Threads:

* Threads are the smallest unit of dispatchable code. A single program can perform two or more tasks simultaneously.

* Threads are light weight processes. Threads are separate paths of execution which are functionally independent of each other.

* A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

* Multi Threading is a single process where there are different paths of execution. For a threaded application, all threads share the same resource that has been allocated for the application. Threads have their own life cycle, properties, different types of threads and various ways of creating.

Lifecycle of Thread:

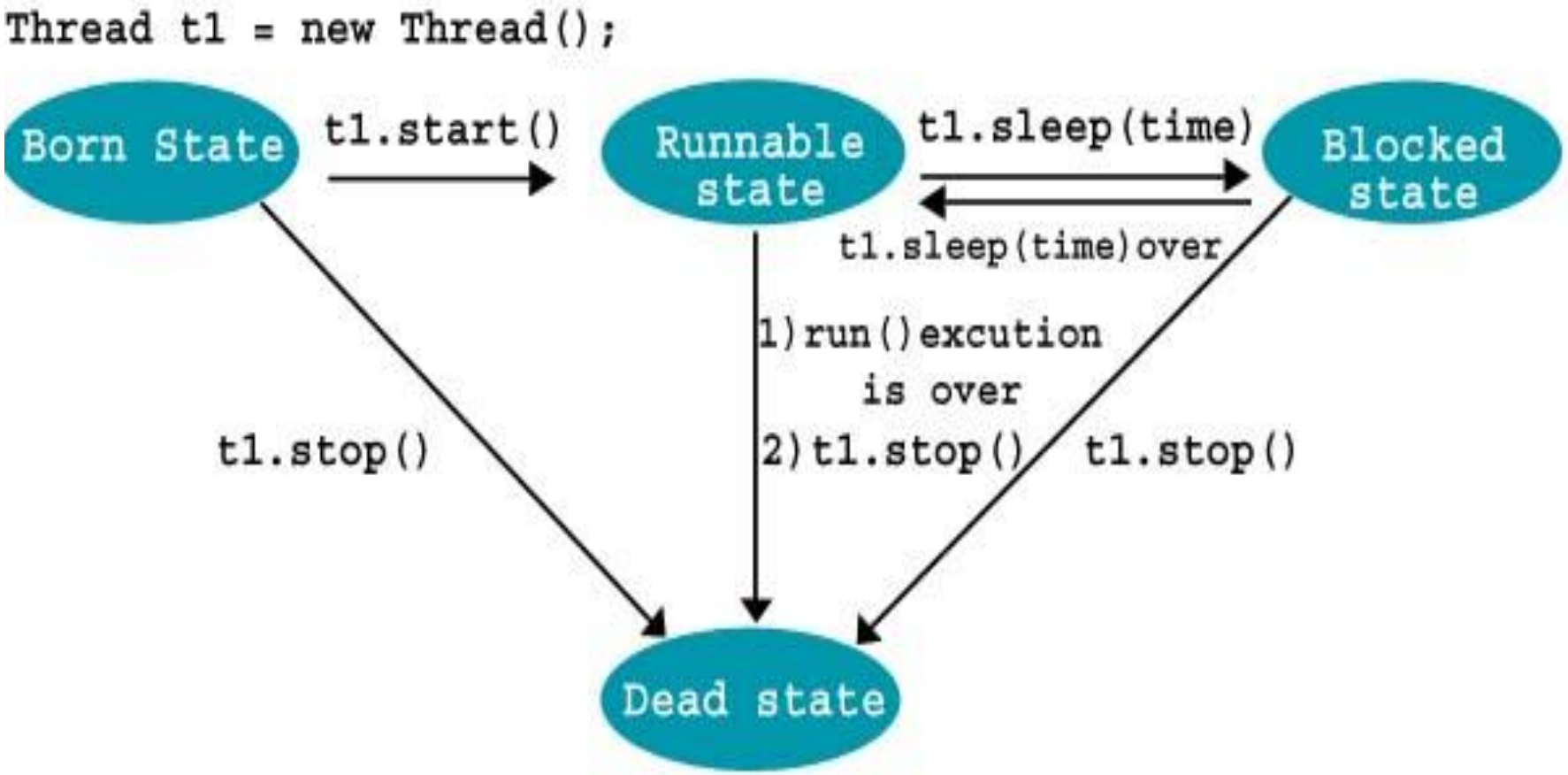

Figure: Thread life cycle

Thread class and the Runnable Interface:

* To create a new thread, the program should either extend **Thread** class or implement the **Runnable** interface.

* **java.lang** package

Commonly used thread constructors are:

Thread()
Thread(String name)
Thread(Runnable r)
Thread(Runnable r, String name)

Thread class methods:

final String getName(): Obtain a thread's name
final void setName(String threadName): To set a name for a thread
final int getPriority(): Obtain a thread's priority
final void setPriority(int level): To set a priority for a thread

final boolean isAlive(): Determine if a thread is still running

public void run(): Entry point for the thread

static void sleep(long milliseconds) throws InterruptedException:
Suspend a thread for a period of time

void start(): Start a thread by calling its run method

The Main Thread:

* When a Java program starts up, the main thread of the program gets executed.

* From the main thread, the child threads will be executed or created.

* It is also the last thread to finish execution because it performs various shutdown actions.

* The main thread can be controlled through a Thread object using the currentThread(), which is a public static member of Thread.

```
static Thread currentThread()      // returns a reference to the
                                   // thread in which it is called
```

```
Class threaddemo
{
        public static void main(String[] args)
        {
                Thread t=Thread.currentThread();
                System.out.println("Current Thread:"+t);

                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");  }
```

Extending Thread class:

* Create a new class that extends Thread, and then create an instance of that class.

* The extending class must override the run() method, which is the entry point for the new thread.

* It must also call start() to begin execution of the new thread.

Example 1

```
public class MyThread extends Thread {
public void run(){
    System.out.println("Thread running");
        }
    public static void main(String[] args) {
    MyThread t= new MyThread();
    t.start();
    }
}
```

Example 2

```
Class newThread extends Thread
{
        newThread()
        {
                super("Demo Thread");
                System.out.println("Child Thread:"+this);
                start();
        }

        public void run()
        {       try
                {       for(int i=5;i>0;i--)
                        {
                                System.out.println("Child Thread:"+i);
                                Thread.sleep(500);
                }       }catch(InterruptedException e);
                          { System.out.println("Child interrupted"); }
                System.out.println("Exiting Child Thread");
        }
```

```
Class extendThread
{
        public static void main(String[] args)
        {
                new newThread();
                try
                {       for(int i=5;i>0;i--)
                        {
                                System.out.println("Main Thread:"+i);
                                Thread.sleep(1000);
                }       }catch(InterruptedException e);
                          { System.out.println("Main Thread interrupted"); }
                System.out.println("Main Thread Exiting");
        }
}
```

Implementing the Runnable Interface:

* Declare the class as implementing the Runnable interface

* Implement the run() method

* Create a thread by defining an object that is instantiated from this runnable class as the target of the thread.

* Call the thread's start() method to run the thread.

Example

```
public class MyThread implements Runnable {
public void run(){
   System.out.println("Thread running");
}
   public static void main(String[] args) {
   MyThread t= new MyThread();
   Thread th= new Thread(t);
   th.start();

   }
}
```

Priority Threads:

* Thread Scheduler can use the thread priorities in the form of integer value to each of its thread to determine the execution schedule of threads.

* Thread gets the ready-to-run state according to their priorities.

* The thread scheduler provides the CPU time to thread of highest priority during ready-to-run state.

* Priorities are integer values from 1 (lowest priority given by the constant Thread.MIN_PRIORITY) to 10 (highest priority given by the constant Thread.MAX_PRIORITY). The default priority is 5 (Thread.NORM_PRIORITY).

```
public class MyThread extends Thread{
public void run(){
   System.out.println("Thread name= "+Thread.currentThread().getName());   }
   public static void main(String[] args) {
   MyThread t= new MyThread();
   MyThread t1= new MyThread();
   MyThread t2= new MyThread();
   System.out.println("Thread t priority="+t.getPriority());
   System.out.println("Thread t1 priority="+t1.getPriority());
   System.out.println("Thread t2 priority="+t2.getPriority());
   t.setName("t");
   t1.setName("t1");
   t2.setName("t2");
   t.setPriority(3);
   t1.setPriority(5);
   t2.setPriority(2);
   System.out.println("Thread t priority="+t.getPriority());
   System.out.println("Thread t1 priority="+t1.getPriority());
   System.out.println("Thread t2 priority="+t2.getPriority());
   t.start();
   t1.start();
   t2.start();  }  }
```

Output

```
Thread t priority=5
Thread t1 priority=5
Thread t2 priority=5

Thread t priority=3
Thread t1 priority=5
Thread t2 priority=2

Thread name= t1
Thread name= t
Thread name= t2
```

Synchronization:

* Resource used only by one thread at a time.

* The objects have implicit monitor associated with them.

* A monitor is an object that is used as a mutually exclusive lock.. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor.

* All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

* To enter an object's monitor, just call a method that has been modified with the synchronized keyword.

Using Synchronized Methods:

* Method can be synchronized by using the synchronized keyword

* When that method is called, the calling thread enters the objects monitor, which then locks the object.

* While locked, no other thread can enter the method on that object.

* When the thread returns from the method, the monitor unlocks the object, allowing it to be used by the next thread.

Using Synchronized block:

```
synchronized(objref)
{
        // statements to be synchronized
}

// Here, objref is a reference to the object for which synchronization
 // is needed.
```

Example program **without** synchronization

```
class First {
  public void display(String msg)  {
  System.out.print ("["+msg);
  System.out.println ("]");
 }  }
class Second extends Thread {
 String msg;
 First fobj;
 Second (First fp,String str) {
  fobj = fp;
  msg = str;
  start();
 }
 public void run()  {
  fobj.display(msg);
 }  }
public class Syncro {
    public static void main(String[] args) {
        First fnew = new First();
  Second ss = new Second(fnew, "welcome");
  Second ss1= new Second (fnew,"new");
  Second ss2 = new Second(fnew, "programmer");  } }
```

In the above program, object **fnew** of class First is shared by all the three running threads(ss, ss1 and ss2) to call the shared method(*void* **display**). Hence the result is unsynchronized and such situation is called **Race condition**.

Output- 1st time execution
[welcome[programmer]
[new]
]

Output- 2nd time execution
[new[programmer]
]
[welcome]

Example program with synchronization keyword

```
class First {
 synchronized public void display(String msg) {
  System.out.print ("["+msg);
  System.out.println ("]");
 } }
class Second extends Thread {
 String msg;
 First fobj;
 Second (First fp,String str) {
  fobj = fp;
  msg = str;
  start(); }
 public void run() {
  fobj.display(msg);
 } }
public class Syncro {
    public static void main(String[] args) {
        First fnew = new First();
  Second ss = new Second(fnew, "welcome");
  Second ss1= new Second (fnew,"new");
  Second ss2 = new Second(fnew, "programmer"); } }
```

To synchronize above program, we must *synchronize* access to the shared **display()** method, making it available to only one thread at a time. This is done by using keyword **synchronized** with display() method.

Output- 1st time execution
[welcome]
[programmer]
[new]

Output-2nd time execution
[welcome]
[programmer]
[new]

Example program using synchronization block

```
class First {
 synchronized public void display(String msg) {
  System.out.print ("["+msg);
  System.out.println ("]"); } }
class Second extends Thread {
 String msg;
 First fobj;
 Second (First fp,String str) {
  fobj = fp;
  msg = str;
  start(); }
 public void run() {
     synchronized (fobj){
   fobj.display(msg);        } } }
public class Syncro {  public static void main(String[] args) {
        First fnew = new First();
  Second ss = new Second(fnew, "welcome");
  Second ss1= new Second (fnew,"new");
  Second ss2 = new Second(fnew, "programmer"); } }
```

Because of synchronized block this program gives the expected output

Output
[programmer]
[welcome]
[new]

- When we use synchronized keyword with a method, it acquires a lock in the object for the whole method. It means that no other thread can use any synchronized method until the current thread, which has invoked it's synchronized method, has finished its execution.
- synchronized block acquires a lock in the object only between parentheses after the synchronized keyword. This means that no other thread can acquire a lock on the locked object until the synchronized block exits. But other threads can access the rest of the code of the method.

InterThread Communication:

* Inter-thread communication is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

* It is implemented by following methods of Object class:

1) wait():

* Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

* The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

public final void wait()throws InterruptedException:
waits until object is notified.
public final void wait(long timeout)throws InterruptedException:
waits for the specified amount of time

2) notify(): resumes a single thread that is waiting on this object's monitor.

public final void notify()

3) notifyAll(): Wakes up all threads that are waiting on this object's monitor.

public final void notifyAll()

Difference between wait() and Sleep():

1) wait() releases the lock
   sleep() doesn't release the lock

2) wait() – Object Class; sleep() – Thread class

3) wait() – non-static method sleep() – static method

4) wait() – should be notified by notify() or notifyAll()
   sleep() – after the specified amount of time

Using isAlive() and join():

* isAlive() and join() are two different methods to check whether a thread has finished its execution.

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

```
final boolean isAlive()
```

* The join() method waits until the thread on which it is called terminates.

```
final void join() throws InterruptedException
final void join(long milliseconds) throws InterruptedException
```