Daemon Thread in Java

- **It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.**
- **Its life depends on user threads.**
- **It is a low priority thread.**

No.	Method	Description
1)	public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
2)	public boolean isDaemon()	is used to check that current is daemon.

LAB Thread:

```
public class DaemonThreadExample1 extends Thread{

   public void run(){

           // Checking whether the thread is Daemon or not
           if(Thread.currentThread().isDaemon()){
               System.out.println("Daemon thread executing");
           }
           else{
               System.out.println("user(normal) thread executing");
           }
   }
   public static void main(String[] args){
          /* Creating two threads: by default they are
           * user threads (non-daemon threads)
           */
          DaemonThreadExample1 t1=new DaemonThreadExample1();
          DaemonThreadExample1 t2=new DaemonThreadExample1();

          //Making user thread t1 to Daemon
        t1.setDaemon(true);

        //starting both the threads
        t1.start();
        t2.start();
   }
}
```

Output:

daemon thread work
user thread work
user thread work

Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

```
public class DaemonThreadEx2 extends Thread {

   public void run(){
          System.out.println("Thread is running");
          System.out.println("Name: "+Thread.currentThread().getName());
          System.out.println("Daemon: "+Thread.currentThread().isDaemon());

   }

    public static void main(String[] args){
         DaemonThreadEx2 t1=new DaemonThreadEx2();
         t1.start();
         // It will throw IllegalThreadStateException
         t1.setDaemon(true);
    }
}
```

Lab-7a // LifeCycleState

```
public class LifeCycleState extends Thread {
    public void run () {
        System.out.println("run method");
    }

    public static void main ( String[] args ) {
        LifeCycleState p1 = new LifeCycleState();
        LifeCycleState p2 = new LifeCycleState();
        System.out.println("p1 State : "+p1.getState());
        System.out.println("p2 State : "+p2.getState());
        p1.start();
        System.out.println("p1 State : "+p1.getState());
        System.out.println("p2 State : "+p2.getState());
        p2.start();
        System.out.println("p1 State : "+p1.getState());
        System.out.println("p2 State : "+p2.getState());
    }
}
```

Output:

p1 State : NEW
p2 State : NEW
p1 State : RUNNABLE

p2 State : NEW
p1 State : RUNNABLE
p2 State : RUNNABLE
run method
run method

Lab-7b // CheckPriority

```
class CheckPriority extends Thread {
    public void run () {
        System.out.println(Thread.currentThread().getPriority());
    }
    public static void main ( String[] args ) {

        CheckPriority tr1 = new CheckPriority();
        CheckPriority tr2 = new CheckPriority();
        tr1.start();
        tr2.start();
    }
}
```

Output:

```
5
5
```

Lab-7c // CheckPriority

```
class SetCheckPriority extends Thread {
    public void run () {

        String threadName = Thread.currentThread().getName();
        int threadPriority = Thread.currentThread().getPriority();;
        System.out.println("Thread Name is " + threadName + "and Thread priority is "
+threadPriority );

    }
    public static void main ( String[] args ) {

        SetCheckPriority tr0 = new SetCheckPriority();
        SetCheckPriority tr1 = new SetCheckPriority();
        SetCheckPriority tr2 = new SetCheckPriority();

        tr0.setPriority(Thread.MIN_PRIORITY);
        tr1.setPriority(Thread.NORM_PRIORITY);
        tr2.setPriority(Thread.MAX_PRIORITY);

        tr0.start();
        tr1.start();
        tr2.start();

    }
}
```

Output:

Thread Name is Thread-2and Thread priority is 10
Thread Name is Thread-1and Thread priority is 5
Thread Name is Thread-0and Thread priority is 1

Synchronization is the process of allowing threads to execute one after another. It provides the control to access multiple threads to shared resources. When we want to allow one thread to access the shared resource, then synchronization is the better option.

1. Synchronization method

```
Lab-7d  // SynchronizedDemo

class Display {
        // Remove synchronized
    public synchronized void wish ( String name ) {
        for (int i = 0; i < 5; i++) {
            System.out.println("good morning:" + name);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }
}

class MyThread extends Thread {
    Display d;
    String name;

    MyThread ( Display d, String name ) {
        this.d = d;
        this.name = name;
    }
    public void run () {
        d.wish(name);
    }
}

class SynchronizedDemo {
    public static void main ( String[] args ) {
        Display d1 = new Display();
        MyThread t1 = new MyThread(d1, "Rajesh");
        MyThread t2 = new MyThread(d1, "Yuvaraj");
        t1.start();
        t2.start();
    }
}
```

Output Remove synchronized:	Output ADD synchronized:

good morning:Rajesh good morning:Yuvaraj good morning:Rajesh good morning:Yuvaraj good morning:Yuvaraj good morning:Rajesh good morning:Yuvaraj good morning:Rajesh good morning:Yuvaraj good morning:Rajesh	good morning:Rajesh good morning:Rajesh good morning:Rajesh good morning:Rajesh good morning:Rajesh good morning:Yuvaraj good morning:Yuvaraj good morning:Yuvaraj good morning:Yuvaraj good morning:Yuvaraj

2. SynchronizedBlock

Lab-7e // SynchronizedBlock

```
class Table1 {
    public void print ( int n ) {
        synchronized (this) {
            for (int i = 1; i <= 10; i++) {
                System.out.println(i * n);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }

            }
        }
    }
}

class thread1 extends Thread {
    Table1 t;

    thread1 ( Table1 t ) {
        this.t = t;
    }

    public void run () {
        t.print(2);
    }
}

class thread2 extends Thread {
    Table1 t;

    thread2 ( Table1 t ) {
        this.t = t;
    }
```

```
    public void run () {
        t.print(3);
    }
}

public class SynchronizedBlock {
    public static void main ( String[] args ) {
        Table1 obj = new Table1();
        //obj.print(2);
        thread1 t1 = new thread1(obj);
        thread2 t2 = new thread2(obj);
        t1.start();
        t2.start();
    }
}
```

Inter Thread Communication and its Program

Lab-7f //

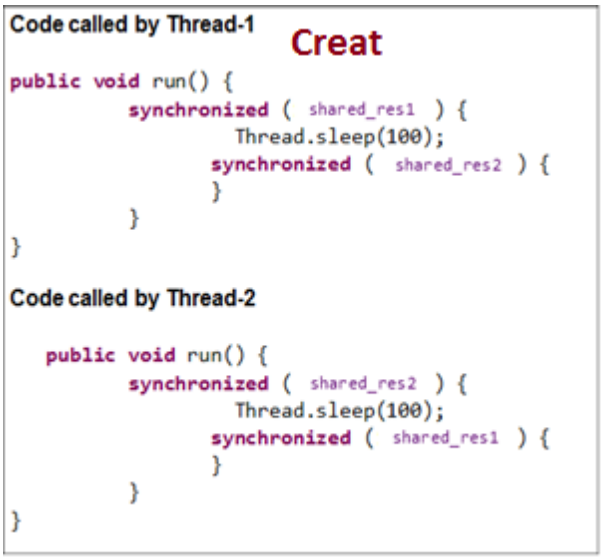
```
// https://cse.iitkgp.ac.in/~dsamanta/java/ch6.htm
// https://www.youtube.com/watch?v=lp3pOF4cu4k
// https://www.youtube.com/watch?v=-CUL0--zfrM
//https://ramj2ee.blogspot.com/2017/02/java-tutorial-java-threads-inter-thread.html


/* The following Java application shows how the transactions in a bank can be carried
out concurrently. */
class Account {
    public static int balance;

    void displayBalance() {
        System.out.println("Balance: " + balance);
    }

    synchronized void deposit(int amount){
        System.out.println("Before Deposit"+ balance);
        this.balance = balance + amount;
        System.out.println("After Deposit" + balance);
        System.out.println( balance + " is deposited");
        displayBalance();
        notify();
    }

    synchronized void withdraw(int amount){
        if(this.balance < amount){
            System.out.println("wait to deposit");
            try {
                wait();
```

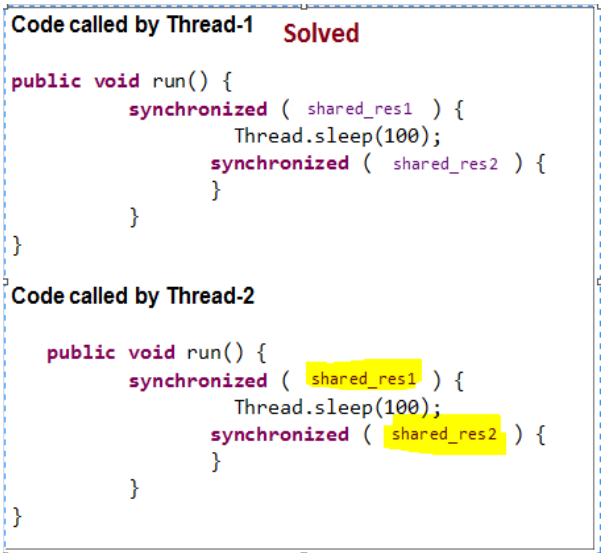
```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.balance = balance - amount;
        System.out.println( amount + " is withdrawn");
        displayBalance();
    }
}
public class IPC_BankTransation {
    public static void main ( String[] args ) {
        Account ABC = new Account();
        ABC.balance = 1000;
        ABC.displayBalance();
        final Account c=new Account();

        new Thread("T1"){
            public void run(){c.withdraw(1500);}
        }.start();

        new Thread("T2"){
            public void run(){c.deposit(900); }
        }.start();
    }
}
```

Output:

Balance: 1000
wait to deposit
Balance: 1900
1500 is withdrawn
Balance: 400

Deadlock Creat and Solve its Program





Lab-7g //

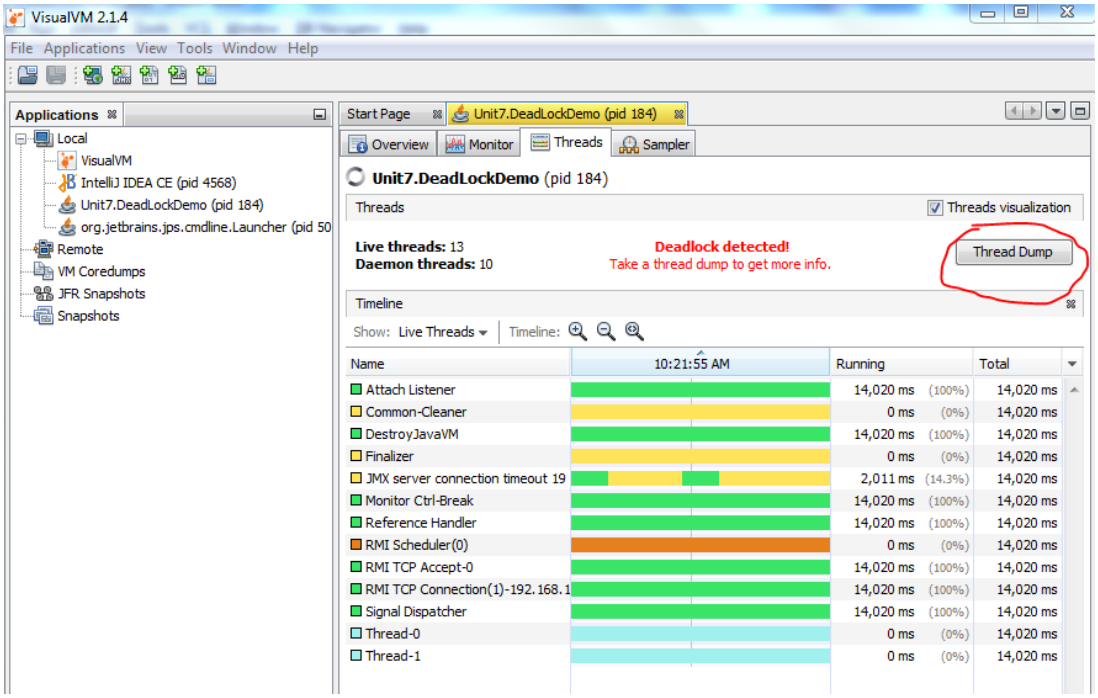
```
package Unit7;

public class DeadLockDemo {
    public static void main(String[] args) {
        //define shared resources
        final String shared_res1 = "Resource1";
        final String shared_res2 = "Resource2";
        // thread_one => locks shared_res1 then shared_res2
        Thread thread_one = new Thread() {
            public void run() {
                synchronized (shared_res1) {
                    System.out.println("Thread one: locked shared resource 1");

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

                    synchronized (shared_res2) {
                        System.out.println("Thread one: locked shared resource 2");
                    }
                }
            }
        };
        // thread_two=> locks shared_res2 then shared_res1
        Thread thread_two = new Thread() {
            public void run() {
                synchronized (shared_res2) {
                    System.out.println("Thread two: locked shared resource 2");

                    try { Thread.sleep(100);} catch (Exception e) {}
                    synchronized (shared_res1) {
```

```
                        System.out.println("Thread two: locked shared resource 1");
                    }
                }
            }
        };

        //start both the threads
        thread_one.start();
        thread_two.start();
    }
}
```

Output:

Thread two: locked shared resource 2

Thread one: locked shared resource 1



For detail click on “Thread Dump” it will give thread name

Viva Question:

Multi-Threading in Java: In computing, the producer-consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

- The producer’s job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

In this problem, we need two threads, Thread t1 (produces the data) and Thread t2 (consumes the data). However, both the threads shouldn’t run simultaneously.

Q&A

Q: What is the thread?

A: A thread is a lightweight subprocess. It is a separate path of execution because each thread runs in a different stack frame. A process may contain multiple threads. Threads share the process resources, but still, they execute independently..

Q: How to implement Threads in java?

A: Threads can be created in two ways i.e. by implementing **java.lang.Runnable interface or extending java.lang.Thread class and then extending run method.**
Thread has its own variables and methods, it lives and dies on the heap. But a thread of execution is an individual process that has its own call stack. Thread are lightweight process in java.
1)Thread creation by implementingjava.lang.Runnableinterface. We will create object of class which implements Runnable interface :

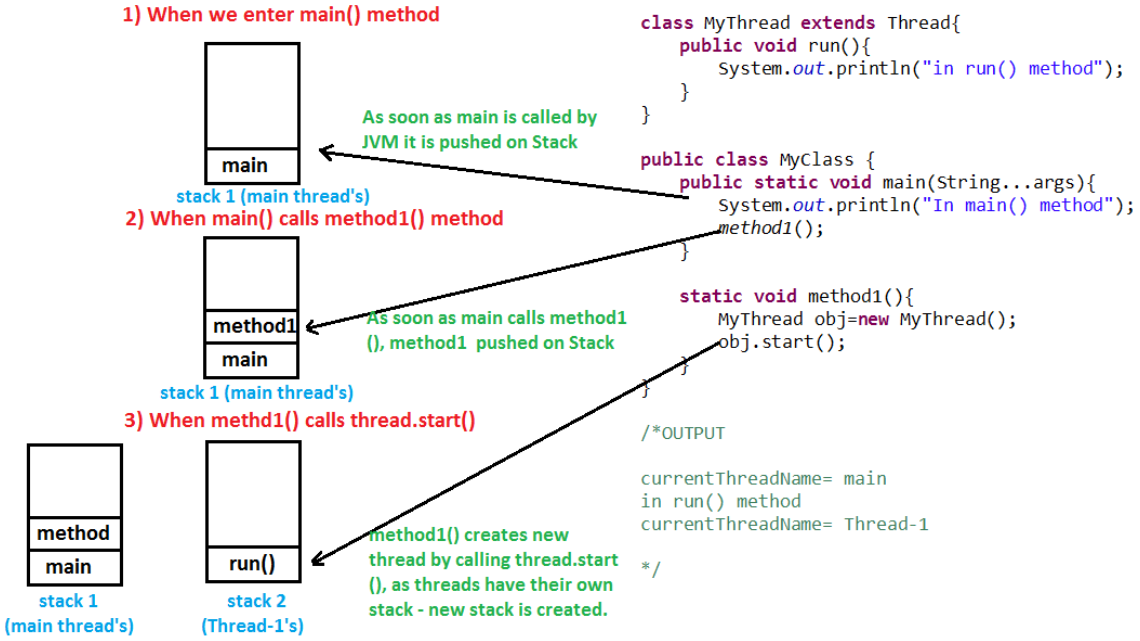
```
MyRunnable runnable=new MyRunnable();

Thread thread=new Thread(runnable);
```

2) And then create Thread object by calling constructor and passing reference of Runnable interface i.e. runnable object :

```
Thread thread=new Thread(runnable);
```

Q: Does Thread implements their own Stack, if yes how?

A: Yes, Threads have their own stack. This is very interesting question, where interviewer tends to check your basic knowledge about how threads internally maintains their own stacks.Shown in figure below.



Q: What is multithreading?

A: Multithreading is a process of executing multiple threads simultaneously. Multithreading is used to obtain the multitasking. It consumes less memory and gives the fast and efficient performance. Its main advantages are:

- Threads share the same address space.
- The thread is lightweight.
- The cost of communication between the processes is low.

Q: What do you understand by inter-thread communication?

A:
- The process of communication between synchronized threads is termed as inter-thread communication.
- Inter-thread communication is used to avoid thread polling in Java.
- The 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 can be obtained by wait(), notify(), and notifyAll() methods.

Q: How can you say Thread behaviour is unpredictable?

A: Thread behaviour is unpredictable because execution of Threads depends on Thread scheduler, thread scheduler may have different implementation on different platforms like windows, unix etc. Same threading program may produce different output in subsequent executions even on same platform. To achieve we are going to create 2 threads on same

Runnable Object, create for loop in run() method and start both threads. There is no surety that which threads will complete first, both threads will enter anonymously in for loop.

Q: How can you ensure all threads that started from main must end in order in which they started and also main should end in last?

A: We can use join() methodto ensure all threads that started from main must end in order in which they started and also main should end in last.In other words waits for this thread to die. Calling join() method internally calls join(0).

Q: What is difference between starting thread with run() and start() method?

A: When you call start() method, main thread internally calls run() method to start newly created Thread, so run() method is ultimately called by newly created thread.
When you call run() method main thread rather than starting run() method with newly thread it start run() method by itself.

Q: What is significance of using Volatile keyword?

A: Java allows threads to access shared variables. As a rule, to ensure that shared variables are consistently updated, a thread should ensure that it has exclusive use of such variables by obtaining a lock that enforces mutual exclusion for those shared variables.
If a field is declared volatile, in that case the Java memory model ensures that all threads see a consistent value for the variable.
Note:volatile is only a keyword, can be used only with variables.

Q: What is race condition in multithreading and how can we solve it?

A: When more than one thread try to access same resource without synchronization causes race condition.
So we can solve race condition by using either synchronized block or synchronized method. When no two threads can access same resource at a time phenomenon is also called as mutual exclusion.

Q: When should we interrupt a thread?

A: We should interrupt a thread when we want to break out the sleep or wait state of a thread. We can interrupt a thread by calling the interrupt() throwing the InterruptedException.

Q: What is the purpose of the Synchronized block?

A: The Synchronized block can be used to perform synchronization on any specific resource of the method. Only one thread at a time can execute on a particular resource, and all other threads which attempt to enter the synchronized block are blocked.

- Synchronized block is used to lock an object for any shared resource.

- The scope of the synchronized block is limited to the block on which, it is applied. Its scope is smaller than a method.

Q: What is the difference between notify() and notifyAll()?

A: The notify() is used to unblock one waiting thread whereas notifyAll() method is used to unblock all the threads in waiting state.

Q: How to detect a deadlock condition? How can it be avoided?

A: We can detect the deadlock condition by running the code on cmd and collecting the Thread Dump, and if any deadlock is present in the code, then a message will appear on cmd. Ways to avoid the deadlock condition in Java:

- **Avoid Nested lock:** Nested lock is the common reason for deadlock as deadlock occurs when we provide locks to various threads so we should give one lock to only one thread at some particular time.
- **Avoid unnecessary locks:** we must avoid the locks which are not required.
- **Using thread join:** Thread join helps to wait for a thread until another thread doesn't finish its execution so we can avoid deadlock by maximum use of join method.

Q: Difference between wait() and sleep().

A: **called from synchronized block** :wait() method is always called from synchronized block i.e. wait() method needs to lock object monitor before object on which it is called. But sleep() method can be called from outside synchronized block i.e. sleep() method doesn't need any object monitor.
IllegalMonitorStateException : if wait() method is called without acquiring object lock than IllegalMonitorStateException is thrown at runtime, but sleep() methodnever throws such exception.
Belongs to which class: wait() method belongs to java.lang.Object class but sleep() method belongs to java.lang.Thread class.
Called on object or thread: wait() method is called on objects but sleep() method is called on Threads not objects.
Thread state: when wait() method is called on object, thread that holded object's monitor goes from running to waiting state and can return to runnable state only when notify() or notifyAll()method is called on that object. And later thread scheduler schedules that thread to go from from runnable to running state.

when sleep() is called on thread it goes from running to waiting state and can return to runnable state when sleep time is up.
When called from synchronized block:when wait() method is called thread leaves the object lock. But sleep()method when called from synchronized block or method thread doesn't leaves object lock.

Q: Similarity between yield() and sleep().

A:

- yield() and sleep() method belongs to java.lang.Thread class.
- yield() and sleep() method can be called from outside synchronized block.
- yield() and sleep() method are called on Threads not objects.

Q: What are daemon threads?

A: Daemon threads are low priority threads which runs intermittently in background for doing garbage collection.

Few salient features of daemon() threads>

- Thread scheduler schedules these threads only when CPU is idle.
- Daemon threads are service oriented threads, they serves all other threads.
- These threads are created before user threads are created and die after all other user threads dies.
- Priority of daemon threads is always 1 (i.e. MIN_PRIORITY).
- User created threads are non daemon threads.
- JVM can exit when only daemon threads exist in system.
- we can use isDaemon() method to check whether thread is daemon thread or not.
- we can use setDaemon(boolean on) method to make any user method a daemon thread.
- If setDaemon(boolean on) is called on thread after calling start() method than IllegalThreadStateException is thrown.

Q: Can a constructor be synchronized?

A: No, constructor cannot be synchronized. Because constructor is used for instantiating object, when we are in constructor object is under creation. So, until object is not instantiated it does not need any synchronization.

- Enclosing constructor in synchronized block will generate compilation error.

- Using synchronized in constructor definition will also show compilation error.

COMPILATION ERROR = Illegal modifier for the constructor in type ConstructorSynchronizeTest; only public, protected & private are permitted
we can use synchronized block inside constructor.

Enlisted below are the means using which we can avoid deadlocks in Java.

#1) By avoiding nested locks

Having nested locks is the most important reason for having deadlocks. Nested locks are the locks that are given to multiple threads. Thus we should avoid giving locks to more than one thread.

#2) Use thread Join

We should use Thread.join with maximum time so that the threads can use the maximum time for execution. This will prevent deadlock that mostly occurs as one thread continuously waits for others.

#3) Avoid unnecessary lock

We should lock only the necessary code. Having unnecessary locks for the code can lead to deadlocks in the program. As deadlocks can break the code and hinder the flow of the program we should be inclined to avoid deadlocks in our programs.

Frequently Asked Questions

Q #1) What is Synchronization and why is it important?

Answer: Synchronization is the process of controlling the access of a shared resource to multiple threads. Without synchronization, multiple threads can update or change the shared resource at the same time resulting in inconsistencies.
Thus we should ensure that in a multi-threaded environment, the threads are synchronized so that the way in which they access the shared resources is mutually exclusive and consistent.

Q #2) What is Synchronization and Non – Synchronization in Java?

Answer: Synchronization means a construct is a thread-safe. This means multiple threads cannot access the construct (code block, method, etc.) at once.
Non-Synchronized constructs are not thread-safe. Multiple threads can access the non-synchronized methods or blocks at any time. A popular non- synchronized class in Java is StringBuilder.

Q #3) Why is Synchronization required?

Answer: When processes need to execute concurrently, we need synchronization. This is because we need resources that may be shared among many processes.
In order to avoid clashes between processes or threads for accessing shared resources, we need to synchronize these resources so that all the threads get access to resources and the application also runs smoothly.

Q #4) How do you get a Synchronized ArrayList?

Answer: We can use Collections.synchronized list method with ArrayList as an argument to convert ArrayList to a synchronized list.

Q #5) Is HashMap Synchronized?

Answer: No, HashMap is not synchronized but HashTable is synchronized.

Distributed Systems

http://www.cs.unc.edu/~dewan/734/current/index.html

// https://cse.iitkgp.ac.in/~dsamanta/java/ch6.htm