Multithreading

# Introduction:

A thread is a single sequential flow of control within a program. It’s a light weight smallest part of the program and it’s also independent. A lot of conventional programming languages are single threaded or sequential. A single-threaded program can handle only one task at any given moment during its execution. In contrast, a multithreaded, or concurrent, program has multiple threads running simultaneously and so may handle multiple tasks at the same time during its execution. It’s not necessary to have a multiprocessor system to run a multithreaded program. Most modern operating systems support multitasking, which allows multithreaded programs to run on a single processor systems on a time sharing basis.

## Advantages:

Multithreaded programs have some advantages over single threaded programs:

* Servers can handle multiple clients simultaneously
* User programs are not blocked because threads are independent and we can perform multiple operations at a time.
* Makes application more responsive to user input. For example, it allows a GUI application to respond to user input immediately even if the application is engaged in time consuming computation
* Threads are independent so if an exception is thrown on one of the threads, other threads will not get effected
* Takes advantage of available multiple processors by executing the threads on different processors in parallel
* Suitable for developing reactive systems, which continuously monitor arrays of sensors and react to control systems according to the sensor readings. For example auto pilot systems, patient monitoring systems.

## Disadvantage:

A major disadvantage of multithreaded programming is that it involves significant overhead, owing to the cost of thread creation, context switching and synchronization.

Single threaded systems have some advantages of their own:

* Reduces overhead on the system as a single thread executes the program
* Reduces maintenance cost of application

Multithreaded programming has more challenges than a single threaded program as each thread executes independently. The exact order of execution of various threads is nondeterministic. Interaction and cooperation among different threads are often complicated which leads to safety and liveness problems.

**Context switching:** It refers to the process of storing and restoring of CPU state so that Thread execution can be resumed from the same point at a later point of time. Context Switching is the essential feature for a multithreaded program, Context switching is a type of process switching where we switch one process with another process. It involves switching of all the process resources with those needed by a new process. This means switching the memory address space. This includes memory addresses, page tables, and kernel resources, caches in the processor.

**Thread Life Cycle:** The life cycle of a thread passes through various stages of life cycle, as follows:

**New:** A thread is created using the Thread class. The program has to start the thread with start() method.

**Runnable:** This is the stage after the thread has been started by the start() method. It’s runnable now and the thread scheduler will schedule it.

**Running:** When the thread starts executing, it’s in "running" state. The scheduler selects one thread from the thread pool, and it starts executing in the application.

**Waiting:** There are multiple threads running/runnable in an application hence the need for synchronization between threads. So, a thread has to wait, till the other thread gets executed.

**Dead:** This is the state when the thread is terminated. The thread is in running state and as soon as it has completed processing it is in "dead state".

**Some useful methods:** Following are some of the useful methods of java.lang.Thread class

Please refer to the oracle’s help page for details on methods and constructors. Note the different constructors: [Thread (Java Platform SE 7)](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html)

void start() It brings the thread to alive state. It’s called on threads at new state.

void run() Use on Runnable run object, then that Runnable object's run method is called

static void sleep(long millis) Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds

void join() Waits for the thread to die

static void yield() A hint to the scheduler that the current thread is willing to yield its current use of a processor

void interrupt() Interrupts the thread

boolean isAlive() Tests if thread is alive

String getName() Gets the name of the thread

long getId() Returns the identifier of this Thread

**Creation of Threads:** A thread is an instance of the java.lang.Thread class. A thread object is also known an active object. Threads can be created in one of the two ways:

* By directly extending the Thread class – java.lang.Thread
* By implementing the Runnable interface – java.lang.Runnable

**Extending the Thread Class:** All threads are instances of the Thread class. Extending this class creates a thread conveniently. The run() method of this class should be overridden in the subclass. It defines the body of the thread similar to main() method of a sequential program. The run() method of a thread is invoked when execution starts and ends when the run() method returns. Here is a template for creating a new thread:

public class MyThread extends Thread {

public void run(){

//the thread body

}

//Other methods and fields

}

To start a new thread defined by MyThread, we have to create an instance of the MyThread class and invoke the start() method on it, which indirectly invokes the run() method.

MyThread p = new MyThread(143);

p.start();

**Extending the Runnable Interface:** This is the alternative way of defining a thread. The Runnable interface has a single method – the run() method:

public class MyThread implements Runnable {

public void run(){

//the thread body

}

//Other methods and fields

}

To start a new thread defined by MyThread, we have to create an instance of the MyThread class and invoke the start() method on that instance:

new Thread (new MyThread()).start();

# Example 1:

Multithreaded program that extends the ‘Thread’ class to create two threads. Both threads start counting from zero but in opposite directions.

**import** java.lang.Thread;

**public** **class** ThreadDemo **extends** Thread {

**protected** **int** count, inc, delay;

**public** ThreadDemo(**int** init, **int** inc, **int** delay) {

**this**.count = init;

**this**.inc = inc;

**this**.delay = delay;

}

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

**try** {

**for**(;;) {

System.***out***.print(count + " ");

count += inc;

*sleep*(delay);

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

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

**new** ThreadDemo(0, 1, 50).start();

**new** ThreadDemo(0, -1, 100).start();

}

}

**Description:** Since there is infinite loop, you must kill the program manually.

We use the protected as access modifies here: **protected** **int** count, inc, delay;

Protected keyword in Java refers to one of its access modifiers. The methods or data members declared as protected can be accessed from: Within the same class. Subclasses of same packages. Different classes of same packages.

The constructor takes three integer parameters: **public** ThreadDemo(**int** init, **int** inc, **int** delay)

The run() method: **public** **void** run()is overridden, codes here are implementing the need of the program just like what we do on main() of a single threaded program.

Two threads are created using the constructor that takes three parameters:

**new** ThreadDemo(0, 1, 50).start();

**new** ThreadDemo(0, -1, 100).start();

The second thread, the one with a negative increment, has a longer delay. Note in the output that the output from the two threads are interleaved.

The program actually involves three threads: the thread that executes the main() method and the two ‘ThreadDemo’ threads that we created inside the main() method. The thread that executes the main() method terminates when the main() method returns from creating the two ‘ThreadDemo’ threads. These two threads run concurrently and independent of each other. By the way, you have to terminate the program manually since there is an infinite loop.

Run the program, here is a sample of the output:

0 0 1 2 -1 3 -2 4 5 6 -3 7 -4 8 9 10 -5 11 12 -6 13 14 -7 15 16 -8 17 -9 18 19 20 -10 21 22 -11 23 -12 24 25 26 -13 27 28 -14 29 30 -15 31 32 -16 33 34 -17 35 -18 36 37 38 -19 39 40 -20 41 42 -21 43 44 -22 45 46 -23 47 48 -24 49 50 -25 51 -26 52 53 -27 54 55 -28 56 57 -29 58 59 60 -30 61 62 -31 63 64 -32 65 66 -33 67 68 -34 69 70 -35 71 72 -36 73 74 -37 75 76 -38 77 78 -39 79 80 -40 81 82 -41 83 84 -42 85 86 -43 87 88 -44 89 90 -45 91 92 -46 93 94 -47 95 96 -48 97 98 -49 99 100 -50 101 102 -51 103 104 -52 105 -53 106 107 108 -54 109 110 -55 111 -56 112 113 -57 114 115 -58 116 117 -59 118 119 -60 120 121 -61 122 123 -62 124 125 -63 126 127 -64 128 129 -65 130 131 -66 132 133 -67 134 135 -68 136 137 -69 138 139 -70 140 141 -71 142 143 -72 144 145 -73 146 147 -74 148 149 -75 150 151 -76 152 153 -77 154 155 -78 156 157 -79 158 159 -80 160 161 -81 162 163 -82 164 165 -83 166 167 -84 168 169 -85 170 171 -86 172

# Example 2:

Multithreaded program that extends the ‘Thread’ class to create a few threads and prints which thread is running.

This class overrides the run() method available in the ‘Thread’ class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread, start() invokes the run() method on the Thread object.

Public class MultithreadingDemo extends Thread {

public void run() {

try {

// Displaying the thread that is running

            System.out.println ("Thread " +

                  Thread.currentThread().getId() + " is running");

   } catch (Exception e) {

// Throw an exception

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

         }

     }

}

//The driver Class

public class Multithread {

public static void main(String[] args) {

int n = 5; // Number of threads

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

MultithreadingDemo obj = new MultithreadingDemo();

             obj.start(); //note: call start() on the object created

         }

     }

}

**Description:** This is another example of creating a thread extending the ‘Thread’ class.

We override the run() method. The method body has the code where we are using an existing method like: Thread.currentThread().getId

The getId method returns an integer value as the Id of the current thread

We create an object of the MultithreadingDemo class, named it ‘obj’.

We call the start() method on this ‘obj’. This starts the thread.

**Output:**

Thread 3 is running

Thread 4 is running

Thread 5 is running

Thread 6 is running

Thread 7 is running

# Example 3:

Multithreaded program that implements the ‘Runnable’ interface to create two threads. Both threads start counting from zero but in opposite directions (like Example1).

**import** java.lang.Runnable;

**public** **class** ThreadDemo2 **implements** Runnable {

**protected** **int** count, inc, delay;

**public** ThreadDemo2(**int** init, **int** inc, **int** delay) {

**this**.count = init;

**this**.inc = inc;

**this**.delay = delay;

}

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

**try** {

**for**(;;) {

System.***out***.print(count + " ");

count += inc;

*sleep*(delay);

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

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

**new** ThreadDemo2(0, 1, 50).start();

**new** ThreadDemo2(0, -1, 100).start();

}

}

**Description:** Since there is infinite loop, you must kill the program manually. This program works exactly like the Example1. We created the thread differently.

Sometimes it’s not possible to create a thread using the ‘Thread’ class because java only supports single inheritance among classes. For example, applets are required to extend the class java.applet.Applet. Thus applets cannot also extend the Thread class at the same time. Java provides and alternative means of defining threads – by implementing the Runnable interface.

# Example 4:

Multithreaded program that implements the ‘Runnable’ interface to create a few threads and prints which thread is running (like Example2).

Public class MultithreadingDemo2 implements Runnable {

public void run() {

try {

// Displaying the thread that is running

            System.out.println ("Thread " +

                  Thread.currentThread().getId() + " is running");

   } catch (Exception e) {

// Throwing an exception

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

         }

     }

}

//The driver Class

public class Multithread2 {

public static void main(String[] args) {

int n = 5; // Number of threads

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

MultithreadingDemo2 obj = new MultithreadingDemo2();

             obj.start();

         }

     }

}

**Description:** This is another example of creating a thread implementing the ‘Runnable’ interface. This program is similar to Exercise2 and produces similar output.

**Thread Priority and Scheduling:** The Java Virtual Machine implements a rather simple scheduling strategy to determine which of the runnable threads should be running. It is based on the priority of each runnable thread. Each thread contains a priority attribute, which is an integer value assigned when the thread is created. By default, a new thread has the same priority as the one that creates it. The priority of thread may change during its life time.

The JVM will select the runnable threads with the highest priority for execution. If more than one runnable thread has the same highest priority, one of them will be selected arbitrarily.

A programmer can, alternatively, assign thread priorities which run from 1 to 10.

public static int MIN\_PRIORITY: This is minimum priority of a thread and it can be assigned a value of 1.

public static int NORM\_PRIORITY: This is default priority of a thread. If it’s not explicitly defined, its value is 5.

public static int MAX\_PRIORITY: This is maximum priority of a thread and it can be assigned a value of 10.

There are two methods to get and to set Thread Priority:

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

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

**Synchronization:** A multithreaded program may face a situation where multiple threads are trying to access same resource at the same time. This will have undesirable results due to concurrency issue, hence the need for synchronization. It’s a process to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time.

Synchronization is implemented using a concept called monitors. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Java programming language provides a very handy way of creating threads and synchronizing their task by using synchronized blocks. You keep shared resources within this block. Following is the general form of the synchronized statement:

synchronized (obj) {

//shared resources – variables and other process

}

Here, the obj is a reference to an object whose lock associates with the monitor that the synchronized statement represents.

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

# Example 5:

Following is a java program of multi-threading with synchronization where we send some messages:

|  |
| --- |
| //Java program to demonstrate synchronization.    import java.io.\*;  import java.util.\*;    //A Class used to send a message  Public class Sender {      public void send(String msg) {          System.out.println("Sending\t"  + msg );          try {              Thread.sleep(1000);          }          catch (Exception e) {              System.out.println("Thread interrupted.");          }          System.out.println("\n" + msg + "Sent");      }  }      //Class to send a message using Threads  Public class ThreadedSend extends Thread {      private String msg;      Sender  sender;        //Receives a message object and a string message to be sent      ThreadedSend(String m,  Sender obj) {          msg = m;          sender = obj;      }        public void run() {            //Only one thread can send a message at a time.          synchronized(sender) {                //Synchronizing the send              sender.send(msg);          }      }  }    // The driver class  Public class SyncDemo {      public static void main(String args[]) {          Sender snd = new Sender();          ThreadedSend S1 = new ThreadedSend( " Hi " , snd );          ThreadedSend S2 = new ThreadedSend( " Bye " , snd );            //Start two threads of ThreadedSend type          S1.start();          S2.start();            //Wait for threads to end          try{              S1.join();              S2.join();          }          catch(Exception e)          {              System.out.println("Interrupted");          }      }  } |

**Output:**

Sending Hi

Hi Sent

Sending Bye

Bye Sent

**Description:** In the above example, we chose to synchronize the Sender object inside the run() method of the ThreadedSend class. The output will be the same every time we run the program.

Alternately, we could define the whole send() block as synchronized and it would produce the same result. Then we don’t have to synchronize the Message object inside the run() method in ThreadedSend class.

//An alternate implementation to demonstrate

//that we can use synchronized with method also.

public class Sender {

    public synchronized void send(String msg) {

        System.out.println("Sending\t" + msg );

try {

            Thread.sleep(1000);

        }

        catch (Exception e) {

            System.out.println("Thread interrupted.");

        }

        System.out.println("\n" + msg + "Sent");

    }

}

If we do not want to synchronize a whole method, it is preferable to synchronize only part of a method. Java synchronized blocks inside methods makes this possible.

//One more alternate implementation to demonstrate

//that synchronized can be used with only a part of a method

Public class Sender {

    public void send(String msg) {

        synchronized(this) {

            System.out.println("Sending\t" + msg );

            try {

                Thread.sleep(1000);

            }

            catch (Exception e) {

                System.out.println("Thread interrupted.");

            }

            System.out.println("\n" + msg + "Sent");

        }

    }

}

Here is the summary view of synchronized block:

//Synchronize a method based on this object

public synchronized void methodA() { ...... }

//Synchronize a block of codes based on ‘this’ object

public void methodB() {

synchronized(this) { ...... }

}

//Synchronize a block of codes based on another object

synchronized(anObject) { ...... }

......

}

Also note that,

1. Synchronization can be controlled at method level or block level. Variables cannot be synchronized. You need to synchronize ALL the methods that access the variables.
2. **Daemon Thread**: There are two kinds of threads, daemon threads and user threads. A daemon thread can be set via the setDaemon(boolean on) method. A daemon thread is an infrastructure thread, e.g., the garbage collector thread and the GUI's event dispatcher thread. The JVM exits when the only threads running are all daemon threads. In other words, the JVM considers its job done, when there is no more user threads and all the remaining threads are its infrastructure threads.
3. When you decide to implement the **Runnable** interface to create a thread, you must override the run() method provided by Runnable interface. This method provides an entry point for the thread and you will put your complete business logic inside this method.
4. In multithreading, users are not blocked as threads are independent and can perform multiple operations at a time.
5. Synchronization is built around an internal entity known as the **lock** or monitor. Each and every object has a lock associated with it. So a thread that needs consistent access to an object’s fields needs to acquire the object’s lock before accessing them, and then release the lock when the work is done.

// The lock() method locks the Lock instance so that all threads calling lock() are //blocked until unlock() is executed.

public class Lock {

private boolean isLocked = false;

public synchronized void lock() throws InterruptedException {

while(isLocked) {

wait();

}

isLocked = true;

}

public synchronized void unlock() {

isLocked = false;

notify();

}

}

1. There are two way to create **Synchronized Arraylist** - 1) Collections.synchronizedList() method and 2) Using CopyOnWriteArrayList. Vector is synchronized and thread-safe and because of this, it is slightly slower than ArrayList. Functionality wise Vector synchronizes at the level of each individual operation. Generally a programmer likes to synchronize a whole sequence of operations. Synchronizing individual operations is both less safe and slower. Vectors are also considered obsolete and unofficially deprecated in java.
2. **Static methods** are the methods in Java that can be called without creating an object of class. They are referenced by the class name itself or reference to the Object of that class. e.g. Math.sqrt() – a method to get the square root in Math class. We are calling the method on the class name itself. As opposed to creating an object first and then calling the method on that object. That would be an instance method.