling event loop, where a single thread continuously checks for events and can block the entire program while waiting for resources, Java's multithreading allows individual threads to pause without halting the execution of other threads. This means that when one thread is blocked—such as waiting for user input or reading data from a network—other threads can continue executing, thus improving overall application responsiveness and efficiency.

Java's multithreading capabilities are designed to function effectively on both single-core and multi-core systems. In single-core environments, multiple threads share CPU time, allowing idle CPU cycles to be utilized efficiently even if they do not run simultaneously. In contrast, multi-core systems can execute multiple threads at the same time, further enhancing performance. The introduction of the Fork/Join Framework in Java supports parallel programming by facilitating the creation of applications that can scale effectively across multi-core environments. Overall, Java's multithreading features provide developers with powerful tools to create responsive and efficient applications that can handle concurrent operations seamlessly.

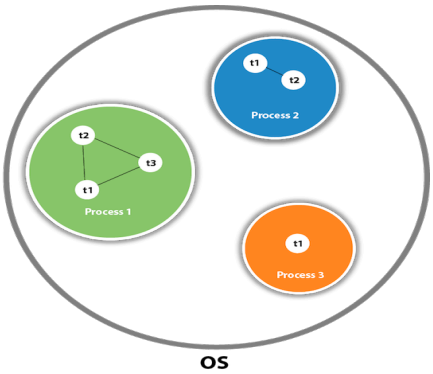
Thread Priorities

In Java, each thread is assigned a priority, which is an integer value that determines its relative importance compared to other threads. While the absolute value of a thread's priority does not influence its execution speed in isolation, it plays a critical role during context switching—when the CPU switches from one thread to another. A thread can yield control voluntarily by sleeping or blocking on I/O, prompting the system to check for the highest-priority thread that is ready to run. Alternatively, a lower-priority thread can be preempted by a higher-priority thread, allowing the latter to take over execution immediately, which exemplifies Java's use of preemptive multitasking.

When multiple threads with the same priority compete for CPU time, their execution can vary depending on the operating system. For instance, Windows typically employs a round-robin scheduling method, automatically time-slicing threads of equal priority. In contrast, other operating systems may require threads to voluntarily yield control if they wish to allow others of the same priority to run. This prioritization mechanism ensures that higher-priority threads receive CPU resources preferentially, enhancing the overall efficiency and responsiveness of multithreaded applications in Java.

Thread

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.



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.

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: 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.

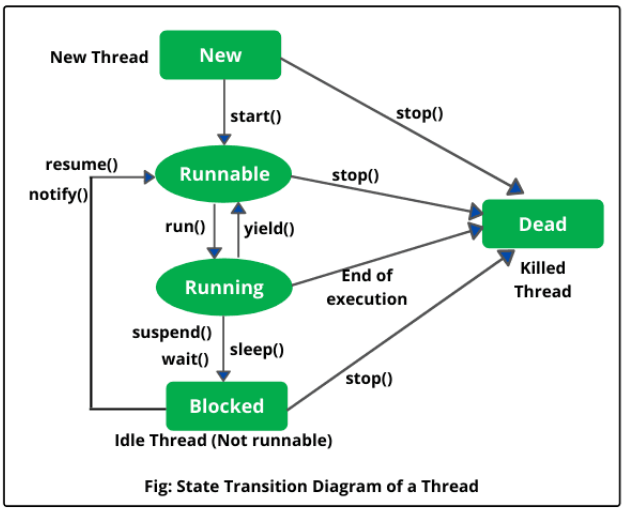


Fig: State Transition Diagram of a Thread

There are two ways to create a thread:

- By extending Thread class
- 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:

- public void run(): is used to perform action for a thread.
- public void start(): starts the execution of the thread.JVM calls the run() method on the thread.
- public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- public Thread currentThread(): returns the reference of currently executing thread.
- public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.
- public void suspend(): is used to suspend the thread(depricated).
- public void resume(): is used to resume the suspended thread(depricated).
- public void stop(): is used to stop the thread(depricated).
- public boolean isDaemon(): tests if the thread is a daemon thread.
- public void interrupt(): interrupts the thread.
- public boolean isInterrupted(): tests if the thread has been interrupted.
- public static boolean interrupted(): tests if the current thread has been interrupted.

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().

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.

Java Thread Example by extending Thread class

```
class Multi extends Thread
{
    public void run()
    {
           System.out.println("thread is running...");
    }
    public static void main(String args[])
    {
      Multi t1=new Multi();  t1.start();
     }
}
Output:thread is running...
```

Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable
{
   public void run()
   {
       System.out.println("thread is running...");
   }
   public static void main(String args[])
   {
      Multi3 m1=new Multi3();
      Thread t1 =new Thread(m1);
      t1.start();
    }
}
Output:thread is running...
```

If you are not extending the Thread class, your class object would not be treated as a thread object. So, you need to explicitly create Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

Creating Multiple Threads

Creating multiple threads in Java involves creating multiple instances of the `Thread` class or instances that implement the `Runnable` interface. This allows concurrent execution of code in a Java program. Let's break down the steps to create multiple threads and explore a few examples.

1. Basic Approach: Extending the `Thread` Class

To create a thread, you can extend the `Thread` class and override its `run()` method. Each instance of this class represents a separate thread.

Example:

```
class MyThread extends Thread {
    private String threadName;

    public MyThread(String name) {
        this.threadName = name;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(threadName + " - Count: " + i);
```

```
            try {
                Thread.sleep(500); // pause for 500ms
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}

public class MultipleThreadsExample {
    public static void main(String[] args) {
        MyThread thread1 = new MyThread("Thread 1");
        MyThread thread2 = new MyThread("Thread 2");

        thread1.start();
        thread2.start();
    }
}
```

In this example:

- We create two threads, `thread1` and `thread2`.
- Both threads will execute the `run()` method independently and print counts.
- `Thread.sleep(500)` pauses each thread for 500ms after each print statement.

2. Implementing the `Runnable` Interface

Another way to create threads is by implementing the `Runnable` interface, which makes the code more flexible since Java supports only single inheritance. You can pass instances of `Runnable` to `Thread` objects and start them.

Example:

```
class MyRunnable implements Runnable {
    private String threadName;

    public MyRunnable(String name) {
        this.threadName = name;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(threadName + " - Count: " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new MyRunnable("Thread 1"));
        Thread thread2 = new Thread(new MyRunnable("Thread 2"));

        thread1.start();
        thread2.start();
    }
}
```

Here, `MyRunnable` implements `Runnable`, and we pass instances of it to `Thread` objects. Both threads run concurrently.

3. Creating Multiple Threads Using Loops

If you want to create a large number of threads, using a loop is efficient.

Example:

```
class WorkerThread implements Runnable {
    private int id;

    public WorkerThread(int id) {
        this.id = id;
    }

    @Override
    public void run() {
        System.out.println("Thread " + id + " is running");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Thread " + id + " has finished");
    }
}

public class MultipleThreadsLoop {
    public static void main(String[] args) {
        int numThreads = 5;
        for (int i = 1; i <= numThreads; i++) {
            Thread thread = new Thread(new WorkerThread(i));
            thread.start();
        }
    }
}
```

In this example:

- We create 5 threads using a `for` loop.
- Each thread prints a starting and finishing message with its ID.

`isAlive()` and `join()`

- **`isAlive()`**: Checks if a thread is still active.
- **`join()`**: Allows one thread to wait for another thread to complete.

Example:

```
class NewThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

public class JoinExample {
    public static void main(String[] args) {
        NewThread t = new NewThread();
        t.start();
        try {
            t.join(); // Waits for t to complete
        } catch (InterruptedException e) {
            System.out.println(e);
        }
        System.out.println("Main thread resumes after t finishes.");
    }
}
```

Thread Priorities

Thread priorities control the order of thread execution, with priority values from `Thread.MIN_PRIORITY` (1) to `Thread.MAX_PRIORITY` (10).

Example:

```
class PriorityThread extends Thread {
    public void run() {
        System.out.println("Thread running with priority: " + this.getPriority());
    }
}

public class PriorityExample {
    public static void main(String[] args) {
        PriorityThread t1 = new PriorityThread();
        PriorityThread t2 = new PriorityThread();

        t1.setPriority(Thread.MIN_PRIORITY); // 1
        t2.setPriority(Thread.MAX_PRIORITY); // 10

        t1.start();
        t2.start();
    }
}
```