

Unit 4

Multithreading

Difference between Multiprocessing and Multithreading

There are two distinct types of multitasking: **Process-based and Thread-based**. It is important to understand the difference between two. The Program in execution is defined as **Process**. Thus, the process based multi tasking is the feature that allows your computer to run two or more programs concurrently. For example we are able to use the java compiler and text editor at the same time. Another example is, we are able to hear the music and also able to get the print outs from the printer.

In the thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that the single program can contain two or more parts, each part of the program is called, **Thread**. For example the text editor can be formatting the text and also printing the text. Although the Java programs make use of the process-based multi tasking environments, but the process-based multi tasking is not under the control of java, Where as the thread-based multitasking is under the control of Java.

Process-Based Multitasking	Thread-Based Multitasking
This deals with "Big Picture"	This deals with Details
These are Heavyweight tasks	These are Lightweight tasks
Inter-process communication is expensive and limited	Inter-Thread communication is inexpensive.
Context switching from one process to another is costly in terms of memory	Context switching is low cost in terms of memory, because they run on the same address space
This is not under the control of Java	This is controlled by Java

Advantage of the Multithreading

- It enables you to write very efficient programs that maximizes the CPU utilization and reduces the idle time.
- Most I/O devices such as network ports, disk drives or the keyboard are much slower than CPU
- A program will spend much of its time just send and receive the information to or from the devices, which in turn wastes the CPU valuable time.
- By using the multithreading, your program can perform another task during this idle time.
- For example, while one part of the program is sending a file over the internet, another part can read the input from the keyboard, while other part can buffer the next block to send.
- It is possible to run two or more threads in multiprocessor or multi core systems simultaneously.

The States of the Thread

A thread can be in one of the several states. In general terms, a thread can **running**. It can be **ready** to run as soon as it gets the CPU time. A running thread can be **suspended**, which is a temporary halt to its execution. It can later be **resumed**. A thread can be **blocked** when waiting for the resource. A thread can be **terminated**.

Single Threaded Program

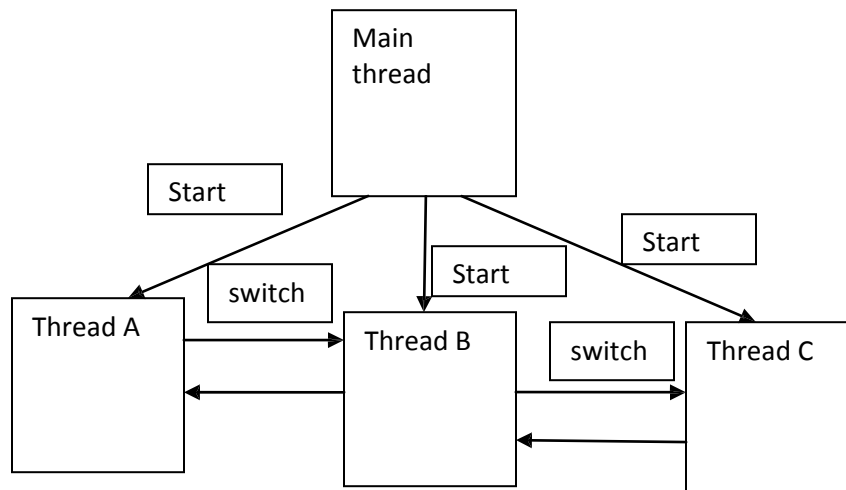
A Thread is similar to simple program that contains single flow of control. It has beginning, body, and ending. The statements in the body are executed in sequence.

For example:

```
class ABC          //Beginning
{
    -----
    ----- //Body
    -----
    -----
}                  //ending
```

Multithreaded Program

A unique property of the java is that it supports the multithreading. Java enables us the multiple flows of control in developing the program. Each separate flow of control is thought as tiny program known as "thread" that runs in parallel with other threads. In the following example when the main thread is executing, it may call thread A, as the Thread A is in execution again a call is mad for Thread B. Now the processor is switched from Thread A to Thread B. After the task is finished the flow of control comes back to the Thread A. The ability of the language that supports multiple threads is called "**Concurrency**". Since threads in the java are small sub programs of the main program and share the same address space, they are called "**light weight processes**".



The Main Thread

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

- It is the thread from which other "child" threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

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( )
```

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread. Let's begin by reviewing the following example:

CurrentThreadDemo.java

```
// 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");  
    }  
}  
}
```

Output

```
C:\Windows\system32\cmd.exe
F:\cse>javac CurrentThreadDemo.java
F:\cse>java CurrentThreadDemo
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
F:\cse>_
```

Creation of Thread

Creating the threads in the Java is simple. The threads can be implemented in the form of object that contains a method "run()". The "run()" method is the heart and soul of any thread. It makes up the entire body of the thread and is the only method in which the thread behavior can be implemented. There are two ways to create thread.

1. Declare a class that implements the **Runnable** interface which contains the run() method .
2. Declare a class that **extends** the **Thread** class and override the run() method.

1. Implementing the Runnable Interface

The Runnable interface contains the run() method that is required for implementing the threads in our program. To do this we must perform the following steps:

- I. Declare a class as implementing the **Runnable** interface
- II. Implement the **run()** method
- III. Create a **Thread** by defining an object that is instantiated from this "runnable" class as the target of the thread
- IV. Call the thread's **start()** method to run the thread.

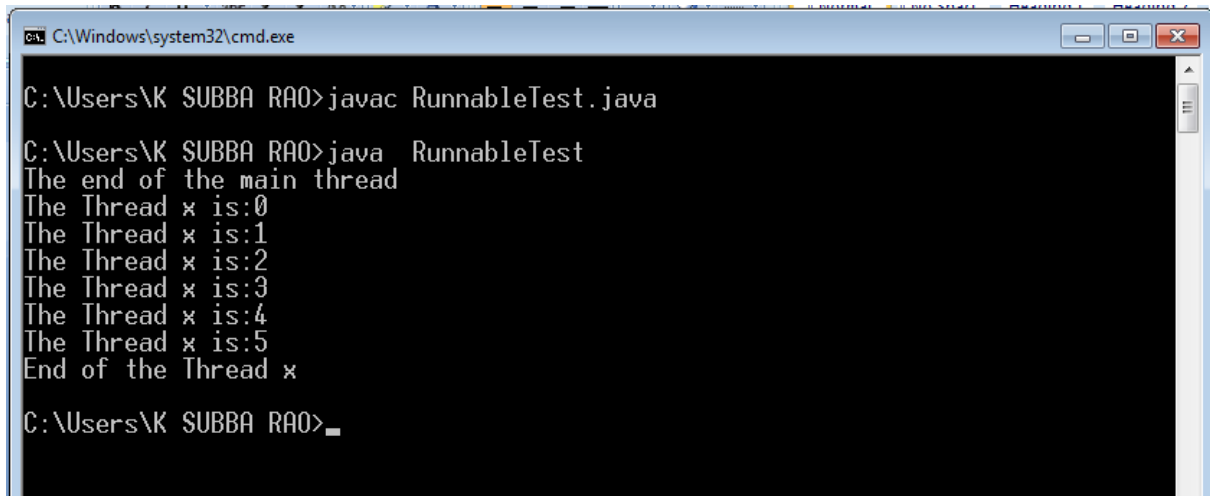
Example program:

Runnable.java

```
class x implements Runnable
{ //1 STEP
    public void run()
    { //2 STEP
        for(int i=0;i<=5;i++)
            System.out.println("The Thread x is:"+i);
            System.out.println("End of the Thread x");
    }
}
class RunnableTest
{
    public static void main(String args[])
    {
        x r=new x();
        Thread threadx=new Thread(r);
        threadx.start();
        System.out.println("The end of the main thread");
    }
}
```

```
}  
}
```

Output:



2. Extending the thread class

We can make our thread by extending the Thread class of java.lang.Thread class. This gives us access to all the methods of the Thread. It includes the following steps:

- I. Declare the class as Extending the Thread class.
- II. Override the "**run()**" method that is responsible for running the thread.
- III. Create a thread and call the "**start()**" method to instantiate the Thread Execution.

Declaring the class

TheThread class can be declared as follows:

```
class MyThread extends Thread  
{  
    -----  
    -----  
    -----  
    -----  
}
```

Overriding the method run()

The run() is the method of the Thread. We can override this as follows:

```
public void run()  
{  
    -----  
    -----  
    -----  
}
```

Starting the new Thread

To actually to create and run an instance of the thread class, we must write the following:

```
MyThread a=new MyThread(); // creating the Thread  
a.start(); // Starting the Thread
```

Example program:

ThreadTest.java

```
import java.io.*;
import java.lang.*;

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

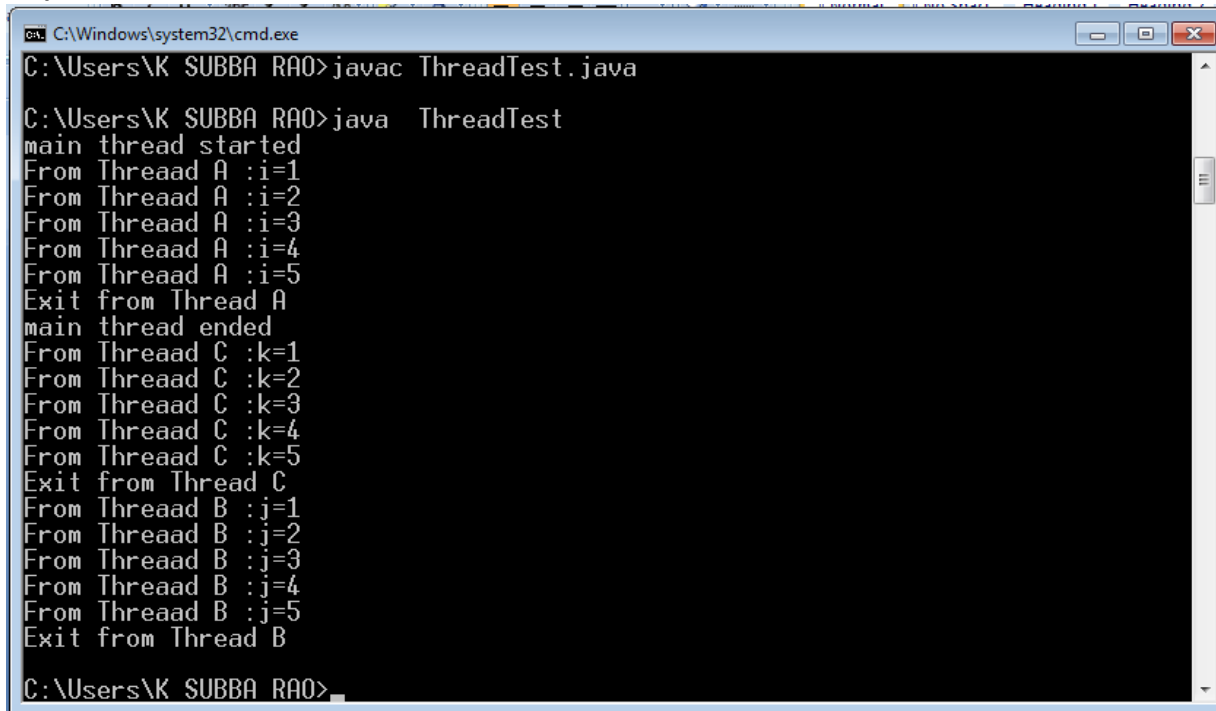
class B extends Thread
{
    public void run()
    {
        for(int j=1;j<=5;j++)
        {
            System.out.println("From Threaad B :j="+j);
        }
        System.out.println("Exit from Thread B");
    }
}

class C extends Thread
{
    public void run()
    {
        for(int k=1;k<=5;k++)
        {
            System.out.println("From Threaad C :k="+k);
        }
        System.out.println("Exit from Thread C");
    }
}

class ThreadTest
{
    public static void main(String args[])
    {
        System.out.println("main thread started");
        A a=new A();
        a.start();
        B b=new B();
    }
}
```

```
        b.start();
        C c=new C();
        c.start();
    System.out.println("main thread ended");
}
}
```

output: First Run



```
C:\Windows\system32\cmd.exe
C:\Users\K SUBBA RAO>javac ThreadTest.java
C:\Users\K SUBBA RAO>java ThreadTest
main thread started
From Threaad A :i=1
From Threaad A :i=2
From Threaad A :i=3
From Threaad A :i=4
From Threaad A :i=5
Exit from Thread A
main thread ended
From Threaad C :k=1
From Threaad C :k=2
From Threaad C :k=3
From Threaad C :k=4
From Threaad C :k=5
Exit from Thread C
From Threaad B :j=1
From Threaad B :j=2
From Threaad B :j=3
From Threaad B :j=4
From Threaad B :j=5
Exit from Thread B
C:\Users\K SUBBA RAO>
```

Second Run: Produces different out put in the second run, because of the processor switching from one thread to other.

```
C:\Windows\system32\cmd.exe
Exit from Thread B

C:\Users\K SUBBA RAO>java ThreadTest
main thread started
From Threaad A :i=1
From Threaad A :i=2
From Threaad A :i=3
From Threaad A :i=4
From Threaad A :i=5
Exit from Thread A
From Threaad B :j=1
From Threaad B :j=2
From Threaad B :j=3
From Threaad B :j=4
From Threaad B :j=5
Exit from Thread B
main thread ended
From Threaad C :k=1
From Threaad C :k=2
From Threaad C :k=3
From Threaad C :k=4
From Threaad C :k=5
Exit from Thread C

C:\Users\K SUBBA RAO>
```

Creating Multiple Threads

So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three child 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.
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.");
} //end of run method
} //end of NewThread
class MultiThreadDemo
{
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);
    }
    catch (InterruptedException e)
    {
        System.out.println("Main thread Interrupted");
    }
    System.out.println("Main thread exiting.");
}
}

```

When a Thread is ended

It is often very important to know which thread is ended. This helps to prevent the main from terminating before the child Thread is terminating. To address this problem "Thread" class provides two methods: **1) Thread.isAlive()** **2) Thread.join()**.

The general form of the "**isAlive()**" method is as follows:

```
final boolean isAlive();
```

This method returns the either "**TRUE**" or "**FALSE**" . It returns "**TRUE**" if the thread is alive, 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()`, shown here:

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.

Example Program:

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable
{
    String name; // name of thread
    Thread t;
    NewThread(String threadname)
    {
        name = threadname;
        t = new Thread( name);
        System.out.println("New thread: " + t.getName());
        t.start(); // Start the thread
    }
// This is the entry point for thread.
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 DemoJoin
{
    public static void main(String args[])
```

```

{
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");
    NewThread ob3 = new NewThread("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());
    // wait for threads to finish
    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.");
}
}

```

The Thread Priorities

Thread priorities are used by the thread scheduler to decide when and which 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)
```

Here, *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 **static final** variables within **Thread**.

You can obtain the current priority setting by calling the **getPriority()** method of **Thread**, shown here:

final int getPriority()

Example Program:

//setting the priorities for thethread

```
class PThread1 extends Thread
{
    public void run()
    {
        System.out.println(" Child 1 is started");
    }
}
class PThread2 extends Thread
{
    public void run()
    {
        System.out.println(" Child 2 is started");
    }
}
class PThread3 extends Thread
{
    public void run()
    {
        System.out.println(" Child 3 is started");
    }
}

class PTest
{
    public static void main(String args[])
    {
//setting the priorities to the thread using the setPriority() method
        PThread1 pt1=new PThread1();
        pt1.setPriority(1); PThread2
        pt2=new PThread2();
        pt2.setPriority(9);
        PThread3 pt3=new PThread3();
        pt3.setPriority(6);
        pt1.start();
        pt2.start();
        pt3.start();
//getting the priority
    }
}
```

```

        System.out.println("The pt1 thread priority is :"+pt1.getPriority());
    }
}

```

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*.

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.

Let us try to understand the problem without synchronization. Here, in the following example to threads are accessing the same resource (object) to print the Table. The Table class contains one method, printTable(int), which actually prints the table. We are creating two Threads, Thread1 and Thread2, which are using the same instance of the Table Resource (object), to print the table. When one thread is using the resource, no other thread is allowed to access the same resource Table to print the table.

Example without the synchronization:

```

Class Table
{
    void printTable(int n)
    { //method not synchronized
        for(int i=1;i<=5;i++)
        {
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            }
            catch(InterruptedException ie)
            {
                System.out.println("The Exception is :"+ie);
            }
        }
    } //end of the printTable() method
}

class MyThread1 extends Thread
{
    Table t;
}

```

```

        MyThread1(Table t)
        {
            this.t=t;
        }
        public void run()
        {
            t.printTable(5);
        }
    } //end of the Thread1

class MyThread2 extends Thread
{
    Table t;
    MyThread2(Table t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(100);
    }
} //end of Thread2

class TestSynchronization1
{
    public static void main(String args[])
    {
        Table obj = new Table();//only one object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    }
}

```

The output for the above program will be as follow:

```

Output: 5
        100
        10
        200
        15
        300
        20
        400
        25
        500

```

In the above output, it can be observed that both the threads are simultaneously accessing the Table object to print the table. Thread1 prints one line and goes to sleep, 400 milliseconds, and Thread1 prints its task.

Using the Java synchronized method

If you declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource. When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

The general form of the synchronized method is:

```
synchronized type method_name(para_list)
{
    //body of the method
}
```

where synchronized is the keyword, method contains the type, and method_name represents the name of the method, and para_list indicate the list of the parameters.

Example using the synchronized method

```
Class Table
{
    synchronized void printTable(int n)
    { //method not synchronized
        for(int i=1;i<=5;i++)
        {
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            }
            catch(InterruptedException ie)
                System.out.println("The Exception is :"+ie);
        }
    } //end of the printTable() method
}

class MyThread1 extends Thread
{
    Table t;
    MyThread1(Table t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(5);
    }
} //end of the Thread1
```

```

class MyThread2 extends Thread
{
    Table t;
    MyThread2(Table t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(100);
    }
} //end of Thread2

class TestSynchronization1
{
    public static void main(String args[])
    {
        Table obj = new Table();//only one object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    }
}

```

```

Output: 5
        10
        15
        20
        25
        100
        200
        300
        400
        500

```

In the above output it can be observed that when Thread1 is accessing the Table object, Thread2 is not allowed to access it. Thread1 preempts the Thread2 from accessing the printTable() method.

<p>Note:</p> <ol style="list-style-type: none"> 1. This way of communications between the threads competing for same resource is called implicit communication. 2. This has one disadvantage due to polling. The polling wastes the CPU time. To save the CPU time, it is preferred to go to the inter-thread communication.
--

Inter-Thread Communication

If two or more Threads are communicating with each other, it is called "inter thread" communication. Using the synchronized method, two or more threads can communicate indirectly. Through, synchronized method, each thread always competes for the resource. This way of competing is called polling. The polling wastes the much of the CPU valuable time. The better solution to this problem is, just notify other threads for the resource, when the current thread has finished its task. This is **explicit communication** between the threads.

Java addresses this polling problem, using via **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 a thread that called **wait()** on the same object.
- **notifyAll()** wakes up all the threads that called **wait()** on the same object. One of the threads will be granted access.

These methods are declared within Object, as shown here:

```
final void wait() throws InterruptedException
```

```
final void notify()
```

```
final void notifyAll()
```

Additional forms of **wait()** exist that allow you to specify a period of time to wait.

Although **wait()** normally waits until **notify()** or **notifyAll()** is called, there is a possibility that in very rare cases the waiting thread could be awakened due to a *spurious wakeup*. In this case, a waiting thread resumes without **notify()** or **notifyAll()** having been called. (In essence, the thread resumes for no apparent reason.) Because of this remote possibility, Sun recommends that calls to **wait()** should take place within a loop that checks the condition on which the thread is waiting. The following example shows this technique.

Example program for producer and consumer problem

```
class Q
{
    int n;
    boolean valueSet = false;
    synchronized int get()
    {
        while(!valueSet)
        try {
            wait();
        }
        catch(InterruptedException e)
        {
```

```

        System.out.println("InterruptedException caught");
    }
    System.out.println("Got: " + n);
    valueSet = false;
    notify();
    return n;
} //end of the get() method
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 the put method
} //end of the class Q

```

```

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++);
        }
    }
} //end of Producer
class Consumer implements Runnable
{
    Q q;
    Consumer(Q q)
    {

```

```

this.q = q;
new Thread(this, "Consumer").start();
}
public void run()
{
    while(true)
    {
        q.get();
    }
}
} //end of Consumer

```

```

class PCFixed
{
public static void main(String args[])
{
    Q q = new Q();
    new Producer(q);
    new Consumer(q);
    System.out.println("Press Control-C to stop.");
}
}

```

Suspending, Blocking and Stopping Threads

Whenever we want stop a thread we can stop from running using "**stop()**" method of thread class. It's general form will be as follows:

```
Thread.stop();
```

This method causes a thread to move from **running** to **dead** state. A thread will also move to dead state automatically when it reaches the end of its method.

Blocking Thread

A thread can be temporarily suspended or blocked from entering into the runnable and running state by using the following methods:

```
sleep()           —blocked for specified time
```

```
suspend()       ---blocked until further orders
```

```
wait()           --blocked until certain condition occurs
```

These methods cause the thread to go into the blocked state. The thread will return to the runnable state when the specified time is elapsed in the case of **sleep()**, the **resume()** method is invoked in the case of **suspend()**, and the **notify()** method is called in the case of **wait()**.

Example program:

The following program demonstrates these methods:

```
// Using suspend() and resume().
```

```
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.
    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(); 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");
}
// wait for threads to finish
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.");
}
}

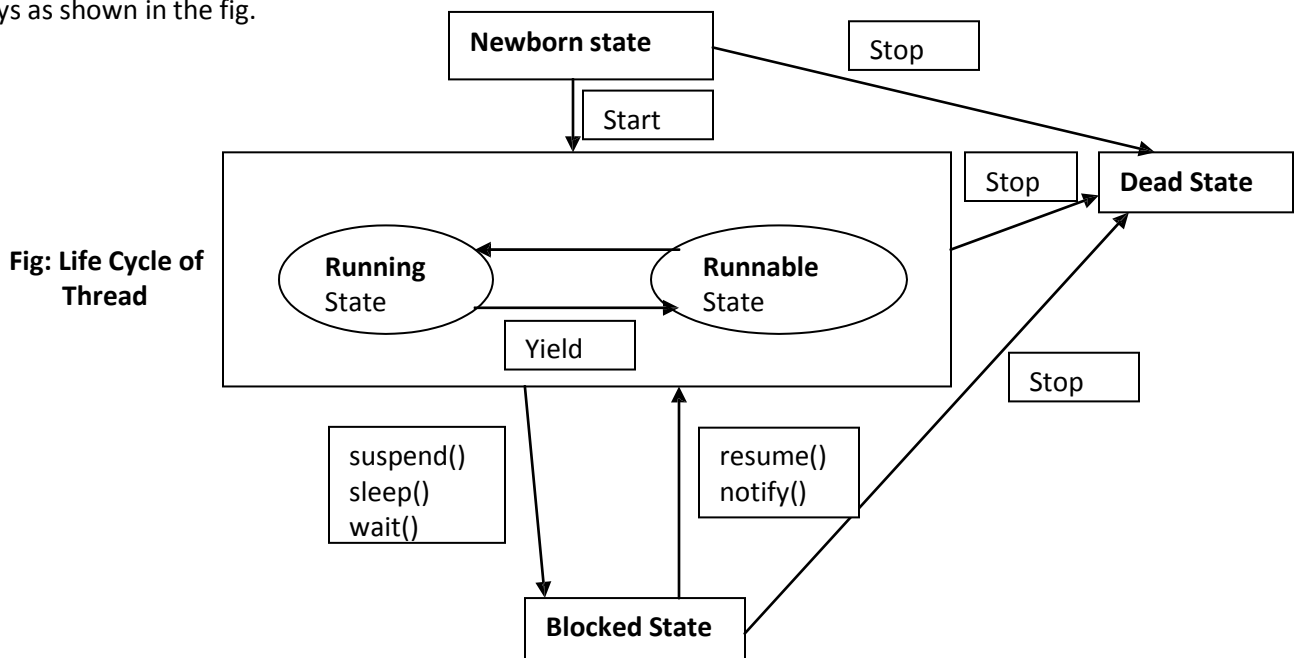
```

Life Cycle of a Thread

During the life time of the thread, there are many states it can enter. They include the following:

- Newborn state
- Runnable State
- Running State
- Blocked state
- Dead state

A thread can always in any one of the five states. It can move from one state to other via variety of ways as shown in the fig.



Newborn State: When we create a thread it is said to be in the new born state. At this state we can do the following:

- schedule it for running using the start() method.
- Kill it using stop() method.

Runnable State: A runnable state means that a thread is ready for execution and waiting for the availability of the processor. That is the thread has joined the queue of the threads for execution. If all the threads have equal priority, then they are given time slots for execution in the round robin fashion, first-come, first-serve manner. The thread that relinquishes the control will join the queue at the end and again waits for its turn. This is known as time slicing.

Running State:

Running state: Running state means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes the control or it is preempted by the other higher priority thread. As shown in the fig. a running thread can be preempted using the suspend(), or wait(), or sleep() methods.

Blocked state: A thread is said to be in the blocked state when it is prevented from entering into runnable state and subsequently the running state.

Dead state: Every thread has a life cycle. A running thread ends its life when it has completed execution. It is a natural death. However we also can kill the thread by sending the stop() message to it at any time.

The Thread Methods:

Sl No	Method Name	Description
1	run()	Used to write the body of the thread
2	start()	Used to start the thread
3	sleep()	Used to make the thread sleep for milliseconds
4	suspend()	Used to suspend a running thread
5	wait()	Waits until further ordered
6	yield	Used to give control to other thread before its turn comes
7	Stop()	Used to stop the thread
8	resume()	Used to start the blocked thread
9	notify()	Used to notify the waiting thread
10	notifyAll()	Used to notify all waiting threads

Thread Exceptions

Note that a call to the sleep() method is always enclosed in try/ catch block. This is necessary because the sleep() method throws an exception, which should be caught. If we fail to catch the exception the program will not compile.

its general form will be as follows:

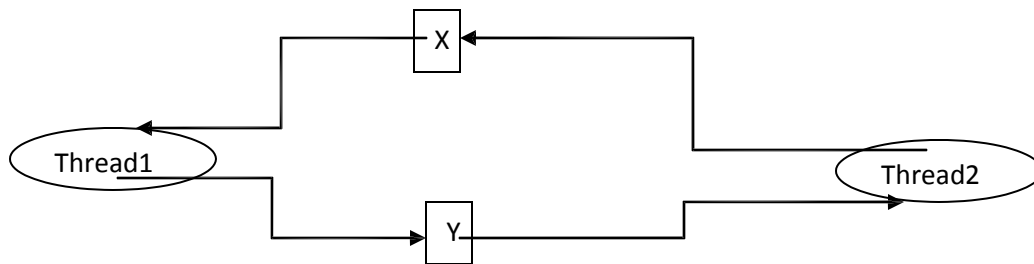
```
try
{
    Thread.sleep(1000);
}
```

```
catch(Exception e)
{
    -----
    -----
}

```

Deadlock

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called **deadlock**.



Here, in the above figure, the resource X is held by Thread1, and at the same time the Thread1 is trying to access the resource which is held by the Thread2. This is causing the circular dependency between two Threads. This is called, Deadlock.

Example program:

TestDead.java

```
public class TestDead
{
    public static void main(String[] args)
    {
        final String resource1 = "John Gardner";
        final String resource2 = "James Gosling";
        // t1 tries to lock resource1 then resource2

        Thread t1 = new Thread()
        {
            public void run()
            {
                synchronized (resource1)
                {
                    System.out.println("Thread 1: locked resource 1");

                    try {
                        Thread.sleep(100);
                    }
                    catch (Exception e)
                    {
                        System.out.println(e);
                    }
                }
            }
        }
    }
}

```

```

        }

        synchronized (resource2)
        {
            System.out.println("Thread 1: locked resource 2");
        }
    }
} //end of run()
}; //end of t1

// t2 tries to lock resource2 then resource1
Thread t2 = new Thread()
{
    public void run()
    {
        synchronized (resource2)
        {
            System.out.println("Thread 2: locked resource 2");

            try { Thread.sleep(100);} catch (Exception e) {}

            synchronized (resource1)
            {
                System.out.println("Thread 2: locked resource 1");
            }
        }
    }
} //end of run()
}; //end of t2

t1.start();
t2.start();
}
}

```

Output:


```
C:\Windows\system32\cmd.exe - java TestDead
F:\cse>javac TestDead.java
F:\cse>java TestDead
Thread 1: locked resource 1
Thread 2: locked resource 2
```

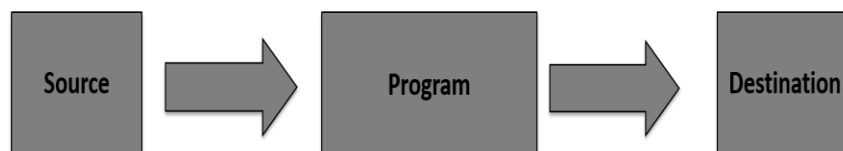
Input/Output: reading and writing data, java.io.package

There are two predefined packages in java that contain classes to perform I/O operations. These are **java.io.***, and **java.nio.***. The **java.io.*** used to perform reading and writing to console and reading and writing to the Files. The **java.nio.***, contains all the classes of java.io.*, and also contain the classes to perform advanced operations such as buffering, memory mapping, character encoding and decoding etc;. The java.io.* package provides separate classes for reading and writing data , these are **byte streams** and **character streams**.

Stream

A stream can be defined as a sequence or flow of data. There are two kinds of Streams.

- **InPutStream:** The InputStream is used to read data from a source.
- **OutPutStream:** the OutputStream is used for writing data to a destination.



Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are , **FileInputStream** and **FileOutputStream**.

Following is an example which makes use of these two classes to copy an input file into an output file:

CopyFile.java

```
import java.io.*;

public class CopyFile {

    public static void main(String args[]) throws IOException
    {

        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");
            int c;
            while ((c = in.read()) != -1)
            {
                out.write(c);
            }
        } //end of try
    finally {
        if (in != null)
        {
            in.close();
        }
        if (out != null)
        {
            out.close();
        }
    } //end of finally
    } //end of main
} //end of class
```

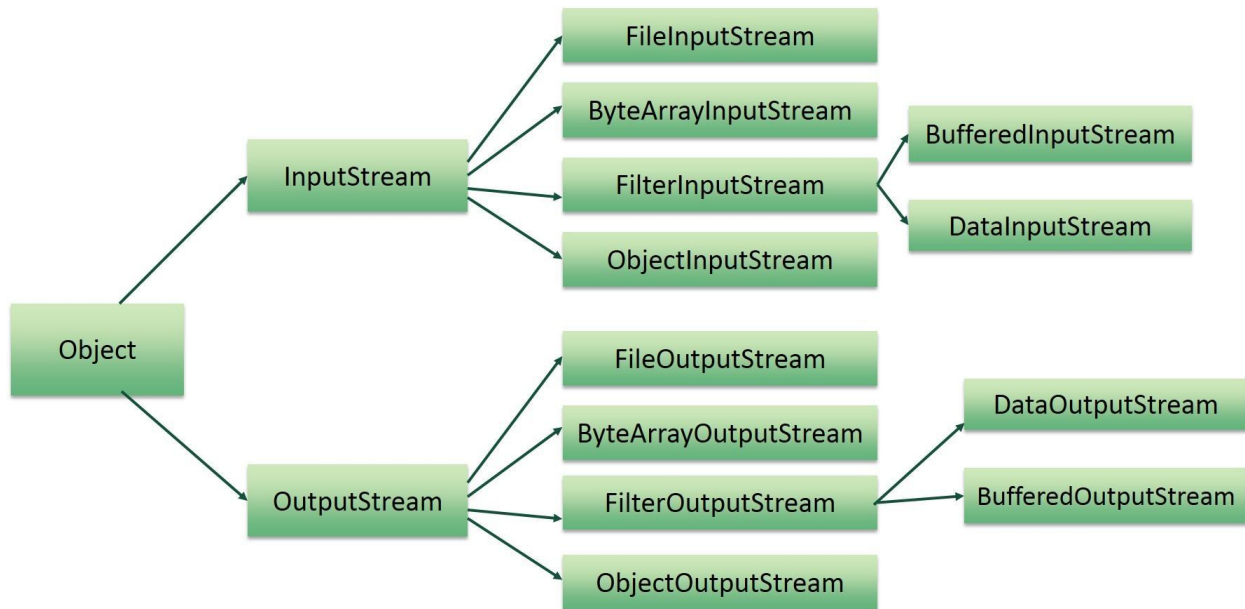
List of methods in **java.io.InputStream** class are as follow:

Method	Description
int available throws IOException	Returns the number of available bytes that can be read from input stream
void close() throws IOException	Closes the input stream
Void mark(int readlimit)	Makes the mark at the current position in the input stream. readlimit defines after how many lines this mark is nullified
Void markSupported()	Mark() method works if this method returns true
Abstract int read()	Used to read next byte from the input stream. It returns the byte, other wise -1 if EOF is encountered
Int read(byte []b)	Reads the byte and stores them in byte array b if return the true, otherwise -1 if EOF is encountered.
Int read(byte []b, int off, int len)	Reads the byte and stores them in byte array b upto the length from the offset off in b.
Void reset()	Resets the current pointer to the position set by the mark
Long skip(long n)	Skips the specified number of bytes from the input stream

List of methods in **java.io.OutputStream** class are as follow:

Method	Description
Void close()	Closes the output stream
Void flush()	Flushes the output stream
Void write(byte [] b)	Writes the contents of the byte array b to the output stream
Void write(byte [] b, int off, int len)	Writes the specified number of bytes (len) to the output stream starting from the offset off in b
Abstract void write(int b)	Abstract method to write to the output stream

Here is a hierarchy of classes to deal with Input and Output streams.



java.io.File class

The **Java.io.File** class is an abstract representation of file and directory pathnames. Following are the important points about File:

- Instances may or may not denote an actual file-system object such as a file or a directory. If it does denote such an object then that object resides in a partition. A partition is an operating system-specific portion of storage for a file system.
- A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as access permissions.
- Instances of the File class are immutable; that is, once created, the abstract pathname represented b

Methods of java.io.File class are as follow:

Method	Description
boolean canExecute()	This method tests whether the application can execute the file denoted by this abstract pathname.
boolean canRead()	This method tests whether the application can read the file denoted by this abstract pathname.
boolean canWrite()	This method tests whether the application can modify the file denoted by this abstract pathname.
boolean exists()	This method tests whether the file or directory denoted by this abstract pathname exists.
String getName()	This method returns the name of the file or directory denoted by this abstract pathname.
String getParent()	This method returns the pathname string of this

	abstract pathname's parent, or null if this pathname does not name a parent directory.
String getPath()	This method converts this abstract pathname into a pathname string.
String[] list()	This method returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
boolean isDirectory()	This method tests whether the file denoted by this abstract pathname is a directory.
boolean isFile()	This method tests whether the file denoted by this abstract pathname is a normal file.

Example program:

FileDemo.java

```
import java.io.File;
class FileDemo
{
public static void main(String args[])
{
    File f=new File(args[0]);
    System.out.println("The file is Executed:"+f.isFile());
    System.out.println("The file is Executed:"+f.canRead());
}
}
```

Output:

```

C:\Windows\system32\cmd.exe
E:\ksr>javac FileDemo.java
E:\ksr>java FileDemo FileDemo.java
The file is Executed:true
The file is Executed:true
E:\ksr>
```

Reading and Writing using the Byte Stream

FileInputStream and FileOutputStream classes are used to read and write respectively. The Constructor will be as follow:

FileInputStream fis=new FileInputStream("FileDemo.java");

The list of methods of **FileInputStream** are as follow:

SN	Methods with Description
1	<p>public void close() throws IOException{}</p> <p>This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.</p>
2	<p>protected void finalize()throws IOException {}</p> <p>This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.</p>
3	<p>public int read(int r)throws IOException{}</p> <p>This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's end of file.</p>
4	<p>public int read(byte[] r) throws IOException{}</p> <p>This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If end of file -1 will be returned.</p>
5	<p>public int available() throws IOException{}</p> <p>Gives the number of bytes that can be read from this file input stream. Returns an int.</p>

Methods of FileOutputStream are as follow:

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

The Constructor will be as follow:

FileOutputStream fos=new FileOutputStream("FileDemo1.java");

List of Methods of FileOutputStream are as Follow:

SN	Methods with Description
----	--------------------------

1	public void close() throws IOException{} This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException
2	protected void finalize()throws IOException {} This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	public void write(int w)throws IOException{} This methods writes the specified byte to the output stream.
4	public void write(byte[] w) Writes w.length bytes from the mentioned byte array to the OutputStream.

Example Program:

```
import java.io.*;

public class fileStreamTest{

    public static void main(String args[]){

        try{
            byte bWrite [] = {11,21,3,40,5};
            OutputStream os = new FileOutputStream("test.txt");
            for(int x=0; x < bWrite.length ; x++){
                os.write( bWrite[x] ); // writes the bytes
            }
            os.close();

            InputStream is = new FileInputStream("test.txt");
            int size = is.available();

            for(int i=0; i< size; i++){
                System.out.print((char)is.read() + " ");
            }
            is.close();
        }
    }
}
```

```
}catch(IOException e){
    System.out.print("Exception");
}
}
}
```

Reading and Writing Using Character Streams

Files can be read and written using character streams. The **FileReader** class used for reading contents of a file, and the **FileWriter** is used to write the contents to the file.

Example program: FileReadDemo.java

```
import java.io.*;
class FileReadDemo
{
    public static void main(String args[]) throws IOException
    {
        File f=new File(args[0]);
        if(f.exists())
        {
            FileReader fr=new FileReader(f);
            int n;
            while((n=fr.read())!=-1)
            {
                System.out.print((char)n);
            }
        }
        FileWriter fw=new FileWriter(args[1]);
        String s="Sample File Copying";
        fw.write(s);
        fw.close();
    }
}
```

Output:

The "Hell.java" contains String contained in the String variable s.


```
C:\Windows\system32\cmd.exe
import java.io.*;
class FileReadDemo
{
    public static void main(String args[]) throws IOException
    {
        File f=new File(args[0]);
        if(f.exists())
        {
            FileReader fr=new FileReader(f);
            int n;
            while((n=fr.read())!=-1)
            {
                System.out.print((char)n);
            }
            FileWriter fw=new FileWriter(args[1]);
            String s="Sample File Copying";
            fw.write(s);
            fw.close();
        }
    }
}
E:\ksr>edit Hell.java
E:\ksr>
```

Reading and Writing Using the Console (Scanner class)

The java.util package contains one particular class called Scanner class, which is used to read and write. A Snap shot of the Scanner class is as follows:

ScannerDemo.java

```
import java.util.*;
class ScannerDemo
{
    public static void main(String args[])
    {
        Scanner s=new Scanner(System.in);
        String name;
        int age;
        System.out.println("Enter your name:");
        name=s.nextLine();
        System.out.println("enter your age:");
        age=s.nextInt();
        System.out.println("Your name is:"+name);
        System.out.println("Your Age is:"+age);
    }
}
```