Module 4

Packages and Interfaces:

Packages, Access Protection,Importing Packages.Interfaces.

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.

## Interface

Interface’s are another form of creating abstract classes in java.

Abstract classes cannot be used to achieve multiple inheritance, but **interfaces can be used to achieve multiple inheritance in java.**

Interface is a mechanism to achieve abstraction (specifying the task’s to be done by derived classes, but interface is not going to implement one)

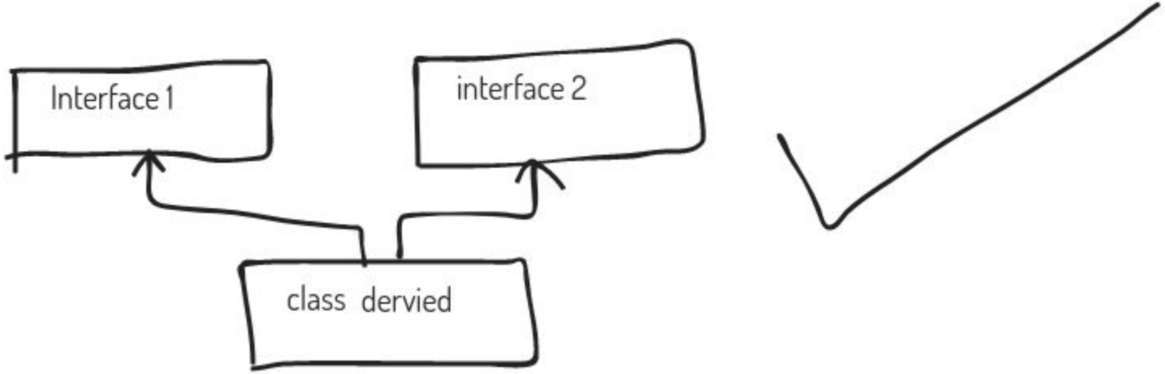
Interface implements abstraction in its basic form, by **not defining any method within it**.

Abstract classes will not implement abstraction to its fullest, because some methods can be defined within it.

There **can be only abstract methods in the interface**.

# Interface

ALLOWED IN JAVA

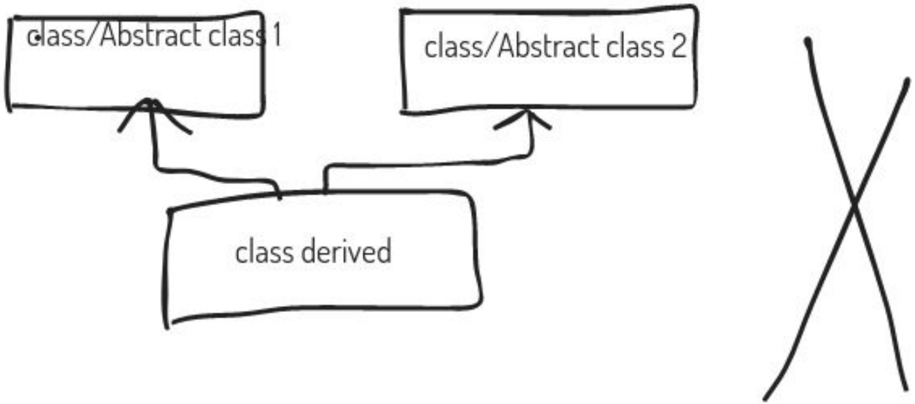


Interface 1

interface 2

class  dervied

NOT ALLOWED IN JAVA



class/Abstract class 1

class/Abstract class 2

class derived

## Interface

Instance cannot be created for an interface, but a reference can be created.

Java Abstract class & Interface represents **“IS-A”** relationship.

A single class can inherit multiple interfaces.  A single class can inherit from only one Abstract class/Non-abstract class.

“**implements**” is the keyword used to inherit the interface.

```
G.F:  <access-specifier>    interface    <interface-name>
          {
             return-type method-name1(parameter-list); //method declaration
             return-type method-name2(parameter-list);
             …….
             type final varname1 = value;   // final and static variables which must be initialized
             type final varname2 = value;    }
```

## Interface

G.F: of implementing interface

```
    class <calss-name> [extends classname] [implements interface [,interface...]]
    {
     ....//class body
    }
```

**[ ] indicates optional**

If a class **implements** an interface, then all the methods in that must be compulsorily defined by the inherited class.

All methods and variables are implicitly public in an interface.

**NOTE: An interface can extend (inherit) another interface.**

## Interface

```
Ex: interface disp {
        public void display();
        int i=90; // by default i is public final static
        }

        class cmp implements disp {
                public void cmp_method()
                {   System.out.println("method of cmp");    }

                @Override
                public void display()
                {   System.out.println("in cmp display");   }
        }
```

# Interface

```
class student implements disp  {
   public void student_method() {   System.out.println("method of student");   }
   @Override
    public void display()  {   System.out.println("in student display");   }
}
class test {
 public static void main(String[] args) {
        disp d = new student();                   d.display();
        //d.student_method(); // CTE

        d = new cmp();                 d.display();
        //d.cmp_method(); // CTE
  }
}
```

## Interface : Another example

```
import java.lang.*;
interface stack
{
  void push(Object obj);
  Object pop();
  int size=3;
}

class intstack implements stack
{
        private int top,a[];

         public intstack()
         { top=-1;  a = new int[size];  }
```

## Interface : Another example

```
public void push(Object obj){
        System.out.println(obj.getClass());
        if (top==size-1)
        {System.out.println("Integer stack full"); return;}

        top=top+1;
        a[top] = ((Integer) obj).intValue();
}

public Object pop()  {
        if (top==-1)
        {System.out.println("Integer stack empty"); return null; }

        Object obj = new Integer(a[top]);
        top--;
        return obj;
}   }
```

# Interface : Another example

```
        class fstack implements stack  {
                int top;          float a[];

          public fstack()
          {    top=-1;  a = new float[size];    }

          public void push(Object obj)
          {
             if (top==size-1)
             {System.out.println("Integer stack full"); return;}

             top=top+1;
             a[top] = ((Float)obj).floatValue();
          }
```

## Interface : Another example

```
          public Object pop()   {
            if (top==-1)
            {  System.out.println("Integer stack empty");   return null; }

            Object obj = new Float(a[top]);
            top--;
            return obj;
          }    }

        class test    {
          public static void main(String args[])
           {
            intstack i = new intstack();
            i.push( 10 );
            System.out.println(   i.pop() );
```

# Interface : Another example

```
fstack j = new fstack();
j.push(10f );
System.out.println(   j.pop()   );

System.out.println("One interface Multiple method invocation");
 stack s = i;      // base class reference pointing to its derived class
s.push(20);
System.out.println(  s.pop()   );
s = j;
s.push(30f);
System.out.println(  s.pop()  );
```

# Interface : Another example

```
            i.push(90);   i.push(10);

            int k  = i.pop() + i.pop( );  //CTE

            k = ((Integer)i.pop()).intValue() +  ((Integer)i.pop()).intValue() ;

            System.out.println(k);
        }
    }
```

i.pop( ) calls toString( ) of Integer wrapper class to obtain stringed integer value.

((Integer)i.pop()).intValue() retrieves value of type int.

# IMPLEMENTING MULTIPLE INTERFACE

```
interface accept_value
{   void accept();    }

interface display_value
{   void display();    }

class test implements accept_value, display_value    {
   public void accept() { System.out.println("In accept");    }
   public void display()  { System.out.println("In display"); }

   public static void main (String[] args)  {
      test a = new test();
      a.accept();
      a.display();
   }   }
```

# INTERFACE EXTENDING ANOTHER INTERFACE

```
interface accept_value
{   void accept();    }

interface display_value extends accept_value
{   void display();    }

class test implements display_value {
    public void accept()    { System.out.println("In accept"); }
    public void display()  { System.out.println("In display"); }

    public static void main (String[] args)  {
        test a = new test();   a.accept();
        a.display();
    }   }
```

## PACKAGE

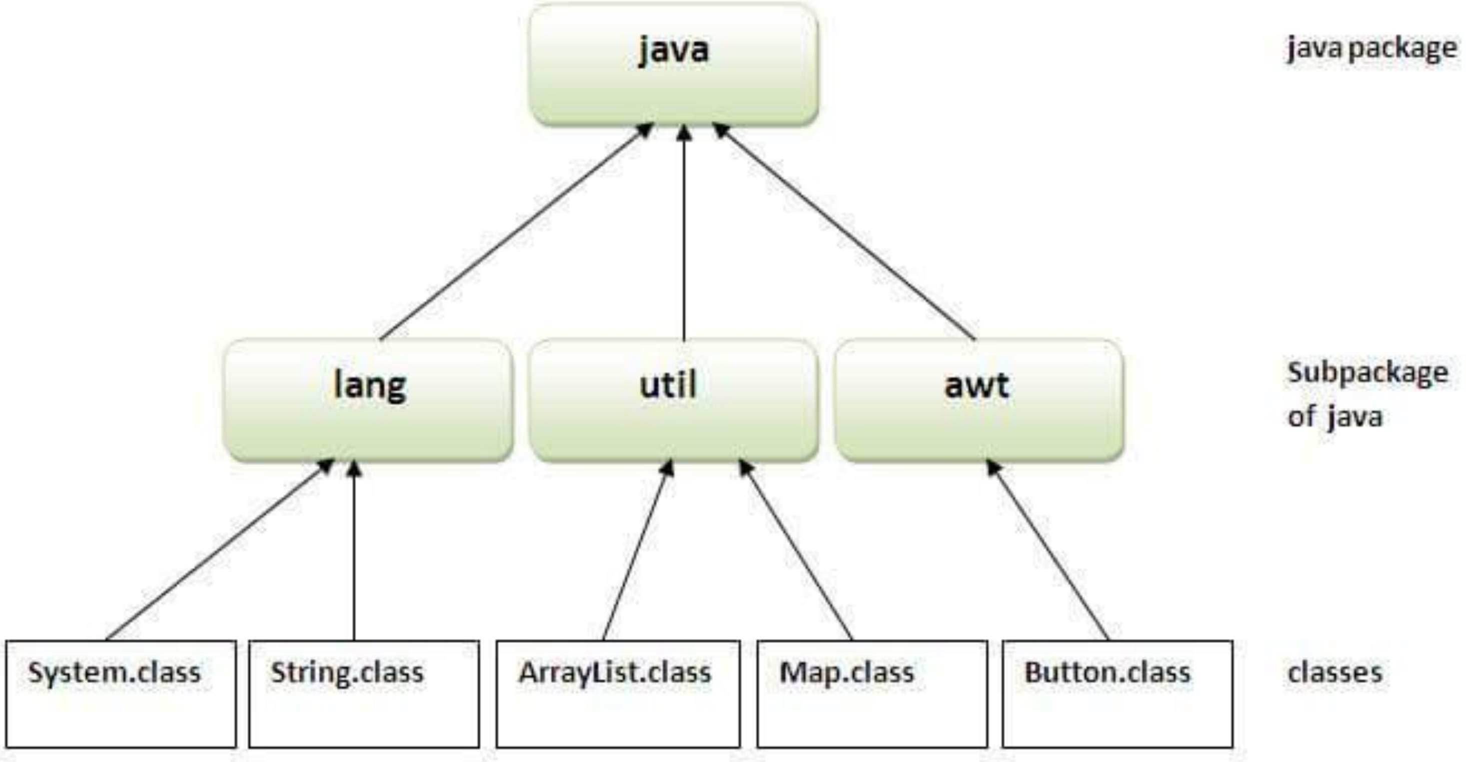
**Package** in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:

- Preventing naming conflicts.
  Ex: there can be two classes with similar names in two packages.
- Providing controlled access:
  protected and default have package level access control.
  A protected member is accessible by classes in the same package and its subclasses (outside the package).
  A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation or data-hiding.
- Usage of classes, interfaces will be easier by using package names.

Package in java can be categorized in two forms, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

# PACKAGE



java

java package

lang

util

awt

Subpackage of java

System.class

String.class

ArrayList.class

Map.class

Button.class

classes

## PACKAGE

Creating a package actually creates a directory with the same name as the package name.
(in Eclipse a default package is created for each project, if a suitable name is provided for the package, then a directory of the same name will be created to hold on to the classes, which are inside the package)

In console based implementation of packages, directory name which matches package name has to be created manually and respective classes must be kept in that directory.

Ex: package college.engineering.cse

college is the directory and main package name
engineering is the sub directory of college and intermediate package name
cse is the subdirectory of engineering and last package name

college ---> engineering ----> cse (Hierarchy of packages)

Any class to be added to this package must be stored in the “cse” directory.

## PACKAGE : Creating Packages

```
package college.engineering;
public class cse
//public specifier for class, to access CSE class outside the package //class.engineering.cse
{
 public static void print()
  {System.out.println("In college.engineering package class CSE"); }
}
```

**ONLY ONE PUBLIC CLASS IS ALLOWED IN ONE PACKAGE**
**ANY OTHER CLASS IN THE SAME PACKAGE MUST NOT BE PUBLIC.**

```
class ise
{
 public static void print()
  { System.out.println("In rns.engineering package class ISE"); }
}
```

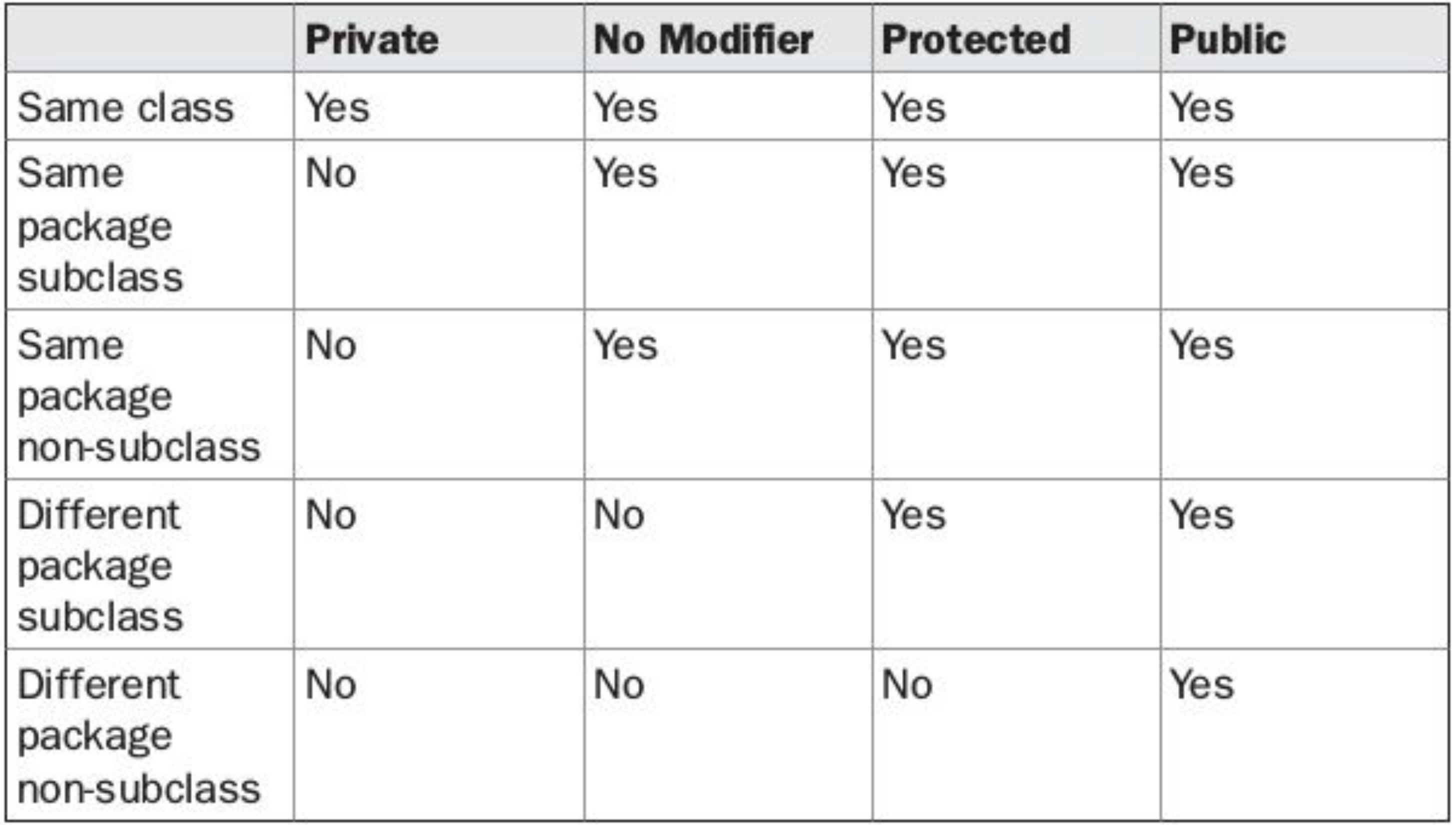
## PACKAGE : Creating Packages

**package college.pucollege;**

```
public class PUC {
  public static void print()
  { System.out.println("In rns.pucollege package class PUC"); }
}

//Test.java
import college.engineering.CSE;
import college.pucollege.*;
class Test  {
  public static void main(String args[])  {
     CSE.print();
     PUC.print();
  }
}
```

# PACKAGE : Access Protection



| | Private | No Modifier | Protected | Public |
|---|---|---|---|---|
| Same class | Yes | Yes | Yes | Yes |
| Same package subclass | No | Yes | Yes | Yes |
| Same package non-subclass | No | Yes | Yes | Yes |
| Different package subclass | No | No | Yes | Yes |
| Different package non-subclass | No | No | No | Yes |

## PACKAGE : Access Protection

Any member declared as public can be accessed from anywhere.

Any member declared as private cannot be accessed outside its class.

Any member declared as protected can be accessed outside the current package, but only inside the subclasses . Non-subclasses cannot access protected members.

When a member has a default access specifier, it is visible to subclasses as well as to other classes in the same package.

A class in a package can have only **public, final or default access specifiers**.

When a class is **declared as public**, it is accessible anywhere in the project. (Project is a combination of several packages).

## PACKAGE : Access Protection

When a class is public, it must be the **only public class** declared in the file, and the **file must have the same name as the class**.

If a class has **default access**, then it can only be accessed by other classes within the same package.

Ex:

**File path: rns\engineering\CSE.java**

```
package rns.engineering;
public class CSE   {
        private int ipri;
        protected int ipro;
        public int ipub;
        int inomod;
}
```

# PACKAGE : Access Protection

**File path: rns\engineering\NonSubClassSamePackage.java**

```
package rns.engineering;
public class NonSubClassSamePackage {

    public void access()
    {
        System.out.println("In Same Package Non-Subclass");
        CSE obj = new CSE();
        //obj.ipri = 90; // CTE
        obj.ipro = 900;
        obj.ipub = 9000;
        obj.inomod = 89;
    }
}
```

## PACKAGE : Access Protection

**File path: rns\engineering\SubClassSamePackage.java**

```
package rns.engineering;
public class SubClassSamePackage extends CSE
{
    public void access()
    {
        System.out.println("In Same Package Sub class");
        //ipri =90; // CTE
        ipro = 900;   ipub = 9000;
        inomod = 89;
    }
}
```

## PACKAGE : Access Protection

**File path: rns\pucollege\NonSubClassDifferentPackage.java**

```
package rns.pucollege;
import rns.engineering.CSE;
public class NonSubClassDifferentPackage  {
    public void access()
    {
        System.out.println("In Different Package non-sub-class");
        CSE cse = new CSE();
        //cse.ipri=90; //CTE
        //cse.ipro = 90; //CTE
        //cse.inomod = 90; // CTE
        cse.ipub = 90;
    }
}
```

# PACKAGE : Access Protection

**File path: rns\pucollege\SubClassDifferentPackage..java**

```
package rns.pucollege;
import rns.engineering.CSE;
public class SubClassDifferentPackage extends CSE
{
    public void acess()
    {
        System.out.println("In Different Package Sub class");
        //ipri=90; // CTE : not visible
        ipro = 900;
        ipub = 9000;
        //inomod = 89; // CTE : not visible
    }
}
```

## PACKAGE : Access Protection

**File path: src\Test..java**

```
package Test;
import rns.engineering.*;
import rns.pucollege.*;
class Test {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        SubClassSamePackage p = new SubClassSamePackage();
        NonSubClassSamePackage q = new NonSubClassSamePackage();
        SubClassDifferentPackage r = new SubClassDifferentPackage();
        NonSubClassDifferentPackage s = new NonSubClassDifferentPackage();
        p.access();                 q.access();
        r.acess();                  s.access();
    }
}
```

**PACKAGE : Importing Packages**

Packages are a way to categorize different classes from each other.

All built in java classes are stored in packages, there is not a single built in java class that is present in the unnamed-default package.

All standard classes are stored in some named package.

It is a tedious task to type long dot-separated package path names for each and every class present in the package to use it.

```
Ex: class test {
      public static void main(String args[]) {
        java.util.Scanner ip = new java.util.Scanner(java.lang.System.in);
        java.lang.System.out.println("Afd");
  }  }
```

Java includes import statements to bring certain classes, or entire packages, into visibility.

**PACKAGE : Importing Packages**

Once imported, a class can be referred to directly, using only its name. The import statement will save a lot of typing.

In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions.

**G.F: import pkg1[.pkg2].(classname|*);**

Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.).

There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, either an explicit class name or a star (*) must be specified, which indicates that the Java compiler must import the entire package or all the classes in the package.

Ex:
```
import java.util.Date;
import java.io.*;
```

## Multithreaded Programming

All modern operating systems support multitasking. There are two different types of multitasking.

1. Process-based 2. Thread-based.

***Process-based multitasking***

A *process* is a program under execution.

If a system allows more than one process to execute simultaneously, it is termed as *process-based multitasking*.

Ex: A process-based multitasking facilitates to run the Java compiler at the same time that you are using a text editor.

Processes are heavyweight tasks that require their own separate **address spaces**.

Interprocess communication is expensive and limited.

Context switching from one process to another is also costly. (Costly in terms of CPU time wastage)

**Multithreaded Programming**

***Thread-based multitasking***

Thread is the smallest unit of dispatchable/executable code.

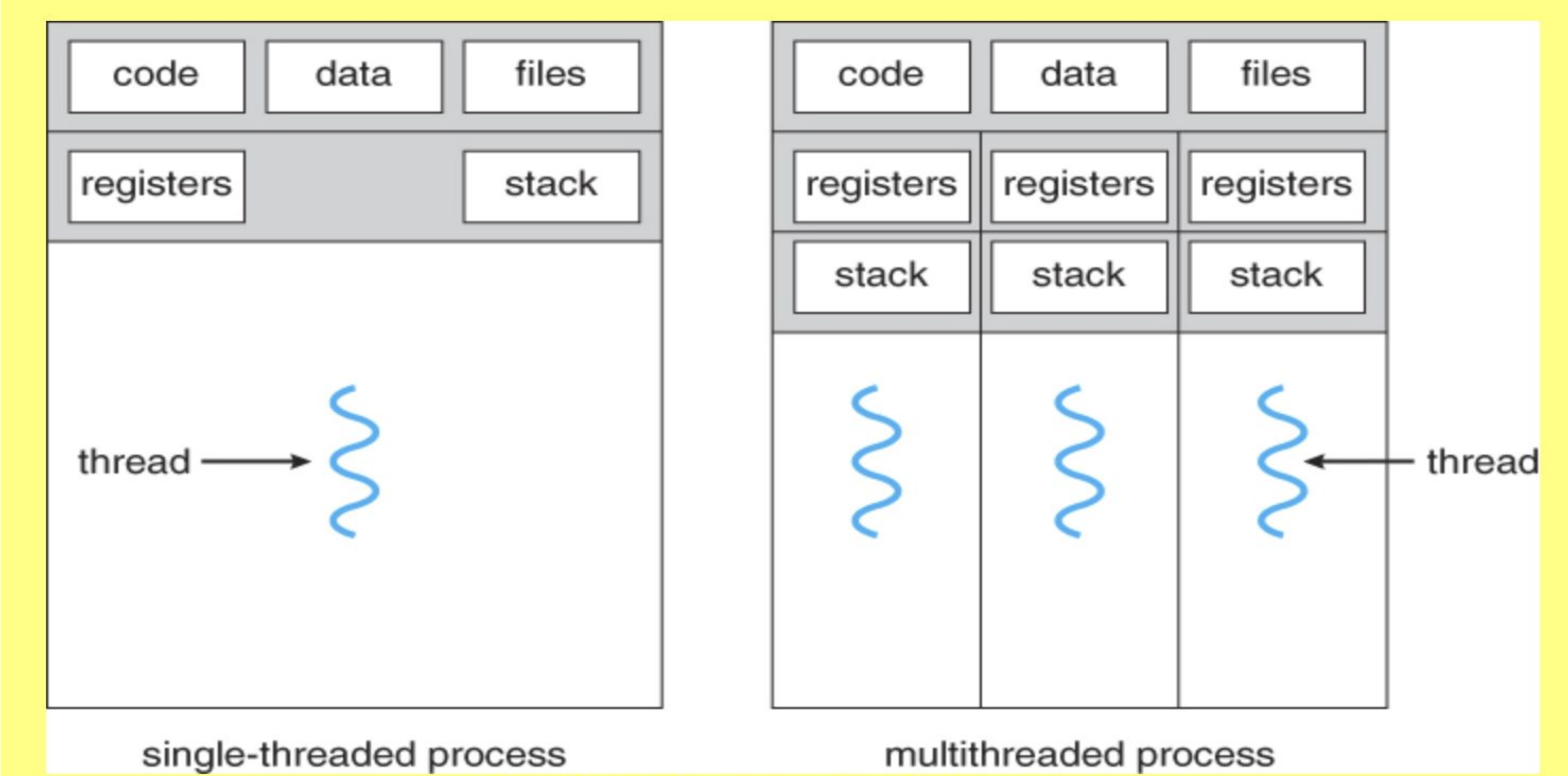


**Figure 4.1 Single-threaded and multithreaded processes**

# Multithreaded Programming

## *Thread-based multitasking*

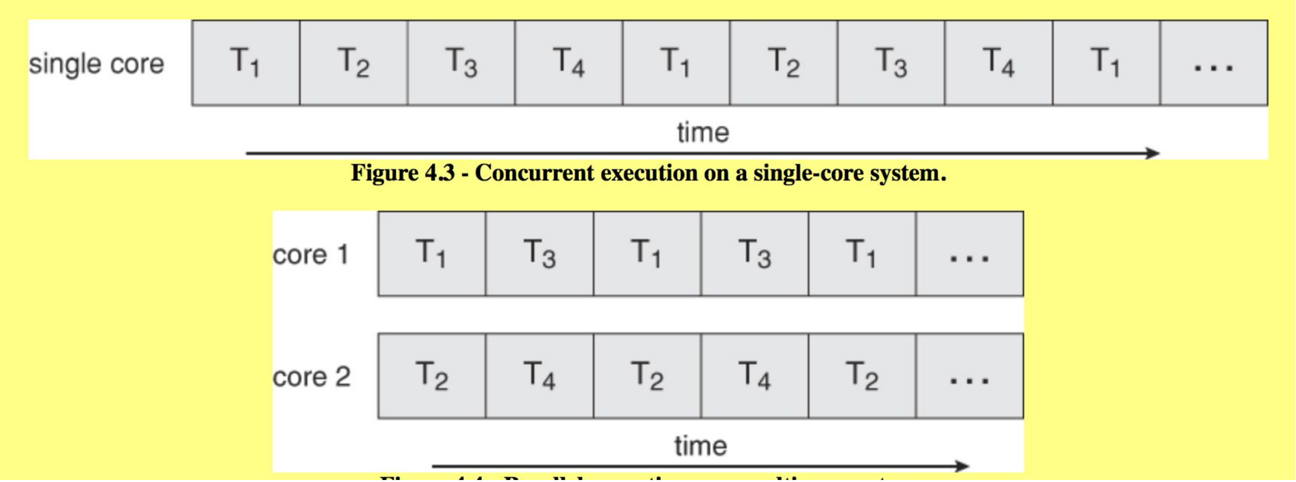


**Figure 4.3 - Concurrent execution on a single-core system.**

A single program can perform two or more tasks simultaneously.

Ex: A text editor can format text at the same time that it is printing.  Printing and formatting will be done by two separate threads which belong to a single process.

“Multitasking threads” require less overhead than “multitasking processes”.

## Multithreaded Programming

***Thread-based multitasking***

Threads are a lightweight process.

They share the same address space of a heavyweight process.

Inter Thread communication is inexpensive.
Context switching from one thread to the next is low cost.

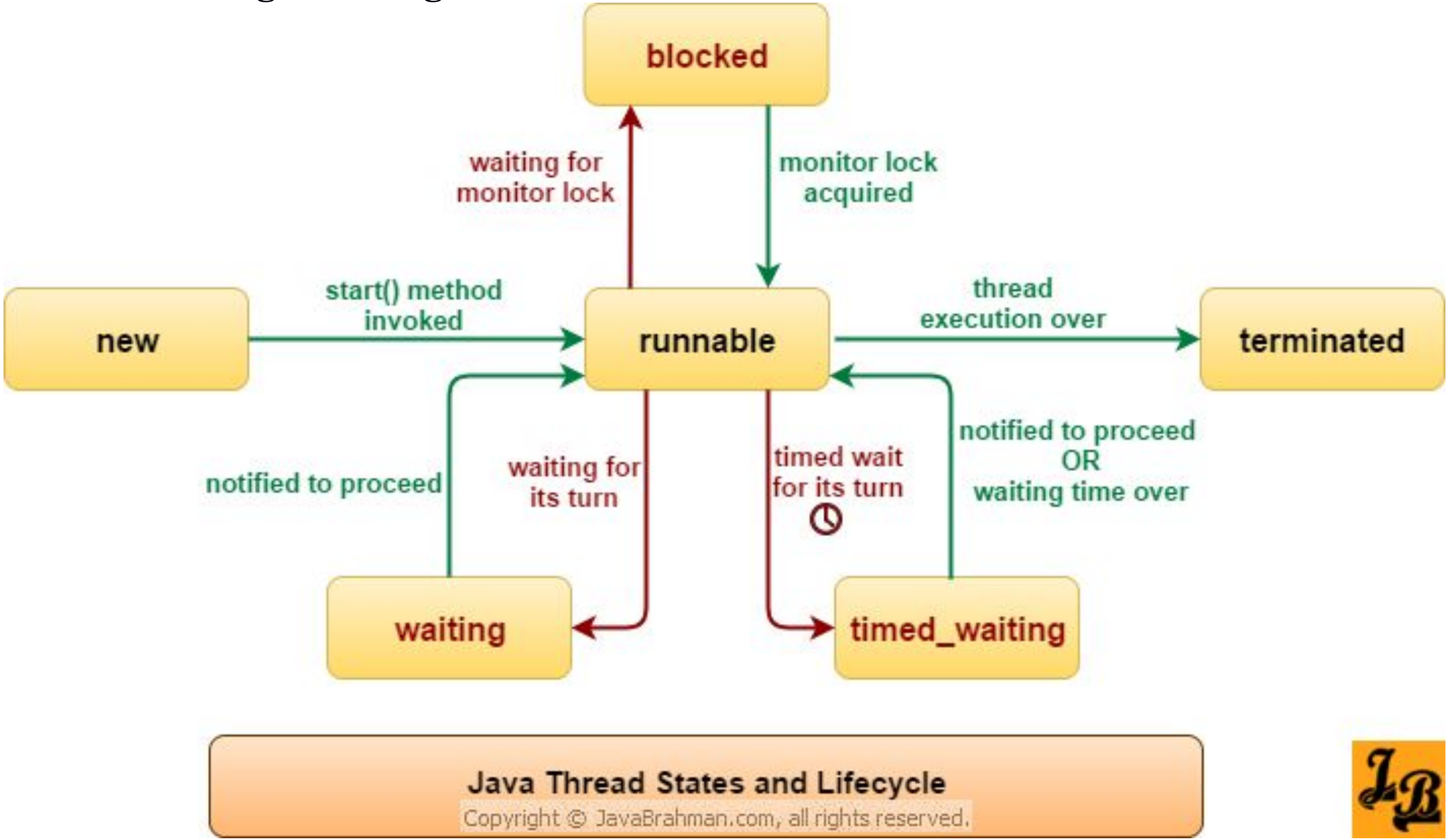
Java provides built-in support for *multithreaded programming*.

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

Multithreading is a specialized form of multitasking.

**While Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java. However, multithreaded multitasking is.**

# Multithreaded Programming : Thread states



Java Thread States and Lifecycle

Copyright © JavaBrahman.com, all rights reserved.

## Multithreaded Programming : Thread class

Multithreading in java is built on the **Thread** class methods, and its associated **Runnable** interface.

To create a new thread, the program will either **extend Thread** or **implement** the **Runnable** interface**.**

The Thread class defines several methods that help manage threads.

| | |
|---|---|
| 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 for the thread. |
| sleep | Suspend a thread for a period of time. |
| start | Start a thread by calling its run method. |

https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

All java programs by default have a **single thread termed as main**.

## Multithreaded Programming : Thread class

When a Java program begins execution, one thread begins running immediately i.e main function.

- **It is the thread from which other “child” threads will be created.**

- Often, it must be the last thread to finish execution because it performs various shutdown actions.

```
   class Thread     //built-in class
   {
     public   static   Thread   currentThread( );
     public   final    void     setName(String threadName);
     public   final    String   getName( );
  public   static   void     sleep(long milliseconds)     throws     InterruptedException
   }
```

## Multithreaded Programming : Thread class

**currentThread( )** returns a reference to the thread in which it is called.

**setName( )** is used to change the internal name of the thread, **threadName** specifies the name of the thread.

Ex: main( ) thread can be controlled through a Thread object.
A reference for main( ) has to be obtained using currentThread( ).

```
class CurrentThreadDemo  {
 public static void main(String args[])   {
     Thread t = Thread.currentThread();
     System.out.println("Current thread: " + t);

     t.setName("My Thread");  // changing the name of the thread

     System.out.println("After name change: " + t);
```

## Multithreaded Programming : Thread class

```
        try {           // becz, Thread.sleep( ) throws InterruptedException
          for(int n = 5; n > 0; n--) {
              System.out.println(n);
              Thread.sleep(1000);
          }
      }
      catch (InterruptedException e)
      { System.out.println("Main thread interrupted"); }
  }
}
```

**Output:**

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

## Multithreaded Programming : Thread class

Numerical values are displayed from 5 to 0 with a sleep interval of 1 second by calling sleep( ) method.

**Exceptions will be generated automatically by Thread.sleep( ), if another thread tries to interrupt a sleeping thread.**

Thread information is displayed in the following manner.

the name of the thread, (which is main)
its priority, (default priority is 5)
and the name of its group. (main is the name of the group)

## Multithreaded Programming : Creating a Thread using Runnable interface

Java has two ways of creating a thread:

- implement the Runnable interface.
- extend the Thread class.

**Implementing Runnable**

Generating a thread in java, using Runnable interface is a two step process.

**1) To generate a thread, is to create a class that implements the Runnable interface.**

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

**Multithreaded Programming : Creating a Thread using Runnable interface**
**run( ) is the entry point for a thread in the program.**

Statements that are supposed to be executed in thread must be coded in run( ) method. Even function calls can also be made within the run ( ) method.

**2) An object of type Thread must be instantiated from within the class which implements Runnable interface.**

Thread defines several constructors, one of them is

```
class Thread {
        Thread( Runnable   threadOb,        String threadName);
        Thread (Runnable threadOb);
}
```

**threadOb:** is an instance of a class that implements the Runnable interface.
**threadName:** name of new thread.

After a new thread object is created, it will not start executing until the **start( )** method is called, which is declared within the Thread class.

```
class stack implements Runnable {
        @Override
        public run() { ……..}

        Thread t;

        public stack () {
                t = new Thread(this, “FT”);
                t.start();
                …..
       }
}

class test {
                Public static void main(String [] argos) {
                                stack s = new stack();
                }
}
```

# Multithreaded Programming : Creating a Thread using Runnable interface

In essence, start( ) executes a call to run( ).

```
      class Thread      {
        public void start( ) throws IllegalThreadStateException
                                                  //if the thread was already started.
      }
```

Thread process must start its execution only once and not many times.

```
Ex:  class NewThread implements Runnable  {
                    Thread t;
```

*//Since, start( ) and run( ) is present only in Thread class, an instance of Thread must be*
*//created here.*

# Multithreaded Programming : Creating a Thread using Runnable interface

```
  NewThread()
  {
    // Thread object is created, which is used to call start( ) method.
    t =      new Thread(this, "Demo Thread");

   /*
      “this” is a reference of type NewThread which implements Runnable interface.
      Hence, it is passed as the first parameter.
   */
    System.out.println("Child thread: " + t);


   // Create a new, second thread.  Becz main is considered as the 1st thread
    t.start(); // Begins thread execution
 }
```

## Multithreaded Programming : Creating a Thread using Runnable interface

```
    // This is the entry point for thread.
    @Override
    public void run()
    {
        for(int i = 5; i > 0; i--)
        {
            System.out.println("Child Thread: " + i);
            try  {
                    Thread.sleep(500);
                }
           catch (InterruptedException e)
           { System.out.println("Child interrupted."); }
        }
           System.out.println("Exiting child thread.");
      }
  }//End of class NewThread
```

## Multithreaded Programming : Creating a Thread using Runnable interface

```
class ThreadDemo {
  public static void main(String args[])  {
    new NewThread(); // create a new thread

     //main thread resumes its execution
     try {
       for(int i = 5; i > 0; i--)  {
             System.out.println("Main Thread: " + i);
             Thread.sleep(1000);
        }   }
    catch (InterruptedException e)
    {System.out.println("Main thread interrupted.");}

   System.out.println("Main thread exiting.");
  }
}//End of class ThreadDemo
```

## Multithreaded Programming : Creating a Thread using Runnable interface

a new Thread object is created by the following statement inside constructor

**t = new Thread(this, "Demo Thread");**

Passing **“this”** as the first argument indicates that the new thread has to be called on the object referenced by this.

Next, start( ) is called, which starts the thread execution beginning from the run( ) method.

After calling start( ), NewThread’s constructor returns to main( ), then main thread resumes its execution, and begins executing for loop.

Both threads (main & Demo Thread) continue executing, sharing the CPU, until their statements get executed.

# Multithreaded Programming : Creating a Thread using Runnable interface

**Output may vary based on processor speed and task load**

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

**Multithreaded Programming : Creating a Thread using Runnable interface**

In a multithreaded program, often the main thread must be the last thread to finish running. (automatically the main thread will not conclude the execution at last, but forcibly it will be made to conclude the execution after all the child threads completes its execution.

In case if any resources that are acquired by the main thread and in turn they are used by the child threads, then the main thread has to wait until the child thread concludes its execution.

Ex: Scanner instance can be instantiated in main thread and in turn it can be used by child thread. Only after the usage of child thread is over, the scanner instance can be closed in the main thread. Closing of scanner instance in child thread, will avoid further usage of the same in main thread.

Habit, of acquiring resources will be done in main because it stays for a longer period of time, since the execution starting point is main and the usual exit of control is from main.)

The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread.

## Multithreaded Programming : Creating a Thread using Extending Thread

Another way to create a thread is to create an user defined class that extends the “Thread” class, and then to create an instance of it.

Extending class must override the run( ) method, which is the entry point for the new thread.

It must also call start( ) to begin execution of the new thread.

Ex:
```
class NewThread extends Thread {
        NewThread()   {
            // Calling Thread class constructor        public Thread(String threadName)
            super("Demo Thread");
            System.out.println("Child thread: " + this);
            start(); // Start the thread, which will in turn call run( ) method.
        }
```

# Multithreaded Programming : Creating a Thread using Extending Thread

```
 // This is the entry point for the second thread.  Overriding run method
  @Override
   public void run()    {
          try {
                for(int i = 5; i > 0; i--)   {
                     System.out.println("Child Thread: " + i);
                     Thread.sleep(500);
                 }
          }
          catch (InterruptedException e)
          { System.out.println("Child interrupted."); }
          System.out.println("Exiting child thread.");
         }
}//End of class NewThread
```

## Multithreaded Programming : Creating a Thread using Extending Thread

```
class ExtendThread    {
  public static void main(String args[])    {
          new NewThread(); // create a new thread, unreferenced instance
          try {
                  for(int i = 5; i > 0; i--)    {
                    System.out.println("Main Thread: " + i);
                    Thread.sleep(1000);
                  }
          }
          catch (InterruptedException e)
          {System.out.println("Main thread interrupted.");}
          System.out.println("Main thread exiting.");
  }    }//End of class ExtendThread
```

Output will be the same as before.

## Multithreaded Programming : Creating a Thread using Extending Thread

**Whether to use *implements (Runnable)* or *extends (Thread)* to create a thread**

Thread class defines several methods that can be overridden by a derived class. Of these methods, the only one that must be overridden is run( ) to create a thread. This is, of course, the same method required by Runnable.

Thread class must be extended only when they are being enhanced or modified in some way.

If none of the other thread methods are overridden, other than run( ) method, then it is best implement Runnable.

**https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html**

Consider

```
        PrimeRun p = new PrimeRun(143);
         new Thread(p).start();
```

# Multithreaded Programming : Creating a Multiple Threads

```
//Create multiple threads.
class NewThread implements Runnable
{

  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
          name = threadname;
          t = new Thread(this, name);
          System.out.println("New thread: " + t);
          t.start(); // Start the thread
  }
```

# Multithreaded Programming : Creating a Multiple Threads

```
    @Override
    public void run()
    {
        try {
            for(int i = 2; i > 0; i--) {
             System.out.println(t.getName() + ": " + i);
             Thread.sleep(1000);
          }
        }
        catch (InterruptedException e)
        {System.out.println(name + "Interrupted");}
        System.out.println(name + " exiting.");
       }
   }
```

# Multithreaded Programming : Creating a Multiple Threads

```
class Test  {
    public static void main(String args[]) {
          new NewThread("One");
          new NewThread("Two");
          new NewThread("Three");
       try {
          // wait for other threads to end
             Thread.sleep(10000);
         //main sleeps for 10 seconds, to ensure that the main thread finishes last.
       }
       catch (InterruptedException e)      { System.out.println("Main thread Interrupted"); }
       System.out.println("Main thread exiting.");
     }
}
```

## Multithreaded Programming : Creating a Multiple Threads

**Output:**

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One: 2
Two: 2
New thread: Thread[Three,5,main]
Three: 2
Two: 1
Three: 1
One: 1
Two exiting.
One exiting.
Three exiting.
Main thread exiting.
```

## Multithreaded Programming : isAlive( ) and join( )

Making the main( ) thread to sleep for a longer duration of time, and allowing child processes to complete its execution is not a competent solution. (Asynchronous waiting)

Two ways exist to determine if a thread is alive or not.

1. **isAlive( )** 2. **join( )** (Synchronous waiting)

```
class Thread {
      final     boolean   isAlive()
      final     void      join( )          throws InterruptedException
}
```

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

join( ) method, puts the thread from which it is called, on wait,
until the thread on which it has called finishes its execution.

## Multithreaded Programming : isAlive( ) and join( )

If thread (which has invoked join() )is interrupted then it will throw InterruptedException.

**Ex: using join( ) method**

```
class NewThread extends Thread
{
    NewThread(String n)
    {
        this.setName(n);
        System.out.println("Child thread: " + this);
        start();
    }
```

# Multithreaded Programming : isAlive( ) and join( )

```
            @Override
            public void run()  {
             try   {
                    for(int i = 3; i > 0; i--)   {
                            System.out.println(  getName() +" Thread: " + i);
                            Thread.sleep(500);       }
             }
             catch (InterruptedException e)   { System.out.println("Child interrupted."); }

         System.out.println("Exiting "+getName()+" child thread.");
        }
        }//End of class NewThread
```

## Multithreaded Programming : isAlive( ) and join( )

```
  class Test  {
        public static void main(String args[])   {
          NewThread nt = new NewThread("First");
          NewThread nt1 = new NewThread("Second");
          try {
                  nt.join();
                  nt1.join();
/* Since nt.join( ) is called from main thread, main thread suspends execution until thread
named “First” completes its execution.  Once nt thread completes its execution, main( ) thread
resumes execution */
           }
           catch (InterruptedException e)    {System.out.println("Main thread interrupted.");}
           System.out.println("Main thread exiting.");
        }  }
```

# Multithreaded Programming : isAlive( ) and join( )

**Output:**

```
Child thread: Thread[First,5,main]
Child thread: Thread[Second,5,main]
First Thread: 3
Second Thread: 3
First Thread: 2
Second Thread: 2
First Thread: 1
Second Thread: 1
Exiting First child thread.
Exiting Second child thread.
Main thread exiting.
```

**Multithreaded Programming : Thread Priorities**
**https://www.informit.com/articles/article.aspx?p=26326&seqNum=5**

Priorities Determine which thread gets CPU allocated and gets executed first.

Thread priority values in java range from 1 to 10, 1 being the least priority and 10 being the highest.

Higher the thread priority, larger is the chance for a process of getting executed first.
Ex:

Two threads are ready to run.
First thread priority is 5 and begins execution.
Second thread priority, consider to be 10 is ready for execution,
Then, First thread may suspend its execution relieving the control to Second thread.

A thread’s priority is also used to decide when to switch from one running thread to the next termed as **"context switch"**.

# Multithreaded Programming : Thread Priorities

setPriority( ) method, is used to set priority for a thread

```
   class Thread {
             final void setPriority(int level)     level specifies the priority
   }
```

level value must be in the range MIN_PRIORITY and MAX_PRIORITY, these values are 1 and 10, respectively.

Default priority of a thread is NORM_PRIORITY, which is equal to 5.

MIN_PRIORITY, MAX_PRIORITY & NORM_PRIORITY’s are defined as **static final variables** within Thread.

Current priority of a thread can be obtained by calling  getPriority( ) method of Thread,

```
     class Thread {
             final int getPriority( )     }
```

## Multithreaded Programming : Thread Priorities

**Theoretically** higher-priority threads get more CPU time than lower-priority threads.

Ex://Demonstrate thread priorities.

```
class clicker implements Runnable   {
    long click = 0;
    Thread t;
    private volatile boolean running = true;

    public clicker(int p)    {
        t = new Thread(this);
        t.setPriority(p);
    }

    public void run()   {
      while (running)
       click++;
    }
```

# Multithreaded Programming : Thread Priorities

```
public void stop()      { running = false; }

public void start()     { t.start(); }                 }

class test    {
public static void main(String args[])
{
        clicker hi = new clicker(Thread.NORM_PRIORITY + 4);
        clicker lo = new clicker(Thread.NORM_PRIORITY - 2);

        hi.start();
        lo.start();
```

## Multithreaded Programming : Thread Priorities

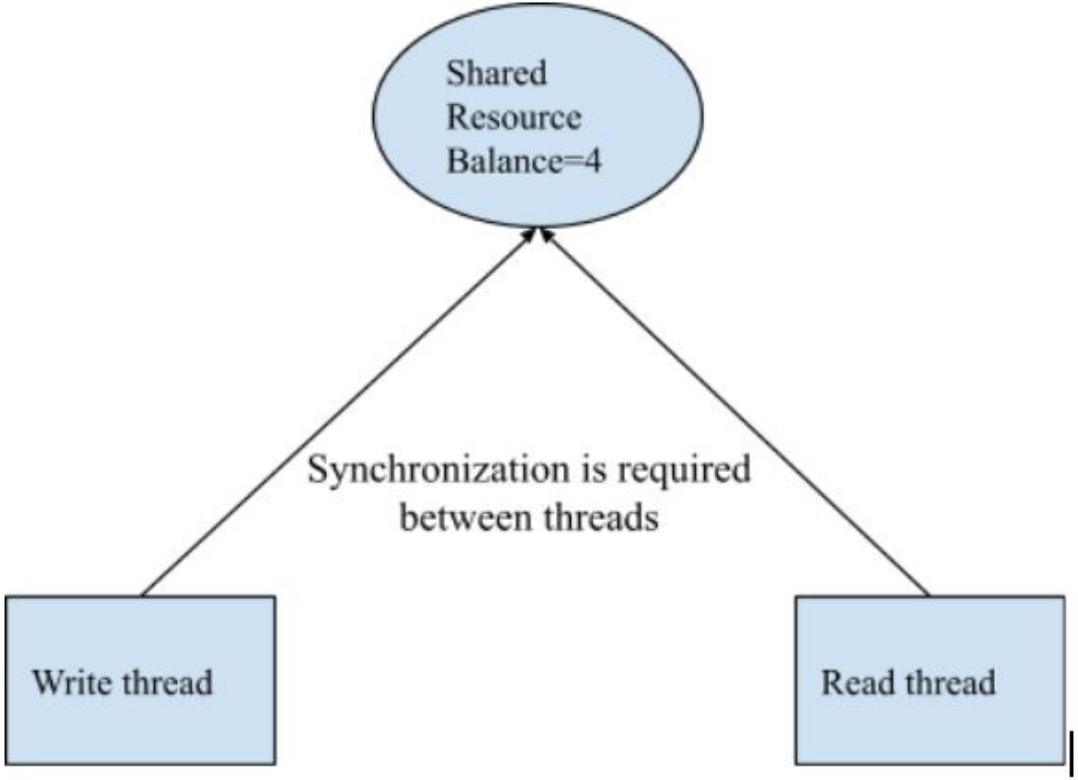
```
try {   Thread.sleep(1);   }
catch (InterruptedException e)
{ System.out.println("Main thread interrupted."); }

lo.stop();
hi.stop();
try {
      hi.t.join();          lo.t.join();
 }
 catch (InterruptedException e)
 { System.out.println("InterruptedException caught"); }

 System.out.println("Low-priority thread: " + lo.click);
 System.out.println("High-priority thread: " + hi.click);      }         }
```

## Multithreaded Programming : Synchronization

Two threads can access a shared resource, if two threads access a resource **simultaneously** error may ensue.



## Multithreaded Programming : Synchronization

Synchronization is a mechanism to achieve exclusive access to shared resources by more than one thread. Key to synchronization is the concept of monitor.

/*

***Monitors***

*Abstract Data Type for handling/defining shared resources*

- *Comprises:*
  - ***Shared Private Data***
    - *The resource Cannot be accessed from outside*
  - ***Procedures that operate on the data***
    - *Gateway to the resource*
    - *Can only act on data local to the monitor*
  - ***Synchronization primitives***
    - *Among threads that access the procedures*

# Multithreaded Programming : Synchronization

## *Monitors guarantee mutual exclusion*

*Only one thread can execute a monitor procedure at any time.*

*"in the monitor"*

*If second thread invokes a monitor procedure at that time*

- *It will be blocked and put into wait for entry to the monitor*
- *In Need of a wait queue*

# Multithreaded Programming : Synchronization

```
Monitor monitor_name
{
      // shared variable declarations

      procedure P1(. . . .) {
          . . . .
      }

      procedure P2(. . . .) {
          . . . .
      }
      :
      procedure PN(. . . .) {
          . . . .
      }

      initialization_code(. . . .) {
          . . . .
      }
}
```

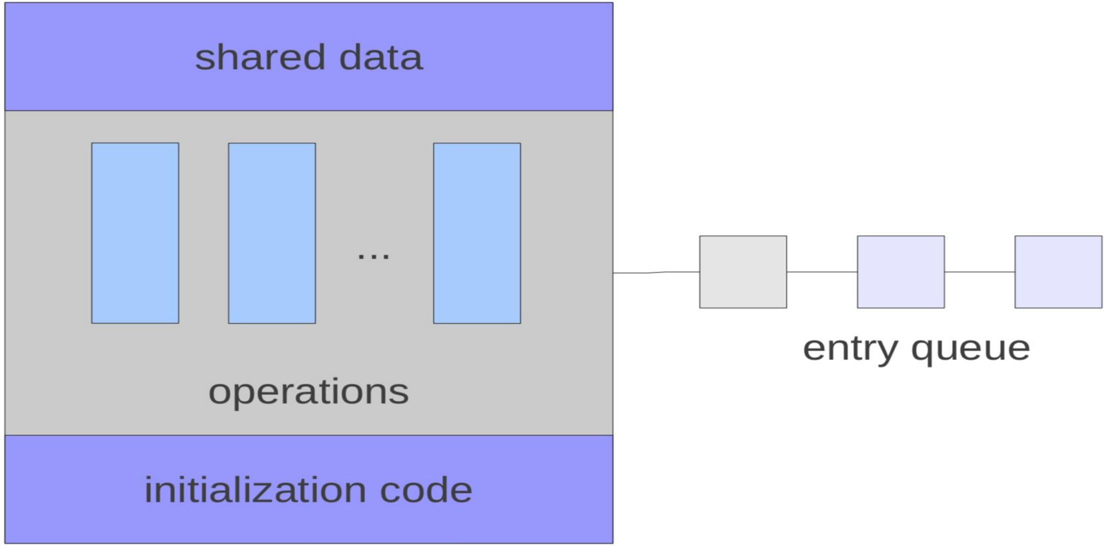
**For example:**

```
Monitor stack
{
      int top;
      void push(any_t *) {
          . . . .
      }

      any_t * pop() {
          . . . .
      }

      initialization_code() {
          . . . .
      }
}
```

# Multithreaded Programming : Synchronization

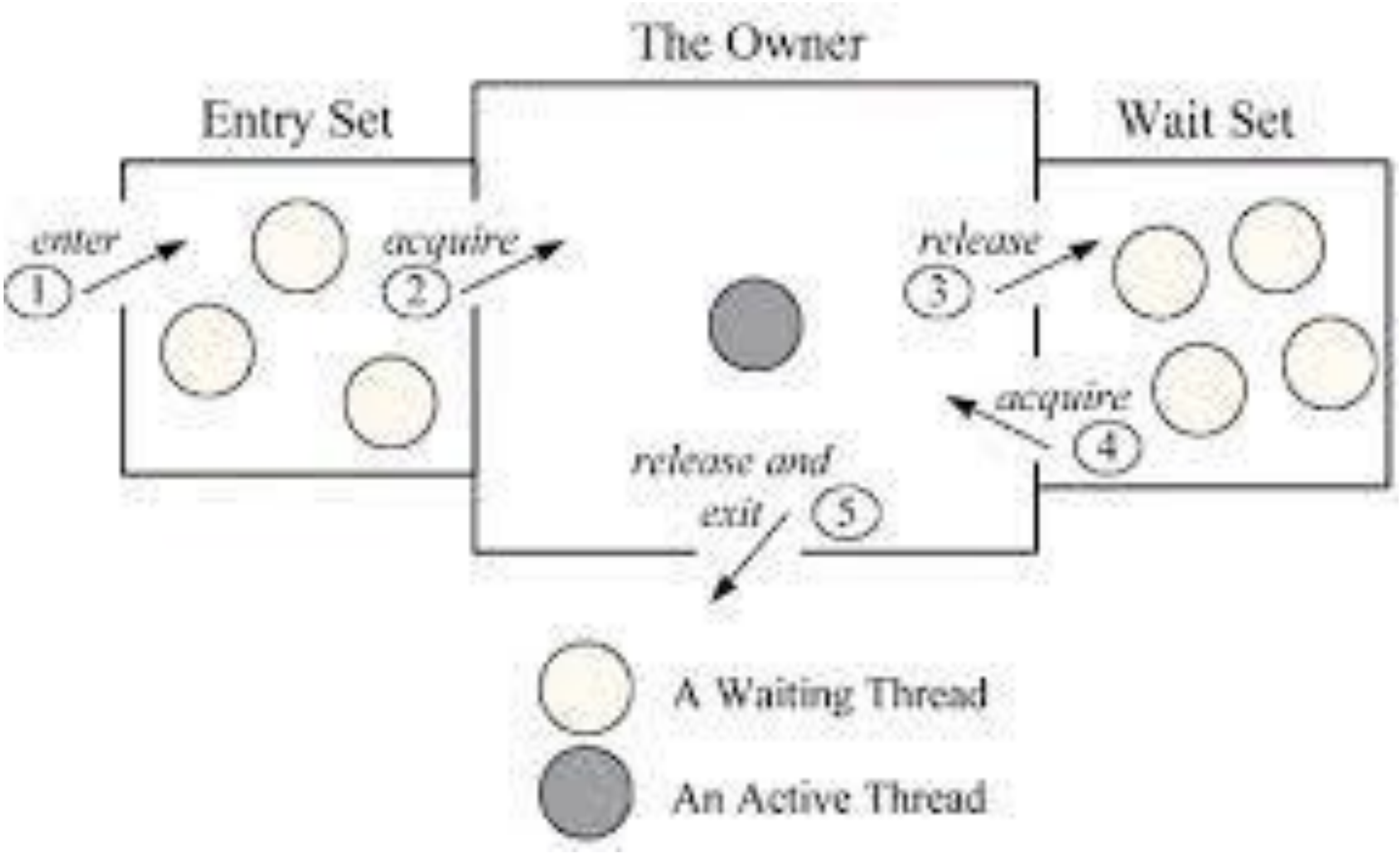


*A monitor is an object that is used as a mutually exclusive lock, or mutex.*

*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. */*

# Multithreaded Programming : Synchronization



# Multithreaded Programming : Synchronization

Ex: Expected output of the below program is

**[Hello]**

**[Synchronized]**

**[World]**

Using 3 different threads, **without any synchronization** between them.

**Pgm Name : NotSynch.java**

```
class Callme  {

     void call(String msg) {
                 System.out.print("[" + msg);
                  try {  Thread.sleep(1000);  }
                 catch(InterruptedException e)
                  {System.out.println("Interrupted");}

                System.out.println("]");
       }    }
```

## Multithreaded Programming : Synchronization

```
class Caller implements Runnable {
        String msg;       // required to hold on to messages
        Callme target;   /* target is the common object for three threads
           Method in this will display the expected output.
           Statements in this method must be executed in a synchronized manner,
           or non-overlapping manner. */

        Thread t;          // object of Thread class required to call start() and run() method.

        public Caller(Callme targ, String s) {
                target = targ;
                msg = s;
                t = new Thread(this);
                t.start();
         }
```

# Multithreaded Programming : Synchronization

```
    public void run() {
            target.call(msg);
    }
}

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

      Callme target = new Callme(); // ***** COMMON DATA SOURCE
      Caller ob1 = new Caller(target, "Hello");
      Caller ob2 = new Caller(target, "Synchronized");
      Caller ob3 = new Caller(target, "World");
/* all threads are simultaneously trying to print the message onto screen which creates race condition*/
```

# Multithreaded Programming : Synchronization

```
    // making the main thread wait for other threads to end its processes.
    try
    {
      ob1.t.join();           ob2.t.join();             ob3.t.join();
    }
    catch(InterruptedException e)
    {System.out.println("Interrupted");}
   }
}
```

**Output:** ***need not be the same output, one of the ways how output will be obtained.***

```
[Hello[Synchronized[World]
]
]
```

## Multithreaded Programming : Synchronization

by calling sleep( ), the call( ) method allows execution to switch to another thread, which results in the mixed-up output of the three message strings.

In this program, all three threads are calling **the same method, on the same object**, at the same time, termed as **race condition,** because the three threads are competing with each other to complete the process.

### Using Synchronized Methods

Instances in Java have their own implicit monitor associated with them, making it easy to achieve synchronization.

(Instances are the data source, which hold on to information.  Synchronization or mutual exclusive access, is required on data sources from methods.)

**“synchronized”** is the keyword used in java to achieve synchronization between several processes accessing the same data source.

## Multithreaded Programming : Synchronization

**When *synchronized* block is used, internally Java uses a monitor also known as intrinsic lock, to provide synchronization.**

**These monitors are bound to an object, thus all synchronized blocks of an object can have only one thread executing/accessing them at any point of time.**

While a thread is inside a synchronized block, making an attempt from other threads to access **the same instance** will be put into the wait state.

To exit the monitor and to give up control of the object to the next waiting thread, control returns from the synchronized method.

In the program “NotSynch.java”, access to call( ) method must be **serialized or synchronized**, by restricting only one thread at a time. To do this, precede call( )’s definition with the keyword **synchronized**

## Multithreaded Programming : Synchronization

```
class Callme
{
   synchronized void call(String msg)   // synchronized methods
   {
      …
   }
}
```

**Only one thread will be allowed to execute/access the synchronized method.**

**If a thread is executing a synchronized method, then no other thread can enter it until it stops executing that method.**

**(Even if the 1st thread that has begun execution and has entered into sleep mode, pre-emption of it is not possible, until the thread itself will relinquish the access)**

**Multithreaded Programming : The synchronized Statement**

Synchronized blocks/methods will not work in all scenarios.

Considering the method belongs to some other class, which is not editable or there is no access to source code, then attaching **synchronized** keyword for the method is not possible.

Solution to this is a synchronized block.

G.F of synchronized statement:

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

Here, object is a reference to the object being synchronized.

*A synchronized block ensures that a call to a method (which is a member of) object occurs only after the current thread has successfully entered the object's monitor.*

# Multithreaded Programming : The synchronized Statement

Ex: Using Synchronized statement

```
class Callme
{
   void call(String msg)
   {
            System.out.print("[" + msg);
            try
            {  Thread.sleep(1000);   }
            catch (InterruptedException e)
            {System.out.println("Interrupted");}

            System.out.println("]");
   }
}
```

# Multithreaded Programming : The synchronized Statement

```
class Caller implements Runnable
{
   String msg;
   Callme target;
   Thread t;

   public Caller(Callme targ, String s)
   {
            target = targ;
            msg = s;
            t = new Thread(this);
            t.start();
   }
```

# Multithreaded Programming : The synchronized Statement

```
        public void run()
        {
                synchronized(target) // synchronized block
                {
                  target.call(msg);
                }
         }
}
```

## Multithreaded Programming : The synchronized Statement

```
class Test   {
  public static void main(String args[])   {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");
        try {
              ob1.t.join();
              ob2.t.join();
              ob3.t.join();
        }
        catch(InterruptedException e)
        {System.out.println("Interrupted");}
  }   }
```

**Multithreaded Programming : Interthread/Interprocess Communication**

Another way to achieve synchronization in java is by using interprocess communication.

*Ex: Consider queuing problem, where one thread is producing some data and another is consuming it.*

*Word document can be considered as a producer and printer can be considered as consumer, where in the document will be printed.*

*Further, assume that the document is ranging in terms of MB size and many pages have to be printed, but the buffer associated with the printer is capable of holding on to only a small amount of information (KB).*

*The producer has to wait until the consumer is finished consuming before it generates(copies) more data(Considering the buffer which is used between them is full).*

# Multithreaded Programming : Interthread/Interprocess Communication

*In a polling system, the consumer would waste many CPU cycles while it was waiting for the producer to produce. Once the producer has finished, it would start polling, just to check whether the consumer has consumed the data, wasting more CPU cycles, and so on. Clearly, this situation is time consuming.*

To avoid polling, Java has an interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods. These methods are implemented as **final methods in Object.**

https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

These methods are declared within Object, as shown here:

```
class Object {
        final void wait( ) throws InterruptedException
                                        //wait( ) method is overloaded
        final void notify( )
        final void notifyAll( )
}
```

## Multithreaded Programming : Interthread/Interprocess Communication

**All three methods can be called only from within a synchronized context.**

**wait( )**
Causes the current thread to wait until either another thread invokes the notify() method
or the notifyAll() method for this object. Current thread must own this object's monitor.

**notify( )**
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.

**notifyAll( )** wakes up all the threads that called wait( ) on the same object.

Alone wait() and notify() will not achieve exact synchronization in IPC.

# Multithreaded Programming : Interthread/Interprocess Communication

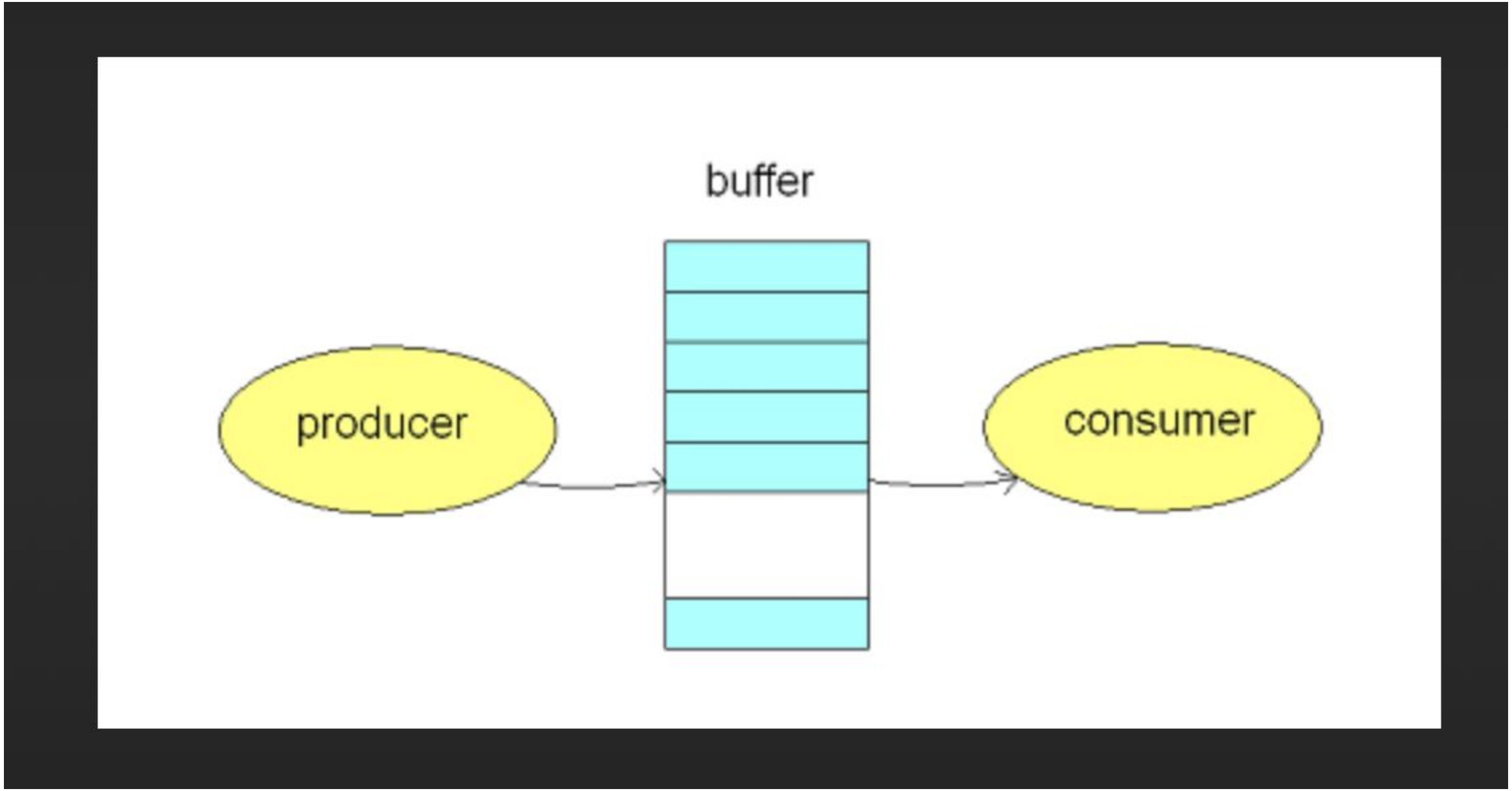
Before calling wait() or notify() a condition has to be checked in a loop to achieve exact synchronization in IPC. The condition will be based on the logical necessity of the application to be solved.

Ex: Simulating a producer and a consumer thread. Assumed that there is only one buffer between these two to hold on to only 1 information.

It is assumed that based on the capacity of the buffer, the producer can produce only 1 information and it has to wait until the consumer consumes this information before producing the next information.

Similarly, consumer can consume only one information, even if it has the capacity of consuming or using more than 1 information, because buffer size is limited to 1.

# Multithreaded Programming : Interthread/Interprocess Communication



## Multithreaded Programming : Interthread/Interprocess Communication

Ex: Following program consists of four classes:

- **Q**, the queue that is being synchronized;
- **Producer**, the threaded object that is producing queue entries;
- **Consumer**, the threaded object that is consuming queue entries; and
- **PC**, class that creates the single **Q**,**Producer**, and **Consumer**.

```
// An incorrect implementation of a producer and consumer without proper IPC.
    class Q  {
     int n;
    /*   "n" is a common variable between producer and consumer thread.
          "n" must be perceived as a buffer.
          Producer thread will fill a value to n.
          Consumer process will pick a value from n.

    get() method will be called by consumer thread to get the value stored in n.
    put() method will be called by producer thread to put a value into n. */
```

# Multithreaded Programming : Interthread/Interprocess Communication

```
    synchronized void  get()   // used by the consumer process
    {
     System.out.println("Got: " + n);
    }

    synchronized void put(int n) //used by the producer process
    {
      this.n = n;
      System.out.println("Put: " + n);
    }
}
```

# Multithreaded Programming : Interthread/Interprocess Communication

```
//Producer thread
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++);
   }   }
```

# Multithreaded Programming : Interthread/Interprocess Communication

```
//Consumer thread
class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
     this.q = q;
     new Thread(this, "Consumer").start();
  }

  public void run() {
    while(true)
      q.get();
  }
}
```

# Multithreaded Programming : Interthread/Interprocess Communication

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

   System.out.println("Press Control-C to stop.");
 }
}
```

**Output:**

```
Put:1
Got:1
Got:1
Got:1
Got:1
```

# Multithreaded Programming : Interthread/Interprocess Communication

Just achieving synchronization between threads is not sufficient for producer-consumer problem, in addition to this, two conditions have to be checked for p-c problem to work in a competent manner.

1. First condition is w.r.t to consumer, whether buffer is filled before consumer consumes from it and

2. Second condition with respect to the producer will be that before the producer fills the buffer with information it is necessary that the buffer is empty.

# Multithreaded Programming : Interthread/Interprocess Communication

**// A correct implementation of a producer and consumer.**

```
class Q
{
        int n;
        volatile boolean valueSet = false;

/*"valueSet" is a common variable used between producer and consumer process to know
  whether the buffer is full or not.
*/
```

# Multithreaded Programming : Interthread/Interprocess Communication

```
synchronized void get()
{
        while(!valueSet)
           try
           {    wait();     }
           catch(InterruptedException e)
           { System.out.println("InterruptedException caught"); }

        System.out.println("Got: " + n);
        valueSet = false;
        notify();
}
```

# Multithreaded Programming : Interthread/Interprocess Communication

```
        synchronized void put(int n)
        {
           while(valueSet)
            try {
              wait();
            }
            catch(InterruptedException e)
            { System.out.println("InterruptedException caught"); }
            this.n = n;
            valueSet = true;
            System.out.println("Put: " + n);
            notify();
         }
}//end of class Q
```

## Multithreaded Programming : Interthread/Interprocess Communication

```
class producer implements Runnable
{
        ………..
```

**Bounded buffer problem is nothing but producer-consumer problem**