**Multithreading :**  Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

**The process of executing multiple threads simultaneously(at the same time) is known as multithreading.**

* The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to maximum utilize the CPU time. A multithreaded program contains two or more parts that can run concurrently. Each such part of a program called thread.
* Threads are lightweight sub-processes, they share the common memory space. In Multithreaded environment, programs that are benefited from multithreading, utilize the maximum CPU time so that the idle time can be kept to minimum.

**Threads can be created by using two mechanisms :**

1. Extending the Thread class
2. Implementing the Runnable Interface

**Thread class Method :**

* **getName():** It is used for Obtaining a thread’s name
* **getPriority():** Obtain a thread’s priority
* **isAlive():** Determine if a thread is still running
* **join():** Wait for a thread to terminate
* **run():** Entry point for the thread
* **sleep():** suspend a thread for a period of time
* **start():** start a thread by calling its run() method

**Thread creation by extending the Thread class :** We create a class that extends the**java.lang.Thread**class. This class overrides the **run()** method available in the Thread class. A thread begins its life inside **run()** method. We create an object of our new class and call **start()** method to start the execution of a thread. **Start()** invokes the **run()** method on the Thread object.

**public** **class** MultithreadExample {

**public** **static** **void** main(String[] args) {

**int** n = 8; // Number of threads

**for** (**int** i = 0; i < n; i++) {

ThreadExample object = **new** ThreadExample();

object.start();

}

}

}

**class** ThreadExample **extends** Thread{

**public** **void** run(){

**try** {

// Displaying the thread that is running

System.***out***.println("Thread " + Thread.*currentThread*().getId() + " is running");

}

**catch** (Exception e) {

// Throwing an exception

System.***out***.println("Exception is caught");

}

}

}

**Thread creation by implementing the Runnable Interface :** We create a new class which implements **java.lang.Runnable** interface and override **run()** method. Then we instantiate a Thread object and call start() method on this object.

**class** MultithreadingDemo **implements** Runnable {

**public** **void** run() {

**try** {

// Displaying the thread that is running

System.***out***.println("Thread " + Thread.*currentThread*().getId() + " is running");

}

**catch** (Exception e) {

// Throwing an exception

System.***out***.println("Exception is caught");

}

}

}

// Main Class

**public** **class** RunnableExample {

**public** **static** **void** main(String[] args){

**int** n = 8; // Number of threads

**for** (**int** i = 0; i < n; i++) {

Thread object = **new** Thread(**new** MultithreadingDemo());

object.start();

}

}

}

**Thread Class vs Runnable Interface :**

* If we extend the Thread class, our class cannot extend any other class because Java doesn’t support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
* We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.
* Using runnable will give you an object that can be shared amongst multiple threads.

**Lifecycle and States of a Thread in Java :**A [thread](http://www.geeksforgeeks.org/multithreading-in-java/) in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

* **New**
* **Runnable**
* **Blocked**
* **Waiting**
* **Timed Waiting**
* **Terminated**

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

When a new thread is created, it is in the new state. The thread has not yet started to run when thread is in this state. When a thread lies in the new state, it’s code is yet to be run and hasn’t started to execute.

**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.



**The most significant benefits of multithreading are:**

1. Better CPU utilization.
2. Simpler program design in some situations.
3. More responsive programs.
4. More fair division of CPU resources between different tasks.

**Main thread in Java :** Java provides built-in support for multithreaded programming. A multi-threaded 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.

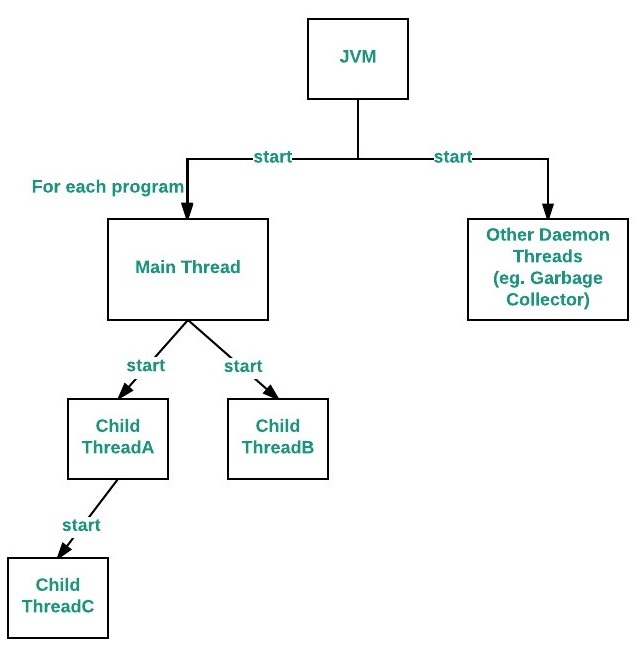
**Main Thread :** When a Java program starts up, one thread begins running immediately. This is usually called the main thread of our program, because it is the one that is executed when our program begins.

Properties :

It is the thread from which other “child” threads will be spawned.

Often, it must be the last thread to finish execution because it performs various shutdown actions.

The **main thread** is created automatically when our program is started. To control it we must obtain a reference to it. This can be done by calling the method **currentThread(** ) which is present in Thread class. This method returns a reference to the thread on which it is called. The default priority of Main thread is 5 and for all remaining user threads priority will be inherited from parent to child.



**Relation between the main() method and main thread in Java :** For each program, a Main thread is created by [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)(Java Virtual Machine). The “Main” thread first verifies the existence of the main() method, and then it initializes the class. Note that from JDK 6, main() method is mandatory in a standalone java application.

The statement “**Thread.currentThread().join()”**, will tell Main thread to wait for this thread(i.e. wait for itself) to die. Thus Main thread wait for itself to die, which is nothing but a deadlock.

**Java Thread Priority in Multithreading :** In a Multi threading environment, thread scheduler assigns processor to a thread based on priority of thread. Whenever we create a thread in Java, it always has some priority assigned to it. Priority can either be given by JVM while creating the thread or it can be given by programmer explicitly.   
Accepted value of priority for a thread is in range of 1 to 10. There are 3 static variables defined in Thread class for priority.  
**public static int MIN\_PRIORITY:**This is minimum priority that a thread can have. Value for this is 1.   
**public static int NORM\_PRIORITY:** This is default priority of a thread if do not explicitly define it. Value for this is 5.   
**public static int MAX\_PRIORITY**: This is maximum priority of a thread. Value for this is 10.

**Get and Set Thread Priority:**

**public final int getPriority():**java.lang.Thread.getPriority() method returns priority of given thread.

**public final void setPriority(int newPriority):** java.lang.Thread.setPriority() method changes the priority of thread to the value newPriority. This method throws IllegalArgumentException if value of parameter newPriority goes beyond minimum(1) and maximum(10) limit.

Note:

* Thread with highest priority will get execution chance prior to other threads. Suppose there are 3 threads t1, t2 and t3 with priorities 4, 6 and 1. So, thread t2 will execute first based on maximum priority 6 after that t1 will execute and then t3.
* Default priority for main thread is always 5, it can be changed later. Default priority for all other threads depends on the priority of parent thread.
* If two threads have same priority then we can’t expect which thread will execute first. It depends on thread scheduler’s algorithm(Round-Robin, First Come First Serve, etc)
* If we are using thread priority for thread scheduling then we should always keep in mind that underlying platform should provide support for scheduling based on thread priority.

 // main thread priority is 6 now

        Thread.currentThread().setPriority(6);

**Synchronized in Java :** [Multi-threaded](http://quiz.geeksforgeeks.org/multithreading-in-java/)programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.

Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

Following is the general form of a synchronized block:

// Only one thread can execute at a time.

// sync\_object is a reference to an object whose lock associates with the monitor.

// The code is said to be synchronized on the monitor object

**synchronized**(sync\_object){

// Access shared variables and other

// shared resources

}

This synchronization is implemented in Java with a concept called monitors. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

We do not always have to synchronize a whole method. Sometimes it is preferable to synchronize only part of a method. Java synchronized blocks inside methods makes this possible.

The synchronized keyword can be used to mark four different types of blocks:

1. **Instance methods**
2. **Static methods**
3. **Code blocks inside instance methods**
4. **Code blocks inside static methods**

**Instance methods**

public class MyCounter {

private int count = 0;

**// Instance methods**

public **synchronized** void add(int value){

this.count += value;

}

}

Notice the use of the synchronized keyword in the add() method declaration. This tells Java that the method is synchronized.

A synchronized instance method in Java is synchronized on the instance (object) owning the method. Thus, each instance has its synchronized methods synchronized on a different object: the owning instance.

Only one thread per instance can execute inside a synchronized instance method. If more than one instance exist, then one thread at a time can execute inside a synchronized instance method per instance. One thread per instance.

This is true across all synchronized instance methods for the same object (instance). Thus, in the following example, only one thread can execute inside either of of the two synchronized methods. One thread in total per instance:

**Static methods :** Static methods are marked as synchronized just like instance methods using the synchronized keyword. Here is a Java synchronized static method example:

public static MyStaticCounter{

private static int count = 0;

public static **synchronized** void add(int value){

count += value;

}

}

Also here the synchronized keyword tells Java that the add() method is synchronized.

Synchronized static methods are synchronized on the class object of the class the synchronized static method belongs to. Since only one class object exists in the Java VM per class, only one thread can execute inside a static synchronized method in the same class.

In case a class contains more than one static synchronized method, only one thread can execute inside any of these methods at the same time. Look at this static synchronized method example:

public static MyStaticCounter{

private static int count = 0;

public static synchronized void add(int value){

count += value;

}

public static synchronized void subtract(int value){

count -= value;

}

}

Only one thread can execute inside any of the two add() and subtract() methods at any given time. If Thread A is executing add() then Thread B cannot execute neither add() nor subtract() until Thread A has exited add().

If the static synchronized methods are located in different classes, then one thread can execute inside the static synchronized methods of each class. One thread per class regardless of which static synchronized method it calls.

**Synchronized Blocks in Instance Methods :** You do not have to synchronize a whole method. Sometimes it is preferable to synchronize only part of a method. Java synchronized blocks inside methods makes this possible.

Here is a synchronized block of Java code inside an unsynchronized Java method:

public void add(int value){

**synchronized**(this){

this.count += value;

}

}

This example uses the Java synchronized block construct to mark a block of code as synchronized. This code will now execute as if it was a synchronized method.

Notice how the Java synchronized block construct takes an object in parentheses. In the example "this" is used, which is the instance the add method is called on. The object taken in the parentheses by the synchronized construct is called a monitor object. The code is said to be synchronized on the monitor object. A synchronized instance method uses the object it belongs to as monitor object.

**Only one thread can execute inside a Java code block synchronized on the same monitor object.**

The following two examples are both synchronized on the instance they are called on. They are therefore equivalent with respect to synchronization:

public class MyClass {

public **synchronized** void log1(String msg1, String msg2){

log.writeln(msg1);

log.writeln(msg2);

}

public void log2(String msg1, String msg2){

synchronized(this){

log.writeln(msg1);

log.writeln(msg2);

}

}

}

Thus only a single thread can execute inside either of the two synchronized blocks in this example.

Had the second synchronized block been synchronized on a different object than this, then one thread at a time had been able to execute inside each method.

**Synchronized Blocks in Static Methods :**Synchronized blocks can also be used inside of static methods. Here are the same two examples from the previous section as static methods. These methods are synchronized on the class object of the class the methods belong to:

public class MyClass {

public static synchronized void log1(String msg1, String msg2){

log.writeln(msg1);

log.writeln(msg2);

}

public static void log2(String msg1, String msg2){

synchronized(MyClass.class){

log.writeln(msg1);

log.writeln(msg2);

}

}

}

Only one thread can execute inside any of these two methods at the same time.

Had the second synchronized block been synchronized on a different object than MyClass.class, then one thread could execute inside each method at the same time.

**Synchronized Blocks in Lambda Expressions :**It is even possible to use synchronized blocks inside a Java Lambda Expression as well as inside anonymous classes.

Here is an example of a Java lambda expression with a synchronized block inside. Notice that the synchronized block is synchronized on the class object of the class containing the lambda expression. It could have been synchronized on another object too, if that would have made more sense (given a specific use case), but using the class object is fine for this example.

import java.util.function.Consumer;

public class SynchronizedExample {

public static void main(String[] args) {

Consumer<String> func = (String param) -> {

**synchronized**(SynchronizedExample.class) {

System.out.println(

Thread.currentThread().getName() +

" step 1: " + param);

try {

Thread.sleep( (long) (Math.random() \* 1000));

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(

Thread.currentThread().getName() +

" step 2: " + param);

}

};

Thread thread1 = new Thread(() -> {

func.accept("Parameter");

}, "Thread 1");

Thread thread2 = new Thread(() -> {

func.accept("Parameter");

}, "Thread 2");

thread1.start();

thread2.start();

}

}

**Java Synchronized Example :**

Here is an example that starts 2 threads and have both of them call the add method on the same instance of Counter. Only one thread at a time will be able to call the add method on the same instance, because the method is synchronized on the instance it belongs to.

public class Example {

public static void main(String[] args){

Counter counter = new Counter();

Thread threadA = new CounterThread(counter);

Thread threadB = new CounterThread(counter);

threadA.start();

threadB.start();

}

}

Here are the two classes used in the example above, Counter and CounterThread.

public class Counter{

long count = 0;

public synchronized void add(long value){

this.count += value;

}

}

public class CounterThread extends Thread{

protected Counter counter = null;

public CounterThread(Counter counter){

this.counter = counter;

}

public void run() {

for(int i=0; i<10; i++){

counter.add(i);

}

}

}

Two threads are created. The same Counter instance is passed to both of them in their constructor. The Counter.add() method is synchronized on the instance, because the add method is an instance method, and marked as synchronized. Therefore only one of the threads can call the add() method at a time. The other thread will wait until the first thread leaves the add() method, before it can execute the method itself.

If the two threads had referenced two separate Counter instances, there would have been no problems calling the add() methods simultaneously. The calls would have been to different objects, so the methods called would also be synchronized on different objects (the object owning the method). Therefore the calls would not block. Here is how that could look:

public class Example {

public static void main(String[] args){

Counter counterA = new Counter();

Counter counterB = new Counter();

Thread threadA = new CounterThread(counterA);

Thread threadB = new CounterThread(counterB);

threadA.start();

threadB.start();

}

}

Notice how the two threads, threadA and threadB, no longer reference the same counter instance. The add method of counterA and counterB are synchronized on their two owning instances. Calling add() on counterA will thus not block a call to add() on counterB.

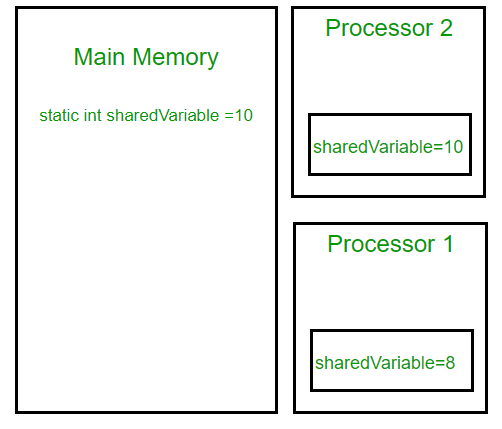
**Java Volatile Keyword :** The Java volatile keyword is used to mark a Java variable as "being stored in main memory". that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

Actually, since Java 5 the volatile keyword guarantees more than just that volatile variables are written to and read from main memory.

Using volatile is yet another way (like synchronized, atomic wrapper) of making class thread safe. Thread safe means that a method or class instance can be used by multiple threads at the same time without any problem.

Suppose that two threads are working on SharedObj. If two threads run on different processors each thread may have its own local copy of sharedVariable. If one thread modifies its value the change might not reflect in the original one in the main memory instantly. This depends on the [write policy](https://en.wikipedia.org/wiki/CPU_cache" \l "Write_policies) of cache. Now the other thread is not aware of the modified value which leads to data inconsistency.

Below diagram shows that if two threads are run on different processors, then value of sharedVariable may be different in different threads.



Note that volatile should not be confused with static modifier. static variables are class members that are shared among all objects. There is only one copy of them in main memory.

**volatile vs synchronized:**Before we move on let’s take a look at two important features of locks and synchronization.

**Mutual Exclusion:** It means that only one thread or process can execute a block of code (critical section) at a time.

**Visibility:** It means that changes made by one thread to shared data are visible to other threads.

Java’s synchronized keyword guarantees both mutual exclusion and visibility. If we make the blocks of threads that modifies the value of shared variable synchronized only one thread can enter the block and changes made by it will be reflected in the main memory. All other thread trying to enter the block at the same time will be blocked and put to sleep.

In some cases we may only desire the visibility and not atomicity. Use of synchronized in such situation is an overkill and may cause scalability problems. Here volatile comes to the rescue. Volatile variables have the visibility features of synchronized but not the atomicity features. The values of volatile variable will never be cached and all writes and reads will be done to and from the main memory. However, use of volatile is limited to very restricted set of cases as most of the times atomicity is desired. For example a simple increment statement such as x = x + 1; or x++ seems to be a single operation but is s really a compound read-modify-write sequence of operations that must execute atomically.

**volatile in Java vs C/C++:**Volatile in java is different from [“volatile” qualifier in C/C++](https://www.geeksforgeeks.org/understanding-volatile-qualifier-in-c/). For Java, “volatile” tells the compiler that the value of a variable must never be cached as its value may change outside of the scope of the program itself. In C/C++, “volatile” is needed when developing embedded systems or device drivers, where you need to read or write a memory-mapped hardware device. The contents of a particular device register could change at any time, so you need the “volatile” keyword to ensure that such accesses aren’t optimized away by the compiler.

**Java ThreadLocal :** The Java ThreadLocal class enables you to create variables that can only be read and written by the same thread. This class provides thread local variable.

Thus, even if two threads are executing the same code, and the code has a reference to the same ThreadLocal variable, the two threads cannot see each other's ThreadLocal variables. Thus, the Java ThreadLocal class provides a simple way to make code **[thread safe](http://tutorials.jenkov.com/java-concurrency/thread-safety.html)**.

* Basically it is an another way to achieve thread safety apart from writing immutable classes.
* Since Object is no more shared there is no requirement of Synchronization which can improve scalability and performance of application.
* It extends class Object.
* ThreadLocal provides thread restriction which is extension of local variable. ThreadLocal are visible only in single thread. No two thread can see each others thread local variable.
* These variable are generally private static field in classes and maintain its state inside thread.

**Creating a ThreadLocal**

You create a ThreadLocal instance just like you create any other Java object - via the new operator. Here is an example that shows how to create a ThreadLocal variable:

**private ThreadLocal threadLocal = new ThreadLocal();**

This only needs to be done once per thread. Multiple threads can now get and set values inside this ThreadLocal, and each thread will only see the value it set itself.

**Set ThreadLocal Value**

Once a ThreadLocal has been created you can set the value to be stored in it using its set() method.

threadLocal.**set**("A thread local value");

**Get ThreadLocal Value**

You read the value stored in a ThreadLocal using its get() method. Here is an example obtaining the value stored inside a Java ThreadLocal:

String threadLocalValue = (String) threadLocal.**get**();

**Remove ThreadLocal Value**

It is possible to remove a value set in a ThreadLocal variable. You remove a value by calling the ThreadLocal remove() method. Here is an example of removing the value set on a Java ThreadLocal:

threadLocal.**remove**();

**Generic ThreadLocal**

You can create a ThreadLocal with a generic type. Using a generic type only objects of the generic type can be set as value on the ThreadLocal. Additionally, you do not have to typecast the value returned by get(). Here is a generic ThreadLocal example:

**private ThreadLocal<String> myThreadLocal = new ThreadLocal<String>();**

Now you can only store strings in the ThreadLocal instance. Additionally, you do not need to typecast the value obtained from the ThreadLocal:

myThreadLocal.set("Hello ThreadLocal");

String threadLocalValue = myThreadLocal.get();

**Initial ThreadLocal Value**

It is possible to set an initial value for a Java ThreadLocal which will get used the first time get() is called - before set() has been called with a new value. You have two options for specifying an initial value for a ThreadLocal:

1. Create a ThreadLocal subclass that overrides the initialValue() method.
2. Create a ThreadLocal with a Supplier interface implementation.

I will show you both options in the following sections.

**Override initialValue()**

The first way to specify an initial value for a Java ThreadLocal variable is to create a subclass of ThreadLocal which overrides its initialValue() method. The easiest way to create a subclass of ThreadLocal is to simply create an anonymous subclass, right where you create the ThreadLocal variable. Here is an example of creating an anonymous subclass of ThreadLocal which overrides the initialValue() method:

**private ThreadLocal myThreadLocal = new ThreadLocal<String>() {**

**@Override protected String initialValue() {**

**return String.valueOf(System.currentTimeMillis());**

**}**

**};**

Note, that different threads will still see different initial values. Each thread will create its own initial value. Only if you return the exact same object from the initialValue() method, will all threads see the same object. However, the whole point of using a ThreadLocal in the first place is to avoid the different threads seeing the same instance.

**Difference between wait and sleep in Java :**

**Sleep():** This Method is used to pause the execution of current thread for a specified time in Milliseconds. Here, Thread does not lose its ownership of the monitor and resume’s it’s execution.

**Wait():** This method is defined in object class. It tells the calling thread (Current Thread) to wait until another thread invoke’s the notify() or notifyAll() method for this object, The thread waits until it reobtains the ownership of the monitor and Resume’s Execution.

|  |  |
| --- | --- |
| **Sleep():** | **Wait():** |
| Sleep() method belongs to Thread class. | Wait() method belongs to Object class. |
| Sleep() method does not release the lock on object during Synchronization. | Wait() method releases lock during Synchronization. |
| There is no need to call sleep() from Synchronized context. | Wait() should be called only from Synchronized context. |
| Sleep() is a static method. | Wait() is not a static method. |
| Sleep() Has Two Overloaded Methods:   * sleep(long millis)millis: milliseconds * sleep(long millis,int nanos) nanos: Nanoseconds | Wait() Has Three Overloaded Methods:   * wait() * wait(long timeout) * wait(long timeout, int nanos) |
| public static void sleep(long millis) throws Interrupted\_Execption | public final void wait(long timeout) |
| synchronized(monitor)  {  Thread.sleep(1000); Here Lock Is Held By The Current Thread  //after 1000 miliseconds, current thread will wake up, or after we call that is interrupt() method  } | synchronized(monitor)  {  monitor.wait() Here Lock Is Released By Current Thread  } |

**Similarity Between Both wait() and sleep() Method:**

Both Make The Current Thread go Into the Not Runnable State.

Both are Native Methods.

**synchronized**(monitor){

**while**(condition == **true**)

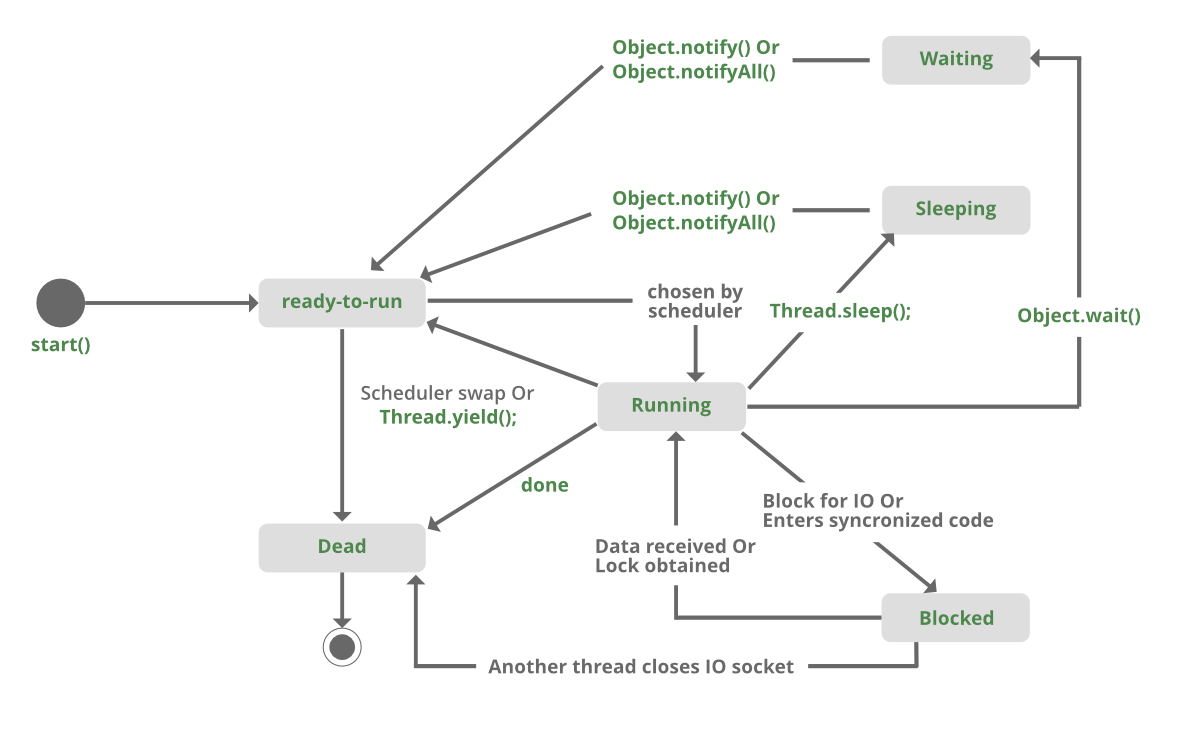
    {

        monitor.wait()  //releases monitor lock

    }

    Thread.sleep(100); //puts current thread on Sleep

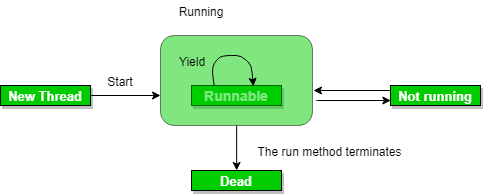
}



**Java Concurrency – yield(), sleep() and join() methods :** We can prevent the execution of a thread by using one of the following methods of Thread class.

**yield():** Suppose there are three threads t1, t2, and t3. Thread t1 gets the processor and starts its execution and thread t2 and t3 are in Ready/Runnable state. Completion time for thread t1 is 5 hour and completion time for t2 is 5 minutes. Since t1 will complete its execution after 5 hours, t2 has to wait for 5 hours to just finish 5 minutes job. In such scenarios where one thread is taking too much time to complete its execution, we need a way to prevent execution of a thread in between if something important is pending. yeild() helps us in doing so.

yield() basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should run. Otherwise, the current thread will continue to run.



**Use of yield method:**

* Whenever a thread calls java.lang.Thread.yield method, it gives hint to the thread scheduler that it is ready to pause its execution. Thread scheduler is free to ignore this hint.
* If any thread executes yield method , thread scheduler checks if there is any thread with same or high priority than this thread. If processor finds any thread with higher or same priority then it will move the current thread to Ready/Runnable state and give processor to other thread and if not – current thread will keep executing.

**public static native void yield()**

// MyThread extending Thread

class MyThread extends Thread{

    public void run()    {

        for (int i=0; i<5 ; i++)

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

                                + " in control");

    }

}

// Driver Class

public class yieldDemo{

    public static void main(String[]args)    {

        MyThread t = new MyThread();

        t.start();

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

            // Control passes to child thread

            Thread.yield();

            // After execution of child Thread

            // main thread takes over

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

                                + " in control");

        }

    }

}

Output may vary in machine to machine but chances of execution of yield() thread first is higher than the other thread because main thread is always pausing its execution and giving chance to child thread(with same priority).

**Note:**

* Once a thread has executed yield method and there are many threads with same priority is waiting for processor, then we can't specify which thread will get execution chance first.
* The thread which executes the yield method will enter in the Runnable state from Running state.
* Once a thread pauses its execution, we can't specify when it will get chance again it depends on thread scheduler.
* Underlying platform must provide support for preemptive scheduling if we are using yield method.

**sleep():** This method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

Syntax:

// sleep for the specified number of milliseconds

public static void sleep(long millis) throws InterruptedException

//sleep for the specified number of milliseconds plus nano seconds

public static void sleep(long millis, int nanos)

throws InterruptedException

Note:

* Based on the requirement we can make a thread to be in sleeping state for a specified period of time
* Sleep() causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, the CPU will be idle (and probably enter a power saving mode).

**yield() vs sleep()**

**yield():** indicates that the thread is not doing anything particularly important and if any other threads or processes need to be run, they can. Otherwise, the current thread will continue to run.

**sleep():** causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, the CPU will be idle (and probably enter a power saving mode).

**join(): The join()** method of a Thread instance is used to join the start of a thread’s execution to end of other thread’s execution such that a thread does not start running until another thread ends. If join() is called on a Thread instance, the currently running thread will block until the Thread instance has finished executing.

The join() method waits at most this much milliseconds for this thread to die. A timeout of 0 means to wait forever

Syntax:

// waits for this thread to die.

**public final void join() throws InterruptedException**

// waits at most this much milliseconds for this thread to die

**public final void join(long millis)**

**throws InterruptedException**

// waits at most milliseconds plus nanoseconds for this thread to die.

The java.lang.Thread.join(long millis, int nanos)

Thread t = **new** Thread(**new** JoinDemo());

        t.start();

        // Waits for 1000ms this thread to die.

        t.join(1000);

        System.out.println("\nJoining after 1000"+

                             " mili seconds: \n");

        System.out.println("Current thread: " +

                                    t.getName());

        // Checks if this thread is alive

        System.out.println("Is alive? " + t.isAlive());

**Note:**

If any executing thread t1 calls join() on t2 i.e; t2.join() immediately t1 will enter into waiting state until t2 completes its execution.

Giving a timeout within join(), will make the join() effect to be nullified after the specific timeout.

|  |  |  |  |
| --- | --- | --- | --- |
| **property** | **yield()** | **join()** | **sleep()** |
| purpose | If a thread wants to pass its execution to give chance to remaining threads of same priority then we should go for yield() | If a thread wants to wait until completing of some other thread then we should go for join() | If a thread does not want to perform any operation for a particular amount of time, then it goes for sleep() |
| Is it overloaded? | NO | YES | YES |
| Is it final? | NO | YES | NO |
| Is it throws? | NO | YES | YES |
| Is it native? | YES | | NO | | --- | | sleep(long ms)->native & sleep (long ms, int ns)-> non native |
| Is it static? | YES | NO | YES |

**Joining Threads in Java :** java.lang.Thread class provides the join() method which allows one thread to wait until another thread completes its execution. If t is a Thread object whose thread is currently executing, then t.join() will make sure that t is terminated before the next instruction is executed by the program.  
If there are multiple threads calling the join() methods that means overloading on join allows the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.  
**There are three overloaded join functions.**

* **join():** It will put the current thread on wait until the thread on which it is called is dead. If thread is interrupted then it will throw InterruptedException.  
  Syntax:

public final void join()

* **join(long millis)** :It will put the current thread on wait until the thread on which it is called is dead or wait for specified time (milliseconds).  
  Syntax:

public final synchronized void join(long millis)

* **join(long millis, int nanos)**: It will put the current thread on wait until the thread on which it is called is dead or wait for specified time (milliseconds + nanos).  
  Syntax:

public final synchronized void join(long millis, int nanos)

// creating two threads

        ThreadJoining t1 = new ThreadJoining();

        ThreadJoining t2 = new ThreadJoining();

        ThreadJoining t3 = new ThreadJoining();

        // thread t1 starts

        t1.start();

**// starts second thread after when first thread t1 has died.**

        try {

            System.out.println("Current Thread: "+Thread.currentThread().getName());

**t1.join();**

        }catch(Exception ex){

            System.out.println("Exception has " + "been caught" + ex);

    }

       // t2 starts

        t2.start();

**// starts t3 after when thread t2 has died(execution completed).**

        try {

            System.out.println("Current Thread: "+ Thread.currentThread().getName());

            t2.join();

        } catch(Exception ex){

            System.out.println("Exception has been" +" caught" + ex);

        }

          t3.start();

    }

**Java Thread suspend() method :**The suspend() method of thread class puts the thread from running to waiting state. This method is used if you want to stop the thread execution and start it again when a certain event occurs. This method allows a thread to temporarily cease execution. The suspended thread can be resumed using the resume() method.

**Syntax**: public final void **suspend**()

t1.start();

 t2.start();

 // suspend t2 thread

 t2.suspend();

// call run() method

 t3.start();

**Thread Pools :** A thread pool is a pool threads that can be "reused" to execute tasks, so that each thread may execute more than one task.

A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing.

A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, thread is contained in the thread pool again.

**Advantage of Java Thread Pool**

* Better performance It saves time because there is no need to create new thread.

**Real time usage :** It is used in Servlet and JSP where container creates a thread pool to process the request.

A server that creates a new thread for every request would spend more time and consume more system resources in creating and destroying threads than processing actual requests.Since active threads consume system resources, a [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)creating too many threads at the same time can cause the system to run out of memory. This necessitates the need to limit the number of threads being created.

* Java provides the **Executor framework** which is centered around the **Executor interface**, its sub-interface –**ExecutorService** and the class-**ThreadPoolExecutor**, which implements both of these interfaces. By using the executor, one only has to implement the Runnable objects and send them to the executor to execute.
* They allow you to take advantage of threading, but focus on the tasks that you want the thread to perform, instead of thread mechanics.
* To use thread pools, we first create a object of **ExecutorService** and pass a set of tasks to it. ThreadPoolExecutor class allows to set the core and maximum pool size.The runnables that are run by a particular thread are executed sequentially.

**Executor Thread Pool Methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| newFixedThreadPool(int) | Creates a fixed size thread pool. |
| newCachedThreadPool() | Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available |
| newSingleThreadExecutor() | Creates a single thread. |

**Basic example of thread pool executor- FixedThreadPool.**

**Steps to be followed**

1. Create a task(Runnable Object) to execute

2. Create Executor Pool using Executors

3. Pass tasks to Executor Pool

4. Shutdown the Executor Pool

**public** **class** Test{

     // Maximum number of threads in thread pool

**static** **final** **int** MAX\_T = 3;

**public** **static** **void** main(String[] args)    {

        // creates five tasks

        Runnable r1 = **new** Task("task 1");

        Runnable r2 = **new** Task("task 2");

        Runnable r3 = **new** Task("task 3");

        Runnable r4 = **new** Task("task 4");

        Runnable r5 = **new** Task("task 5");

        // creates a thread pool with MAX\_T no. of

        // threads as the fixed pool size(Step 2)

        ExecutorService pool = Executors.newFixedThreadPool(MAX\_T);

        // passes the Task objects to the pool to execute (Step 3)

        pool.execute(r1);

        pool.execute(r2);

        pool.execute(r3);

        pool.execute(r4);

        pool.execute(r5);

        // pool shutdown ( Step 4)

        pool.shutdown();

    }

}

**class** Task **implements** Runnable  {

**private** String name;

**public** Task(String s)    {

        name = s;

    }

    // Prints task name and sleeps for 1s

    // This Whole process is repeated 5 times

**public** **void** run()

    {

**try**

        {

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

            {

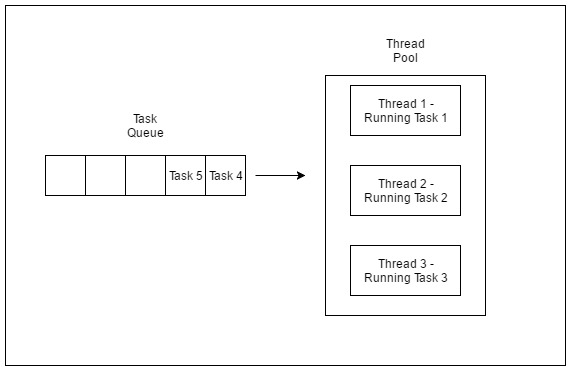
**if** (i==0)

…..

}}

}

As seen in the execution of the program, the task 4 or task 5 are executed only when a thread in the pool becomes idle. Until then, the extra tasks are placed in a queue.



One of the main advantages of using this approach is when you want to process 100 requests at a time, but do not want to create 100 Threads for the same, so as to reduce JVM overload. You can use this approach to create a ThreadPool of 10 Threads and you can submit 100 requests to this ThreadPool.

ThreadPool will create maximum of 10 threads to process 10 requests at a time. After process completion of any single Thread,

ThreadPool will internally allocate the 11th request to this Thread

and will keep on doing the same to all the remaining requests.

**Deadlock :** A deadlock is when two or more threads are blocked waiting to obtain locks that some of the other threads in the deadlock are holding. Deadlock can occur when multiple threads need the same locks, at the same time, but obtain them in different order.

**Example :** thread 1 locks A, and tries to lock B, and thread 2 has already locked B, and tries to lock A, a deadlock arises. Thread 1 can never get B, and thread 2 can never get A. In addition, neither of them will ever know. They will remain blocked on each their object, A and B, forever. This situation is a deadlock.

The situation is illustrated below:

Thread 1 locks A, waits for B

Thread 2 locks B, waits for A

Here is an example of a TreeNode class that call synchronized methods in different instances:

public class TreeNode {

TreeNode parent = null;

List children = new ArrayList();

public synchronized void addChild(TreeNode child){

if(!this.children.contains(child)) {

this.children.add(child);

child.setParentOnly(this);

}

}

public synchronized void addChildOnly(TreeNode child){

if(!this.children.contains(child){

this.children.add(child);

}

}

public synchronized void setParent(TreeNode parent){

this.parent = parent;

parent.addChildOnly(this);

}

public synchronized void setParentOnly(TreeNode parent){

this.parent = parent;

}

}

If a thread (1) calls the parent.addChild(child) method at the same time as another thread (2) calls the child.setParent(parent) method, on the same parent and child instances, a deadlock can occur. Here is some pseudo code that illustrates this:

Thread 1: parent.addChild(child); //locks parent

--> child.setParentOnly(parent);

Thread 2: child.setParent(parent); //locks child

--> parent.addChildOnly()

First thread 1 calls parent.addChild(child). Since addChild() is synchronized thread 1 effectively locks the parent object for access from other treads.

Then thread 2 calls child.setParent(parent). Since setParent() is synchronized thread 2 effectively locks the child object for acces from other threads.

**More Complicated Deadlocks**

Deadlock can also include more than two threads. This makes it harder to detect. Here is an example in which four threads have deadlocked:

Thread 1 locks A, waits for B

Thread 2 locks B, waits for C

Thread 3 locks C, waits for D

Thread 4 locks D, waits for A

Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.

**Database Deadlocks :**

A more complicated situation in which deadlocks can occur, is a database transaction. A database transaction may consist of many SQL update requests. When a record is updated during a transaction, that record is locked for updates from other transactions, until the first transaction completes. Each update request within the same transaction may therefore lock some records in the database.

If multiple transactions are running at the same time that need to update the same records, there is a risk of them ending up in a deadlock.

**For example :**

Transaction 1, request 1, locks record 1 for update

Transaction 2, request 1, locks record 2 for update

Transaction 1, request 2, tries to lock record 2 for update.

Transaction 2, request 2, tries to lock record 1 for update.

**Example :**

**class** Shared{

    // first synchronized method

**synchronized** **void** test1(Shared s2){

        System.out.println("test1-begin");

        Util.sleep(1000);

        // taking object lock of s2 enters

        // into test2 method

        s2.test2();

        System.out.println("test1-end");

    }

    // second synchronized method

**synchronized** **void** test2()    {

        System.out.println("test2-begin");

        Util.sleep(1000);

        // taking object lock of s1 enters

        // into test1 method

        System.out.println("test2-end");

    }

}

**DeadLock Prevention :**

In Java, a deadlock is a programming situation where two or more threads are blocked forever. A deadlock condition will occur with at least two threads and two or more resources.

**How To Avoid Deadlock**

* **Avoid Nested Locks:** A deadlock mainly happens when we give locks to multiple threads. Avoid giving a lock to multiple threads if we already have given to one.
* **Avoid Unnecessary Locks:** We can have a lock only those members which are required. Having a lock unnecessarily can lead to a deadlock.
* **Using Thread.join():** A deadlock condition appears when one thread is waiting other to finish. If this condition occurs we can use Thread.join() with the maximum time the execution will take.

**Daemon thread in Java :** Daemon thread is a low priority thread that runs in background to perform tasks such as garbage collection.

The main thread cannot be set as daemon thread. Because a thread can be set daemon before its running and as soon as the program starts the main thread starts running and hence cannot be set as daemon thread.

public final void setDaemon(boolean on)

Marks this thread as either a daemon thread or a user thread. The Java Virtual Machine exits when the only threads running are all daemon threads.

This method must be invoked before the thread is started.

**Properties:**

* They can not prevent the JVM from exiting when all the user threads finish their execution.
* JVM terminates itself when all user threads finish their execution
* If JVM finds running daemon thread, it terminates the thread and after that shutdown itself. JVM does not care whether Daemon thread is running or not.
* It is an utmost low priority thread.

****Methods:****

**void setDaemon(boolean status):** This method is used to mark the current thread as daemon thread or user thread. For example if I have a user thread tU then tU.setDaemon(true) would make it Daemon thread. On the other hand if I have a Daemon thread tD then by calling tD.setDaemon(false) would make it user thread.

**public final void setDaemon(boolean on)**

**boolean isDaemon():** This method is used to check that current is daemon. It returns true if the thread is Daemon else it returns false.

**public final boolean isDaemon()**

**What is difference between user thread and daemon thread?**

Java offers two types of threads: **user threads** and **daemon threads**. User threads are high-priority threads. The JVM will wait for any user thread to complete its task before terminating it. On the other hand, daemon threads are low-priority threads whose only role is to provide services to user threads.

**Is Garbage Collector A daemon thread?**

Java Garbage Collector runs as a Daemon Thread (i.e. a low priority thread that runs in the background to provide services to user threads or perform JVM tasks).

**Exceptions in Daemon thread :** If you call the setDaemon() method after starting the thread, it would throw IllegalThreadStateException.

**Java ExecutorService :**