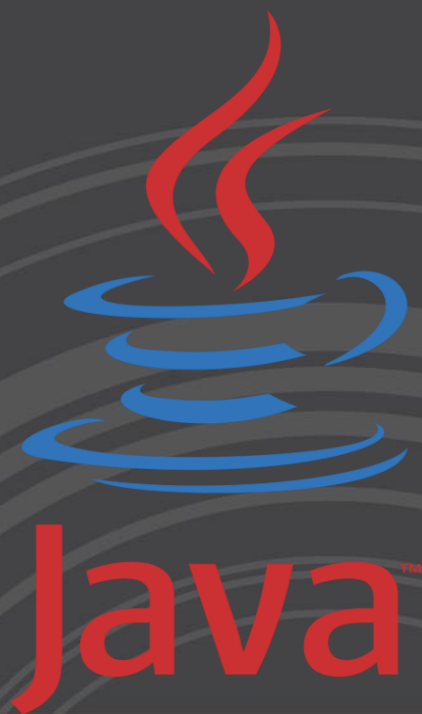


# *Object Oriented Programming*



**Java**<sup>TM</sup>



# *Unit 7*



## **MultiThreading**

Overview of Multithreading: Overview of Multithreading, Creating Threads  
Extending the thread class: Implement Thread, Implement the runnable interface.

Synchronization: Synchronization, Implement synchronize, Synchronization Issues, Race Condition, Interthread communication

# Threads

- A thread is a **small part of program which can run independently with other threads and main() method.**
- Threads are **independent**. If there occurs exception in one thread, it doesn't affect other threads.
- It uses a **shared memory area**.
- A thread is a:
  - ✓ Facility to **allow multiple activities** within a single process
  - ✓ Referred as **lightweight process**
  - ✓ A thread is a **series of executed statements**
  - ✓ Each thread **has its own** program counter, stack and local variables
  - ✓ A thread is a **nested sequence** of method calls
  - ✓ Its **shares memory, files** and per-process state
- **Multithreading in java is a process of executing multiple threads simultaneously.**



OS

# Threads

## ➤ Need of a thread or why we use Threads?

- To perform **asynchronous or background processing**
- Increases the **responsiveness** of GUI applications
- Take advantage of **multiprocessor systems**
- Simplify program logic when there are **multiple independent entities**

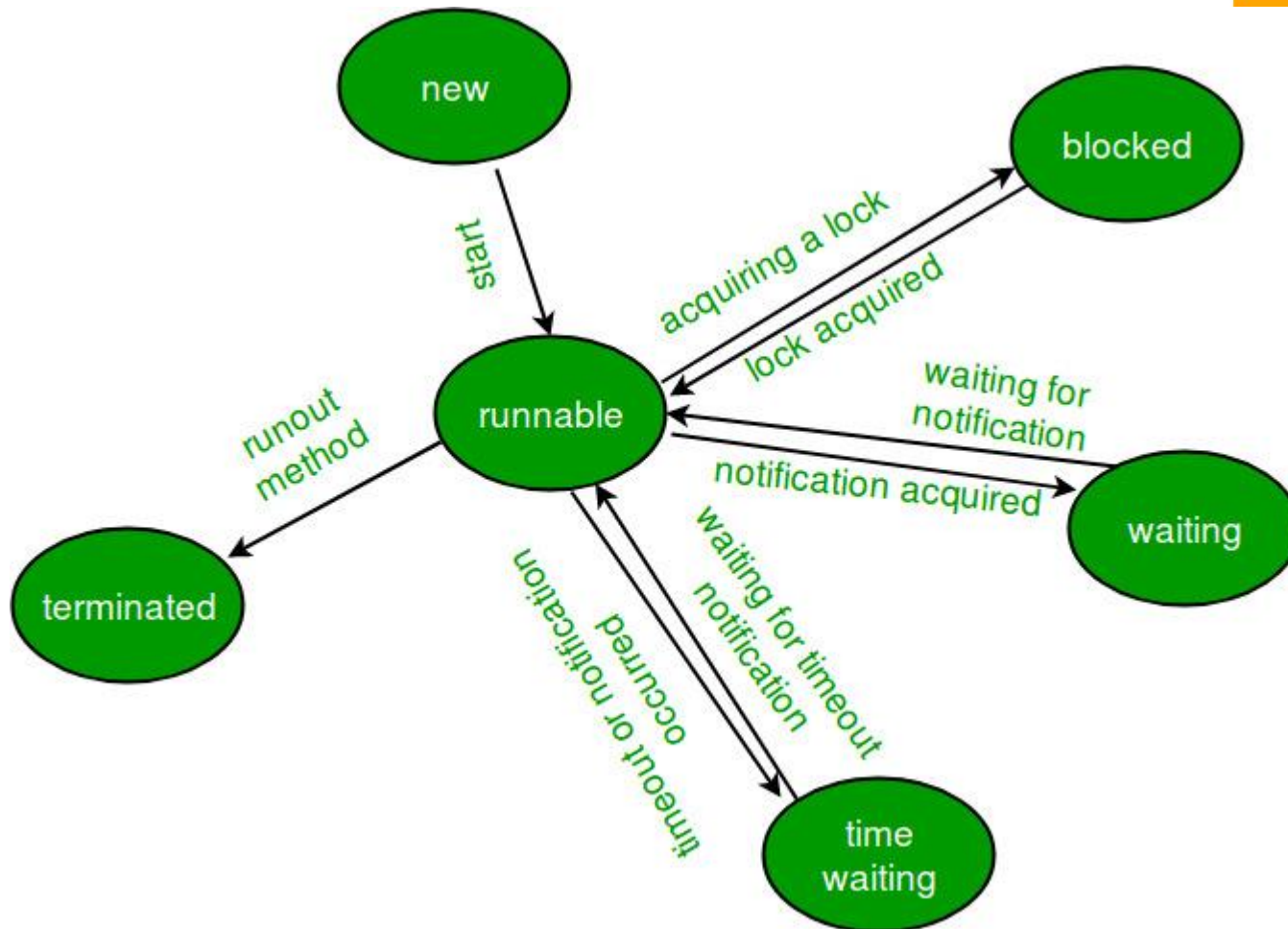
## ➤ Advantages of Multithreading:

- It doesn't block the user because threads are independent and you can perform multiple operations at the same time.
- You can perform many operations together, so it saves time.
- Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.
- Helps achieve multitasking in Java.

# *Thread Control Methods & Thread Life Cycle*

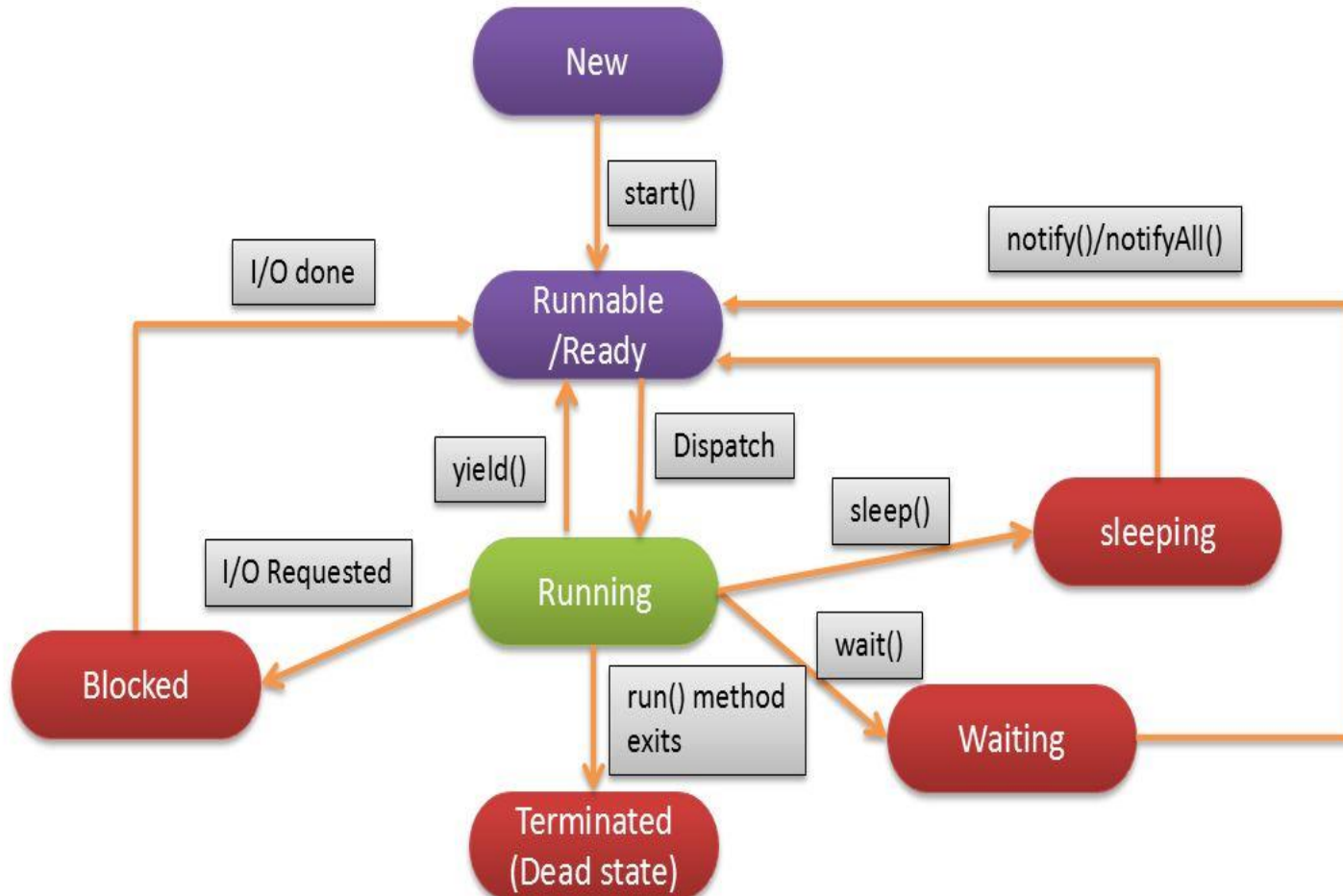
- The **java.lang.Thread** defines the following methods:
  - ✓ **public void start()**
  - ✓ **public void stop()**
  - ✓ **public void suspend()**
  - ✓ **public void sleep(int time) throws InterruptedException**
  - ✓ **public void resume()**
- The life cycle of the thread in java is controlled by JVM. During the complete life span of a thread, it goes from five phases:
  - ❖ New state
  - ❖ Runnable state
  - ❖ Running state
  - ❖ Blocked state(Non-Runnable)
  - ❖ Terminated(Dead) state

# Thread Control Methods & Thread Life Cycle



# Thread Life Cycle

Life cycle of a Thread (Thread States)



# Thread Life Cycle (Contd.)

- **New Thread:** When a new thread is created, it is in the new state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute. It is also referred to as a **born thread**.
- **Running State:** A thread is in running when its execution is currently going on or when thread is handling the controls. Only one thread can be running at a time.
- **Runnable State:** A thread that is **ready to run** is moved to runnable state. In this state, a thread might **actually be running** or it might be ready to run at any instant of time. It is the responsibility of the **thread scheduler** to give the thread, time to run. A **multi-threaded program** allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.



# Thread Life Cycle (Contd.)

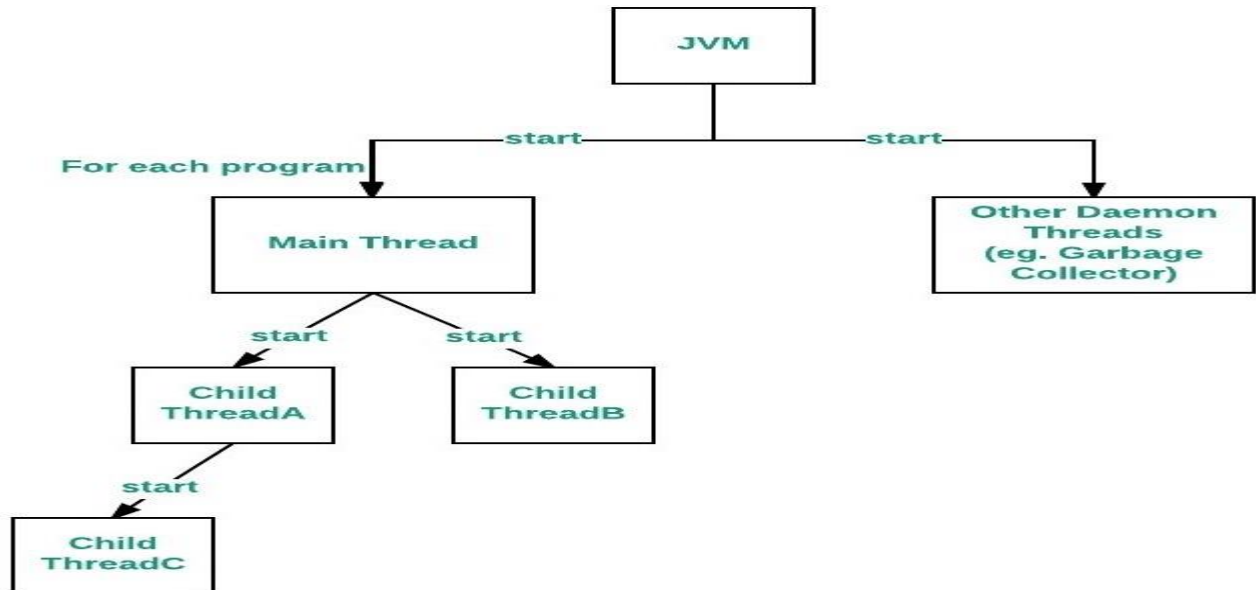
- **Blocked/Waiting state:** When a thread is temporarily inactive, then it's in one of the following states:
  - ✓ Blocked
  - ✓ Waiting
- For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It's the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread. A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states do not consume any CPU cycle.
- **A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread.** When the protected section is unlocked, the scheduler picks one of the thread which is blocked for that section and moves it to the runnable state. Whereas, a thread is in the waiting state when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.
- If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

# Thread Life Cycle (Contd.)

- **Timed Waiting:** A thread lies in timed waiting state when it calls a method with a time out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to timed waiting state.
- **Terminated State:** A thread terminates because of either of the following reasons:
  - Because it **exits normally**. This happens when the code of thread has entirely executed by the program.
  - Because there occurred some **unusual erroneous event**, like segmentation fault or an unhandled exception.
- A thread that lies in terminated state does no longer consumes any cycles of CPU.

# The Main Thread

- When a Java program starts up, one thread begins running immediately. This is usually called the **main** thread of our program, because it is the one that is executed when our program begins.
- **Properties :**
  - ✓ It is the thread from which other “child” threads will be spawned.
  - ✓ Often, it must be the last thread to finish execution because it performs various shutdown actions.



# Creating a Thread

## ➤ There are **two ways to create a thread**:

1. By extending Thread class
2. By implementing Runnable interface.

## ➤ **Constructors**

- Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.
- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

# Creating a Thread

- 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.

- **start() method** of Thread class is used to start a newly created thread. It performs following tasks:

- ✓ A new thread starts(with new callstack).
- ✓ The thread moves from New state to the Runnable state.
- ✓ When the thread gets a chance to execute, its target run() method will run.

## Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long milliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long milliseconds):** waits for a thread to die for the specified milliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(deprecated).
16. **public void resume():** is used to resume the suspended thread(deprecated).
17. **public void stop():** is used to stop the thread(deprecated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

# *Extending the Thread class*

```
class TExample extends Thread
{
    public void run()
    {
        System.out.println("Thread is running!!");
    }
    public static void main(String args[])
    {
        TExample t1=new TExample();
        t1.start();
    }
}
```

# *Implementing the Runnable interface*

```
class TExample implements Runnable
{
    public void run()
    {
        System.out.println("Thread is running!!");
    }
    public static void main(String args[])
    {
        TExample t1=new TExample();
        Thread td =new Thread(t1);
        td.start();
    }
}
```

NOTE: If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create Thread class object. You need to pass the object of the class that implements Runnable so that the class run() method may execute.



# Thread Example

```
class MultithreadingDemo extends Thread
```

```
{  
    public void run()
```

```
{
```

```
        System.out.println ("Thread is running: "+Thread.currentThread().getId());
```

```
    }
```

```
}
```

```
public class Multithread
```

```
{
```

```
    public static void main(String[] args)
```

```
{
```

```
        int n = 8;
```

```
        for (int i=0; i<n; i++)
```

```
        {
```

```
            MultithreadingDemo object = new MultithreadingDemo();
```

```
            object.start();
```

```
        }
```

```
    }
```

```
}
```

```
Thread is running: 18  
Thread is running: 16  
Thread is running: 19  
Thread is running: 20  
Thread is running: 15  
Thread is running: 17  
Thread is running: 14  
Thread is running: 21
```

# Thread Example

```
class MultithreadingDemo extends Thread
{
    public void run()
    {
        for(int x=1;x<=3;x++)
        {
            System.out.println("Thread class: "+x);
        }
    }
}

public class ThreadPrograms
{
    public static void main(String[] args)
    {
        MultithreadingDemo object = new MultithreadingDemo();
        object.start();
        for(int x=11;x<=13;x++)
        {
            System.out.println("Parent class: "+x);
        }
        System.out.println("Done");
    }
}
```

```
Parent class: 11
Thread class: 1
Parent class: 12
Thread class: 2
Parent class: 13
Thread class: 3
Done|
```

# *sleep() Method*

- The **sleep()** method of **Thread** class is used to sleep a thread for the specified amount of time.
- Syntax of sleep() method in java
- The Thread class provides two methods for sleeping a thread:  
**public static void sleep(long milliseconds)throws InterruptedException**  
**public static void sleep(long milliseconds, int nanos)throws InterruptedException**
- While using sleep(), always handle the exception it throws.

# Thread Example

```
package threadprograms;

class MultithreadingDemo extends Thread
{
    public void run()
    {
        for(int x=1;x<=3;x++)
        {
            System.out.println("Thread class: "+x);
            try
            {
                Thread.sleep(500);
            }
            catch(Exception e){}
        }
    }
}
```

```
public class ThreadPrograms
{
    public static void main(String[] args)
    {
        MultithreadingDemo object = new MultithreadingDemo();
        object.start();
        for(int x=11;x<=13;x++)
        {
            System.out.println("Parent class: "+x);
        }
        System.out.println("Done");
    }
}
```

Thread class: 1

Parent class: 11

Parent class: 12

Parent class: 13

Done

Thread class: 2

Thread class: 3

BUILD SUCCESSFUL (total time: 3 seconds)

# Thread sleep() Example

```
class TestSleepMethod1 extends Thread
{
    public void run()
    {
        for(int i=1;i<5;i++)
        {
            try{Thread.sleep(1000);} catch(InterruptedException e){System.out.println(e);}
            System.out.println(i);
        }
    }
    public static void main(String args[])
    {
        TestSleepMethod1 t1=new TestSleepMethod1();
        TestSleepMethod1 t2=new TestSleepMethod1();

        t1.start();
        t2.start();
    }
}
```

run:

1

1

2

2

3

3

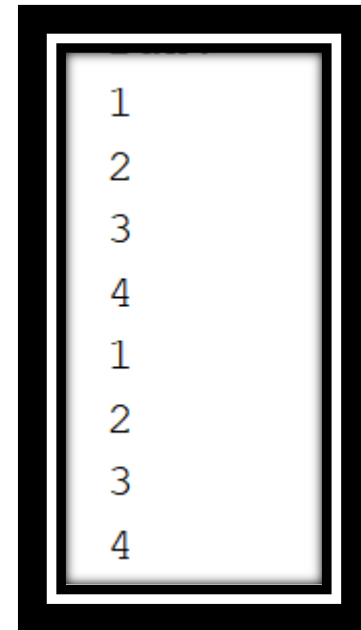
4

4

# *Calling run() directly*

```
class TestSleepMethod1 extends Thread
{
    public void run()
    {
        for(int i=1;i<5;i++)
        {
            try{Thread.sleep(1000);} catch(InterruptedException e){System.out.println(e);}
            System.out.println(i);
        }
    }
    public static void main(String args[])
    {
        TestSleepMethod1 t1=new TestSleepMethod1();
        TestSleepMethod1 t2=new TestSleepMethod1();

        t1.run();
        t2.run();
    }
}
```



# *isAlive() and join()*

- Sometimes one thread needs to know when another thread is ending. In java, **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.
- **join() method** is used more commonly than **isAlive()**. This method waits until the thread on which it is called terminates.
- **final Boolean isAlive()**
- **final void join() throws InterruptedException**
- **final void join(long milliseconds) throws InterruptedException**

# *isAlive()*

```
TestSleepMethod1.java - Notepad
File Edit Format View Help

class TestSleepMethod1 extends Thread{
public void run()
{
    System.out.println("thread 1");
    try
    {Thread.sleep(500);}
    catch(InterruptedException ie){}
    System.out.println("thread 2");
}

public static void main(String args[]){
    TestSleepMethod1 t1=new TestSleepMethod1();
    TestSleepMethod1 t2=new TestSleepMethod1();
    System.out.println(t1.isAlive());
    t1.start();
    System.out.println(t1.isAlive());
    t2.start();
    System.out.println(t2.isAlive());
}
}
```

```
C:\Windows\System32\cmd.exe

C:\Users\Virendra\Desktop\Core Java>java TestSleepMethod1
false
true
true
thread 1
thread 1
thread 2
thread 2

C:\Users\Virendra\Desktop\Core Java>javac TestSleepMethod1.java

C:\Users\Virendra\Desktop\Core Java>java TestSleepMethod1
false
true
thread 1
thread 1
true
thread 2
thread 2
```



# join()

TestSleepMethod1.java - Notepad

File Edit Format View Help

```
public void run()
{
    System.out.println("thread 1");
    try
    {Thread.sleep(500);}
    catch(InterruptedException ie){}
    System.out.println("thread 2");
}

public static void main(String args[]){
    TestSleepMethod1 t1=new TestSleepMethod1();
    TestSleepMethod1 t2=new TestSleepMethod1();
    t1.start();
    try
    {
        t1.join();
    }
    catch(InterruptedException ie){}|
    t2.start();
    }
}
```

C:\Windows\System32\cmd.exe

C:\Users\Virendra\Desktop\Core Java>javac TestSleepMethod1.java

C:\Users\Virendra\Desktop\Core Java>java TestSleepMethod1

thread 1

thread 2

thread 1

thread 2

C:\Users\Virendra\Desktop\Core Java>

# Thread Priority

- **Each thread have a priority.** Priorities are represented by a number between **1 and 10**.
- In most cases, thread scheduler schedules the threads according to their priority (known as **preemptive scheduling**). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses
- 3 constants defined in **Thread class**:
  - **public static int MIN\_PRIORITY**
  - **public static int NORM\_PRIORITY**
  - **public static int MAX\_PRIORITY**
- **Default priority of a thread is (NORM\_PRIORITY).** The value of **MIN\_PRIORITY** is **1** and the value of **MAX\_PRIORITY** is **10**.

# Thread Priority Example

```
class TestPriority extends Thread
{
    public void run()
    {
        System.out.println("running thread name is:"+Thread.currentThread().getName());
        System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
    }

    public static void main(String args[])
    {
        TestPriority m1=new TestPriority();
        TestPriority m2=new TestPriority();
        m1.setPriority(Thread.MIN_PRIORITY);
        m2.setPriority(Thread.MAX_PRIORITY);
        m1.start();
        m2.start();
    }
}
```

```
running thread name is:Thread-1
running thread name is:Thread-0
running thread priority is:10
running thread priority is:1
```

# *getName(), setName() and getId()*

```
class TestJoinMethod extends Thread{
    public void run()
    {
        System.out.println("Thread is running...");
    }
    public static void main(String args[]){
        TestJoinMethod t1=new TestJoinMethod();
        TestJoinMethod t2=new TestJoinMethod();
        System.out.println("Name of t1:"+t1.getName());
        //System.out.println(Thread.currentThread().getName());
        System.out.println("Name of t2:"+t2.getName());
        //System.out.println(Thread.currentThread().getName());
        System.out.println("id of t1:"+t1.getId());

        t1.start();
        t2.start();

        t1.setName("SYIT");
        System.out.println("After changing name of t1:"+t1.getName());
    }
}
```

Name of t1:Thread-0

Name of t2:Thread-1

id of t1:10

After changing name of t1:SYIT

Thread is running...

Thread is running...

# Synchronization

- **Synchronization** in java is the capability to **control the access of multiple threads to any shared resource.**
- The synchronized keyword guards' critical sections of code
- Either methods or arbitrary sections of code may be synchronized.
- Java Synchronization is better option where we want to **allow only one thread to access the shared resource.**
- If one thread is writing some data and another thread which is reading data at the same time, might create inconsistency in the application.
- When there is a need to access the shared resources by two or more threads, then synchronization approach is utilized.
- The synchronization is mainly used to
  - To prevent thread interference.
  - To prevent consistency problem.

# Synchronization

- ❑ **synchronized method**
- ❑ If you declare any method as **synchronized**, it is known as **synchronized method**.
- ❑ **Synchronized method** is used to lock an object for any shared resource.
- ❑ When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

# *Synchronization*

```
import java.util.*;
class Sender
{
    public void send(String msg)
    {
        System.out.println("Sending\t" + msg);
        try
        {
            Thread.sleep(1000);
        }
        catch (Exception e)
        {
            System.out.println("Thread interrupted.");
        }
        System.out.println("\n" + msg + "Sent");
    }
}
```

```
class ThreadedSend extends Thread
{
    private String msg;
    Sender sender;
    ThreadedSend(String m, Sender obj)
    {
        msg = m;
        sender = obj;
    }

    public void run()
    {
        synchronized(sender)
        {
            sender.send(msg);
        }
    }
}
```

# Synchronization

```
class SyncDemo
```

```
{
```

```
    public static void main(String args[])
```

```
    {
```

```
        Sender snd = new Sender();
```

```
        ThreadedSend S1 = new ThreadedSend(" Hi ", snd);
```

```
        ThreadedSend S2 = new ThreadedSend(" Bye ", snd);
```

```
        S1.start();
```

```
        S2.start();
```

```
    }
```

```
}
```

Select C:\Windows\System32\cmd.exe

C:\Users\Virendra\Desktop\Core Java\mypack>java SyncDemo

Sending Hi

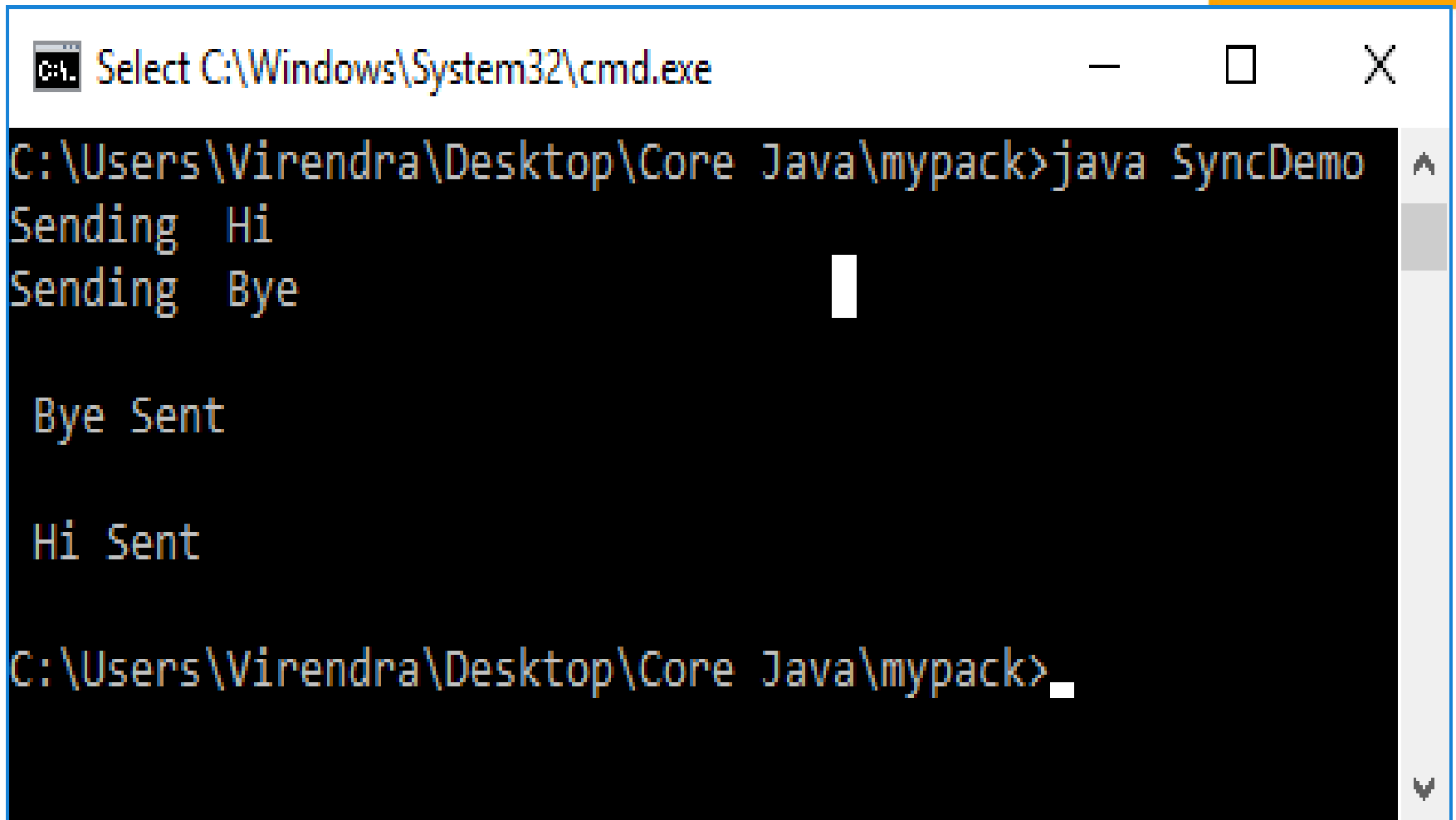
Hi Sent

Sending Bye

Bye Sent



# *Without Synchronization*



The screenshot shows a Windows command prompt window titled "Select C:\Windows\System32\cmd.exe". The prompt is at the directory "C:\Users\Virendra\Desktop\Core Java\mypack>". The user has entered the command "java SyncDemo". The output of the program is as follows:

```
C:\Users\Virendra\Desktop\Core Java\mypack>java SyncDemo
Sending  Hi
Sending  Bye

Bye Sent

Hi Sent

C:\Users\Virendra\Desktop\Core Java\mypack>_
```

The output demonstrates a race condition where the messages "Bye Sent" and "Hi Sent" are interleaved, indicating that the program did not execute in a synchronized manner.

# *Alternative ways of Synchronization(synchronized method)*

```
class Sender
{
    public synchronized void send(String msg)
    {
        System.out.println("Sending\t" + msg );
        try
        {
            Thread.sleep(1000);
        }
        catch (Exception e)
        {
            System.out.println("Thread interrupted.");
        }
        System.out.println("\n" + msg + "Sent");
    }
}
```

# *Alternative ways of Synchronization(synchronized block)*

```
class Sender
{
    public void send(String msg)
    {
        synchronized(this)
        {
            System.out.println("Sending\t" + msg );
            try
            {
                Thread.sleep(1000);
            }
            catch (Exception e)
            {
                System.out.println("Thread interrupted.");
            }
            System.out.println("\n" + msg + "Sent");
        }
    }
}
```

# *Synchronization Issues*

Use synchronization sparingly

- o Can slow performance by reducing concurrency.
- o Can sometimes lead to fatal conditions such as deadlock.

# *Race Condition*

## **Race Condition**

- A race condition is a **situation in which two or more threads or processes are reading or writing some shared data**, and the result depends on the timing of how the threads are scheduled.
- Running more than one thread inside the same application does not by itself cause problems.
- The problems arise when multiple threads access the same resources. It is safe to let multiple threads read the same resources, if the resources do not change.

## **Preventing Race condition**

- Race conditions can be avoided by **proper thread synchronization in critical sections**.
- Thread synchronization can be achieved using **locks or synchronized blocks of code**.

# *Inter-thread Communication*

- **Inter-thread communication** happens when two or more threads share information.
- Following keywords are used for interthread communication.
- **wait(), notify() and notifyAll()** methods of object class are used for this purpose.
- **wait()** - When you call wait method on the object then it tell threads to give up the lock and go to sleep state unless and until some other thread enters in same monitor and calls notify or notifyAll methods on it.
- **notify()** - When you call notify method on the object, it wakes one of thread waiting for that object. So if multiple threads are waiting for an object, it will wake of one of them. Which one it will wake up, depends on OS implementation.
- **notifyAll()** - notifyAll will wake up all threads waiting on that object unlike notify which wakes up only one of them. Which one will wake up first depends on thread priority and OS implementation.

# *wait*

## wait() method syntax

```
synchronized( lockObject )  
{  
    while( ! condition )  
    {  
        lockObject.wait();  
    }  
  
    //take the action here;  
}
```

# *notify*

notify() method syntax

```
synchronized(lockObject)
{
    //establish_the_condition;

    lockObject.notify();

    //any additional code if needed
}
```



# *notifyAll*

notifyAll() method syntax

```
synchronized(lockObject)
{
    establish_the_condition;

    lockObject.notifyAll();
}
```