

## Thread I

- ✓ Multithreading in Java is a process of executing multiple threads simultaneously.
- ✓ A thread is a lightweight sub-process, the smallest unit of processing.
- ✓ Multiprocessing and multithreading, both are used to achieve multitasking.
- ✓ However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- ✓ Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.
- ✓ Java Multithreading is mostly used in games, animation, etc.

## Thread II

### Multitasking

✓ Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

#### ① Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.

### Thread III

- Cost of communication between the process is high. Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

#### ② Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

## Thread IV

### What is Thread in Java

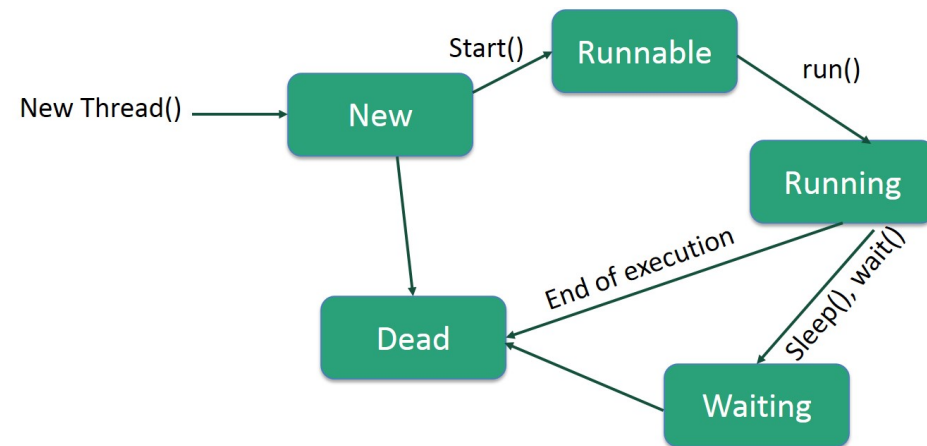
- ✓ A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.
- ✓ Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

### Advantages of Java Multithreading

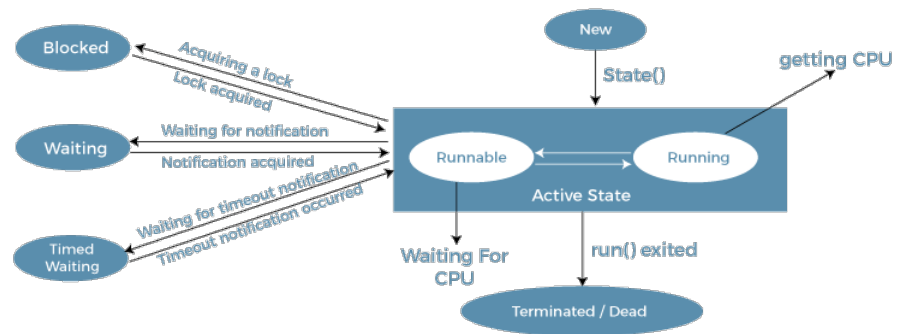
- It doesn't block the user because threads are independent and you can perform multiple operations at the same time.
- You can perform many operations together, so it saves time.
- Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

## Thread V

### Life cycle of a Thread (Thread States)



## Thread VI



Life Cycle of a Thread

## Thread VII

✓ In Java, a thread always exists in any one of the following states. These states are:

- ① New
- ② Active
- ③ Blocked/Waiting
- ④ Timed Waiting
- ⑤ Terminated

① **New:** Whenever a new thread is created, it is always in the new state.

✓ For a thread in the new state, the code has not been run yet and thus has not begun its execution.

## Thread VIII

- ② **Active:** When a thread invokes the start() method, it moves from the new state to the active state.
- ✓ The active state contains two states within it: one is runnable, and the other is running.
  - **Runnable:** A thread, that is ready to run is then moved to the runnable state.
    - ✓ In the runnable state, the thread may be running or may be ready to run at any given instant of time.
    - ✓ It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.
    - ✓ A program implementing multithreading acquires a fixed slice of time to each individual thread.

## Thread IX

- ✓ Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time.
- ✓ Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.
- **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.
- ③ **Blocked or Waiting:** Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

## Thread X

- ④ **Timed Waiting:** Sometimes, waiting for leads to starvation.

*For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section.*

✓ *In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation.*

✓ *To avoid such scenario, a timed waiting state is given to thread B.*

✓ *Thus, thread lies in the waiting state for a specific span of time, and not forever.*

✓ *A real example of timed waiting is when we invoke the sleep() method on a specific thread.*

✓ *The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.*

## Thread XI

5 **Terminated:** A thread reaches the termination state because of the following reasons:

- When a thread has finished its job, then it exists or terminates normally.
- **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

## Thread XII

### Thread Class and Runnable Interface

#### Thread

- It is a class.
- It can be used to create a thread.
- It has multiple methods such as 'start' and 'run'.
- It requires more memory space.
- Since multiple inheritance is not allowed in Java, hence, after a class extends the Thread class, it can't extend to any other class.
- Every thread creates a unique object and associates with it.

### Thread XIII

- We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

#### Runnable

- It is a functional interface.
- It can be used to create a thread.
- *It has a single abstract method 'run'.*
- It requires less memory space.
- When a class implements the 'runnable' interface, the class can extend to other classes.
- Multiple threads can share the same objects.

## Thread XIV

Sr. No.	Key	Thread	Runnable
1	Basic	Thread is a class. It is used to create a thread	Runnable is a functional interface which is used to create a thread
2	Methods	It has multiple methods including start() and run()	It has only abstract method run()
3		Each thread creates a unique object and gets associated with it	Multiple threads share the same objects.
4	Memory	More memory required	Less memory required
5	Limitation	Multiple Inheritance is not allowed in Java hence after a class extends Thread class, it can not extend any other class	If a class is implementing the runnable interface then your class can extend another class.

## Thread XV

✓ We can create Thread by either by implementing a runnable interface or by extending Thread class.

### Create a Thread by Implementing a Runnable Interface

If your class is intended to be executed as a thread then you can achieve this by implementing a Runnable interface. You will need to follow three basic steps -

**Step 1:** As a first step, you need to implement a run() method provided by a Runnable interface. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of the run() method -

```
1 public void run()
```

## Thread XVI

**Step 2:** As a second step, you will instantiate a Thread object using the following constructor -

```
1 Thread(Runnable threadObj, String threadName);
```

Where, threadObj is an instance of a class that implements the Runnable interface and threadName is the name given to the new thread.

**Step 3:** Once a Thread object is created, you can start it by calling start() method, which executes a call to run( ) method. The start() method of Thread class is used to start a newly created thread. It performs the 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.

```
1 void start();
```

## Thread XVII

### Example of Runnable

```
1 class RunnableExample implements Runnable
2 {
3     public void run()
4     {
5         System.out.println("Thread is running for Runnable Implementation");
6     }
7     public static void main(String args[])
8     {
9         RunnableExample runnable=new RunnableExample();
10        Thread t1 =new Thread(runnable);
11        t1.start();
12    }
13 }
```

## Thread XVIII

### Create a Thread by Extending a Thread Class

✓ The second way to create a thread is to create a new class that extends Thread class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

**Step 1:** You will need to override run( ) method available in Thread class. This method provides an entry point for the thread and you will put your complete business logic inside this method.

```
1 public void run()
```

**Step 2:** Once Thread object is created, you can start it by calling start() method, which executes a call to run( ) method. The start() method of Thread class is used to start a newly created thread. It performs the following tasks:

## Thread XIX

- 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.

```
1 void start( );
```

## Thread XX

### Commonly used Constructors of Thread class:

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

## Thread XXI

### Commonly used methods of Thread class:

S.N.	Modifier and Type	Method	Description
1)	void	start()	It is used to start the execution of the thread.
2)	void	run()	It is used to do an action for a thread.
3)	static void	sleep()	It sleeps a thread for the specified amount of time.
4)	static Thread	currentThread()	It returns a reference to the currently executing thread object.
5)	void	join()	It waits for a thread to die.
6)	int	getPriority()	It returns the priority of the thread.
7)	void	setPriority()	It changes the priority of the thread.
8)	String	getName()	It returns the name of the thread.
9)	void	setName()	It changes the name of the thread.
10)	long	getId()	It returns the id of the thread.

### Thread XXII

11)	boolean	isAlive()	It tests if the thread is alive.
12)	static void	yield()	It causes the currently executing thread object to pause and allow other threads to execute temporarily.
13)	void	suspend()	It is used to suspend the thread.
14)	void	resume()	It is used to resume the suspended thread.
15)	void	stop()	It is used to stop the thread.
16)	void	destroy()	It is used to destroy the thread group and all of its subgroups.
17)	boolean	isDaemon()	It tests if the thread is a daemon thread.
18)	void	setDaemon()	It marks the thread as daemon or user thread.
19)	void	interrupt()	It interrupts the thread.
20)	boolean	isinterrupted()	It tests whether the thread has been interrupted.
21)	static boolean	interrupted()	It tests whether the current thread has been interrupted.
22)	static int	activeCount()	It returns the number of active threads in the current thread's thread group.

### Thread XXIII

23)	void	checkAccess()	It determines if the currently running thread has permission to modify the thread.
24)	static boolean	holdLock()	It returns true if and only if the current thread holds the monitor lock on the specified object.
25)	static void	dumpStack()	It is used to print a stack trace of the current thread to the standard error stream.
26)	StackTraceElement[]	getStackTrace()	It returns an array of stack trace elements representing the stack dump of the thread.
27)	static int	enumerate()	It is used to copy every active thread's thread group and its subgroup into the specified array.
28)	Thread.State	getState()	It is used to return the state of the thread.
29)	ThreadGroup	getThreadGroup()	It is used to return the thread group to which this thread belongs

## Thread XXIV

30)	String	toString()	It is used to return a string representation of this thread, including the thread's name, priority, and thread group.
31)	void	notify()	It is used to give the notification for only one thread which is waiting for a particular object.
32)	void	notifyAll()	It is used to give the notification to all waiting threads of a particular object.
33)	void	setContextClassLoader()	It sets the context ClassLoader for the Thread.
34)	ClassLoader	getContextClassLoader()	It returns the context ClassLoader for the thread.
35)	static Thread.UncaughtExceptionHandler	getDefaultUncaughtExceptionHandler()	It returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.
36)	static void	setDefaultUncaughtExceptionHandler()	It sets the default handler invoked when a thread abruptly terminates due to an uncaught exception.

## Thread XXV

### Thread Priorities

- Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.
- Java thread priorities are in the range between MIN\_PRIORITY (a constant of 1) and MAX\_PRIORITY (a constant of 10). By default, every thread is given priority NORM\_PRIORITY (a constant of 5).
- Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

## Thread XXVI

### Example of Thread

```
1 class ThreadExample extends Thread
2 {
3     public void run()
4     {
5         System.out.println("Thread is running");
6     }
7     public static void main(String args[])
8     {
9         ThreadExample t1=new ThreadExample();
10        t1.start();
11    }
12 }
```

## Thread XXVII

### Implementation of Thread States

✓ In Java, one can get the current state of a thread using the `Thread.getState()` method. The `java.lang.Thread.State` class of Java provides the constants ENUM to represent the state of a thread. These constants are:

Thread state	Description
<code>public static final Thread.State NEW</code>	It represents the first state of a thread that is the NEW state
<code>public static final Thread.State RUNNABLE</code>	It represents the runnable state. It means a thread is waiting in the queue to run.
<code>public static final Thread.State BLOCKED</code>	It represents the blocked state. In this state, the thread is waiting to acquire a lock.
<code>public static final Thread.State WAITING</code>	It represents the waiting state. A thread will go to this state when it invokes the <code>Object.wait()</code> method, or <code>Thread.join()</code> method with no timeout. A thread in the waiting state is waiting for another thread to complete its task.
<code>public static final Thread.State TIMED_WAITING</code>	It represents the timed waiting state. The main difference between waiting and timed waiting is the time constraint. Waiting has no time constraint, whereas timed waiting has the time constraint. A thread invoking the following method reaches the timed waiting state.
<code>public static final Thread.State TERMINATED</code>	It represents the final state of a thread that is terminated or dead. A terminated thread means it has completed its execution.

### Thread Scheduler in Java I

- ✓ A component of Java that decides which thread to run or execute and which thread to wait is called a thread scheduler in Java.
- ✓ In Java, a thread is only chosen by a thread scheduler if it is in the runnable state.
- ✓ However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones.
- ✓ There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. Priority and Time of arrival.
- ✓ **Priority:** Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.
- ✓ **Time of Arrival:** Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case,

## Thread Scheduler in Java II

arrival time of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.

✓ Thread Scheduler Algorithms:

**First Come First Serve Scheduling:** In this scheduling algorithm, the scheduler picks the threads that arrive first in the runnable queue. Observe the following table:

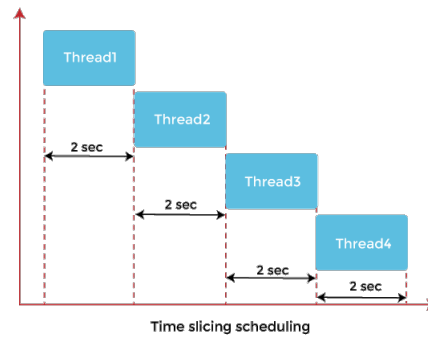
Threads	Time of Arrival
t1	0
t2	1
t3	2
t4	3



First Come First Serve Scheduling

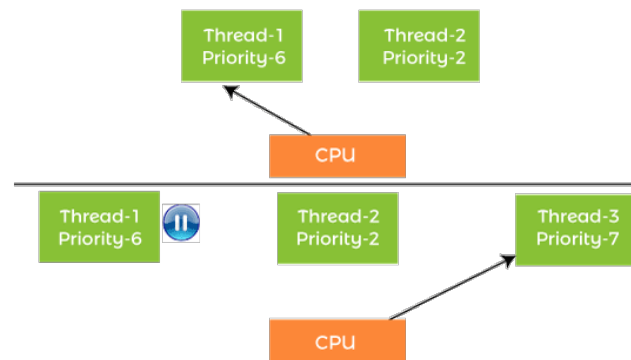
### Thread Scheduler in Java III

**Time-slicing scheduling:** Usually, the First Come First Serve algorithm is non-preemptive, which is bad as it may lead to infinite blocking (also known as starvation). To avoid that, some time-slices are provided to the threads so that after some time, the running thread has to give up the CPU. Thus, the other waiting threads also get time to run their job.



## Thread Scheduler in Java IV

**Preemptive-Priority Scheduling:** The name of the scheduling algorithm denotes that the algorithm is related to the priority of the threads.



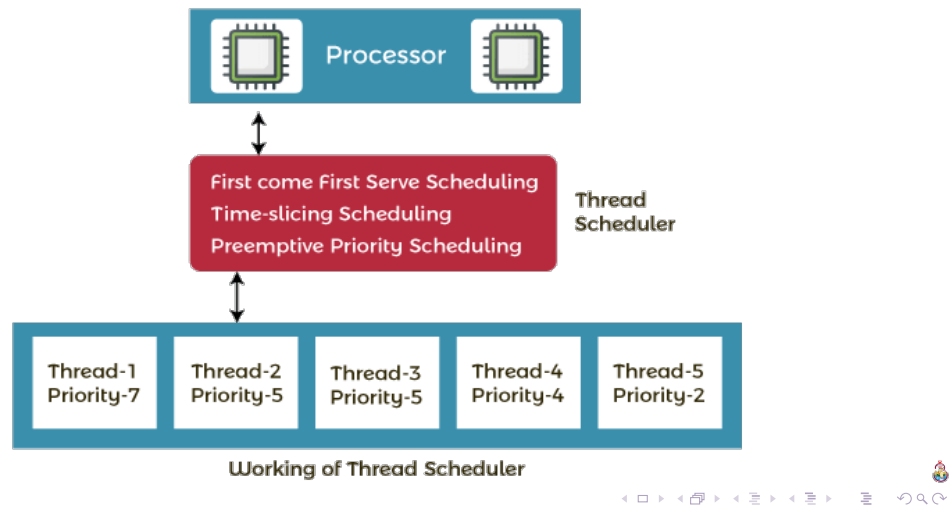
Preemptive-Priority Scheduling

### Thread Scheduler in Java V

*Suppose there are multiple threads available in the runnable state. The thread scheduler picks that thread that has the highest priority. Since the algorithm is also preemptive, therefore, time slices are also provided to the threads to avoid starvation. Thus, after some time, even if the highest priority thread has not completed its job, it has to release the CPU because of preemption.*

## Thread Scheduler in Java VI

### Working of the Java Thread Scheduler



### Thread Scheduler in Java VII

*Let's understand the working of the Java thread scheduler. Suppose, there are five threads that have different arrival times and different priorities. Now, it is the responsibility of the thread scheduler to decide which thread will get the CPU first.*

*The thread scheduler selects the thread that has the highest priority, and the thread begins the execution of the job. If a thread is already in runnable state and another thread (that has higher priority) reaches in the runnable state, then the current thread is pre-empted from the processor, and the arrived thread with higher priority gets the CPU time.*

### Thread Scheduler in Java VIII

*When two threads (Thread 2 and Thread 3) having the same priorities and arrival time, the scheduling will be decided on the basis of FCFS algorithm. Thus, the thread that arrives first gets the opportunity to execute first.*

## Thread Scheduler in Java IX

**Can we start a thread twice:** No. After starting a thread, it can never be started again. If you do so, an `IllegalThreadStateException` is thrown. In such case, thread will run once but for second time, it will throw exception.

```
1 public class TestThreadTwice1 extends Thread
2 {
3     public void run()
4     {
5         System.out.println("running...");
6     }
7     public static void main(String args[])
8     {
9         TestThreadTwice1 t1=new TestThreadTwice1();
10        t1.start();
11        t1.start();
12    }
13 }
```

## Thread Scheduler in Java X

### What if we call Java run() method directly instead start() method?

- ✓ Each thread starts in a separate call stack.
- ✓ Invoking the run() method from the main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

```
1 class TestCallRun1 extends Thread
2 {
3     public void run()
4     {
5         System.out.println("running...");
6     }
7     public static void main(String args[])
8     {
9         TestCallRun1 t1=new TestCallRun1();
10        t1.run(); //fine, but does not start a separate call stack
11    }
12 }
```

## Thread Scheduler in Java XI

### Problem if you direct call run() method

```
1 class TestCallRun2 extends Thread
2 {
3     public void run()
4     {
5         for(int i=1; i<5; i++)
6         {
7             try
8             {
9                 Thread.sleep(500);
10            }
11            catch (InterruptedException e)
12            {
13                System.out.println(e);
14            }
15            System.out.println(i);
16        }
17    }
18    public static void main(String args[])
```

## Thread Scheduler in Java XII

```
19     {  
20         TestCallRun2 t1=new TestCallRun2();  
21         TestCallRun2 t2=new TestCallRun2();  
22  
23         t1.run();  
24         t2.run();  
25     }  
26 }  
27 Output:  
28 1  
29 2  
30 3  
31 4  
32 1  
33 2  
34 3  
35 4
```

As we can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object.

## Java Thread Methods I

**public void start():** It starts the execution of the thread and then calls the run() on this Thread object.

Example:

```
1 public class StartExp1 implements Runnable
2 {
3     public void run()
4     {
5         System.out.println("Thread is running...");
6     }
7     public static void main(String args[])
8     {
9         StartExp1 thread1=new StartExp1();
10        thread1.start();
11    }
12 }
```

## Java Thread Methods II

**public void run():** This thread is used to do an action for a thread. The run() method is instantiated if the thread was constructed using a separate Runnable object.

Example:

```
1 public class RunExp1 implements Runnable
2 {
3     public void run()
4     {
5         System.out.println("Thread is running...");
6     }
7     public static void main(String args[])
8     {
9         RunExp1 r1=new RunExp1();
10        Thread thread1 =new Thread(r1);
11        thread1.start();
12    }
13 }
```

### Java Thread Methods III

**public static void sleep():** This blocks the currently running thread for the specified amount of time.

Example:

```
1 public class SleepExp1 extends Thread
2 {
3     public void run()
4     {
5         for(int i=1; i<5; i++)
6         {
7             try
8             {
9                 Thread.sleep(500);
10            }
11            catch (InterruptedException e)
12            {
13                System.out.println(e);
14            }
15        }
16    }
17 }
```

## Java Thread Methods IV

```
15         System.out.println(i);
16     }
17 }
18 public static void main(String args[])
19 {
20     SleepExp1 thread1=new SleepExp1();
21     SleepExp1 thread2=new SleepExp1();
22     thread1.start();
23     thread2.start();
24 }
25 }
```

## Java Thread Methods V

**public static Thread currentThread():** It returns a reference to the currently running

thread.

```
1 public class CurrentThreadExp extends Thread
2 {
3     public void run()
4     {
5         System.out.println(Thread.currentThread().getName());
6     }
7     public static void main(String args[])
8     {
9         CurrentThreadExp thread1=new CurrentThreadExp();
10        CurrentThreadExp thread2=new CurrentThreadExp();
11        thread1.start();
12        thread2.start();
13    }
14 }
```

## Java Thread Methods VI

**public void join():** It causes the current thread to block until the second thread terminates or the specified amount of milliseconds passes.

Example:

```
1 public class JoinExample1 extends Thread
2 {
3     public void run()
4     {
5         for(int i=1; i<=4; i++)
6         {
7             try
8             {
9                 Thread.sleep(500);
10            }
11            catch(Exception e)
12            {
13                System.out.println(e);
14            }
15        }
16    }
17 }
```

## Java Thread Methods VII

```
15         System.out.println(i);
16     }
17 }
18 public static void main(String args[])
19 {
20     JoinExample1 thread1 = new JoinExample1();
21     JoinExample1 thread2 = new JoinExample1();
22     JoinExample1 thread3 = new JoinExample1();
23     thread1.start();
24     try
25     {
26         thread1.join();
27     }
28     catch(Exception e)
29     {
30         System.out.println(e);
31     }
32     thread2.start();
33     thread3.start();
```

## Java Thread Methods VIII

```
34     }  
35 }
```

## Java Thread Methods IX

**public final int getPriority():** It is used to check the priority of the thread. When a thread is created, some priority is assigned to it. This priority is assigned either by the JVM or by the programmer explicitly while creating the thread.

Example:

```
1 public class JavaGetPriorityExp extends Thread
2 {
3     public void run()
4     {
5         System.out.println("running thread name is:" + Thread.currentThread().getName());
6     }
7     public static void main(String args[])
8     {
9         JavaGetPriorityExp t1 = new JavaGetPriorityExp();
10        JavaGetPriorityExp t2 = new JavaGetPriorityExp();
11        System.out.println("t1 thread priority : " + t1.getPriority());
12        System.out.println("t2 thread priority : " + t2.getPriority());
13        t1.start();
```

## Java Thread Methods X

```
14         t2.start();  
15     }  
16 }
```

## Java Thread Methods XI

**public final void setPriority():** This method is used to change the priority of the thread.

The priority of every thread is represented by the integer number from 1 to 10.

- public static int MIN\_PRIORITY
- public static int NORM\_PRIORITY
- public static int MAX\_PRIORITY

Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10.

## Java Thread Methods XII

```
1 public class JavaSetPriorityExp1 extends Thread
2 {
3     public void run()
4     {
5         System.out.println("Priority of thread is: "+Thread.currentThread().getPriority());
6     }
7     public static void main(String args[])
8     {
9         JavaSetPriorityExp1 t1=new JavaSetPriorityExp1();
10        t1.setPriority(Thread.MAX_PRIORITY);
11        t1.start();
12    }
13 }
```

### Java Thread Methods XIII

**public final String getName():** This method of thread class is used to return the name of the thread. We cannot override this method in our program, as this method is final. Example:

```
1 public class GetNameExample extends Thread
2 {
3     public void run()
4     {
5         System.out.println("Thread is running...");
6     }
7     public static void main(String args[])
8     {
9         // creating two threads
10        GetNameExample thread1=new GetNameExample();
11        GetNameExample thread2=new GetNameExample();
12        System.out.println("Name of thread1: "+ thread1.getName());
13        System.out.println("Name of thread2: "+thread2.getName());
14        thread1.start();
15        thread2.start();
```

## Java Thread Methods XIV

```
16     }  
17 }
```

## Java Thread Methods XV

**public final void setName():** This method changes the name of the thread.

Example:

```
1 public class SetNameExample extends Thread
2 {
3     public void run()
4     {
5         System.out.println("running...");
6     }
7     public static void main(String args[])
8     {
9         SetNameExample thread1=new SetNameExample();
10        SetNameExample thread2=new SetNameExample();
11        thread1.start();
12        thread2.start();
13        thread1.setName("Kadamb Sachdeva");
14        thread2.setName("Great learning");
15        System.out.println("After changing name of thread1: "+thread1.getName());
16        System.out.println("After changing name of thread2: "+thread2.getName());
```

## Java Thread Methods XVI

```
17     }  
18 }
```

## Java Thread Methods XVII

**public long getId():** It returns the identifier of the thread. The thread ID is a number generated when the thread was created. This ID cannot be changed during its lifetime. But when the thread is terminated, the ID can be reused.

Example:

```
1 public class GetIdExample extends Thread
2 {
3     public void run()
4     {
5         System.out.println("running...");
6     }
7     public static void main(String args[])
8     {
9         GetIdExample thread1=new GetIdExample();
10        System.out.println("Name of thread1: "+thread1.getName());
11        System.out.println("Id of thread1: "+thread1.getId());
12        thread1.start();
13    }
```

## Java Thread Methods XVIII

14 }

## Java Thread Methods XIX

**public final boolean isAlive():** This method checks if the thread is alive. A thread is in the alive state if the start() method of thread class has been called and the thread has not yet died.

Example:

```
1 public class JavaIsAliveExp extends Thread
2 {
3     public void run()
4     {
5         try
6         {
7             Thread.sleep(300);
8             System.out.println("is run() method isAlive "+Thread.currentThread().isAlive());
9         }
10        catch (InterruptedException ie)
11        {
12        }
13    }
```

## Java Thread Methods XX

```
14 public static void main(String[] args)
15 {
16     JavaIsAliveExp thread1 = new JavaIsAliveExp();
17     System.out.println("before starting thread isAlive: "+thread1.isAlive());
18     thread1.start();
19     System.out.println("after starting thread isAlive: "+thread1.isAlive());
20 }
21 }
```

## Java Thread Methods XXI

**public static void yield():** This method pauses the execution of the current thread to execute other threads temporarily.

Example:

```
1 public class JavaYieldExp extends Thread
2 {
3     public void run()
4     {
5         for (int i=0; i<3 ; i++)
6             System.out.println(Thread.currentThread().getName() + " in control");
7     }
8     public static void main(String[]args)
9     {
10         JavaYieldExp thread1 = new JavaYieldExp();
11         JavaYieldExp thread2 = new JavaYieldExp();
12         thread1.start();
13         thread2.start();
14         for (int i=0; i<3; i++)
15         {
```

## Java Thread Methods XXII

```
16         thread1.yield();
17         System.out.println(Thread.currentThread().getName() + " in control");
18     }
19 }
20 }
```

### Java Thread Methods XXIII

**public final void suspend():** This method is used to suspend the currently running thread temporarily. Using the resume() method, you can resume the suspended thread.

Example:

```
1 public class JavaSuspendExp extends Thread
2 {
3     public void run()
4     {
5         for(int i=1; i<5; i++)
6         {
7             try
8             {
9                 sleep(500);
10                System.out.println(Thread.currentThread().getName());
11            }
12            catch (InterruptedException e)
13            {

```

## Java Thread Methods XXIV

```
14         System.out.println(e);
15     }
16     System.out.println(i);
17 }
18 }
19 public static void main(String args[])
20 {
21     JavaSuspendExp thread1=new JavaSuspendExp ();
22     JavaSuspendExp thread2=new JavaSuspendExp ();
23     JavaSuspendExp thread3=new JavaSuspendExp ();
24     thread1.start();
25     thread2.start();
26     thread2.suspend();
27     thread3.start();
28 }
29 }
```

## Java Thread Methods XXV

**public final void resume():** This method is used to resume the suspended thread. It is only used with the suspend() method.

Example:

```
1 public class JavaResumeExp extends Thread
2 {
3     public void run()
4     {
5         for(int i=1; i<5; i++)
6         {
7             try
8             {
9                 sleep(500);
10                System.out.println(Thread.currentThread().getName());
11            }
12            catch (InterruptedException e)
13            {
14                System.out.println(e);
```

## Java Thread Methods XXVI

```
15         }
16         System.out.println(i);
17     }
18 }
19 public static void main(String args[])
20 {
21     JavaResumeExp thread1=new JavaResumeExp ();
22     JavaResumeExp thread2=new JavaResumeExp ();
23     JavaResumeExp thread3=new JavaResumeExp ();
24     thread1.start();
25     thread2.start();
26     thread2.suspend();
27     thread3.start();
28     thread2.resume();
29 }
30 }
```

## Java Thread Methods XXVII

**public final void stop():** As the name suggests, this method is used to stop the currently running thread. Remember, once the thread execution is stopped, it cannot be restarted.

Example:

```
1 public class JavaStopExp extends Thread
2 {
3     public void run()
4     {
5         for(int i=1; i<5; i++)
6         {
7             try
8             {
9                 sleep(500);
10                System.out.println(Thread.currentThread().getName());
11            }
12            catch (InterruptedException e)
13            {

```

## Java Thread Methods XXVIII

```
14         System.out.println(e);
15     }
16     System.out.println(i);
17 }
18 }
19 public static void main(String args[])
20 {
21     JavaStopExp thread1=new JavaStopExp ();
22     JavaStopExp thread2=new JavaStopExp ();
23     JavaStopExp thread3=new JavaStopExp ();
24     thread1.start();
25     thread2.start();
26     thread3.stop();
27     System.out.println("Thread thread3 is stopped");
28 }
29 }
```

### Java Thread Methods XXIX

**public void destroy():** This thread method destroys the thread group as well as its subgroups.

- ✓ A ThreadGroup represents a set of threads.
- ✓ A thread group can also include the other thread group.
- ✓ The thread group creates a tree in which every thread group except the initial thread group has a parent.
- ✓ A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

## Java Thread Methods XXX

```
1 public class JavaDestroyExp extends Thread
2 {
3     JavaDestroyExp(String threadname, ThreadGroup tg)
4     {
5         super(tg, threadname);
6         start();
7     }
8     public void run()
9     {
10        for (int i = 0; i < 2; i++)
11        {
12            try
13            {
14                Thread.sleep(10);
15            }
16            catch (InterruptedException ex)
17            {
18                System.out.println("Exception encountered");
19            }
20        }
21    }
22 }
```

## Java Thread Methods XXXI

```
20     }
21     System.out.println(Thread.currentThread().getName() + " finished executing");
22 }
23 public static void main(String arg[]) throws InterruptedException, SecurityException
24 {
25     ThreadGroup g1 = new ThreadGroup("Parent thread");
26     ThreadGroup g2 = new ThreadGroup(g1, "child thread");
27     JavaDestroyExp thread1 = new JavaDestroyExp("Thread-1", g1);
28     JavaDestroyExp thread2 = new JavaDestroyExp("Thread-2", g1);
29     thread1.join();
30     thread2.join();
31     g2.destroy();
32     System.out.println(g2.getName() + " destroyed");
33     g1.destroy();
34     System.out.println(g1.getName() + " destroyed");
35 }
36 }
```

### Java Thread Methods XXXII

**public final boolean isDaemon():** This thread method will check if the thread is a daemon thread or not. If it is a daemon thread, then it will return true else, it will return false.

- ✓ daemon thread is a thread that will not stop the Java Virtual Machine (JVM) from exiting when the program is ended, but the thread is still running.
- ✓ Daemon thread in Java is a low-priority thread that runs in the background to perform tasks such as garbage collection. Daemon thread in Java is also a service provider thread that provides services to the user thread.

### Java Thread Methods XXXIII

```
1 public class JavaIsDaemonExp extends Thread
2 {
3     public void run()
4     {
5         //checking for daemon thread
6         if(Thread.currentThread().isDaemon())
7         {
8             System.out.println("daemon thread work");
9         }
10        else
11        {
12            System.out.println("user thread work");
13        }
14    }
15    public static void main(String[] args)
16    {
17        JavaIsDaemonExp thread1=new JavaIsDaemonExp();
18        JavaIsDaemonExp thread2=new JavaIsDaemonExp();
19        JavaIsDaemonExp thread3=new JavaIsDaemonExp();
```

## Java Thread Methods XXXIV

```
20     thread1.setDaemon(true);
21     thread1.start();
22     thread2.start();
23     thread3.start();
24 }
25 }
```

### Java Thread Methods XXXV

**public final void setDaemon(boolean on):** This method of a thread is used to identify or mark the thread either daemon or a user thread. The JVM automatically terminates this thread when all the user threads die.

*This thread method must run before the start of the execution of the thread.*

```
1 public class JavaSetDaemonExp1 extends Thread
2 {
3     public void run()
4     {
5         if(Thread.currentThread().isDaemon())
6         {
7             System.out.println("daemon thread work");
8         }
9         else
10        {
11            System.out.println("user thread work");
12        }
13    }
```

## Java Thread Methods XXXVI

```
14  public static void main(String[] args)
15  {
16      JavaSetDaemonExp1 thread1=new JavaSetDaemonExp1();
17      JavaSetDaemonExp1 thread2=new JavaSetDaemonExp1();
18      JavaSetDaemonExp1 thread3=new JavaSetDaemonExp1();
19      thread1.setDaemon(true);
20      thread1.start();
21      thread2.setDaemon(true);
22      thread2.start();
23      thread3.start();
24  }
25 }
```

## Java Thread Methods XXXVII

**public void interrupt():** This method of a thread is used to interrupt the currently executing thread. This method can only be called when the thread is in sleeping or waiting state.

*But if the thread is not in the sleeping or waiting state, then the interrupt() method will not interrupt the thread but will set the interrupt flag to true.*

Example:

```
1 public class JavaInterruptExp1 extends Thread
2 {
3     public void run()
4     {
5         try
6         {
7             Thread.sleep(1000);
8             System.out.println("javatpoint");
9         }
```

## Java Thread Methods XXXVIII

```
10     catch (InterruptedException e)
11     {
12         throw new RuntimeException("Thread interrupted..." + e);
13     }
14 }
15 public static void main(String args[])
16 {
17     JavaInterruptExp1 thread1 = new JavaInterruptExp1();
18     thread1.start();
19     try
20     {
21         thread1.interrupt();
22     }
23     catch (Exception e)
24     {
25         System.out.println("Exception handled " + e);
26     }
27 }
28 }
```

### Java Thread Methods XXXIX

**public boolean isInterrupted():** This thread method is used to test whether the thread is interrupted or not. It will return the value of the internal flag as true or false, i.e. if the thread is interrupted, it will return true else, it will return false.

```
1 public class JavaIsInterruptedExp extends Thread
2 {
3     public void run()
4     {
5         for(int i=1; i<=3; i++)
6         {
7             System.out.println("doing task....: "+i);
8         }
9     }
10    public static void main(String args[])throws InterruptedException
11    {
12        JavaIsInterruptedExp thread1=new JavaIsInterruptedExp();
13        JavaIsInterruptedExp thread2=new JavaIsInterruptedExp();
14        thread1.start();
```

## Java Thread Methods XL

```
15      thread2.start();
16      System.out.println("is thread interrupted.: "+thread1.isInterrupted());
17      System.out.println("is thread interrupted.: "+thread2.isInterrupted());
18      thread1.interrupt();
19      System.out.println("is thread interrupted.: "+thread1.isInterrupted());
20      System.out.println("is thread interrupted.: "+thread2.isInterrupted());
21  }
22 }
```

## Java Thread Methods XLI

**public static boolean interrupted():** This thread method is used to check if the current thread is interrupted or not. If this threading method is to be called twice in succession, then the second call will return as false.

*If the interrupt status of the thread is true, then this thread method will set it to false.*

Example:

```
1 public class JavaInterruptedExp extends Thread
2 {
3     public void run()
4     {
5         for(int i=1; i<=3; i++)
6         {
7             System.out.println("doing task....: "+i);
8         }
9     }
```

## Java Thread Methods XLII

```
10 public static void main(String args[])throws InterruptedException
11 {
12     JavaInterruptedExp thread1=new JavaInterruptedExp();
13     JavaInterruptedExp thread2=new JavaInterruptedExp();
14     thread1.start();
15     thread2.start();
16     System.out.println("is thread thread1 interrupted..." +thread1.interrupted());
17     thread1.interrupt();
18     System.out.println("is thread thread1 interrupted..." +thread1.interrupted());
19     System.out.println("is thread thread2 interrupted..." +thread2.interrupted());
20 }
21 }
```

### Java Thread Methods XLIII

**public static int activeCount():** This method of the thread is used to return the no. of active threads in the currently executing thread's thread group.

*The number returned by this threading method is only an estimate number as the number of threads dynamically changes while this method traverses internal data structures.*

```
1 public class JavaActiveCountExp extends Thread
2 {
3     JavaActiveCountExp(String threadname, ThreadGroup tg)
4     {
5         super(tg, threadname);
6         start();
7     }
8     public void run()
9     {
10        System.out.println("running thread name is:"
11        +Thread.currentThread().getName());
```

## Java Thread Methods XLIV

```
12     }
13     public static void main(String arg[])
14     {
15         ThreadGroup g1 = new ThreadGroup("parent thread group");
16         JavaActiveCountExp thread1 = new JavaActiveCountExp("Thread-1", g1);
17         JavaActiveCountExp thread2 = new JavaActiveCountExp("Thread-2", g1);
18         System.out.println("number of active thread: " + g1.activeCount());
19     }
20 }
```

## Java Thread Methods XLV

**public final void checkAccess():** This thread method identifies if the current thread has permission to modify the thread.

```
1 public class JavaCheckAccessExp extends Thread
2 {
3     public void run()
4     {
5         System.out.println(Thread.currentThread().getName()+" finished executing");
6     }
7     public static void main(String arg[]) throws InterruptedException, SecurityException
8     {
9         JavaCheckAccessExp thread1 = new JavaCheckAccessExp();
10        JavaCheckAccessExp thread2 = new JavaCheckAccessExp();
11        thread1.start();
12        thread2.start();
13        thread1.checkAccess();
14        System.out.println(t1.getName() + " has access");
15        thread2.checkAccess();
16        System.out.println(t2.getName() + " has access");
```

## Java Thread Methods XLVI

```
17     }  
18 }
```

## Java Thread Methods XLVII

**public static boolean holdsLock(Object obj):** This thread method checks if the currently executing thread holds the monitor lock on the specified object. If it does, then this threading method will return true.

Example:

```
1 public class JavaHoldLockExp implements Runnable
2 {
3     public void run()
4     {
5         System.out.println("Currently executing thread is: " + Thread.currentThread().getName());
6         System.out.println("Does thread holds lock? " + Thread.holdsLock(this));
7         synchronized (this)
8         {
9             System.out.println("Does thread holds lock? " + Thread.holdsLock(this));
10        }
11    }
12    public static void main(String[] args)
13    {
```

### Java Thread Methods XLVIII

```
14      JavaHoldLockExp g1 = new JavaHoldLockExp();
15      Thread thread1 = new Thread(g1);
16      thread1.start();
17  }
18 }
```

## Multitasking/Multithreading/Multiprocessing I

✓ How to perform single task by multiple threads in Java?

If you have to perform a single task by many threads, have only one run() method.

```
1 class TestMultitasking1 extends Thread
2 {
3     public void run()
4     {
5         System.out.println("task one");
6     }
7     public static void main(String args[])
8     {
9         TestMultitasking1 t1=new TestMultitasking1();
10        TestMultitasking1 t2=new TestMultitasking1();
11        TestMultitasking1 t3=new TestMultitasking1();
12
13        t1.start();
14        t2.start();
15        t3.start();
16    }
17 }
```

## Multitasking/Multithreading/Multiprocessing II

### performing single task by multiple threads

```
1 class TestMultitasking2 implements Runnable
2 {
3     public void run()
4     {
5         System.out.println("task one");
6     }
7
8     public static void main(String args[])
9     {
10        Thread t1 = new Thread(new TestMultitasking2()); //passing anonymous object of
11        Thread t2 = new Thread(new TestMultitasking2());
12        t1.start();
13        t2.start();
14    }
15 }
16
17 }
```

### Multitasking/Multithreading/Multiprocessing III

✓ How to perform multiple tasks by multiple threads (multitasking in multithreading)?

If you have to perform multiple tasks by multiple threads, have multiple run() methods.

#### performing two tasks by two threads

```
1 class Simple1 extends Thread
2 {
3     public void run()
4     {
5         System.out.println("task one");
6     }
7 }
8
9 class Simple2 extends Thread
10 {
11     public void run()
12     {
13         System.out.println("task two");
14     }
15 }
```

## Multitasking/Multithreading/Multiprocessing IV

```
16
17 class TestMultitasking3
18 {
19     public static void main(String args[])
20     {
21         Simple1 t1=new Simple1();
22         Simple2 t2=new Simple2();
23
24         t1.start();
25         t2.start();
26     }
27 }
```

## Example I

No race condition:

```
1 import java.util.Random;
2 import java.util.concurrent.*;
3
4 public class MultiThreadedSumAlternative
5 {
6     private static final int TOTAL_NUMBERS = 1_000_000;
7     private static final int NUM_THREADS = 10;
8     private static final int NUMBERS_PER_THREAD = TOTAL_NUMBERS / NUM_THREADS;
9
10    // Thread class that returns the result
11    static class SumTask implements Callable<Long>
12    {
13        private int threadId;
14
15        public SumTask(int threadId)
16        {
17            this.threadId = threadId;
18        }
19
20        @Override
```

## Example II

```
21     public Long call()
22     {
23         Random random = new Random();
24         long partialSum = 0;
25         for (int i = 0; i < NUMBERS_PER_THREAD; i++)
26         {
27             partialSum += random.nextInt(1000);
28         }
29         System.out.println("Thread " + threadId + " calculated sum: " + partialSum);
30         return partialSum;
31     }
32 }
33
34
35
36 public static void main(String[] args)
37 {
38     System.out.println("Starting multi-threaded sum calculation...");
39     System.out.println("Total numbers: " + TOTAL_NUMBERS);
40     System.out.println("Threads: " + NUM_THREADS);
```

### Example III

```
41 System.out.println("Numbers per thread: " + NUMBERS_PER_THREAD);
42
43 ExecutorService executor = Executors.newFixedThreadPool(NUM_THREADS);
44 Future<Long>[] futures = new Future[NUM_THREADS];
45
46 try
47 {
48     // Submit all tasks
49     for (int i = 0; i < NUM_THREADS; i++)
50     {
51         futures[i] = executor.submit(new SumTask(i + 1));
52     }
53
54     // Collect and sum all results
55     long totalSum = 0;
56     for (int i = 0; i < NUM_THREADS; i++)
57     {
58         totalSum += futures[i].get(); // This waits for the result
59     }
60
61     System.out.println("\nFinal total sum: " + totalSum);
62
```

### Example IV

```
63     }
64     catch (InterruptedException | ExecutionException e)
65     {
66         System.out.println("Error in execution: " + e.getMessage());
67     }
68     finally
69     {
70         executor.shutdown();
71     }
72 }
73 }
```

## Example V

Race condition:

```
1 public class SimplestRace
2 {
3     static int counter = 0;
4     public static void main(String[] args) throws InterruptedException
5     {
6         Thread t1 = new Thread(new Runnable()
7         {
8             @Override
9             public void run()
10            {
11                for (int j = 0; j < 1000; j++)
12                {
13                    unsafeCounter++;
14                }
15            }
16        });
17    }
18 }
19
```

### Example VI

```
20 Thread t2 = new Thread() ->
21 {
22     for (int i = 0; i < 1000; i++) counter++;
23 }
24 );
25
26 t1.start();
27 t2.start();
28
29 t1.join();
30 t2.join();
31
32 System.out.println("Final counter: " + counter);
33 // Should be 2000, but often less due to race condition!
34 }
35 }
```

## Example VII

```
1 public class SimpleRaceCondition
2 {
3     static int balance = 100;
4
5     public static void main(String[] args) throws InterruptedException
6     {
7         // Create two threads (like two people)
8         Thread person1 = new Thread() ->
9         {
10             if (balance >= 100)
11             {
12                 // Simulate some delay
13                 try
14                 {
15                     Thread.sleep(10);
16                 }
17                 catch (Exception e)
18                 {
19
20                 }
```

### Example VIII

```
21         balance = balance -100;
22         System.out.println("Person 1 withdrew $100");
23     }
24 }
25 );
26
27 Thread person2 = new Thread() ->
28 {
29     if (balance >= 100)
30     {
31         // Simulate some delay
32         try
33         {
34             Thread.sleep(10);
35         }
36         catch (Exception e)
37         {
38         }
39     }
40     balance = balance -100;
```

### Example IX

```
41         System.out.println("Person 2 withdrew $100");
42     }
43 }
44 );
45
46 // Start both threads at almost same time
47 person1.start();
48 person2.start();
49
50 // Wait for both to finish
51 person1.join();
52 person2.join();
53
54 System.out.println("Final balance: $" + balance);
55 }
56 }
```

### Example X

```
1 // WITHOUT synchronization -RACE CONDITION!
2 totalSum += partialSum;
3
4 // This could happen:
5 // Thread 1: reads totalSum (value = 100)
6 // Thread 2: reads totalSum (value = 100)
7 // Thread 1: adds 50 writes 150
8 // Thread 2: adds 30 writes 130 (WRONG! Should be 180)

1 public synchronized void myMethod()
2 {
3     // Only one thread can execute this method at a time
4     // Lock is on the current object instance (this)
5 }
```

### Example XI

```
1 public void myMethod()
2 {
3     // Some non-critical code here
4     synchronized(this)
5     {
6         // or any object
7         // Critical section -only one thread at a time
8     }
9     // More non-critical code here
10 }
11
1 public static synchronized void myStaticMethod()
2 {
3     // Lock is on the Class object (MyClass.class)
4 }
```

## Example XII

### Example 1: Basic Counter (The Problem)

```
1 class Counter
2 {
3     private int count = 0;
4     public void increment()
5     {
6         count++; // NOT thread-safe!
7     }
8     public int getCount()
9     {
10        return count;
11    }
12 }
13
14 }
```

### Example XIII

Fixed with Synchronized:

```
1 class SafeCounter
2 {
3     private int count = 0;
4
5     // Synchronized method
6     public synchronized void increment()
7     {
8         count++;
9     }
10
11    public synchronized int getCount()
12    {
13        return count;
14    }
15 }
```

## Example XIV

### Using Synchronized Block

```
1 class SafeCounter
2 {
3     private int count = 0;
4     private final Object lock = new Object(); // Explicit lock object
5
6     public void increment()
7     {
8         synchronized(lock)
9         {
10             count++;
11         }
12     }
13
14     public int getCount()
15     {
16         synchronized(lock)
17         {
18             return count;
19         }
20     }
21 }
```

### Example XV

$$\begin{array}{r} 20 \\ 21 \end{array} \left. \vphantom{\begin{array}{r} 20 \\ 21 \end{array}} \right\}$$

## Example XVI

Sum 1 million numbers through 10 threads without join():

```
1 public class MillionSumMultiThreaded
2 {
3     private static final int TOTAL_NUMBERS = 1_000_000;
4     private static final int THREAD_COUNT = 10;
5     private static final int NUMBERS_PER_THREAD = TOTAL_NUMBERS / THREAD_COUNT;
6
7     private static long totalSum = 0;
8
9     static class SumWorker extends Thread
10    {
11        private final int start;
12        private final int end;
13        private long partialSum = 0;
14
15        public SumWorker(int start, int end)
16        {
17            this.start = start;
18            this.end = end;
19        }
20    }
```

## Example XVII

```
21     @Override
22     public void run()
23     {
24         for (int i = start; i <= end; i++)
25         {
26             partialSum += i;
27         }
28         System.out.println(Thread.currentThread().getName() +
29             "calculated sum from " + start + " to " + end +
30             " = " + partialSum);
31     }
32     public long getPartialSum()
33     {
34         return partialSum;
35     }
36 }
37
38 public static void main(String[] args)
39 {
40
```

### Example XVIII

```
41 SumWorker[] workers = new SumWorker[THREAD_COUNT];
42 Thread[] threads = new Thread[THREAD_COUNT];
43
44 // Create and start threads
45 for (int i = 0; i < THREAD_COUNT; i++)
46 {
47     int start = (i * NUMBERS_PER_THREAD) + 1;
48     int end = (i == THREAD_COUNT - 1) ? TOTAL_NUMBERS : (i + 1) *
        NUMBERS_PER_THREAD;
49
50     workers[i] = new SumWorker(start, end);
51     threads[i] = new Thread(workers[i], "Thread-" + (i + 1));
52     threads[i].start();
53 }
54
55 // Wait for all threads to complete and accumulate results
56 try
57 {
58     for (int i = 0; i < THREAD_COUNT; i++)
59     {
60         threads[i].join();
```

### Example XIX

```
61         totalSum += workers[i].getPartialSum();
62     }
63 }
64 catch (InterruptedException e)
65 {
66     e.printStackTrace();
67 }
68
69 // Print final result
70 System.out.println("\nTotal sum of first " + TOTAL_NUMBERS + " numbers: " + totalSum);
71
72 // Verify with formula:  $n(n+1)/2$ 
73 long expectedSum = (long) TOTAL_NUMBERS * (TOTAL_NUMBERS + 1) / 2;
74 System.out.println("Expected sum (using formula): " + expectedSum);
75 System.out.println("Result is correct: " + (totalSum == expectedSum));
76 }
77 }
```

### Example XX

Sum 1 million numbers through 10 threads with join():

```
1 public class MillionSumSequentialThreads
2 {
3     private static final int TOTAL_NUMBERS = 1_000_000;
4     private static final int THREAD_COUNT = 10;
5     private static final int NUMBERS_PER_THREAD = TOTAL_NUMBERS / THREAD_COUNT;
6
7     private static long totalSum = 0;
8
9     static class SumWorker extends Thread
10    {
11        private final int start;
12        private final int end;
13        private long partialSum = 0;
14
15        public SumWorker(int start, int end)
16        {
17            this.start = start;
18            this.end = end;
19        }
20    }
```

## Example XXI

```
21     @Override
22     public void run()
23     {
24         for (int i = start; i <= end; i++)
25         {
26             partialSum += i;
27         }
28         System.out.println(Thread.currentThread().getName() +
29             "calculated sum from " + start + " to " + end +
30             " = " + partialSum);
31     }
32     public long getPartialSum()
33     {
34         return partialSum;
35     }
36 }
37
38 public static void main(String[] args)
39 {
40
```

## Example XXII

```
41 SumWorker[] workers = new SumWorker[THREAD_COUNT];
42 Thread[] threads = new Thread[THREAD_COUNT];
43
44 System.out.println("Starting threads in sequence...\n");
45
46 // Create, start and immediately join each thread
47 for (int i = 0; i < THREAD_COUNT; i++)
48 {
49     int start = (i * NUMBERS_PER_THREAD) + 1;
50     int end = (i == THREAD_COUNT - 1) ? TOTAL_NUMBERS : (i + 1) *
        NUMBERS_PER_THREAD;
51
52     workers[i] = new SumWorker(start, end);
53     threads[i] = new Thread(workers[i], "Thread-" + (i + 1));
54
55     // Start the thread
56     threads[i].start();
57
58     // Immediately join -this will block until the thread completes
59     try
60     {
61         threads[i].join();
```

### Example XXIII

```
62         totalSum += workers[i].getPartialSum();
63         System.out.println(" Thread-" + (i + 1) + " completed and joined successfully\n");
64     }
65     catch (InterruptedException e)
66     {
67         e.printStackTrace();
68     }
69 }
70
71 // Print final result
72 System.out.println("Total sum of first " + TOTAL_NUMBERS + " numbers: " + totalSum);
73
74 // Verify with formula:  $n(n+1)/2$ 
75 long expectedSum = (long) TOTAL_NUMBERS * (TOTAL_NUMBERS + 1) / 2;
76 System.out.println("Expected sum (using formula): " + expectedSum);
77 System.out.println("Result is correct: " + (totalSum == expectedSum));
78 }
79 }
```