Multithreaded Programming

- Java provides built-in support for ***multithreaded 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.
- Multitasking is supported by virtually all modern operating systems.
- There are two distinct types of multitasking:
  - process-based
  - thread-based

- A ***process*** is, a program that is executing.
- Thus, ***process-based*** multitasking is the feature that allows your computer to run two or more programs concurrently.
- For example, we can run Java compiler while using 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.
- For instance, 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.
- Inter-process 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.
- Inter-thread communication is inexpensive, and context switching from one thread to the next is low cost.
- While Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java.
- However, thread-based multitasking can be controlled by Java.

Advantages:

- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.
- This is especially important for the interactive, networked environment in which Java operates.
- For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it.
- Even local file system resources are read and written at a much slower pace than they can be processed by the CPU.
- And, of course, user input is much slower than the computer.
- In a single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though the CPU is sitting idle most of the time.
- Multithreading lets you gain access to this idle time and put it to good use.

The Java Thread Model

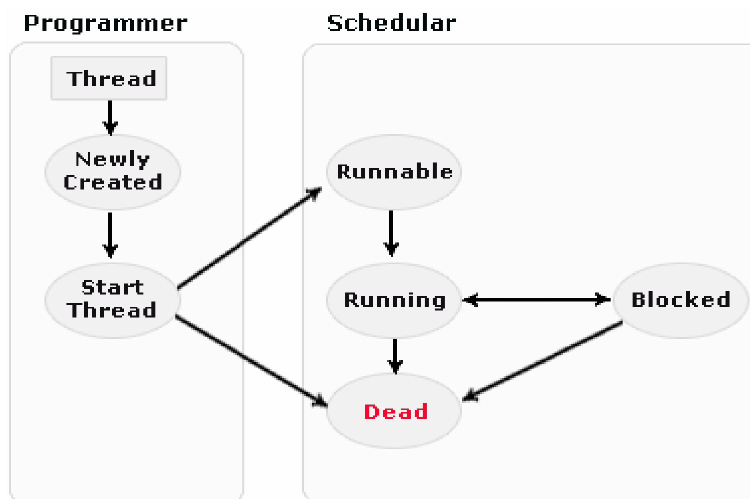
- In Java, all the class libraries are designed with multithreading in mind.
- In fact, Java uses threads to enable the entire environment to be asynchronous.
- First let us understand the drawbacks of single-threaded programming.

- Single-threaded systems use an approach called an ***event loop* with *polling.***
- In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.
- Once this polling mechanism returns, then the event loop dispatches control to the appropriate event handler.
- Until this event handler returns, nothing else can happen in the system.
- This wastes CPU time.
- It can also result in one part of a program dominating the system and preventing any other events from being processed.
- 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.

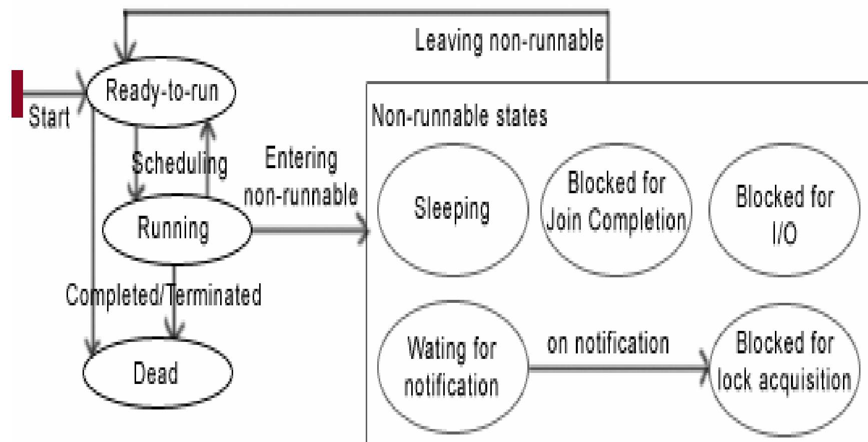
- The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated.
- One thread can pause without stopping other parts of your program.
- When a thread blocks in a Java program, only the single thread that is blocked pauses.
- All other threads continue to run.

- 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.

Different states of a single thread are :



Different states implementing Multiple-Threads are:



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.
- As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running.
- Instead, 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:

- The rules that determine when a context switch takes place are simple:

  – *A thread can voluntarily relinquish control.* This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

  – *A thread can be preempted by a higher-priority thread.* In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing— by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called ***preemptive multitasking****.*

- In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated.
- For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion.
- For other types of operating systems, threads of equal priority must voluntarily yield control to their peers.
- If they don't, the other threads will not run.

Synchronization

- Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it.
- For example, if we have two threads to communicate and share information, we need some way to ensure that they don't conflict with each other.
- That is, we must prevent one thread from writing data while another thread is in the middle of reading it.
- For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the ***monitor***.
- The monitor is a control mechanism.
- We can think of a monitor as a very small box that can hold only one thread.
- Once a thread enters a monitor, all other threads must wait until that thread exits the monitor.
- In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

- In Java, 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.
- This enables you to write very clear and concise multithreaded code, because synchronization support is built into the language.

Messaging

- After you divide your program into separate threads, you need to define how they will communicate with each other.
- In many programming languages, you must depend on the operating system to establish communication between threads.
- But, 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** class encapsulates a thread of execution.
- Since you can't directly refer to the state of a running thread, you will deal with it through its proxy, the **Thread** instance that generates it.
- To create a new thread, your program will either extend **Thread** class or implement the **Runnable** interface.

Method	Meaning
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:
  - It is the thread from which other “child” threads will be produced.
  - Often, it must be the last thread to finish execution because it performs various shutdown actions.

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

        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");
        }
    }
}
```

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


Creating a Thread

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

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( ).**
- In a class implementing Runnable, we need to create a **Thread** object.

```
class MyThread implements Runnable
{
    Thread t;
    MyThread()
    {
        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 MyThread();
        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.
```

- In a multithreaded program, often the main thread must be the last thread to finish running.
- In fact, for some older JVMs, if the main thread finishes before a child thread has completed, then the Java run-time system may “hang.”
- The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds.
- This causes the child thread to terminate earlier than the main thread.

Extending Thread

- 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 newthread.
- Here is the preceding program rewritten to extend **Thread**:

```
class MyThread extends Thread
{
    MyThread()
    {
        super( "Demo Thread");
        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 MyThread();
        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.
```

Choosing an Approach

- At this point, you might be wondering why Java has two ways to create child threads, and which approach is better.
- 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, the 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 you will not be overriding any of **Thread**'s other methods, it is probably best simply to implement **Runnable**.

Creating Multiple Threads

- In the previous example, we have seen one main thread and one child thread.
- We can even create many threads in a single program.

```
class MyThread implements Runnable
{
    String name;
    Thread t;

    MyThread(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 MyThread("One");
        new MyThread("Two");
        new MyThread("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.");
    }
}
```

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

Using ***isAlive()*** and ***join()***

- Usually, we want *main* thread to finish last.
- To achieve this, we will give *sleep()* on *main()* with some higher value.
- We can have an alternative by using ***isAlive*** and ***join()***.
- isAlive checks whether the thread is still running.
- **join()** method waits until the thread on which it is called terminates**.**


```
class MyThread implements Runnable
{
    String name;
    Thread t;

    MyThread(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 DemoJoin
{
    public static void main(String args[])
    {
        MyThread ob1 = new MyThread("One");
        MyThread ob2 = new MyThread("Two");
        MyThread ob3 = new MyThread("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.");
    }
}
```


Thread Priorities

- The thread scheduler uses thread priorities to decide when each thread should be allowed to run.
- Theoretically, we say that higher-priority threads will get more CPU time than lower-priority threads.
- But, practically, this is not the case always.
- The amount of CPU time allocated to a thread depends of several factors even other than priority.
- For example, how an operating system handles multitasking can affect the availability of CPU time.
- Similarly, the number of tasks (or processes) running concurrently, while running a Java program containing multiple threads, may affect the CPU time-sharing.

- Sometimes, higher priority thread may preempt lower priority thread.
- For example, when a lower priority thread is running, if a higher priority thread resumes from sleeping or waiting, it may preempt the lower priority thread.

- Theoretically, threads of equal priority should get equal access to the CPU.
- But, Java is designed to work in a wide range of environments.
- And, the way one environment implements multitasking may differ from that in the other environment.
- For proper multi-threaded programming, the threads having same priority should yield the control voluntarily once in a while to ensure that all threads have a chance to run under a non-preemptive operating system.
- Usually, this will happen as one or the other thread will be blocked for some I/O operation and in the meanwhile other threads will get CPU.
- But, the programmer can not rely on this nature of OS.
- Because some threads are CPU-intensive and dominate the CPU.
- For such threads, we need to yield the control occasionally to allow other threads to run.

- 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**.
- To obtain the current priority setting by calling the **getPriority( )** method of **Thread**, shown here:
  final int getPriority( )


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

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

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

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

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

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

       try
       {
          Thread.sleep(10000);
       } catch (InterruptedException e)
       {
         System.out.println("Main thread interrupted.");
       }

       lo.end();
       hi.end();
```



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

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

Output:

Low-priority Thread: 4033358155
High-priority Thread: 4044925651

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.
- Synchronization is achieved using 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.
- In two ways we can achieve synchronization in Java –
  - Using synchronized methods
  - Using synchronized statements

Using Synchronized Methods

- Synchronization is easy in Java, because 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
{
    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()
        {
                target.call(msg);
        }
}
```

```
class Synch
{
    public static void main(String args[])
    {
            Callme target = new Callme();
            Caller ob1 = new Caller(target, "Hello");
            Caller ob2 = new Caller(target, "Synchronized");
            Caller ob3 = new Caller(target, "World");

            try
            {
                    ob1.t.join();
                    ob2.t.join();
                    ob3.t.join();
            } catch(InterruptedException e)
            {
                    System.out.println("Interrupted");
            }
    }
}
```


- As you can see, by calling **sleep( )**, the **call( )** method allows execution to switch to another thread.
- This results in the mixed-up output of the three message strings.
- In this program, nothing exists to stop all three threads from calling the same method, at the same time.
- This is known as a ***race condition,*** because the three threads are racing each other to complete the method.
- In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur.
- This can cause a program to run right one time and wrong the next.
- To fix the preceding program, you must ***serialize*** access to **call( )**.
- That is, you must restrict its access to only one thread at a time.
- To do this, you simply need to precede **call( )**'s definition with the keyword **synchronized.**

- Thus, we need to use **synchronized** keyword to handle the race conditions.
- Once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance.
- However, non-synchronized methods on that instance will continue to be callable.

The synchronized Statement

- **synchronized** methods may not work in all cases.
- To understand why, consider the following.
- Imagine that you 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 you, but by a third party, and you do not have access to the source code.
- Thus, you can't add **synchronized** to the appropriate methods within the class.
- To synchronize the object of this class, we need to put the calls to its methods inside a **synchronized** block.

- This is the general form of the **synchronized** statement:

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

- Here, *object* is a reference to the object being synchronized.
- A synchronized block ensures that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object*'s monitor.
- The previous program can be modified using synchronized block by altering only ***run()*** method as –

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


Interthread Communication

- In the previous examples, we have seen how one thread blocks other threads unconditionally from asynchronous access to few methods.
- Though, such a usage of implicit monitors is powerful, in Java, we can achieve this also by using interthread communication.
- Multithreading replaces event loop programming (Sequential programming) by dividing your tasks into discrete, logical units.
- Threads also overcome the problems with polling.
- 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.

- For example, consider the queuing problem, where one thread is producing some data and another is consuming it.
- **Suppose that the producer has to wait until the consumer is finished before it generates more data.**
- In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce.
- Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on.
- Clearly, this situation is undesirable.

- 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**.
- 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.

- 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.


```
class Q
{   int n;
    synchronized int get()
    {       System.out.println("Got: " + n);
            return n;
    }
    synchronized void put(int n)
    {       this.n = n;
            System.out.println("Put: " + n);
    }
}
class Producer implements Runnable
{   Q q;
    Producer(Q q)
    {   this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run()
    {       int i = 0;
            for(int j=0;j<10;j++)
                    q.put(i++);
    }
}
```

```
class Consumer implements Runnable
{   Q q;
    Consumer(Q q)
    {   this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run()
    {       for(int j=0;j<10;j++)
                    q.get();
    }
}
class PC
{
   public static void main(String args[])
   {
      Q q = new Q();
      new Producer(q);
      new Consumer(q);
   }
}
```

- The output of previous program shows that the producer thread is not waiting for consumer thread to consume the data.
- So, there is a possibility that the consumer is receiving same data repeatedly.


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

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

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

```
class Producer implements Runnable
{   Q q;
    Producer(Q q)
    {   this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run()
    {       int i = 0;
            for(int j=0;j<10;j++)
                        q.put(i++);
    }
}
class Consumer implements Runnable
{   Q q;
    Consumer(Q q)
    {   this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run()
    {       for(int j=0;j<10;j++)
            q.get();
    }
}
```

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


Deadlock

- A special type of error that you 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.
- Deadlock is a difficult error to debug for two reasons:
  - In general, it occurs only rarely, when the two threads time-slice in just the right way.
  - It may involve more than two threads and two synchronized objects.

```
class A
{
    synchronized void foo(B b)
    {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered A.foo");
        try
        {
            Thread.sleep(1000);
        } catch(Exception e)
        {
            System.out.println("A Interrupted");
        }
        System.out.println(name + " trying to call B.last()");
        b.last();
    }

    synchronized void last()
    {
        System.out.println("Inside A.last");
    }
}
```



```
class B
{
    synchronized void bar(A a)
    {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered B.bar");
        try
        {
            Thread.sleep(1000);
        } catch(Exception e)
        {
            System.out.println("B Interrupted");
        }
        System.out.println(name + " trying to call A.last()");
        a.last();
    }

    synchronized void last()
    {
        System.out.println("Inside A.last");
    }
}
```

```
class Deadlock implements Runnable
{
    A a = new A();
    B b = new B();

    Deadlock()
    {
            Thread.currentThread().setName("MainThread");
            Thread t = new Thread(this, "RacingThread");
            t.start();
            a.foo(b); // get lock on a in this thread.
            System.out.println("Back in main thread");
    }

    public void run()
    {
            b.bar(a); // get lock on b in other thread.
            System.out.println("Back in other thread");
    }

    public static void main(String args[])
    {
            new Deadlock();
    }
}
```


Suspending, Resuming, and Stopping Threads

- Sometimes, suspending execution of a thread is useful.
- For example, a separate thread can be used to display the time of day.
- If the user doesn't want a clock, then its thread can be suspended.
- Suspending a thread and restarting the suspended thread is easy.
- The older versions of Java, two methods ***suspend()*** and ***resume()*** belonging to ***Thread*** class were used for this purpose.
- The general form of these methods are:
  - final void suspend( )
  - final void resume( )

```
class MyThread implements Runnable
{
    String name;
    Thread t;

    MyThread(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(500);
                        }
            } catch (InterruptedException e)
            {
                        System.out.println(name + " interrupted.");
            }
            System.out.println(name + " exiting.");
    }
}
```



```
class SuspendResume
{
    public static void main(String args[])
    {
      MyThread ob1 = new MyThread("One");
      MyThread ob2 = new MyThread("Two");

      try
      {
            Thread.sleep(1000);
            System.out.println("Suspending thread One");
            ob1.t.suspend();
            Thread.sleep(1000);
            System.out.println("Resuming thread One");
            ob1.t.resume();
            System.out.println("Suspending thread Two");
            ob2.t.suspend();
            Thread.sleep(1000);
            System.out.println("Resuming thread Two");
            ob2.t.resume();
      } 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.");
}
}
```

- The **Thread** class also defines a method called **stop( )** that stops a thread.

- Its signature is shown here:
  final void stop( )

- Once a thread has been stopped, it cannot be restarted using **resume( )**.

The Modern Way of Suspending, Resuming, and Stopping Threads

- Though the usage of ***suspend(), resume()*** and ***stop()*** seems to be perfect, they are deprecated in newer versions of Java.

- This was done because **suspend( )** can sometimes cause serious system failures.

- Assume that a thread has obtained locks on critical data structures.

- If that thread is suspended at that point, those locks are not relinquished (or given up).

- Other threads that may be waiting for those resources can be deadlocked.

- As ***resume()*** is used only with ***suspend()***, it also has been deprecated.

- The **stop( )** method of the **Thread** class was also deprecated because this method can sometimes cause serious system failures.

- Assume that a thread is writing to a critically important data structure and has completed only part of its changes.

- If that thread is stopped at that point, that data structure might be left in a corrupted state.

- Since, all these methods have been deprecated, the ***run()*** method should be designed such a way that it periodically checks to determine whether that thread should suspend, resume, or stop its own execution.

- This is accomplished by establishing a flag variable that indicates the execution state of the thread.


```
class MyThread implements Runnable
{
    String name;
    Thread t;
    boolean suspendFlag;

    MyThread(String threadname)
    {
       name = threadname;
       t = new Thread(this, name);
       System.out.println("New thread: " + t);
       suspendFlag = false;
       t.start();
    }
    void mysuspend()
    {
          suspendFlag = true;
    }
    synchronized void myresume()
    {
          suspendFlag = false;
          notify();
    }
```

```
public void run()
{
  try
  {
    for(int i = 5; i > 0; i--)
    {
      System.out.println(name + ": " + i);
      Thread.sleep(500);

      synchronized(this)
      {
          while(suspendFlag)
             wait();
      }
    }
  } catch (InterruptedException e)
  {
   System.out.println(name+ " interrupted.");
   }

   System.out.println(name + " exiting.");
}
```

```
class SuspendResume
{
    public static void main(String args[])
    {
      MyThread ob1 = new MyThread("One");
      MyThread ob2 = new MyThread("Two");

      try
      {
            Thread.sleep(1000);
            System.out.println("Suspending thread One");
            ob1.mysuspend();
            Thread.sleep(1000);
            System.out.println("Resuming thread One");
            ob1.myresume();
            System.out.println("Suspending thread Two");
            ob2.mysuspend();
            Thread.sleep(1000);
            System.out.println("Resuming thread Two");
            ob2.myresume();
      } 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.");
}
}
```


Using Multithreading

- The key to utilizing Java's multithreading features effectively is to think concurrently rather than serially.
- For example, when you 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.
- A word of caution is in order, however: If you create too many threads, you can actually degrade the performance of your program rather than enhance it.
- Remember, some overhead is associated with context switching.
- If you create too many threads, more CPU time will be spent changing contexts than executing your program!

Pre-requisites for java.io

- Java programs perform I/O through streams.
- A *stream* is a logical device that either produces or consumes information.
- A stream is linked to a physical device by the Java I/O system.
- All streams behave in the same manner, even if the actual physical devices to which they are linked differ.
- Thus, the same I/O classes and methods can be applied to any type of device.
- Java defines two types of streams: byte and character.
- *Byte streams* are used for reading or writing binary data.
- *Character streams* provide a convenient means for handling input and output of characters.

Reading Console Input

- In Java, console input is accomplished by reading from **System.in**.
- To obtain a character based stream that is attached to the console, wrap **System.in** in a **BufferedReader** object.
- **BufferedReader** supports a buffered input stream. Its most commonly used constructor is shown here:

  BufferedReader(Reader *inputReader*)
- Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created.
- To obtain an **InputStreamReader** object that is linked to **System.in**, use the following constructor:

  InputStreamReader(InputStream *inputStream*)
- Because **System.in** refers to an object of type **InputStream**, it can be used for *inputStream.*

- Putting it all together, the following line of code creates a **BufferedReader** that is connected to the keyboard:

```
BufferedReader br = new BufferedReader(new
                                InputStreamReader(System.in));
```

- After this statement executes, **br** is a character-based stream that is linked to the console through **System.in**.
- To read a character from a **BufferedReader** , we use ***read()*** method.
- Each time that **read( )** is called, it reads a character from the input stream and returns it as an integer value.
- It returns –1 when the end of the stream is encountered.


```
import java.io.*;

class BRRead
{
    public static void main(String args[]) throws IOException
    {
            char c;
            BufferedReader br = new
                                BufferedReader(new InputStreamReader(System.in));
            System.out.println("Enter characters, 'q' to quit.");

            do
            {
                        c = (char) br.read();
                        System.out.println(c);
            } while(c != 'q');
    }
}
```

Output:

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
This output
```

```java
import java.io.*;

class BRRead
{
   public static void main(String args[]) throws IOException
   {
        int x;
        BufferedReader br = new BufferedReader(new
                                InputStreamReader(System.in));

        System.out.println("Enter a number:");
        x=Integer.parseInt((br.readLine()).toString());

        x=x+5;
        System.out.println(x);
   }
}
```


Input/Output: Exploring java.io

- Most of the programs require external data as input and they produce some output.
- Data is retrieved from an *input* source.
- The results of a program are sent to an *output* destination.
- In Java, these sources or destinations may be network connection, memory buffer, or disk file etc.
- Although physically different, these devices are all handled by the same abstraction: the *stream* which is a logical device that either produce or consume information.
- A stream is linked to a physical device by the Java I/O system.

The Java I/O Classes

BufferedInputStream	FileWriter	PipedOutputStream
BufferedOutputStream	FilterInputStream	PipedReader
BufferedReader	FilterOutputStream	PipedWriter
BufferedWriter	FilterReader	PrintStream
ByteArrayInputStream	FilterWriter	PrintWriter
ByteArrayOutputStream	InputStream	PushbackInputStream
CharArrayReader	InputStreamReader	PushbackReader
CharArrayWriter	LineNumberReader	RandomAccessFile
Console	ObjectInputStream	Reader
DataInputStream	ObjectInputStream.GetField	SequenceInputStream
DataOutputStream	ObjectOutputStream	SerializablePermission
File	ObjectOutputStream.PutField	StreamTokenizer
FileDescriptor	ObjectStreamClass	StringReader
FileInputStream	ObjectStreamField	StringWriter
FileOutputStream	OutputStream	Writer
FilePermission	OutputStreamWriter	
FileReader	PipedInputStream	

Interfaces defined by java.io

Closeable	FileFilter	ObjectInputValidation
DataInput	FilenameFilter	ObjectOutput
DataOutput	Flushable	ObjectStreamConstants
Externalizable	ObjectInput	Serializable

File

- Although most of the classes defined by **java.io** operate on streams, the **File** class does not.
- It deals directly with files and the file system.
- That is, the **File** class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself.
- A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.
- A directory in Java is treated simply as a **File** with one additional property—a list of filenames that can be examined by the **list( )** method.

- The following constructors can be used to create **File** objects:
  - File(String *directoryPath*)
  - File(String *directoryPath*, String *filename*)
  - File(File *dirObj*, String *filename*)
  - File(URI *uriObj*)
- Here, *directoryPath* is the path name of the file,
  filename is the name of the file or subdirectory,
  dirObj is a **File** object that specifies a directory, and
  uriObj is a **URI (Uniform Resource Identifier)** object that describes a file.
- Example:

```
File f1 = new File("/");              //file object created with only directory path
File f2 = new File("/","autoexec.bat");  //with directory path and file name
File f3 = new File(f1,"autoexec.bat");  //same as first declaration
```

NOTE

- Java does the right thing with path separators between UNIX and Windows conventions.
- If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.
- Remember, if you are using the Windows convention of a backslash character (\), you will need to use its escape sequence (\\) within a string.

File Methods:

- **File** defines many methods that obtain the standard properties of a **File** object.
- For example,
  - **getName( )** returns the name of the file,
  - **getParent( )** returns the name of the parent directory etc.


```
import java.io.File;

class FileDemo
{
    static void disp(String s)
    {
            System.out.println(s);
    }
    public static void main(String args[])
    {
            File f1 = new File("/test.txt");

            disp("File Name: " + f1.getName());
            disp("Path: " + f1.getPath());
            disp("Abs Path: " + f1.getAbsolutePath());
            disp("Parent: " + f1.getParent());
            disp(f1.exists() ? "exists" : "does not exist");
            disp(f1.canWrite() ? "is writeable" : "is not writeable");
            disp(f1.canRead() ? "is readable" : "is not readable");
            disp("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
            disp(f1.isFile() ? "is normal file" : "might be a named pipe");
            disp(f1.isAbsolute() ? "is absolute" : "is not absolute");
            disp("File last modified: " + f1.lastModified());
            disp("File size: " + f1.length() + " Bytes");
    }
}
```

- **isFile( )** returns **true** if called on a file and **false** if called on a directory.
- Also, **isFile( )** returns **false** for some special files, such as device drivers and named pipes, so this method can be used to make sure the file will behave as a file.
- The **isAbsolute( )** method returns **true** if the file has an absolute path and **false** if its path is relative.
- **File** also includes two useful utility methods.
- The first is **renameTo( )**, shown here:

  boolean renameTo(File *newName*)
- Here, the filename specified by *newName* becomes the new name of the invoking **File** object.
- It will return **true** upon success and **false** if the file cannot be renamed.

- The second utility method is **delete( )**, which deletes the disk file represented by the path of the invoking **File** object.
- It is shown here:

  boolean delete( )
- You can also use **delete( )** to delete a directory if the directory is empty.
- **delete( )** returns **true** if it deletes the file and **false** if the file cannot be removed.

Few more *File* Methods

Method	Description
void deleteOnExit( )	Removes the file associated with the invoking object when the Java Virtual Machine terminates.
long getFreeSpace( )	Returns the number of free bytes of storage available on the partition associated with the invoking object.
long getTotalSpace( )	Returns the storage capacity of the partition associated with the invoking object.
boolean isHidden( )	Returns true if the invoking file is hidden. Returns false otherwise.
boolean setLastModified(long millisec)	Sets the time stamp on the invoking file to that specified by millisec, which is the number of milliseconds from January 1, 1970, Coordinated Universal Time (UTC).
boolean setReadOnly( )	Sets the invoking file to read-only.

Directories

- A directory is a **File** that contains a list of other files and directories.
- When you create a **File** object and it is a directory, the **isDirectory( )** method will return **true**.
- In this case, you can call **list( )** on that object to extract the list of other files and directories inside.
- It has two forms. The first is shown here:
  String[ ] list( )
- The list of files is returned in an array of **String** objects.

```
import java.io.File;

class DirList
{    public static void main(String args[])
     {
          String dirname = "/java";
          File f1 = new File(dirname);

          if (f1.isDirectory())
          {
               System.out.println("Directory of " + dirname);
               String s[] = f1.list();
               for (int i=0; i < s.length; i++)
               {
                    File f = new File(dirname + "/" + s[i]);

                    if (f.isDirectory())
                         System.out.println(s[i] + " is a directory");
                    else
                         System.out.println(s[i] + " is a file");
               }
          }
          else
               System.out.println(dirname + " is not a directory");
     }
}
```


Using FilenameFilter

- Some times, we may want the **list() methd** to return only the files matching the required extension or pattern.
- To do this, we must use a second form of **list( )**, shown here:
  String[ ] list(FilenameFilter *FFObj*)
- In this form, *FFObj* is an object of a class that implements the **FilenameFilter** interface.
- **FilenameFilter** defines only a single method, **accept( )**, which is called once for each file in a list.
- Its general form is given here:
  boolean accept(File *directory*, String *filename*)
- The **accept( )** method returns **true** for files in the directory specified by *directory* that should be included in the list (that is, those that match the *filename* argument), and returns **false** for those files that should be excluded.

```java
import java.io.*;

class OnlyExt implements FilenameFilter
{
    String ext;

    public OnlyExt(String ext)
    {
        this.ext = "." + ext;
    }

    public boolean accept(File dir, String name)
    {
        return name.endsWith(ext);
    }
}
```

```java
class DirListOnly
{
  public static void main(String args[])
  {
     String dirname = "/Chetana";

     File f1 = new File(dirname);
     FilenameFilter only = new
                    OnlyExt("doc");

     String s[] = f1.list(only);

     for (int i=0; i < s.length; i++)
          System.out.println(s[i]);

  }
}
```


The listFiles( ) Alternative

- There is a variation to the **list( )** method, called **listFiles( )**, which you might find useful.
- The signatures for **listFiles( )** are shown here:
  - File[ ] listFiles( ) — returns all files
  - File[ ] listFiles(FilenameFilter *FFObj*) — returns those files that satisfy the specified **FilenameFilter**.
  - File[ ] listFiles(FileFilter *FObj*) — returns those files with path names that satisfy the specified **FileFilter**
- **FileFilter** defines only a single method, **accept( )**, which is called once for each file in a list.
- Its general form is given here:
  boolean accept(File *path*)
- The **accept( )** method returns **true** for files that should be included in the list (that is, those that match the *path* argument), and **false** for those that should be excluded.

Creating Directories

- Another two useful **File** utility methods are **mkdir( )** and **mkdirs( )**.
- The **mkdir( )** method creates a directory, returning **true** on success and **false** on failure.
- Failure indicates that the path specified in the **File** object already exists, or that the directory cannot be created because the entire path does not exist yet.
- To create a directory for which no path exists, use the **mkdirs( )** method.
- It creates both a directory and all the parents of the directory.

The Closeable and Flushable Interfaces

- In JDK 5, two interfaces were added to **java.io**: **Closeable** and **Flushable**.
- They offer a uniform way of specifying that a stream can be closed or flushed.
- Objects of a class that implements **Closeable** can be closed.
- It defines the **close( )** method, shown here:

```
void close( ) throws IOException
```

- This method closes the invoking stream, releasing any resources that it may hold.
- This interface is implemented by all of the I/O classes that open a stream that can be closed.

- Objects of a class that implements **Flushable** can force buffered output to be written to the stream to which the object is attached.

- It defines the **flush( )** method, shown here:

```
void flush( ) throws IOException
```

- Flushing a stream typically causes buffered output to be physically written to the underlying device.

- This interface is implemented by all of the I/O classes that write to a stream.

The Stream Classes

- Java's stream-based I/O is built upon four abstract classes: **InputStream**, **OutputStream**, **Reader**, and **Writer**.

- They are used to create several concrete stream subclasses.

- Although your programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream classes.

- **InputStream** and **OutputStream** are designed for byte streams.

- **Reader** and **Writer** are designed for character streams.

- The byte stream classes and the character stream classes form separate hierarchies.

- In general, you should use the character stream classes when working with characters or strings, and use the byte stream classes when working with bytes or other binary objects.

The Byte Streams

- A byte stream can be used with any type of object, including binary data.
- Hence, byte streams are important to many types of programs.
- The byte stream classes are topped by **InputStream** and **OutputStream**.

InputStream:

- **InputStream** is an abstract class that defines Java's model of streaming byte input.
- It implements the **Closeable** interface.
- Most of the methods in this class will throw an **IOException** on error conditions.

OutputStream

- This also is an abstract class that defines streaming byte output.
- It implements the **Closeable** and **Flushable** interfaces and throws an **IOException** in the case of errors.

The Methods Defined by InputStream

Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Further read attempts will generate an IOException.
void mark(int numBytes)	Places a mark at the current point in the input stream that will remain valid until numBytes bytes are read.
boolean markSupported( )	Returns true if mark( )/reset( ) are supported by the invoking stream.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte buffer[ ])	Attempts to read up to buffer.length bytes into buffer and returns the actual number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte buffer[ ], int offset, int numBytes)	Attempts to read up to numBytes bytes into buffer starting at buffer[offset], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long numBytes)	Ignores (that is, skips) numBytes bytes of input, returning the number of bytes actually ignored.

The Methods Defined by OutputStream

Method	Description
void close( )	Closes the output stream. Further write attempts will generate an IOException.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int b)	Writes a single byte to an output stream. Note that the parameter is an int, which allows you to call write( ) with expressions without having to cast them back to byte.
void write(byte buffer[ ])	Writes a complete array of bytes to an output stream.
void write(byte buffer[ ], int offset, int numBytes)	Writes a subrange of numBytes bytes from the array buffer, beginning at buffer[offset].

FileInputStream :

- The **FileInputStream** class creates an **InputStream** that can be used to read bytes from a file.
- Its two most common constructors are shown here:
  - FileInputStream(String *filepath*)
  - FileInputStream(File *fileObj*)
- The following example creates two **FileInputStream**s that use the same file:

```
FileInputStream f1 = new  FileInputStream("/test.txt")
File f= new File("/test.txt");
FileInputStream f2 = new FileInputStream(f);
```

- When a **FileInputStream** is created, it is also opened for reading.
- **FileInputStream** overrides six of the methods in the abstract class **InputStream**.
- The **mark( )** and **reset( )** methods are not overridden, and any attempt to use **reset( )** on a **FileInputStream** will generate an **IOException**.

```
import java.io.*;

class FISDemo
{
        public static void main(String args[]) throws IOException
    {
            int size;
            InputStream f = new FileInputStream("FISDemo.java");

            size= f.available();
            System.out.println("Total Available Bytes: " + size);
            int n = size/40;

            System.out.println("First " + n + " bytes of the file");

            for (int i=0; i < n; i++)
                    System.out.print((char) f.read());

            System.out.println("\nStill Available: " + f.available());
            System.out.println("Reading the next " + n + "bytes using array");
```



```
            byte b[] = new byte[n];

            if (f.read(b) != n)
                    System.out.println("couldn't read " + n + " bytes.");

            System.out.println(new String(b, 0, n));
            System.out.println("\nStill Available: " + (size = f.available()));
            System.out.println("Skipping half of remaining bytes with skip()");

            f.skip(size/2);
            System.out.println("Still Available: " + f.available());
            System.out.println("Reading " + n/2 + " into the end of array");
            if (f.read(b, n/2, n/2) != n/2)
                    System.out.println("couldn't read " + n/2 + " bytes.");

            System.out.println(new String(b, 0, b.length));
            System.out.println("\nStill Available: " + f.available());
            f.close();
    }
}
```

OUTPUT:

```
Total Available Bytes: 1237
First 30 bytes of the file
import java.io.*;

class FIS
Still Available: 1207
Reading the next 30bytes using array
Demo
{
	public static

Still Available: 1177
Skipping half of remaining bytes with skip()
Still Available: 589
Reading 15 into the end of array
Demo
{
      " + n + " byte

Still Available: 574
```


FileOutputStream :

- **FileOutputStream** creates an **OutputStream** that you can use to write bytes to a file.
- Its most commonly used constructors are shown here:
  - FileOutputStream(String *filePath*)
  - FileOutputStream(File *fileObj*)
  - FileOutputStream(String *filePath*, boolean *append*)
  - FileOutputStream(File *fileObj*, boolean *append*)
- They can throw a **FileNotFoundException**.
- Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.
- If *append* is **true**, the file is opened in append mode.
- If the specified file is not existing, then it will be created and opened for writing when FileOutputStream object is created.
- If an attempt is made to open a read-only file, an **IOException** will be thrown.

```
import java.io.*;

class FOSDemo
{
    public static void main(String args[]) throws IOException
    {
        String source = "Now is the time for all good men to come to the aid of \n”
                        + “ their country and pay their due taxes.";

        byte buf[] = source.getBytes();
        OutputStream f1 = new FileOutputStream("file1.txt");

        for (int i=0; i < buf.length; i++)
            f1.write(buf[i]);
        f1.close();

        OutputStream f2 = new FileOutputStream("file2.txt");
        f2.write(buf);
        f2.close();
    }
}
```


- In the above program, we are creating a reference to *abstract* class *OutputStream* (f1 and f2) which are going to store objects of *FileOutputStream* class.
- After executing the above program, two files viz. *file1.txt* and *file2.txt* will be created in a current working directory.
- The contents of these files will be same as the string *source* given in the program.

The Character Streams

- While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters.
- Since one of the main purposes of Java is to support the “write once, run anywhere” philosophy, it was necessary to include direct I/O support for characters.
- As explained earlier, at the top of the character stream hierarchies are the **Reader** and **Writer** abstract classes.

Reader:

- **Reader** is an abstract class that defines Java’s model of streaming character input. It implements the **Closeable** and **Readable** interfaces. All of the methods in this class (except for **markSupported( )**) will throw an **IOException** on error conditions.

Writer:

- **Writer** is an abstract class that defines streaming character output. It implements the **Closeable**, **Flushable**, and **Appendable** interfaces. All of the methods in this class throw an **IOException** in the case of errors.

The Methods Defined by Reader

Method	Description
abstract void close( )	Closes the input source. Further read attempts will generate an IOException.
void mark(int numChars)	Places a mark at the current point in the input stream that will remain valid until numChars characters are read.
boolean markSupported( )	Returns true if mark( )/reset( ) are supported on this stream.
int read( )	Returns an integer representation of the next available character from the invoking input stream. –1 is returned when the end of the file is encountered.
int read(char buffer[ ])	Attempts to read up to buffer.length characters into buffer and returns the actual number of characters that were successfully read. –1 is returned when the end of the file is encountered.
abstract int read(char buffer[ ], int offset, int numChars)	Attempts to read up to numChars characters into buffer starting at buffer[offset], returning the number of characters successfully read. –1 is returned when the end of the file is encountered.
boolean ready( )	Returns true if the next input request will not wait. Otherwise, it returns false.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long numChars)	Skips over numChars characters of input, returning the number of characters actually skipped.

The Methods Defined by Writer

Method	Description
Writer append(char ch)	Appends ch to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence chars)	Appends chars to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence chars, int begin, int end)	Appends the subrange of chars specified by begin and end–1 to the end of the invoking ouput stream. Returns a reference to the invoking stream.
abstract void close( )	Closes the output stream. Further write attempts will generate an IOException.
abstract void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int ch)	Writes a single character to the invoking output stream. Note that the parameter is an int, which allows you to call write with expressions without having to cast them back to char.
void write(char buffer[ ])	Writes a complete array of characters to the invoking output stream.
abstract void write(char buffer[], int offset, int numChars)	Writes a subrange of numChars characters from the array buffer, beginning at buffer[offset] to the invoking output stream.
void write(String str)	Writes str to the invoking output stream.
void write(String str, int offset, int numChars)	Writes a subrange of numChars characters from the string str, beginning at the specified offset.

FileReader

- The **FileReader** class creates a **Reader** that you can use to read the contents of a file.
- Its two most commonly used constructors are shown here:
  - FileReader(String *filePath*)
  - FileReader(File *fileObj*)
- Either can throw a **FileNotFoundException**.
- Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.

```
import java.io.*;

class FRDemo
{
    public static void main(String args[]) throws IOException
    {
        FileReader fr = new FileReader("FRDemo.java");
        BufferedReader br = new BufferedReader(fr);
        String s;

        while((s = br.readLine()) != null)
            System.out.println(s);

        fr.close();
    }
}
```

- The above program reads each line in the file *FRDemo.java* (the same program file) and prints on to the monitor.
- Observe that *BufferedReader* class which is used for reading data from console (keyboard) itself is used for reading data from the file also.

FileWriter

- **FileWriter** creates a **Writer** that you can use to write to a file.
- Its most commonly used constructors are shown here:
  - FileWriter(String *filePath*)
  - FileWriter(String *filePath*, boolean *append*)
  - FileWriter(File *fileObj*)
  - FileWriter(File *fileObj*, boolean *append*)
- They can throw an **IOException**.
- Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.
- If *append* is **true**, then output is appended to the end of the file.
- If the specified file is not existing, then a new file will be opened for writing when a **FileWriter** object is created**.**
- In the case where you attempt to open a read-only file, an **IOException** will be thrown.

```
import java.io.*;
class FWDemo
{    public static void main(String args[]) throws IOException
     {
          String source = "Now is the time for all good men  to come to the aid of their \n"
                         + " country and pay their due taxes.";

          char buffer[] = new char[source.length()];
          source.getChars(0, source.length(), buffer, 0);   //copy content of source to buffer

          FileWriter f1 = new FileWriter("file1.txt");

          for (int i=0; i < buffer.length; i++)
               f1.write(buffer[i]);
          f1.close();

          FileWriter f2 = new FileWriter("file2.txt");
          f2.write(buffer);
          f2.close();
     }
}
```

- After executing the above program, two files viz. *file1.txt* and *file2.txt* will be created in a current working directory. The contents of these files will be same as the string *source* given in the program.

The Console Class

- Java SE 6 adds the **Console** class.
- It is used to read from and write to the console, if one exists.
- It implements the **Flushable** interface.
- Though **Console** class provides most of the functionalities as that of **System.in** and **System.out,** it is much useful in reading strings from the console.
- **Console** supplies no constructors.
- Instead, a **Console** object is obtained by calling **System.console( )**, which is shown here:

  static Console console( )
- If a console is available, then a reference to it is returned. Otherwise, **null** is returned.
- A console will not be available in all cases. Thus, if **null** is returned, no console I/O is possible.

- Whether a virtual machine has a console is dependent upon the underlying platform and also upon the manner in which the virtual machine is invoked.
- If the virtual machine is started from an interactive command line without redirecting the standard input and output streams then its console will exist and will typically be connected to the keyboard and display from which the virtual machine was launched.
- If the virtual machine is started automatically, for example by a background job scheduler, then it will typically not have a console.
- **Console** defines the methods shown in following Table.
- Notice that the input methods, such as **readLine( )**, throw **IOError** if an input error occurs.
- **IOError** is a new exception added by Java SE 6, and it is a subclass of **Error**.
- It indicates an I/O failure that is beyond the control of your program.
- Thus, you will not normally catch an **IOError**.

The Methods Defined by Console

Method	Description
void flush( )	Causes buffered output to be written physically to the console.
Console format(String fmtString, Object...args)	Writes args to the console using the format specified by fmtString.
Console printf(String fmtString, Object...args)	Writes args to the console using the format specified by fmtString.
Reader reader( )	Returns a reference to a Reader connected to the console.
String readLine( )	Reads and returns a string entered at the keyboard. Input stops when the user presses ENTER. If the end of the console input stream has been reached, null is returned. An **IOError** is thrown on failure.
String readLine(String fmtString, Object...args)	Displays a prompting string formatted as specified by fmtString and args, and then reads and returns a string entered at the keyboard. Input stops when the user presses ENTER. If the end of the console input stream has been reached, null is returned. An **IOError** is thrown on failure.
char[ ] readPassword( )	Reads a string entered at the keyboard. Input stops when the user presses ENTER. The string is not displayed. If the end of the console input stream has been reached, null is returned. An **IOError** is thrown on failure.

The Methods Defined by Console (Contd)

Method	Description
char[ ] readPassword(String fmtString, Object... args)	Displays a prompting string formatted as specified by fmtString and args, and then reads a string entered at the keyboard. Input stops when the user presses ENTER. The string is not displayed. If the end of the console input stream has been reached, null is returned. An IOError is thrown on failure.
PrintWriter writer( )	Returns a reference to a Writer connected to the console.


```
import java.io.*;

class ConsoleDemo
{
    public static void main(String args[])
    {
            String str;
            Console con;

            con = System.console();

            if(con == null)
                    return;

            str = con.readLine("Enter a string: ");
            con.printf("Here is your string: %s\n", str);
    }
}
```

Output:

```
Enter a string: Hello
Here is your string: Hello
```

Serialization

- ***Serialization*** is the process of writing the state of an object to a byte stream.
- This is useful when you want to save the state of your program to a persistent storage area, such as a file.
- At a later time, you may restore these objects by using the process of ***deserialization***.
- Serialization is also needed to implement Remote Method Invocation (RMI).
- RMI allows a Java object on one machine to invoke a method of a Java object on a different machine.
- An object may be supplied as an argument to that remote method.
- The sending machine serializes the object and transmits it.
- The receiving machine deserializes it.

Note:

- Primary purpose of java serialization is to write an object into a stream, so that it can be transported through a network and that object can be rebuilt again.
- When there are two different parties involved, you need a protocol to rebuild the exact same object again.
- Java serialization API just provides you that.
- Other ways you can leverage the feature of serialization is, you can use it to perform a ***deep copy.***

- Assume that an object to be serialized has references to other objects, which, in turn, have references to still more objects.
- This set of objects and the relationships among them form a directed graph.
- There may also be circular references within this object graph.
- That is, object X may contain a reference to object Y, and object Y may contain a reference back to object X.
- Objects may also contain references to themselves.
- The object serialization and deserialization facilities have been designed to work correctly in these scenarios.
- If you attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized.
- Similarly, during the process of deserialization, all of these objects and their references are correctly restored.

Stream Benefits

- The streaming interface to I/O in Java provides a clean abstraction for a complex and often cumbersome task.
- The composition of the filtered stream classes allows you to dynamically build the custom streaming interface to suit your data transfer requirements.
- Java programs written to ensure that the abstract, high-level **InputStream**, **OutputStream**, **Reader**, and **Writer** classes will function properly in the future even when new and improved concrete stream classes are invented.
- This model works very well when we switch from a file system–based set of streams to the network and socket streams.
- Finally, serialization of objects plays an important role in many types of Java programs.
- Java's serialization I/O classes provide a portable solution to this sometimes tricky task.