The Basics of Multiple Threads in Java:

A thread is a single sequential flow of control-each thread within a program becomes its own thread of execution. This is how an application can execute multiple threads concurrently. The most critical advantage of doing so is maximizing CPU power and throughput.

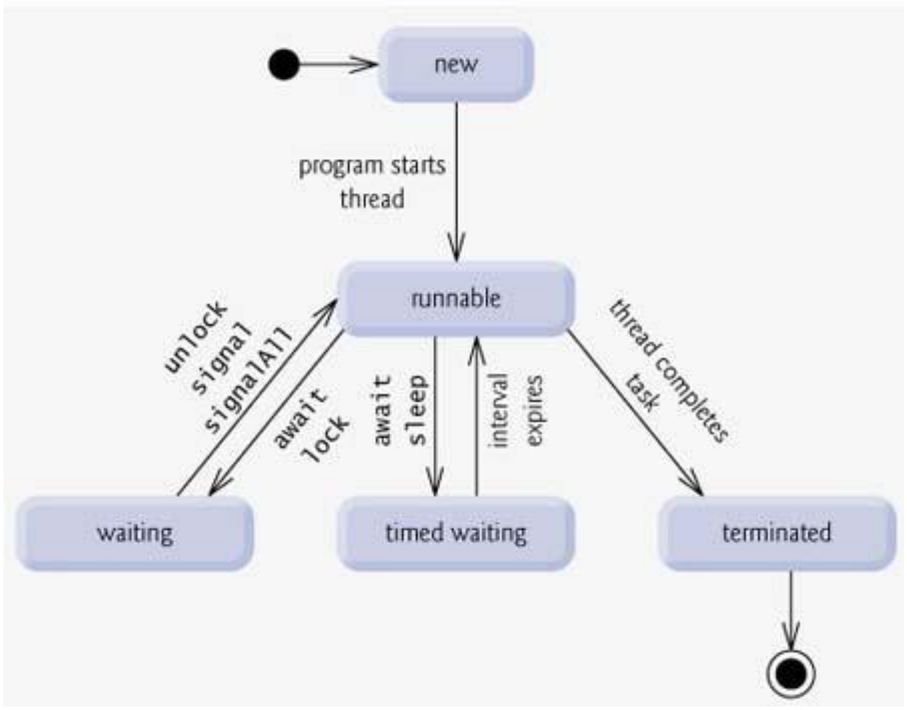
The simplest way to imagine two threads is to consider the following mathematical equation: 2 + 5 + 9 + 3. The entire equation is composed of three addition operations. The sequential solution involves one thread calculating 2 + 5 = 7, and then adding 9 to this, which gives 16, and ultimately adding 3. The end result is 19.

Assume that one addition takes one second. The entire process took three whole seconds (hence 2 + 5 = 7; 7 + 9 = 16; 16 + 3 = 19).

Suppose we have three threads. While the first thread calculates 2 + 5, the second calculates 9+3. Assuming that both of them require equally one second to calculate, this means both results are calculated in the first second. Now the first thread finally adds those two results and finds out that 7 + 12 is 19. This addition once again took another second. But the entire calculation process took a total of two seconds. It ran 33 percent faster than the original process.

Life Cycle of a Thread:

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



Above mentioned stages are explained here:

- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting:** Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

A traditional single-threaded application runs into a point where it "takes time" to accomplish a task, and then the entire program halts until that operation is completed, at which time the thread is able to move further. With multi-threading execution, if one thread "takes time," it won't block the flow of execution of other threads.

We should also note that if one of the threads modifies a thread-specific variable, then all of the threads will end up "noticing" and working with the new value. Another way of threading is Synchronization which is a process of controlling the access of shared resources by the multiple threads in such a manner that only one thread can access a particular resource at a time. A thread invokes the join() method on another thread in order to wait for the other thread to complete its execution. Threads are always in one of the following states: running, resumed, ready to run, suspended, or blocked.

The first is self-explanatory. The second is a kind of "ready to run" state after being suspended or blocked. A thread in the third state is obviously ready but has not started yet. The fourth state means the thread is temporarily suspended, while a thread in the final state is waiting for something.

Each thread also has a given priority. These are numeric values ranging from 1 to 10. The default priority level is 5 (NORM_PRIORITY constant). A higher priority thread during execution has priority over a lower valued one. The highest priority level, 10, is referred to by the constant MAX_PRIORITY, while the lowest 1 is akin to MIN_PRIORITY. These values are final static int's (meaning "constants" in everyday coding language). You can modify a thread's priority at any time after its creation using the setPriority() method and retrieve the thread priority value using getPriority() method.

Under Java there are two possible ways to create threads. The first route involves doing a simple inheritance from the Thread class. The Thread class can be found inside the java.lang package. The second way is implementing the Runnable interface.

Thread inheritance:

```
import java.util.Date;

public class Th1 extends Thread{
private int a;
public Th1(int a){
this.a = a;}
public void run(){
for (int i = 1; i <= a; ++i){
System.out.println(getName() + " is " + i+" at "+new Date());
try{
sleep(1000); }
catch(InterruptedException e){}
}}
public static void main(String args[]){
Th1 thr1, thr2;
thr1 = new Th1(2);
thr2 = new Th1(3);

thr1.start();
thr2.start();}}
```

Output:

```
Thread-1 is 1 at Mon Jan 01 14:26:34 AST 2024
Thread-0 is 1 at Mon Jan 01 14:26:34 AST 2024
Thread-0 is 2 at Mon Jan 01 14:26:35 AST 2024
Thread-1 is 2 at Mon Jan 01 14:26:35 AST 2024
Thread-1 is 3 at Mon Jan 01 14:26:36 AST 2024
```

In the previous code we have explicitly called the start() method of the two threads. This is one way of doing it. However, some people prefer launching the start() method instantaneously as soon as the object instance is created (basically, a new thread is initialized). You can do this by calling the start() method from the constructor.

```
public Th1(int a){

this.a = a;

start(); // this is the extra line, it starts the threads!

}
```

The Runnable interface:

The Java programming language does not allow multiple inheritances. This is the major pitfall of this solution. This is why the other solution, which implements the Runnable interface, was designed. The Runnable interface defines only one method, and that's the run() method. Once again we need to code this method when implementing the Runnable interface. Right after, in order to create multiple threads, we first need to create Runnable object instances, and then we're going to pass-by-value the objects to the Thread constructor.

```
 import java.util.Date;

class MyRunnable implements Runnable{
private int a;
public MyRunnable(int a){
this.a = a;}
public void run(){
for (int i = 1; i <= a; ++i){
System.out.println(Thread.currentThread().getName() + " is " + i + " at "+ new Date());
try{
Thread.sleep(1000);}
catch (InterruptedException e){}
}}

public static void main(String args[]){
MyRunnable thr1, thr2;
thr1 = new MyRunnable(2);
thr2 = new MyRunnable(3);
Thread t1 = new Thread(thr1);
Thread t2 = new Thread(thr2);
t1.start();
t2.start();}}
```

Example:

```
import java.util.Date;


class MyRunnable implements Runnable{
private int a;
static int c=0; // if c not static then every object has its own c
public MyRunnable(int a){
this.a = a;
}
public void run(){
for (int i = 1; i <= a; ++i){
		if(Thread.currentThread().getName().equals("T1"))
		System.out.println(Thread.currentThread().getName() + "  " + this.c++);
		else
		System.out.println(Thread.currentThread().getName() + "  " + this.c);
try{
Thread.sleep(1000);
}
catch (InterruptedException e){}
}
}

public static void main(String args[]){
MyRunnable thr1, thr2;
thr1 = new MyRunnable(2);
thr2 = new MyRunnable(3);
Thread t1 = new Thread(thr1,"T1");
Thread t2 = new Thread(thr2,"T2");


t1.start();
t2.start();
t2.setPriority(1);

}
}
```

Thread Methods:

Following is the list of important methods available in the Thread class.

SN	Methods with Description
1	**public void start()** Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	**public void run()** If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	**public final void setName(String name)** Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	**public final void setPriority(int priority)** Sets the priority of this Thread object. The possible values are between 1 and 10.
5	**public final void setDaemon(boolean on)** A parameter of true denotes this Thread as a daemon thread.
6	**public final void join(long millisec)** The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	**public void interrupt()** Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	**public final boolean isAlive()** Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object

. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

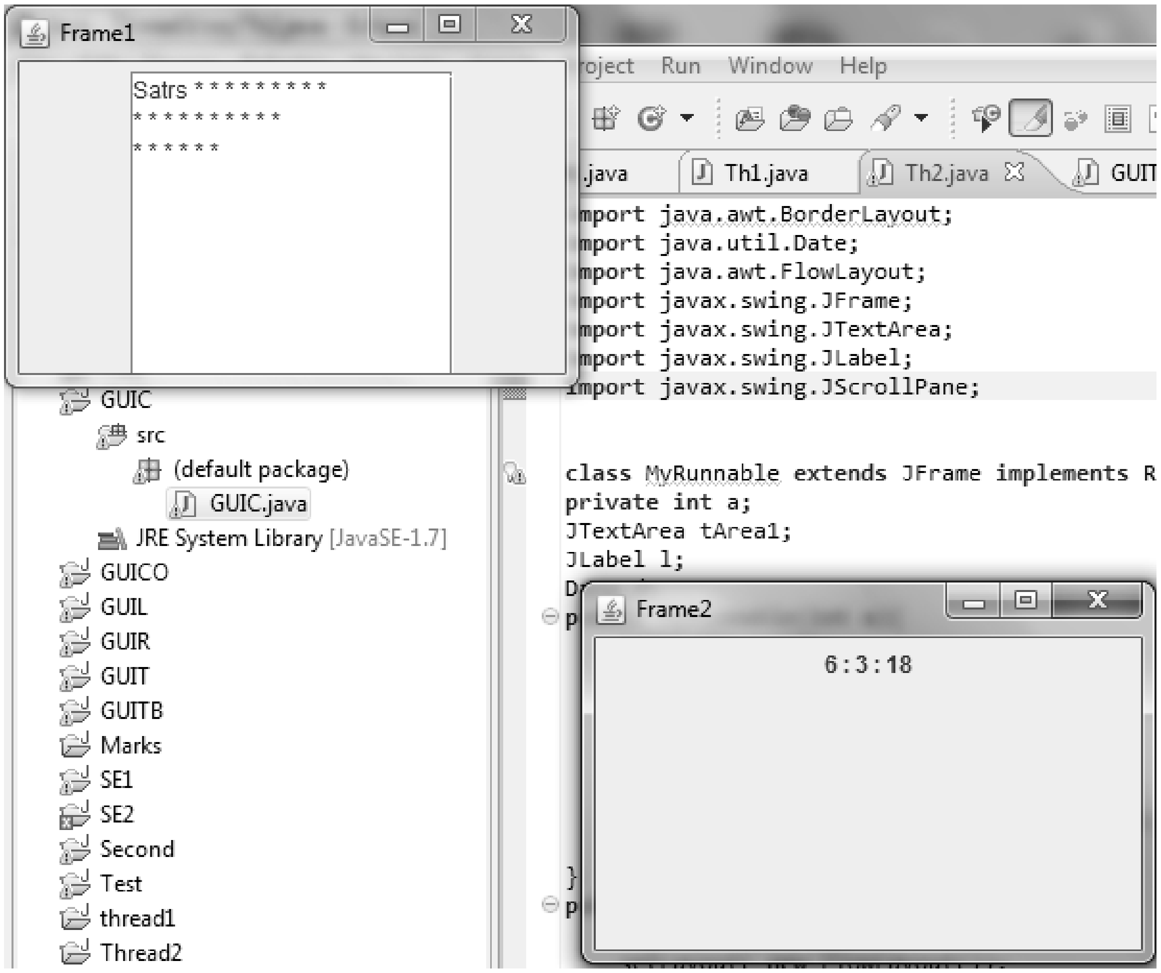
SN	Methods with Description
1	**public static void yield()** Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled
2	**public static void sleep(long millisec)** Causes the currently running thread to block for at least the specified number of milliseconds
3	**public static boolean holdsLock(Object x)** Returns true if the current thread holds the lock on the given Object.
4	**public static Thread currentThread()** Returns a reference to the currently running thread, which is the thread that invokes this method.
5	**public static void dumpStack()** Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

//Thread example

```
import java.awt.BorderLayout;
import java.util.Date;
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.JLabel;
import javax.swing.JScrollPane;

class MyRunnable extends JFrame implements Runnable{
private int a;
JTextArea tArea1;
JLabel l;
Date d;
public MyRunnable(int a){
        super("Frame1");
        setLayout( new FlowLayout());
        tArea1 = new JTextArea("Satrs", 10, 15 );
        add( new JScrollPane( tArea1 ));
        setSize(300,200);
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.a = a;}
public MyRunnable(float b){
        super("Frame2");
        setLayout( new FlowLayout());
        d=new Date();
        l=new JLabel(d.getHours()+" : "+d.getMinutes()+" : "+d.getSeconds());
        add(l);
        setSize(300,200);
        setVisible(true);
        setLocation(300,300);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.a = (int)b;}
public void run(){
        String tt="",t="";
        for (int i = 1; i <= a; ++i){
        if(Thread.currentThread().getName().equals("T1"))
        {t=tArea1.getText();
         tt="*";
         if(i%10==0)
         tArea1.setText( t+"\n"+tt );
         else
         tArea1.setText( t+" "+tt );}
        else
        {d=new Date();
        l.setText(d.getHours()+" : "+d.getMinutes()+" : "+d.getSeconds());}
try{
Thread.sleep(500);}
catch (InterruptedException e){}
}}
```

```
public static void main(String args[]){
MyRunnable thr1, thr2;
thr1 = new MyRunnable(500);
thr2 = new MyRunnable(500.0f);
Thread t1 = new Thread(thr1,"T1");
Thread t2 = new Thread(thr2,"T2");
t1.start();
t2.start();
}}
```



```
Satrs *********
**********
******
```

```
mport java.awt.BorderLayout;
mport java.util.Date;
mport java.awt.FlowLayout;
mport javax.swing.JFrame;
mport javax.swing.JTextArea;
mport javax.swing.JLabel;
import javax.swing.JScrollPane;

class MyRunnable extends JFrame implements R
private int a;
JTextArea tArea1;
JLabel l;
```

```
6:3:18
```