



Java Programming

Unit – 4

Multi Threading

Dr. Y. J. Nagendra Kumar


Professor of IT

Dean - Technology and Innovation Cell - GRIET





TABLE OF CONTENTS

- 
- 01 Creating Threads
 - 02 Thread Life Cycle
 - 03 Synchronization
 - 04 Inter Thread Communication



Creating Threads



Multi Tasking

- There are two distinct types of multitasking:
 1. Process-based
 2. Thread-based.
- **Process-based** multitasking is the feature that allows our computer to run two or more programs concurrently.
- In a **Thread-based** multitasking environment, a single program can perform two or more tasks simultaneously.





Thread

- 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.
- Multithreading is a specialized form of multitasking.
- Multithreading enables us to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.



Difference between Process and Thread

Process	Thread
A Process is a program under execution. A Process contains many threads	A Thread is a single flow of execution and is a segment of a Program
Creation of new Process requires a new address space and resources	Threads can be created in the same address space. It saves memory space and OS resources
Processes have different code and data segments.	Threads share the code and data segments i.e., if one thread modifies a variable, all threads see the new value of that variable
Processes are heavy weight components	Threads are light weight components.
Communication between processes can be established using Inter Process Communication mechanism. Ex: Sockets and pipes	Communication between threads are very simple and efficient. Ex: Inter Thread Communication





Main Thread

- When a Java program starts up, one thread begins running immediately. This is usually called the main thread of our program, because it is the one that is executed when our program begins.
- The main thread is important for two reasons:
 - It is the thread from which other “child” threads will be spawned.
 - Often it must be the last thread to finish execution because it performs various shutdown actions.



Main Thread Program



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





- A reference to the current thread (the main thread, in this case) is obtained by calling `currentThread()`, and this reference is stored in the local variable `t`.
- `setName()` to change the internal name of the thread.
- The argument to `sleep()` specifies the delay period in milliseconds.
- Notice the try/catch block around this loop. The `sleep()` method in `Thread` might throw an `InterruptedException`.
- The name of the thread, its priority, and the name of its group.





Creating a Thread

- In the most general sense, we create a thread by instantiating an object of type Thread.
- Java defines two ways
 - We can implement the Runnable interface.
 - We can extend the Thread class.





Implementing Runnable

- The easiest way to create a thread is to create a class that implements the Runnable interface.
- 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.
- After we create a class that implements Runnable, we will instantiate an object of type Thread from within that class.





Implementing Runnable

- Thread defines several constructors.

`Thread(Runnable threadOb, String threadName)`

- In this constructor, threadOb is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName.
- After the new thread is created, it will not start running until you call its start() Method
- void start()



Runnable Interface Program

```
class NewThread implements Runnable
{
    Thread t;
    NewThread()
    {t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start();
    }
    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[])
    {
        new NewThread();
        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.");
    }
}
```



Extending Thread

- The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.
- The 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.



Extending Thread Class Program

```
class NewThread1 extends Thread
{
    NewThread1()
    {
        System.out.println("Child thread: " + this);
        start();
    }
    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[])
    {
        new NewThread1();
        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.");
    }
}
```





Creating Multiple Threads

- So far, we have been using only two threads: the main thread and one child thread.
- However, our program can spawn as many threads as it needs.



Creating Multiple Threads Program

```
class NewThread2 implements Runnable
{
    String name;
    Thread t;
    NewThread2(String threadname)
    {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start();
    }
    public void run()
    {
        try
        {
            for(int i = 5; i > 0; i--)
            {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        }
    }
}
```

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

class MultiThreadDemo
{
    public static void main(String args[])
    {
        new NewThread2("One");
        new NewThread2("Two");
        new NewThread2("Three");

        System.out.println("Main thread exiting.");
    }
}
```





Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- A higher-priority thread can also preempt a lower-priority one.
- To set a thread's priority, use the `setPriority()` method, which is a member of `Thread`.
- `final void setPriority(int level)`

Here, `level` specifies the new priority setting for the calling thread.



Thread Priorities

- 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.
- We can obtain the current priority setting by calling the `getPriority()` method of Thread, shown here:

```
final int getPriority()
```



Thread Priorities Program


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



Thread Priorities Program



```
class ThreadPriority
{
    public static void main(String []args)
    {
        A a=new A();
        B b=new B();
        C c=new C();

        c.setPriority(Thread.MAX_PRIORITY);
        b.setPriority(Thread.NORM_PRIORITY);
        //b.setPriority(c.getPriority()-2));
        a.setPriority(Thread.MIN_PRIORITY);
        //a.setPriority(b.getPriority()+1));
        System.out.println("Start Thread C ");
        c.start();
        System.out.println("Start Thread B ");
        b.start();
        System.out.println("Start Thread A ");
        a.start();
        System.out.println("End of Main Thread ");

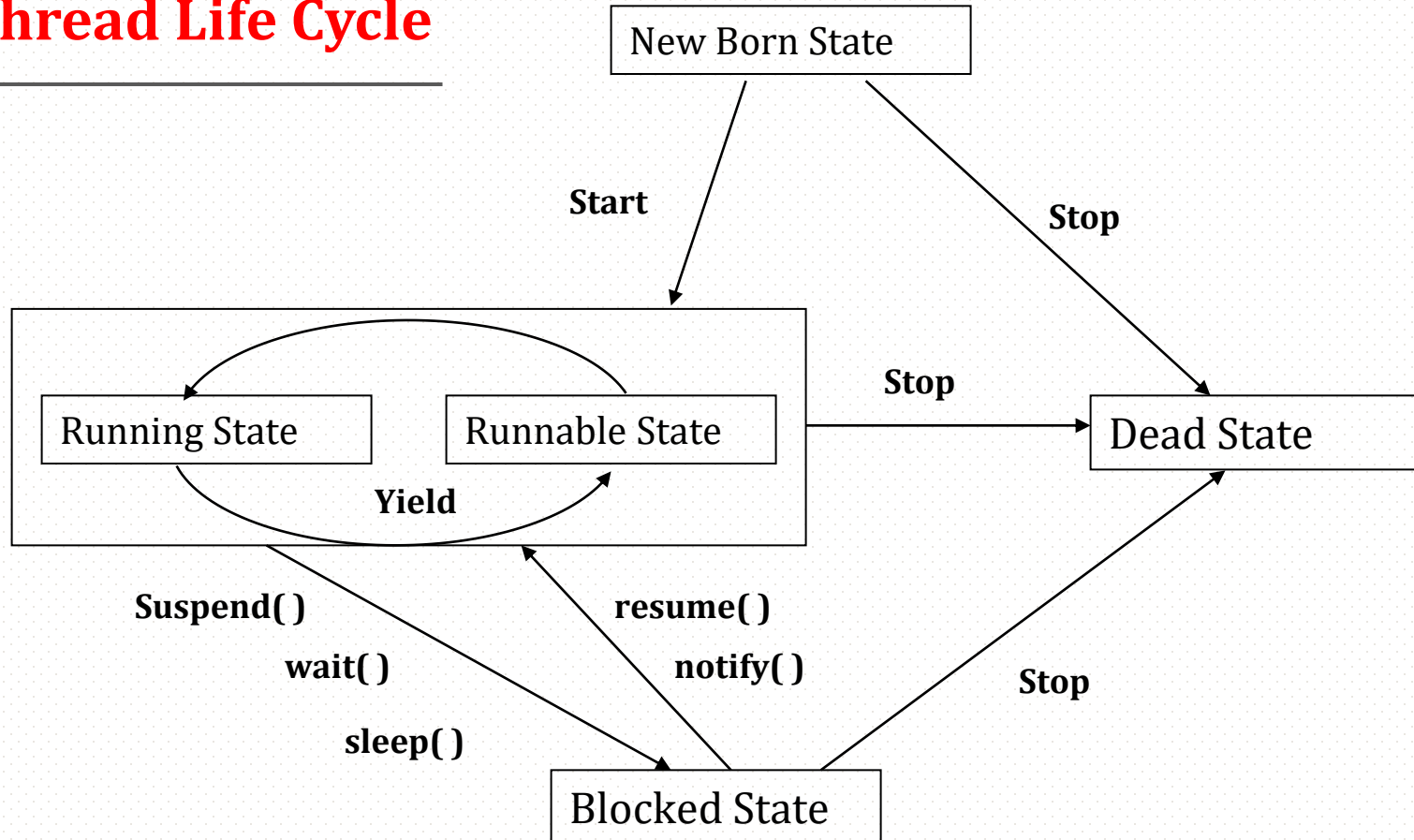
    }
}
```



Thread Life Cycle

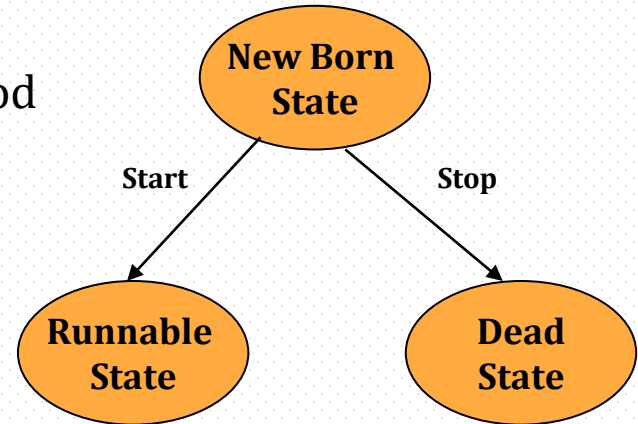


Thread Life Cycle



New born State

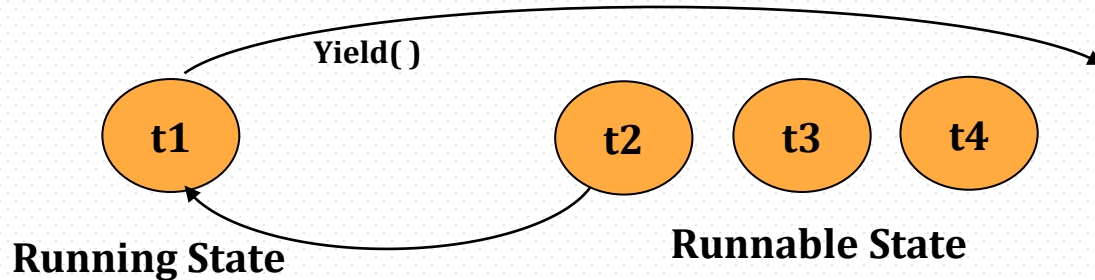
- When we create a thread object then the thread is in New born state.
- It has two alternatives
 - Schedule it for running using `start()` method
 - Kill it using `stop()` method





Runnable State

- It means thread is ready for execution and is waiting for the availability of the processor. If all threads have equal priority then they are given time slots for execution in Round Robin fashion (FIFO)
- We can move the threads from running state to runnable state by giving `yield()` command.





Running State

- It means the processor has given its time to the thread for its execution. The thread runs until it relinquishes the control on its own or it is preempted by a higher priority thread

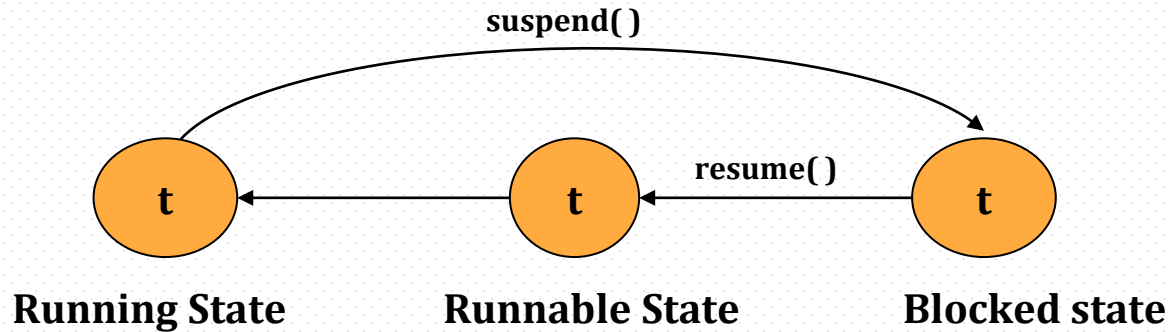
Blocked State

- A thread is said to be blocked when it is prevented from entering into runnable state and subsequently running state. It is also called “Not runnable state”



Suspend()

- A thread can be suspended using suspend() method

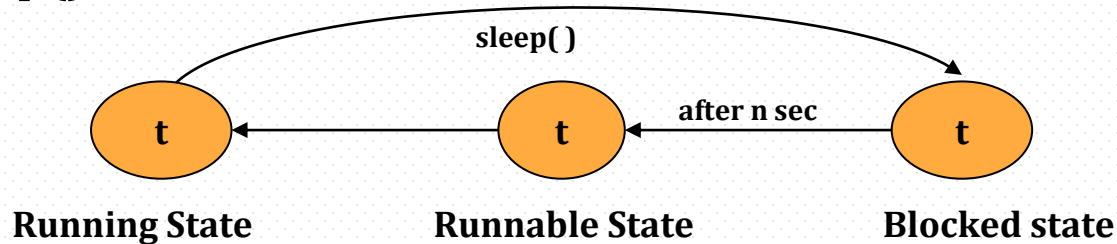


- A suspended thread can be received by using resume() method



Sleep()

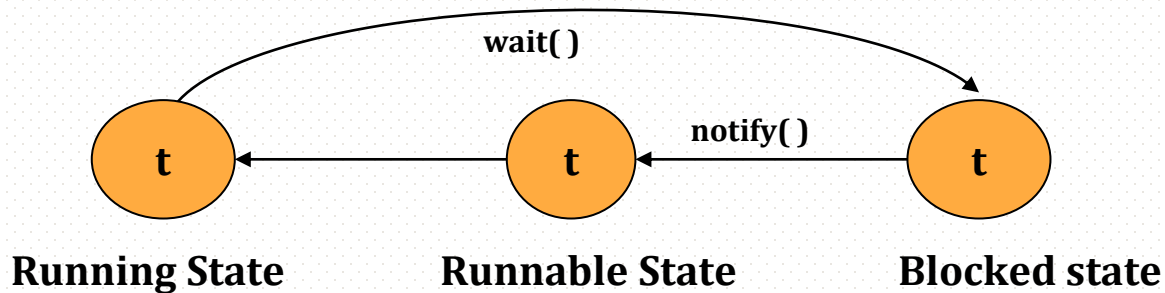
- We can put a thread into sleep mode for a specified time period using sleep() method



wait()

- The thread has to wait until some event occurs. This is done using a wait() method. The thread can be scheduled to run again using notify() command





Dead State

- A running thread ends its life when it is completed its execution is called “ Natural Death” otherwise we can kill using a stop() message then it is called “Preemptive death”



Thread Life Cycle Program

```
class A extends Thread
{
    public void run()
    {
        for (int i=1;i<=5; i++)
        {
            if(i==1) yield();
            System.out.println("\tFrom 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("\tFrom Thread B : j="+j);
            if(j==3) stop();
        }
        System.out.println("Exit From B ");
    }
}
```

```
class C extends Thread
{
    public void run()
    {
        for (int k=1;k<=5; k++) {
            System.out.println("\tFrom Thread C : K="+k);
            if(k==1) try { sleep(1000);}
                       catch(Exception e) {}
        } System.out.println("Exit From C ");
    }
}

class ThreadMethods
{
    public static void main(String []args)
    {
        System.out.println("Start Thread A "); new A().start();
        System.out.println("Start Thread B "); new B().start();
        System.out.println("Start Thread C "); new C().start();
        System.out.println("End of Main Thread ");
    }
}
```



Synchronization



Synchronization

- One thread may try to read a record from a file while another is still writing to the same file. This time we may get strange results.
- Java enables us to overcome this problem using a technique known as 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.





Synchronization

- 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.



Synchronized Method

- When we declare a method synchronized, Java creates a “monitor” and hands it over to the thread that calls the method first time.
- As long as the thread holds the monitor, no other thread can enter the synchronized section of code.
- A monitor is like a key and the thread that holds the key can only open the lock.
- This is the general form of the synchronized statement:

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



Synchronization Program

```
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();
    }
}
```

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

class Synch1
{
    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");
    }
}
```





Suspend - Resume

- The **suspend()** method of thread class puts the thread from running to waiting state.
- This method is used if you want to stop the thread execution and start it again when a certain event occurs.
- This method allows a thread to temporarily cease execution. The suspended thread can be resumed using the resume() method.



Suspend – Resume Program

```
class NewThread implements Runnable
{
    String name; Thread t;
    NewThread(String threadname)
    {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start();
    }
    public void run()
    {
        try
        {
            for(int i = 15; i > 0; i--)
            {
                System.out.println(name + ": " + i);
                Thread.sleep(200);
            }
        }
    }
}
```

```
        catch (InterruptedException e)
        { System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }
}
class SuspendResume
{
    public static void main(String args[])
    {NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");
    try
    {
        Thread.sleep(1000);
        ob1.t.suspend();
        System.out.println("Suspending thread One");
        Thread.sleep(1000);
        ob1.t.resume();
    }
}
```



Suspend – Resume Program

```
System.out.println("Resuming thread One");
    ob2.t.suspend();

System.out.println("Suspending thread Two");
    Thread.sleep(1000);
    ob2.t.resume();

System.out.println("Resuming thread Two");
    }
    catch (InterruptedException e)
    {
System.out.println("Main thread Interrupted");
    }
}
```

```
try {
System.out.println("Waiting for threads to finish.");
    ob1.t.join();
    ob2.t.join();
}
catch (InterruptedException e)
{
System.out.println("Main thread Interrupted");
}

System.out.println("Main thread exiting.");
}
}
```



Inter Thread Communication



Interthread Communication

- One thread is producing some data and another is consuming it.
- The producer has to wait until the consumer is finished before it generates more data.
- Java includes an elegant interprocess communication mechanism via the `wait()`, `notify()`, and `notifyAll()` methods. These methods are implemented as final methods in `Object`.
- All three methods can be called only from within a synchronized context.





Interthread Communication

- `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()`.

`final void wait()` throws `InterruptedException`

- `notify()` wakes up the first thread that called `wait()` on the same object.

`final void notify()`

- `notifyAll()` wakes up all the threads that called `wait()` on the same object. The highest priority thread will run first.

`final void notifyAll()`



Producer Consumer Program

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

    synchronized int get()
    {
        if(!valueSet)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
                System.out.println("Exception caught");
            }
        }

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

```
synchronized void put(int n)
{
    if(valueSet)
    {
        try
        {
            wait();
        }
        catch(InterruptedException e)
        {
            System.out.println("InterruptedException
caught");
        }
    }

    this.n = n;
    valueSet = true;
    System.out.println("Put: " + n);
    notify();
}
}
```

Producer Consumer Program

```
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++);
        }
    }
}

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 PC1
{
    public static void main(String args[])
    {
        Q q = new Q();

        new Producer(q);
        new Consumer(q);

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





Daemon Threads

- Java treats threads as User threads or Daemon threads
- User threads are the default. When main thread terminates, the JVM checks to see if any other user thread is running.
- If so, JVM does not terminate the application. On the other hand if JVM detects only Daemon threads the application terminates





Daemon Threads

- Daemon threads are designed as low level background threads that perform useful work. One example of Daemon thread is the “Garbage Collector thread”
- To create a Daemon thread call `setDaemon` method with a boolean true argument value i.e., `setDaemon(true)`
- To determine whether a thread object is associated with a Daemon thread call `isDaemon()`. It returns boolean value T/F





Using `isAlive()` and `join`

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





Using `isAlive()` and `join`

- While **`isAlive()`** is occasionally useful, the method that we will more commonly use to wait for a thread to finish is called **`join()`**, shown here:
 - final void `join()` throws `InterruptedException`
- This method waits until the thread on which it is called terminates.



isAlive() and join() Program

```
class NewThread4 implements Runnable
{
    String name;          Thread t;
    NewThread4(String threadname)
    {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start();
    }
    public void run()
    {
        try
        {
            for(int i = 5; i > 0; i--)
            { System.out.println(name + ": " + i);
              Thread.sleep(1000);    }
        }
        catch (InterruptedException e)
        { System.out.println(name + " interrupted."); }
        System.out.println(name + " exiting.");
    }
}
```


isAlive() and join() Program

```
class isalivejoin
{
    public static void main(String args[])
    {
        NewThread4 ob1 = new NewThread4("One");
        NewThread4 ob2 = new NewThread4("Two");
        NewThread4 ob3 = new NewThread4("Three");
        System.out.println("Thread One is alive: "+ ob1.t.isAlive());
        System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
        System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
        try
        {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();ob2.t.join();ob3.t.join();
        }
        catch (InterruptedException e)
        {
            System.out.println("Main thread Interrupted"); }
        System.out.println("Thread One is alive: "+ ob1.t.isAlive());
        System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
        System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
        System.out.println("Main thread exiting.");
    }
}
```

End of Unit IV

