**MULTI – THREADING**

Consider the following program

class Sample

{

fun1()

{}

fun2()

{}

fun3()

{}

public static void main(String args[])

{

Sample s = new Sample();

s.fun1();

s.fun2();

s.fun3();

}

}

Normally, if a class is compiled and JVM executes the program, then the order of execution is that, first fun1() is called and after execution of fun1(), control comes back to the main() and then fun2() is called and so on.

Now let’s assume that fun1() has some statements which involves data transfer. We know that data transfer is not the job of processor. It is the job of a separate individual circuit, which functions under the control of the processor.

Since fun1() has data transfer statements, processor assigns that job to the circuit. During this time the processor would remain idle.

The state of CPU where CPU waits for the circuit to transfer the data is known as idle state.

To increase the efficiency of the processor, it is required that processor do some useful work during this idle state. The useful work is nothing but, executing other functions present in the program. This need led to the concept of the multi – tasking.

**Multi – tasking**

The concept of executing multiple functionalities simultaneously is known as multi – tasking.

If we apply the concept of multi – tasking to our program, the control flow would be as follows:

A part of fun1() will be executed and then control switches to function fun2(). Now, here also a part of fun2() is executed and then control switches to fun3(). A part of fun3() is executed and then control again comes back to fun1(). Now it continues executing fun1() from where it had stopped earlier. Similarly for fun2() and fun3(). (The sequence may change).

**Thus, concept of multi – tasking came into existence to avoid the idle state of CPU.**

In multi – tasking we said that, **a part of** a functionality is executed one at a time. Now, the question is part means how many statements.

This problem is resolved by the concept of scheduling.

**Scheduling**

It is the process in which a specific time is allocated to functionality where the control remains in that particular functionality for that specific time period.

Once the time period is lapsed, control will switches to another functionality with another time slice and so on.

Scheduling is supervised by the scheduler.

The time slice allocated will be in the order of nano seconds.

Now it is clear that, without applying multi – tasking one function will depend upon another function i.e. after execution of fun1(), fun2() will be executed and so on.

**Advantages of multi – tasking**

1. Multi – tasking avoids idle state of CPU i.e. it decreases the response time.
2. It is used to make the functions gets executed independently.

**Note**

Actually, in real time we use multi – tasking to make the most of the second advantage rather than the first, because in case of small projects, idle state of CPU is not a big concern.

**Multi – tasking is of two types**

1. Thread based multi – tasking
2. Process based multi – tasking

**Java supports only thread based multi – tasking.**

**Process based multi – tasking**

Concept of executing more than one program simultaneously which are present at different locations of RAM is known as process based multi – tasking.

For example, while typing a Java program we can listen to a song and at the same time we can download a file from the net, all these tasks are executing simultaneously and there is no relationship between these tasks. All these tasks have their own independent address.

Thus, in process based multi – tasking, addresses of both the programs have to be maintained, since control has to shift from one part of the RAM to another.

This type of multi – tasking is developed at Operating System level.

This increases the overhead of the processor. Thus, it is the dis – advantage of process based multi – tasking.

**Operating System**

Process 1 Process 2 Process 3

Editplus VLC Media Browser

Player

1010 2020 3030

**Thread based multi – tasking**

Concept of executing more than one functionality concurrently belonging to the same memory domain (i.e. same program) is known as thread based multi – tasking.

This type of multi – tasking is developed at programmatic level.

**Operating System**

Process 1 Process 2

2020

Thread – 0 Thread – 1 Thread - 2

1010

Thread – 0 Thread – 1 Thread - 2

**Note**

In general, process based multi – tasking is called just multi – tasking and thread based multi – tasking is called multi – threading.

**What is the difference between multi – tasking and multi – threading?**

Multi – tasking is heavy weight because switching between contexts is slow because each process is stored at separate address.

Multi – threading is light weight because switching between contexts is fast because each thread is stored at same address.

**What is thread?**

A thread is an independent sequential flow of execution. It is a stack created in Java Stack Area.

**An overview on Java Threads**

We can consider JVM is also a process. When JVM is created in its JSA (Java Stack Area), by default two threads are created with names:

1. main : to execute Java methods
2. Garbage collector: to destroy UN – referenced objects.

So, by default Java is a multi – threaded programming language.

Thread execution is of two types:

1. Single thread execution
2. Multithreading execution

**Sequential execution**

Sequential execution means single thread execution. It executes the methods one after one. It takes more time to complete all methods execution.

**Concurrent execution**

Concurrent execution means multi – threading execution. Executing multiple tasks in “start – suspend – resume – end” fashion is called concurrent execution. It means both tasks will starts at different point of time and one task is paused while other task is executing. In Java, the JVM allows an application to have multiple threads of execution running concurrently. When a program contains multiple threads, then the CPU can switch between the two threads to execute them at the same time. It takes less time to complete all methods execution. To have a concurrent execution developer must create user defined thread in the program.

**What is multi – threading?**

It is a process of creating multiple threads in JSA for executing multiple tasks concurrently to finish their execution in short time by using processor idle time effectively.

**How can we create user defined thread in JVM?**

Thread class and Runnable interface are the two structures using which we can create user defined thread in JVM.

**Implementing a thread in Java**

In Java, each task is an instance of **Runnable** interface, also called a runnable object. Tasks are objects. To create tasks, we have to first define a class for tasks. A task class must implement the **Runnable** interface. The Runnable interface is rather simple. All it contains is the **run()** method, for which we have to provide the body.

After providing body, we need to execute the functionality as a thread. This is done as follows:

There is a class named **Thread** present in the API. This class supports many functionalities which make a given functionality to get executed as a thread. One such functionality is the **start()** method.

The **start()** method recognizes the **run()** method of **Runnable** interface and then the **run()** method is executed as a thread.

**Introduction to run() method**

1. It is the initial point of user – defined thread execution.
2. It is actually defined in **Runnable** interface and is implemented in **Thread** class.
3. It is implemented as empty method in **Thread** class.
4. To execute our logic in user defined thread, we must override **run()** method.

interface Runnable class Thread implements Runnable

{ {

public void run(); public void run()

} {}

}

Now we can implement thread using either of the following two ways:

class A implements Runnable class A extends Thread

{ {

public void run() public void run()

{ // logic defined } { // logic defined }

} }

MyRunnable (**class**)

public void run() {}

public void start();

Thread (**class**)

public void run();

Runnable (**interface**)

**Override**

public void run()

{

// logic defined

}

public void run()

{

// logic defined

}

MyThread (**class**)

**Note**

Whether we extend Thread class or implement Runnable interface directly, we are using **run()** method of the Runnable interface i.e. using Runnable interface directly or indirectly is compulsory.

**How do you define thread in Java?**

By using Runnable interface.

**Different ways to create custom threads in java**

In java we can create user defined threads in two ways:

1. Implementing Runnable interface
2. Extending Thread class

In both the approaches we should override run() method in sub class with the logic that should be executed in user defined thread concurrently, and should call start() method in Thread class object to create thread of execution in Java Stack Area.

**The Thread class**

The Thread class contains the constructors for creating threads for tasks, and the methods for controlling threads.

|  |  |
| --- | --- |
| Thread() | Creates an empty Thread |
| Thread(Runnable r) | Creates a Thread for specific task |
| void start() | Starts the thread that causes the run() method to be invoked by the JVM |
| boolean isAlive () | Test whether the thread is currently running |
| void setPriority (int p) | Sets priority ranging from 1 to 10 for this thread |
| void join() | Waits for this thread to finish |
| void sleep (long millis) | Puts a thread to sleep for a specified time in milliseconds |
| void yield() | Causes a thread to pause temporarily and allows other threads to execute |
| void interrupt() | Interrupts this thread |

**Note**

public void sleep(long millis) throws InterruptedException.

InterruptedException is a checked exception, so sleep method caller should handle or report this exception else it leads to CE: unreported exception must be caught or declared to be thrown. Java forces to catch checked exception, it have to put in try – catch block. If a sleep method is invoked in a loop, we must wrap the loop in the try – catch block.

**Internal Structure of Thread class**

class Thread

{

Runnable target;

Thread()

{

target = null;

}

Thread(Runnable target)

{

this.target = target;

}

public void start()

{

// passing current thread object to JVM

}

}

**start()** method didn’t call **run()** method. Internally it contains native logic (**in C language).**

**start()** method drop the object and passes current thread object to JVM.

JVM calls the **run()** method.

**Developing custom thread implementing from Runnable**

class MyRunnable implements Runnable

{

public void run()

{

System.out.println("Run");

}

public static void main(String[] args)

{

MyRunnable mr = new MyRunnable();

Thread th = new Thread(mr);

th.start();

}

}

**JVM Architecture**

**Heap Area Java Stack Area**

MyRunnable **Main thread Thread - 0**

run()

if ( this.target != null)

{

target.run();

}

else

{

this.run();

}

main()

{

mr

th

th.start()

}

**1**

**1010**

Thread Object

**3**

**2**

1010

Null 1010

2020

**2020**

**5**

**4**

Thread pool

**7**

**6**

2020.run()

**Developing custom thread extending from Thread**

class MyThread extends Thread

{

public void run()

{

System.out.println("Run");

}

public static void main(String[] args)

{

MyThread mt = new MyThread();

mt.start();

}

}

**Executing logic with two threads (main and custom thread)**

class MyRunnable implements Runnable

{

public void run()

{

for (int i = 0; i <= 50; i++)

{

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

}

}

public static void main(String[] args)

{

MyRunnable mr = new MyRunnable();

Thread th = new Thread(mr);

th.start();

for (int i = 0; i <= 50; i++)

{

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

}

}

}

**Output:** “main method for loop” and “run method logic” are executed concurrently.

In the above program, we can’t say which one starts first or ends first. Since, both the methods are getting executed concurrently, the start / end of execution of a method completely depends upon the time slice allocated for each function.

Thus, because of scheduling, we can’t guess the output of thread based programs.

**Can we call run() method directly from main method?**

Yes, we can call run() method directly. If we call it directly, user defined thread in JSA is not be created. It is executed in main thread.

class RunDirect implements Runnable

{

public void run()

{

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

{

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

}

}

public static void main(String[] args)

{

RunDirect rd = new RunDirect();

Thread th = new Thread(rd);

th.run();

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

{

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

}

}

}

Output

main() and run() are executed in same thread sequentially. Since run() is called before main(), first run() output is printed, then later main() output is printed.

**What is the output of the above program if we call start() before run() method?**

class RunDirect implements Runnable

{

public void run()

{

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

{

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

}

}

public static void main(String[] args)

{

RunDirect rd = new RunDirect();

Thread th = new Thread(rd);

th.start();

th.run();

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

{

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

}

}

}

Output

run() method logic is executed concurrently, one is from Thread-0 and another is from main() thread. After completion of run() method execution in main thread, main() method for loop execution starts.

**If start() method is called on Thread class object directly, is subclass run() method executed?**

No, run method is executed from Thread class. (As we know that in case of inheritance parent class object can’t call child class method).

class MyThread extends Thread

{

public void run()

{

System.out.println("Run");

}

public static void main(String[] args)

{

System.out.println("Main");

Thread mt = new Thread();

mt.start();

}

}

Output

Main

**Can we call start() method more than once on the same thread object?**

No, it leads to exception **java.lang.IllegalThreadStateException**.

**Then how can we create multiple user defined threads in JavaStack Area?**

There are two ways to create more than one user defined threads in Java Stack Area.

1. Create multiple thread sub – class objects and call start() methodon each thread object.
2. Create multiple sub – classes from Thread class or Runnable interface, create its object and call start() method.

**In first approach, all thread execute the same run() method logic, because all its thread objects are created from the same class.**

class MultipleThread1 implements Runnable

{

Thread th = new Thread();

public void run()

{

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

{

System.out.println(th.getName() + " Run: " + i);

}

}

public static void main(String[] args)

{

MultipleThread1 mt1 = new MultipleThread1();

MultipleThread1 mt2 = new MultipleThread1();

Thread th1 = new Thread(mt1);

Thread th2 = new Thread(mt2);

th1.start();

th2.start();

}

}

Output

As we can notice that same for loop is executed from all threads.

**In second approach, all threads executes run() method, but with different logic, because thread objects are created with different class. This is the actual approach used in projects to develop multi – threading.**

class ThreadA implements Runnable

{

Thread th = new Thread();

public void run()

{

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

{

System.out.println(th.getName() + " ThreadA: " + i);

}

}

}

class ThreadB implements Runnable

{

Thread th = new Thread();

public void run()

{

for (int i = 11; i <= 20; i++)

{

System.out.println(th.getName() + " ThreadB: " + i);

}

}

}

class MultipleThread

{

public static void main(String[] args)

{

ThreadA ta = new ThreadA();

ThreadB tb = new ThreadB();

Thread th1 = new Thread(ta);

Thread th2 = new Thread(tb);

th1.start();

th2.start();

}

}

Output

As we can notice that, different for loops are executed for different threads.

**Note**

getName() is used to retrieve the name of currently executing thread.

**Write a multi – threading model application for executing addition and subtraction logic concurrently. These logic should add and subtract numbers from 1 to 50. In every iteration, you must display result.**

class Add implements Runnable

{

public void run()

{

int sum = 0;

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

{

sum += i;

System.out.println("Add: " + sum);

}

}

}

class Sub implements Runnable

{

int diff = 0;

public void run()

{

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

{

diff -= i;

System.out.println("Diff: " + diff);

}

}

}

class AddSub

{

public static void main(String args[])

{

Add a = new Add();

Sub s = new Sub();

Thread th1 = new Thread(a);

Thread th2 = new Thread(s);

th1.start();

th2.start();

}

}

**Can we override start() method?**

Yes, we can start() method, as it is non – final method.

**If we override start() method, is custom thread created?**

No, custom thread is not created.

**Then why thread class developer not declared start() method as final?**

It is project requirement: before starting this thread, if we want to do some validations and calculations to update current thread object state, we should override start method in subclass with this validation logic and then we should start custom thread.

class OverrideStart extends Thread

{

@Override

public void run()

{

System.out.println("Run");

}

@Override

public void start()

{

System.out.println("Start");

}

public static void main(String[] args)

{

OverrideStart os = new OverrideStart();

os.start();

System.out.println("Main");

}

}

Output

Start

Main

**If we call run() from overriding start(), in which thread run() is executed?**

Custom thread is not created, run() is executed in main thread as start() method is executed in main() thread.

class OverrideStart extends Thread

{

@Override

public void run()

{

System.out.println("Run");

}

@Override

public void start()

{

System.out.println("Start");

run();

}

public static void main(String[] args)

{

OverrideStart os = new OverrideStart();

os.start();

System.out.println("Main");

}

}

Output

Start

Run

Main

**How can we start custom thread from overriding method?**

We must place **super.start()** at the end of overriding **start()** method.

class OverrideStart extends Thread

{

@Override

public void run()

{

System.out.println(getName() + " Run");

}

@Override

public void start()

{

System.out.println(getName() + " Start");

super.start();

}

public static void main(String[] args)

{

OverrideStart os = new OverrideStart();

os.start();

System.out.println(os.getName() + " Main");

}

}

**Write a program to show time difference between single and multiple threads execution.**

class PrintNumbers

{

void print1to50()

{

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

{

System.out.print(i + " ");

try

{

Thread.sleep(100);

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

}

}

void print50to1()

{

for (int i = 50; i >= 1; i--)

{

System.out.print(i + " ");

try

{

Thread.sleep(100);

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

}

}

}

class SingleThreadExecution

{

public static void main(String[] args)

{

PrintNumbers pn = new PrintNumbers();

long time1 = System.currentTimeMillis();

pn.print1to50();

System.out.println();

pn.print50to1();

long time2 = System.currentTimeMillis();

System.out.println("\nTime taken: " + ((time2 –

time1)/1000) + "secs");

}

}

Output

Sequential Execution: takes 10 secs to complete two tasks

class MultiThreadExecution implements Runnable

{

static PrintNumbers pn = new PrintNumbers();

@Override

public void run()

{

pn.print50to1();

}

public static void main(String[] args)

{

MultiThreadExecution mt = new MultiThreadExecution();

Thread th = new Thread(mt);

long time1 = System.currentTimeMillis();

th.start();

pn.print1to50();

long time2 = System.currentTimeMillis();

System.out.println("\nTime Taken: " + ((time2 –

time1)/1000) + "secs");

}

}

Output

Concurrent Execution: takes 5 secs to complete two tasks.

**Thread execution control**

Sometimes, it is necessary to control the execution of the thread. Thread execution can be controlled in three ways:

1. Pausing thread execution for a given period of time using **sleep()** method.
2. Pausing thread execution until other thread execution is complete using **join()** method.
3. Executing threads sequentially using **synchronized** keyword.
4. **Pausing thread execution for a given period of time**

There is a static function named **sleep()** in Thread class which takes time as argument.

**public static void sleep(long millis) throws InterruptedException**

**public static void sleep(long millis, int nanos) throws InterruptedException**

sleep() method causes currently executing thread to sleep (cease execution) for the specified number of milliseconds plus specified number of nanoseconds.

The value of millis should not be negative or the value of nanos is should be in the range 0 – 999999 else it leads to exception **java.lang.IllegalArgumentException.**

**InterruptedException** is a checked exception, so the sleep() caller should handle or report this exception else it leads to CE: unreported exception must be caught or declared to be thrown.

class SleepDemo implements Runnable

{

@Override

public void run()

{

for (int i = 11; i <= 20; i++)

{

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

try

{

Thread.sleep(100);

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

}

}

public static void main(String[] args)

{

SleepDemo sd = new SleepDemo();

Thread th = new Thread(sd);

th.start();

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

{

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

}

}

}

Suppose if we want to skip the try block, then we should use **throws Exception** in the function declarator in which **sleep()** method is called (here it is **run()** method). But this leads to a problem, because if we will use **throws Exception** in **run()** method, then it directly violates the concept of overriding. The signature of **run()** method defined in Thread class is just **public void run()**. When we call **start()** method, it searches for overridden **run()** method and when it does not find any, it gives a compilation error.

**What happens if we use Thread.sleep() in main()?**

class SleepDemo implements Runnable

{

@Override

public void run()

{

for (int i = 11; i <= 20; i++)

{

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

}

}

public static void main(String[] args)

{

SleepDemo sd = new SleepDemo();

Thread th = new Thread(sd);

th.start();

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

{

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

try

{

Thread.sleep(100);

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

}

}

}

1. **Pausing thread execution until other thread execution is completed**

Pausing thread execution until other thread execution is completed is called thread join. Thread class has below overloaded join methods to perform this task:  
**public final void join() throws InterruptedException**

**public final synchronized void join(long millis) throws InterruptedException**

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

No – arg join() method pauses thread execution until completion of other thread execution. If other thread execution is blocked forever, this thread execution is also blocked forever.

Whereas parameterized join() method does not block thread execution until completion of other thread execution. Its execution is resumed after completion of given time.

**Difference between join(long) and sleep(long)**

**sleep(long)** method pauses thread execution independent of other thread execution for the given period of time completely. It does not allow thread to run until the given time is completed.

**join(long)** method pauses thread execution dependent on other thread’s execution for the given period of time. It pause thread execution only for the given period of time or if that other thread execution is completed before the given time, current thread execution is resumed immediately even though given paused time is not finished.

class ThreadA extends Thread implements Runnable

{

@Override

public void run()

{

int sum = 0;

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

{

sum += i;

}

System.out.println("Sum: " + sum);

}

}

class ThreadB implements Runnable

{

@Override

public void run()

{

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

{

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

}

}

}

class ThreadC implements Runnable

{

Thread ta;

ThreadC(Thread ta)

{

this.ta = ta;

}

@Override

public void run()

{

for (int i = 11; i <= 20; i++)

{

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

if (i == 15)

{

try

{

ta.join();

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

}

}

}

}

class JoinDemo

{

public static void main(String[] args)

{

ThreadA ta = new ThreadA();

ThreadB tb = new ThreadB();

ThreadC tc = new ThreadC(ta);

Thread th1 = new Thread(ta);

Thread th2 = new Thread(tb);

Thread th3 = new Thread(tc);

th1.start();

th2.start();

th3.start();

}

}

**In the above program, we have four threads, th1, th2, th3 and main(). What happens if main() gets executed first? Is it going to terminate the program?**

No, every thread in the program is by default joined, as the last statement in main(). This ensures that before main() termination, all other threads have completes its execution. Always main() thread is the last terminated thread.

**What would be the consequences when multiple thread act upon a single object?**

Let us assume, we have a class Common, which has a function fun1(). Let this fun1() be available to two threads named ThreadA and ThreadB.

class Common

{

public void fun1(String s)

{

System.out.print("Hello ");

try

{

Thread.sleep(1822);

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

System.out.println(s + " World");

}

}

class ThreadA implements Runnable

{

Common c;

ThreadA(Common c)

{

this.c = c;

}

@Override

public void run()

{

c.fun1("Java");

}

}

class ThreadB implements Runnable

{

Common c;

ThreadB(Common c)

{

this.c = c;

}

@Override

public void run()

{

c.fun1("C++");

}

}

class CommonTest

{

public static void main(String[] args)

{

Common c1 = new Common();

ThreadA ta = new ThreadA(c1);

ThreadB tb = new ThreadB(c1);

Thread th1 = new Thread(ta);

Thread th2 = new Thread(tb);

th1.start();

th2.start();

}

}

The above function is supposed to give an output:

Hello Java World

Hello C++ World

But the output will be

Hello Hello C++ World

Java World

This is so because, the two threads ta and tb are entering into fun1() and acting upon it concurrently. We can get the desired output if we avoid ta and tb acting upon fun1() concurrently. This is nothing but the concept of thread safe.

1. **Executing threads sequentially or Synchronization**

Thread safe is the concept of avoiding multiple threads acting upon the same functionality simultaneously. Thread safety can be achieved by using a keyword **synchronized.**

Synchronized is developed by using:

1. Synchronized methods
2. Synchronized blocks

**What happens when we declare methods as synchronized?**

When we call synchronized method, the current object of this method is locked by this current thread. So that other threads can’t use this locked object for executing either the same synchronized method or other synchronized methods. The thread that locks the object is called monitor. This current object is unlocked only after completion of that synchronized method execution either normally or abnormally.

**What is the difference between synchronized methods and blocks?**

1. If method is declared as synchronized, that method’s complete logic is executed in sequence from multiple threads by using same object whereas if we declare block as synchronized, only the statements written inside that block are executed sequentially not complete method logic.

**Synchronized method Synchronized block**

public class bank public class bank

{ {

private double balance; private double balance;

public synchronized void withdraw(int amt) public void withdraw(int amt)

{ {

System.out.println(balance); System.out.println(balance);

balance -= amt; synchronozed(this)

System.out.println(balance); {

} balance -= amt;

} }

System.out.println(balance);

}

1. Using synchronization method we can only lock current object of the method whereas using synchronized block we can block either current object or argument object of the method, we must pass object referenced variable to synchronized block.

class Example class Example

{ {

void m1() void m1(Sample s)

{ {

synchronized(this) synchronized(s)

{} {}

} }

} }

**Need of synchronization**

We must develop synchronization to get correct results in modifying object’s data from multiple threads. We get the data corruption problem only when multiple threads accessing same object. If different object is using by multiple threads then no need to implement synchronization.

**Program to demonstrate data corruption and solution using synchronization**

class Add

{

int x;

int y;

synchronized void add(int x, int y)

{

this.x = x;

this.y = y;

try

{

Thread.sleep(1000);

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

int res = this.x + this.y;

System.out.println("In " + Thread.currentThread().getName()

+ " Result: " + res);

}

}

class Thread1 implements Runnable

{

Add a;

Thread1(Add a)

{

this.a = a;

}

@Override

public void run()

{

a.add(30, 40);

}

}

class Thread2 implements Runnable

{

Add a;

Thread2(Add a)

{

this.a = a;

}

@Override

public void run()

{

a.add(50, 60);

}

}

class SynchronozedMethodDemo

{

public static void main(String[] args)

{

Add a = new Add();

Thread1 th1 = new Thread1(a);

Thread2 th2 = new Thread2(a);

Thread th3 = new Thread(th1);

Thread th4 = new Thread(th2);

th3.start();

th4.start();

}

}

Execute add method without synchronized keyword to find the output difference.

**Program to demonstrate synchronized block**

class PrintTable

{

void print(int a)

{

synchronized(this)

{

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

{

System.out.println("In " +

Thread.currentThread().getName() + " " + (i\*a));

try

{

Thread.sleep(1000);

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

}

}

}

}

class MyRunnable1 implements Runnable

{

PrintTable pt;

MyRunnable1(PrintTable pt)

{

this.pt = pt;

}

@Override

public void run()

{

pt.print(5);

}

}

class MyRunnable2 implements Runnable

{

PrintTable pt;

MyRunnable2(PrintTable pt)

{

this.pt = pt;

}

@Override

public void run()

{

pt.print(10);

}

}

class SynchronizedBlockDemo

{

public static void main(String[] args)

{

PrintTable pt = new PrintTable();

MyRunnable1 mr1 = new MyRunnable1(pt);

MyRunnable2 mr2 = new MyRunnable2(pt);

Thread th1 = new Thread(mr1);

Thread th2 = new Thread(mr2);

th1.start();

th2.start();

}

}

**Inter Thread Communication**

Suppose if we want the threads getting executed according to our logic and specification, then how should we define the threads?

**How can we gain partial control on the scheduling mechanism which is supervised by the local operating system?**

For achieving this we make use of several controls:

1. **wait()**
2. **notify()**
3. **notifyAll()**

**wait()**

It is a method belonging to the object class. Whenever we need to suspend a thread unconditionally, then we will use wait() method.

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

**public final native void wait(long milliSecs) throws InterruptedException**

**public final void wait(long milliSecs, int nanoSecs) throws InterruptedException.**

**Note**: wait() method works only on synchronized blocks.

**notify()**

This method is also present in Object class. Whenever we need to resume a suspended (waiting) thread, then we use notify() method.

**public final native void notify()**

**notifyAll()**

This method is also used in Object class. It is used for the same purpose as notify(), but if there are multiple threads are available.

**public final native void notify()**

To gain the above mentioned partial control, the threads should have a sign of mutual understanding between them i.e. they should be able to communicate with one another. This is known as **Thread Synchronization** or **Inter Thread Communication.**

The process of executing threads in sequence in loop with communication is called Inter thread Communication. We develop this concept when two different dependent tasks want to be executed continuously in sequence by two different threads.

**Why wait(), notify() and notifyAll() methods are defined in Object class, why not in Thread class?**

These three methods are not only working on Thread objects, they are also working on normal objects for unlocking objects from the Thread and for notifying above lock availability on that Object. So, these three methods must be defined in Object class to work on normal objects also.

**What will happen if we do not call notify() or notifyAll() methods on waiting threads?**

That thread is in the block state forever.

**What is the difference between sleep(100), join(100), wait(100) method calls?**

* **sleep(100)** blocks thread execution independent of other thread for 1oo milliseconds.
* **join(100)** blocks thread execution by depending on either other thread till it is completed or till 100 milliseconds completed, whichever coming first, thread resumes its execution immediately.
* **wait(100)** blocks thread execution by depending on either other thread till it is called notify() or till 100 milliseconds completed, whichever coming first thread resume its execution immediately.

**What is the rule in using these three methods or in what type of methods these three methods are allowed?**

These three methods are allowed only in synchronized methods. If we call them in non – synchronized methods JVM throws an exception **java.lang.IllegalMonitorException**. Because to release the lock that thread should be the monitor of that object.

**The best example for the inter thread communication application is Producer and Consumer application. Producer should produce goods only when goods are consumed, and consumer can consume goods only when goods are produced.**

class Factory

{

int items;

boolean itemsInFactory;

Factory()

{

items = 0;

itemsInFactory = false;

}

synchronized void produce(int items)

{

if (itemsInFactory)

{

try

{

wait();

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

}

this.items = items;

itemsInFactory = true;

System.out.println("Produced Items: " + items);

try

{

Thread.sleep(1000);

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

notify();

}

synchronized void consume()

{

if (!itemsInFactory)

{

try

{

wait();

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

}

System.out.println("Consumed Items: " + items);

try

{

Thread.sleep(1000);

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

itemsInFactory = false;

notify();

}

}

class Producer implements Runnable

{

Factory fa;

Producer(Factory fa)

{

this.fa = fa;

new Thread(this).start();

}

@Override

public void run()

{

int i = 1;

while (i <= 10)

{

fa.produce(i++);

}

}

}

class Consumer implements Runnable

{

Factory fa;

Consumer(Factory fa)

{

this.fa = fa;

new Thread(this).start();

}

@Override

public void run()

{

int i = 1;

while (i <= 10)

{

fa.consume();

i++;

}

}

}

class ITCWaitAndNotify

{

public static void main(String[] args)

{

Factory bajaj = new Factory();

new Producer(bajaj);

new Consumer(bajaj);

}

}

**Note**

1. notify() resumes one thread which was already in waiting state.
2. notifyAll() resumes all thread that are in wait state.

**Consider the following scenario:**

Wait state Executing State

T2 -> run() { notify(). . . }

Totally we have four threads – t1, t2, t3 and t4. T1, T3, t4 are in waiting state whereas T2 is in execution state.

**Now if we call notify() in T2, which thread is it going to resume? Is it going to resume the thread that has entered into the wait() state long back?**

No, it is completely decided by the local OS.

Thus, because of this improper notifying, sometimes, we may not be able to achieve inter – thread communication, when there are more than two threads.

To overcome this problem we assign priorities or preferences to the thread.

**Thread Priorities**

Every thread created in JVM is assigned with a priority. The priority range is between 1 to 10.

* 1 is called minimum priority
* 5 is called normal priority
* 10 is called maximum priority.

In thread class below three variables are defined to represent above three values:

* public static final int **MIN\_PRIORITY**
* public static final int **NORM\_PRIORITY**
* public static final int **MAX\_PRIORITY**

Thus, when we have multiple threads in wait state and if we call notify() in the current thread, then the thread with highest priority will be notified first.

**Note**

1. A thread inherits the priority from its parent thread. The default priority of every thread is normal priority 5, because the main thread priority is 5.
2. Priorities should be assigned to the corresponding threads before calling the start() method.
3. Priority concept can be used only when there is a possibility for the threads to enter the wait state.
4. Local OS will never schedule the threads depending upon their priorities.
5. New priority value range should between 1 to 10, else it leads to exception **java.lang.IllegalArgumentException**.

The priority of the thread can be set using the **setPriority()** method. It is a non – static method.

The priority of the thread can be read or retrieve using **getPriority()** method. It is a non – static method.

**public final void setPriority(int newPriority)**

**public final int getPriority()**

**Thread Name**

User defined thread created with the default name **Thread - <index>**, where index is the integer number starts with 0. So, the first user defined thread name will be Thread – 0, second thread name is Thread – 1, etc…

The name of thread can be set using the **setName()** method and read using the **getName()** method, both of which are defined in Thread class. Both are non – static methods.

**public final void setName(String name)**

**public final String getName()**

The default thread name can be changed by using either

* At time of thread object creation using String parameterized constructor
* After object creation using above set method.

**Program to demonstrate Priority and Name of the thread**

class MyThread extends Thread

{

MyThread()

{}

MyThread(String name)

{

super(name);

}

public void run()

{

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

{

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

}

}

}

class ThreadNameAndPriority

{

public static void main(String[] args)

{

MyThread mt1 = new MyThread();

MyThread mt2 = new MyThread("child2");

System.out.println("m1 initial name and priority");

System.out.println(mt1.getName());

System.out.println(mt1.getPriority());

System.out.println("m2 intial name and priority");

System.out.println(mt2.getName());

System.out.println(mt2.getPriority());

mt1.setName("child1");

mt1.setPriority(6);

mt2.setPriority(9);

System.out.println("m1 changed name and priority");

System.out.println(mt1.getName());

System.out.println(mt1.getPriority());

System.out.println("m2 changed name and priority");

System.out.println(mt2.getName());

System.out.println(mt2.getPriority());

mt1.start();

mt2.start();

}

}

From the above output, we can conclude that since thread 2 priority is higher than thread 1 priority, but thread 2 is not executed faster than thread 1. It is so because as an absolute value, a priority is meaningless. A higher priority thread does not run faster than a lower priority thread, if it is the only thread running.

**Thread Execution Procedure**

JVM executes thread based on their priorities and scheduling.

**Thread Scheduler**

Schedulers in JVM implementations usually employee one of the following two strategies:

1. Preemptive Scheduling  
   If a thread with a higher priority than the current running thread moves to the ready to run state, the current running thread can be preemptive(moved to the ready to run state) to let the higher priority thread execute.
2. Time – sliced or Round Robin scheduling  
   A running thread is allowed to execute for a fixed length of time, after which it moves to the Ready to run state to await its turn to run again.

**Note**

Thread scheduling and implementations are platform dependent, therefor how thread will be scheduled is unpredictable.

**Deadlock**

Deadlock is a situation where if two threads are waiting on each other to complete their execution. Deadlock is occurred due to wrong usage of synchronized keyword. If first thread is calling a synchronized method by using a locked object of second thread and second thread calling a synchronized method by using a locked object of first thread, then deadlock occur.

For example, let us assume one thread enters a synchronized method of object X and another thread enters a synchronized method of object Y.

Now, if the thread in X needs to call the synchronized method in Y, it will not get access because the method in Y is synchronized and already a thread is acting on it.

If the thread in Y, tries to call the synchronized method in X, it waits forever, because to access the method in X, it would have to release its own lock on the object Y, so that thread acting on X would complete and this will not happen.

Hence, there is a circular dependency on X and Y o each other and it results in a deadlock.

**Program to demonstrate deadlock**

class FirstClass

{

synchronized void firstClassMethod(SecondClass sc)

{

String name = Thread.currentThread().getName();

System.out.println(name + " is entered in the

fc.firstClassMethod");

try

{

Thread.sleep(100);

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

System.out.println(name + " is trying to call

sc.lastMethod");

sc.lastMethod();

}

synchronized void firstMethod()

{

System.out.println("Inside fc.firstMethod");

}

}

class SecondClass

{

synchronized void secondClassMethod(FirstClass fc)

{

String name = Thread.currentThread().getName();

System.out.println(name + " is entered in

sc.secondClassMethod");

try

{

Thread.sleep(100);

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

System.out.println(name + " is trying to call

fc.firstMethod");

fc.firstMethod();

}

synchronized void lastMethod()

{

System.out.println("Inside sc.lastMethod");

}

}

class DeadLock implements Runnable

{

FirstClass fc = new FirstClass();

SecondClass sc = new SecondClass();

DeadLock()

{

Thread th = new Thread(this);

th.start();

fc.firstClassMethod(sc);

System.out.println("Back in main thread");

}

public void run()

{

sc.secondClassMethod(fc);

System.out.println("Back to another thread");

}

public static void main(String[] args)

{

new DeadLock();

}

}

**Different cases in wait()**

1. wait() method is called in non – synchronized method without **try/catch** or **throws** keywords.

class WaitCases

{

void m1()

{

wait();

}

public static void main(String[] args)

{

WaitCases wc = new WaitCases();

wc.m1();

}

}

It leads to CE: unreported exception java.lang.InterruptedException; must be caught or declare to be thrown.

1. wait() method is called in non – synchronized method in **try/catch** block.

class WaitCases

{

void m1()

{

try

{

wait();

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

}

public static void main(String[] args)

{

WaitCases wc = new WaitCases();

wc.m1();

}

}

It leads to RE: java.lang.IllegalMonitorStateException

1. No – arg wait() method is called in synchronized method but notify() method is not called from other thread.

class WaitCases

{

synchronized void m1()

{

try

{

System.out.println("Hello");

wait();

System.out.println("Bye");

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

}

public static void main(String[] args)

{

WaitCases wc = new WaitCases();

wc.m1();

}

}

This thread execution is blocked forever.

1. long – parameter wait() method is called in synchronized method but notify() is not called from other thread.

class WaitCases

{

synchronized void m1()

{

try

{

System.out.println("Hello");

wait(3000);

System.out.println("Bye");

}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

}

public static void main(String[] args)

{

WaitCases wc = new WaitCases();

wc.m1();

}

}

This thread execution is resumed after the given time is elapsed.

**Types of thread**

Java allows us to create two types of thread. They are:

1. Non – Daemon Threads
2. Daemon Threads

A thread that executes main lofic of the project is called non – daemon thread.

A thread that is running in background to provide services to non – daemon threads is called daemon thread. So, we can say daemon threads are service thread.

**Consider the following example**

JVM executes a .class file and while in this process, many objects will be created. Simultaneously, garbage collector collects all the un – referenced objects in the background. Thus, garbage collector is a background process.

In other words, garbage collector is an infinite loop program and runs as long as JVM is in execution. That garbage collector thread provides its services in the background to support the JVM thread, and as long as JVM thread is in execution, the GC thread will also be in execution.

In this case GC thread is nothing but a Daemon Thread.

Concept of Daemon Thread is to design a thread with infinite loop which would be supporting the parent thread as a background process and it is supposed to get executed along with the parent thread. Also, this background thread should stop; when the parent thread stops.

Every user defined thread is created as non – daemon thread by default, because main thread is a non – daemon thread and daemon property is also inherited from parent thread.

To check whether a thread is daemon or non – daemon, Thread class has below method,

**public final boolean isDaemon()**

returns true if thread is Daemon, else returns false.

class DaemonCheck extends Thread

{

public static void main(String[] args)

{

DaemonCheck dc = new DaemonCheck();

System.out.println(dc.isDaemon());

}

}

Output

False

**Daemon thread creation**

To create user – define thread as daemon thread, Thread class has below method,

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

if on value is true – thread is created as daemon, else it is created as non – daemon.

So, the daemon property default value is false.

**Rule**

setDaemon() method cannot be called after start() method call, it leads to RE: **java.lang.IllegalThreadStateException**, because once a thread is created as non – daemon thread, it cannot be converted as daemon thread.

class DaemonCheck extends Thread

{

public static void main(String[] args)

{

DaemonCheck dc = new DaemonCheck();

dc.setDaemon(true);

dc.start();

}

}

No error

class DaemonCheck extends Thread

{

public static void main(String[] args)

{

DaemonCheck dc = new DaemonCheck();

dc.start();

dc.setDaemon(true);

}

}

RE: IllegalThreadStateException

**Note**

Daemon Thread can never be joined, not even in main (which is the default join).

class DaemonCheck implements Runnable

{

DaemonCheck()

{

Thread th = new Thread(this);

th.setDaemon(true);

th.start();

}

public void run()

{

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

{

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

}

}

public static void main(String[] args)

{

new DaemonCheck();

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

{

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

}

}

}

Output

Daemon Thread execution will completed in middle, all iteration output will not be printed.

**What is an inline thread?**

A thread that is created with anonymous thread object is called inline thread. Anonymous thread object means a thread that is created without extending from thread class or implementing from Runnable interface, explicitly called anonymous thread object. The other name for anonymous thread object is inline thread.

**Note**

It is not the official word given by SUN Microsystems. It is invented by our textbook authors or software industry experts.

**How can we create Thread without extending from Thread class or implementing from Runnable interface explicitly?**

Using anonymous inner class.

**Program to demonstrate anonymous inner class with Thread object**

class AnonymousThread

{

public static void main(String[] args)

{

( new Thread()

{

public void run()

{

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

{

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

}

}

}).start();

for (int i = 11; i <= 20; i++)

{

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

}

}

}

**Program to demonstrate anonymous inner class with Runnable object**

class AnonymousThread

{

public static void main(String[] args)

{

( new Thread

(

new Runnable()

{

public void run()

{

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

{

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

}

}

}

)

).start();

for (int i = 11; i <= 20; i++)

{

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

}

}

}

**Thread Life Cycle**

Understanding the life cycle of a thread is valuable when programming with threads. Threads can exist in different state. Just because of a thread’s **start()** method has been called, it does not mean that the thread has access to the CPU and can start executing straight away. Several factors determine how it will proceed. Once thread is created, it is available in any one the below five states.

1. **New State**

A thread has been created, but it has not yet started. A thread is started by calling its **start()** method.

1. **Ready to Run state**

This start is also called Runnable state, also called Queue. A thread starts life in the Ready – to – run state by calling start method by calling and waits for its turn. The thread scheduler decides which thread runs and for how long.

1. **Running state**

If a thread is in the Running state, it means that the thread is currently executing.

1. **Dead state**

Once in this state, the thread cannot ever run again.

1. **Non Runnable state**

A running thread can transit to one of the non – runnable states, depending on the circumstances. A thread remains in a non – runnable state until a special transition occurs. A thread does not go directly to the Running state from a non – runnable state, but transits first to the Ready – to – run state.

The non – runnable state can be characterized as follows:

* **Sleeping**: The thread sleeps for a specified amount of time.
* **Blocked for I/O**: The thread waits for a blocking operation to complete.
* **Blocked for join completion**: The thread awaits completion of another thread.
* **Waiting for notification**: The thread awaits notification from another thread.
* **Blocked for lock acquisition**: The thread waits to acquire the lock object.