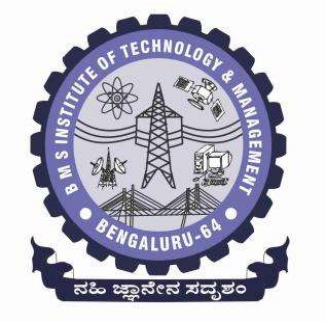
B M S INSTITUTE OF TECHNOLOGY & MANAGEMENT

YELAHANKA, BENGALURU – 560064.



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MODULE -4 NOTES OF

OBJECT ORIENTED CONCEPTS -18CS45

[As per Choice Based Credit System (CBCS) scheme]

(Effective from the academic year 2018 -2019)

SEMESTER – IV

Prepared by,

Mr. Muneshwara M S

Asst. Prof, Dept. of CSE

VISION AND MISSION OF THE CS&E DEPARTMENT

Vision

To develop technical professionals acquainted with recent trends and technologies of computer science to serve as valuable resource for the nation/society.

Mission:

Facilitating and exposing the students to various learning opportunities through dedicated academic teaching, guidance and monitoring.

VISION AND MISSION OF THE INSTITUTE

Vision

To emerge as one of the finest technical institutions of higher learning, to develop engineering professionals who are technically competent, ethical and environment friendly for betterment of the society.

Mission

Accomplish stimulating learning environment through high quality academic instruction, innovation and industry-institute interface

MODULE 4 PACKAGES AND INTERFACES

The following concepts should be learn in this Module

Packages, Access Protection,Importing Packages.Interfaces. Multi Threaded Programming:Multi Threaded Programming: What are threads? How to make the classes threadable ; Extending threads; Implementing runnable; Synchronization; Changing state of the thread; Bounded buffer problems, producer consumer problems.

Text book 2: CH: 9 Ch 11: RBT: L1, L2, L3

Packages:

Packages in Java is a mechanism to encapsulate a group of classes, interfaces and sub packages.To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name.

This is the general form of the **package** statement:

package ***pkg*;**

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.

You can create a hierarchy of packages.

package ***pkg1*[*.pkg2*[*.pkg3*]];**

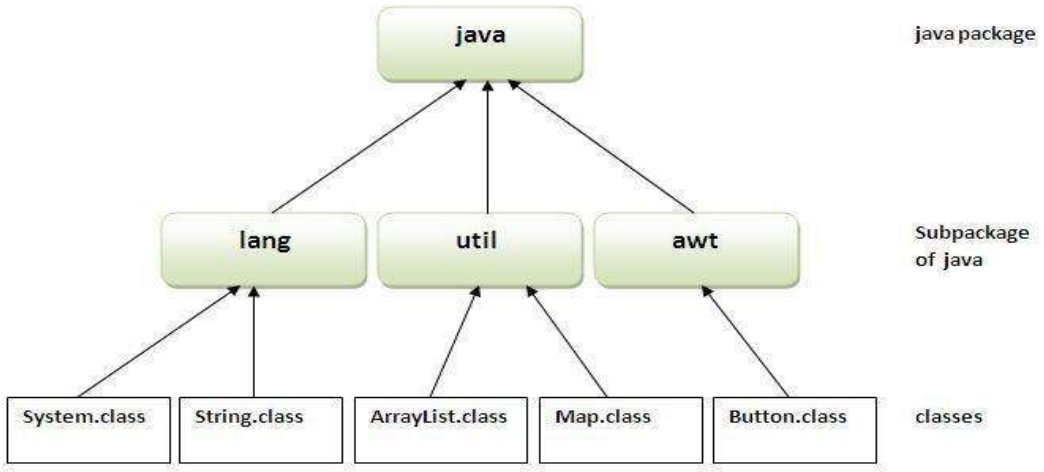
Ex: package java.awt.image;

Advantages of using a package

- ***Reusability:*** Reusability of code is one of the most important requirements in the software industry. Reusability saves time, effort and also ensures consistency. A class once developed can be reused by any number of programs wishing to incorporate the class in that particular program.
- ***Easy*** to locate the files.
- In real life situation there may arise scenarios where we need to define files of the same name. This may lead to "name-space collisions". Packages are a way of avoiding "name-space collisions".

Package are categorized into two forms

- **Built-in Package**:-Existing Java package for example java.lang, java.util etc.
- **User-defined-package:-** Java package created by user to categorized classes and interface



How to compile & Run java package?

If you are not using any IDE, you need to follow this:

Compile :- javac -d . Simple.java

Run :- java mypack.Simple

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not sub packages. The import keyword is used to make the classes and interface of another package accessible to the current package.

```
//save by A.java

package pack;

public class A{

  public void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

import pack.*;

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

A obj = new A();

obj.msg();

}

}
```

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

//save by A.java

```
package pack;

public class A{

public void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

import pack.A;

class B{

public static void main(String args[]){

A obj = new A();

obj.msg();

}

}
```

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

```
//save by A.java

package pack;

public class A{

public void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

class B{

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

```
pack.A obj = new pack.A();//using fully qualified name

obj.msg();}}
```

Access Specifiers

- private: accessible only in the class
- default : so-called “package” access — accessible only in the same package
- protected: accessible (inherited) by subclasses, and accessible by code in same package
- public: accessible anywhere the class is accessible, and inherited by subclasses

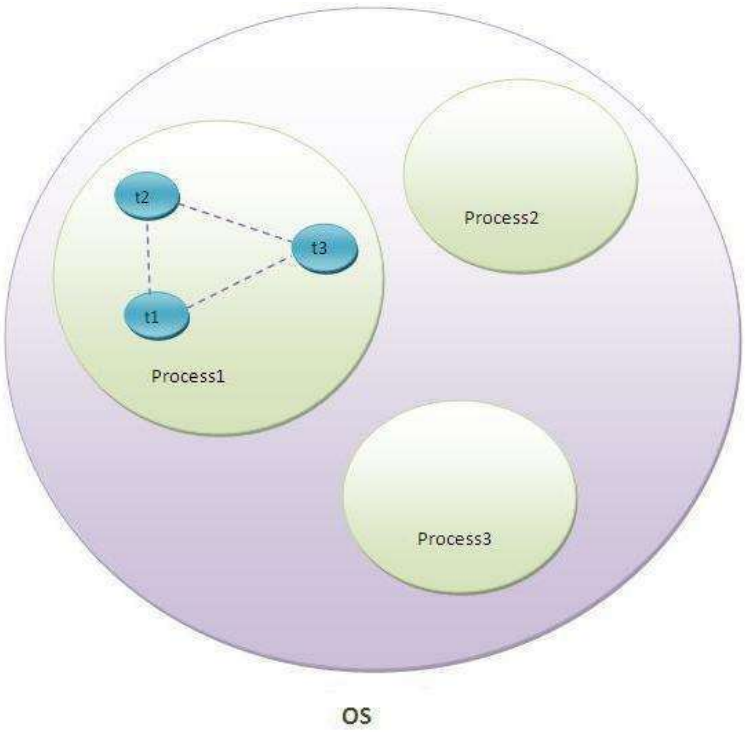
Notice that private protected is not syntactically legal.

Access By	private	package	protected	public
the class itself	yes	yes	yes	yes
a subclass in same package	no	yes	yes	yes
non-subclass in same package	no	yes	yes	yes
a subclass in other package	no	no	yes	yes
non-subclass in other package	no	no	no	yes

Multi-Threaded Programming

Thread:

- A thread is a lightweight sub process, a smallest unit of processing.
- It is a separate path of execution.
- Threads are independent, if there occurs exception in one thread, it doesn't affect other threads.
- It shares a common memory area.



t1, t2, t3 are threads

Java provides built-in support for multithreaded programming. **The process of executing multiple threads simultaneously is known as multithreading.** 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 a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

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

1) Process-based Multitasking (Multiprocessing)

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the processes is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.
- Less efficient

2) Thread-based Multitasking (Multithreading)

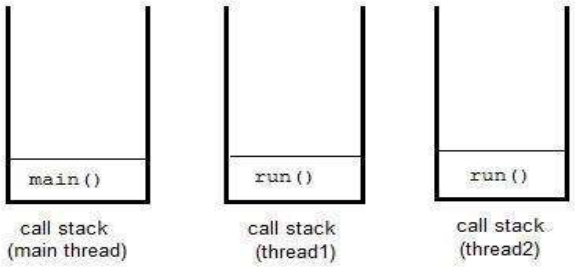
- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the threads is low.
- Highly efficient

Multithreading has several advantages over Multiprocessing such as;

- Threads are lightweight compared to processes
- Threads share the same address space and therefore can share both data and code.

- Context switching between threads is usually less expensive than between processes
- Cost of thread intercommunication is relatively low that that of process intercommunication
- Threads allow different tasks to be performed concurrently.

An instance of Thread class is just an object, like any other object in java. But a thread of execution means an individual "lightweight" process that has its own call stack. In java each thread has its own call stack



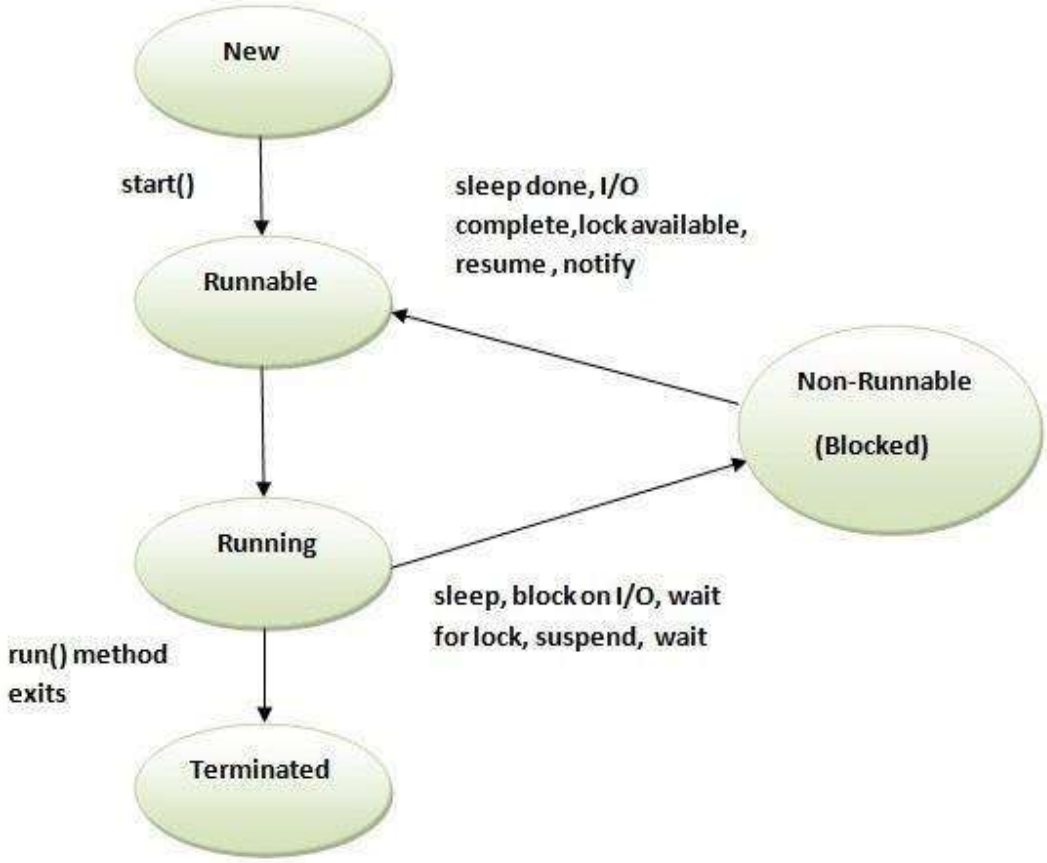
The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your 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.

Life cycle of a Thread (Thread States)



The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated

1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

When we call start() function on Thread object, it's state is changed to Runnable. The control is given to Thread scheduler to finish it's execution. Whether to run this thread instantly or keep it in runnable thread pool before running, depends on the OS implementation of thread scheduler.

3) Running

When thread is executing, it's state is changed to Running. Thread scheduler picks one of the thread from the runnable thread pool and change its state to Running. Then CPU starts executing this thread. A thread can change state to Runnable, Dead or Blocked from running state depends on time slicing, thread completion of run() method or waiting for some resources.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run. Waiting − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state, only when another thread signals the waiting thread to continue its execution.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

Thread Creation

When the thread starts it schedules the time slots for several sub threads and the JVM scheduler schedules the time slot to every thread based on round robin technique or priority based.

Method Signature	**Description**
String getName()	Retrieves the name of running thread in the current context in String format
void start()	This method will start a new thread of execution by calling run() method of Thread/runnable object.
void run()	This method is the entry point of the thread. Execution of thread starts from this method.
void sleep(int sleeptime)	This method suspend the thread for mentioned time duration in argument (sleeptime in ms)
void yield()	By invoking this method the current thread pause its execution temporarily and allow other threads to execute.
void join()	This method used to queue up a thread in execution. Once called on thread, current thread will wait till calling thread completes its execution

There are two ways to create thread in java;

- Implement the Runnable interface (java.lang.Runnable)
- By Extending the Thread class (java.lang.Thread)

Extending Thread class

Creates a thread by a new class that extends Thread class. This creates an instance of that class. The extending class must override run() method which is the entry point of new thread.

```
class Multi extends Thread{

public void run(){

System.out.println("thread is running...");

}

public static void main(String args[]){

Multi t1=new Multi();

t1.start();

 }

}
```

thread is running

Implementing the Runnable Interface

The easiest way to create a thread is to create a class that implements the runnable interface. After implementing runnable interface, the class needs to implement the **run()** method, which is of form,

public void run()

- run() method introduces a concurrent thread into your program. This thread will end when run() returns.
- You must specify the code for your thread inside run() method.
- run() method can call other methods, can use other classes and declare variables just like any other normal method.

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

Syntax :

Thread(Runnable threadOb, String threadName);

In this constructor, threadOb is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName.

Ex : ***Thread t = new Thread(mt);***

```
class MyThread implements Runnable
{
 public void run()

 {
  System.out.println("concurrent thread started running..");
 }
}
```

```
class MyThreadDemo
{
 public static void main( String args[] )
 {
  MyThread mt = new MyThread();
  Thread t = new Thread(mt);
  t.start();
 }
```

concurrent thread started running..

Creating Multiple Threads by implementing Runnable Interface

```
package applet1;
import java.io.*;
import java.util.*;
class A implements Runnable {
  public void run() {
    try {
      for (int i = 5; i > 0; i--) {
        System.out.println(i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println( "Interrupted");
    }
```

```
    System.out.println(" exiting.");
  }
}
class B implements Runnable {

        public void run() {
          try {
            for (int i = 15; i > 10; i--) {
              System.out.println(i);
              Thread.sleep(1000);
            }
          } catch (InterruptedException e) {
            System.out.println("Interrupted");
          }
          System.out.println(" exiting.");
        }
      }
class multithread {
  public static void main(String args[]) {
    A obj1 = new A();
    B obj2= new B();
    Thread t1=new Thread(obj1);
    Thread t2=new Thread(obj2);
    t1.start();
    t2.start();
  }}
```

5

15

14

4

13

3

12

2

11

1

exiting.

exiting.

Notice the call to sleep(10000) in main( ). This causes threads B and A to sleep for ten seconds and ensures that it will finish last.

Note : Write a java application program for generating 3 threads to perform the following operations.

i)Reading n numbers ii) Printing prime numbers iii) Computing average of n numbers

isAlive() and join() methods

In java, isAlive() and join() are two different methods to check whether a thread has finished its execution.

The isAlive() method returns true if the thread upon which it is called is still running otherwise it returns false.

final boolean isAlive()

```
public class MyThread extends Thread
{
        public void run()
        {
                System.out.println("r1 ");
                try {
                Thread.sleep(500);
                }
                catch(InterruptedException ie) { }
                System.out.println("r2 ");
        }
        public static void main(String[] args)
        {
                MyThread t1=new MyThread();
                MyThread t2=new MyThread();
                t1.start();
                t2.start();
```

```
                System.out.println(t1.isAlive());
                System.out.println(t2.isAlive());
        }
}
```

Output :

r1

true

true

r1

r2

r2

But, join() method is used more commonly than isAlive(). This method waits until the thread on which it is called terminates

final void join() throws InterruptedException

Using join() method, we tell our thread to wait until the specified thread completes its execution. There are overloaded versions of join() method, which allows us to specify time for which you want to wait for the specified thread to terminate.

```
public class MyThread extends Thread
{
        public void run()
        {
                System.out.println("r1 ");
                try {
                Thread.sleep(500);
```

```
            }catch(InterruptedException ie){ }
            System.out.println("r2 ");

        }
        public static void main(String[] args)
        {
            MyThread t1=new MyThread();
            MyThread t2=new MyThread();
            t1.start();

            try{
                t1.join();              //Waiting for t1 to finish
            }catch(InterruptedException ie){}

            t2.start();
        }
}
```

r1

r2

r1

r2

In this above program join() method on thread t1 ensures that t1 finishes it process before thread t2 starts.

Specifying time with join()

If in the above program, we specify time while using join() with t1, then t1 will execute for that time, and then t2 will join it.

```
t1.join(1500);
```

Synchronization

At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is

called synchronization. The synchronization keyword in java creates a block of code referred to as critical section.

Key to synchronization is the concept of the **monitor**. A **monitor** is an object that is used as a mutually exclusive lock. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor.

Why use Synchronization

The synchronization is mainly used to

- To prevent thread interference.
- To prevent consistency problem.

Thread Synchronization & Mutual Exclusive

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

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. static synchronization.

Using Synchronized Blocks

Synchronized block can be used to perform synchronization on any specific resource of the method. Synchronized block is used to lock an object for any shared resource. Scope of synchronized block is smaller than the method.

General Syntax :

```
synchronized (object)
{
//statement to be synchronized

}
```

```
class Table{

 void display(int n)
{

  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);
                }
```

```
    }
   }
  }//end of the method
}

class A extends Thread
{
Table t;

        A(Table t)
        {
        this.t=t;
        }
public void run(){
        t.display(5);
        }

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

```
        }
}

public class synch
{
public static void main(String args[])
{
Table obj = new Table();      //only one object

A t1=new A(obj);
B t2=new B(obj);
t1.start();
t2.start();
}
}
```

```
5
10
15
20
25
100
200
300
400
500
```

Using Synchronized Methods

If you declare any method as synchronized, it 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.

```
class Table{


 synchronized void display(int n)
{
  for(int i=1;i<=5;i++)
        {
                System.out.println(n*i);
                try{
                 Thread.sleep(400);
                }
                catch(Exception e){
                System.out.println(e);
                }
    }
   }
  }//end of the method



class A extends Thread
{
Table t;
```

```
        A(Table t)
        {
        this.t=t;
        }
public void run(){
        t.display(5);
        }

}
class B extends Thread{
Table t;
        B(Table t){
        this.t=t;
        }
public void run(){
        t.display(100);
}
}

public class synch
{
public static void main(String args[])
{
Table obj = new Table();      //only one object
A t1=new A(obj);
```

```
B t2=new B(obj);
t1.start();

t2.start();
}
}
```

5

10

15

20

25

100

200

300

400

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

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

Solution:

This problem can be implemented or solved by different ways in Java, classical way is using **wait** and **notify** method to communicate between Producer and Consumer thread and blocking each of them on individual condition like full queue and empty queue.

wait( ) tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ) or notifyAll( ).

notify( ) wakes up a thread that called wait( ) on the same object.

notifyAll( ) wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access.

These methods are declared within Object, as shown here:

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

```
public class ProducerConsumerTest {
  public static void main(String[] args) {
    Myclass c = new Myclass();
    Producer p1 = new Producer(c, 1);
    Consumer c1 = new Consumer(c, 1);
    p1.start();

    c1.start();
  }
}
```

```
class Myclass {
  private int contents;
  private boolean available = false;

  public synchronized int get() {
    while (available == false) {
      try {
        wait();
      } catch (InterruptedException e) {}
    }
    available = false;
    notifyAll();
    return contents;
  }
  public synchronized void put(int value) {
    while (available == true) {
      try {
        wait();
      } catch (InterruptedException e) { }
    }
    contents = value;
    available = true;
    notifyAll();

  }
}
class Consumer extends Thread {
```

```
  private Myclass Myclass;
  private int number;

  public Consumer(Myclass c, int number) {
    Myclass = c;

    this.number = number;
  }
  public void run() {
    int value = 0;
    for (int i = 0; i < 10; i++) {
      value = Myclass.get();
      System.out.println("Consumer #" + this.number + " got: " + value);
    }
  }
}
class Producer extends Thread {
  private Myclass Myclass;
  private int number;
  public Producer(Myclass c, int number) {
    Myclass = c;
    this.number = number;

  }
  public void run() {
    for (int i = 0; i < 10; i++) {
      Myclass.put(i);
```

```
        System.out.println("Producer #" + this.number + " put: " + i); try
        {
          sleep((int)(Math.random() * 100));
        } catch (InterruptedException e) { }
      }
    }}
```

Producer #1 put: 0

Consumer #1 got: 0

Producer #1 put: 1

Consumer #1 got: 1

Producer #1 put: 2

Consumer #1 got: 2

Producer #1 put: 3

Consumer #1 got: 3

Producer #1 put: 4

Consumer #1 got: 4

Producer #1 put: 5

Consumer #1 got: 5

Producer #1 put: 6

Consumer #1 got: 6

Producer #1 put: 7

Consumer #1 got: 7

Producer #1 put: 8

Consumer #1 got: 8

Producer #1 put: 9

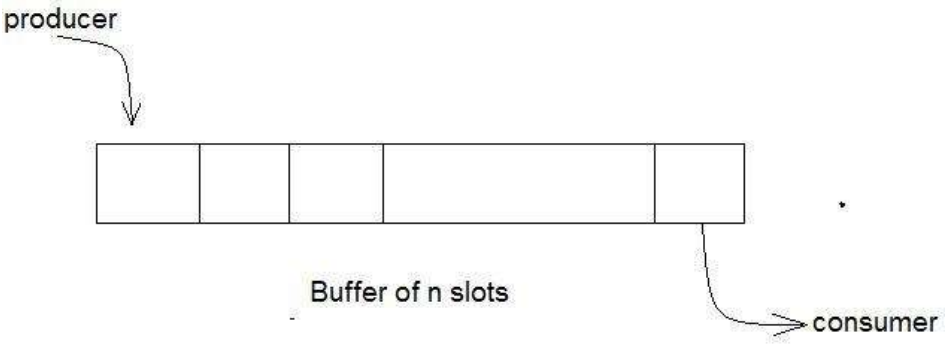
Consumer #1 got: 9

Bounded Buffer Problem

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization.

Problem Statement:

There is a buffer of **n** slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

```
import java.io.*;
import java.util.*;

 class Buffer
 {
private final int MaxBuffSize;
private int[] store;
private int BufferStart, BufferEnd, BufferSize;

public Buffer(int size)
{
MaxBuffSize = size;
BufferEnd = -1;
BufferStart = 0;
BufferSize = 0;
store = new int[MaxBuffSize];

}
public synchronized void insert(int ch)
{
try
{
        while (BufferSize == MaxBuffSize) {
        wait();
        }
        BufferEnd = (BufferEnd + 1) % MaxBuffSize;
```

```
        BufferSize++;
        notifyAll();
        }
catch (InterruptedException e)
        {

        }
}
public synchronized int delete() {
int ch=0;
        try {
while (BufferSize == 0) {
        wait();
        }

        ch = store[BufferStart];
BufferStart = (BufferStart + 1) % MaxBuffSize;
BufferSize--;
notifyAll();

} catch (InterruptedException e) {

}
        return ch;
}
}
```

```
class Consumer extends Thread {
private final Buffer buffer;
public Consumer(Buffer b) {
buffer = b;
}
public void run() {
while (!Thread.currentThread().isInterrupted()) {
int c = buffer.delete();
System.out.print(c);
}
}
}
class Producer extends Thread {
private final Buffer buffer;

public Producer(Buffer b) {
buffer = b;
}
public void run() {
for(int c=0;c<10;c++)
        buffer.insert(c);
}
}
```

```
class boundedbuffer {

public static void main(String[] args) {

System.out.println("program starting");

Buffer buffer = new Buffer(5); // buffer has size 5

Producer prod = new Producer(buffer);

Consumer cons = new Consumer(buffer);

prod.start();

cons.start();

try {

prod.join();

cons.interrupt();

} catch (InterruptedException e) {}
System.out.println("End of Program");



}

}
```

0

1

2

3

4

5

6

End of Program

Readers Writer Problem

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

Problem Statement:

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non-zero number of readers accessing the resource.

Solution:

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Refer any sample program for reader writer problem