UNIT - 4

MULTI THREADING

I/O

Multi Threading

Background Information

Multitasking: Ability to execute two or more tasks in parallel or simultaneously is known as multitasking. Multitasking is of two types: 1) Process based multitasking and 2) Thread based multitasking.

Process based multitasking: Executing two or more processes simultaneously is known as process based multitasking. For example, we can listen to music and browse internet at the same time. The processes in this example are the music player and browser.

Thread based multitasking: Thread is a part of process i.e., a process can contain one or more threads. If two or more such threads execute simultaneously, it is known as thread based multitasking or multithreading.

For example, using a browser we can navigate through the webpage and at the same time download a file. In this example, navigation is one thread and downloading is another thread.

Another example for multithreading is in a word-processing application like MS Word, we can type text in one thread and spell checker checks for mistakes in another thread.

Java doesn't provide control over process based multitasking. But, it allows programmers to control multithreading.

Advantage of multithreading is, it increases CPU utilization i.e., CPU is made busy (always executing some thread) without leaving it idle.

Multithreading in Java

A thread is a separate flow of execution in a program. All Java programs by default contain a single thread called the "Main thread". A thread contains a set of statements like a method in Java.

The difference between thread and a method is, unlike methods, threads can run simultaneously or in parallel with other threads.

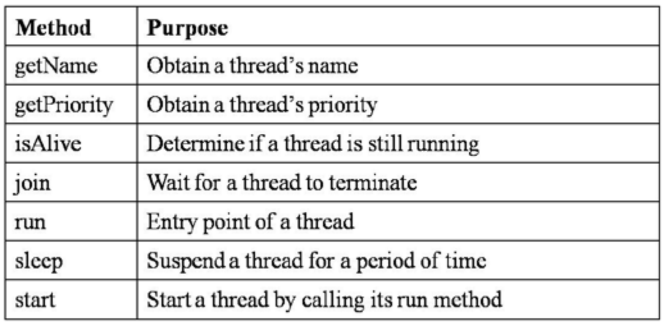
On a machine with single CPU threads cannot run in parallel exactly. Only one thread can run at a time. This kind of concurrency is called quasi-concurrency. But, on a machine with multiple cores or multiple CPUs, each CPU can run a single thread in parallel.

The Thread class and the Runnable Interface

Threads in Java are supported through Thread class and Runnable interface. A single thread is represented in a Java program as Thread class instance (object). To create a new thread in a Java program, we can either:

1. Extend the Thread class or
2. Implement the Runnable interface

The Thread class contains the following methods:



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

Main Thread

When a Java program is executed, by default it starts a single thread of execution. That thread is known as the "Main thread". The main thread is important due to two reasons:

- It is used to create child threads.
- It is the last thread to terminate during which it will perform various shutdown operations.

We can control the main thread by obtaining a reference to it using the Thread object as shown in the below program:

```
public class Driver
{
        public static void main(String[] args)
        {
                Thread mainThread = Thread.currentThread();
                System.out.println("Current thread: " + mainThread);
                mainThread.setName("My Thread");
                System.out.println("After name change: " + mainThread);
                try
                {
                        for(int i = 1; i <= 5; i++)
                        {
                                System.out.println("i = " + i);
                                Thread.sleep(1000);
                        }
                }
                catch(InterruptedException e)
                {
                        System.out.println("Thread is interrupted!");
                }
        }
}
```

In the above program main thread reference is obtained by using the currentThread() method in Thread class. The setName() method is used to assign a new name to the main thread. The new name assigned to the main thread is My Thread.

The sleep() method is used to pause the current thread (main thread) for certain amount of time which is specified in milliseconds. In the above program 1000 in the sleep() method pauses the thread for 1 second.

Output of the above program is:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
i = 1
i = 2
i = 3
i = 4
i = 5
```

In the above output, [main,5,main], first main refers to the name of the thread, 5 refers to the default priority of the main thread and last refers to the thread group name which is also main.

Syntax of some the methods used above are as follows:

```
static Thread currentThread()
static void sleep(long milliseconds) throws InterruptedException

final void setName(String threadName)

final String getName()
```

Thread Methods

isAlive() and join() Methods

Even though threads are independent most of the time, there might some instances at which we want a certain thread to wait until all other threads complete their execution. In such situations we can use isAlive() and join() methods. Syntax of these methods is as follows:

```
final boolean isAlive()

final void join() throws InterruptedException

final void join(long milliseconds) throws InterruptedException
```

The isAlive() method can be used to find whether the thread is still running or not. If the thread is still running, it will return true. Otherwise, it will return false.

The join() method makes the thread to wait until the invoking thread terminates. Below program demonstrates the use of isAlive() and join() methods:

```
class MyThread implements Runnable
{
	Thread t;
	MyThread()
	{
        t = new Thread(this);
        System.out.println("Child thread: " + t);
        t.start();
    }
    @Override
    public void run()
    {
        try
        {
            for(int i = 1; i <= 3; i++)
            {
                System.out.println(t.getName() + ": " + i);
                Thread.sleep(500);
            }
        }
        catch(InterruptedException e)
        {
            System.out.println("Child thread is interrupted!");
        }
        System.out.println("Child thread is terminated");
    }
}
public class Driver
{
    public static void main(String[] args)
```

```
	{
		MyThread my = new MyThread();
		try
		{
			System.out.println("Child thread is running: " + my.t.isAlive());
			System.out.println("Main thread is waiting...");
			my.t.join();
		}
		catch(InterruptedException e)
		{
			System.out.println("Main thread is interrupted!");
		}
		System.out.println("Child thread is running: " + my.t.isAlive());
		System.out.println("Main thread terminated");
	}
}
```

Output of the above program is:

```
Child thread: Thread[Thread-0,5,main]
Child thread is running: true
Main thread is waiting…
Thread-0: 1
Thread-0: 2
Thread-0: 3
Child thread is terminated
Child thread is running: false
Main thread terminated
```

In the above program, main thread is waiting for the child thread to complete its execution and then it terminates.

Suspending, Resuming and Stopping Threads

Based on the requirements sometimes you might want to suspend, resume or stop a thread. For doing such operations on a thread, before Java 2, thread API used to contain suspend(), resume() and stop() methods.

But all these methods might lead to undesired behavior of the program or machine. For example, these methods might lead to deadlock when a thread locked and is accessing a data structure and suddenly it is suspended or stopped. All other threads will be waiting indefinitely for gaining access to the data structure.

Because of this drawback of suspend(), resume() and stop() methods, they have been deprecated (should not be used) from Java 2 on wards.

Instead we have to check whether the thread is suspended or not through code itself. We can do that using a boolean variable which acts like a flag (ON / OFF).

Below program demonstrates the use of the boolean variable suspendFlag which is initially false. We write two more methods mySuspend() and myResume() which changes the value of the variable suspendFlag.

We have also used two more methods wait() and notify() from Object class. The wait() method suspends the current thread and the notify() method resumes (wakes up) the execution of the current thread.

```
class MyThread implements Runnable
{
        Thread t;
        boolean suspendFlag;
        MyThread(String name)
```

```
	{
		t = new Thread(this, name);
		System.out.println("Child thread: " + t);
		suspendFlag = false;
		t.start();
	}

	@Override
	public void run()
	{
		try
		{
			for(int i = 1; i <= 5; i++)
			{
				System.out.println(t.getName() + ": " + i);
				Thread.sleep(500);
				synchronized(this)
				{
					while(suspendFlag)
						wait();
				}
			}
		}
		catch(InterruptedException e)
		{
```

```
                System.out.println(t.getName() + " is interrupted!");
            }
            System.out.println(t.getName() + " is terminated");
        }

        public synchronized void mySuspend()
        {
            suspendFlag = true;
        }

        public synchronized void myResume()
        {
            suspendFlag = false;
            notify();
        }
}

public class Driver
{
        public static void main(String[] args)
        {
            MyThread thread1 = new MyThread("First Thread");
            MyThread thread2 = new MyThread("Second Thread");
            try
            {
```

```
                        System.out.println("Main thread is waiting...");
                        Thread.sleep(1000);
                        thread1.mySuspend();
                        System.out.println("---Suspending thread 1---");
                        Thread.sleep(1000);
                        thread1.myResume();
                        System.out.println("---Resuming thread 1---");
                        Thread.sleep(1000);
                        thread1.t.join();
                        thread2.t.join();
                }
                catch(InterruptedException e)
                {
                        System.out.println("Main thread is interrupted!");
                }
                System.out.println("Main thread terminated");
        }
}
```

Output of the above program is:

```
Child thread: Thread[First Thread,5,main]
Child thread: Thread[Second Thread,5,main]
Main thread is waiting…
First Thread: 1
Second Thread: 1
First Thread: 2
Second Thread: 2
—Suspending thread 1—
First Thread: 3
Second Thread: 3
Second Thread: 4
—Resuming thread 1—
First Thread: 4
Second Thread: 5
First Thread: 5
Second Thread is terminated
First Thread is terminated
Main thread terminated
```

In the above output you can observer that first thread is suspended for some time and has been resumed again.

Below program demonstrates stopping a thread:

```
class MyThread implements Runnable
{
	Thread t;
	private volatile boolean stopFlag = false;
	MyThread()
	{
		t = new Thread(this);
		t.start();
	}
	public void run()
	{
		while(!stopFlag)
		{
			System.out.println("Child thread running!");
			try
			{
```

```
                    Thread.sleep(200);
                }
                catch(InterruptedException e) {}
            }
            System.out.print("Child thread stopped!");
        }
        public void stop()
        {
            stopFlag = true;
        }
}
public class Driver
{
        public static void main(String[] args) throws InterruptedException
        {
            MyThread m = new MyThread();
            Thread.sleep(1000);
            m.stop();
        }
}
```

Output of the above program is:

```
Child thread running!
Child thread running!
Child thread running!
Child thread running!
Child thread running!
Child thread stopped!
```

Thread Priorities

In a uni-processor system, when several threads are competing for the CPU, you might want a certain thread to get more CPU time (burst time) over the remaining threads. We can use thread priorities in such situation.

The thread class provides three final static variables (constants) namely: MIN_PRIORITY, NORM_PRIORITY, and MAX_PRIORITY whose values are 1, 5 and 10 respectively. Priority values can be in the range 1 to 10. 1 denotes minimum priority and 10 denotes maximum priority.

A higher priority thread always gets more CPU time over the lesser priority thread. When a lesser priority thread is currently running on a CPU and a higher priority thread resumes (after wait or sleep or unblocked from I/O), it will preempt the lesser priority thread.

By default all the newly created threads will have a priority of 5. For assigning a new priority to the thread, we can use setPriority() method of Thread class whose syntax is as follows:

```
final void setPriority(int priority_level)
```

For retrieving the priority of the thread we can use getPriority() method of Thread class whose syntax is as follows:

```
final int getPriority()
```

Below program demonstrates the use of setPriority() and getPriority() methods:

```
class MyThread1 implements Runnable
{
	Thread t;
	MyThread1(String name)
	{
		t = new Thread(this, name);
		System.out.println("Child thread: " + t);
		t.start();
	}

	@Override
	public void run()
	{
		try
		{
			for(int i = 1; i <= 10; i++)
			{
				System.out.println(t.getName() + ": " + i);
				Thread.sleep(100);
			}
		}
		catch(InterruptedException e)
		{
			System.out.println(t.getName() + " is interrupted!");
		}
		System.out.println(t.getName() + " is terminated");
	}
}
```

```
class MyThread2 implements Runnable
{
	Thread t;
	MyThread2(String name)
	{
		t = new Thread(this, name);
		System.out.println("Child thread: " + t);
		t.start();
	}

	@Override
	public void run()
	{
		try
		{
			for(int i = 1; i <= 10; i++)
			{
				System.out.println(t.getName() + ": " + i);
				Thread.sleep(500);
			}
		}
		catch(InterruptedException e)
		{
			System.out.println(t.getName() + " is interrupted!");
		}
```

```
                System.out.println(t.getName() + " is terminated");
        }
}

public class Driver
{
        public static void main(String[] args)
        {
                MyThread1 thread1 = new MyThread1("First Thread");
                MyThread2 thread2 = new MyThread2("Second Thread");
                thread1.t.setPriority(10);
                thread2.t.setPriority(1);
                System.out.println("Priority of first thread is: " + thread1.t.getPriority());
                System.out.println("Priority of second thread is: " + thread2.t.getPriority());
                try
                {
                        System.out.println("Main thread is waiting...");
                        thread1.t.join();
                        thread2.t.join();
                }
                catch(InterruptedException e)
                {
                        System.out.println("Main thread is interrupted!");
                }
                System.out.println("Main thread terminated");
        }
}
```

Output of the above program is:

```
Child thread: Thread[First Thread,5,main]
First Thread: 1
Child thread: Thread[Second Thread,5,main]
Priority of first thread is: 10
Priority of second thread is: 1
Main thread is waiting…
Second Thread: 1
First Thread: 2
First Thread: 3
First Thread: 4
First Thread: 5
First Thread: 6
Second Thread: 2
First Thread: 7
First Thread: 8
First Thread: 9
First Thread: 10
Second Thread: 3
First Thread is terminated
Second Thread: 4
Second Thread: 5
Second Thread: 6
Second Thread: 7
Second Thread: 8
Second Thread: 9
Second Thread: 10
Second Thread is terminated
Main thread terminated
```

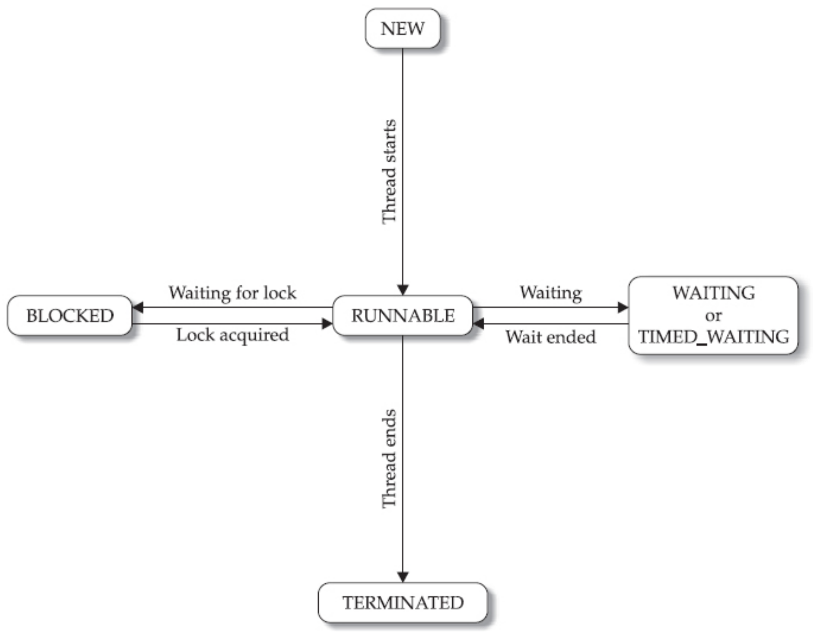
Although the sleeping time of first thread is less than the second thread, in general, thread with higher priority gets more CPU time.

Thread States

Life cycle of a thread refers to all the actions or activities done by a thread from its creation to termination. A thread can be in any one of the following five states during its life cycle:

- **New:** A thread is created but didn't begin its execution.
- **Runnable:** A thread that either is executing at present or that will execute when it gets the access to CPU.
- **Terminated:** A thread that has completed its execution.
- **Waiting:** A thread that is suspended because it is waiting for some action to occur. For example, it is waiting because of a call to non-timeout version of wait() method or join() method or sleep() method.
- **Blocked:** A thread that has suspended execution because it is waiting to acquire a lock or waiting for some I/O event to occur.

The life cycle of a thread is illustrated in the following diagram:



The state of a thread can be retrieved using the getState() method of the Thread class. The syntax of this method is as follows:

```
Thread.State getState()
```

The getState() method returns one of the values in State enumeration. The constants in State enumerations are:

1. NEW
2. RUNNABLE
3. TERMINATED
4. BLOCKED
5. WAITING
6. TIMED_WAITING

Below example demonstrates the use of getState() method:

```
class MyThread implements Runnable
{
        Thread t;
        MyThread(String name)
        {
                t = new Thread(this, name);
                System.out.println("Child thread: " + t);
                System.out.println("State: " + t.getState());
                t.start();
        }
        @Override
        public void run()
        {
                System.out.println("State: " + t.getState());
                try
                {
                        for(int i = 1; i <= 3; i++)
                        {
                                System.out.println(t.getName() + ": " + i);
                                Thread.sleep(2000);
                        }
                }
                catch(InterruptedException e)
                {
                        System.out.println(t.getName() + " is interrupted!");
                }
                System.out.println(t.getName() + " is terminated");
        }
}

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

```
        MyThread thread1 = new MyThread("First Thread");
        try
        {
            System.out.println("Main thread is waiting...");
            thread1.t.sleep(1000);
            System.out.println("State: " + thread1.t.getState());
            thread1.t.join();
            System.out.println("State: " + thread1.t.getState());
        }
        catch(InterruptedException e)
        {
            System.out.println("Main thread is interrupted!");
        }
        System.out.println("Main thread terminated");
    }
}
```

Output for the above program is:

```
Child thread: Thread[First Thread,5,main]
State: NEW
Main thread is waiting…
State: RUNNABLE
First Thread: 1
State: TIMED_WAITING
First Thread: 2
First Thread: 3
First Thread is terminated
State: TERMINATED
Main thread terminated
```

Daemon Threads

A daemon thread is a thread which runs in the background. Example for daemon thread in Java is the garbage collector. In Java, thread can be divided into two categories:

- User threads
- Daemon threads

A user thread is a general thread which is created by the user. A daemon thread is also a user thread which is made as a daemon thread (background thread).

Difference between a user thread and a daemon thread is, a Java program will not terminate if there is at least one user thread in execution. But, a Java program can terminate if there are one or more daemon threads in execution.

To work with daemon threads, Java provides two methods:

```
void setDaemon(boolean flag)
boolean isDaemon()
```

The setDaemon() method is used to convert a user thread into a daemon thread. The isDaemon() thread can be used to know whether a thread is a daemon thread or not.

Below program demonstrates a daemon thread:

```
class ChildThread implements Runnable
{
	Thread t;
	ChildThread(String name)
	{
		t = new Thread(this, name);
		t.setDaemon(true);
		t.start();
	}
	public void run()
	{
		try
		{
			while(true)
			{
				System.out.println("This thread is a daemon thread: 
"+t.isDaemon());
				System.out.println(t.getName()+": Hi");
				Thread.sleep(1000);
			}
		}
		catch(InterruptedException e)
		{
			System.out.println("Child thread is interrupted");
		}
	}
}
class DaemonThread
{
	public static void main(String args[])
```

```
        {
                ChildThread one = new ChildThread("Daemon Thread");
        }
}
```

Output of the above program is:

```
This thread is a daemon thread: true

Daemon Thread: Hi

Daemon Thread: Hi

Daemon Thread: Hi

Daemon Thread: Hi

…
```

Thread Groups

A thread group is a collection of threads. A thread group allows the programmer to maintain a group of threads more effectively. To support thread groups Java provides a class named ThreadGroup available in java.lang package.

Some of the constructors available in ThreadGroup class are:

ThreadGroup(String group-name)

ThreadGroup(ThreadGroup parent, String group-name)

After creating a thread group using one of the above constructors, we can add a thread to the thread group using one of the following Thread constructors:

Thread(ThreadGroup ref, Runnable obj)

Thread(ThreadGroup ref, String thread-name)

Thread(ThreadGroup ref, Runnable obj, String thread-name)

Following are some of the methods available in ThreadGroup class:

- **getName()** – To get the name of the thread group
- **setMaxPriority()** – To set the maximum priority of all the threads in the group
- **setMinPriority()** – To set the minimum priority of all the threads in the group
- **start()** – To start the execution of all the threads in the group
- **list()** – To print information of the thread group and the threads in the group.

Below program demonstrates the use of thread groups in Java:

```
class ChildThread implements Runnable
{
        Thread t;
        ChildThread(ThreadGroup g, String name)
        {
                t = new Thread(g,this,name);
                g.setMaxPriority(8);
                System.out.println(t.getName()+" belongs to the group: "+g.getName());
```

```
		System.out.println("Maximum priority of "+g.getName()+" is:
"+g.getMaxPriority());
		t.start();
	}
	public void run()
	{
		try
		{
			for(int i=1;i<=10;i++)
			{
				System.out.println(t.getName()+":"+i);
				Thread.sleep(1000);
			}
		}
		catch(InterruptedException e)
		{
			System.out.println(t.getName()+" is interrupted");
		}
	}
}

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

```
                ThreadGroup tg1 = new ThreadGroup("Group A");
                ChildThread one = new ChildThread(tg1, "First Thread");
                ChildThread two = new ChildThread(tg1, "Second Thread");
                try
                {
                        Thread.sleep(3000);
                        System.out.println("All the threads in the group will be stopped");
                        tg1.stop();
                }
                catch(InterruptedException e)
                {
                        System.out.println("main thread is interrupted");
                }
                tg1.list();
        }
}
```

Output of the above program is:

```
First Thread belongs to the group: Group A
Maximum priority of Group A is: 8
Second Thread belongs to the group: Group A
Maximum priority of Group A is: 8
First Thread:1
Second Thread:1
First Thread:2
Second Thread:2
Second Thread:3
First Thread:3
All the threads in the group will be stopped
java.lang.ThreadGroup[name=Group A,maxpri=8]
```

```
Thread[First Thread,5,Group A]
Thread[Second Thread,5,Group A]
```

Synchronization

Why Synchronization is Needed?

When two or more threads are accessing the same resource like a variable or a data structure, it may lead to inconsistent data or values. Such conditions that lead to inconsistency are known as race conditions.

As an example let's consider a variable count whose value is 7 at present. Consider two operations: count = count + 1 which is executed by thread1 and another operation count = count – 1 which is executed by thread2. Note that both threads are sharing the common variable count.

If both threads execute in parallel then the sequence of operations can be either:

```
count = count + 1

count = count – 1
```

or

```
count = count – 1

count = count + 1
```

In both sequences the end value of count will be 7 which is a consistent value. But often a single high-level language statement will be converted to multiple assembly language statements.

The count = count + 1 will be converted to following assembly language statements:

A: R1 = count

B: R1 = R1 + 1

C: count = R1

Statements are labelled as A, B and C for convenience. R1 is a CPU register. Similarly count = count – 1 will be converted to following assembly language statements:

D: R2 = count

E: R2 = R2 – 1

F: count = R2

Again statements are labelled as D, E and F for convenience. R2 is another CPU register. Statements A, B and C are executed by thread1 in parallel with statements D, E and F of thread2.

Let the value of count be 7. Now consider the following statement execution sequence: A, B, D, E, C, F as shown below:

A: R1 = count (R1 = 7)

B: R1 = R1 + 1 (R1 = 8)

D: R2 = count (R2 = 7)

E: R2 = R2 – 1 (R2 = 6)

C: count = R1 (count = 8)

F: count = R2 (count = 6)

End value of count after the above execution sequence is 6 which is an inconsistent value and the execution sequence can be considered as an example that led to race condition.

To prevent race conditions we can use a mechanism known as synchronization.

What is Synchronization?

Synchronization is a mechanism which restricts simultaneous access to a shared resource by multiple threads.

Synchronization is used at large in operating systems like Windows, Linux, MacOSX etc. No synchronization might lead to a deadlock which is a serious problem. In Java, synchronization is supported by the synchronized keyword.

Using synchronized keyword we can create:

- Synchronized methods
- Synchronized blocks

Synchronization in Java

The key to synchronization in Java is monitor. A monitor is an object which is used for obtaining a mutual exclusive lock. Once a thread acquires a lock, it is said to have entered the monitor. When one thread is inside the monitor, no other thread is allowed to acquire a lock until that thread exits the monitor.

Every object in Java has an implicit monitor associated with it. To enter an object's monitor, a method which is modified using synchronized keyword should be called.

Synchronized Methods

A method that is modified with the synchronized keyword is called a synchronized method. Syntax of a synchronized method is as follows:

```
synchronized return-type method-name(parameters)
{ … }
```

First let's consider a program which doesn't contain the synchronized method:

```
class MyName
{
        void printName()
        {
                try
                {
                        System.out.print("[My name ");
                        Thread.sleep(200);
                        System.out.print("is ");
                        Thread.sleep(500);
                        System.out.println("Suryateja]");
                }
                catch(Exception e)
                {
                        System.out.println("Thread Interrupted");
                }
        }
}

class MyThread implements Runnable
{
        Thread t;
        MyName myname;
        MyThread(MyName myname)
        {
                this.myname = myname;
                t = new Thread(this);
                t.start();
        }

        @Override
        public void run()
        {
                myname.printName();
        }
}

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

```
    {
        MyName myname = new MyName();
        MyThread thread1 = new MyThread(myname);
        MyThread thread2 = new MyThread(myname);
        MyThread thread3 = new MyThread(myname);
        try
        {
            thread1.t.join();
            thread2.t.join();
            thread3.t.join();
        }
        catch(InterruptedException e)
        {
            System.out.println("Main thread is interrupted!");
        }
        System.out.println("Main thread terminated");
    }
}
```

Output of the above program is:

```
[My name [My name [My name is is is Suryateja]
Suryateja]
Suryateja]
Main thread terminated
```

As you can observe in the above program, all three threads are executing the same method printName() simultaneously and hence the inconsistent output. For consistent output we can make the printName() method synchronized.

Below example demonstrates the use of synchronized methods in Java:

```
class MyName
{
	synchronized void printName()
	{
		try
		{
			System.out.print("[My name ");
			Thread.sleep(200);
			System.out.print("is ");
			Thread.sleep(500);
			System.out.println("Suryateja]");
		}
		catch(Exception e)
		{
			System.out.println("Thread Interrupted");
		}
	}
}

class MyThread implements Runnable
{
	Thread t;
	MyName myname;
	MyThread(MyName myname)
	{
		this.myname = myname;
		t = new Thread(this);
		t.start();
	}

	@Override
	public void run()
	{
		myname.printName();
	}
}

public class Driver
{
	public static void main(String[] args)
	{
		MyName myname = new MyName();
		MyThread thread1 = new MyThread(myname);
		MyThread thread2 = new MyThread(myname);
		MyThread thread3 = new MyThread(myname);
		try
		{
			thread1.t.join();
			thread2.t.join();
```

```
                    thread3.t.join();
            }
            catch(InterruptedException e)
            {
                    System.out.println("Main thread is interrupted!");
            }
            System.out.println("Main thread terminated");
        }
}
```

Output of the above program is:

```
[My name is Suryateja]
[My name is Suryateja]
[My name is Suryateja]
Main thread terminated
```

In the above example synchronized method is printName() of MyName class.

Synchronized Blocks

Sometimes while working with third-party classes, we might want to synchronize the access to methods in those classes. In such cases, synchronized methods cannot be used. Instead we can use synchronized blocks. Syntax of a synchronized block is as follows:

```
synchronized(object-reference)

{   //Statements to be synchronized }
```

The above program can be modified to contain a synchronized block as shown below:

```
class MyName

{
	void printName()
	{
		try
		{
			System.out.print("[My name ");
			Thread.sleep(200);
			System.out.print("is ");
			Thread.sleep(500);
			System.out.println("Suryateja]");
		}
		catch(Exception e)
		{
			System.out.println("Thread Interrupted");
		}
	}
}

class MyThread implements Runnable
{
	Thread t;

	MyName myname;

	MyThread(MyName myname)

	{

		this.myname = myname;

		t = new Thread(this);

		t.start();

	}



	@Override

	public void run()

	{

		synchronized(myname)

		{

			myname.printName();

		}

	}

}
```

```
public class Driver
{
	public static void main(String[] args)
	{
		MyName myname = new MyName();
		MyThread thread1 = new MyThread(myname);
		MyThread thread2 = new MyThread(myname);
		MyThread thread3 = new MyThread(myname);
		try
		{
			thread1.t.join();
			thread2.t.join();
			thread3.t.join();
		}
		catch(InterruptedException e)
		{
			System.out.println("Main thread is interrupted!");
		}
		System.out.println("Main thread terminated");
```

```
        }
}
```

Output of the above program is:

```
[My name is Suryateja]
[My name is Suryateja]
[My name is Suryateja]
Main thread terminated
```

In the above program the call to the method printName() is written inside a synchronized block.

Inter Thread Communication

Although we can restrict the access of data or code to a single thread at a time by using synchronization, it can't guarantee the consistent execution of our logic.

As an example let's consider the conventional Producer-Consumer problem which is generally introduced in Operating Systems course. In this problem Producer process or thread produces items and stores in a queue and Consumer process or thread consumes items from the queue. Note that the Consumer thread should wait until the Producer produces at least one item.

In our example we will maintain a class Q which contains a single variable n. The Producer thread stores item (integer) into n and Consumer thread retrieves the item (integer) from n.

Consumer thread must wait until the Producer places an item in n and Producer should wait until the Consumer retrieves the item in n. This kind of inter thread communication can be achieved using the following pre-defined methods in Object class:

```
final void wait() throws InterruptedException

final void notify()

final void notifyAll()
```

Consider the following example Java Program which provides solution to Producer-Consumer problem:

```
class Q
{
	int n;
	synchronized void put(int n)
	{
		this.n = n;
		System.out.println("Put: " + n);
	}
	synchronized int get()
	{
		System.out.println("Got: " + n);
		return n;
	}
}

class Producer implements Runnable
```

```
{
        Q q;
        Producer(Q q)
        {
                this.q = q;
                new Thread(this, "Producer").start();
        }
        public void run()
        {
                int i = 0;
                while(true)
                {
                        q.put(i++);
                }
        }
}

class Consumer implements Runnable
{
        Q q;
        Consumer(Q q)
        {
                this.q = q;
                new Thread(this, "Consumer").start();
        }
        public void run()
        {
                while(true)
                {
                        q.get();
                }
        }
}

public class Driver
{
        public static void main(String[] args)
        {
                Q q = new Q();
                Producer p = new Producer(q);
                Consumer c = new Consumer(q);
        }
}
```

Although the put() and get() methods are sychronized, the output of the above program looks like this:

```
Put: 1

Put: 2

Put: 3

Got: 3
```

```
Got: 3

Got: 3

Put: 4

Put: 5

Put: 6

Got: 6

Got: 6

…
```

To make the Producer wait until Consumer retrieves the item and Consumer wait until Producer places an item, we can use the wait() and notify() methods as shown in the below program:

```
class Q
{
	int n;
	boolean statusFlag = false;
	synchronized void put(int n)
	{
		try
		{
			while(statusFlag)
			{
				wait();
			}
```

```
                }
                catch(InterruptedException e){}
                this.n = n;
                System.out.println("Put: " + n);
                statusFlag = true;
                notify();
        }
        synchronized int get()
        {
                try
                {
                        while(!statusFlag)
                        {
                                wait();
                        }
                }
                catch(InterruptedException e){}
                statusFlag = false;
                System.out.println("Got: " + n);
                notify();
                return n;
        }
}

class Producer implements Runnable
```

```
{
        Q q;
        Producer(Q q)
        {
                this.q = q;
                new Thread(this, "Producer").start();
        }
        public void run()
        {
                int i = 0;
                while(true)
                {
                        q.put(i++);
                }
        }
}

class Consumer implements Runnable
{
        Q q;
        Consumer(Q q)
        {
                this.q = q;
                new Thread(this, "Consumer").start();
        }
        public void run()
        {
                while(true)
                {
                        q.get();
                }
        }
}

public class Driver
{
        public static void main(String[] args)
        {
                Q q = new Q();
                Producer p = new Producer(q);
                Consumer c = new Consumer(q);
        }
}
```

Output of the above program is:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
...
```