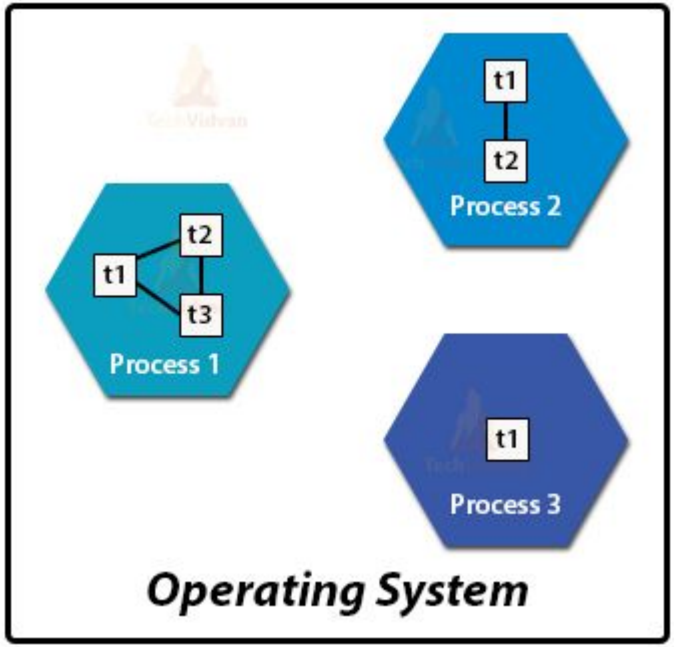
Multithreading

Multithreading

Multithreading
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 part of execution.
Multithreading is a specialized form of **thread-based multitasking** where a single program can perform two or more tasks simultaneously.
The **process-based multitasking** is the feature that allows your computer to run two or more programs concurrently.
Threads are lightweight.
Multithreading enables us to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

Multithreading



Multithreading

Achieve multithreading in java

In java language multithreading can be achieve in two different ways.

1. Using Thread class
2. Using Runnable interface

Multithreading

Using Thread Class

1. Create any user defined class and make that one as a derived class of Thread class.

   class MyThread extends Thread{ }

2. Override run() method of Thread class (It contains the logic of perform any operation)
3. Create an object for user-defined thread class.

MyThread obj=new MyThread();

4. Call start() method of thread class to execute the thread

t.start();

Multithreading

Using Runnable Interface

1. Define the class that implements the Runnable interface and implement the run () method of the Runnable interface in the class.
2. Create an instance of the defined class.
3. Create an instance of the Thread class using the Thread (Runnable target) constructor.
4. Start the thread by invoking the start () method on your Thread object.

Multithreading

Thread class : Some important methods

getName( ) : Obtain a thread’s name

setName ( ): Set 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

yield( ): Static method which current thread uses to tell the thread scheduler that it is ready to pause its execution. However thread scheduler is free to ignore it. This can move the current thread to Ready/Runnable state.

Multithreading

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread. The main thread is important because of two reasons:

- It is the thread from which other “child” threads will be created (spawned).
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a Thread object. To do so we must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread. Its general form is:

static Thread currentThread( )

Multithreading

static Thread currentThread( )

This method returns a reference to the thread in which it is called.
Once we have a reference to the main thread, we can control it just like any other thread.

Creating a Thread

By implementing Runnable interface
By extending the Thread class.

Multithreading

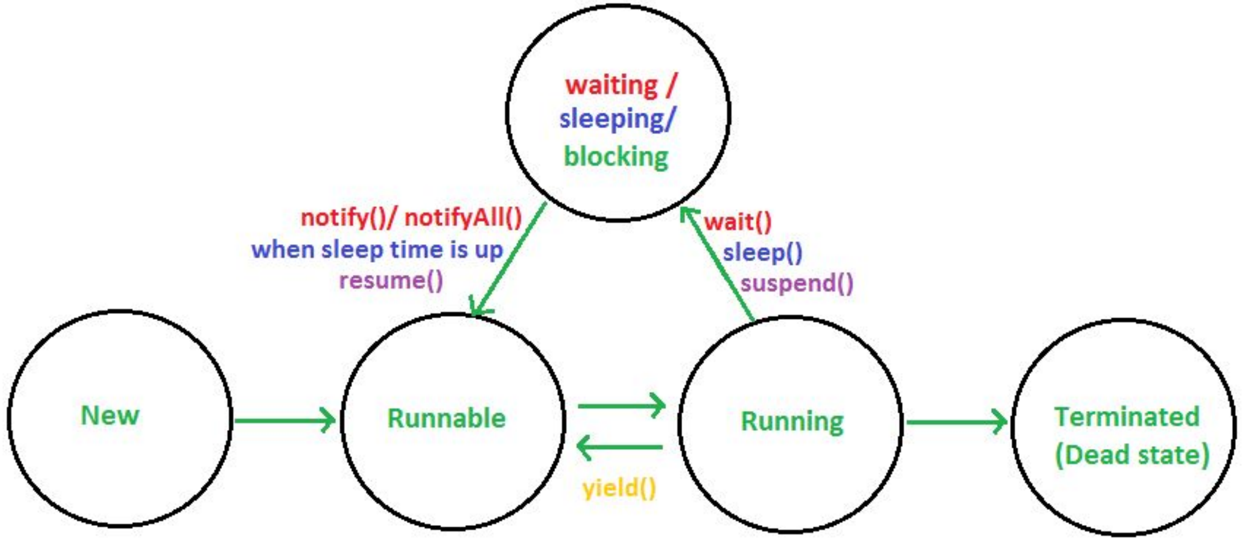


Fig. THREAD STATES

Multithreading

Thread Priorities

Thread class has a method

final void setPriority(int level);

The value of level should be in range MIN_PRIORITY (1) and MAX_PRIORITY (10). Default priority of any thread is NORM_PRIORITY (5). These three are static final variables within Thread class.

Ex. threadObj.setPriority(NORM_PRIORITY + 2);

Multithreading

Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called **synchronization**.
- Key to synchronization is the concept of **the monitor** (also called a **semaphore**).
- **A monitor is an object** that is used as **mutually exclusive lock** or mutex i.e. only one thread can have a monitor at a given time

Multithreading

Synchronization

There are **two ways of synchronization**

1. Synchronized methods
2. Synchronized block/statement

Ex. Synchronized methods

```
synchronized void fun( ) { …. }
```

Ex. Sysnchronized block

```
void fun( ) {
        // other code
        synchronized (object) {
                // statements to be synchronized
        }
         // object is the reference of the object being synchronized
}
```

Multithreading

A thread **acquires monitor** (object) if it has entered the synchronized method or block for that object.

Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method **on the same instance**. However, non-synchronized methods on that instance will continue to be callable.

Multithreading

Interthread Communication

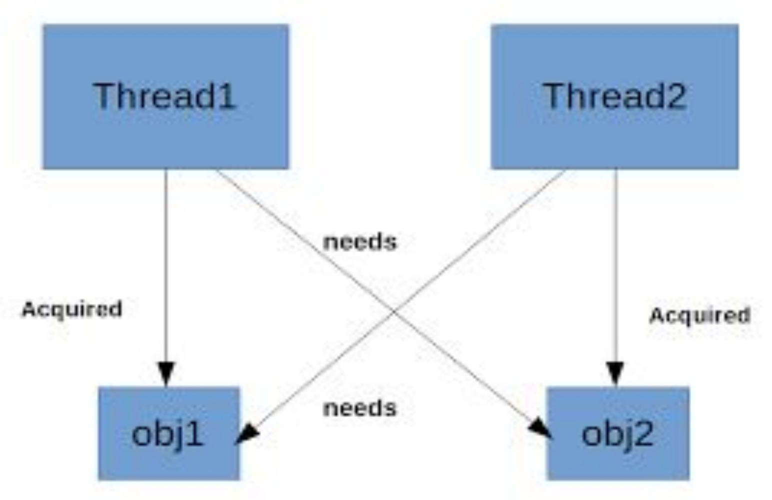
Interthread communication is done using following three methods implemented as **final** of class **Object**.

- **wait ( )** - tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).
- **notify ( )** - Wakes up a thread that called wait( ) on the same object.
- **notifyAll( )** - Wakes up all the threads that called wait( ) on the same oject. One of the threads will be granted access.

Multithreading

Deadlock

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Here's an example.



Multithreading

Suspending, Resuming and Stopping Threads

- Suspend ( ) to pause, resume ( ) to restart and stop ( ) to stop - **Deprecated** Thread class method
- These methods were deprecated as sometimes these can cause serious system failure.
- In place of using these methods, **we can use wait ( ) and notify ( ) methods** for suspending and resuming the threads.

Multithreading

ThreadGroup Demo

```
class MyThread implements Runnable
{
	Thread t ;
	public MyThread(ThreadGroup tGroup,String tName)
	{
		t = new Thread(tGroup,this,tName);
	}

	public void run()
	{
		System.out.println(Thread.currentThread().getName() + " from ThreadGroup: " +
		t.getThreadGroup().getName() + " started running ");
		try {
			while(true)
			{
```

Multithreading

```
                Thread.sleep(200);
                System.out.println(Thread.currentThread().getName() + " is running ");
            }
        }
        catch (InterruptedException e)
        {
            System.out.println(Thread.currentThread().getName() + " is interrupted ");
        }
    }
}

class ThreadGroupDemo
{
    public static void main(String args[])
    {
        ThreadGroup tgroup = new ThreadGroup("Group A");
```

Multithreading

```
        MyThread t1 = new MyThread(tgroup,"Thread1");
        MyThread t2 = new MyThread(tgroup,"Thread2");
        MyThread t3 = new MyThread(tgroup,"Thread3");
        t1.t.start();
        t2.t.start();
        t3.t.start();
        try{
            Thread.sleep(2000);
            tgroup.interrupt();
        }
        catch (InterruptedException e)
        {
            System.out.println(e);
        }
    }
}
```

Note : We can create group of ThreadGroup also. Ex. ThreadGroup main = new ThreadGroup(“Main”);

Multithreading

ThreadGroup

We can create **group (tree) of thread groups** using following constructor

ThreadGroup(ThreadGroup parent, String name)

Example:

```
 ThreadGroup main = new ThreadGroup(“Main”);
ThreadGroup subgroup1 = new ThreadGroup(main,”Subgroup1”);
ThreadGroup subgroup2 = new ThreadGroup(main,”Subgroup2”);
```

ThreadGroup class has various methods like getName( ), getParent( ), destroy ( ), interrupt ( ), activeCount( ) etc.

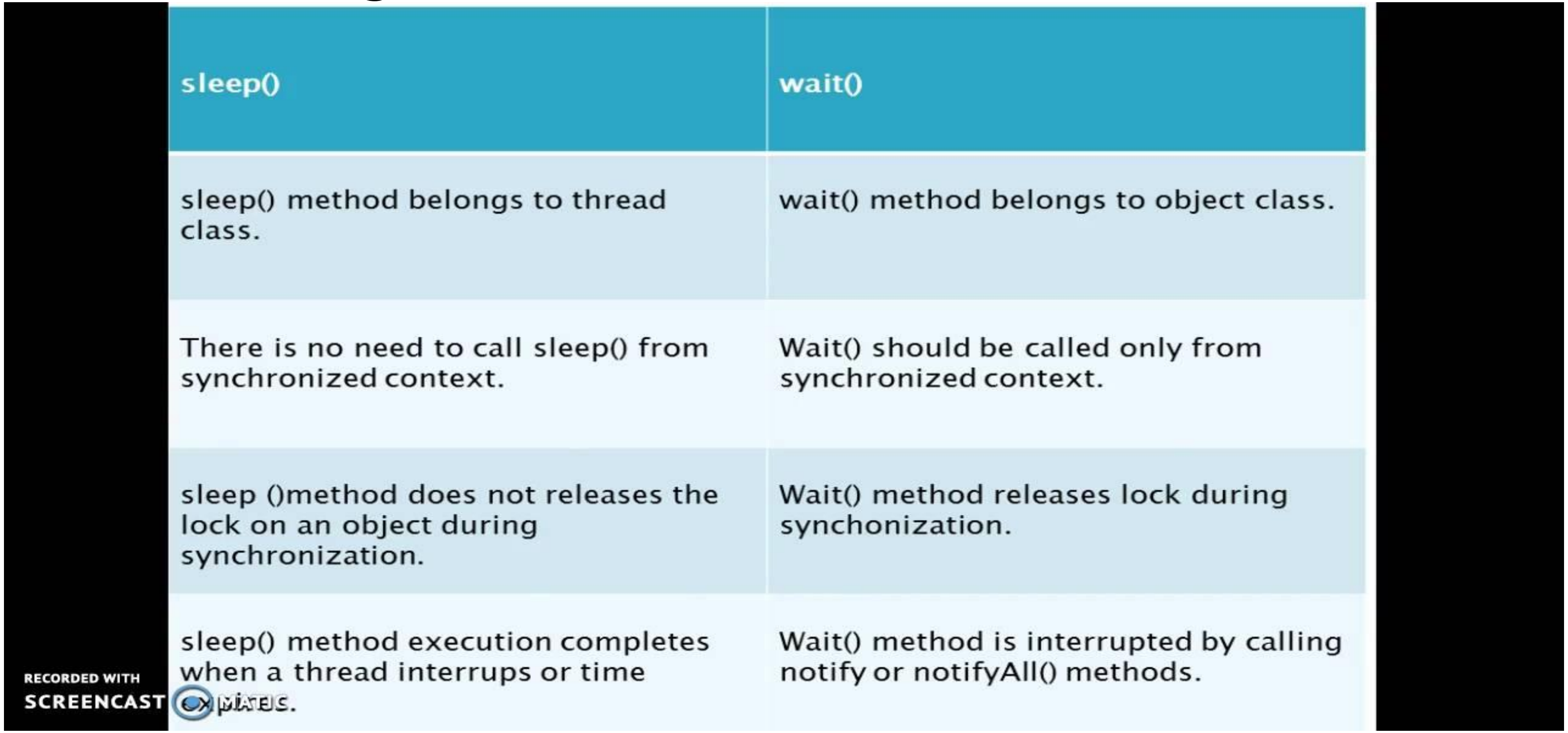
Multithreading

Daemon Thread

Daemon thread is a low priority thread which usually runs in background just like garbage collector.
JVM terminates itself when all user threads (non-daemon threads) finish their execution, JVM does not care whether Daemon thread is running or not, if JVM finds running daemon thread (upon completion of user threads), it terminates the thread and after that shutdown itself.

Creating a thread daemon thread
public final void setDaemon(boolean on)
Ex. t.setDaemon(true);
How to know whether a thread is a daemon thread or not?
boolean isDaemon() : It returns true if the thread is Daemon else it returns false.

Multithreading



sleep()	wait()
sleep() method belongs to thread class.	wait() method belongs to object class.
There is no need to call sleep() from synchronized context.	Wait() should be called only from synchronized context.
sleep ()method does not releases the lock on an object during synchronization.	Wait() method releases lock during synchonization.
sleep() method execution completes when a thread interrups or time expires.	Wait() method is interrupted by calling notify or notifyAll() methods.

Multithreading

Difference Between wait ( ) and join ( )

Difference between wait() and join() method is that former must be called from synchronized method or block but later can be called without a synchronized block in Java.

wait () is the method inside Object class whereas join () is the method inside Thread class.