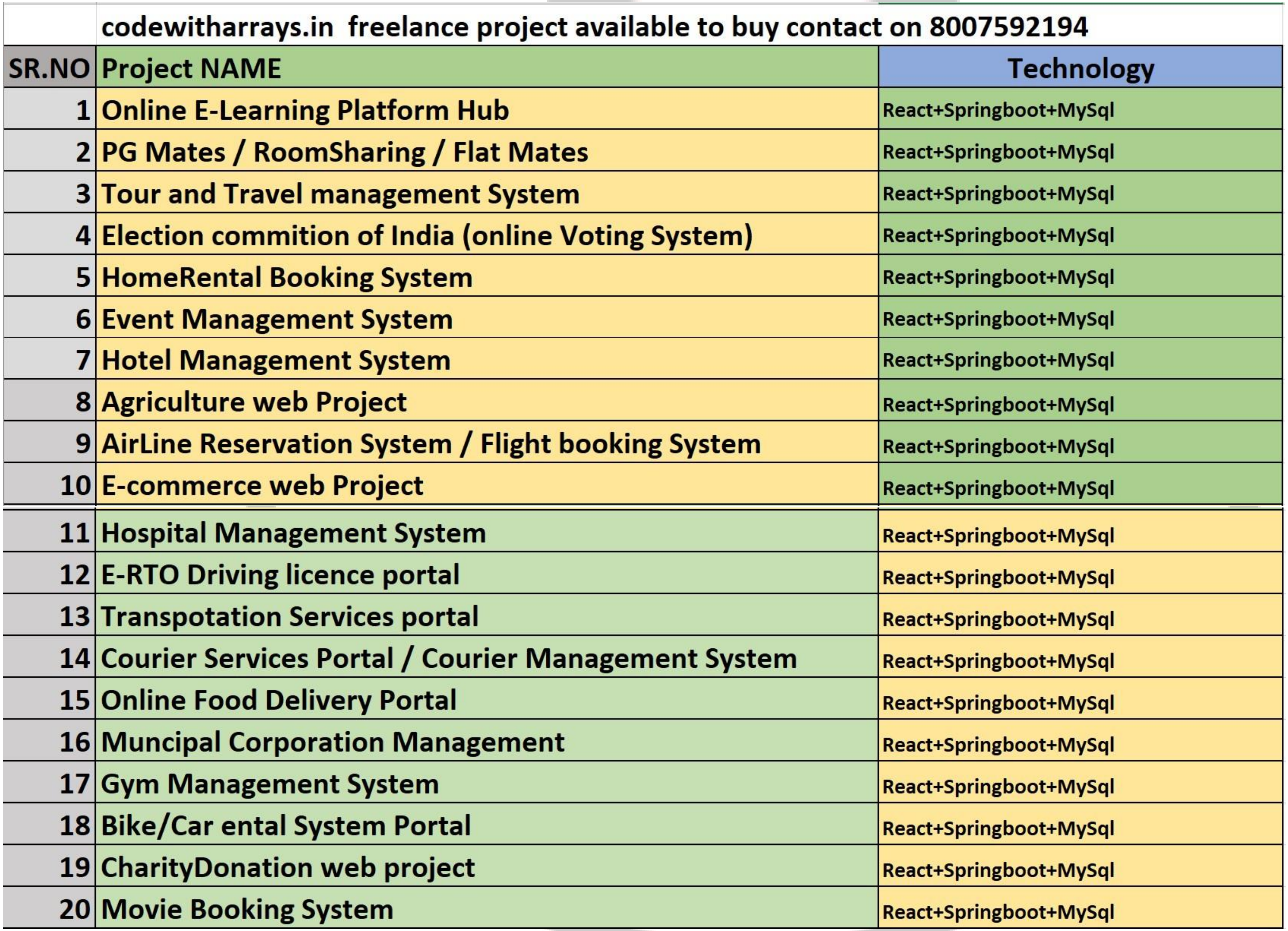

	codewitharrays.in freelance project available to buy contact on 8007592194	
SR.NO	Project NAME	Technology
1	Online E-Learning Platform Hub	React+Springboot+MySql
2	PG Mates / RoomSharing / Flat Mates	React+Springboot+MySql
3	Tour and Travel management System	React+Springboot+MySql
4	Election commition of India (online Voting System)	React+Springboot+MySql
5	HomeRental Booking System	React+Springboot+MySql
6	Event Management System	React+Springboot+MySql
7	Hotel Management System	React+Springboot+MySql
8	Agriculture web Project	React+Springboot+MySql
9	AirLine Reservation System / Flight booking System	React+Springboot+MySql
10	E-commerce web Project	React+Springboot+MySql
11	Hospital Management System	React+Springboot+MySql
12	E-RTO Driving licence portal	React+Springboot+MySql
13	Transpotation Services portal	React+Springboot+MySql
14	Courier Services Portal / Courier Management System	React+Springboot+MySql
15	Online Food Delivery Portal	React+Springboot+MySql
16	Muncipal Corporation Management	React+Springboot+MySql
17	Gym Management System	React+Springboot+MySql
18	Bike/Car ental System Portal	React+Springboot+MySql
19	CharityDonation web project	React+Springboot+MySql
20	Movie Booking System	React+Springboot+MySql

	freelance_Project available to buy contact on 8007592194	
21	Job Portal web project	React+Springboot+MySql
22	LIC Insurance Portal	React+Springboot+MySql
23	Employee Management System	React+Springboot+MySql
24	Payroll Management System	React+Springboot+MySql
25	RealEstate Property Project	React+Springboot+MySql
26	Marriage Hall Booking Project	React+Springboot+MySql
27	Online Student Management portal	React+Springboot+MySql
28	Resturant management System	React+Springboot+MySql
29	Solar Management Project	React+Springboot+MySql
30	OneStepService LinkLabourContractor	React+Springboot+MySql
31	Vehical Service Center Portal	React+Springboot+MySql
32	E-wallet Banking Project	React+Springboot+MySql
33	Blogg Application Project	React+Springboot+MySql
34	Car Parking booking Project	React+Springboot+MySql
35	OLA Cab Booking Portal	React+NextJs+Springboot+MySql
36	Society management Portal	React+Springboot+MySql
37	E-College Portal	React+Springboot+MySql
38	FoodWaste Management Donate System	React+Springboot+MySql
39	Sports Ground Booking	React+Springboot+MySql
40	BloodBank mangement System	React+Springboot+MySql

41	Bus Tickit Booking Project	React+Springboot+MySql
42	Fruite Delivery Project	React+Springboot+MySql
43	Woodworks Bed Shop	React+Springboot+MySql
44	Online Dairy Product sell Project	React+Springboot+MySql
45	Online E-Pharma medicine sell Project	React+Springboot+MySql
46	FarmerMarketplace Web Project	React+Springboot+MySql
47	Online Cloth Store Project	React+Springboot+MySql
48	Train Ticket Booking Project	React+Springboot+MySql
49	Quizz Application Project	JSP+Springboot+MySql
50	Hotel Room Booking Project	React+Springboot+MySql

51	Online Crime Reporting Portal Project	React+Springboot+MySql
52	Online Child Adoption Portal Project	React+Springboot+MySql
53	online Pizza Delivery System Project	React+Springboot+MySql
54	Online Social Complaint Portal Project	React+Springboot+MySql
55	Electric Vehical management system Project	React+Springboot+MySql
56	Online mess / Tiffin management System Project	React+Springboot+MySql
57		React+Springboot+MySql
58		React+Springboot+MySql
59		React+Springboot+MySql
60		React+Springboot+MySql

Spring Boot + React JS + MySQL Project List

Sr.No	Project Name	YouTube Link
1	Online E-Learning Hub Platform Project	https://youtu.be/KMjyBaWmgzg?si=YckHuNzs7eC84-IW
2	PG Mate / Room sharing/Flat sharing	https://youtu.be/4P9cIHg3wvk?si=4uEsi0962CG6Xodp
3	Tour and Travel System Project Version 1.0	https://youtu.be/-UHOBywHaP8?si=KHHfE_A0uv725f12
4	Marriage Hall Booking	https://youtu.be/VXz0kZQi5to?si=llOS-QG3TpAFP5k7
5	Ecommerce Shopping project	https://youtu.be/vJ_C6LkhrZ0?si=YhcBylSErvdn7paq
6	Bike Rental System Project	https://youtu.be/FIzsAmIBCbk?si=7ujQTJqEgkQ8ju2H
7	Multi-Restaurant management system	https://youtu.be/pvV-pM2Jf3s?si=PgvnT-yFc8ktrDxB
8	Hospital management system Project	https://youtu.be/IynIouBZvY4?si=CXzQs3BsRkjKhZCw
9	Municipal Corporation system Project	https://youtu.be/cVMx9NVyI4I?si=qX0oQt-GT-LR_5jF
10	Tour and Travel System Project version 2.0	https://youtu.be/_4u0mB9mHXE?si=gDiAhKBowi2gNUKZ

Sr.No	Project Name	YouTube Link
11	Tour and Travel System Project version 3.0	https://youtu.be/Dm7nOdpasWg?si=P_Lh2gcOFhlyudug
12	Gym Management system Project	https://youtu.be/J8_7Zrkg7ag?si=LcxV51ynfUB7OptX
13	Online Driving License system Project	https://youtu.be/3yRzsMs8TLE?si=JRI_z4FDx4Gmt7fn
14	Online Flight Booking system Project	https://youtu.be/m755rOwdk8U?si=HURvAY2VnizIyJlh
15	Employee management system project	https://youtu.be/lD1iE3W_GRw?si=Y_jv1xV_BljhrD0H
16	Online student school or college portal	https://youtu.be/4A25aEKfei0?si=RoVgZtxMk9TPdQvD
17	Online movie booking system project	https://youtu.be/Lfjv_U74SC4?si=fiDvrhhrjb4KSlSm
18	Online Pizza Delivery system project	https://youtu.be/Tp3izreZ458?si=8eWAOzA8SVdNwlyM
19	Online Crime Reporting system Project	https://youtu.be/0UlzReSk9tQ?si=6vN0e70TVY1GOwPO
20	Online Children Adoption Project	https://youtu.be/3T5HC2HKyT4?si=bntP78niYH802I7N

Multithreading Interview Questions and Answers

Q. What are the states in the lifecycle of a Thread?

A java thread can be in any of following thread states during it’s life cycle i.e. New, Runnable, Blocked, Waiting, Timed Waiting or Terminated. These are also called life cycle events of a thread in java.

- New
- Runnable
- Running
- Non-Runnable (Blocked)
- Terminated

1. New: The thread is in new state if you create an instance of Thread class but before the invocation of start() method.
2. Runnable: The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
3. Running: The thread is in running state if the thread scheduler has selected it.
4. Non-Runnable (Blocked): This is the state when the thread is still alive, but is currently not eligible to run.
5. Terminated: A thread is in terminated or dead state when its run() method exits.

Q. What are the different ways of implementing thread?

There are two ways to create a thread: * extends Thread class * implement Runnable interface

1. Extends Thread class
Create a thread by a new class that extends Thread class and create an instance of that class. The extending class must override run() method which is the entry point of new thread.

```java
public class MyThread extends Thread {

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

Output

```
Thread started running..
```

2. Implementing the Runnable Interface

After implementing runnable interface, the class needs to implement the run() method, which is `public void run()`.

- run() method introduces a concurrent thread into your program. This thread will end when run() returns.
- You must specify the code for your thread inside run() method.
- run() method can call other methods, can use other classes and declare variables just like any other normal method.

```
class MyThread implements Runnable {

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

Difference between Runnable vs Thread

- Implementing Runnable is the preferred way to do it. Here, you're not really specializing or modifying the thread's behavior. You're just giving the thread something to run. That means composition is the better way to go.
- Java only supports single inheritance, so you can only extend one class.
- Instantiating an interface gives a cleaner separation between your code and the implementation of threads.
- Implementing Runnable makes your class more flexible. If you extend Thread then the action you're doing is always going to be in a thread. However, if you implement Runnable it doesn't have to be. You can run it in a thread, or pass it to some kind of executor service, or just pass it around as a task within a single threaded application.

<b><a href="#">↥ back to top</a></b>

Q. What is the difference between Process and Thread?

Both processes and threads are independent sequences of execution. The typical difference is that threads run in a **shared memory space**, while processes run in **separate memory spaces**.

Process

- An executing instance of a program is called a process.
- Some operating systems use the term **task** to refer to a program that is being executed.

- A process is always stored in the main memory also termed as the primary memory or random access memory.
- Therefore, a process is termed as an active entity. It disappears if the machine is rebooted.
- Several process may be associated with a same program.
- On a multiprocessor system, multiple processes can be executed in parallel.
- On a uni-processor system, though true parallelism is not achieved, a process scheduling algorithm is applied and the processor is scheduled to execute each process one at a time yielding an illusion of concurrency.

Thread

- A thread is a subset of the process.
- It is termed as a **lightweight process**, since it is similar to a real process but executes within the context of a process and shares the same resources allotted to the process by the kernel.
- Usually, a process has only one thread of control – one set of machine instructions executing at a time.
- A process may also be made up of multiple threads of execution that execute instructions concurrently.
- Multiple threads of control can exploit the true parallelism possible on multiprocessor systems.
- On a uni-processor system, a thread scheduling algorithm is applied and the processor is scheduled to run each thread one at a time.
- All the threads running within a process share the same address space, file descriptors, stack and other process related attributes.
- Since the threads of a process share the same memory, synchronizing the access to the shared data within the process gains unprecedented importance.

Q. What is difference between user Thread and daemon Thread?

Daemon threads are low priority threads which always run in background and user threads are high priority threads which always run in foreground. User Thread or Non-Daemon are designed to do specific or complex task where as daemon threads are used to perform supporting tasks.

Difference Between Daemon Threads And User Threads In Java

- User threads are created by the application (user) to perform some specific task. Where as daemon threads are mostly created by the JVM to perform some background tasks like garbage collection.

- JVM will wait for user threads to finish their tasks. JVM will not exit until all user threads finish their tasks. On the other side, JVM will not wait for daemon threads to finish their tasks. It will exit as soon as all user threads finish their tasks.

- User threads are high priority threads, They are designed mainly to execute some important task in an application. Where as daemon threads are less priority threads. They are designed to serve the user threads.
- User threads are foreground threads. They always run in foreground and perform some specific task assigned to them. Where as daemon threads are background threads. They always run in background and act in a supporting role to user threads.
- JVM will not force the user threads to terminate. It will wait for user threads to terminate themselves. On the other hand, JVM will force the daemon threads to terminate if all the user threads have finished their task.

create Daemon Thread

```
/**
 * Java Program to demonstrate difference beween a daemon and a user
thread .
 *
 */
public class DaemonThreadDemo {

    public static void main(String[] args) throws InterruptedException
{

        // main thread is a non-daemon thread
        String name = Thread.currentThread().getName();
        boolean isDaemon = Thread.currentThread().isDaemon();

        System.out.println("name: " + name + ", isDaemon: " +
isDaemon);

        // Any new thread spawned from main is also non-daemon or user
thread
        // as seen below:
        Runnable task = new Task();
        Thread t1 = new Thread(task, "T1");
        System.out.println("Thread spawned from main thread");
        System.out.println("name: " + t1.getName() + ", isDaemon: " +
t1.isDaemon());

        // though you can make a daemon thread by calling setDaemon()
        // before starting it as shown below:
        t1.setDaemon(true);
        t1.start();

        // let's wait for T1 to finish
        t1.join();
    }
    private static class Task implements Runnable {

        @Override
        public void run() {
            Thread t = Thread.currentThread();
            System.out.println("Thread made daemon by calling
setDaemon() method");
            System.out.println("name: " + t.getName() + ", isDaemon: "
+ t.isDaemon());

            // Any new thread created from daemon thread is also
daemon
            Thread t2 = new Thread("T2");
            System.out.println("Thread spawned from a daemon thread");
            System.out.println("name: " + t2.getName() + ", isDaemon:
" + t2.isDaemon());
        }
    }
}
```

Output

```
name: main, isDaemon: false
Thread spawned from main thread
name: T1, isDaemon: false
Thread made daemon by calling setDaemon() method
name: T1, isDaemon: true
Thread spawned from a daemon thread
name: T2, isDaemon: true
```

<b><a href="#">↥ back to top</a></b>

Q. How does thread communicate with each other?

Inter-thread communication is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of Object class:

- wait()
- notify()
- notifyAll()

```
class ThreadA {

    public static void main(String [] args) {
      ThreadB b = new ThreadB();
      b.start();
      synchronized(b) {
       try {
            System.out.println("Waiting for b to complete...");
            b.wait();
       } catch (InterruptedException e) {}
            System.out.println("Total is: " + b.total);
      }
    }
}

class ThreadB extends Thread {
 int total;

 public void run() {
   synchronized(this) {
    for(int i = 0; i < 100; i++) {
        total += i;
    }
    notify();
   }
  }
}
```

Q. What do you understand about Thread Priority?

Every thread in Java has a priority that helps the thread scheduler to determine the order in which threads scheduled. The threads with higher priority will usually run before and more frequently than lower priority threads. By default, all the threads had the same priority, i.e., they regarded as being equally distinguished by the scheduler, when a thread created it inherits its priority from the thread that created it.

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

- public static int MIN_PRIORITY
- public static int NORM_PRIORITY
- public static int MAX_PRIORITY

```
class TestMultiPriority1 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[]) {
        TestMultiPriority1 m1 = new TestMultiPriority1();
        TestMultiPriority1 m2 = new TestMultiPriority1();
        m1.setPriority(Thread.MIN_PRIORITY);
        m2.setPriority(Thread.MAX_PRIORITY);
        m1.start();
        m2.start();
    }
}
```

Output

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

<b><a href="#">↥ back to top</a></b>

Q. What is Thread Scheduler and Time Slicing?

Thread scheduler in java is the part of the JVM that decides which thread should run. The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Preemptive scheduling: The highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence.

Time slicing: A task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

Example: Thread Scheduler

```java
class FirstThread extends Thread {
    public void run(){
        for(int i = 0; i < 10; ++i) {
            System.out.println("I am in first thread");
            try{
                Thread.sleep(1000);
            }
            catch(InterruptedException ie) {
                System.out.println("Exception occurs ");
            }
        }
    }
}

class SecondThread {
    public static void main(String[] args){
        FirstThread ft = new FirstThread();
        ft.start();
        for(int j = 1; j < 10; ++j) {
            System.out.println("I am in second thread");
        }
    }
}
```

Output

```
cmd> java SecondThread
I am in second thread
I am in first thread
I am in second thread
I am in second thread
I am in second thread
I am in second thread
I am in second thread
I am in second thread
I am in second thread
I am in second thread
I am in first thread
I am in first thread
I am in first thread
I am in first thread
I am in first thread
I am in first thread
I am in first thread
I am in first thread
I am in first thread
```

```
<b><a href="#">↥ back to top</a></b>
```

Q. What is context-switching in multi-threading?

Context Switching is the process of storing and restoring of CPU state so that Thread execution can be resumed from the same point at a later point of time. Context Switching is the essential feature for multitasking operating system and support for multi-threaded environment.

Q. What is Deadlock? How to analyze and avoid deadlock situation?

Deadlock is a programming situation where two or more threads are blocked forever, this situation arises with at least two threads and two or more resources.

```java
class HelloClass {
    public synchronized void first(HiClass hi) {
        try {
            Thread.sleep(1000);
        }
        catch(InterruptedException ie) {}
        System.out.println(" HelloClass is calling  HiClass second() 
method");
        hi.second();
    }

    public synchronized void second() {
        System.out.println("I am inside second method of HelloClass");
    }
}

class HiClass {
    public synchronized void first(HelloClass he) {
        try {
            Thread.sleep(1000);
        }
        catch(InterruptedException ie){}
        System.out.println(" HiClass is calling HelloClass second()
method");
        he.second();
    }

    public synchronized void second() {
        System.out.println("I am inside second method of HiClass");
    }
}

class DeadlockClass extends Thread {
    HelloClass he = new HelloClass();
    HiClass hi = new HiClass();

    public void demo() {
        this.start();
        he.first(hi);
    }
    public void run() {
        hi.first(he);
    }

    public static void main(String[] args) {
        DeadlockClass dc = new DeadlockClass();
        dc.demo();
    }
}
```

Output

```
cmd> java DeadlockClass
HelloClass is calling HiClass second() method
HiClass is calling HelloClass second() method
```

Avoid deadlock

1. Avoid Nested Locks: This is the most common reason for deadlocks, avoid locking another resource if you already hold one. It’s almost impossible to get deadlock situation if you are working with only one object lock. For example, here is the another implementation of run() method without nested lock and program runs successfully without deadlock situation.

```
public void run() {
    String name = Thread.currentThread().getName();
    System.out.println(name + ' acquiring lock on ' + obj1);
    synchronized (obj1) {
        System.out.println(name + ' acquired lock on ' + obj1);
        work();
    }
    System.out.println(name + ' released lock on ' + obj1);
    System.out.println(name + ' acquiring lock on ' + obj2);
    synchronized (obj2) {
        System.out.println(name + ' acquired lock on ' + obj2);
        work();
    }
    System.out.println(name + ' released lock on ' + obj2);
    System.out.println(name + ' finished execution.');
}
```

2. Lock Only What is Required: You should acquire lock only on the resources you have to work on, for example in above program I am locking the complete Object resource but if we are only interested in one of it's fields, then we should lock only that specific field not complete object.

3. Avoid waiting indefinitely: You can get deadlock if two threads are waiting for each other to finish indefinitely using thread join. If your thread has to wait for another thread to finish, it's always best to use join with maximum time you want to wait for thread to finish.

<b><a href="#">↥ back to top</a></b>

Q. What is Thread Pool? How can we create Thread Pool in Java?

A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing. Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.

Java provides the Executor framework which is centered around the Executor interface, its sub-interface –**ExecutorService** and the class-**ThreadPoolExecutor**, which implements both of these interfaces. By using the executor, one only has to implement the Runnable objects and send them to the executor to execute.

To use thread pools, we first create a object of ExecutorService and pass a set of tasks to it. ThreadPoolExecutor class allows to set the core and maximum pool size.The runnables that are run by a particular thread are executed sequentially.

```
// Java program to illustrate
// ThreadPool
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

// Task class to be executed (Step 1)
class Task implements Runnable
{
    private String name;

    public Task(String s) {
        name = s;
    }

    // Prints task name and sleeps for 1s
    // This Whole process is repeated 5 times
    public void run() {
        try {
            for (int i = 0; i<=5; i++) {
                if (i == 0) {
                    Date d = new Date();
                    SimpleDateFormat ft = new
SimpleDateFormat("hh:mm:ss");
                    System.out.println("Initialization Time for"
                            + " task name - "+ name +" = "
+ft.format(d));
                    //prints the initialization time for every task
                }
                else {
                    Date d = new Date();
                    SimpleDateFormat ft = new
SimpleDateFormat("hh:mm:ss");
                    System.out.println("Executing Time for task name -
"+
                            name +" = " +ft.format(d));
                    // prints the execution time for every task
                }
                Thread.sleep(1000);
            }
            System.out.println(name+" complete");
        }
        catch(InterruptedException e) {
            e.printStackTrace();
        }
    }
}
public class Test
{
     // Maximum number of threads in thread pool
    static final int MAX_T = 3;

    public static void main(String[] args) {
        // creates five tasks
        Runnable r1 = new Task("task 1");
        Runnable r2 = new Task("task 2");
        Runnable r3 = new Task("task 3");
        Runnable r4 = new Task("task 4");
        Runnable r5 = new Task("task 5");

        // creates a thread pool with MAX_T no. of
        // threads as the fixed pool size(Step 2)
        ExecutorService pool = Executors.newFixedThreadPool(MAX_T);

        // passes the Task objects to the pool to execute (Step 3)
        pool.execute(r1);
        pool.execute(r2);
        pool.execute(r3);
        pool.execute(r4);
        pool.execute(r5);

        // pool shutdown ( Step 4)
        pool.shutdown();
    }
}
```

Output

```
Initialization Time for task name - task 2 = 02:32:56
Initialization Time for task name - task 1 = 02:32:56
Initialization Time for task name - task 3 = 02:32:56
Executing Time for task name - task 1 = 02:32:57
Executing Time for task name - task 2 = 02:32:57
Executing Time for task name - task 3 = 02:32:57
Executing Time for task name - task 1 = 02:32:58
Executing Time for task name - task 2 = 02:32:58
Executing Time for task name - task 3 = 02:32:58
Executing Time for task name - task 1 = 02:32:59
Executing Time for task name - task 2 = 02:32:59
Executing Time for task name - task 3 = 02:32:59
Executing Time for task name - task 1 = 02:33:00
Executing Time for task name - task 3 = 02:33:00
Executing Time for task name - task 2 = 02:33:00
Executing Time for task name - task 2 = 02:33:01
Executing Time for task name - task 1 = 02:33:01
Executing Time for task name - task 3 = 02:33:01
task 2 complete
task 1 complete
task 3 complete
Initialization Time for task name - task 5 = 02:33:02
Initialization Time for task name - task 4 = 02:33:02
Executing Time for task name - task 4 = 02:33:03
Executing Time for task name - task 5 = 02:33:03
Executing Time for task name - task 5 = 02:33:04
Executing Time for task name - task 4 = 02:33:04
Executing Time for task name - task 4 = 02:33:05
Executing Time for task name - task 5 = 02:33:05
Executing Time for task name - task 5 = 02:33:06
Executing Time for task name - task 4 = 02:33:06
Executing Time for task name - task 5 = 02:33:07
Executing Time for task name - task 4 = 02:33:07
task 5 complete
task 4 complete
```

Risks in using Thread Pools

- Deadlock
- Thread Leakage
- Resource Thrashing

↥ back to top

Q. Why wait(), notify() and notifyAll() must be called from inside of the synchronized block or method.?

`wait()` forces the thread to release its lock. This means that it must own the lock of an object before calling the `wait()` method of that (same) object. Hence the thread must be in one of the object's synchronized methods or synchronized block before calling wait().

When a thread invokes an object's `notify()` or `notifyAll()` method, one (an arbitrary thread) or all of the threads in its waiting queue are removed from the waiting queue to the entry queue. They then actively contend for the object's lock, and the one that gets the lock goes on to execute.

Q. What is the difference between wait() and sleep() method?

1. Class belongs: The wait() method belongs to `java.lang.Object` class, thus can be called on any Object. The sleep() method belongs to `java.lang.Thread` class, thus can be called on Threads.

2. Context: The wait() method can only be called from Synchronized context i.e. using synchronized block or synchronized method. The sleep() method can be called from any context.

3. Locking: The wait() method releases the lock on an object and gives others chance to execute. The sleep() method does not releases the lock of an object for specified time or until interrupt.

4. Wake up condition: A waiting thread can be awake by notify() or notifyAll() method. A sleeping can be awaked by interrupt or time expires.

5. Execution: Each object has each wait() method for inter-communication between threads. The sleep() method is static method belonging to Thread class. There is a common mistake to write t.sleep(1000) because sleep() is a class method and will pause the current running thread not t.

```
synchronized(LOCK) {
    Thread.sleep(1000); // LOCK is held
}


synchronized(LOCK) {
    LOCK.wait(); // LOCK is not held
}
```

Q. What is static synchronization?

Static synchronization is achieved by static synchronized methods. Static synchronized method locked on **class** and non-static synchronized method locked on **current object** i.e. static and non-static synchronized methods can run at same time. It can produce inconsistency problem.

If static synchronized method is called a class level lock is acquired and then if an object is tries to access non-static synchronized method at the same time it will not be accessible because class level lock is already acquired.

```
/**
 * This program is used to show the multithreading
 * example with synchronization using static synchronized method.
 * @author codesjava
 */
class PrintTable {
    public synchronized static void printTable(int n) {
       System.out.println("Table of " + n);
       for(int i = 1; i <= 10; i++) {
           System.out.println(n*i);
           try {
             Thread.sleep(500);
           } catch(Exception e) {
             System.out.println(e);
           }
        }
    }
}

class MyThread1 extends Thread {
    public void run() {
        PrintTable.printTable(2);
    }
}

class MyThread2 extends Thread {
    public void run() {
        PrintTable.printTable(5);
    }
}
```

```
public class MultiThreadExample {
    public static void main(String args[]) {

        //creating threads.
        MyThread1 t1 = new MyThread1();
        MyThread2 t2 = new MyThread2();

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

Output

```
Table of 2
2
4
6
8
10
12
14
16
18
20
Table of 5
5
10
15
20
25
30
35
40
45
50
```

```
<b><a href="#">↑ back to top</a></b>
```

Q. How is the safety of a thread achieved?

- Immutable objects are by default thread-safe because there state can not be modified once created. Since String is immutable in Java, its inherently thread-safe.
- Read only or final variables in Java are also thread-safe in Java.
- Locking is one way of achieving thread-safety in Java.
- Static variables if not synchronized properly becomes major cause of thread-safety issues.

- Example of thread-safe class in Java: Vector, Hashtable, ConcurrentHashMap, String etc.
- Atomic operations in Java are thread-safe e.g. reading a 32 bit int from memory because its an atomic operation it can't interleave with other thread.
- local variables are also thread-safe because each thread has there own copy and using local variables is good way to writing thread-safe code in Java.
- In order to avoid thread-safety issue minimize sharing of objects between multiple thread.
- Volatile keyword in Java can also be used to instruct thread not to cache variables and read from main memory and can also instruct JVM not to reorder or optimize code from threading perspective.

Q. What is difference between start() and run() method of thread class?

- When program calls `start()` method a **new Thread** is created and code inside `run()` method is executed in new Thread while if you call `run()` method directly **no new Thread is created** and code inside `run()` will execute on current Thread.

- `start()` Can't be invoked more than one time otherwise throws `java.lang.IllegalStateException` but `run()` can be invoked multiple times

```
class MyThread extends Thread
{
    @Override
    public void run() {
        System.out.println("I am executed by "
+currentThread().getName());
    }
}

public class ThreadExample
{
    public static void main(String[] args) {

        MyThread myThread = new MyThread();

        // Calling run() method directly
        myThread.run();

        // Calling start() method. It creates a new thread which
executes run() method
        myThread.start();
    }
}
```

Output

```
I am executed by main
I am executed by Thread-0
```

<b><a href="#">↥ back to top</a></b>

Q. Why we use Vector class?

Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index. They are very similar to ArrayList but Vector is **synchronised** and have some legacy method which collection framework does not contain. It extends AbstractList and implements List interfaces.

```java
import java.util.*;
class Vector_demo {

    public static void main(String[] arg) {

        // create default vector
        Vector v = new Vector();
        v.add(10);
        v.add(20);
        v.add("Numbers");
        System.out.println("Vector is " + v);
    }
}
```

<b><a href="#">↥ back to top</a></b>

Q. What is Thread Group? Why it's advised not to use it?

ThreadGroup creates a group of threads. It offers a convenient way to manage groups of threads as a unit. This is particularly valuable in situation in which you want to suspend and resume a number of related threads.

- The thread group form 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 not to access information about its thread group's parent thread group or any other thread group.

```java
// Java code illustrating Thread Group
import java.lang.*;
class NewThread extends Thread
{
    NewThread(String threadname, ThreadGroup tgob) {
        super(tgob, threadname);
        start();
    }
    public void run() {

        for (int i = 0; i < 1000; i++) {
            try {
                Thread.sleep(10);
            }
            catch (InterruptedException ex) {
                System.out.println("Exception encounterted");
            }
        }
    }
}
public class ThreadGroupExample
{
    public static void main(String arg[]) {
        // creating the thread group
        ThreadGroup tg = new ThreadGroup("Parent Thread Group");

        NewThread t1 = new NewThread("One", tg);
        System.out.println("Starting One");
        NewThread t2 = new NewThread("Two", tg);
        System.out.println("Starting two");

        // checking the number of active thread
        System.out.println("Number of active thread: "
                           + tg.activeCount());
    }
}
```

Q. How do you stop a thread in java?

A thread is automatically destroyed when the run() method has completed. But it might be required to kill/stop a thread before it has completed its life cycle. Modern ways to suspend/stop a thread are by using a boolean flag and `Thread.interrupt()` method.

```
class MyThread extends Thread
{
    //Initially setting the flag as true
    private volatile boolean flag = true;

    //This method will set flag as false
    public void stopRunning() {
        flag = false;
    }

    @Override
    public void run() {

        //This will make thread continue to run until flag becomes
false
        while (flag) {
            System.out.println("I am running....");
        }
        System.out.println("Stopped Running....");
    }
}

public class MainClass
{
    public static void main(String[] args) {

        MyThread thread = new MyThread();
        thread.start();
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //call stopRunning() method whenever you want to stop a thread
        thread.stopRunning();
    }
}
```

Output

```
I am running….
I am running….
I am running….
I am running….
I am running….
I am running….
I am running….
I am running….
Stopped Running….
```

<b><a href="#">↥ back to top</a></b>

Q. Can we call run() method of a Thread class?

No, you can not directly call run method to start a thread. You need to call start method to create a new thread. If you call run method directly, it won't create a new thread and it will be in same stack as main.

```
class CustomThread extends Thread {

 public void run() {
  for (int i = 0; i < 5; i++) {
    try {
       Thread.sleep(300);
    } catch (InterruptedException e) {
       e.printStackTrace();
    }
     System.out.println("Thread is running :"+i);
  }
 }
}
```

```
public class StartThreadAgainMain {

 public static void main(String[] args) {
   CustomThread ct1 = new CustomThread();
   CustomThread ct2 = new CustomThread();
   ct1.run();
   ct2.run();
 }
}
```

Output

```
Thread is running :0
Thread is running :1
Thread is running :2
Thread is running :3
Thread is running :4
Thread is running :0
Thread is running :1
Thread is running :2
Thread is running :3
Thread is running :4
```

<b><a href="#">↥ back to top</a></b>

Q. What is difference between Yield and Sleep method in Java?

1. Currently executing thread state: sleep() method causes the currently executing thread to sleep for the number of milliseconds specified in the argument. yield() method temporarily pauses the currently executing thread to give a chance to the remaining waiting threads of the same priority to execute. If there is no waiting thread or all the waiting threads of low priority then the current thread will continue its execution.

2. Interrupted Exception: Sleep method throws the Interrupted exception if another thread interrupts the sleeping thread. yield method does not throw Interrupted Exception.

3. Give up monitors: Thread.sleep() method does not cause cause currently executing thread to give up any monitors while yield() method give up the monitors.

Q. What is ThreadLocal?

The Java ThreadLocal class enables you to create variables that can only be read and written by the same thread. Thus, even if two threads are executing the same code, and the code has a reference to the same ThreadLocal variable, the two threads cannot see each other's ThreadLocal variables. Thus, the Java ThreadLocal class provides a simple way to make code thread safe that would not otherwise be so.

Creating a ThreadLocal

```
private ThreadLocal threadLocal = new ThreadLocal();
```

Set ThreadLocal Value

```
threadLocal.set("A thread local value");
```

Get ThreadLocal Value

```
String threadLocalValue = (String) threadLocal.get();
```

Remove ThreadLocal Value

```
threadLocal.remove();
```

Example

```
// Java code illustrating get() and set() method
public class ThreadLocalExample {

    public static void main(String[] args) {

        ThreadLocal<Number> tlObj = new ThreadLocal<Number>();

        // setting the value
        tlObj.set(100);
        System.out.println("value = " + tlObj.get());
    }
}
```

<b><a href="#">↥ back to top</a></b>

Q. What is Java Thread Dump, How can we get Java Thread dump of a Program?

A Java thread dump is a way of finding out what every thread in the JVM is doing at a particular point in time. This is especially useful if your Java application sometimes seems to hang when running under load, as an analysis of the dump will show where the threads are stuck.

You can generate a thread dump under Unix/Linux by running `kill -QUIT <pid>`, and under Windows by hitting `Ctl + Break`. Thread dump is the list of all the threads, every entry shows information about thread which includes following in the order of appearance.

Java Thread Dump Example

```
2019-12-26 22:28:39
Full thread dump Java HotSpot(TM) 64-Bit Server VM (23.5-b02 mixed
mode):

"Attach Listener" daemon prio=5 tid=0x00007fb7d8000000 nid=0x4207
waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"Timer-0" daemon prio=5 tid=0x00007fb7d4867000 nid=0x5503 waiting on
condition [0x00000001604d9000]
```

```
   java.lang.Thread.State: TIMED_WAITING (sleeping)
    at java.lang.Thread.sleep(Native Method)
    at
com.journaldev.threads.MyTimerTask.completeTask(MyTimerTask.java:19)
    at com.journaldev.threads.MyTimerTask.run(MyTimerTask.java:12)
    at java.util.TimerThread.mainLoop(Timer.java:555)
    at java.util.TimerThread.run(Timer.java:505)

"Service Thread" daemon prio=5 tid=0x00007fb7d482c000 nid=0x5303
runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread1" daemon prio=5 tid=0x00007fb7d482b800 nid=0x5203
waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread0" daemon prio=5 tid=0x00007fb7d4829800 nid=0x5103
waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" daemon prio=5 tid=0x00007fb7d4828800 nid=0x5003
runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"Finalizer" daemon prio=5 tid=0x00007fb7d4812000 nid=0x3f03 in
Object.wait() [0x000000015fd26000]
   java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    - waiting on <0x0000000140a25798> (a
java.lang.ref.ReferenceQueue$Lock)
    at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:135)
    - locked <0x0000000140a25798> (a
java.lang.ref.ReferenceQueue$Lock)
    at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:151)
    at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:177)

"Reference Handler" daemon prio=5 tid=0x00007fb7d4811800 nid=0x3e03 in
Object.wait() [0x000000015fc23000]
   java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    - waiting on <0x0000000140a25320> (a java.lang.ref.Reference$Lock)
    at java.lang.Object.wait(Object.java:503)
    at
java.lang.ref.Reference$ReferenceHandler.run(Reference.java:133)
    - locked <0x0000000140a25320> (a java.lang.ref.Reference$Lock)

"main" prio=5 tid=0x00007fb7d5000800 nid=0x1703 waiting on condition
[0x0000000106116000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
```

```
    at java.lang.Thread.sleep(Native Method)
    at com.journaldev.threads.MyTimerTask.main(MyTimerTask.java:33)

"VM Thread" prio=5 tid=0x00007fb7d480f000 nid=0x3d03 runnable
"GC task thread#0 (ParallelGC)" prio=5 tid=0x00007fb7d500d800
nid=0x3503 runnable
"GC task thread#1 (ParallelGC)" prio=5 tid=0x00007fb7d500e000
nid=0x3603 runnable
"GC task thread#2 (ParallelGC)" prio=5 tid=0x00007fb7d5800000
nid=0x3703 runnable
"GC task thread#3 (ParallelGC)" prio=5 tid=0x00007fb7d5801000
nid=0x3803 runnable
"GC task thread#4 (ParallelGC)" prio=5 tid=0x00007fb7d5801800
nid=0x3903 runnable
"GC task thread#5 (ParallelGC)" prio=5 tid=0x00007fb7d5802000
nid=0x3a03 runnable
"GC task thread#6 (ParallelGC)" prio=5 tid=0x00007fb7d5802800
nid=0x3b03 runnable
"GC task thread#7 (ParallelGC)" prio=5 tid=0x00007fb7d5803800
nid=0x3c03 runnable
"VM Periodic Task Thread" prio=5 tid=0x00007fb7d481e800 nid=0x5403
waiting on condition

JNI global references: 116
```

- **Thread Name**: Name of the Thread
- **Thread Priority**: Priority of the thread
- **Thread ID**: Represents the unique ID of the Thread
- **Thread Status**: Provides the current thread state, for example RUNNABLE, WAITING, BLOCKED. While analyzing deadlock look for the blocked threads and resources on which they are trying to acquire lock.
- **Thread callstack**: Provides the vital stack information for the thread. This is the place where we can see the locks obtained by Thread and if it’s waiting for any lock.

Tools

- jstack
- JVisualVM
- JMC
- ThreadMXBean
- APM Tool – App Dynamics
- JCMD
- VisualVM Profiler

↥ back to top

Q. What will happen if we don't override Thread class run() method?

If we don't override Thread class run() method in our defined thread then Thread class run() method will be executed and we will not get any output because Thread class run() is with an empty implementation.

It is highly recommended to override run() method because it improves the performance of the system. If we override run() method in the user-defined thread then in run() method we will define a job and Our created thread is responsible to execute run() method.

```
abstract class NotOverridableRunMethod extends Thread {
    abstract public void run();
}

class ParentMain {
    public static void main(String[] args) {
        OverrideRunMethod orn = new OverrideRunMethod();
        orn.start();
        System.out.println("Thread class run() method will be executed
with empty implementation");
    }
}
```

Output

```
cmd> java ParentMain
Thread class run() method will be executed with empty implementation
I am in run() method
```

Q. What is difference between the Thread class and Runnable interface for creating a Thread?

A thread can be defined in two ways. First, by extending a **Thread class** that has already implemented a Runnable interface. Second, by directly implementing a **Runnable interface**.

Difference

THREAD CLASS	RUNNABLE INTERFACE
Each thread creates a unique object and gets associated with it.	Multiple threads share the same objects.
As each thread create a unique object, more memory required.	As multiple threads share the same object less memory is used.
In Java, multiple inheritance not allowed hence, after a class extends Thread class, it can not extend any other class.	If a class define thread implementing the Runnable interface it has a chance of extending one class.
A user must extend thread class	If you only want to specialize run method then

THREAD CLASS	RUNNABLE INTERFACE
only if it wants to override the other methods in Thread class.	implementing Runnable is a better option.
Extending Thread class introduces tight coupling as the class contains code of Thread class and also the job assigned to the thread	Implementing Runnable interface introduces loose coupling as the code of Thread is separate form the job of Threads.

Q. What does join() method?

`java.lang.Thread` class provides the join() method which allows one thread to wait until another thread completes its execution.

```
public class ThreadJoinExample {

    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable(), "t1");
        Thread t2 = new Thread(new MyRunnable(), "t2");
        Thread t3 = new Thread(new MyRunnable(), "t3");

        t1.start();

        //start second thread after waiting for 2 seconds or if it's
dead
        try {
            t1.join(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        t2.start();

        //start third thread only when first thread is dead
        try {
            t1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        t3.start();

        //let all threads finish execution before finishing main
thread
        try {
            t1.join();
            t2.join();
            t3.join();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        System.out.println("All threads are dead, exiting main
thread");
    }

}

class MyRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println("Thread started:
"+Thread.currentThread().getName());
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Thread ended:
"+Thread.currentThread().getName());
    }
}
```

Output

```
Thread started: t1
Thread started: t2
Thread ended: t1
Thread started: t3
Thread ended: t2
Thread ended: t3
All threads are dead, exiting main thread
```

<b><a href="#">↥ back to top</a></b>

Q. What is race-condition?

Race condition in Java occurs in a multi-threaded environment **when more than one thread try to access a shared resource** (modify, write) at the same time. Since multiple threads try to race each other to finish executing a method thus the name **race condition**.

Example of race condition in Java

```
class Counter  implements Runnable{
    private int c = 0;

    public void increment() {
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        c++;
    }

    public void decrement() {
        c--;
    }

    public int getValue() {
        return c;
    }

    @Override
    public void run() {
        //incrementing
        this.increment();
        System.out.println("Value for Thread After increment "
        + Thread.currentThread().getName() + " " + this.getValue());
        //decrementing
        this.decrement();
        System.out.println("Value for Thread at last "
        + Thread.currentThread().getName() + " " + this.getValue());

    }
}

public class RaceConditionExample {
    public static void main(String[] args) {
        Counter counter = new Counter();
        Thread t1 = new Thread(counter, "Thread-1");
        Thread t2 = new Thread(counter, "Thread-2");
        Thread t3 = new Thread(counter, "Thread-3");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output

```
Value for Thread After increment Thread-2 3
Value for Thread at last Thread-2 2
Value for Thread After increment Thread-1 2
Value for Thread at last Thread-1 1
Value for Thread After increment Thread-3 1
Value for Thread at last Thread-3 0
```

Using synchronization to avoid race condition in Java

To fix the race condition we need to have a way to restrict resource access to only one thread at a time. We have to use `synchronized` keyword to synchronize the access to the shared resource.

```java
//This class' shared object will be accessed by threads
class Counter  implements Runnable{
    private int c = 0;

    public  void increment() {
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        c++;
    }

    public  void decrement() {
        c--;
    }

    public  int getValue() {
        return c;
    }

    @Override
    public void run() {
        synchronized(this) {
            // incrementing
            this.increment();
            System.out.println("Value for Thread After increment "
             + Thread.currentThread().getName() + " " +
this.getValue());
            //decrementing
            this.decrement();
            System.out.println("Value for Thread at last " +
Thread.currentThread().getName()
                + " " + this.getValue());
        }
    }
}

public class RaceConditionExample {
    public static void main(String[] args) {
        Counter counter = new Counter();
        Thread t1 = new Thread(counter, "Thread-1");
        Thread t2 = new Thread(counter, "Thread-2");
        Thread t3 = new Thread(counter, "Thread-3");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output

```
Value for Thread After increment Thread-2 1
Value for Thread at last Thread-2 0
Value for Thread After increment Thread-3 1
Value for Thread at last Thread-3 0
Value for Thread After increment Thread-1 1
Value for Thread at last Thread-1 0
```

<b><a href="#">↥ back to top</a></b>

Q. What is Lock interface in Java Concurrency API? What is the Difference between ReentrantLock and Synchronized?

A `java.util.concurrent.locks.Lock` is a thread synchronization mechanism just like synchronized blocks. A Lock is, however, more flexible and more sophisticated than a synchronized block. Since Lock is an interface, you need to use one of its implementations to use a Lock in your applications. `ReentrantLock` is one such implementation of Lock interface.

```
Lock lock = new ReentrantLock();

lock.lock();

//critical section
lock.unlock();
```

Difference between Lock Interface and synchronized keyword

- Having a timeout trying to get access to a `synchronized` block is not possible. Using `Lock.tryLock(long timeout, TimeUnit timeUnit)`, it is possible.
- The `synchronized` block must be fully contained within a single method. A Lock can have it's calls to `lock()` and `unlock()` in separate methods.

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ConcurrencyLockExample implements Runnable {

    private Resource resource;
    private Lock lock;

    public ConcurrencyLockExample(Resource r) {
        this.resource = r;
        this.lock = new ReentrantLock();
    }

    @Override
    public void run() {
        try {
            if(lock.tryLock(10, TimeUnit.SECONDS)) {
              resource.doSomething();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            //release lock
            lock.unlock();
        }
        resource.doLogging();
    }
}
```

Q. What is the difference between the Runnable and Callable interface?

Runnable and Callable interface both are used in the multithreading environment. Runnable is the core interface provided for representing multi-threaded tasks and Callable is an improved version of Runnable that was added in Java 1.5.

Difference between Callable and Runnable in Java

1. Checked Exception: Callable's call() method can throw checked exception while Runnable run() method can not throw checked exception.

2. Return value: Return type of Runnable run() method is void , so it can not return any value. while Callable can return the Future object, which represents the life cycle of a task and provides methods to check if the task has been completed or canceled.

3. Implementation: Callable needs to implement call() method while Runnable needs to implement run() method.

4. Execution: Limitation of Callable interface lies in java is that one can not pass it to Thread as one pass the Runnable instance. There is no constructor defined in the Thread class which accepts a Callable interface.

Example: Callable Interface

```
// Java program to illustrate Callable and FutureTask
// for random number generation
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.FutureTask;

class CallableExample implements Callable
{
  public Object call() throws Exception {
    Random generator = new Random();
    Integer randomNumber = generator.nextInt(5);
    Thread.sleep(randomNumber * 1000);
    return randomNumber;
  }
}

public class CallableFutureTest
{
  public static void main(String[] args) throws Exception {

    // FutureTask is a concrete class that
    // implements both Runnable and Future
    FutureTask[] randomNumberTasks = new FutureTask[5];

    for (int i = 0; i < 5; i++) {
      Callable callable = new CallableExample();

      // Create the FutureTask with Callable
      randomNumberTasks[i] = new FutureTask(callable);

      // As it implements Runnable, create Thread
      // with FutureTask
      Thread t = new Thread(randomNumberTasks[i]);
      t.start();
    }

    for (int i = 0; i < 5; i++) {
      // As it implements Future, we can call get()
      System.out.println(randomNumberTasks[i].get());
    }
  }
}
```

Output

```
4
2
3
3
0
```

Example: Runnable Interface

```
// Java program to illustrate Runnable
// for random number generation
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.FutureTask;
```

```
class RunnableExample implements Runnable
{
    // Shared object to store result
    private Object result = null;

    public void run() {
        Random generator = new Random();
        Integer randomNumber = generator.nextInt(5);

        // As run cannot throw any Exception
        try {
            Thread.sleep(randomNumber * 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Store the return value in result when done
        result = randomNumber;

        // Wake up threads blocked on the get() method
        synchronized(this) {
            notifyAll();
        }
    }

    public synchronized Object get() throws InterruptedException  {
        while (result == null)
            wait();

        return result;
    }
}

public class RunnableTest
{
    public static void main(String[] args) throws Exception {
        RunnableExample[] randomNumberTasks = new RunnableExample[5];

        for (int i = 0; i < 5; i++) {
            randomNumberTasks[i] = new RunnableExample();
            Thread t = new Thread(randomNumberTasks[i]);
            t.start();
        }

        for (int i = 0; i < 5; i++)
            System.out.println(randomNumberTasks[i].get());
    }
}
```

Output

```
0
4
3
1
4
```

```
<b><a href="#">↥ back to top</a></b>
```

Q. What is the Thread's interrupt flag? How does it relate to the InterruptedException?

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true.

Example: **Interrupting a thread that stops working**

```java
// Java Program to illustrate the concept of interrupt() method
// while a thread stops working
class ThreadsInterruptExample extends Thread {

    public void run() {
        try {
            Thread.sleep(1000);
            System.out.println("Task");
        } catch (InterruptedException e) {
            throw new RuntimeException("Thread interrupted");
        }
    }
    public static void main(String args[]) {
        ThreadsInterruptExample t1 = new ThreadsInterruptExample();
        t1.start();
        try {
            t1.interrupt();
        }
        catch (Exception e) {
            System.out.println("Exception handled");
        }
    }
}
```

Output

```
Exception in thread "Thread-0" java.lang.RuntimeException: Thread
interrupted
```

Q. What is Java Memory Model (JMM)? Describe its purpose and basic ideas.

The Java memory model used internally in the JVM divides memory between thread stacks and the heap. Each thread running in the Java virtual machine has its own thread stack. The thread stack contains information about what methods the thread has called to reach the current point of execution.

The thread stack also contains all local variables for each method being executed (all methods on the call stack). A thread can only access it's own thread stack. Local variables created by a thread are invisible to all other threads than the thread who created it. Even if two threads are executing the exact same code, the two threads will still create the local variables of that code in each their own thread stack. Thus, each thread has its own version of each local variable.

All local variables of primitive types (boolean, byte, short, char, int, long, float, double) are fully stored on the thread stack and are thus not visible to other threads. One thread may pass a copy of a pritimive variable to another thread, but it cannot share the primitive local variable itself.

The heap contains all objects created in your Java application, regardless of what thread created the object. This includes the object versions of the primitive types (e.g. Byte, Integer, Long etc.). It does not matter if an object was created and assigned to a local variable, or created as a member variable of another object, the object is still stored on the heap.

```
<b><a href="#">↥ back to top</a></b>
```

Q. Describe the conditions of livelock and starvation?

Livelock occurs when two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work. These processes are not in the waiting state, and they are running concurrently. This is different from a deadlock because in a deadlock all processes are in the waiting state.

```
var l1 = .... // lock object like semaphore or mutex etc
var l2 = .... // lock object like semaphore or mutex etc

    // Thread1
    Thread.Start( ()=> {

    while (true) {

        if (!l1.Lock(1000)) {
            continue;
        }
        if (!l2.Lock(1000)) {
            continue;
        }

        // do some work
    });

    // Thread2
    Thread.Start( ()=> {

      while (true) {

        if (!l2.Lock(1000)) {
            continue;
        }
        if (!l1.Lock(1000)) {
            continue;
        }
        // do some work
    });
```

starvation describes a situation where a greedy thread holds a resource for a long time so other threads are blocked forever. The blocked threads are waiting to acquire the resource but they never get a chance. Thus they starve to death.

Starvation can occur due to the following reasons:

- Threads are blocked infinitely because a thread takes long time to execute some synchronized code (e.g. heavy I/O operations or infinite loop).
- A thread doesn't get CPU's time for execution because it has low priority as compared to other threads which have higher priority.
- Threads are waiting on a resource forever but they remain waiting forever because other threads are constantly notified instead of the hungry ones.

`<b><a href="#">↥ back to top</a></b>`

Q. How do I share a variable between 2 Java threads?

We should declare such variables as static and volatile.

Volatile variables are shared across multiple threads. This means that individual threads won't cache its copy in the thread local. But every object would have its own copy of the variable so threads may cache value locally.

We know that static fields are shared across all the objects of the class, and it belongs to the class and not the individual objects. But, for static and non-volatile variable also, threads may cache the variable locally.

Q. What are the main components of concurrency API?

The concurrency API are designed and optimized specifically for synchronized multithreaded access. They are grouped under the `java.util.concurrent` package.

Main Components

- Executor
- ExecutorService
- ScheduledExecutorService
- Future
- CountDownLatch
- CyclicBarrier
- Semaphore
- ThreadFactory
- BlockingQueue
- DelayQueue
- Locks
- Phaser

Q. What is difference between CyclicBarrier and CountDownLatch in Java?

Both CyclicBarrier and CountDownLatch are used to implement a scenario where one Thread waits for one or more Thread to complete their job before starts processing. The differences are:

1. Definition: CountDownLatch is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

CyclicBarrier is a synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.

2. Reusable: A CountDownLatch is initialized with given count. count reaches zero by calling of countDown() method. The count can not be reset.

CyclicBarrier is used to reset count. The barrier is called cyclic because it can be reused after the waiting threads are released (or count become zero).

Q. What is Semaphore in Java concurrency?

A Semaphore is a thread synchronization construct that can be used either to send signals between threads to avoid missed signals, or to guard a critical section like you would with a lock. Java 5 comes with semaphore implementations in the java.util.concurrent package.

1. Simple Semaphore:

```
public class Semaphore {
  private boolean signal = false;

  public synchronized void take() {
    this.signal = true;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(!this.signal) wait();
    this.signal = false;
  }
}
```

The take() method sends a signal which is stored internally in the Semaphore. The release() method waits for a signal. When received the signal flag is cleared again, and the release() method exited.

2. Counting Semaphore:

```
public class CountingSemaphore {
  private int signals = 0;

  public synchronized void take() {
    this.signals++;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(this.signals == 0) wait();
    this.signals--;
  }
}
```

3. Bounded Semaphore

```
public class BoundedSemaphore {
  private int signals = 0;
  private int bound   = 0;

  public BoundedSemaphore(int upperBound){
    this.bound = upperBound;
  }

  public synchronized void take() throws InterruptedException{
    while(this.signals == bound) wait();
    this.signals++;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(this.signals == 0) wait();
    this.signals--;
    this.notify();
  }
}
```

4. Using Semaphores as Locks

```
BoundedSemaphore semaphore = new BoundedSemaphore(1);

...

semaphore.take();

try{
  //critical section
} finally {
  semaphore.release();
}
```

<b><a href="#">↥ back to top</a></b>

Q. What is Callable and Future in Java concurrency?

Future and FutureTask in Java allows to write asynchronous code. A Future interface provides methods **to check if the computation is complete, to wait for its completion and to retrieve the results of the computation**. The result is retrieved using Future's get() method when the computation has completed, and it blocks until it is completed. We need a callable object to create a future task and then we can use Java Thread Pool Executor to process these asynchronously.

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.logging.Level;
import java.util.logging.Logger;
/**
* Java program to show how to use Future in Java. Future allows to
write
* asynchronous code in Java, where Future promises result to be
available in
* future
**/
public class FutureExample {

    private static final ExecutorService threadpool =
Executors.newFixedThreadPool(3);

    public static void main(String args[]) throws
InterruptedException, ExecutionException {

        FactorialCalculator task = new FactorialCalculator(10);
        System.out.println("Submitting Task ...");

        Future future = threadpool.submit(task);
        System.out.println("Task is submitted");

        while (!future.isDone()) {
            System.out.println("Task is not completed yet....");
            Thread.sleep(1); //sleep for 1 millisecond before checking
again
        }
        System.out.println("Task is completed, let's check result");
        long factorial = future.get();
        System.out.println("Factorial of 1000000 is : " + factorial);

        threadpool.shutdown();
    }
    private static class FactorialCalculator implements Callable {

        private final int number;

        public FactorialCalculator(int number) {
            this.number = number;
        }
        @Override public Long call() {
            long output = 0;
            try {
                output = factorial(number);
            } catch (InterruptedException ex) {

Logger.getLogger(Test.class.getName()).log(Level.SEVERE, null, ex);
            }
            return output;
        }
        private long factorial(int number) throws InterruptedException
{
            if (number < 0) {
                throw new IllegalArgumentException("Number must be
greater than zero");
            }
            long result = 1;
            while (number > 0) {
                Thread.sleep(1); // adding delay for example
                result = result * number;
                number--;
            }
            return result;
        }
    }
}
```

Output

```
Submitting Task ...
Task is submitted Task is not completed yet....
Task is not completed yet....
Task is not completed yet....
Task is completed, let's check result
Factorial of 1000000 is : 3628800
```

```
<b><a href="#">↥ back to top</a></b>
```

Q. What is blocking method in Java?

Blocking methods in Java are those methods which block the executing thread until their operation finished. Example of blocking method is `InputStream read()` method which blocks until all data from InputStream has been read completely.

```java
public class BlcokingCallTest {

    public static void main(String args[]) throws
FileNotFoundException, IOException  {
       System.out.println("Calling blocking method in Java");
       int input = System.in.read();
       System.out.println("Blocking method is finished");
    }
}
```

Examples of blocking methods in Java:

- **InputStream.read()** which blocks until input data is available, an exception is thrown or end of Stream is detected.
- **ServerSocket.accept()** which listens for incoming socket connection in Java and blocks until a connection is made.
- **InvokeAndWait()** wait until code is executed from Event Dispatcher thread.

Q. What is atomic variable in Java?

Atomic variables allow us to perform atomic operations on the variables. The most commonly used atomic variable classes in Java are `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, and `AtomicReference`. These classes represent an int, long, boolean and object reference respectively which can be atomically updated. The main methods exposed by these classes are:

- `get()`: gets the value from the memory, so that changes made by other threads are visible; equivalent to reading a volatile variable
- `set()`: writes the value to memory, so that the change is visible to other threads; equivalent to writing a volatile variable
- `lazySet()`: eventually writes the value to memory, may be reordered with subsequent relevant memory operations. One use case is nullifying references, for the sake of garbage collection, which is never going to be accessed again. In this case, better performance is achieved by delaying the null volatile write
- `compareAndSet()`: same as described in section 3, returns true when it succeeds, else false

- `weakCompareAndSet()`: same as described in section 3, but weaker in the sense, that it does not create happens-before orderings. This means that it may not necessarily see updates made to other variables

```
public class SafeCounterWithoutLock {

    private final AtomicInteger counter = new AtomicInteger(0);

    public int getValue() {
        return counter.get();
    }
    public void increment() {
        while(true) {
            int existingValue = getValue();
            int newValue = existingValue + 1;
            if(counter.compareAndSet(existingValue, newValue)) {
                return;
            }
        }
    }
}
```

```
<b><a href="#">↥ back to top</a></b>
```

Q. What is Executors Framework?

The Executor framework helps to **decouple a command submission from command execution**. In the java.util.concurrent package there are three interfaces:

1. Executor — Used to submit a new task.

2. ExecutorService — A subinterface of Executor that adds methods to manage lifecycle of threads used to run the submitted tasks and methods to produce a Future to get a result from an asynchronous computation.

3. ScheduledExecutorService — A subinterface of ExecutorService, to execute commands periodically or after a given delay.

Example: **Java ExecutorService**

```
ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

executorService.shutdown();
```

First an ExecutorService is created using the Executors newFixedThreadPool() factory method. This creates a thread pool with 10 threads executing tasks.

Second, an anonymous implementation of the Runnable interface is passed to the execute() method. This causes the Runnable to be executed by one of the threads in the ExecutorService.

`<b><a href="#">↥ back to top</a></b>`

Q. What are the available implementations of ExecutorService in the standard library?

The ExecutorService interface has three standard implementations:

- **ThreadPoolExecutor**: for executing tasks using a pool of threads. Once a thread is finished executing the task, it goes back into the pool. If all threads in the pool are busy, then the task has to wait for its turn.
- **ScheduledThreadPoolExecutor**: allows to schedule task execution instead of running it immediately when a thread is available. It can also schedule tasks with fixed rate or fixed delay.
- **ForkJoinPool**: is a special ExecutorService for dealing with recursive algorithms tasks. If you use a regular ThreadPoolExecutor for a recursive algorithm, you will quickly find all your threads are busy waiting for the lower levels of recursion to finish. The ForkJoinPool implements the so-called work-stealing algorithm that allows it to use available threads more efficiently.

Q. What kind of thread is the Garbage collector thread?

- Daemon thread ## Q. How can we pause the execution of a Thread for specific time?
- **Thread.sleep(...)**: causes the currently executing thread to sleep (cease execution) for the specified number of milliseconds plus the specified number of nanoseconds, subject to the precision and accuracy of system timers and schedulers. The thread does not lose ownership of any monitors;
- **Thread.yield()**: causes the currently executing thread object to temporarily pause and allow other threads to execute;
- **Monitors**: you call wait on the object you want to lock and you release the lock by calling notify on the same object.

Q. What is difference between Executor.submit() and Executer.execute() method?

execute():

- Takes Runnable object as parameter.
- Returns void.
- Useful when you want to execute a task asynchronously in thread pool but doesn't bother about it's result.

- Example: Delegating a request (for which no response required) to another service, sending an email.

submit():

- Takes Runnable or Callable object as parameter.
- Returns Future object.
- Useful when the calling thread needs the output from the task executed. Using Future object, you can get result, check whether the task is completed without failure or can request cancelling the task before its completion.
- Example: Parallel stream search in Java 8.

```
<b><a href="#">↥ back to top</a></b>
```

Q. What is Phaser in Java concurrency?

The Phaser allows us to build logic in which **threads need to wait on the barrier before going to the next step of execution**. Phaser is similar to other synchronization barrier utils like CountDownLatch and CyclicBarrier.

CountDownLatch vs CyclicBarrier vs Phaser

1. CountDownLatch:

- Created with a fixed number of threads
- Cannot be reset
- Allows threads to wait(CountDownLatch#await()) or continue with its execution(CountDownLatch#countDown()).

2. CyclicBarrier:

- Can be reset.
- Does not a provide a method for the threads to advance. The threads have to wait till all the threads arrive.
- Created with fixed number of threads.

3. Phaser:

- Number of threads need not be known at Phaser creation time. They can be added dynamically.
- Can be reset and hence is, reusable.
- Allows threads to wait(Phaser#arriveAndAwaitAdvance()) or continue with its execution(Phaser#arrive()).
- Supports multiple Phases(hence the name phaser).

PhaserExample.java

```java
import java.util.concurrent.Phaser;

public class PhaserExample
```

```
{
  public static void main(String[] args) throws InterruptedException
  {
     Phaser phaser = new Phaser();
     phaser.register();//register self... phaser waiting for 1
party (thread)
     int phasecount = phaser.getPhase();

     System.out.println("Phasecount is "+phasecount);
     new PhaserExample().testPhaser(phaser,2000);//phaser waiting
for 2 parties
     new PhaserExample().testPhaser(phaser,4000);//phaser waiting
for 3 parties
     new PhaserExample().testPhaser(phaser,6000);//phaser waiting
for 4 parties
     //now that all threads are initiated, we will de-register
main thread
     //so that the barrier condition of 3 thread arrival is meet.
     phaser.arriveAndDeregister();
         Thread.sleep(10000);
         phasecount = phaser.getPhase();
     System.out.println("Phasecount is "+phasecount);

  }

  private void testPhaser(final Phaser phaser,final int sleepTime) {
    phaser.register();
    new Thread() {
      @Override
      public void run() {
            try {
System.out.println(Thread.currentThread().getName()+" arrived");
              phaser.arriveAndAwaitAdvance();//threads
register arrival to the phaser.
              Thread.sleep(sleepTime);
            } catch (InterruptedException e) {
              e.printStackTrace();
            }
System.out.println(Thread.currentThread().getName()+" after passing
barrier");
        }
      }.start();
  }
}
```

Output

```
Phasecount is 0
Thread-0 arrived
```

```
Thread-2 arrived
Thread-1 arrived
Thread-0 after passing barrier
Thread-1 after passing barrier
Thread-2 after passing barrier
Phasecount is 1
```

```
<b><a href="#">↥ back to top</a></b>
```

Q. How to stop a Thread in Java?

A thread is automatically destroyed when the run() method has completed. But it might be required to kill/stop a thread before it has completed its life cycle. Modern ways to suspend/stop a thread are by using a **boolean flag** and **Thread.interrupt()** method.

Example: Stop a thread Using a boolean variable

```
/**
* Java program to illustrate
* stopping a thread using boolean flag
*
**/
class MyThread extends Thread {

    // Initially setting the flag as true
    private volatile boolean flag = true;

    // This method will set flag as false
    public void stopRunning() {
        flag = false;
    }

    @Override
    public void run() {

        // This will make thread continue to run until flag becomes
false
        while (flag) {
            System.out.println("I am running....");
        }
        System.out.println("Stopped Running....");
    }
}

public class MainClass {

    public static void main(String[] args) {

        MyThread thread = new MyThread();
        thread.start();
        try {
            Thread.sleep(100);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }

        // call stopRunning() method whenever you want to stop a
thread
        thread.stopRunning();
    }
}
```

Output:

```
I am running….
I am running….
I am running….
I am running….
I am running….
Stopped Running….
```

Example: Stop a thread Using interrupt() Method

```
/**
* Java program to illustrate
* stopping a thread using interrupt() method
*
**/
class MyThread extends Thread {

    @Override
    public void run() {

        while (!Thread.interrupted()) {
            System.out.println("I am running....");
        }
        System.out.println("Stopped Running.....");
    }
}

public class MainClass {

    public static void main(String[] args) {

        MyThread thread = new MyThread();
        thread.start();

        try {
            Thread.sleep(100);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }

        // interrupting the thread
        thread.interrupt();
    }
}
```

Output:

```
I am running….
I am running….
I am running….
I am running….
I am running….
Stopped Running….
```

<b><a href="#">↥ back to top</a></b>

Q. Why implementing Runnable is better than extending thread?

Features	implements Runnable	extends Thread
Inheritance option	extends any java class	No
Reusability	Yes	No
Object Oriented Design	Good,allows composition	Bad
Loosely Coupled	Yes	No
Function Overhead	No	Yes

Example:

```
/**
* Java program to illustrate defining Thread
* by extending Thread class
*
**/
// Here we cant extends any other class
class ThreadExample extends Thread
{
    public void run() {
        System.out.println("Run method executed by child Thread");
    }
    public static void main(String[] args) {
        ThreadExample obj = new ThreadExample();
        obj.start();
        System.out.println("Main method executed by main thread");
    }
}
```

Output

```
Main method executed by main thread
Run method executed by child Thread
```

```java
/**
* Java program to illustrate defining Thread
* by implements Runnable interface
*
**/
class RunnableExample
{
    public static void m1() {
        System.out.println("Runnable interface Example");
    }
}

// Here we can extends any other class
class Test extends RunnableExample implements Runnable
{
    public void run() {
        System.out.println("Run method executed by child Thread");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.m1();
        Thread t1 = new Thread(t);
        t1.start();
        System.out.println("Main method executed by main thread");
    }
}
```

Output

```
Runnable interface Example
Main method executed by main thread
Run method executed by child Thread
```

```
<b><a href="#">↥ back to top</a></b>
```

Q. Tell me about join() and wait() methods?

The `wait()` and `join()` methods are used to pause the current thread. The `wait()` is used in with `notify()` and `notifyAll()` methods, but `join()` is used in Java to wait until one thread finishes its execution.

The `wait()` is mainly used for shared resources, a thread notifies other waiting thread when a resource becomes free. On the other hand `join()` is used for waiting a thread to die.

Q. How to implement thread-safe code without using the synchronized keyword?

- **Atomic updates**: A technique in which you call atomic instructions like compare and set provided by the CPU
- **java.util.concurrent.locks.ReentrantLock**: A lock implementation that provides more flexibility than synchronized blocks
- **java.util.concurrent.locks.ReentrantReadWriteLock**: A lock implementation in which reads do not block reads
- **java.util.concurrent.locks.StampedLock** a nonreeantrant Read-Write lock with the possibility of optimistically reading values.
- **java.lang.ThreadLocal**: No need for synchronization if the mutable state is confined to a single thread. This can be done by using local variables or `java.lang.ThreadLocal`.

Q. What is difference between ArrayBlockingQueue & LinkedBlockingQueue in Java Concurrency?

Q. What is PriorityBlockingQueue in Java Concurrency?

Q. What is DelayQueue in Java Concurrency?

Q. What is SynchronousQueue in Java?

Q. What is Exchanger in Java concurrency?

Q. What is Busy Spinning? Why will you use Busy Spinning as wait strategy?

```
<b><a href="#">↥ back to top</a></b>
```

https://www.youtube.com/@codewitharrays

https://www.instagram.com/codewitharrays/

https://t.me/codewitharrays Group Link: https://t.me/ccee2025notes

+91 8007592194 +91 9284926333

codewitharrays@gmail.com

https://codewitharrays.in/project