Multithreading

Process:

* an instance of a program/application loaded into RAM.
* 
* Suppose, if you execute a Java project, if you start a browser and if you start a music player in your system, then Operating system creates 3 processes and will be stored into RAM.
* OS comes with a schedular, which will provide CPU time for a process to run on a CPU core.
* The OS schedular will provide a time slice, approximately 100ms for each process, if they are are executing on a single cpu core.



* A process from creation to completion, it goes into different states.
* These states represent the process life cycle.
* 1. New 2. Ready 3. Running 4. Waiting 5. Dead



Thread:

* Each process contains atleast one thread, called main thread.
* If a process has multiple independent tasks, then we can create multiple theads in a process, where each thread will perform a task.
* If threads are not created then the tasks are executed sequentially on a single cpu core, hence the process completion time will be more.
* If threads are created then the tasks are parallelly executed on different CPU cores and hence the process is completed fastly. The performance of an application will be increased.
* In a Java program, if you want to execute mulitple tasks parallelly then you have to use multithreading.
* For example, a Java program wants to insert the records into the database and also wants to generate the reports. In this case we can create a thread for inserting the records and other thread for generating the reports.
* A thread is a group of statements, which are executed independently from the rest of the program execution.

creating threads:

* There are 2 ways to create a thread.
1. by extending Thread class
2. by implementing Runnable interface
* The logic of a thread should be defined in run() method.

ex:

 class MyThread extends Thread {

 @Override

 public void run() {

 //define the logic here

 }

 }

 (or)

 class MyThread2 implements Runnable {

 @Override

 public void run() {

 //define the logic here

 }

 }

* If a thread is created by extending Thread class, then it can be started directly.

MyThread t1 = new MyThread();

t1.start();

* If a thread is cretaed by implementing Runnable interface, then it can’t be started directly,
* First wrap the the Runnable object into Thread object, then start the thread.

MyThread2 t2 = new MyThread2(); // Runnable oject

t2.start(); //compile-time error.

Thread t3 = new Thread(t2); // wrapping

t3.start(); //correct

**package** com.ashokit.thread;

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

 @Override

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

 **for**(**int** i = 1; i <= 10; i++ ) {

 System.***out***.println("i = " + i);

 **try** {

 Thread.*sleep*(2000); //2 sec

 }

 **catch**(InterruptedException ex) {

 ex.printStackTrace();

 }

 }

 }

}

**class** MyThread2 **implements** Runnable{

 @Override

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

 **for**(**int** j = 11; j <= 20; j++ ) {

 System.***out***.println("j = " + j);

 **try** {

 Thread.*sleep*(2000); //2 sec

 }

 **catch**(InterruptedException ex) {

 ex.printStackTrace();

 }

 }

 }

}

**public** **class** Main {

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

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

 //wrapping Runnable object into a Thread object

 //we are wrapping, because Runnable object

 //doesn't have a start() method.

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

 t1.start();

 t2.start();

 }

}

Q) can we start a thread for twice, by calling start() method?

A) if we write,

 t1.start();

 t1.start();

 then

 we won’t get any compile-time error, but at runtime, IllegalThreadStateException will be thrown.

Q) can we call run() method of a thread directly?

A) Yes. But the threads will execute sequentially, but not parallelly.

Q) which way is better to create a thread, either extending Thread class

 or implmenting Runnable interface?

A) implementing Runnable interface is better approach, because still a thread class can inherit another parent class.

Thread life cycle:

* From object creation to termination, a thread under goes with different states. This is called life cycle.
* A thread life cycls has 5 states/stages.
* 1. newborn state
* 2. runnable state
* 3. running state
* 4. non-runnable state
* 5. dead state



* when a thread object is created then the thread enters into newborn state.
* when start() method is called on a thread object then the thread is moved to the Runnable state.
* mulitple threads can sit in Runnable state at a time.
* Thread schedular will pick up one thread from Runnable state and moved

it into Running state.

* In Running state, run() method is called.
* When the time slice is lapsed/when yield() method is called, a thread is moved back from Running to Runnable state by the schedular.
* when sleep() is called, a thread is moved from Running state to the non-runnable state.
* once sleep time is completed, a thread is moved to Runnable state.
* when wait() is called, a thread is moved from Running state to non-runnable state.
* when notify()/notifyAll() method is called, then a waiting thread goes to Runnable state.
* If a thread’s execution is finished, then it goes to dead state.

yield() method:

--------------

 yield() method tells the schedular that a thread is willing to come out of Running state and it wants to give chance to other thread to enter into running state.

 yield() is a static method, so it can be called with classname.

 Thread.yield();

**package** com.ashokit.thread;

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

 @Override

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

 //Thread.currentThread() returns a current thread reference

 //getName() returns the name of the current thread.

 System.***out***.println("Inside " + Thread.*currentThread*().getName());

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

 System.***out***.println("i = "+i);

 Thread.*yield*();

 }

 System.***out***.println( Thread.*currentThread*().getName() + " finished");

 }

}

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

 @Override

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

 System.***out***.println("Inside " + Thread.*currentThread*().getName());

 **for**(**int** j=1; j<=5; j++) {

 System.***out***.println("j = "+j);

 Thread.*yield*();

 }

 System.***out***.println( Thread.*currentThread*().getName() + " finished");

 }

}

**public** **class** MainClass {

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

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

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

 t1.start();

 t2.start();

 }

}

Thread priority:

* A thread priority is an integer value and its range is from 1 to 10.
* A thread priority is an indication to the schedular that which thread should get running state first. But it depends on thread schedular.
* The default priority of a thread is 5.
* In Thread class, 3 class level constants are defined to indicate the mininum, normal and the maximum priority that a thread can have.
* public static final int MIN\_PRIORITY = 1;
* public static final int NORM\_PRIORITY = 5;
* public static final int MAX\_PRIORITY = 10;
* a thread priority can be changed by calling setPriority() and can be retrieved by calling getPriority().
* if we set the priority for a thread < 1 or > 10 then at runtime, IllegalArgumentException will be thrown.

t1.setPriority(Thread.MAX\_PRIORIRY);

t2.setPriority(11); //exception

joining the threads:

* If one thread wants to wait until the other threads execution is completed then we need to join the threads.
* If t1 and t2 are the threads, and if we join t1 with t2 then t2 will wait until t1’s execution is completed.
* For example, thread1 is generating a report and thread2 is proofreading a report, then thread2 can proofread after generating the report. So thread2 has to wait until thread1 has generated the report. For this, we can join thread1 with thread2.

//example1

/\*

 \* This example has two threads, main thread and other thread

 \* we have joined other thread with the main thread

 \* the main thread execution is suspended until the other thread

 \* execution is finished.

 \*/

**package** com.ashokit.thread;

**class** MyThread **extends** Thread {

 @Override

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

 System.***out***.println("Inside " +Thread.*currentThread*().getName());

 **for**(**int** i = 1; i <=50; i++) {

 System.***out***.println(" i = "+i);

 **try** {

 Thread.*sleep*(2000);

 }

 **catch**(InterruptedException ex) {

 ex.printStackTrace();

 }

 }

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

 }

}

**public** **class** MainClass {

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

 Thread.*currentThread*().setName("Main Thread");

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

 MyThread t1 = **new** MyThread();

 t1.setName("Other Thread");

 t1.start();

 **for**(**int** j = 51; j <= 70; j++ ) {

 System.***out***.println(" j = "+j);

 **if**(j==55) {

 t1.join();

 }

 Thread.*sleep*(2000);

 }

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

 }

}

//example2

/\*

 \* This program has three threads, main thread, other thread1 and other thread2

 \* The other thread1 is joined with main thread and other thread2 is joined with

 \* other thread1.

 \*/

**package** com.ashokit.thread;

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

 MyThread2 t2;

 **public** MyThread1(MyThread2 t2) {

 **this**.t2 = t2;

 }

 @Override

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

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

 **for**(**int** i=1; i <= 10; i++) {

 System.***out***.println(" i = " + i);

 **try** {

 **if** ( i == 6 )

 {

 t2.join();

 }

 Thread.*sleep*(2000);

 }

 **catch**(Exception e) {

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

 }

 }

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

 }

}

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

 @Override

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

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

 **for**(**int** j=11; j <= 20; j++) {

 System.***out***.println(" j = " + j);

 **try** {

 Thread.*sleep*(5000);

 }

 **catch**(Exception e) {

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

 }

 }

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

 }

}

**public** **class** Solution {

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

 Thread.*currentThread*().setName("Main thread");

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

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

 t2.setName("Other thread2");

 MyThread1 t1 = **new** MyThread1(t2);

 t1.setName("Other thread1");

 t1.start();

 t2.start();

 **for** ( **int** k = 101; k <= 111; k++) {

 System.***out***.println("k = " + k);

 **if**( k == 105 ) {

 t1.join();

 }

 }

 }

}

Q) what is the difference between a process and a thread?

A) 1. process contains one or more threads.

 a thread does not contain processes.

 2.A process has a separate address space.

 A thread does not have a separate address space.

 3.process context switch is heavy. Because, cpu has to switch

 from one address space to another address space.

 thread context switch is light weight. Because, cpu no need to

 switch, as the threads are part of the same process.

 4.process to process communication requires additional resources

 like sockets.

 thread to thread communication does not require additional resources.

Threads synchronization:

* If multiple threads are using a shared resource and trying to read/modify the data at the same time then we will get unpredictable results.
* Suppose, two people are withdrawing the money from the same bank account at the same time, then there is a chance of overdrawing the money.
* For example, balance available is 5000.0, and the two people at a time can see the balance as 5000.0 and if they withdraw 3000.0 each, then the total amount withdrawn is 6000.0. This is overdrawing the amount and leads to inconsistency.
* Threads synchronization controls access to a shared resource by multiple threads, by allowing only one thread to act on the shared resource at the same time.
* Threads synchronization can be implemented using synchronized keyword.
* with synchronized keyword(non-access modifier) we can create synchronized methods or synchronized blocks.
* synchronization works by acquiring the lock on an object by a thread.
* In Java, every object has a lock/monitor.
* A thread acquires the lock on the object, by calling synchronized method or synchronized block.
* Only one thread can acquire the lock on an object at a time.
* If two threads are calling a synchronized method at the same time, then one of the two threads acquires lock on the object.
* when the execution of a synchronized method/block is completed by a thread then the lock gets released.

/\*

 \* In this example, Course is a shared resource for two threads

 \* The two threads are trying to register for the same course

 \* which has only one seat available.

 \* If synchronization is not applied, there is possibility that

 \* the two threads registered for the same seat.

 \* To avoid this inconsistency, we made the registerForCourse() method

 \* as synchronized.

 \* One thread at a time can execute the synchronized method, so that

 \* we can avoid the inconsitency.

 \*/

**package** com.ashokit.thread;

**class** Course {

 **private** String courseName;

 **private** **int** numOfSeatsAvailable;

 **public** Course(String courseName, **int** numOfSeatsAvailable) {

 **super**();

 **this**.courseName = courseName;

 **this**.numOfSeatsAvailable = numOfSeatsAvailable;

 }

 **public** **synchronized** **void** registerForCourse(**int** rollno) {

 **try** {

 **if** ( **this**.numOfSeatsAvailable - 1 < 0 ) {

 **throw** **new** Exception("Sorry, seats are not available!. Your rollno : " + rollno);

 }

 System.***out***.println("Booking successful!!! Your rollno is : " + rollno);

 numOfSeatsAvailable -= 1;

 System.***out***.println("Available seats now : " + **this**.numOfSeatsAvailable);

 }

 **catch**(Exception ex) {

 System.***out***.println("ERROR : " + ex.getMessage());

 }

 }

}

**class** RegisterThread **extends** Thread {

 Course course;

 **int** rollno;

 RegisterThread(Course course, **int** rollno) {

 **this**.course = course;

 **this**.rollno = rollno;

 }

 @Override

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

 course.registerForCourse(rollno);

 }

}

**public** **class** MainClass {

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

 Course course = **new** Course("CSE", 1);

 RegisterThread t1 = **new** RegisterThread(course, 101);

 RegisterThread t2 = **new** RegisterThread(course, 102);

 t1.start();

 t2.start();

 }

}

synchronized block:

* Suppose, if few statements of a method leads to inconsistency, if they are executed by multiple threads at the same time, but if we make entire method as synchronized then it will increase waiting time for the other threads.
* So, instead of creating synchronized method, we can define a synchronized block inside the method with only the statements that cause inconsistency and by keeping the remaining statements at outside of the synchronized block.

For ex:

 public void doWork() {

 statement1;

 statement2;

 synchronized(this) {

 statement3;

 statement4;

 statement5;

 }

 statement6;

 statement7;

 }

Q) synchronized method/block, which is better?

A) synchronized block, because it reduces waiting time for other threads.

volatile keyword:

* In Java, when a thread is executing on a CPU core, it allows a thread to cache variable for better performance.
* If there is a shared variable(static) for two threads, then the variable gets cached and the threads are performing read/write operations on the variable in cache.
* So, the changes made to the shared variable by one thread does not immediately reflect in the main memory. Hence, the other thread can’t access the updated value.
* To avoid this issue, volatile is provided and if we declare a variable as volatile, then a thread can not store it in cache and the threads can perform read/write operations directly on the shared variable in main memory.



Inter-thread communication:

* Inter-thread communication is possible between synchronized threads.
* If we want one synchronized thread to wait and wants to give chance to another synchronized thread to execute then we need inter-thread communication.
* For inter-thread communication, we have to use the following of the Object class.
1. wait() : moves current thread to waiting state, until another invokes notify()/notifyAll() method.
2. wait(millis) : moves current thread to waiting state until another thread invokes notify()/notifyAll() method, or time has lapsed.
3. notify(): notifies a single waiting thread, to resume
4. notifyAll(): notifies all waiting threads, to resume

/\*

 \* In this example, Course is a shared resource for three threads

 \* The two threads are trying to register for the same course

 \* which has only one seat available and the third thread

 \* is cancelling the seat.

 \* The one thread gets booking success, and the other thread waits to

 \* see for the cancellation of seat.

 \* The cancellation thread cancels the seat and notifies the waiting thread.

 \* so, the other thread also gets booking successful.

 \*

 \*/

**package** com.ashokit.thread;

**class** Course {

 **private** String courseName;

 **private** **int** numOfSeatsAvailable;

 **public** Course(String courseName, **int** numOfSeatsAvailable) {

 **super**();

 **this**.courseName = courseName;

 **this**.numOfSeatsAvailable = numOfSeatsAvailable;

 }

 **public** **synchronized** **void** registerForCourse(**int** rollno) {

 **try** {

 **if**(**this**.numOfSeatsAvailable - 1 < 0)

 wait();

 **if** ( **this**.numOfSeatsAvailable - 1 < 0 ) {

 **throw** **new** Exception("Sorry, seats are not available!. Your rollno : " + rollno);

 }

 System.***out***.println("Booking successful!!! Your rollno is : " + rollno);

 numOfSeatsAvailable -= 1;

 System.***out***.println("Available seats now : " + **this**.numOfSeatsAvailable);

 System.***out***.println("===================================================");

 }

 **catch**(Exception ex) {

 System.***out***.println("ERROR : " + ex.getMessage());

 }

 }

 **public** **synchronized** **void** cancelSeats() {

 **try** {

 **this**.numOfSeatsAvailable += 1;

 System.***out***.println("Cancellation Successful");

 System.***out***.println("Available seats now : " + **this**.numOfSeatsAvailable);

 System.***out***.println("==============================================");

 notify();

 }

 **catch**(Exception ex) {

 ex.printStackTrace();

 }

 }

}

**class** RegisterThread **extends** Thread {

 Course course;

 **int** rollno;

 RegisterThread(Course course, **int** rollno) {

 **this**.course = course;

 **this**.rollno = rollno;

 }

 @Override

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

 course.registerForCourse(rollno);

 }

}

**class** CancelThread **extends** Thread {

 Course course;

 CancelThread(Course course) {

 **this**.course = course;

 }

 @Override

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

 course.cancelSeats();

 }

}

**public** **class** MainClass {

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

 Course course = **new** Course("CSE", 1);

 RegisterThread t1 = **new** RegisterThread(course, 101);

 RegisterThread t2 = **new** RegisterThread(course, 102);

 t1.start();

 t2.start();

 **try** {

 Thread.*sleep*(8000);

 }

 **catch**(Exception ex) {

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

 }

 CancelThread t3 = **new** CancelThread(course);

 t3.start();

 }

}

Q) why wait(), notify() and notifyAll() methods are given in Object class, not in Thread class?

A) 1. When you synchronize on an object, the thread acquires lock

 associated with the object.

 2.In Inter-thread communication, the threads coordination happens by

 acquiring and releasing the locks on the object. So, the wait(),

 notify() and notifyAll() are ties to the lock of the object, not the

 thread.

1. That’s why these methods are given in Object class, but not in Thread class.

Deadlock:

* Suppose, we have two threads and two resources.
* thread1 holds the lock on resource1 and waiting for lock on resource2 and thread2 holds the lock on resource2 and waiting for lock on resource1.
* In this case, thread1 is blocked for resource2 and thread2 is blocked for resource1.
* So, the two threads will never complete. This is called deadlock situation.
* Deadlocks can be prevented by making the two threads to acquire the locks on the resources in the same order.

**package** com.ashokit.thread;

**public** **class** MainClass {

 **static** **final** Object ***res1*** =**new** Object();

 **static** **final** Object ***res2*** = **new** Object();

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

 Runnable task1 = () -> {

 **synchronized**(***res1***) {

 System.***out***.println(Thread.*currentThread*().getName() +" has acquired lock on resource1");

 **try** {

 Thread.*sleep*(2000);

 }

 **catch**(InterruptedException ie) {

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

 }

 System.***out***.println(Thread.*currentThread*().getName()+ " is waiting to acquire lock on resource2");

 **synchronized**(***res2***) {

 System.***out***.println(Thread.*currentThread*().getName() +" has acquired lock on resource2");

 }

 }

 System.***out***.println(Thread.*currentThread*().getName()+" : is completed");

 };

 Runnable task2 = () -> {

 **synchronized**(***res2***) {

 System.***out***.println(Thread.*currentThread*().getName() +" has acquired lock on resource2");

 **try** {

 Thread.*sleep*(2000);

 }

 **catch**(InterruptedException ie) {

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

 }

 System.***out***.println(Thread.*currentThread*().getName()+ " is waiting to acquire lock on resource1");

 **synchronized**(***res1***) {

 System.***out***.println(Thread.*currentThread*().getName() +" has acquired lock on resource1");

 }

 }

 System.***out***.println(Thread.*currentThread*().getName()+": is completed");

 };

 Thread t1 = **new** Thread(task1, "Thread1");

 Thread t2 = **new** Thread(task2, "Thread2");

 t1.start();

 t2.start();

 }

}

Q) what are the types of threads?

A) 2 types of threads

 1. user threads

 2. daemon threads

\* by default, each thread is a user thread.

\* Main thread will not terminate the currently running JVM, until user threads are completed.

\* If make any thread as daemon thread, then Main thread does not wait for the completion of deamon thread.

\* If all the user threads in application are completed, then main thread will terminate the JVM. So, the deamon threads are also terminated.

\* setDeamon() method can make a thread as daemon thread.

\* isDeamon() method can be used to verify that a thread is daemon thread or user thread.

\* we create daemon threads in an application, to perform any background activities. For example, performance monitoring of an application.

\* In JVM, Garbage Collector thread is a daemon thread and it executes continously until JVM is terminated.

//Daemon thread demo

**package** com.ashokit.thread;

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

 @Override

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

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

 **for**(**int** i=1; i<=30; i++) {

 System.***out***.println("i = " + i);

 **try** {

 Thread.*sleep*(2000);

 }

 **catch**(InterruptedException ex) {

 System.***out***.println("Error : " + ex.getMessage());

 }

 }

 System.***out***.println(Thread.*currentThread*().getName() + " : is finished");

 }

}

**public** **class** MainClass {

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

 Thread.*currentThread*().setName("Main Thread");

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

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

 t1.setName("Daemon Thread");

 t1.setDaemon(**true**);

 t1.start();

 **for**(**int** j=1; j<=10; j++) {

 System.***out***.println("j = " + j);

 **try** {

 Thread.*sleep*(2000);

 }

 **catch**(InterruptedException ex) {

 System.***out***.println("Error : " + ex.getMessage());

 }

 }

 System.***out***.println(Thread.*currentThread*().getName() + " : is finished");

 }

}