6.5 Major Concept of Multithreaded Programming

6.5.1 Synchronization

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues.

For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time.

This is implemented using a concept called monitors. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Java programming language provides a very handy way of creating threads and synchronizing their task by using synchronized blocks.

You keep shared resources within this block. Following is the general form of the synchronized statement:

```
synchronized(objectidentifier)
 {
// Access shared variables and other shared resources
}
```

Here, the objectidentifier is a reference to an object whose lock associates with the monitor that the synchronized statement represents. Now we are going to see two examples, where we will print a counter using two different threads.

When threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized() block, then it prints counter very much in sequence for both the threads.

6.5.2 Interthread Communication

If you are aware of interprocess communication then it will be easy for you to understand interthread communication. Interthread communication is important when you develop an application where two or more threads exchange some information.

There are three simple methods and a little trick which makes thread communication possible.

All the three methods are listed below:

Sr. NO.	Methods with Description
1	**public void wait()** Causes the current thread to wait until another thread invokes the notify().
2	**public void notify()** Wakes up a single thread that is waiting on this object's monitor.
3	**public void notifyAll()** Wakes up all the threads that called wait( ) on the same object.

These methods have been implemented as final methods in Object, so they are available in all the classes. All three methods can be called only from within a synchronized context.

6.5.3 Deadlock

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.

Deadlock occurs when multiple threads need the same locks but obtain them in different order.

A Java multithreaded program may suffer from the deadlock condition because the synchronized keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object. Here is an example.

Example

```
public class TestThread{
public static Object Lock1 = new Object();
public static Object Lock2 = new Object();
public static void main(String args[]){
ThreadDemo1 T1 = new ThreadDemo1();
ThreadDemo2 T2 = new ThreadDemo2();
T1.start();
T2.start();
}
private static class ThreadDemo1 extends Thread{
public void run(){
synchronized (Lock1){
System.out.println("Thread 1: Holding lock 1...");
try{
 Thread.sleep(10);
```

```
 }
catch (InterruptedException e)
 { }
System.out.println("Thread 1: Waiting for lock 2...");
synchronized (Lock2){
System.out.println("Thread 1: Holding lock 1 & 2...");
}
}
}
}
private static class ThreadDemo2 extends Thread{
public void run(){
synchronized (Lock2){
System.out.println("Thread 2: Holding lock 2...");
try{
Thread.sleep(10);
}
catch (InterruptedException e)
 { }
System.out.println("Thread 2: Waiting for lock 1...");
synchronized (Lock1){
System.out.println("Thread 2: Holding lock 1 & 2...");
}
}
}
}
}
```