11. MULTITHREADING

1. Multithreading Concept:

Multithreading is a specialialized form of multitasking. Multithreaded program contains two or more parts are running concurrently. Each part is known as thread.

2. Java Thread Model:

Difference between Process and Thread:

Process	Thread
Prgram in execution	Part of a process
Heavyweight	Lightweight
Each process has its own address space	Threads shares the address space
Multiprocessing: More overhead	Multithreading: Less overhead
Interprocess communication is expensive and limited	Interthread communication is less expensive
Context switching requires high cost	Context switching requires low cost
Example: downloading files along with listening music	Example: in editor, character printing along with formatting

The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

3. Thread States:

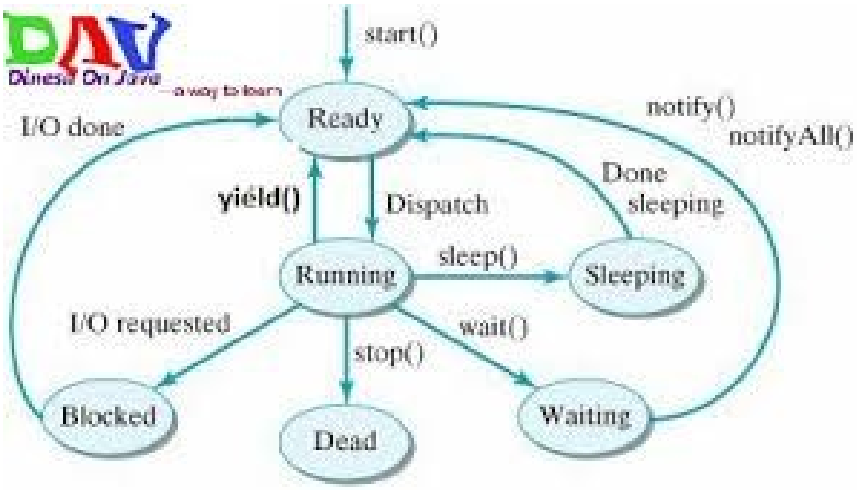
1. New

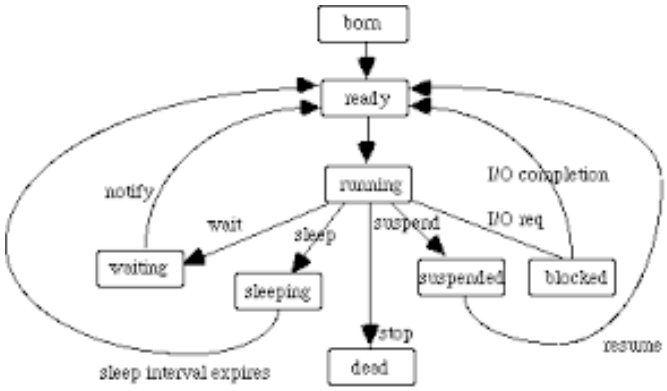
A thread is said to be in new state when we created the thread instance, but we have not yet called *start()* on the thread newly created thread instance. Even though it is a live thread object, it is not a thread of execution. At this state, thread is not active.

2. Runnable / Ready

In the Runnable state a thread is eligible to run, however the Thread Scheduler not selects the thread to start its execution. A thread will get into Runnable state after we call *start()* method. Also a thread can come back to Runnable state from other states like sleeping, waiting or blocked states. The important point to remember is that, once we call *start()* method on any thread instance, the thread will move to Runnable state which makes the thread eligible to run. The Thread Scheduler may or may not have picked the thread to run.

11. MULTITHREADING





3. Running

Once the Thread Scheduler selects a thread to run from the runnable thread pool, the thread starts execution. This is the state we call a thread is in Running state and thread becomes thread of execution.

4. Waiting

If any thread is waiting for any shared resource then it will call wait method. It will come out of wait when either timer expires or it will get notification by some other thread.

5. Sleeping

Thread can go to sleep state if its execution is paused for some period by calling sleep method.

6. Blocked:

Thread will go to blocked state if is waiting for I/O. e.g. User Input. When I/O get completed, it will come out of this state and goes to ready state.

7. Suspended

If thread is suspended for some period then it will go to suspended state. It will come out of this state when it gets resumed or timer expires.

From all these states the thread becomes eligible to run again, however Thread Scheduler ultimately decides which runnable thread becomes thread of execution.

8. Dead

A thread is considered dead once its *run()* method completed execution. Although a thread's *run()* method completed execution it is still a Thread object, but this Thread object is not a thread of execution. Once the thread completes its *run()* method and dead, it cannot be brought back to thread of execution or even to runnable state. Invoking *start()* method on a dead thread causes runtime exception.

A thread is not eligible to run if it is in waiting, sleeping or blocked state. But the thread is alive, and it is not in runnable state. A thread can transition back to runnable state when particular event occurs.

4. Thread class methods:

Thread class resides in lang package. Thread class defines several methods and variables to manage the threads. Some of those:

Method	Meaning
getName	Obtain a thread's name.
setName	Sets new name.
getPriority	Obtain a thread's priority.
SetPriority	Sets new priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

5. Thread Priorities:

These are used by schedular to decide which thread to chose from ready state to execute. In theory, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority.

The following methods are used to get the priority and set new priority to threads.

```
final int getPriority( )
final void setPriority(int level)
```

The level is defined by following three constants.

Thread class has three constant variables to assign priorities. The values are as follows:

All three variables are **public static and final**

Variable	Value
MIN_PRIORITY	1
MAX_PRIORITY	10
NORM_PRIORITY	5

6. Runnable interface:

It contains a method signature :

public void run();

This method should be overriden in a class to assign work to child thread.

7. The main thread:

It is a parent thread from which we can create multiple threads. The execution of main thread starts with main method. i.e. main method is entry point of main thread. It get started automatically by JVM. We can control this thread by using Thread class properties.

To control the thread and to obtain its reference following method is used.

public static Thread currentThread()

This method returns thread object of current executing thread. The thread object contain following information regarding the current thread.

1. Name of thread
2. Priority of thread
3. Group of thread

By default, following is the property of main thread.

1. name of main thread : main.
2. priority : 5
3. group : main

A thread group is a data structure that controls the state of a collection of threads as a whole. It decides the working of thread. To control the thread group of thread we have ThreadGroup class in java.

Example:

```
//Controlling the main Thread.
class CurrentThreadDemo
{
        public static void main(String args[])
        {
```

```
                Thread t = Thread.currentThread();
                System.out.println("Current thread: " + t);

                // change the name of the thread
                t.setName("My Thread");
                System.out.println("After name change: " + t);
                try
                {
                        for(int n = 5; n > 0; n--)
                        {
                                System.out.println(n);
                                System.out.println("Thread is sleeping");
                                Thread.sleep(1000);
                                System.out.println("Thread woke up");
                        }
                }
                catch (InterruptedException e)
                {
                        System.out.println("Main thread interrupted");
                }
        }
}
```

Output:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
Thread is sleeping
Thread woke up
4
Thread is sleeping
Thread woke up
3
Thread is sleeping
Thread woke up
2
Thread is sleeping
Thread woke up
1
Thread is sleeping
Thread woke up
```

To change or access the name we use following thread class methods:

final void setName(String threadName)

final String getName( )

8. Forms of sleep method:

This method causes the thread from which it is called to pause the execution for the specified period of time. This method is of Thread class and it is overloaded.

It has following syntax:

1. public static void sleep(long milliseconds) throws InterruptedException

- The number of milliseconds to suspend is specified in milliseconds.

2. public static void sleep(long milliseconds, int nanoseconds) throws InterruptedException

- The time period to suspend is specified in milliseconds and nanoseconds.

9. Creating Threads in java:

There are two ways to create child threads.
1. extend Thread class
2. implement Runnable interface

Why two ways?

1. When u want to use and extend the more properties of Thread class, then use first way.
2. Go for second way for following reasons:
   - If you want to just create and want to execute the threads, then use second way.
   - One class can not extend at a time two or more classes as java does not support multiple inheritance. But one clas can implement more than one iterface.

That is

1. **extending Thread class:** use this way when a subclass want to access more properties of Thread class along with creation of child thread. But class then can not extend another class.
2. **implementing Runnable interface:** use this way when a class only want to create a child thread and execute it by overriding run method and does not need more properties of thread class. By using this approach a class can extend other class also and it can implement other interfaces also.

This is most better way to create the child threads.

1. Creating threads by extending Thread class:

By extending Thread class, a class can acquire the object of Thread. Also a child thread can directly access the properties of Thread class.

For this way to create the threads, following Thread class constructors are used.

- Thread()
- Thread(String name)
- Thread(ThreadGroup group, String name)

11. MULTITHREADING

The following are the steps to create, start and execute child threads.

- start the thread using start method
- implement run method and assign work to child thread.

Note: If we try to call start method two times for single thread object then it will throw the exception **IllegalThreadStateException**

Example by extendingThread class:

```
class ChildThread extends Thread
{
	ChildThread()
	{
		super("demochild");  // to assign name to the thread.
		start();

	}
	public void run()
	{
		System.out.println("execution of child thread starts from here....");
		try{
		for(int i = 5; i > 0; i--) {
			System.out.println(Thread.currentThread().getName()+" : "+ i);
			Thread.sleep(500);
			}
			} catch (InterruptedException e) {
			System.out.println("Child interrupted.");
			}
			System.out.println("Exiting child thread.");
	}
}
public class ExtendThreadDemo {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		ChildThread ct=new ChildThread();
	}
}
```

Output:

execution of child thread starts from here....

```
demochild : 5
demochild : 4
demochild : 3
demochild : 2
demochild : 1
Exiting child thread.
```

2. Creating threads by implementing Runnable interface:

It is an easiest way to create and execute threads.

In this type, The thread object has to be created explicitly by calling Thread class constructor.

The following constructors are only used to create child thread using this way.

- Thread(Runnable threadOb);
- Thread(Runnable threadOb, String name)
- Thread(ThreadGroup groupOb, Runnable threadOb)
- Thread(ThreadGroup groupOb, Runnable threadOb, String threadName)

Runnable interface contains a method signature, **public void run();**

It has following syntax:

```
public interface Runnable
{
        void run();
}
```

In a class implementing Runnable interface, this method has to override to assign the work to child thread. Here we can create the thread in different class and we can execute the threads in different class which is implementing Runnable interface.

Steps to create thread:

- Create Thread class object using one of the above constructors
- Start the thread
- Override run method and assing work to child thread

Example using Runnable interface:

1. where creating, starting and executing a thread in same class

```
class ChildThread1 implements Runnable
{
        Thread child;
```

```
        ChildThread1()
        {
                child=new Thread(this,"childthread");
                child.start();
                child.setName("demochild"); // to assign name to the thread.
                child.setPriority(Thread.MIN_PRIORITY);
        }
        public void run()
        {
                System.out.println("execution of child thread starts from here....");
                try{
                for(int i = 5; i > 0; i--) {
                        System.out.println(Thread.currentThread().getName()+" : "+ i);
                        Thread.sleep(500);
                        }
                        } catch (InterruptedException e) {
                        System.out.println("Child interrupted.");
                        }
                        System.out.println("Exiting child thread.");
        }
}
public class RunnableDemo {

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                ChildThread1 ct=new ChildThread1();
        }
}
```

Output:

```
execution of child thread starts from here....
childthread : 5
childthread : 4
childthread : 3
childthread : 2
childthread : 1
Exiting child thread.
```

2. where creating starting the thread in one class and executing that thread in another class

```
class ChildThread2 implements Runnable {

        public void run()
```

```
        {
                System.out.println("execution of child thread starts from here....");
                try{
                for(int i = 5; i > 0; i--) {
                        System.out.println(Thread.currentThread().getName()+" : "+ i);
                        Thread.sleep(500);
                        }
                        } catch (InterruptedException e) {
                        System.out.println("Child interrupted.");
                        }
                        System.out.println("Exiting child thread.");

        }
}

class createThread
{
        Thread child;
        createThread( ChildThread2 ct)
        {

                child=new Thread(ct,"childthread");
                child.start();
                child.setName("demochild"); // to assign name to the thread.
                child.setPriority(Thread.MIN_PRIORITY);
        }
}
public class RunnableDemo1 {

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                ChildThread2 ct=new ChildThread2();
                createThread ct1=new createThread(ct);
        }
}
```

Output:

```
execution of child thread starts from here....
childthread1 : 5
childthread1 : 4
```

childthread1 : 3
childthread1 : 2
childthread1 : 1
Exiting child thread.

10. Creating Multiple Threads:

Example:

```
class ChildThreads implements Runnable
{
	Thread child;
	ChildThreads(String name)
	{

		child=new Thread(this,name);
		child.start();

	}
	public void run()
	{String name=Thread.currentThread().getName();
		System.out.println("execution of "+name+" starts from here....");
		try{

		for(int i = 5; i > 0; i--) {
			System.out.println(name+" : "+ i);
			Thread.sleep(500);
			}
			} catch (InterruptedException e) {
			System.out.println("Child interrupted.");
			}
				System.out.println("Exiting "+name);

	}
}
public class MultipleThreads {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		ChildThreads ct1=new ChildThreads("Child 1");
```

```
                ChildThreads ct2=new ChildThreads("Child 2");
                ChildThreads ct3=new ChildThreads("Child 3");
                try {
                        for(int i = 5; i > 0; i--) {
                        System.out.println("Main Thread: " + i);
                        Thread.sleep(1000);
                        }
                        } catch (InterruptedException e) {
                        System.out.println("Main thread interrupted.");
                        }
                        System.out.println("Main thread exiting.");
        }

}
```

Output:

```
execution of Child 1 starts from here....
Main Thread: 5
execution of Child 2 starts from here....
Child 2 : 5
Child 1 : 5
execution of Child 3 starts from here....
Child 3 : 5
Child 2 : 4
Child 1 : 4
Child 3 : 4
Main Thread: 4
Child 2 : 3
Child 1 : 3
Child 3 : 3
Child 2 : 2
Child 1 : 2
Child 3 : 2
Main Thread: 3
Child 2 : 1
Child 1 : 1
Child 3 : 1
Exiting Child 2
Exiting Child 1
Exiting Child 3
```

Main Thread: 2
Main Thread: 1
Main thread exiting.

11. Using isAlive and join methods:

If the sleeping period of parent thread is less than the that of child thread, parent thread may exit first before the completion of its child thread. But it should not be happened. Thus parent thread should wait until all its child threads complete their work and exit. At last parent thread should exit. To wait for threads, parent threads calls join method, and it will not return from there until the child thread on which join is called, exits. Generally join method is used to wait for a thread until it dies.

isAlive method is used to check whether the thread on which this method is called, is alive or not. This methods returns true if it is alive otherwise false. This method is used when the parent thread has to assign some work to its child after checking that thread is alive or not.

Syntax of both the methods:

```
public final void join( ) throws InterruptedException
public final boolean isAlive( )
```

Example:

```
class ChildThreads implements Runnable
{
        Thread child;
        ChildThreads(String name)
        {

                child=new Thread(this,name);
                child.start();

        }
        public void run()
        {String name=Thread.currentThread().getName();
                System.out.println("execution of "+name+" starts from here....");
                try{

                for(int i = 5; i > 0; i--) {
                        System.out.println(name+" : "+ i);
                        Thread.sleep(1000);


                        }
```

```
                        } catch (InterruptedException e) {
                        System.out.println("Child interrupted.");
                        }
                        System.out.println("Exiting "+name);
        }
}
public class MultipleThreads {

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                ChildThreads ct1=new ChildThreads("Child 1");
                ChildThreads ct2=new ChildThreads("Child 2");
                ChildThreads ct3=new ChildThreads("Child 3");

                System.out.println(ct1.child.getName()+" is alive? "+ct1.child.isAlive());
                System.out.println(ct2.child.getName()+" is alive? "+ct2.child.isAlive());
                System.out.println(ct3.child.getName()+" is alive? "+ct3.child.isAlive());
                try {
                                for(int i = 5; i > 0; i--) {
                                        System.out.println("Main Thread: " + i);
                                        Thread.sleep(500);
                                        }
                        System.out.println("Main thread waiting....");

                        ct1.child.join();
                        ct2.child.join();
                        ct3.child.join();

                        System.out.println(ct1.child.getName()+" is alive? "+ct1.child.isAlive());
                        System.out.println(ct2.child.getName()+" is alive? "+ct2.child.isAlive());
                        System.out.println(ct3.child.getName()+" is alive? "+ct3.child.isAlive());

                        } catch (InterruptedException e) {
                        System.out.println("Main thread interrupted.");
                        }
                        System.out.println("Main thread exiting.");
        }

}
```

Output:

```
execution of Child 1 starts from here....
Child 1 : 5
execution of Child 3 starts from here....
Child 3 : 5
execution of Child 2 starts from here....
Child 2 : 5
Child 1 is alive? true
Child 2 is alive? true
Child 3 is alive? true
Main Thread: 5
Main Thread: 4
Child 1 : 4
Child 3 : 4
Child 2 : 4
Main Thread: 3
Main Thread: 2
Child 1 : 3
Child 3 : 3
Child 2 : 3
Main Thread: 1
Main thread waiting....
Child 1 : 2
Child 2 : 2
Child 3 : 2
Child 1 : 1
Child 3 : 1
Child 2 : 1
Exiting Child 1
Exiting Child 3
Exiting Child 2
Child 1 is alive? false
Child 2 is alive? false
Child 3 is alive? false
Main thread exiting.
```

12. Thread Synchronization:

When two or more threads want to access same shared resource at a time, then race condition will arise and resource may go to inconsistent state.

Thus synchronization is to achieve the lock on resource. The thread having lock on the resource

only use that resource at a time. All others thread should wait for that resource until the thread holding the resources releases the lock. In java shares resourse is a shared object i.e. shared memory location.

There are two ways to achieve synchroniztion among threads

1. using synchronized methods
2. using synchronized blocks

In both the ways, **synchronized** keyword is used

1. using synchronized methods:

Declare the method as synchronized, if a thread enters into synchronized method on shared instance, all other threads which are trying to call the same or other synchronized methods on same instance, will wait until that threads comes out of that synchronized method.

Syntax:

```
accessmodifier synchronized returntype mehtodname(parameterlist)
{
        // synchronized code
}
```

Example:

```
public synchronized void deposit(int amt)
{
        // code for deposting amount in account
}
```

Example: (without synchronization)

In following programs, husband and wife threads are simultaneously depositing and withdrawing some amount from the joint account at the same time. If two threads are not synchronized, threads get the final balance in inconsistent state, as both threads do not know that they are working simultaneously.

```
import java.util.Scanner;

class HusbandThread implements Runnable
{
        Thread husband;


        Bank b;
```

```
        HusbandThread(Bank b)
        {
                this.b=b;
                husband=new Thread(this,"depositthread");
                husband.start();
        }
        public void run()
        {
                b.deposit();
        }
}

class WifeThread implements Runnable
{
        Thread wife;
        Bank b;
        WifeThread(Bank b)
        {
                this.b=b;
                wife=new Thread(this,"withdrawthread");
                wife.start();
        }
        public void run()
        {
                b.withdraw();
        }
}

class Bank
{
        private volatile double balance=50000; // initial balance
        private int withdrawamt;
        private int depositamt;

        public double deposit()
        {       try
                {
                Scanner sc=new Scanner(System.in);


                System.out.println("Balance before deposit: "+balance);
```

```
                Thread.sleep(1000);
                System.out.println("Enter deposit amount:");
                depositamt=sc.nextInt();
                Thread.sleep(500);
                balance=balance+depositamt;
                Thread.sleep(1500);
                System.out.println("Balance after deposit: "+balance);
                }
                catch(Exception e)
                {
                        e.printStackTrace();
                }
                return balance;
        }
        public double withdraw()
        {
                try
                {
                Scanner sc=new Scanner(System.in);
                System.out.println("Balance before withdraw: "+balance);
                Thread.sleep(500);
                System.out.println("Enter withdraw amount:");
                 withdrawamt=sc.nextInt();
                        Thread.sleep(1000);
                balance=balance - withdrawamt;
                Thread.sleep(1500);
                System.out.println("Balance after withdraw: "+balance);

                }
                catch(Exception e)
                {
                e.printStackTrace();
                }
                return balance;
        }

}
public class ThreadSynchronization {



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

```
                // TODO Auto-generated method stub
                Bank b=new Bank();
                HusbandThread ht=new HusbandThread(b);
                WifeThread wt=new WifeThread(b);
        }
}
```

Output:

```
Balance before withdraw: 50000.0
Balance before deposit: 50000.0
Enter withdraw amount:
Enter deposit amount:
3000
2000
Balance after withdraw: 47000.0
Balance after deposit: 49000.0
```

Here, the husband has read the balance before deposit as 50000 but he will get balance after depositing 2000 as 49000. Thus he feels that the balance is in consistent state as inbetween the wife thread withdraws the amount.
The solution for this is do thread synchronization.

Example: (with synchronization)

The following programs synchronizes both the methods. Thus if the thread enters in one of the synchronized method, other will wait for any synchronized method until first comes out.

```
import java.util.Scanner;
class HusbandThread implements Runnable
{
        Thread husband;
        Bank b;

        HusbandThread(Bank b)
        {
                this.b=b;
                husband=new Thread(this,"depositthread");
                husband.start();


        }
        public void run()
```

```
        {
                b.deposit();
        }
} // end clas HusbandThread

class WifeThread implements Runnable
{
        Thread wife;
        Bank b;
        WifeThread(Bank b)
        {
                this.b=b;
                wife=new Thread(this,"withdrawthread");
                wife.start();
        }
        public void run()
        {
                b.withdraw();
        }
}// end class WifeThread

class Bank
{
        private volatile double balance=50000; // initial balance
        private int withdrawamt;
        private int depositamt;

        public synchronized double deposit()
        {       try
                {
                Scanner sc=new Scanner(System.in);
                System.out.println("Balance before deposit: "+balance);
                Thread.sleep(1000);
                System.out.println("Enter deposit amount:");
                depositamt=sc.nextInt();
                Thread.sleep(500);
                balance=balance+depositamt;
                Thread.sleep(1500);


                System.out.println("Balance after deposit: "+balance);
                }
```

```
                catch(Exception e)
                {
                        e.printStackTrace();
                }
                return balance;

        }
        public synchronized double withdraw()
        {
                try
                {
                Scanner sc=new Scanner(System.in);
                System.out.println("Balance before withdraw: "+balance);
                Thread.sleep(500);
                System.out.println("Enter withdraw amount:");
                 withdrawamt=sc.nextInt();
                        Thread.sleep(1000);
                balance=balance - withdrawamt;
                Thread.sleep(1500);
                System.out.println("Balance after withdraw: "+balance);
                }
                catch(Exception e)
                {
                e.printStackTrace();
                }
                return balance;
        }
}// end class Bank

public class ThreadSynchronization {

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                Bank b=new Bank();
                HusbandThread ht=new HusbandThread(b);
                WifeThread wt=new WifeThread(b);
        }
}// end class ThreadSynchronization
```

Output:

Balance before deposit: 50000.0

Enter deposit amount:
2000
Balance after deposit: 52000.0
Balance before withdraw: 52000.0
Enter withdraw amount:
5000
Balance after withdraw: 47000.0

2. using synchronized blocks:

Two reasons why to use synchronized blocks:

- to lock minimum and only required code use synchronized blocks, as if we use synchronized methods, the nonrequired code also get synchronized and waiting period of other waiting threads may get increased.
- If a class on which methods to be synchronized, is not accessible, we can not modify definition of methods by attaching synchronized keyword, instead call the methods from synchronized blocks, then automatically methods will get synchronized

Syntax:

```
synchronized(object)
{
        //code to be synchronized
}
```

Here **object** is the class object on which synchronization is to be done.

Example: (using synchronized blocks)

```
class HusbandThread1 implements Runnable
{
        Thread husband;
        Bank1 b;
        Scanner sc=new Scanner(System.in);
        HusbandThread1(Bank1 b)
        {
                this.b=b;
                husband=new Thread(this,"depositthread");


                husband.start();
        }
```

```
        public void run()
        {
                int depositamt;
                synchronized(b) // synchronized block
                {
                        System.out.println("Balance before deposit: "+b.getBalance());

                        System.out.println("Enter deposit amount:");
                        depositamt=sc.nextInt();
                        double balance=b.deposit(depositamt); // method also will get synchronized
                        System.out.println("Balance after deposit: "+balance);
                }
        }
}

class WifeThread1 implements Runnable
{
        Thread wife;
        Bank1 b;
        Scanner sc=new Scanner(System.in);
        WifeThread1(Bank1 b)
        {
                this.b=b;
                wife=new Thread(this,"withdrawthread");
                wife.start();
        }
        public void run()
        { int withdrawamt;

                synchronized(b) // synchronized block
                {
                        System.out.println("Balance before withdraw: "+b.getBalance());

                        System.out.println("Enter withdraw amount:");
                        withdrawamt=sc.nextInt();

                        double balance=b.withdraw( withdrawamt); // method will get synchronized
                        System.out.println("Balance after withdraw: "+balance);

                }
        }
}
```

```
class Bank1
{
	private volatile double balance=50000; // initial balance

	public double getBalance() {
		return balance;
	}
	public void setBalance(double balance) {
		this.balance = balance;
	}

	public synchronized double deposit(int depositamt)
	{	try
		{

		Thread.sleep(500);
		balance=balance+depositamt;
		Thread.sleep(1500);

		}
		catch(Exception e)
		{
			e.printStackTrace();
		}
		return balance;

	}

	public synchronized double withdraw(int withdrawamt)
	{
		try
		{

		balance=balance - withdrawamt;
		Thread.sleep(1500);



		}
		catch(Exception e)
		{
```

```
                e.printStackTrace();
                }
                return balance;
        }

}
public class SynchronizedBlocks {

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                Bank1 b=new Bank1();
                HusbandThread1 ht=new HusbandThread1(b);
                WifeThread1 wt=new WifeThread1(b);
        }

}
```

Output:

Balance before deposit: 50000.0
Enter deposit amount:
2000
Balance after deposit: 52000.0
Balance before withdraw: 52000.0
Enter withdraw amount:
1000
Balance after withdraw: 51000.0

13. InterThread Communication:

In thread synchronization, a waiting thread has to repeatedly check the required resource is releasesd by other threads or not. This is called polling. The thread consumes more CPU cycles in polling. To avoid polling, interthread communication is required. Communication is required among multiple thread objects

Inter thread communication is achieved via three methods of Object class.

- wait() - it is overloaded
- notify()
- notifyAll()

Syntax:

- **final void wait()throws InterruptedException**

threads waits until it will not get notification from other thread

- **final void wait(long miliseconds)throws InterruptedException**
- **final void wait(long miliseconds, int nanoseconds)throws InterruptedException**

thread waits until **either** it will not get notification from other thread **or** timer does not expire

- **final void notify()**

thread notifies waiting thread to wake up then it will go to ready state

- **final void notifyAll()**

thread notifies all waiting threads to wake up but all will go to ready state

For example :

In Producer Consumer problem, Consumer has to wait until Producer does not produce the value and Producer has to wait until Consumer does not consume the value. After producing the value, Producer has to notify to Consumer and after consuming value, Consumer has to notify to Producer.

The same problem is implemented below.

Example:

```
class Operation {
	int n;
	volatile boolean flagset = false;
	synchronized int get() {
		if(flagset==false)
			try {
				wait();
			} catch(InterruptedException e) {
				System.out.println("InterruptedException caught");
		}
		System.out.println("Got: " + n);
		flagset= false;
		notify();
		return n;
	}
	synchronized void put(int n) {
		if(flagset==true)
			try {
				wait();
			} catch(InterruptedException e) {
				System.out.println("InterruptedException caught");
		}
		this.n = n;
```

```
                flagset  = true;
                System.out.println("Put: " + n);
                notify();
        }
}//end Operation

class Producer implements Runnable {
        Operation q;
        Producer(Operation q) {
                this.q = q;
                new Thread(this, "Producer").start();
        }
        public void run() {
                int i = 0;
                try {
                        while(true) {
                                q.put(i);
                                i++;
                                Thread.sleep(2000);
                        }
                }
                catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
                }
        }
}//end Producer

class Consumer implements Runnable {
        Operation q;
        Consumer(Operation q) {
                this.q = q;
                new Thread(this, "Consumer").start();
        }
        public void run() {
                try {
                        while(true) {
                                q.get();
                                Thread.sleep(2000);
                        }
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
```

```
                    e.printStackTrace();
                }
        }
}//end Consumer

public class ProducerConsumerProblem {
        public static void main(String args[]) {
                Operation q=new Operation();
                new Producer(q);
                new Consumer(q);
                }
}// end ProducerConsumerProblem
```

Output:

```
Put: 0
Got: 0
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
Put: 6
Got: 6
Put: 7
Got: 7
Put: 8
Got: 8
```

Here volatile keyword is used for the variable flagset as multiple threads can change the value of volatile variable and always updated value is accessed. Here flagset value is changed by two threads from true to false and vice versa.

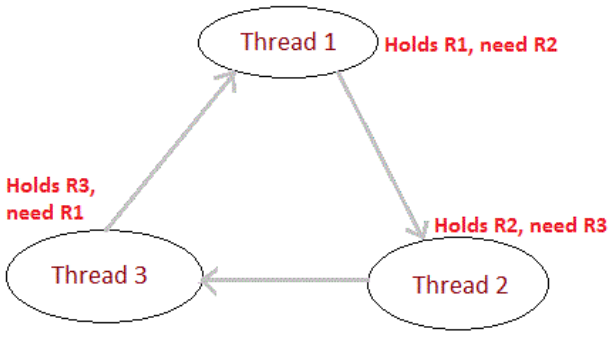
13.1 Diferrence between sleep, wait and join method:

sleep method	wait method	join method
It is used to pause the execution	If a thread needs to wait for any	If a thread t1 calls join on t2, it

of thread for some period of time	resource it calls wait method	will wait until t2 completes its execution
It is of Thread class	It is of Object class	It is of Thread class
It has two overloaded forms 1. public static void sleep(long miliseconds) 2. public static void sleep(long miliseconds, int nanoseconds)	It has three overloaded forms 1. public final void wait() 2. public final void wait(long miliseconds) 3. public final void wait(long miliseconds, int nanoseconds)	It has three overloaded forms 1. public void join() 2. public void join(long miliseconds) 3. public void join(long miliseconds, int nanoseconds)
It is class method	It is instance method and is final	It is instance method
When time period expires thread comes out of sleep state	If a thread calls wait(), then it will come out of waiting state when another thread send notification by calling notify() or notifyAll() If a thread calls wait([param]), then it will wait for the notification only until time get expires, after that it will come out of waiting state.	If a thread calls join(), then it will come out of waiting state when another thread completes its execution If a thread calls join([param]), then it will wait only until time get expires, after that it will come out of waiting state.
It is not needed for the interthread communication	It is needed in interthread communication	It is not needed for the interthread communication

14. DeadLocks:

It is a Circular dependancy. If a thread holding one resource and waiting for other resource which is hold by other thread, but that thread is waiting for the resource hold by waitng thread.
If deadlock occures, one of the thread's execution is stopped.



Deadlock Condition

15. Suspending Resuming and Stopping the threads:

The methods suspend, resume and stop are deprecated.