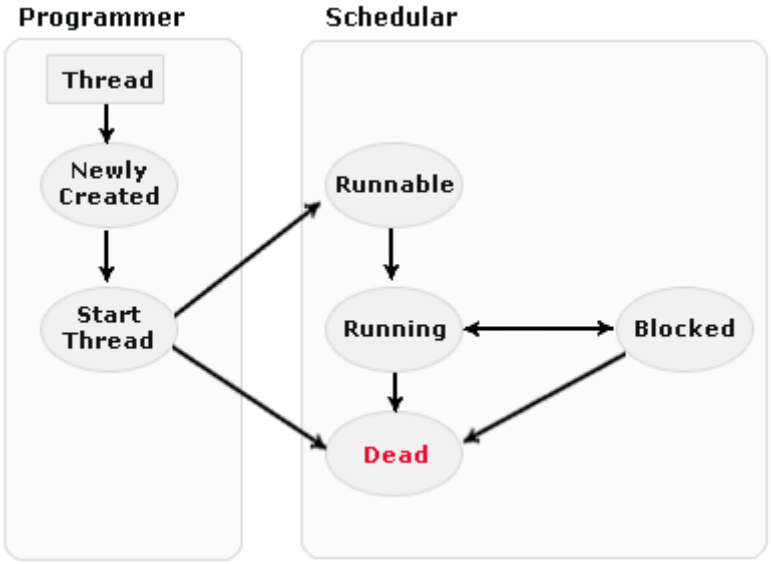
7. MULTITHREDED PROGRAMMING

❖ What is thread?

- A thread is a single sequential flow of control within a program.
- Thread is a path of the execution in a program.
- **Muti-Threading:**
  - Executing more than one thread (part of a task) parallel using a single processor is called multithreading
  - Thread: A thread can be loosely defined as a separate stream of execution that takes place simultaneously with and independently of everything else that might be happening.
  - A thread is like a classic program that starts at point A and executes until it reaches point B.
  - It does not have an event loop. A thread runs independently of anything else happening in the computer.
  - Without threads an entire program can be held up by one CPU intensive task or one infinite loop, intentional or otherwise.
  - With threads the other tasks that don't get stuck in the loop can continue processing without waiting for the stuck task to finish.
  - It turns out that implementing threading is harder than implementing multitasking in an operating system
  - The reason it's relatively easy to implement multitasking is that individual programs are isolated from each other. Individual threads, however, are not.
  - To return to the printing example, suppose that while the printing is happening in one thread, the user deletes a large chunk of text in another thread.
  - What's printed? The document as it was before the deletion? The document as it was after the deletion? The document with some but not all of the deleted text? Or does the whole system go down in flames? Most often in a non-threaded or poorly threaded system, it's the latter.
  - Threaded environments like Java allow a thread to put locks on shared resources so that while one thread is using data no other thread can touch that data.
  - This is done with synchronization. Synchronization should be used sparingly since the purpose of threading is defeated if the entire system gets stopped waiting for a lock to be released.
  - The proper choice of objects and methods to synchronize is one of the more difficult things to learn about threaded programming
  - **Creating threads using the thread class**

❖ Life Cycle of Thread

- When you are programming with threads, understanding the life cycle of thread is very valuable.
- While a thread is alive, it is in one of several states.
- By invoking start() method, it doesn?t mean that the thread has access to CPU and start executing straight away. Several factors determine how it will proceed
- **Different states of a thread are :**



- **New state:**
  - After the creations of Thread instance the thread is in this state but before the start() method invocation.
  - At this point, the thread is considered not alive.
- **Runnable (Ready-to-run) state:**
  - A thread start its life from Runnable state.
  - A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also.
  - On this state a thread is waiting for a turn on the processor.
- **Running state:**
  - A thread is in running state that means the thread is currently executing.
  - There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool.
- **Dead state:**
  - A thread can be considered dead when its run() method completes.
  - If any thread comes on this state that means it cannot ever run again.
- **Blocked:**
  - A thread can enter in this state because of waiting the resources that are hold by another thread.

❖ **Methods that can be applied apply on a Thread:**

- Some Important Methods defined in **java.lang.Thread** are shown in the table:

Method	Return Type	Description
currentThread( )	Thread	Returns an object reference to the thread in which it is invoked.
getName( )	String	Retrieve the name of the thread object or instance.
start( )	void	Start the thread by calling its run method.
run( )	void	This method is the entry point to execute thread, like the main method for applications.
sleep( )	void	Suspends a thread for a specified amount of time (in milliseconds).
isAlive( )	boolean	This method is used to determine the thread is running or not.
activeCount( )	int	This method returns the number of active threads in a particular

		thread group and all its subgroups.
interrupt( )	void	The method interrupt the threads on which it is invoked.
yield( )	void	By invoking this method the current thread pause its execution temporarily and allow other threads to execute.
join( )	void	This method and join(long millisec) Throws InterruptedException. These two methods are invoked on a thread. These are not returned until either the thread has completed or it is timed out respectively.

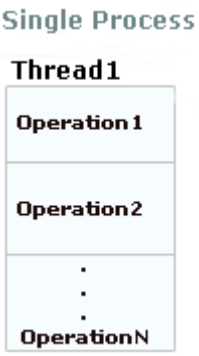
❖ Process

- A **process** is an instance of a computer program that is executed sequentially.
- It is a collection of instructions which are executed simultaneously at the rum time.
- Thus several processes may be associated with the same program.
- For example, to check the **spelling** is a single process in the **Word Processor** program and you can also use other processes like **printing, formatting, drawing, etc**. associated with this program

❖ The Java thread model

➢ Thread

- A thread is a **lightweight** process which exist within a program and executed to perform a special task.
- Several threads of execution may be associated with a single process.
- Thus a process that has only one thread is referred to as a **single-threaded** process, while a process with multiple threads is referred to as a **multi-threaded** process.
- In Java Programming language, thread is a sequential path of code execution within a program.
- Each thread has its own local variables, program counter and lifetime.
- In single threaded runtime environment, operations are executes sequentially i.e. next operation can execute only when the previous one is complete.
- It exists in a common memory space and can share both data and code of a program
- Threading concept is very important in Java through which we can increase the speed of any application.
- You can see diagram shown below in which a thread is executed along with its several operations with in a single process.



❖ The main thread

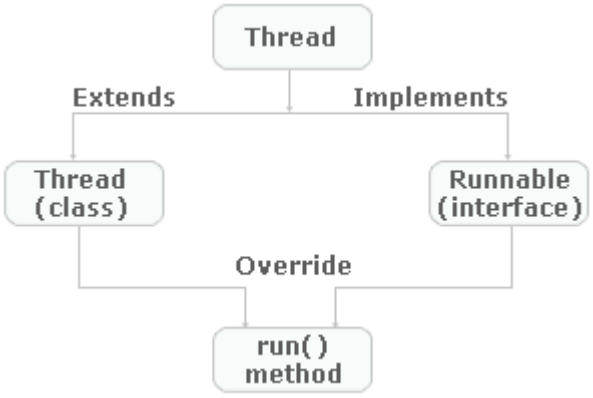
- When any standalone application is running, it firstly execute the **main()** method runs in a one thread, called the main thread.
- If no other threads are created by the main thread, then program terminates when the main() method complete its execution.
- The main thread creates some other threads called child threads.
- The main() method execution can finish, but the program will keep running until the all threads have complete its execution.

❖ Features of Threads:

1. Creating a Thread
2. Context Switch
3. Priorities
4. Synchronization
5. Messaging

❖ Creating a thread

- In Java, an object of the Thread class can represent a thread.
- Thread can be implemented through any one of two ways:
  - **Extending the java.lang.Thread Class**
  - **Implementing the java.lang.Runnable Interface**



➢ Extending the java.lang.Thread Class

- For creating a thread a class have to extend the Thread Class. For creating a thread by this procedure you have to follow these steps:
  - Extend the **java.lang.Thread** Class.
  - Override the **run( )** method in the subclass from the Thread class to define the code executed by the thread.
  - Create an **instance** of this subclass. This subclass may call a Thread class constructor by subclass constructor.
  - Invoke the **start( )** method on the instance of the class to make the thread eligible for running.
- **Example:- Single thread creation extending the "Thread" Class:**

```
class MyThread extends Thread
{
        String s=null;
        MyThread(String s1)
        {
                s=s1;
```

```
                start();
        }
        public void run()
        {
                System.out.println(s);
        }
}
public class RunThread
{
        public static void main(String args[])
        {
                MyThread m1=new MyThread("Thread started....");
        }
}
```

- **Implementing the java.lang.Runnable Interface**
  - The procedure for creating threads by implementing the Runnable Interface is as follows:
    - A Class implements the **Runnable** Interface, override the run() method to define the code executed by thread. An object of this class is Runnable Object.
    - Create an object of **Thread** Class by passing a Runnable object as argument.
    - Invoke the **start( )** method on the instance of the Thread class.
    - **Example: Thread creation implenting the Runnable interface:**

```
class MyThread1 implements Runnable
{
        Thread t;
        String s=null;
        MyThread1(String s1)
        {
                s=s1;
                t=new Thread(this);
                t.start();
        }
        public void run()
        {
                System.out.println(s);
        }
}
public class RunableThread
{
        public static void main(String args[])
        {
                MyThread1 m1=new MyThread1("Thread started....");
        }
}
```

  - There are two reasons for implementing a Runnable interface preferable to extending the Thread Class.
    - If you extend the **Thread** Class, that means that subclass cannot extend any other Class, but if you implement **Runnable** interface then you can do this.
    - The class implementing the **Runnable** interface can avoid the full overhead of **Thread** class which can be excessive.

❖ Creating multiple threads

- Like creation of a single thread, You can also create more than one thread (multithreads) in a program using class **Thread** or implementing interface **Runnable**.
- Example:- **implenting the Runnable interface:**
-

```
class MyThread1 implements Runnable
{
        Thread t;
        String s=null;
        MyThread1(String s1)
        {
                s=s1;
                t=new Thread(this);
                t.start();
        }
        public void run()
        {
                for(int i=0;i<5;i++)
                {
                        System.out.println("Thread-->" + s + " " + i);
                }
        }
}
public class MultipleThread
{
        public static void main(String args[])
        {
                new MyThread1("One"); // start threads
                new MyThread1("Two");
                new MyThread1("Three");
        }
}
```

❖ Using isAlive( ) & join( )

- isAlive() method is used to check whether thread is running or not. it returns true if it is running otherwise it returns false.
- ***final boolean isAlive( )***
- Java thread's join() method is used to join the next thread at the end of current thread. When the execution of current thread stops the next thread executes.
- The join() method in Java Thread class has three overloaded method,
  - **void join();**
  - **void join(long timeout);**
  - **void join(long timeout, int nanoseconds);**
- void join()
  - This join method accepts no arguments and forces the thread to wait till it dies.
- void join(long timeout)
  - This join method accepts only one argument as timeout. The timeout value is milliseconds. This method forces the thread to wait for given milliseconds for the completion of the specified thread.
- void join(long timeout, int nanoseconds)

- This join method accepts two arguments. First as timeout in milliseconds and second an integer value for nanoseconds. The amount of timeout in milliseconds is along with the nanoseconds. This method also forces the thread to wait for the given time.

➢ **Example:**

```
public class Join1 implements Runnable
{
    public void run()
    {
            Thread t = Thread.currentThread();
            //Tests if this thread is alive
            System.out.println(t.getName()+" is alive "+t.isAlive());
    }
    public static void main(String args[]) throws Exception
    {
            Thread t = new Thread(new Join1());
            t.start();
            //Waits for this thread to die.
            t.join();
            //Tests if this thread is alive
            System.out.println(t.getName()+" is alive "+t.isAlive());
    }
}
```

❖ Thread priorities

- ➢ Every thread created has some priority.
- ➢ Threads are executed according to their priority.
- ➢ Threads with higher priority are executed before threads with lower priority. A newly created thread has same priority as the thread that creates it.
- ➢ Java thread priorities are integer values ranging from 1 to 10.
- ➢ 1 is the lowest priority and 10 is the highest priority.
- ➢ The default priority of a thread is 5.
- ➢ The thread priorities are assigned to some constant literals as follows,
  - Thread.MIN_PRIORITY : It is the minimum priority of any thread i.e. 1
  - Thread.MAX_PRIORITY : It is the maximum priority of any thread i.e. 10
  - Thread.NORM_PRIORITY : It is the normal and default priority of any thread i.e. 5
- ➢ To set the thread priorities we can use setPriority() method. Its syntax is,
  - **final void** setPriority(**int** level);
- ➢ To obtain the current priority of a thread we can use getPriority() method. Its syntax is,
  - **final int** getPriority();

- **Example:**

```
public class priority implements Runnable
{
        public void run()
        {
                for (int i=1;i<=3;i++)
                {
                        System.out.println(i + " This is thread " +
                        Thread.currentThread().getName());
                }
        }
        public static void main(String[] args)
        {
                Thread t1 = new Thread(new priority(), "Thread  A");
                Thread t2 = new Thread(new priority(), "Thread  B");
                Thread t3 = new Thread(new priority(), "Thread  C");
                t1.setPriority(10);
                t1.start();
                t2.start();
                t3.start();
        }
}
```

❖ Synchronization

- Synchronization is the capabilility of control the access of multiple threads to any shared resource.
- Synchronization is better in case we want only one thread can access the shared resource at a time.
- **Why use Synchronization?**
  - The synchronization is mainly used to
    1. To prevent thread interference.
    2. To prevent consistency problem.

- **Types of Synchronization**
  - There are two types of synchronization
    1. Process Synchronization
    2. Thread Synchronization

Here we will learn Thread Synchronization

- Only methods (or blocks) can be synchronized, Classes and variable cannot be synchronized.
- Each object has just one lock.
- All methods in a class need not to be synchronized. A class can have both synchronized and non-synchronized methods.
- If two threads wants to execute a synchronized method in a class, and both threads are using the same instance of the class to invoke the method then only one thread can execute the method at a time.
- If a class has both synchronized and non-synchronized methods, multiple threads can still access the class's non-synchronized methods. If you have methods that don't access the data you're trying to protect, then you don't need to synchronize them.

- Synchronization can cause a hit in some cases (or even deadlock if used incorrectly), so you should be careful not to overuse it.
- If a thread goes to sleep, it holds any locks it has.it doesn't release them.
- A thread can acquire more than one lock. For example, a thread can enter a synchronized method, thus acquiring a lock, and then immediately invoke a synchronized method on a different object, thus acquiring that lock as well. As the stack unwinds, locks are released again.
- You can synchronize a block of code rather than a method.
- Constructors cannot be synchronized

➢ In non synchronized multithreaded application, it is possible for one thread to modify a shared object while another thread is in the process of using or updating the object's value.

➢ **There are two ways to synchronized the execution of code:**

- **Exampl**e: Synchronized Methods

```
class A extends Thread
{
    public void run()
    {
            display();
    }
    public static  synchronized void display()
    {
            int i;
            for( i=0;i<10;i++)
            {
                    try
                    {
                            sleep(1000);
                            System.out.println(Thread.currentThread().getName()+" "+i);
                    }
                    catch (Exception e)
                    {
                             System.out.println(e);
                    }
                    }
            }
}
class Sn1
{
    public static void main(String[] args)
    {
            A a=new A();
            Thread t1=new Thread(a);
            Thread t2=new Thread(a);
            t1.start();
            t2.start();
    }
}
```

- **Example:**

```
class A extends Thread
{
    public void run()
```

```
    {
        synchronized(this)    // synchronized block
        {
            int i;
            for( i=0;i<10;i++)
            {
            try
            {
                sleep(1000);
                System.out.println(Thread.currentThread().getName()+" "+i);
            }
            catch (Exception e)
            {
            }
            }
        }
    }
}
class Sn
{
    public static void main(String[] args)
    {
        A a=new A();
        Thread t1=new Thread(a);
        Thread t2=new Thread(a);
        t1.start();
        t2.start();
    }
}
```

❖ Inter thread communication

- Java provides a very efficient way through which multiple-threads can communicate with each-other.
- This way reduces the CPU?s idle time i.e. A process where, a thread is paused running in its critical region and another thread is allowed to enter (or lock) in the same critical section to be executed. This technique is known as **Interthread communication** which is implemented by some methods.
- These methods are defined in "**java.lang**" package and can only be called within synchronized code shown as:

Method	Description
wait( )	It indicates the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls method **notify()** or **notifyAll()**.
notify( )	It wakes up the first thread that called **wait()** on the same object.
notifyAll( )	Wakes up (Unlock) all the threads that called **wait( )** on the same object. The highest priority thread will run first.

- All these methods must be called within a try-catch block.
- **Example:**

```
class MyOwnRunnable implements Runnable
{
        private Object o;
        public MyOwnRunnable( Object o)
        {
                this.o = o;
        }
        public void run()
        {
                synchronized (o)
                {
                        for (int i = 0; i < 10; i++)
                        {
                                System.out.println(Thread.currentThread().getName()+" "+i);
                        }
                        o.notifyAll();
                }
        }
}
public class Wait1
{
        public static void main(String[] args)
        {
                try
                {
                        Object o = new Object();
                        MyOwnRunnable  a =new MyOwnRunnable(o);
                        Thread thread1 = new Thread(a);
                        Thread thread2 = new Thread(a);
                        Thread thread3 = new Thread(a);
                        synchronized (o)
                        {
```

```
                        thread1.start();
                        o.wait();
                        thread2.start();
                        o.wait();
                        thread3.start();
                    }
                }
                catch (InterruptedException e)
                {
                    e.printStackTrace();
                }
            }
        }
```

❖ Suspending, resuming, stopping threads

- While the suspend( ), resume( ), and stop( ) methods defined by **Thread** class seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs and obsolete in newer versions of Java.
- The following example illustrates how the wait( ) and notify( ) methods that are inherited from Object can be used to control the execution of a thread.
- This example is similar to the program in the previous section. However, the deprecated method calls have been removed. Let us consider the operation of this program.
- The NewThread class contains a boolean instance variable named suspendFlag, which is used to control the execution of the thread. It is initialized to false by the constructor.
- The run( ) method contains a synchronized statement block that checks suspendFlag. If that variable is true, the wait( ) method is invoked to suspend the execution of the thread. The mysuspend( ) method sets suspendFlag to true. The myresume( ) method sets suspendFlag to false and invokes notify( ) to wake up the thread. Finally, the main( ) method has been modified to invoke the mysuspend( ) and myresume( ) methods.

- **Example:**

```
class MyThread implements Runnable
{
        Thread t;
        boolean suspended;
        boolean stopped;
        MyThread(String name)
        {
                t = new Thread(this, name);
                suspended = false;
                stopped = false;
                t.start();
        }
        public void run()
        {
                try
                {
                        for (int i = 1; i < 10; i++)
                        {
                                System.out.print("\t$");
                                Thread.sleep(50);
                                synchronized (this)
                                {
                                        while (suspended)
                                                wait();
                                        if (stopped)
                                                break;
                                }
                        }
                }
                catch (InterruptedException exc)
                {
                        System.out.println(t.getName() + " interrupted.");
                }
                System.out.println("\n" + t.getName() + " exiting.");
        }

        synchronized void stop()
        {
                stopped = true;
                suspended = false;
                notify();
        }
        synchronized void suspend()
        {
                suspended = true;
        }
        synchronized void resume()
        {
                suspended = false;
                notify();
```

```
        }
}
public class Main
{
        public static void main(String args[]) throws Exception
        {
                MyThread mt = new MyThread("MyThread");
                Thread.sleep(100);
                mt.suspend();
                Thread.sleep(100);
                mt.resume();
                Thread.sleep(100);
                mt.suspend();
                Thread.sleep(100);
                mt.resume();
                Thread.sleep(100);
                mt.stop();
        }
}
```