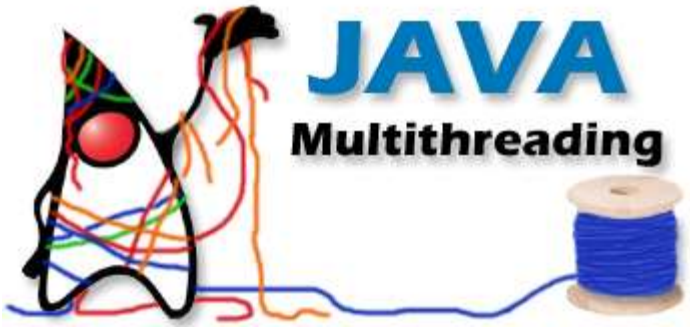
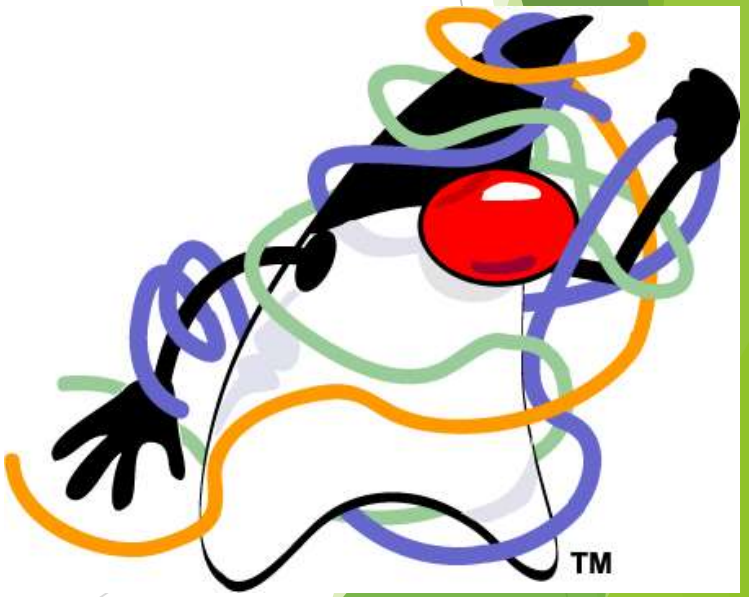
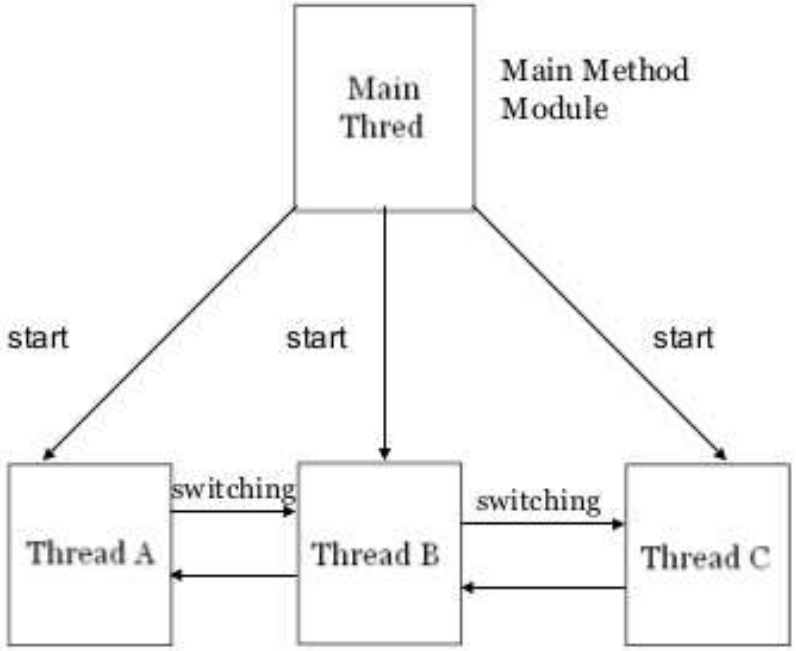


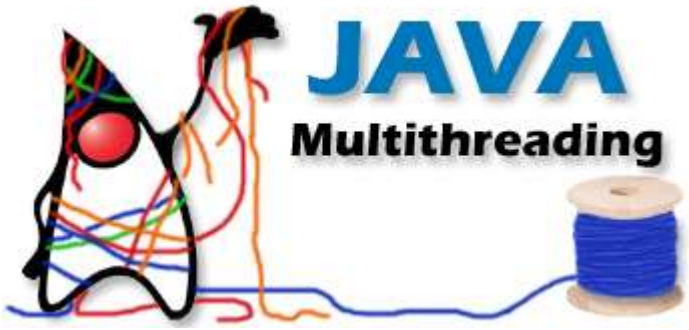
Multithreading in Java



A Multithreaded Program



- Single-threaded systems use an approach called an **event loop with polling.**
- **Multithreading in java** is a process of executing multiple threads simultaneously.
- Thread is basically a lightweight sub-process, a smallest unit of processing.



Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.

2) You **can perform many operations together so it saves time.**

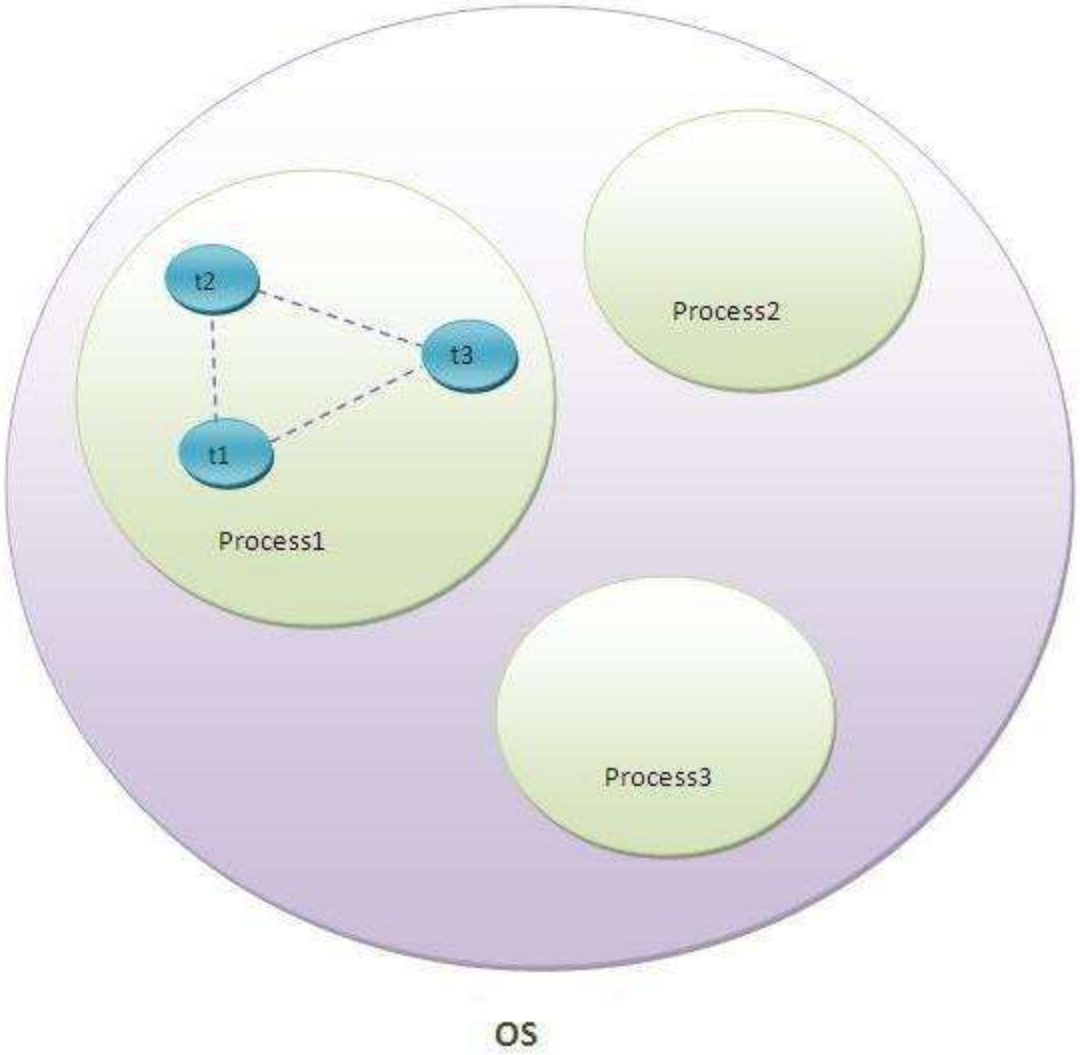
3) Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

What is Thread in java

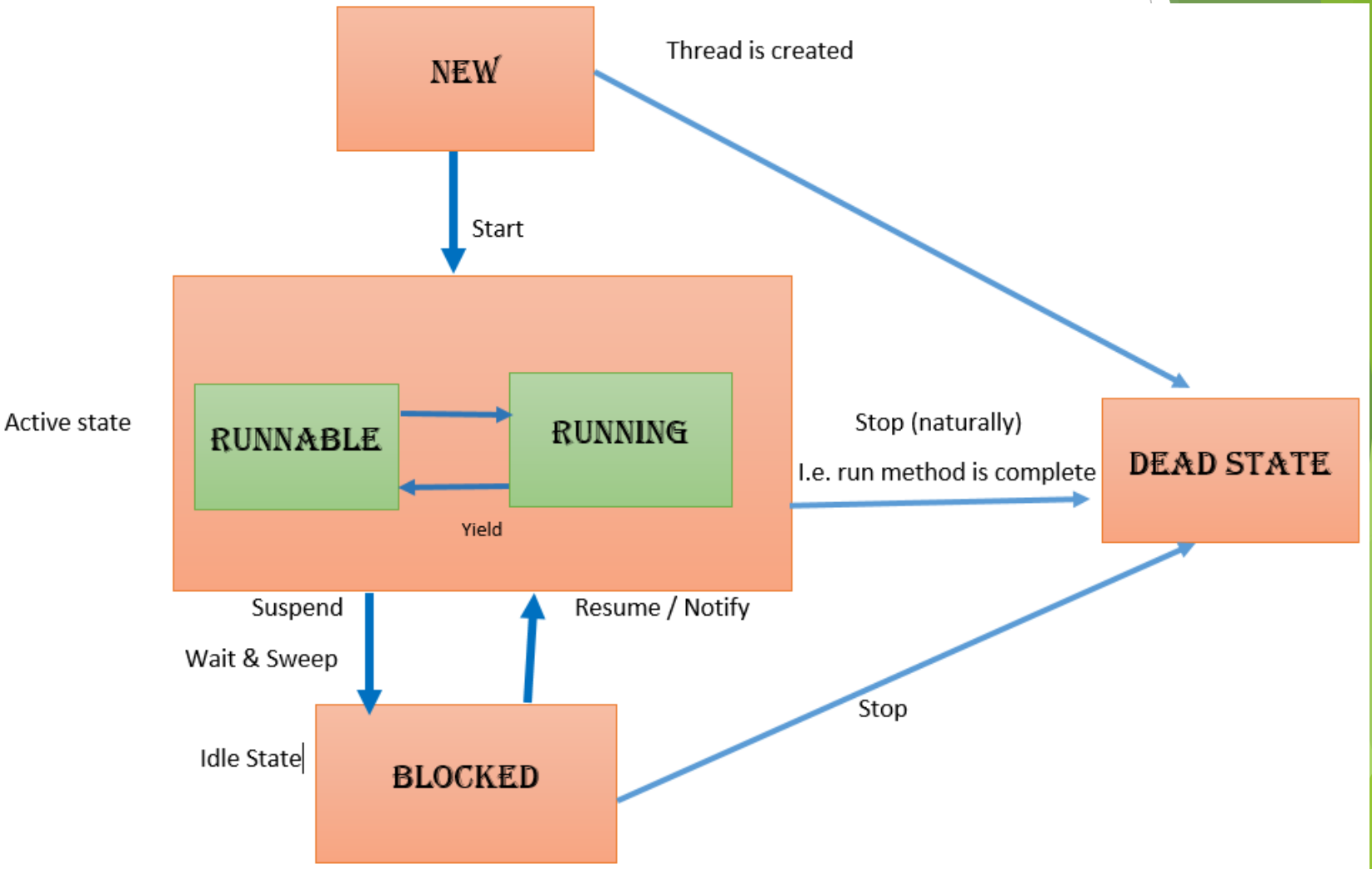
- A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.
- Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.

Messaging

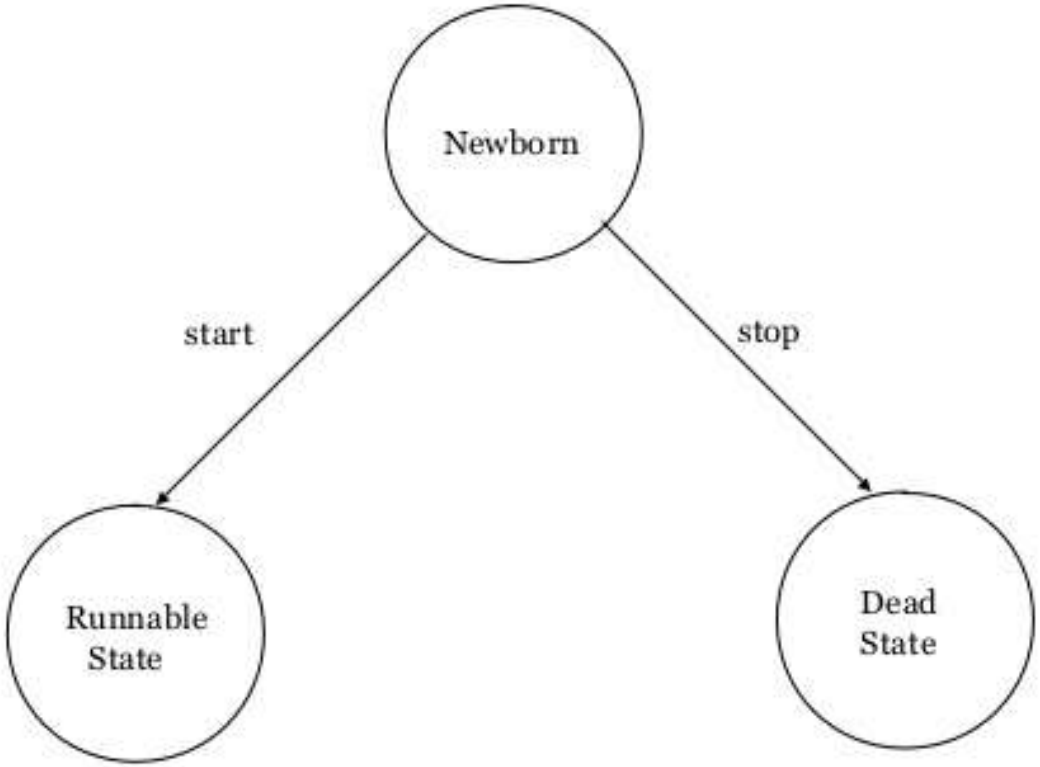
- When programming with some other languages, you must depend on the operating system to establish communication between threads
- By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.


t2
t3
t1
Process1
Process2
Process3
OS

Life cycle of a Thread



Scheduling a Newborn Thread



The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

- New
- Runnable
- Running
- Non-Runnable (Blocked)
- Terminated

1) New

- The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

- The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

- The thread is in running state if the thread scheduler has selected it.

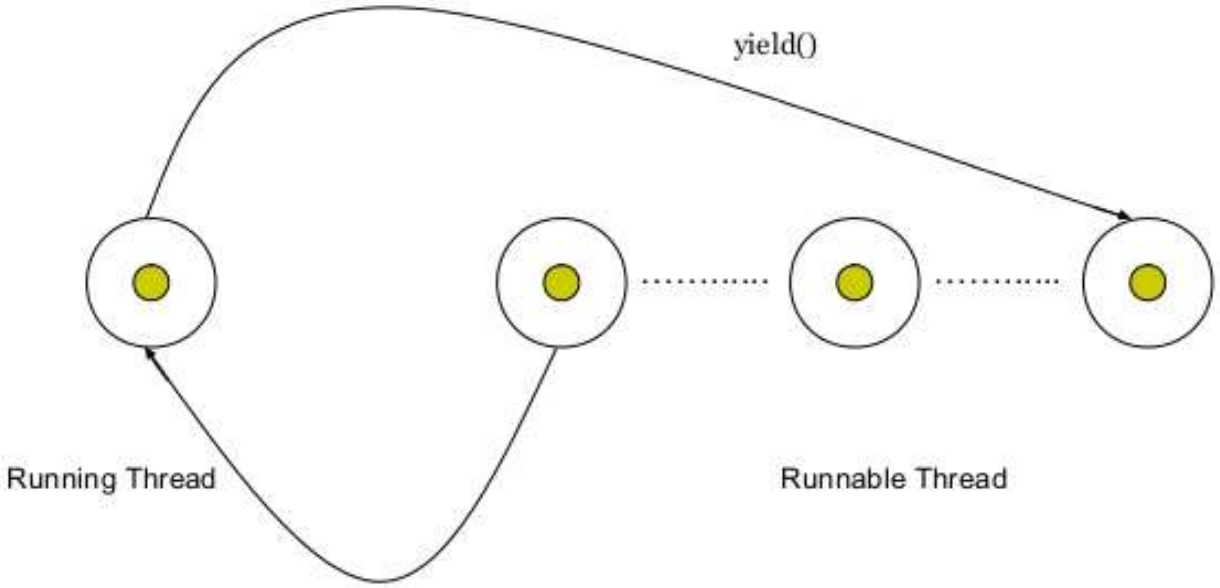
4) Non-Runnable (Blocked)

- This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

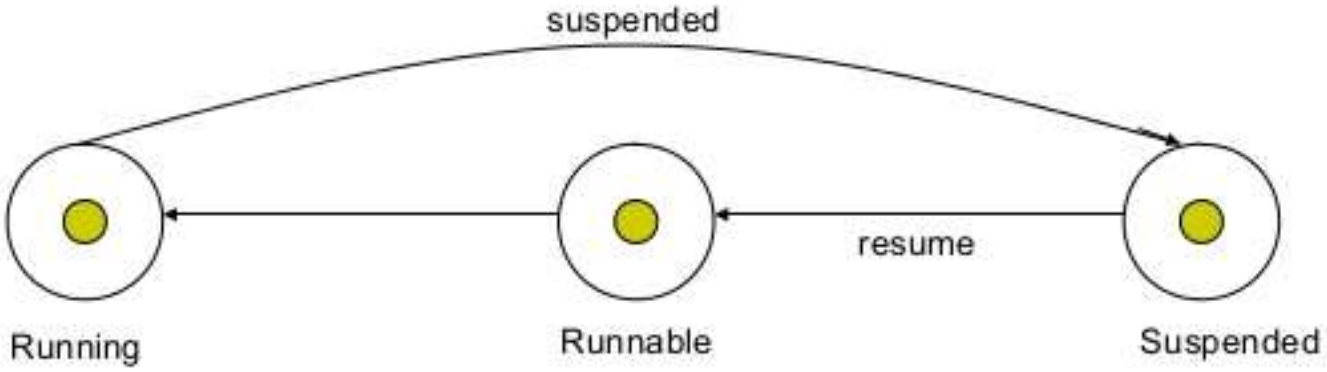
- A thread is in terminated or dead state when its run() method exits.

Releasing Control Using yield()



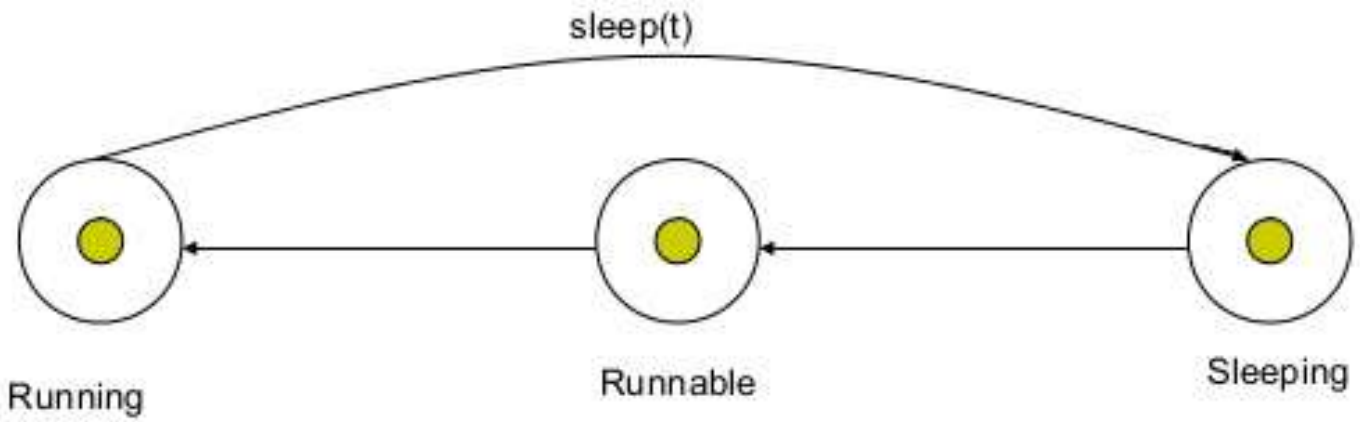
1. Suspend() and resume() Methods:-

This approach is useful when we want to suspend a thread for some time due to certain reason, but do not want to kill it.

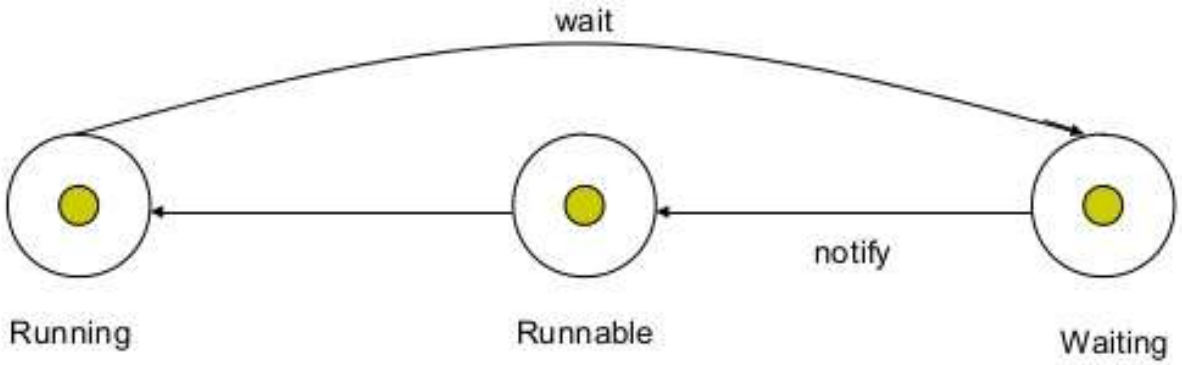


2. Sleep() Method :-

This means that the thread is out of the queue during this time period. The thread re-enter the runnable state as soon as this time period is elapsed.



3. <u>Wait() and notify() methods</u> :- blocked until certain condition occurs



How to create thread

By extending Thread class

By implementing Runnable interface.

Thread class:

- Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

1. Thread()
2. Thread(String name)
3. Thread(Runnable r)
4. Thread(Runnable r,String name)

```
class A extends Thread
{
        public void run()
                {for (int i=0;i<=5;i++)
System.out.println("In thread A"+i);
System.out.println("Exiting thread A");
                }
}

class B extends Thread {
public void run()
{for (int j=0;j<=5;j++)
System.out.println("In thread B"+j);
System.out.println("Exiting thread B");
}
}
```

```
class ThreadDemo
{
public static void main (String args[])
{
A oba = new A();
oba.start();
B obb=new B();
obb.start();
}
}
```

Common methods of thread class

- **public void run():** is used to perform action for a thread.
- **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
- **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- **public void join():** waits for a thread to die.
- **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
- **public int getPriority():** returns the priority of the thread.
- **public int setPriority(int priority):** changes the priority of the thread.
- **public String getName():** returns the name of the thread.
- **public void setName(String name):** changes the name of the thread.

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

- public void run(): is used to perform action for a thread

Using Runnable interface

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Main Thread

```
class CurrentThreadDemo {
 public static void main(String args[]) {
 Thread t = Thread.currentThread();
 System.out.println("Current thread: " + t);
 // change the name of the thread
 t.setName("My Thread");
 System.out.println("After name change: " +
t);
 try { for(int n = 5; n > 0; n--) {
System.out.println(n);
 Thread.sleep(1000); }
 } catch (InterruptedException e) {
System.out.println("Main thread
interrupted"); }
 }}
```

- Current thread: Thread[main,5,main]
- After name change: Thread[My Thread,5,main]
- 5
- 4
- 3
- 2
- 1

isAlive() and join()

isAlive()

- The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise

final boolean isAlive( )

join()

- This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it.
- Additional forms of **join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

final void join( ) throws InterruptedException

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
  }
}

class DemoJoin {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");
    NewThread ob3 = new NewThread("Three");
```

```
    System.out.println("Thread One is alive: "
                       + ob1.t.isAlive());
    System.out.println("Thread Two is alive: "
                       + ob2.t.isAlive());
    System.out.println("Thread Three is alive: "
                       + ob3.t.isAlive());
    // wait for threads to finish
    try {
      System.out.println("Waiting for threads to finish.");
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    System.out.println("Thread One is alive: "
                       + ob1.t.isAlive());
    System.out.println("Thread Two is alive: "
                       + ob2.t.isAlive());
    System.out.println("Thread Three is alive: "
                       + ob3.t.isAlive());

    System.out.println("Main thread exiting.");
  }
}
```

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
```

```
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.

In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

setPriority()

To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**.

This is its general form:

final void setPriority(int *level*)

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5. These priorities are defined as **static final** variables within **Thread**.

getPriority()

- You can obtain the current priority setting by calling the **getPriority( )** method of **Thread** :
- final int getPriority( )

THANKSSSS!!!!!!!!!!!!