### Threads

**Threads** are light weight process.

we can create multiple threads in java, even if you don't create any thread, one thread at least do exist. i.e. main thread.

**Process**

A program in execution is often referred as Process.

It has its own address space, a call stack and link to any resources.

A computer system normally has multiple processes running at a time.

The OS keeps track of all these processes and facilities their execution by sharing the processing time of the CPU among them.

One process can have multiple threads.

In process, all threads share system resource like heap memory.

Processes are instances of programs which typically run independent to each other, e.g. if you start a java program the operating system spawns a new process which runs in parallel to other programs. Inside those processes we can utilize threads to execute code concurrently, so we can make the most out of the available cores of the CPU.

**Thread**

Thread is a path of execution within the process.

Every process has at least one thread called main thread.

The main thread can create additional threads within the process.

Threads within a process share process resources including memory and open files. However, each thread has its own call stack.

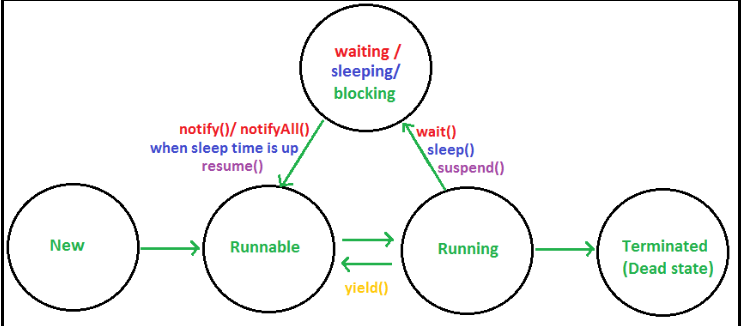
Since threads shares the same address space of the process, creating new threads and communicating between them is more efficient.

**Exception thrown by one thread will not affect another thread, as all threads are independent and have different stack.**

**When we call start() method on thread, it internally calls run() method with newly created thread. So, if we don’t override run() method newly created thread won’t be called and nothing will happen**

**Life Cycle of Thread**

New --> Runnable --> Running --> Terminated



**New State**

A thread that has not yet started is in this state.

- When instance of thread is created using new operator, it is in new state, but start() method has not been invoked on thread yet, thread is not eligible to run

- Thread object is considered alive, but thread is not alive yet.

**Runnable State**

- When start() method is called on thread it enters **runnable state**.

- As soon as Thread enters runnable state it is eligible to run, but not running.

A Thread executing in the Java Virtual Machine is in this state.

**Running State**

- Thread Scheduler selects the thread to go from runnable state to running state. In running state Thread starts executing by run () method.

**Blocked**

A thread that is blocked waiting for a monitor lock is in this state.

**Waiting**

A thread is waiting indefinitely for another thread to perform a particular action is in this state.

**Time-Waiting**

A thread is waiting with a specified waiting time.

**Terminated**

A thread is considered dead when its run () method completes. Once a thread is dead, it cannot be started again doing so will throw RuntimeException i.e. IllegalThreadStateException.

**public** **class** ThreadLifeCycle **implements** Runnable {

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

System.***out***.println(Thread.*currentThread*().getState());// RUNNABLE State

System.***out***.println("run2:"+Thread.*currentThread*().getState()); // RUNNABLE State

}

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

ThreadLifeCycle tlc = **new** ThreadLifeCycle();

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

System.***out***.println(t.getState()); // NEW State

t.start();

System.***out***.println(t.getState()); // RUNNABLE State

Thread.*sleep*(5000);

System.***out***.println(t.getState()); //TERMINATED State

}

}

Output:

NEW

RUNNABLE

RUNNABLE

run2:RUNNABLE

TERMINATED

**destroy()**

destroy() method put thread directly into dead state.

**start()**

When start() method is used a new thread is created and code inside run() method is executed.

**run()**

When run() method is called directly no new thread is created and code inside run() will execute on current thread.

**Sleep() method**

Currently executing thread object pauses for certain time. Once the specified time is getting over, then the same thread will continue its execution.

When sleep() method is called on thread it goes from running to waiting state and can return to runnable state when sleep time is up.

But **sleep() method must catch or throw compile time exception** i.e. interrupted exception.

But sleep() method causes the currently executing thread to sleep for the specified number of milliseconds.

**yield() method**

**yield means** give up or to surrender.

**Currently executing thread object pauses temporarily** and **allows other thread to execute**.

**If there is no waiting thread, then the same thread will continue its execution**.

When yield() method is called on thread it goes from running into runnable state, not in waiting state. Thread is eligible to run but not running and can be picked by scheduler at any time.

yield() method **need not to catch or throw any exception**.

yield() method stops thread for unpredictable time, that depends on thread scheduler.

**Definition** : yield() method when called on thread gives a hint to the thread scheduler that the current thread is willing to yield its current use of a processor. The thread scheduler is free to ignore this hint.

**Waiting time**: yield() method stops thread for unpredictable time.

**Static method**: yield() is a static method, hence calling Thread.yield() **causes currently executing thread to yield**.

**synchronized block**: **thread need not to acquire object lock before calling yield() method** i.e. **yield() method can be called from outside synchronized block**.

**join()**

The join() method is **used to pause the execution of currently running thread until specified thread is dead(finished execution).**

Using join() method - We make sure that **the threads executes in a particular order**.

**We generally have more than one thread, thread does not have guarantee the order of execution of threads**.

scenario :

1. We have created and started the Thread T1
2. We call T1.join(), this means that’s main thread continues only after thread T1 has finished the execution.
3. Again, we have created and started the Thread T2 and T2 will execute after the completion of T1. It will not get the control in the thread T2.

**import** java.util.Date;

**public** **class** ThreadExampleJoin **implements** Runnable{

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

**try** {

System.***out***.println(Thread.*currentThread*().getName()+":"+"run method startts");

Thread.*sleep*(3000);

System.***out***.println(Thread.*currentThread*().getName()+":"+"run method ends");

}**catch**(Exception e) { System.***out***.println("run Exception"+e); }

}

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

ThreadExampleJoin te = **new** ThreadExampleJoin();

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

t.start();

Thread.*sleep*(100);

System.***out***.println(**new** Date() +":"+"Main Thread continues");

Thread t2 = **new** Thread(te);

System.***out***.println(**new** Date() +":"+"Create Thread T2");

t2.start();

System.***out***.println(**new** Date() +":"+"Main Thread continues after t2.start");

t2.join();

//t.join();

System.***out***.println(**new** Date() +":"+"Main Thread continues after t2.join");

}

}

**Output:**

Thread-0:run method startts

Sat Mar 14 12:41:06 IST 2020:Main Thread continues

Sat Mar 14 12:41:06 IST 2020:Create Thread T2

Sat Mar 14 12:41:06 IST 2020:Main Thread continues after t2.start

Thread-1:run method startts

Thread-0:run method ends

Thread-1:run method ends

Sat Mar 14 12:41:09 IST 2020:Main Thread continues after t2.join

**Race Condition**

When **multiple threads try to read and write a shared variable concurrently** and **these read and write operations overlap in execution**, then **final outcome depends on the order in which read and write takes place**, **which is unpredictable.** This phenomenon is called **Race Condition**.

**class** Counter {

**int** count = 0;

**public** **void** increment() {

count = count + 1;

}

**public** **int** getCount() {

**return** count;

}

}

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.TimeUnit;

**public** **class** RaceConditionExample {

//When multiple threads try to read and write a shared variable concurrently, and these read and write operations overlap in execution, then the final outcome depends on the order in which the reads and writes take place, which is unpredictable. This phenomenon is called Race condition.

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

ExecutorService executorService = Executors.*newFixedThreadPool*(10);

Counter counter = **new** Counter();

**for**(**int** i = 0; i < 1000; i++) {

executorService.submit(() -> counter.increment());

}

executorService.shutdown();

executorService.awaitTermination(60, TimeUnit.***SECONDS***);

System.***out***.println("Final count is : " + counter.getCount());

}

}

**Output:**

Final count is : 997

**Deadlock**

Deadlock is a **condition that occurs when two threads are waiting for each other to release the resources**.

Deadlock **can be easily avoided by resource ordering**.

The real reason for deadlock is not multiple threads, but **it is the way that they are requesting lock**. **If you provide an order access, then the problem will be resolved.**

With this technique, assign order on all the objects whose locks must be acquired and ensure that the locks are acquired in that order.

**Acquire lock on resources in a particular order** and **then releasing acquired lock in reverse order** **won’t create any deadlock.**

**Deadlock Example**

**class** A **implements** Runnable {

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

**synchronized**(A.**class**) {

System.***out***.println("Acquired the A lock");

**synchronized**(B.**class**) {

System.***out***.println("Internally acquires the B lock in the A.class");

}}}}

**class** B **implements** Runnable {

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

**synchronized**(B.**class**) {

System.***out***.println("Acquired the B lock");

**synchronized**(A.**class**) {

System.***out***.println("Internally acquires the A lock in the B.class");

}}}}

**public** **class** DeadLockExample {

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

Thread a = **new** Thread(**new** A());

Thread b = **new** Thread(**new** B());

a.start();

b.start();

}}

Output :

Acquired the A lock

Acquired the B lock

**DeadLockFixedExample**

**class** C **implements** Runnable {

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

**synchronized**(C.**class**) {

System.***out***.println("Acquired the A lock");

**synchronized**(D.**class**) {

System.***out***.println("Internally acquires the B lock in the A.class");

}}}}

**class** D **implements** Runnable {

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

**synchronized**(C.**class**) {

System.***out***.println("Acquired the B lock");

**synchronized**(D.**class**) {

System.***out***.println("Internally acquires the A lock in the B.class");

}}}}

**public** **class** DeadLockFixedExample {

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

Thread c = **new** Thread(**new** C());

Thread d = **new** Thread(**new** D());

c.start();

d.start();

}

}

**Output:**

Acquired the A lock

Internally acquires the B lock in the A.class

Acquired the B lock

Internally acquires the A lock in the B.class

**Daemon Thread**

The Daemon threads are low priority threads that provides the background support and services to the user threads.

The JVM does not wait for Daemon thread before exiting.

While it waits for user threads, it does not exist until unless all the user threads finish their execution.

**public** **class** DaemonThreadExample **implements** Runnable {

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

**try** {

**if** (Thread.*currentThread*().isDaemon()) {

System.***out***.println(Thread.*currentThread*().getName() + " is Daemon thread");

} **else** {

System.***out***.println(Thread.*currentThread*().getName() + " is User thread");

}

} **catch** (Exception e) {

System.***out***.println("run Exception" + e);

}

}

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

DaemonThreadExample te = **new** DaemonThreadExample();

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

Thread t1 = **new** Thread(te);

Thread daemon = **new** Thread(() -> System.***out***.println("Hello from daemon!"));

daemon.setDaemon(**true**);

daemon.start();

t.setDaemon(**true**);

t.start();

t1.start();

// System.out.println(new Date() +":"+"Main Thread Ends");

}

}

Output :

Hello from daemon!

Thread-1 is User thread

Thread-0 is Daemon thread

**Synchronized**

At times, **when more than one thread try to access a shared resource**, **we need to ensure that resource will be used by only one thread at a time.** The process by which this is achieved is called synchronization.

Synchronization **guarantees that no two threads can execute synchronize method.**

Whenever **thread enters into synchronized method** or block, **it acquires a lock** and **whenever it leaves synchronized method or lock, it releases a lock**.

**synchronized keyword can be used on different levels.**

1. non-static method (instance method)

2. static method

3. Code blocks

**non static method (Object Level lock)**

If you declare any method is synchronized, it is known as synchronized method.

Synchronized method is **used to lock an object for any shared resources**.

**One thread will execute code block on given instance of the class**. **This should always be done to make instance level thread safe**.

**static method - Static synchronization (Class level lock)**

**If you make any static method as synchronized**, **the lock will be on the class not object**.

* **Class level locking make sure static data to be thread safe.**
* **Class level lock prevent multiple threads to enter into synchronized in any of all available instance of the class on run time. This means if runtime there are 100 instances of same class, only one threads will be able to execute**
* synchronized method in any one of instance at time, and all other instances will be locked for other threads.

**Static Synchronized method** synchronize on the class object.

**Non-Static synchronized method** synchronize on the instance of class.

**Synchronized block**

Synchronized block can be **used to perform synchronization on specific resource of the method**.

**Suppose we have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block**.

Synchronized block is used to lock an object for any shared resource.

**Scope of synchronization block is smaller than the method.**

All code which goes into block is synchronized on the current object.

Synchronized block takes on argument and it is called mutex.

If the synchronized block is defined inside non-static method, instance initializer or constructors, then this mutex must be instance of that class.

If the synchronized block is defined inside static method, then this mutext must be like ClassName.class

class Shared{

static void staticMethod(){

synchronized(Shared.class){

// static synchronized block

}

}

void nonstaticmethod(){

synchronized(this){

//non static synchronized block

}

}

void anotherstaticmethod(){

synchronized(new shared()){

}

}

**Class Level Synchronization**

**class** Table {

**static** **synchronized** **void** printTable(**int** n) {// synchronized method

**for** (**int** i = 1; i <= 3; i++) {

System.***out***.println(Thread.*currentThread*().getName() + "--> n: " + n + " i: " + i + " Result: " + n \* i);

**try** {

Thread.*sleep*(400);

} **catch** (Exception e) {

System.***out***.println(e);

}

}

}

**static** **synchronized** **void** printTableA(**int** n) {// synchronized method

**for** (**int** i = 1; i <= 3; i++) {

System.***out***.println(Thread.*currentThread*().getName() + "--> n: " + n + " i: " + i + " Result: " + n \* i);

**try** {

Thread.*sleep*(400);

} **catch** (Exception e) {

System.***out***.println(e);

}

}

}

}

**class** Table2 {

**static** **synchronized** **void** printTableB(**int** n) {// synchronized method

**for** (**int** i = 1; i <= 3; i++) {

System.***out***.println(Thread.*currentThread*().getName() + "--> n: " + n + " i: " + i + " Result: " + n \* i);

**try** {

Thread.*sleep*(400);

} **catch** (Exception e) {

System.***out***.println(e);

}

}

}

}

**class** MyThread1 **extends** Thread {

Table t;

MyThread1(Table t) {

**this**.t = t;

}

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

t.*printTable*(5);

}

}

**class** MyThread2 **extends** Thread {

Table t;

MyThread2(Table t) {

**this**.t = t;

}

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

t.*printTableA*(100);

}

}

**class** MyThread3 **extends** Thread {

Table2 t;

MyThread3(Table2 t) {

**this**.t = t;

}

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

t.*printTableB*(100);

}

}

**public** **class** ClassLevelSynchronization {

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

Table obj = **new** Table();// only one object

Table2 obj2 = **new** Table2();// only one object

MyThread1 t1 = **new** MyThread1(obj);

MyThread2 t2 = **new** MyThread2(obj);

MyThread3 t3 = **new** MyThread3(obj2);

t1.start();

t2.start();

t3.start();

}

}

Output:

Thread-0--> n: 5 i: 1 Result: 5

Thread-2--> n: 100 i: 1 Result: 100

Thread-2--> n: 100 i: 2 Result: 200

Thread-0--> n: 5 i: 2 Result: 10

Thread-2--> n: 100 i: 3 Result: 300

Thread-0--> n: 5 i: 3 Result: 15

Thread-1--> n: 100 i: 1 Result: 100

Thread-1--> n: 100 i: 2 Result: 200

Thread-1--> n: 100 i: 3 Result: 300

**Object Level Synchronization**

**class** Table {

**static** **synchronized** **void** printTable(**int** n) {// synchronized method

**for** (**int** i = 1; i <= 3; i++) {

System.***out***.println(Thread.*currentThread*().getName() + "--> n: " + n + " i: " + i + " Result: " + n \* i);

**try** {

Thread.*sleep*(400);

} **catch** (Exception e) {

System.***out***.println(e);

}

}

}

}

**class** MyThread1 **extends** Thread {

Table t;

MyThread1(Table t) {

**this**.t = t;

}

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

t.*printTable*(5);

}

}

**class** MyThread2 **extends** Thread {

Table t;

MyThread2(Table t) {

**this**.t = t;

}

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

t.*printTable*(100);

}

}

**public** **class** ObjectLevelSynchronization {

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

Table obj = **new** Table();// only one object

Table obj2 = **new** Table();

MyThread1 t1 = **new** MyThread1(obj);

// MyThread2 t2=new MyThread2(obj);

MyThread2 t2 = **new** MyThread2(obj2);

t1.start();

t2.start();

}

}

**Output :**

Thread-0--> n: 5 i: 1 Result: 5

Thread-0--> n: 5 i: 2 Result: 10

Thread-0--> n: 5 i: 3 Result: 15

Thread-1--> n: 100 i: 1 Result: 100

Thread-1--> n: 100 i: 2 Result: 200

Thread-1--> n: 100 i: 3 Result: 300

**Static Synchronized Method**

**class** Table {

**static** **synchronized** **void** printTable(**int** n) {// synchronized method

**for** (**int** i = 1; i <= 3; i++) {

System.***out***.println(Thread.*currentThread*().getName() + "--> n: " + n + " i: " + i + " Result: " + n \* i);

}

}

**static** **synchronized** **void** printTableA(**int** n) {// synchronized method

**for** (**int** i = 1; i <= 3; i++) {

System.***out***.println(Thread.*currentThread*().getName() + "--> n: " + n + " i: " + i + " Result: " + n \* i);

}

}

**static** **synchronized** **void** printTableB(**int** n) {// synchronized method

**for** (**int** i = 1; i <= 3; i++) {

System.***out***.println(Thread.*currentThread*().getName() + "--> n: " + n + " i: " + i + " Result: " + n \* i);

}

}

}

**public** **class** StaticSynchronizedMethod {

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

Table obj = **new** Table();// only one object

Thread t1 = **new** Thread(() -> {

obj.*printTable*(5);

});

Thread t2 = **new** Thread(() -> {

obj.*printTableA*(10);

});

Thread t3 = **new** Thread(() -> {

obj.*printTableB*(100);

});

t1.start();

t2.start();

t3.start();

}

}

Output:

Thread-0--> n: 5 i: 1 Result: 5

Thread-0--> n: 5 i: 2 Result: 10

Thread-0--> n: 5 i: 3 Result: 15

Thread-2--> n: 100 i: 1 Result: 100

Thread-2--> n: 100 i: 2 Result: 200

Thread-2--> n: 100 i: 3 Result: 300

Thread-1--> n: 10 i: 1 Result: 10

Thread-1--> n: 10 i: 2 Result: 20

Thread-1--> n: 10 i: 3 Result: 30