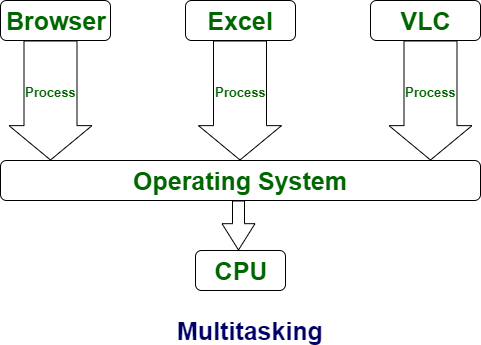
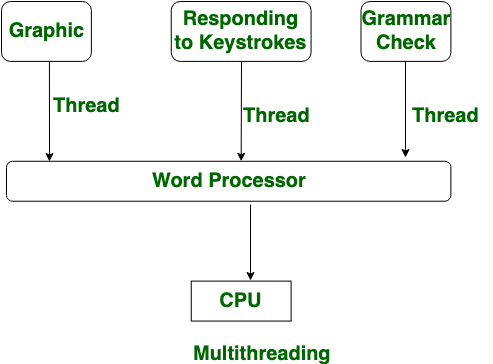
**UNIT-7(Threads):**

Threads allow a program to operate more efficiently by doing multiple things at the same time.

**Multitasking(or Multiprocessing)** lets the **CPU** perform various tasks simultaneously (threads, process, program, task), while **multithreading (with in Process)** helps in the execution of various threads in a **single process(Program)** simultaneously.

**Multithreading** is faster. In **multitasking**, termination of a process takes more time. While in **multithreading**, termination of thread takes less time.



* **A thread refers to the smallest unit of execution within a program.**
* We use multithreading than multiprocessing because threads share a common memory area.
* They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
* Java Multithreading is mostly used in **games**, **animation** etc.
* Java provides built-in support for multithreading through the **java.lang.Thread** class and the **java.lang.Runnable** interface.

**Creating Thread:**

We can create a new thread by either extending the **Thread** class or implementing the **Runnable** interface.

**Extending java.lang.Thread:**

* Extending the **Thread** class:

public class App {

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

         MyThread thread = new MyThread();

        thread.start(); // Start the thread

    }

}

class MyThread extends Thread {

    public void run() {

        // Code to be executed in the thread

        System.out.println("Thread is running.");

    }

}

Or,

public class App {

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

        Thread t1=new Thread() {

            public void run() {

                // Code to be executed in the thread

                System.out.println("Thread is running.");

            }

        };

        t1.start();

    }

}

**Implementing java.lang.Runnable:**

* Implementing the **Runnable** interface:

public class App {

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

        MyRunnable runnable = new MyRunnable();

        Thread thread = new Thread(runnable);

        thread.start(); // Start the thread

    }

}

class MyRunnable implements Runnable {

    public void run() {

        // Code to be executed in the thread

        System.out.println("Thread is running.");

    }

}

Or,

public class App {

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

        Runnable r1=new Runnable() {

            public void run() {

                // Code to be executed in the thread

                System.out.println("Thread is running.");

            }

        };

       Thread t1=new Thread(r1);

       t1.start();

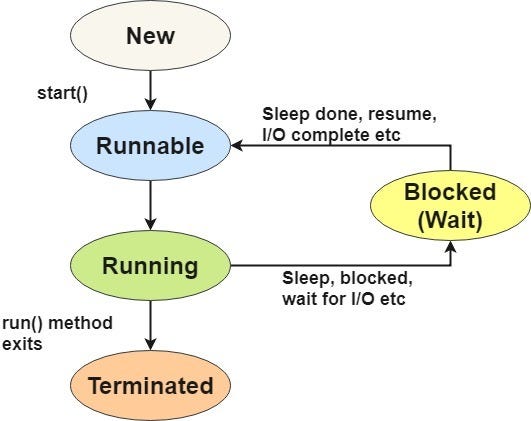
    }

}

**Life cycle of a Thread (Thread States)**

A thread can be in one of the five states. The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. **New**: Thread created, not yet started.
2. **Runnable**: Started, ready for execution.
3. **Running**: Currently executing its code.
4. **Non-Runnable (Blocked)**: Waiting for a condition to proceed.
5. **Terminated**: Finished its execution, cannot be restarted.



**Priority of a Thread (Thread Priority):**

Each thread have a priority. Priorities are represented by a number between 1 and 10. 3 constants defined in Thread class:

1. public static int MIN\_PRIORITY
2. public static int NORM\_PRIORITY
3. public static int MAX\_PRIORITY

Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10.

Example:

Lab: Write a simple java program to demonstrate how thread Priority handled.

public class App {

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

        TestMultiPriority1 m1=new TestMultiPriority1();

        TestMultiPriority1 m2=new TestMultiPriority1();

        m1.setPriority(Thread.MIN\_PRIORITY);

        m2.setPriority(Thread.MAX\_PRIORITY);

        m1.start();

        m2.start();

    }

}

class TestMultiPriority1 extends Thread{

public void run()

    {

    System.out.println(Thread.currentThread().getPriority());

    }

}

Output:

1

10

**Synchronization**

* Synchronization in Java is the capability to *control the access of multiple threads to any* ***shared resource****.*
* In a multithreaded environment, if multiple threads access and modify **shared resources** simultaneously, it can lead to problems such as data corruption, and unpredictable behavior.
* Java provides mechanisms for **synchronization** to help address these issues.

## **Types of Synchronization**

Synchronization is classified into two types

* Process Synchronization
* Thread Synchronization

#### ****Process Synchronization:****

Process Synchronization is a technique used to coordinate the execution of multiple processes. It ensures that the shared resources are safe and in order.

***Thread Synchronization***

There are two types of thread synchronization mutual exclusive and inter-thread communication.

* Mutual Exclusive
  + Synchronized method.
  + Synchronized block.
  + Static synchronization.
* Cooperation (Inter-thread communication in java)

Mutual Exclusive helps **keep threads from interfering with one another** while sharing data. It can be achieved by using the following three ways:

* By Using Synchronized Method
* By Using Synchronized Block
* By Using Static Synchronization

**Java synchronized method**

* It is a method that can be declared synchronized using the keyword **“synchronized”** before the method name.
* It is used for **locking** an object for any shared resources.
* The object gets the lock whenever the **synchronized method** is called.
* The lock does not release until the **thread completes** its function

Lab: Write java program to demonstrate Multithreading using synchronized Method.

public class App {

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

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

        MyThread1 t1=new MyThread1(obj);

        MyThread2 t2=new MyThread2(obj);

        t1.start();

        t2.start();

    }

}

class Table{

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

   for(int i=1;i<=5;i++){

     System.out.println(n\*i);

     try{

      Thread.sleep(400);

     }catch(Exception e)

{System.out.println(e);}

   }

 }

}

class MyThread1 extends Thread{

    Table t;

    MyThread1(Table t){

    this.t=t;

    }

    public void run(){

    t.printTable(5);

    }

}

class MyThread2 extends Thread{

    Table t;

    MyThread2(Table t){

    this.t=t;

    }

    public void run(){

    t.printTable(100);

    }

}

Output

100

200

300

400

500

5

10

15

20

25

**Synchronized Block**

If a block is declared as synchronized then the code which is written *inside a method is only executed instead of the whole code*. **It is used when sequential access to code is required**.

public class App {

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

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

        MyThread1 t1=new MyThread1(obj);

        MyThread2 t2=new MyThread2(obj);

        t1.start();

        t2.start();

    }

}

class Table{

 public void printTable(int n){//synchronized method

 synchronized(this){    //This is a synchronized block

     for(int i=1;i<=5;i++){

      System.out.println(n\*i);

      try{

       Thread.sleep(400);

      }catch(Exception e){System.out.println(e);}

     }

   }

 }

}

class MyThread1 extends Thread{

    Table t;

    MyThread1(Table t){

    this.t=t;

    }

    public void run(){

    t.printTable(5);

    }

}

class MyThread2 extends Thread{

    Table t;

    MyThread2(Table t){

    this.t=t;

    }

    public void run(){

    t.printTable(100);

    }

}

**inter-thread communication or Co-operation** is all about allowing synchronized threads to communicate with each other.

Inter-thread communication is the mechanism by which threads coordinate and exchange information in a multithreaded program. It allows threads to work together, share data, and synchronize their actions to achieve a specific task or goal. In Java, inter-thread communication can be accomplished using methods like **wait(), notify(), and notifyAll()** within synchronized blocks.

* wait()
* notify()
* notifyAll()

**wait():** Causes the current thread to release the lock it holds and wait until another thread calls notify() or notifyAll() on the same object.

**notify():** Wakes up one waiting thread (chosen arbitrarily) that's waiting on the same object. The awakened thread will continue once the notifying thread releases the lock.

**notifyAll():** Wakes up all waiting threads that are waiting on the same object. They compete for the lock once it's released by the notifying thread

Lab: Write a Java program to demonstrate **inter-thread communication or Co-operation** threading.

public class App {

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

       Customer c=new Customer();

       Thread t1= new Thread()

        {

            public void run()

            {

                c.withdraw(15000);

            }

        };

        t1.start();

       Thread t2= new Thread()

        {

            public void run()

            {

                c.deposit(10000);

            }

        };

        t2.start();

    }

}

class Customer{

int amount=10000;

    synchronized void **withdraw**(int amount)

    {

        System.out.println("going to withdraw...");

        if(this.amount<amount)

        {

        System.out.println("Less balance; waiting for deposit...");

        try

        {

            wait();

        }

        catch(Exception e){}

        }

        this.amount-=amount;

        System.out.println("withdraw completed...");

    }

    synchronized void **deposit**(int amount)

    {

    System.out.println("going to deposit...");

    this.amount+=amount;

    System.out.println("deposit completed... ");

    notify();

    }

}

**Thread Deadlock:**Deadlock in multithreading is a situation where two or more threads are unable to proceed with their execution because each is waiting for the other(s) to release a resource they need. This can lead to a situation where the program becomes unresponsive and stuck. Deadlocks are a common issue in multithreaded programming and can be challenging to identify and resolve.



public class App {

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

        final String resource1 = "Nepal";

        final String resource2 = "China";

        Thread t1=new Thread()

        {

            public void run()

            {

                synchronized(resource1)

                {

                    System.out.println("Thread 1: Locked Resource1");

                    try {

                        Thread.sleep(100);

                    } catch (Exception e) {

                    }

                    synchronized(resource2)

                    {

                        System.out.println("Thread 1: Locked Resource 2");

                    }

                }

            }

        };

        Thread t2=new Thread()

        {

            public void run()

            {

                synchronized(resource2)

                {

                    System.out.println("Thread 2: Locked Resource2");

                    try {

                        Thread.sleep(100);

                    } catch (Exception e) {

                    }

                    synchronized(resource1)

                    {

                        System.out.println("Thread 2: Locked Resource 1");

                    }

                }

            }

        };

        t1.start();

        t2.start();

    }

}

Output:

Thread 1: Locked Resource1

Thread 2: Locked Resource2

**Deadlock Solution:**

Let's change the order of the lock and run of the same program to see if both the threads still wait for each other.

public class App {

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

        final String resource1 = "Nepal";

        final String resource2 = "China";

        Thread t1=new Thread()

        {

            public void run()

            {

                synchronized(resource1)

                {

                    System.out.println("Thread 1: Locked Resource1");

                    try {

                        Thread.sleep(100);

                    } catch (Exception e) {

                    }

                    System.out.println("Thread 1: Waiting for lock 2...");

                    synchronized(resource2)

                    {

                        System.out.println("Thread 1: Locked Resource 2");

                    }

                }

            }

        };

        Thread t2=new Thread()

        {

            public void run()

            {

                synchronized(resource1)

                {

                    System.out.println("Thread 2: Locked Resource2");

                    try {

                        Thread.sleep(100);

                    } catch (Exception e) {

                    }

                    System.out.println("Thread 2: Waiting for lock 2...");

                    synchronized(resource2)

                    {

                        System.out.println("Thread 2: Locked Resource 1");

                    }

                }

            }

        };

        t1.start();

        t2.start();

    }

}

Output:

Thread 1: Locked Resource1

Thread 1: Waiting for lock 2...

Thread 1: Locked Resource 2

Thread 2: Locked Resource2

Thread 2: Waiting for lock 2...

Thread 2: Locked Resource 1

1.What is Thread Priority? Write a java program to demonstrate how thread Priority handled.

2. Why Synchronization is done in Multithreaded Program?Write java program to demonstrate Multithreading using synchronized Method.

3.What is inter-thread communication in Multithreading? Write a java program to demonstrate inter-thread communication in Multithreaded program.