Module 4

Interfaces
2 forms of creating abstract types in java
1. Abstract classes
2. Interface

Interface is a mechanism to achieve abstraction (specifying what a class must do but not how it does it).

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 interface.

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

Interfaces are used to achieve multiple inheritance in Java.

Java Interface also represents the **"IS-A"** relationship.

A single class can inherit multiple interfaces.

A single class **cannot** inherit multiple abstract classes.

"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;
}
```

Ex:

```
interface   test
{    }

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

If a class **inherits or implements** an interface "test", then all the methods in the test must be compulsorily

defined by the inherited class.

All methods and variables are implicitly public.

NOTE: An interface can extend (inherit) another 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");   }
}

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
```

```
  }
}
```

Ex:

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

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

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

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

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

   /* cmp i = new cmp(9,0);  // i is a reference of type Integer
       Object obj = i;  // generates   java.lang.ClassCastException
       int a = ((Integer) i).intValue(); /// Exception.
   */
  }

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

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

```
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();
  }

  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(new Integer(10));
     System.out.println(   ((Integer)i.pop()).intValue()   );



     fstack j = new fstack();
     j.push(new Float(10));
     System.out.println(   ((Float)j.pop()).floatValue()   );

     System.out.println("One interface Multiple method invocation");
    stack s = i;// base class reference pointing to its derived class

     s.push(new Integer(20));
     System.out.println(   ((Integer)s.pop()).intValue()   );
```

```
    s = j;
    s.push(new Float(30));
    System.out.println(   ((Float)s.pop()).intValue()   );
  }
}
```

IMPLEMENTING MULTIPLE INTERFACE & INTERFACE EXTENDING ANOTHER INTERFACE

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

interface display_value
{   void display();   }

/*
interface accept_value
{    void accept();   }

//INTERFACE INHERITING PROPERTIES OF ANOTHER INHERITANCE
interface display_value extends accept_value
{
        void display();
}*/

//class   test   implements   display_value
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();
```

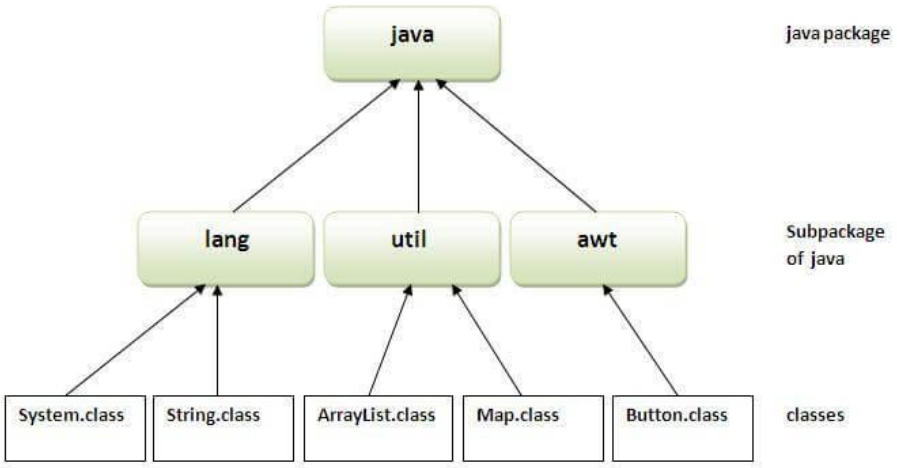
```
    }
}
```

Packages
****

A java package is a group of similar types of classes, interfaces and sub-packages.

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 provides access protection.

Package removes naming collision.

Package is used to categorize the classes and interfaces so that they can be easily maintained.

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 rns.engineering.cse
```

rns is the main directory and main package name
engineering is the sub directory of rns and intermediate package name
cse is the subdirectory of engineering and final package name

rns ---> engineering ----> cse

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

Creating packages
****

package rns.engineering;

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

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

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

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

Table: Class member access

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-sub-classes cannot access protected members.

When a member 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).

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 is **final**, it cannot be inherited by another 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;
}
```

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;   }     }
```

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;  }     }
```

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;
        }
}
```

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    }    }
```

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();    }       }
```

Ex:

File: p1\Protection.java

```
package p1;
public class Protection
{
  int n = 1;
  private int n_pri = 2;
  protected int n_pro = 3;
  public int n_pub = 4;

  public Protection() {
    System.out.println("Protection constructor");
    System.out.println("n = " + n);
    System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

File: p1\Derived.java

```
package p1;
class Derived extends Protection {
  Derived() {
    System.out.println("Derived constructor");
    System.out.println("n = " + n);

    //class only
    //System.out.println("n_pri = "4 + n_pri);

    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
   }
}
```

File: p1\SamePackage.java

```
package p1;

class SamePackage {
    SamePackage() {
      Protection p = new Protection();
      System.out.println("Same package non-derived constructor");
```

```
      System.out.println("n = " + p.n);

      //class only
      //System.out.println("n_pri = " + p.n_pri);

            System.out.println("n_pro = " + p.n_pro);
            System.out.println("n_pub = " + p.n_pub);
   }
}//End of class SampePackage
```

File: p2\Protection2.java

```
package p2;
import p1.*;

class Protection2 extends p1.Protection {
  Protection2() {
  System.out.println("derived other package constructor");
  // class or package only
  //System.out.println("n = " + n);

  // class only
  //System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

File: p2\ OtherPackage.java

```
package p2;
import p1.*;

class OtherPackage {
  OtherPackage() {
    p1.Protection p = new p1.Protection();
    System.out.println("other package constructor");

   // class or package only
   // System.out.println("n = " + p.n);

   // class only
   // System.out.println("n_pri = " + p.n_pri);

   // class, subclass or package only
   // System.out.println("n_pro = " + p.n_pro);
   System.out.println("n_pub = " + p.n_pub);
}
}
```

```
// Demo package p1.
package p1;
package p2;

// Instantiate the various classes in p1.
public class Demo {
  public static void main(String args[]) {
    Protection ob1 = new Protection();
    Derived ob2 = new Derived();
    SamePackage ob3 = new SamePackage();

        Protection2 ob1 = new Protection2();
        OtherPackage ob2 = new OtherPackage();
  }
}
```

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 builtin 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.

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, you specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package or all the classes in the package.

Ex:

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

Multi-Threaded 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.

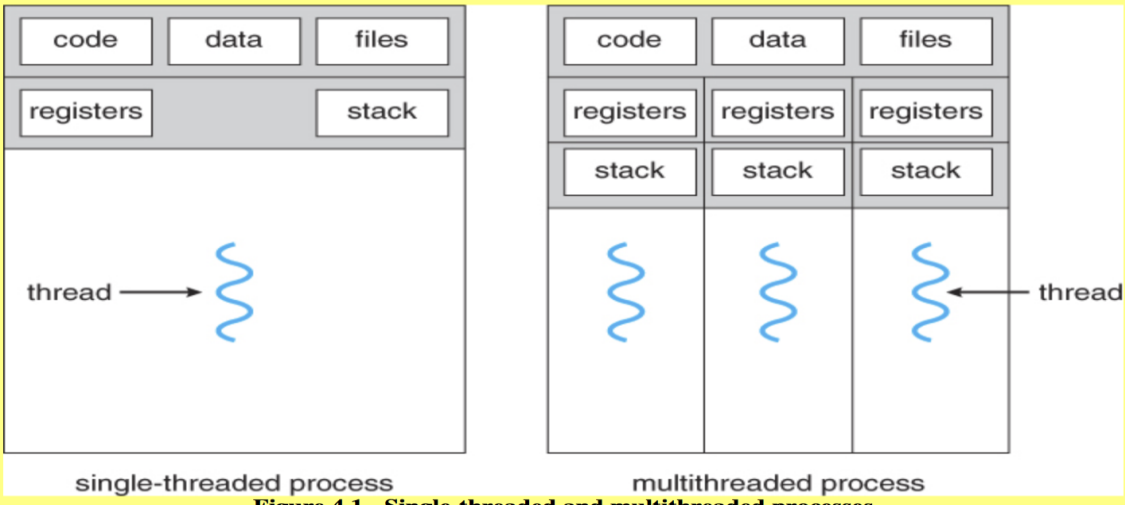
Processes are heavyweight tasks that require their own separate **address spaces**.
*/****Address space*** *is the amount of memory allocated for all possible addresses for a computational entity, such as a device, a file, a server, or a networked computer.*

Address space may refer to a range of either physical or virtual addresses accessible to a processor or reserved for a process.

On a computer, each computer device and process is allocated address space, which is some portion of the processor's address space/*

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

Thread-based multitasking
Thread is the smallest unit of dispatchable/executable code.



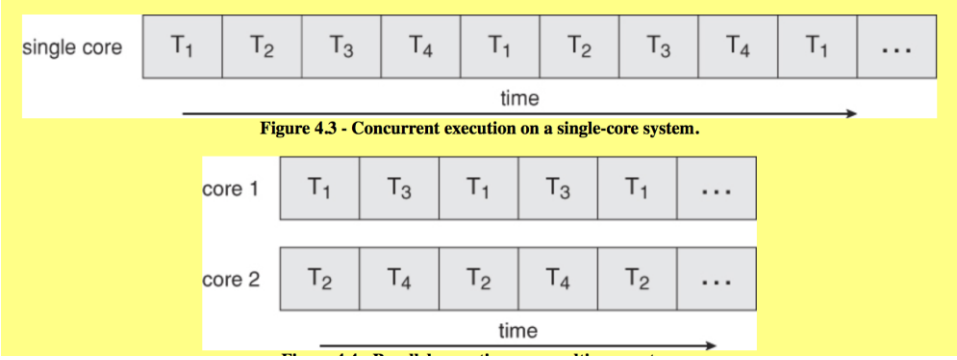


Figure 4.3 - Concurrent execution on a single-core system.

https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

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".

Threads are a lightweight process.
They share the same address space and cooperatively share the same 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.

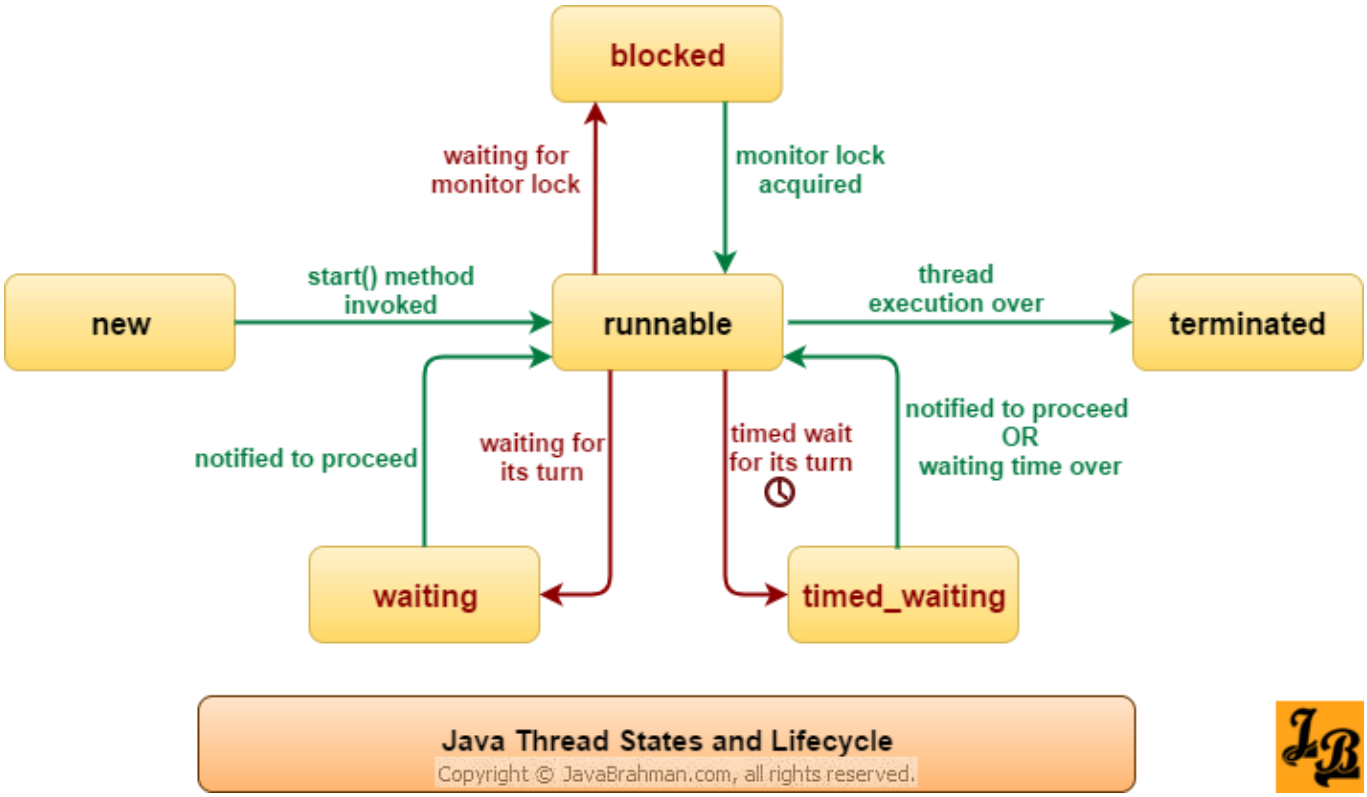
Multithreading enables to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.
Ex: Multithreading is especially important for the interactive, networked environment in which Java operates, because idle time is common.

The 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. And, user input is much slower than the computer.

In a single-threaded environment, a program has to wait for each of these tasks to finish before it can proceed to the next one—even though the CPU is sitting idle most of the time.
Multithreading lets you gain access to this idle time and put it to good use.

Thread states

****



"Thread" Class and "Runnable" Interface

****

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

Thread class, encapsulates a thread under execution.

To create a new thread, the program will either e**xtend Thread o**r i**mplement t**he R**unnable 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**.

The Main Thread

****

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
}
```

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( ).

```
// Controlling the main Thread.
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);

        try   // becz, Thread.sleep( ) is anticipated to generate 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

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)

Creating a Thread
****
A thread is generated by **instantiating an object of the builtin class Thread**.

Java has two ways of creating a thread:
• implement the Runnable interface.
• extend the Thread class.

Implementing Runnable
****
Generating a thread in java, implementing Runnable interface is a two step process.

1) A way to generate a thread is to create a class that implements the Runnable interface.

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

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);
}
```

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.

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

  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
  }

 // This is the entry point for thread.
 @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

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

     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
```

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 this object.

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, it enters its for loop.

Both threads continue running, sharing the CPU, until their loops finish.

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.
```

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.

Extending Thread
****
Another way to create a thread is to create a new 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.
  }

 // This is the entry point for the second thread.  Overriding run method, which is present in "Thread"
 // class.
   @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
```

```
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.

Whether to use *implements* or *extends* 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 no other thread methods are overridden other than run( ) method, then it is best suggested to implement Runnable.

Creating 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
  }


// This is the entry point for thread.
  @Override
  public void run() {
/*!!!
```

```
  super.run();
  Is a valid statement at this point.  Generic run() in Thread class does not do any specific task and is
  negligible.
*/
    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.");
  }   }


class Test  {
public static void main(String args[]) {
          new NewThread("One"); // start threads
          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.");
 }  }
```

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.

isAlive( ) and join( )
****

Making main( ) thread to sleep for a long duration of time, 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 current thread on wait until the thread on which it has called finishes its execution.

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(); // Start the thread
 }

@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

class Test  {
  public static void main(String args[])   {
     NewThread nt = new NewThread("First"); // create a new thread
     NewThread nt1 = new NewThread("Second");

     try   {
                nt.join();
                nt1.join();
/* Since nt.join( ) is called on main thread, main thread suspends execution until thread named "First"
   completes its execution.
   Once nt thread completes its execution, main( ) thread resumes execution and subsequent call to
   nt.join( ) will not execute, because already nt thread has finished its execution.
*/
     }
     catch (InterruptedException e)    {System.out.println("Main thread interrupted.");}
    System.out.println("Main thread exiting.");
  }   }//End of class ThreadDemo
```

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.
```

Ex: using isAlive( ) method

```
class NewThread extends Thread   {

 NewThread(String n)
 {
   this.setName(n);
   System.out.println("Child thread: " + this);
   start(); // Start the thread
 }

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
```

```
class Test {
  public static void main(String args[]) {
    NewThread nt = new NewThread("First"); // create a new thread
    NewThread nt1 = new NewThread("Second");

        while(nt.isAlive());

        while(nt1.isAlive());

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

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.
```

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 if assumed to be 10 comes in 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"**.

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( )
}
```

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

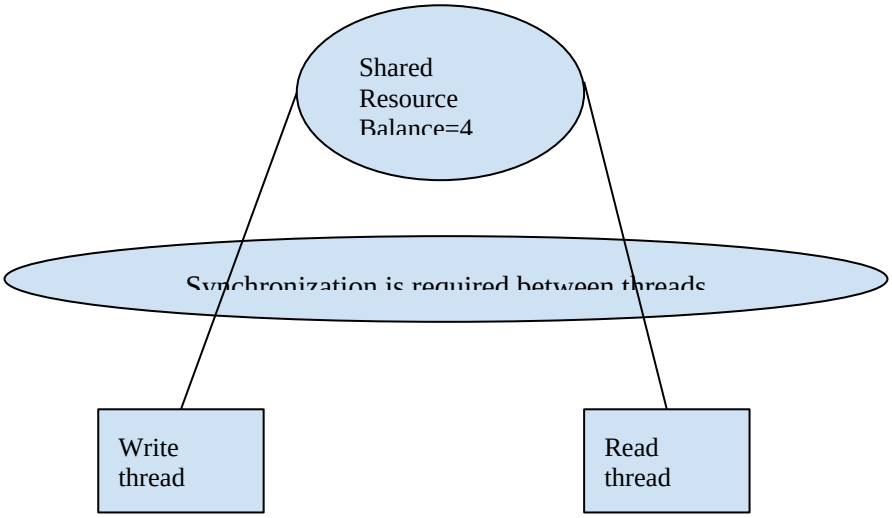
Ex:

```
//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++;
        }

        public void stop()     { running = false; }

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

class test   {
public static void main(String args[])    {
         (Thread.currentThread()).setPriority(Thread.MAX_PRIORITY);

         clicker hi = new clicker(Thread.NORM_PRIORITY + 4);
         clicker lo = new clicker(Thread.NORM_PRIORITY - 2);

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

 lo.stop();
 hi.stop();


// Wait for child threads to terminate.
```

```
 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);
}
}
```

Synchronization
****

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



Synchronization is a mechanism to achieve exclusive access to shared resources by more than one thread.

Key to synchronization is the concept of monitor (also called a semaphore).

/*
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

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 block and wait for entry to the monitor

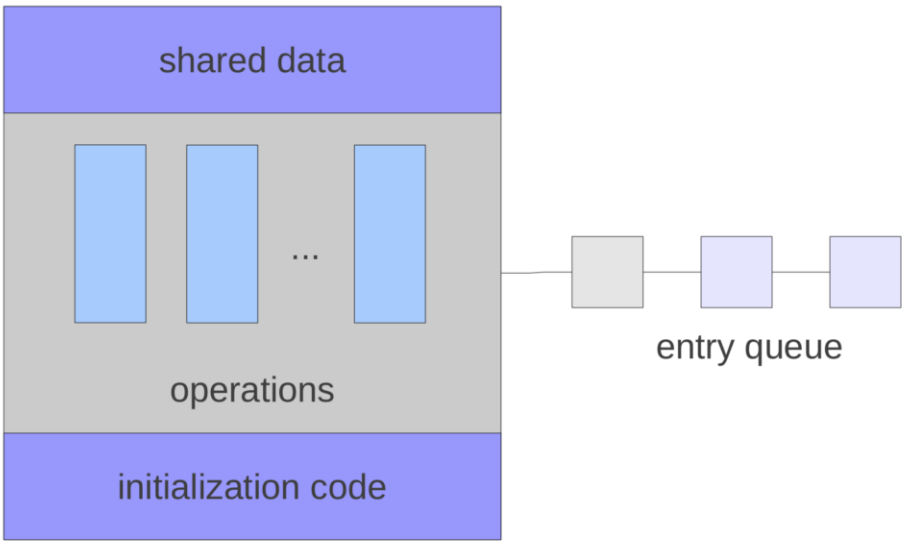
Need for a wait queue

Structure of a Monitor

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

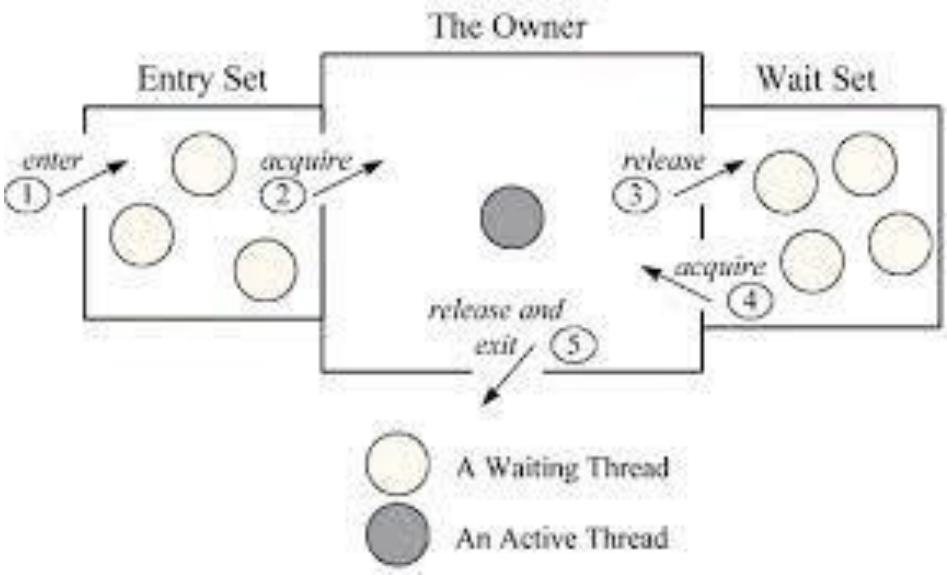


*/

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.



Ex: Expected output of the below program is

[Hello]
[Synchronized]
[World]

By using 3 different threads, **without any synchronization** between them.

PGM NAME: NotSynch

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

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

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();
   }

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

class NotSynch {
   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");

/* all threads are simultaneously trying to print the message onto screen which creates race condition*/
     // 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]
]
]
```

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, hence it is easy to achieve synchronization in Java.

(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.

When *synchronized* block is used, internally Java uses a monitor also known as monitor lock or intrinsic lock, to provide synchronization. These monitors are bound to an object, thus all synchronized blocks of the same object can have only one thread executing them at any point of time.

While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) **on the same instance** have to wait.

To exit the monitor and to give up control of the object to the next waiting thread, the owner of the monitor (which is a method which belongs to a thread) simply 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**

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

Only one thread will be allowed to execute in 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)

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.

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("]");
   }   }

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();
 }

 // synchronize calls to call()
 public void run() {
  synchronized(target)
  { // synchronized block
    target.call(msg);
  }
}     }

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");

    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    }
```

```
    catch(InterruptedException e)
    {System.out.println("Interrupted");}
  }
}
```

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 document is ranging in terms of Mb size and pages together contents have to be printed, but the buffer associated with the printer is capable of holding on to only kb amount of information.

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).

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

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

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( ) method.

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

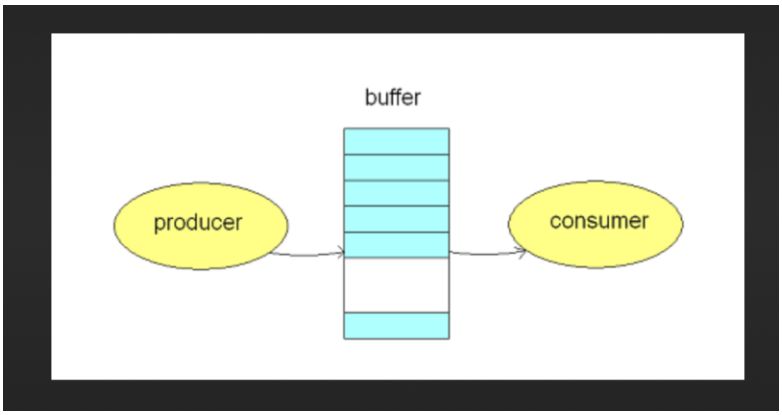
Alone wait() and notify() will not achieve exact synchronization in IPC.

Before calling wait() or notify() a condition has to be checked in a loop to achieve exact synchronization in IPC.

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.



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.
*/
  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);
   }
}

//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++);
   }    }
//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();
  }
}


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
```

/*

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 must be empty.

***/**

```
// 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.
*/
        synchronized void get()
        {
          while(!valueSet)
            try {
              wait();
            }
            catch(InterruptedException e)
            { System.out.println("InterruptedException caught"); }
          System.out.println("Got: " + n);
          valueSet = false;
          notify();
        }

        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


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