OBJECT ORIENTED DESIGN AND PROGRAMMING

Presentation Material			
Department of Computer Science & Engineering			
Course Code:	20CS2402	Semester:	IV
Course Title:	OBJECT ORIENTED DESIGN AND PROGRAMMING	Year:	II
Faculty Name:	Prof. Pavithra. K		

OBJECT ORIENTED DESIGN AND PROGRAMMING

MULTI-THREADED PROGRAMMING

MODULE 3
MULTI-THREADED PROGRAMMING

Syllabus

MULTI-THREADED PROGRAMMING:

Multi-Threaded Programming: Java Thread Model; The main Thread; Creating a thread and multiple threads; Extending threads; Implementing Runnable; Synchronization; Inter Thread Communication; producer consumer problem.

INPUT/OUTPUT:

I/O Basic; Reading console input Writing Console output.

MULTI-THREADED PROGRAMMING

Java provides built-in support for multithreaded programming.

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus , multithreading is a specialized form of multitasking.

MULTITASKING: Multitasking is performing two or more tasks at the same time . Nearly all operating systems are capable of multitasking.

Multitasking techniques are mainly of 2 types

- Process Based Multitasking (Multiprocessing)
- Thread Based Multitasking(Multithreading)

MULTITASKING

- A multitasking operating system is an operating system that gives the perception of 2 or more tasks/jobs/processes running at the same time.
- It does this by dividing system resources amongst the tasks/jobs/processes and switching between the tasks/jobs/processes while they are executing over and over again.
- Usually CPU processes only one task at a time but the switching is so fast that it looks like CPU is executing multiple processes at a time.

PROCESS BASED MULTITASKING

- A process is a program executing within its own address space . A process consists of the memory space allocated by the operating system.Thus, process-based multitasking is the feature that allows computer to run two or more programs concurrently.

Example : process-based multitasking enables one to run the Java compiler at the same time while using a text editor or visiting a web site. All the three programs are running concurrently.

- In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

Disadvantages:

- A process is heavyweight.
- Cost of communication between the process is high/expensive.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.
- unable to gain access over idle time of CPU.

THREAD BASED MULTITASKING

- Thread-based multitasking is a single program performing more than one task at the same time.
- In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously.
- Example: Using a browser one 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.
- Also in a word-processing application like MS Word, we can type text in one thread and spell checker checks for mistakes in another thread.
- Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.

THREAD BASED MULTITASKING

- Thread based multitasking requires less overhead.
- Threads share same address space.
- Thread to Thread communication is not expensive.
- It allows taking gain access over idle time taken by CPU.
- It is comparatively light weight.
- It has faster data rate multi-tasking.

COMPARISION OF PROCESS-BASED AND THREAD-BASED MULTITASKING

- Java programs make use of process-based multitasking environments, process-based multitasking is not under Java's control. However, multithreaded multitasking is.

PROCESS-BASED MULTITASKING	THREAD-BASED MULTITASKING
In process based multitasking two or more processes and programs can be run concurrently.	In thread based multitasking two or more threads can be run concurrently.
In process based multitasking a process or a program is the smallest unit.	In thread based multitasking a thread is the smallest unit.
Process based multitasking requires more overhead.	Thread based multitasking requires less overhead.
Process requires its own address space.	Threads share same address space.
Process to Process communication is expensive	Thread to Thread communication is not expensive.
unable to gain access over idle time of CPU.	gain access over idle time taken by CPU.
It is comparatively heavy weight.	It is comparatively light weight.
It has slower data rate multi-tasking.	It has faster data rate multi-tasking.

ADVANTAGES OF THREAD-BASED MULTITASKING

- Multithreading enables to write efficient programs that make maximum use of the processing power available in the system. One important way multithreading achieves this is by keeping idle time to a minimum.

Example:

- Transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than they can be processed by the CPU. user input is much slower than the computer. In a single-threaded environment , program has to wait for each of these tasks to finish before it can proceed to the next one and most of the time the program is idle or waiting for input.

ADVANTAGES OF THREAD-BASED MULTITASKING

- Multithreading helps to reduce this idle time because another thread can run when one is waiting.

Example :

- While one part of the program is sending a file over the Internet , another part can be reading keyboard input and still another can be buffering the next block of data to send.

ADVANTAGES OF THREAD-BASED MULTITASKING

Dayananda Sagar University Bengaluru

- Most commercial applications use multi-threading extensively. This is done for several reasons:
- **For faster processing of background/batch tasks:** When multiple tasks must be performed simultaneously, multi-threading allows the different tasks to proceed in parallel. The overall processing time is reduced as a result.
- **To take advantage of modern processors:** Most modern systems have multiple processors and each processor has multiple cores. Multi-threading allows different threads to be run by different processors, thereby allowing the system resources to be more efficiently used.

ADVANTAGES OF THREAD-BASED MULTITASKING

Dayananda Sagar University Bengaluru

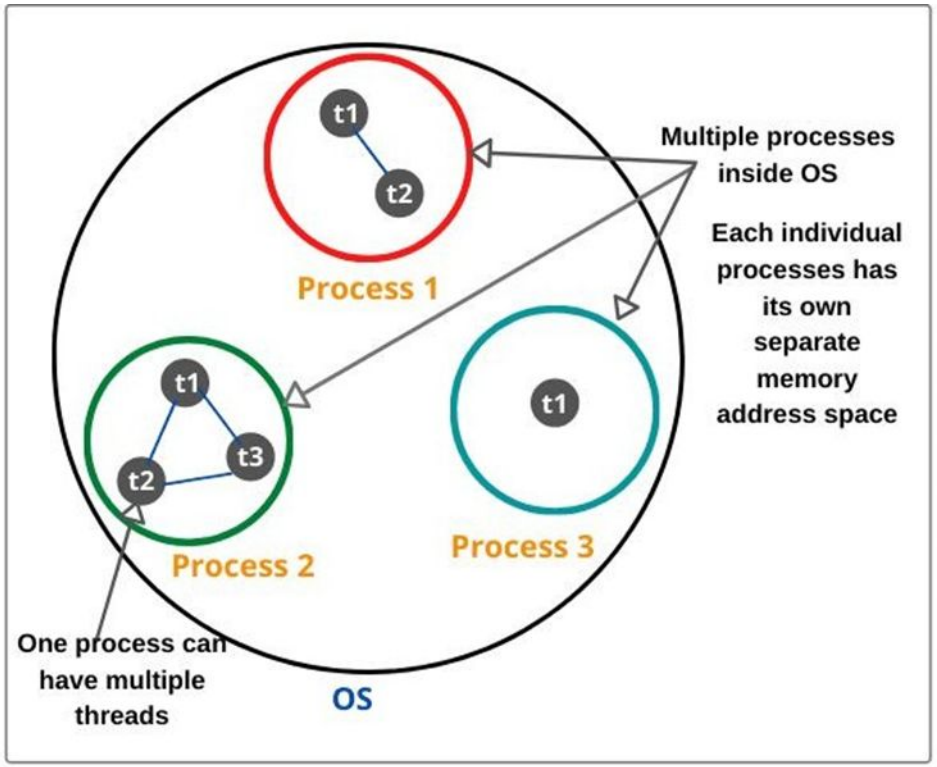
- Most commercial applications use multi-threading extensively. This is done for several reasons:
  - **For reducing response times: Users** expect applications to be fast. By breaking the processing needed for a request into smaller chunks and having different threads handle the processing in parallel, response time can be reduced.
  - **To serve multiple users at the same time:** Application servers like Tomcat, JBoss, Oracle WebLogic and IBM WebSphere are expected to support thousands of users in parallel. Multi-threading is the only way this can be achieved. One thread is spawned by the application server for each request to be handled.

THE JAVA THREAD MODEL

- A thread is a lightweight sub-process, the smallest unit of processing It is a part of the program that is running . A thread is a single sequential flow of control within a process.
- It is a separate path of execution . Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.
- Multiprocessing and multithreading, both are used to achieve multitasking .

THE JAVA THREAD MODEL

- However, multithreading is used than multiprocessing because threads use a shared memory area.
- They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- Java Multithreading is mostly used in games, animation, etc.



THE JAVA THREAD MODEL

- The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading.
- Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.
- SINGLE-THREADED SYSTEMS
- Single-threaded systems use an approach called an event loop with polling. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.

THE JAVA THREAD MODEL

- Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the program.
- This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed.
- In general, in a single-threaded environment, when a thread blocks (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

THE JAVA THREAD MODEL

JAVA MULTI-THREADED SYSTEMS

- 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.

Example

- The idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere.
- When a thread blocks in a Java program, only the single thread that is blocked pauses. All other thread continue to run.

THE JAVA THREAD MODEL

NOTE:

- Java's multithreading features work in both single and multicore systems. In a single core system, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time. Therefore, in a single-core system, two or more threads do not actually run at the same time, but idle CPU time is utilized. However, in multi-core systems, it is possible for two or more threads to actually execute simultaneously. In many cases, this can further improve program efficiency and increase the speed of certain operations.

THREAD LIFECYCLE

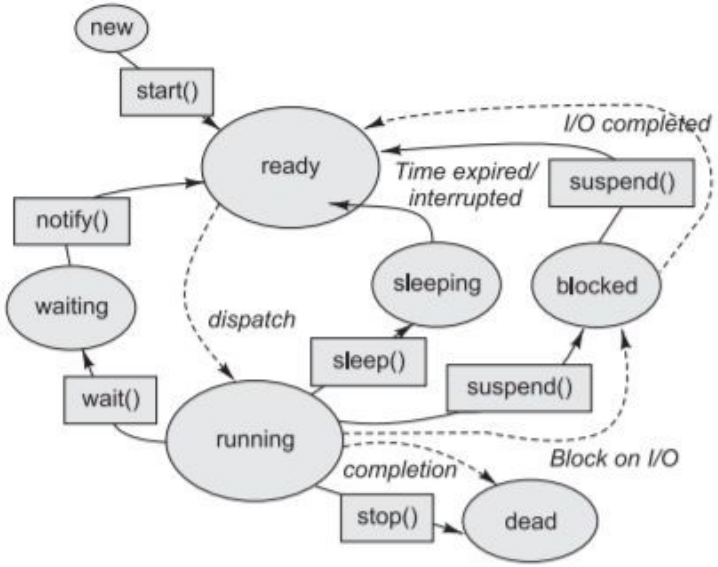
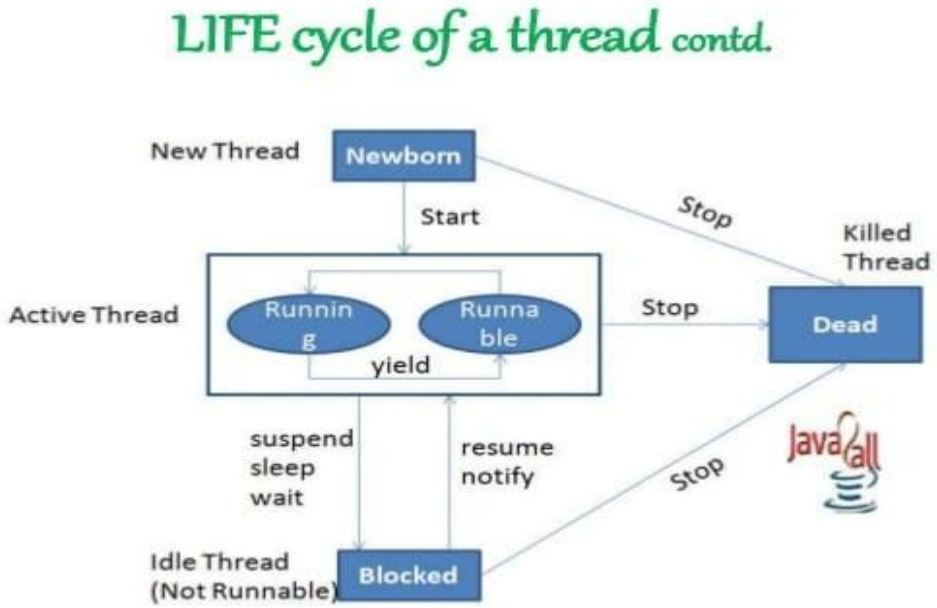


Fig. 14.4 Life cycle of Java threads



THREAD LIFECYCLE

1.Newborn State:

The thread is born and is said to be in new born state.

- The thread is not yet scheduled for running.
- At this state, we can do only one of the following:
- Schedule it for running using start() method.
- Kill it using stop() method.

2.Runnable State:

- The thread is ready for execution
- Waiting for the availability of the processor.

The thread has joined the queue.

3. Running State:

- Thread is executing . The processor has given its time to the thread for its execution.
- The thread runs until it gives up control on its own or taken over by other threads.

4.Blocked State:

- A thread is said to be blocked when it is prevented from entering into the runnable and the running state.
- This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements.
- A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again..
- This state is achieved when we Invoke suspend() or sleep() or wait() methods.

5.Dead State:

- Every thread has a life cycle.
- A running thread ends its life when it has completed executing its run() method. It is a natural death.
- A thread can be killed in born, or in running, or even in "not runnable" (blocked) condition. . It is called premature death.
- This state is achieved when stop() method is invoked or the thread completes it execution.

THREAD LIFECYCLE

- At any given time, a thread can be in only one state. These states are JVM states as they are not linked to operating system thread states.
- When the object of a user Thread class is created, the thread moves to the NEW state. After invocation of the start() method, the thread shifts from the NEW to the ready (RUNNABLE) state and is then dispatched to running state by the JVM thread scheduler . After gaining a chance to execute, the run() method will be invoked.
- Depending on program operations or invocation of methods such as wait(), sleep(), and suspend() or an I/O operation, the thread moves to the WAITING, SLEEPING, and BLOCKED states respectively.

THREAD LIFECYCLE

- It should be noted that the thread is still alive. After the completion of an operation that blocked it (I/O or sleep) or receiving an external signal that wakes it up, the thread moves to ready state.
- Then the JVM thread scheduler moves it to running state in order to continue the execution of remaining operations. When the thread completes its execution, it will be moved to TERMINATED state. A dead thread can never enter any other state, not even if the start() method is invoked on it.

THREAD PRIORITIES

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.
- Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running.
- A thread's priority is used to decide when to switch from one running thread to the next. This is called a context switch.

THREAD PRIORITIES

The rules that determine when a context switch takes place are

- A thread can voluntarily relinquish control. This is done by explicitly yielding, sleeping , or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
- A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing—by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called preemptive multitasking.

THREAD PRIORITIES

NOTE: In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated.

For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion.

For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

- Each thread is assigned a priority, which affects the order in which it is scheduled for running

THREAD PRIORITIES

Java permits us to set the priority of a thread using the setPriority() method as follows:

ThreadName.setPriority(int Number);

- The int Number is an integer value to which the thread's priority is set.

The Thread class defines several priority constants:

1.public static int MIN_PRIORITY= 1

2.public static int NORM_PRIORITY=5

3. public static int MAX_PRIORITY = 10

- The default setting is NORM_PRIORITY. Most user level processes should use NORM_PRIORITY.

SYNCHRONIZATION

- Multithreading introduces an asynchronous behavior to the programs, there must be a way to enforce synchronicity when it is needed.
- **For example :** if two threads communicate and share a complicated data structure, such as a linked list ,there must be some way to ensure that they don't conflict with each other. That is , one must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the monitor.

SYNCHRONIZATION

- The monitor is a control mechanism first defined by C.A.R.Hoare. Monitors can be thought as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.
- Most multithreaded systems expose monitors as objects that program must explicitly acquire and manipulate. Java provides a cleaner solution. There is no class “Monitor”; instead, each object has its own implicit monitor that is automatically entered when one of the object’s synchronized methods is called.
- Once a thread is inside a synchronized method, no other thread can call any other synchronized method on thesame object. This enables to write very clear and concise multithreaded code, because synchronization support is built into the language.

MESSAGING

- After the program is divided into separate threads, one need to define how they will communicate with each other. When programming with some other languages, one must depend on the operating system to establish communication between threads. This, of course, adds overhead.
- By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

TYPES OF THREADS IN JAVA

Java offers two types of threads: user threads and daemon threads.

- User threads are high-priority threads created by the programmer. The JVM will wait for any user thread to complete its task before terminating it.
- On the other hand, daemon threads are low-priority threads created by the JVM whose only role is to provide services to user threads . Daemon threads are usually designed to run in the background for the purpose of servicing the user thread.

TYPES OF THREADS IN JAVA

- Example: Garbage collector thread

The main thread in the user thread made available by the JVM .

This thread is launched in the main method . From main thread all the other methods are created.

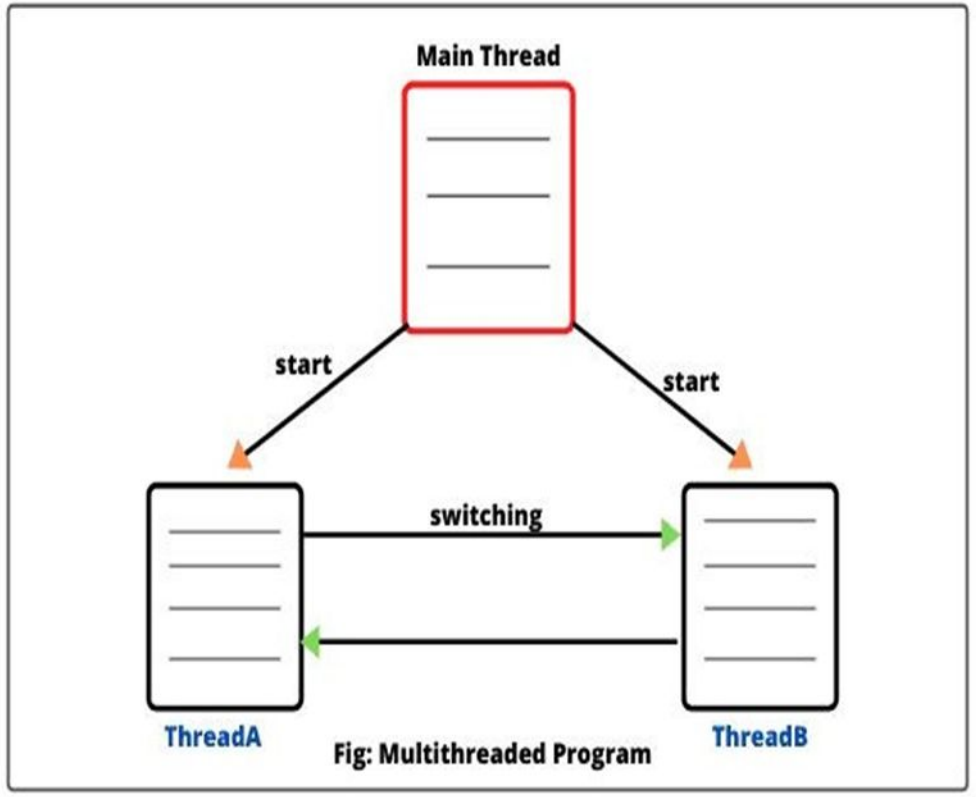


Fig: Multithreaded Program

CREATING THREADS

- A thread in Java is represented by an object of thread class .
- The threads in java can be created in two ways

1.By extending the Thread class

2.By implementing the Runnable interface

- The run() and start() are the 2 inbuilt methods that are used for implementing threads in java.

CREATING THREAD BY EXTENDING THREAD CLASS

CREATING THREAD Contd.

1. By Extending Thread class

```
class Multi extends Thread                          // Extending thread class
{
public void run()                                   // run() method declared
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi t1=new Multi();                    //object initiated
t1.start();                              // run() method called through start()
 }
}
```

Output: thread is running...

CREATING THREAD BY IMPLEMENTING RUNNABLE INTERFACE

CREATING THREAD Contd.

2. By implementing Runnable interface

```
class Multi3 implements Runnable                    // Implementing Runnable interface
{
public void run()
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi3 m1=new Multi3();                              // object initiated for class
Thread t1 =new Thread(m1);                           // object initiated for thread
t1.start();
 } }
```

Output: thread is running...

THREAD METHODS

Modifier and Type	Method	Description
void	start()	It is used to start the execution of the thread.
void	run()	It is used to do an action for a thread.
static void	sleep()	It sleeps a thread for the specified amount of time.
static Thread	currentThread()	It returns a reference to the currently executing thread object.
void	join()	It waits for a thread to die.
int	getPriority()	It returns the priority of the thread.
void	setPriority()	It changes the priority of the thread.
String	getName()	It returns the name of the thread.

THREAD METHODS

void	setName()	It changes the name of the thread
long	getId()	It returns the id of the thread.
boolean	isAlive()	It tests if the thread is alive.
static void	yield()	It causes the currently executing thread object to pause and allow other threads to execute temporarily.
void	suspend()	It is used to suspend the thread.
void	resume()	It is used to resume the suspended thread.
void	stop()	It is used to stop the thread.
void	destroy()	It is used to destroy the thread group and all of its subgroups.
boolean	isDaemon()	It tests if the thread is a daemon thread.
void	setDaemon()	It marks the thread as daemon or user thread.

THE MAIN THREAD

- When a Java program starts up, one thread begins running immediately. This is usually called the main thread of the program, because it is the one that is executed when the program begins. The main thread is important for two reasons:
- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.
- Although the main thread is created automatically when the program is started, it can be controlled through a Thread object. To do so, one must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread.

THE MAIN THREAD

Its general form is shown here:

static Thread currentThread( )

- This method returns a reference to the thread in which it is called. Once the reference to the main thread is obtained, one can control it just like any other thread.

CREATING MULTIPLE THREADS

- So far, only two threads are being used: the main thread and one child thread . However, a program can spawn as many threads as it needs.
- Basically, when one need to perform several tasks at a time, one can create multiple threads to perform multiple tasks in a program.

SYNCHRONIZATION IN JAVA

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

The synchronization is mainly used to

- To prevent thread interference.
- To prevent data inconsistency/Race condition problem.
- There are two types of synchronization
- Process Synchronization
- Thread Synchronization

THREAD SYNCHRONIZATION IN JAVA

- There are two types of thread synchronization mutual exclusive and inter-thread communication.
- Mutual Exclusion

1. Synchronized method.
2. Synchronized block.
3. Static synchronization.

- Cooperation (Inter-thread communication in java)

MUTUAL EXCLUSION

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

- By Using Synchronized Method(object level lock)
- By Using Synchronized Block(object level lock)
- By Using Static Synchronization(class level lock)

MUTUAL EXCLUSION

Concept of Lock in Java

- Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

UNDERSTANDING THE PROBLEM WITHOUT SYNCHRONIZATION

- In this example, there is no synchronization, so output is inconsistent.

```
class Table
{
        void printTable(int n)//method not synchronized
        {
                 for(int i=1;i<=5;i++)
                {
                         System.out.println(n*i);
                         try
                        {
                                 Thread.sleep(400);
                        }
                        catch(Exception e)
                        {
                                System.out.println(e);
                        }
                 }

        }
}
```

```
class MyThread1 extends Thread
{
        Table t;
        MyThread1(Table t)
        {
                this.t=t;
        }
        public void run()
        {
                t.printTable(5);
        }

}
```

UNDERSTANDING THE PROBLEM WITHOUT SYNCHRONIZATION

```
class MyThread2 extends Thread
{
        Table t;
        MyThread2(Table t)
        {
                this.t=t;
        }
        public void run()
        {
                t.printTable(100);
        }
}
```

```
class TestSynchronization1
{
        public static void main(String args[])
        {
                Table obj = new Table();//only one object
                MyThread1 t1=new MyThread1(obj);
                MyThread2 t2=new MyThread2(obj);
                t1.start();
                t2.start();
}
}
```

```
C:\Users\Shwetha G S\Desktop\javaexampleprograms>java TestSynchronization1
5
100
10
200
15
300
20
400
25
500
```

JAVA SYNCHRONIZED METHOD

- Any method that is declared as synchronized , is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.
- NOTE : Synchronized keyword cannot be used with the predefined methods ,since run is a pre-defined method in Thread class , it is not possible to use synchronized keyword with run method.

JAVA SYNCHRONIZED METHOD

```
class Table
{
        public synchronized void printTable(int n)//method synchronized
        {
                 for(int i=1;i<=5;i++)
                {
                         System.out.println(n*i);
                         try
                        {
                                 Thread.sleep(400);
                        }
                        catch(Exception e)
                        {
                                System.out.println(e);
                        }
                 }

        }
}
```

```
class MyThread1 extends Thread
{
        Table t;
        MyThread1(Table t)
        {
                this.t=t;
        }
        public void run()
        {
                t.printTable(5);
        }

}
```

JAVA SYNCHRONIZED METHOD

```
class MyThread2 extends Thread
{
        Table t;
        MyThread2(Table t)
        {
                this.t=t;
        }
        public void run()
        {
                t.printTable(100);
        }
}
```

```
class TestSynchronization1
{
        public static void main(String args[])
        {
                Table obj = new Table();//only one object
                MyThread1 t1=new MyThread1(obj);
                MyThread2 t2=new MyThread2(obj);
                t1.start();
                t2.start();
}
}
```

```
C:\Users\Shwetha G S\Desktop\javaexampleprograms>java TestSynchronization1
5
10
15
20
25
100
200
300
400
500
```

SYNCHRONIZED BLOCK IN JAVA

- Synchronized block can be used to perform synchronization on any specific resource of the method.
- Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized method.

POINTS TO REMEMBER

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.
- A Java synchronized block doesn't allow more than one , to provide access control to a shared resource.

SYNCHRONIZED BLOCK IN JAVA

- The system performance may degrade because of the slower working of synchronized keyword.
- Java synchronized block is more efficient than Java synchronized method.

Syntax

```
synchronized (object reference expression)
{
//code block
}
```

SYNCHRONIZED BLOCK IN JAVA


```
class Table
{
        void printTable(int n)//method synchronized
        {
                synchronized(this)
                {
                for(int i=1;i<=5;i++)
                {
                         System.out.println(n*i);
                         try
                        {
                                Thread.sleep(400);
                        }
                        catch(Exception e)
                        {
                                System.out.println(e);
                        }
                 }
                }

        }
}
```

```
class MyThread1 extends Thread
{
        Table t;
        MyThread1(Table t)
        {
                this.t=t;
        }
        public void run()
        {
                t.printTable(5);
        }

}
```

SYNCHRONIZED BLOCK IN JAVA

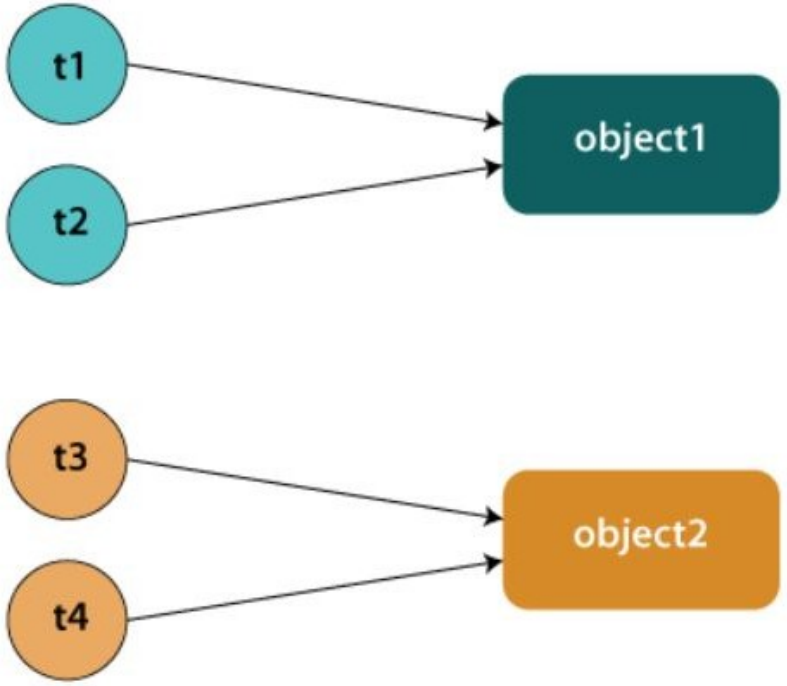
```
class MyThread2 extends Thread
{
        Table t;
        MyThread2(Table t)
        {
                this.t=t;
        }
        public void run()
        {
                t.printTable(100);
        }
}
```

```
class TestSynchronization1
{
        public static void main(String args[])
        {
                Table obj = new Table();//only one object
                MyThread1 t1=new MyThread1(obj);
                MyThread2 t2=new MyThread2(obj);
                t1.start();
                t2.start();
}
}
```

```
C:\Users\Shwetha G S\Desktop\javaexampleprograms>java TestSynchronization1
100
200
300
400
500
5
10
15
20
25
```

PROBLEM WITHOUT STATIC SYNCHRONIZATION

- In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock.
- But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. We don't want interference between t1 and t3 or t2 and t4.
- Static synchronization solves this problem.



STATIC SYNCHRONIZATION

- The execution of thread becomes execution of one thread for each instance if a method of the instance is synchronized but when there are more than once instance of the same class, it becomes a problem which requires synchronization at the class level for providing only one lock for all the instances of the class than having synchronization at the object level and this is called static synchronization in Java which can be performed in two ways, one is by having a static synchronized method and the other one is having a synchronized block of code within the static method.
- if static method is synchronized, then the lock will be on the class not on object.

Syntax:

```
synchronized static return_type class_name{}
```

STATIC SYNCHRONIZATION

```
class Table
{
        synchronized static void printTable(int n)
        {
                for(int i=1;i<=10;i++)
                {
                         System.out.println(n*i);
                         try
                        {
                                Thread.sleep(400);
                        }
                        catch(Exception e){}
                }
        }
}
class MyThread1 extends Thread
{
        public void run()
        {
                Table.printTable(1);
        }
}
```

```
class MyThread2 extends Thread
{
        public void run()
        {
                Table.printTable(10);
        }
}
class MyThread3 extends Thread
{
        public void run()
        {
                Table.printTable(100);
        }
}
```

STATIC SYNCHRONIZATION

```
class MyThread4 extends Thread
{
        public void run()
        {
                Table.printTable(1000);
        }
}
public class TestSynchronization4
{
        public static void main(String t[])
        {
                MyThread1 t1=new MyThread1();
                MyThread2 t2=new MyThread2();
                MyThread3 t3=new MyThread3();
                MyThread4 t4=new MyThread4();
                t1.start();
                t2.start();
                t3.start();
                t4.start();
        }
}
```

```
1
2
3
4
5
6
7
8
9
10
10
20
30
40
50
60
70
80
90
100
100
200
```

INTER-THREAD COMMUNICATION IN JAVA

- In Mutual exclusion some threads unconditionally blocked other threads from asynchronous access to certain methods. This use of the implicit monitors in Java objects is powerful, but one can achieve a more subtle level of control through interprocess communication.
- Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.
- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed .

INTER-THREAD COMMUNICATION IN JAVA

Dayananda Sagar University Bengaluru

It is implemented by following methods of Object class:

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

These methods can be called only from within a synchronized method or synchronized block of code otherwise, an exception named IllegalMonitorStateException is thrown.

All these methods are declared as final. Since it throws a checked exception, therefore, you must be used these methods within Java try-catch block.

WAIT() METHOD

- The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
- The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
▪ public final void wait()throws InterruptedException	▪ It waits until object is notified.
▪ public final void wait(longtimeout)throws InterruptedException	▪ It waits for the specified amount of time.

NOTIFY() METHOD &NOTIFYALL() METHOD

- The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

public final void notify() NOTIFYALL()

- Wakes up all threads that are waiting on this object's monitor.

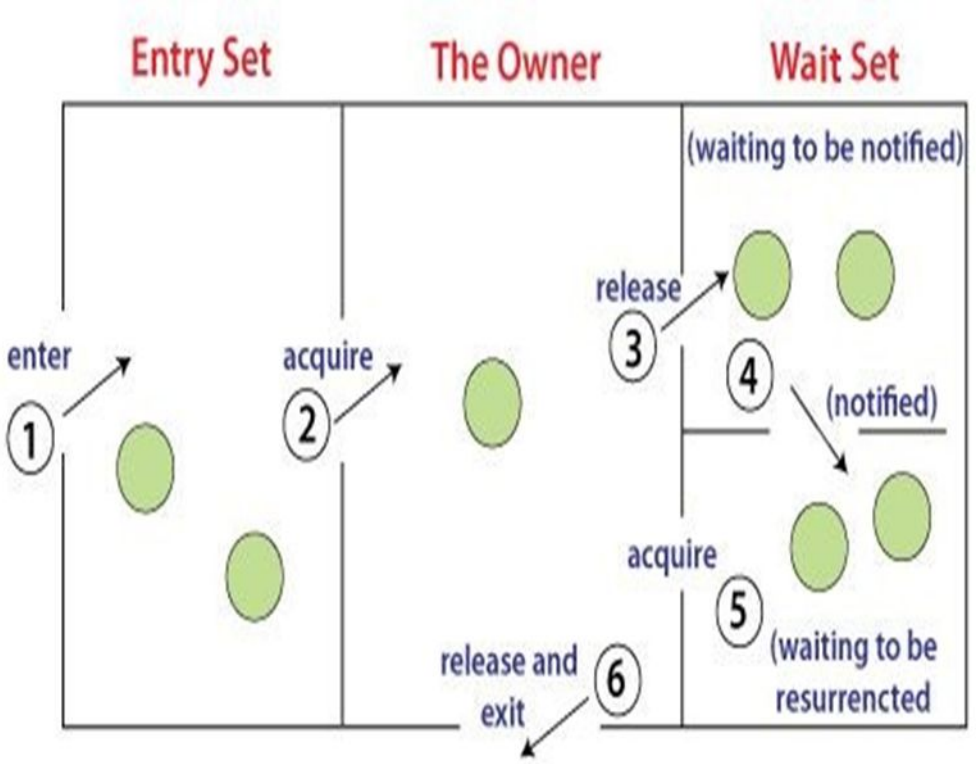
Syntax:

public final void notifyAll()

UNDERSTANDING THE PROCESS OF INTER-THREAD COMMUNICATION

- Threads enter to acquire lock.
- Lock is acquired by one thread.
- Now thread goes to waiting state if one call wait() method on the object. Otherwise it releases the lock and exits.
- If one call notify() or notifyAll() method, thread moves to the notified state (runnable state).
- Now thread is available to acquire lock.
- After completion of the task, thread releases the lock and exits the monitor state of the object.

UNDERSTANDING THE PROCESS OF INTER-THREAD COMMUNICATION



Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

DIFFERENCE BETWEEN WAIT AND SLEEP

wait()	sleep()
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.

INTER-THREAD COMMUNICATION

Two or more threads can communicate with each other using

(i) wait()
(ii) notify()
(iii) notifyAll()

WHY ?

(i) polling: To check some condition repeatedly to take appropriate action, once the condition is true. [wastes CPU time]

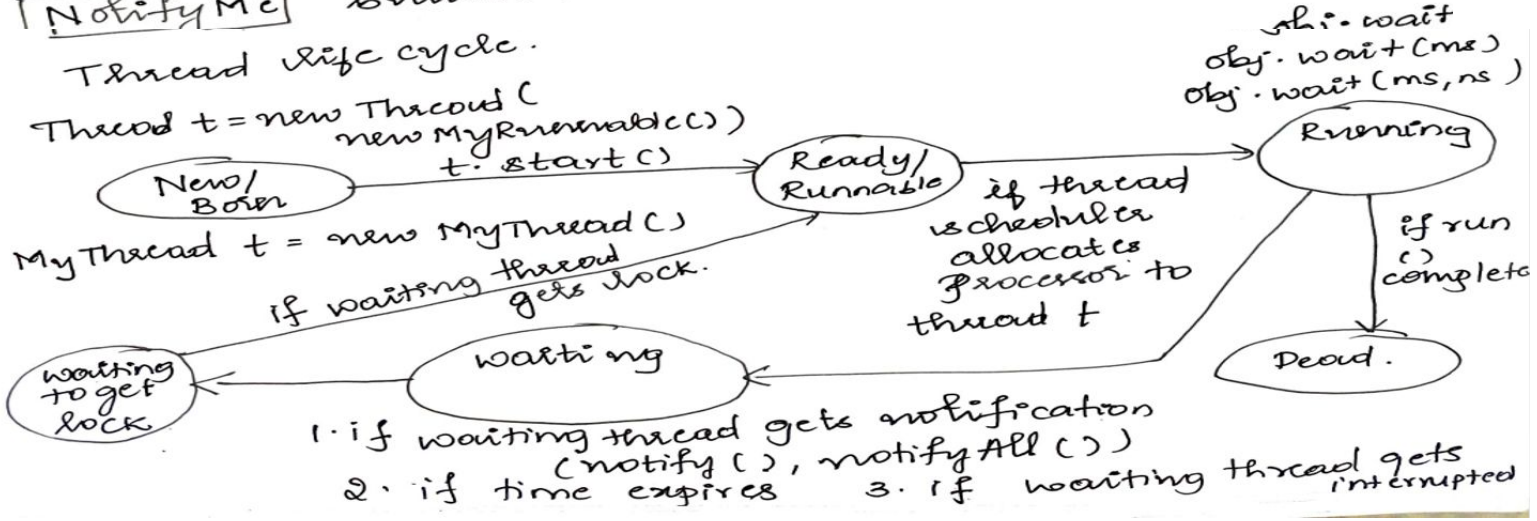
(ii) To reduce the wastage of CPU time due to polling.

Example: Buying item from e-commerce website.

If the item is out of stock then the buyer needs to keep checking if the item is available or not wasting user resources.

NotifyMe button.

Thread life cycle.



wait()
notify()
notifyAll()
} methods of object class and not thread class.

1. wait()

If any thread calls the wait() method, it causes the current thread to release the lock and wait until another thread invokes the notify() or notifyAll() method for this object / or a specified amount of time has elapsed.

Syntax:

public final void wait() throws InterruptedException.
(waits until object is notified)

public final void wait(long timeout) throws InterruptedException.
↓
waits for the specified amount of time.

2. notify(): This method is used to wake up a single thread and releases the object lock.

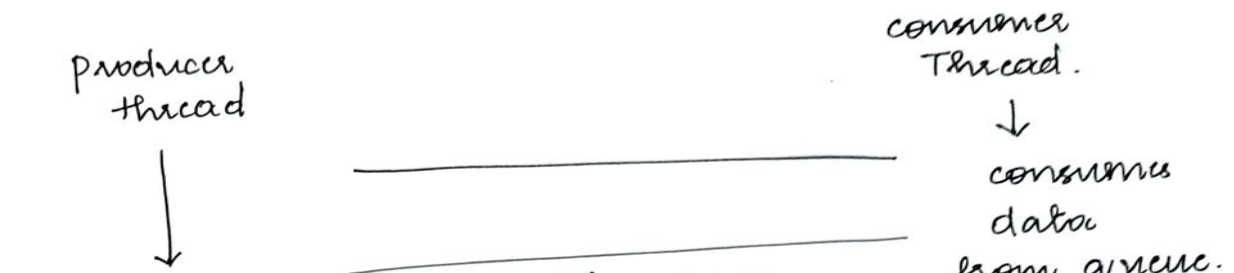
Syntax: public final void notify()

notifyAll(): This method is used to wake up all threads that are in waiting state.

Syntax: public final void notifyAll()

NOTE: To call wait(), notify() or notifyAll() method on any object, thread should own the lock of that object i.e. the thread should be inside synchronized area.

PRODUCER- CONSUMER PROBLEM.



creates data & puts inside the queue

The producer can produce maximum of n items.

once the queue is full, the producer has to wait until the space is created in queue.

Boundary condition.

(*) the producer should not produce when the queue is full.

(*) consumer should not consume when the queue is empty.

```
Class Queue
{
        int item;
        void produce(int x)
        {
                item = x;
        }
        void consume()
        {
                System.out.println(item + "consumed");
        }
}
class producer extends Thread
{
        Queue Q;
        producer(Queue Q)
        {       this.Q = Q;
        }
        public void run()
        {       for(int i=1; i<11; i++)
            {       System.out.println(i+"being produced");
                    Q.produce(i);
            }
        }
}
Class consumer extends Thread.
{
        Queue Q;
        consumer(Queue Q)
        {       this.Q = Q;
        }
}
```

```
public void run()
{
        for (int i=1; i<11; i++)
        {
                Q.consume();
        }
}
}
public class producerconsumerproblem
{
        public static void main(String args[])
        {
                Queue Q = new Queue();
                producer p = new producer(Q);
                Consumer C = new consumer(Q);
                P.start()
                Q.start()
        }
```

O/p::

Make (produce & consume methods as synchro -nized methods.

We are unable to control over the storing & consuming the item one by one.

However, synchronization helped to entee only one thread at a time to the Queue object.

But when to allow to enter producer into (Queue) godown to store the item. & when to allow to enter consumer to consume the item from the Queue is not in our control.

They use Queue object in mutually exclusive manner. But here we need to control over the allocation of the Queue object to particular thread.

WHAT TO DO NOW.

1. we need to allow the producer
   (*) to enter into Queue only if there is no item.
   (*) put item into the Queue &
   (*) send notification to other thread that the item is ready to be consumed.
   (*) otherwise wait itself for confirmation from consumer.

1. Allow the consumer
only if there is an item in the Queue to
be consumed.
2. consume the item
3. Send notification to other thread that
the Queue is empty.
4. otherwise wait for the confirmation from
producer.

Inter-thread communication is used to gain
exact control over the execution.

if flag is low means that the godown is empty & let the producer produce item & make flag high.

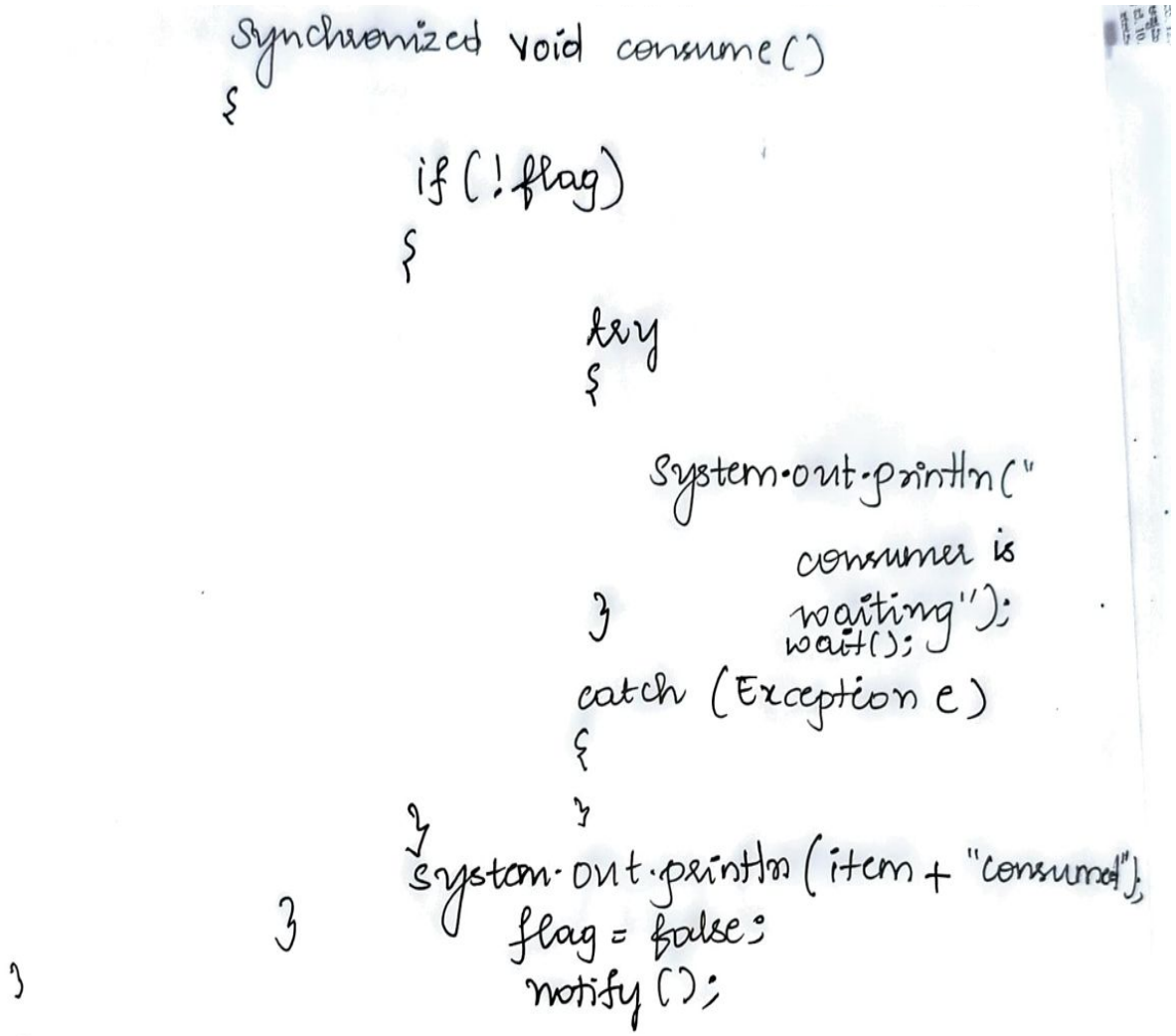
since flag is high, the producer will wait for its state to be false (low)

consumer

if flag is high, consumer will consume the item & make the flag low.

since flag is low, consumer will wait for its state to be (true/high)

```
class Queue
{
    int item;
    boolean flag = false;
    Synchronized void produce (int x)
    {
        if (flag)
        {
            try
            { System.out.println("producer is waiting");
              wait();
            }
            catch (Exception e)
            {
            }
        }
        item = x
        flag = true
        notify();
    }
```



```
Synchronized void consume()
{
        if(!flag)
        {
                try
                {
                        System.out.println("
                                consumer is
                        }       waiting");
                        wait();
                catch (Exception e)
                {
                }
        }
        System.out.println(item + "consumed");
        flag = false;
        notify();
}
```

Input and Output

Java read input from console

By default, to read from system console, we can use the Console class. This class provides methods to access the character-based console, if any, associated with the current Java process. To get access to Console, call the method System.console().

Console gives three ways to read the input:

String readLine() – reads a single line of text from the console.

char[] readPassword() – reads a password or encrypted text from the console with echoing disabled

Reader reader() – retrieves the Reader object associated with this console. This reader is supposed to be used by sophisticated applications.

Java program to read console input with readLine()

Dayananda Sagar University Bengaluru

Console readLine() method example

```
Console console = System.console();

if(console == null) {
  System.out.println("Console is not available to
current JVM process");
  return;
}
String userName = console.readLine("Enter the
username: ");
System.out.println("Entered username: " + userName);
```

Program output

Console

```
Enter the username: lokesh
Entered username: lokesh
```

Java print output to console

The easiest way to write the output data to console is System.out.println() statements. Still, we can use printf() methods to write formatted text to console.

Java program to write to console with System.out.println

System.out.println() method example

```
System.out.println("Hello, world!");
```

Program output

Console

```
Hello, world!
```

Java print output to console

Dayananda Sagar University Bengaluru

Java program to write to console with printf()

The printf(String format, Object... args) method takes an output string and multiple parameters which are substituted in the given string to produce the formatted output content. This formatted output is written in the console.

Console printf() method example

```
String name = "Lokesh";
int age = 38;

console.printf("My name is %s and my age is
%d", name, age);
```

Program output

Console

```
My name is Lokesh and my age is 38
```