Multithreaded Programming

- Java provides built-in support for multi-*threaded programming. A multithreaded program contains two or more* parts that can run concurrently.
- Each part of such a program is called a *thread,* and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

- There are two distinct types of multitasking: process-based and thread-based.
- A *process is, in essence, a program that is executing. Thus, process-based* multitasking is the feature that allows our computer to run two or more programs concurrently. Like running the Java compiler at the same time using a text editor.
- In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
- In a *thread-based multitasking environment, the thread is the smallest unit of* dispatchable code. This means that a single program can perform two or more tasks simultaneously. Like a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

- Multitasking threads require less overhead than multitasking processes. 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.
- Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost.
- Multithreading enables us to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

The Java Thread Model

- The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind.
- In general, in a singled-threaded environment, when a thread *blocks (that is, suspends* execution) because it is waiting for some resource, the entire program stops running.
- The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program.
- When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

- Threads exist in several states. A thread can be *running*. *It can be ready to run as* soon as it gets CPU time. A running thread can be *suspended*, *which temporarily* suspends its activity.

- A suspended thread can then be *resumed*, *allowing it to pick up* where it left off.

- A thread can be *blocked when waiting for a resource. At any time, a* thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

Thread Priorities

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another.
- A thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch. The rules that determine* when a context switch takes place are simple:

1. *A thread can voluntarily relinquish control.*
2. *A thread can be preempted by a higher-priority thread.(*This is called *preemptive multitasking.)*

Synchronization

- Because multithreading introduces an asynchronous behavior to your programs, there must be a way for us to enforce synchronization when we need it. For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, we need some way to ensure that they don't conflict with each other.
- Java uses the *monitor for this. The monitor is a control mechanism* first defined by C.A.R. Hoare.
- A monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.
- Each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object.

Messaging

- Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

The Thread Class and the Runnable Interface

- Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. Thread encapsulates a thread of execution.
- To create a new thread, our program will either extend Thread or implement the Runnable interface.

- The Thread class defines several methods that help manage threads. Like
- 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.

The Main Thread

- When a Java program starts up, one thread begins running immediately. This is usually called the *main thread of your program, because it is the one that is executed* when your program begins. The main thread is important for two reasons:

1. It is the thread from which other “child” threads will be spawned.
2. Often it must be the last thread to finish execution because it performs various shutdown actions.

- Although the main thread is created automatically when your program is started, it can be controlled through a Thread object. To do so, you must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread. Its general form is shown here:
- static Thread currentThread( )

```
// Controlling the main Thread.
class CurrentThreadDemo {
public static void main(String args[]) {
Thread t = Thread.currentThread();
System.out.println("Current thread: " + t);
// change the name of the thread
t.setName("My Thread");
System.out.println("After name change: " + t);
try {
for(int n = 5; n > 0; n--) {
System.out.println(n);
Thread.sleep(1000);
} }
catch (InterruptedException e) {
System.out.println("Main thread interrupted");
} } }
```

o/p of the code is->

Current thread: Thread[main,5,main]

After name change: Thread[My Thread,5,main]

5

4

3

2

1

- This displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is **main. Its priority is 5, which is the default value, and main** is also the name of the group of threads to which this thread belongs. A *thread group is* a data structure that controls the state of a collection of threads as a whole.

- The general form of sleep() is->

  static void sleep(long *milliseconds) throws InterruptedException*

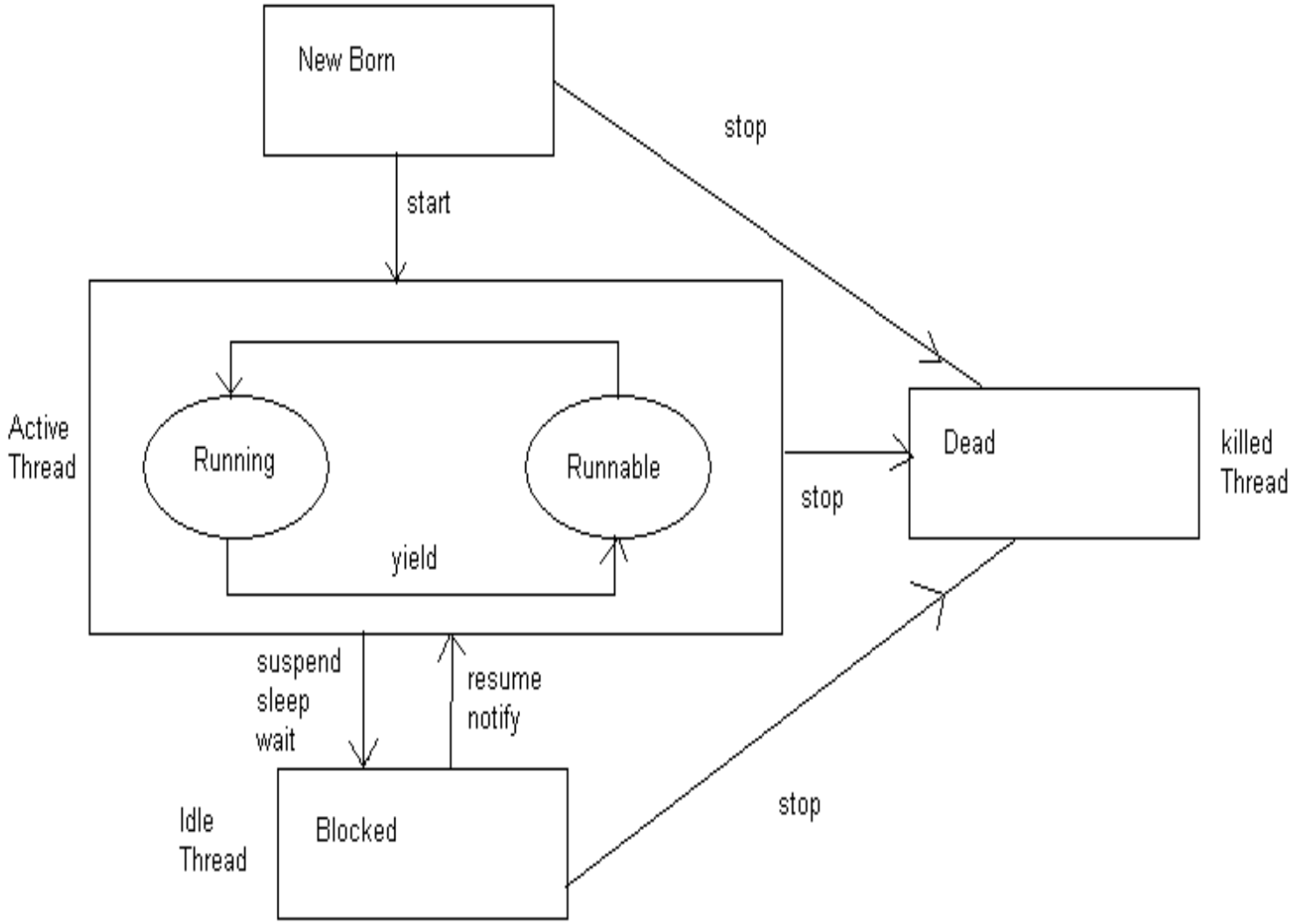
  static void sleep(long *milliseconds, int nanoseconds) throws InterruptedException*
- This second form is useful only in environments that allow timing periods as short as nanoseconds.
- The two more methods of Thread class are

  final void setName(String *threadName)*

  final String getName( )

Creating a Thread

- In the most general sense, we create a thread by instantiating an object of type **Thread.**
- Java defines two ways in which this can be accomplished:
- implement the **Runnable interface.**
- extend the **Thread class.**


New Born
stop
start
Active
Thread
Running
Runnable
yield
Dead
killed
Thread
stop
suspend
sleep
wait
resume
notify
Idle
Thread
Blocked
stop

Extend the Thread class

```
class A extends Thread {
 public void run()  {
  for(int i=1;i<=5;i++)
   {  System.out.println("\t From Thread A:i= "+i); }
    System.out.println("Exit From A");
 }
}
class B extends Thread{
 public void run() {
  for(int j=1;j<=5;j++)
  {    System.out.println("\t From Thread B:j= "+j);  }
  System.out.println("Exit From B");
 }
 }
```

```
class C extends Thread{
 public void run() {
  for(int k=1;k<=5;k++)
  {    System.out.println("\t From Thread C:k= "+k); }
  System.out.println("Exit From C");
 }
}
class test {
 public static void main(String args[])
 {
  new A().start(); // A thread1=new A(); thread1.start();
  new B().start();
  new C().start();
 }
}
```

```
C:\>java test
        From Thread A:i= 1
        From Thread B:j= 1
        From Thread B:j= 2
        From Thread A:i= 2
        From Thread B:j= 3
        From Thread A:i= 3
        From Thread A:i= 4
        From Thread A:i= 5
Exit From A
        From Thread B:j= 4
        From Thread B:j= 5
Exit From B
        From Thread C:k= 1
        From Thread C:k= 2
        From Thread C:k= 3
        From Thread C:k= 4
        From Thread C:k= 5
Exit From C
```

Implementing Runnable

```
class A implements Runnable{
 public void run()  {
  for(int i=1;i<=5;i++)
  {    System.out.println("\t From Thread A:i= "+i);
   }
   System.out.println("Exit From A");
 }
}
class test{
 public static void main(String args[])  {
  A obj_1=new A();
  Thread thread1= new Thread(obj_1);
  thread1.start();                    //new Thread(  new A()  ).start()
  System.out.println("Exit From Main Thread");
  }
}
```

```
C:\>java test
Exit From Main Thread
        From Thread A:i= 1
        From Thread A:i= 2
        From Thread A:i= 3
        From Thread A:i= 4
        From Thread A:i= 5
Exit From A
```

```
// Create a second thread.
class NewThread implements Runnable {
Thread t;
NewThread() {    // Create a new, second thread
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for the second thread.
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.");
}
}
```

```
class ThreadDemo {
public static void main(String args[]) {
NewThread t1= new NewThread();
//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.");
}  }
```

- 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.

- To implement Runnable, a class need only implement a single method called run( ),which is declared like this:

  public void run( )
- Inside run( ), we will define the code that constitutes the new thread.
- Thread defines several constructors. Like

  Thread(Runnable *threadOb, String threadName)*

  Thread(Runnable threadOb)
- In this constructor, *threadOb is an instance of a class that implements the Runnable* interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName.*
- After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread. In essence, start( ) executes a call to run( ). The start( ) method is shown here:

  void start( )

Choosing an Approach

- Why Java has two ways to create child threads, and which approach is better. The answers to these questions turn on the same point.
- The **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( ).*** *This is, of course, then* same method required when you **implement Runnable**.
- Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if we will not be overriding any of Thread's other methods, it is probably best simply to implement Runnable.
- This is actually up to us to choose an approach.

Using isAlive( ) and join( )

- We want the main thread to finish last. In the preceding examples, this is accomplished by calling sleep( ) within main( ), with a long enough delay to ensure that all child threads terminate prior to the main thread. But it is unsatisfactory solution.
- **How can one thread know when another thread has ended?**
- Thread provides a means by which you can answer this question. **Two ways exist to determine whether a thread has finished**. First, you can call **isAlive( )** on the thread.
- This method is defined by Thread, and its general form is

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

- While **isAlive( ) is occasionally useful, the method that you will more commonly** use to wait for a thread to finish is called **join( ):**

  final void join( ) throws InterruptedException
- This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins it.*
- Additional forms of **join( ) allow you to specify a maximum amount of time that** you want to wait for the specified thread to terminate.

```
class C extends Thread{
   C(String name) {
       super(name);  }
  public void run() {
    for(int k=1;k<=5;k++){
      System.out.println("\t From Thread "+this.getName()+" :k= "+k);
  }
  System.out.println("Exit From "+this.getName());
 }  }
class test {
 public static void main(String args[]) {
  C th1=new C("th1");
  C th2=new C("th2");
  C th3=new C("th3");
  th1.start();
  th2.start();
  th3.start();
```

```
  System.out.println("Thread One is alive: "+ th1.isAlive());
  System.out.println("Thread Two is alive: "+ th2.isAlive());
  System.out.println("Thread Three is alive: "+ th3.isAlive());

  try {
   System.out.println("Waiting for threads to finish.");
   th1.join();
   th2.join();
   th3.join();
  }
 catch (InterruptedException e) {
  System.out.println("Main thread Interrupted");
}
 System.out.println("Main thread exiting.");
}
}
```

```
C:\>java test
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
        From Thread th2 :k= 1
        ……..
        From Thread th2 :k= 5
Exit From th2
        From Thread th3 :k= 1
        ……..
        From Thread th3 :k= 5
Exit From th3
        From Thread th1 :k= 1
        From Thread th1 :k= 2
        From Thread th1 :k= 3
        From Thread th1 :k= 4
        From Thread th1 :k= 5
Exit From th1
Main thread exiting.
```

- In the previous code the join method in main waits for all the child threads to complete. Thus the main thread completes in the last.
- If join statement are removed then it may be possible that the main thread exits before the child threads.
- The case may also occur that before printing the status of a child thread in main thread(by isAlive() method), the child thread is already finished/exit. In that case the isAlive() will return false.

```
C:\>java test
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Main thread exiting.
        From Thread th3 :k= 1
        .....
        From Thread th3 :k= 5
Exit From th3
        From Thread th1 :k= 1
        .....
        From Thread th1 :k= 5
Exit From th1
        From Thread th2 :k= 1
        .......
        From Thread th2 :k= 5
Exit From th2
```

```
C:\>java test
        From Thread th1 :k= 1
        ......
        From Thread th3 :k= 5
Exit From th3
        From Thread th2 :k= 1
        From Thread th2 :k= 2
Thread One is alive: true
        From Thread th2 :k= 3
        .....
        From Thread th2 :k= 5
Exit From th2
        From Thread th1 :k= 4
        From Thread th1 :k= 5
Exit From th1
Thread Two is alive: true
Thread Three is alive: false
Main thread exiting.
```

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads.
- In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.)
- A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.
- To set a thread's priority, use the setPriority( ) method, which is a member of Thread. This is its general form:

final void setPriority(int *level)*

- *level specifies the new priority setting for the calling thread. The value of level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5.
- These priorities are defined as final variables within Thread class.
- The high priority should be used very carefully as it affect the other threads.

```
class A extends Thread {
 int count1;
 private volatile boolean flag1 = true;
 public void run()   {
  while(flag1)   {
     count1++;
    }  }
  public void stop1() {
   flag1 = false;
  } }

class B extends Thread{
 int count2;
 private volatile boolean flag2 = true;
 public void run() {
  while(flag2)  {
   count2++;
  }  }
 public void stop1() {
  flag2 = false;
 } }
```

```
class test {
 public static void main(String args[])  {
  A threadA =new A();
  B threadB =new B();
  threadB.setPriority(Thread.MAX_PRIORITY);//5+1
  threadA.setPriority(Thread.MIN_PRIORITY);//1

  System.out.println("Start Thread A");   threadA.start();
  System.out.println("Start Thread B");   threadB.start();

  try {       Thread.sleep(1000);   }
   catch (InterruptedException e) {
          System.out.println("Main thread interrupted.");   }

   threadA.stop1();
   threadB.stop1();
  System.out.println("High-priority thread: " + threadB.count2);
  System.out.println("Low-priority thread: " + threadA.count1);
  System.out.println("End of Main Thread ");
 }  }
```

```
C:\>java test
Start Thread A
Start Thread B
High-priority thread: 461821456
Low-priority thread: 439896882
End of Main Thread
```

- The output of this code is not fixed and will be different for each run. But the count value of high priority thread will always be higher then that of lower priority thread.

Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization.*
- *Java provides unique,* language-level support for it. Key to synchronization is the concept of the monitor (also called a *semaphore).*
- *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. These other threads are said to* be *waiting for the monitor. A thread that owns a monitor can reenter the same monitor* if it so desires.

- We can synchronize our code in either of two ways. Both involve the use of the synchronized keyword, and both are examined here.
- **Using synchronized Methods**
- **The synchronized Statement**
- **synchronized Methods :** All objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the synchronized keyword. 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 relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

```
class callme{
 static int i=1;
 //synchronized
static void print() {
   System.out.print("( HI ");
   try {     Thread.sleep(1000);
     }
   catch(InterruptedException e) {     System.out.println("Interrupted");
     }
    System.out.println(i+" )");
    i++;
   }   }
class A extends Thread {
 public void run()  {
 for(int i=1;i<=5;i++)
  callme.print();
} }
```

```
class B extends Thread{
 public void run() {
  for(int i=1;i<=5;i++)
  callme.print();
 }
}
class test {
 public static void main(String args[])
 {
  A th1= new A();
  th1.start(); // new A().start();
  B th2=new B();
  th2.start();
  }
}
```

(HI (HI 1)
1)
(HI (HI 3)
(HI 3)
(HI 5)
(HI 5)
(HI 7)
(HI 7)
(HI 9)
9)

- Here the value of variable i is updated in unordered way by both the threads A and B. The output may change for each run. This can lead to a series problem if i is some important shared data or a control flag. The output printing is also not in order. To solve the problem we need **synchronization** over the print() method.
- If the print() method is defined as:

  synchronized static void print(){---------}

- Then the method is synchronized and the output will be->

(HI 1)
(HI 2)
(HI 3)
(HI 4)
(HI 5)
(HI 6)
(HI 7)
(HI 8)
(HI 9)
(HI 10)

- Also the output will remain same for each run.(As only one thread can execute the print() method at a time and other thread has to wait for completion of current call to print method.)

- **The synchronized Statement:** While creating synchronized methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases.
- Consider the following. Imagine that we want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use synchronized methods. Further, this class was not created by us, but by a third party, and we do not have access to the source code. Thus, we can't add synchronized to the appropriate methods within the class. How can access to an object of this class be synchronized?
- The solution to this problem is quite easy: simply put calls to the methods defined by this class inside a **synchronized block**.

```
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 that is a member of *object occurs only after the current* thread has successfully entered *object's monitor.*

```
class callme{
 int i=1;          //non-static member
 void print()  {  //non-static member
  System.out.print("( HI ");
   try {     Thread.sleep(1000);   }
   catch(InterruptedException e) { System.out.println("Interrupted"); }
  System.out.println(i+" )");
  i++;
 }  }
class A extends Thread {
 callme obj;
 A(callme target) {  obj=target;  }
public void run()  {
 for(int i=1;i<=5;i++) {
  synchronized(obj)
  { obj.print(); } }
}  }
```

```
class B extends Thread{
 callme obj;
 B(callme target){  obj=target;  }
 public void run() {
  for(int i=1;i<=5;i++)
  { synchronized(obj)
   {obj.print(); } }
 } }
class test {
 public static void main(String args[]) {
  callme obj1=new callme();
  A th1= new A(obj1);
  th1.start(); // new A().start();
  B th2=new B(obj1);
  th2.start();
  }
}
```

- Here both thread are referring to the same instance/object of class callme. The call to method print() is in synchronized block this time.

(HI 1)
(HI 2)
(HI 3)
(HI 4)
……..
(HI 9)
(HI 10)

So here without changing the original class callme the call to print method is executed in synchronized block. This will have the same effect as the previous case.(Where the print method is ynchronized.)

Interthread Communication

- Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time.
- To avoid polling, Java includes an elegant interprocess communication mechanism via the **wait( ), notify( ),** and **notifyAll( )** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. 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( ).
- **notify( )** wakes up the first thread that called wait( ) on the same object.
- **notifyAll( )** wakes up all the threads that called wait( ) on the same object. The highest priority thread will run first.

- These methods are declared within Object, as shown here:

  final void wait( ) throws InterruptedException

  final void notify( )

  final void notifyAll( )

```
class A {
public static void main(String[]args)throws InterruptedException {
B b =new B();
b.start();
synchronized(b)     //thread got lock
{
System.out.println("Calling wait method");
b.wait();
System.out.println("Got notification");  }
System.out.println(b.total);
}  }
class B extends Thread{
int total=0;
public void run() {
synchronized (this) //.thread got lock
{
System.out.println("Starting calculation");
for(int i=0;i<=1000;i++)  {  total=total+I;  }
System.out.println("Giving notification call");
notify();    //thread releases lock again      }} }
```

Calling wait method

Starting calculation

Giving notification call

Got notification

500500

Deadlock

- A special type of error that we need to avoid that relates specifically to multitasking is *deadlock, which occurs when two threads have a circular dependency on a pair of* synchronized objects.
- For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.
- The programmer need to take care of such things while using the multithreading.

Suspending, Resuming, and Stopping Threads

- Prior to Java 2, a program used suspend( ) and resume( ), which are methods defined by Thread, to pause and restart the execution of a thread. They have the form shown below:

  final void suspend( )

  final void resume( )
- The Thread class also defines a method called stop( ) that stops a thread. final void stop( )
- Once a thread has been stopped, it cannot be restarted using resume( ).
- While the suspend( ), resume( ), and stop( ) methods defined by Thread seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs. All these method of the Thread class are deprecated in Java 2.

--The older method. Gives warning for deprecation.

```
class A extends Thread{
 public void run()  {
  try{
  for(int i=1;i<=5;i++)
  {
      System.out.println("\t From Thread :i= "+i);
      Thread.sleep(500);
   }
   }
   catch (InterruptedException e) {
   System.out.println("Thread Interrupted");
}
  System.out.println("Exit From Thread");
 }
}
```

```
class test{
 public static void main(String args[])  {
  A thread1= new A();
    thread1.start();
  try {
     Thread.sleep(1000);
     thread1.suspend();
      System.out.println("Suspending thread One");
     Thread.sleep(1000);
     thread1.resume();
     System.out.println("Resuming thread One");
    }
 catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
 System.out.println("Main thread exiting.");
}
}
```

```
From Thread :i= 1
        From Thread :i= 2
Suspending thread One
Resuming thread One
Main thread exiting.
        From Thread :i= 3
        From Thread :i= 4
        From Thread :i= 5
Exit From Thread
```

- In Java 2 the approach in that a thread must be designed so that the **run( ) method periodically checks to determine whether that thread should suspend,** resume, or stop its own execution. Typically, this is accomplished by establishing a flag variable that indicates the execution state of the thread.

Using Multithreading

- The key to utilizing multithreading support effectively is to think concurrently rather than serially. For example, when we have two subsystems within a program that can execute concurrently, make them individual threads. With the careful use of multithreading, you can create very efficient programs.
- But if we create too many threads, we can actually degrade the performance of our program rather than enhance it. Some overhead is associated with context switching. If we create too many threads, more CPU time will be spent changing contexts than executing our program!