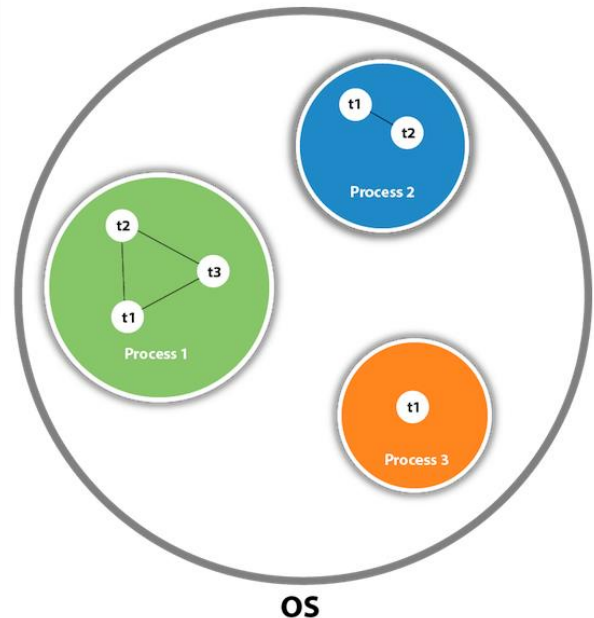


MODULE III

1. MULTITHREADING

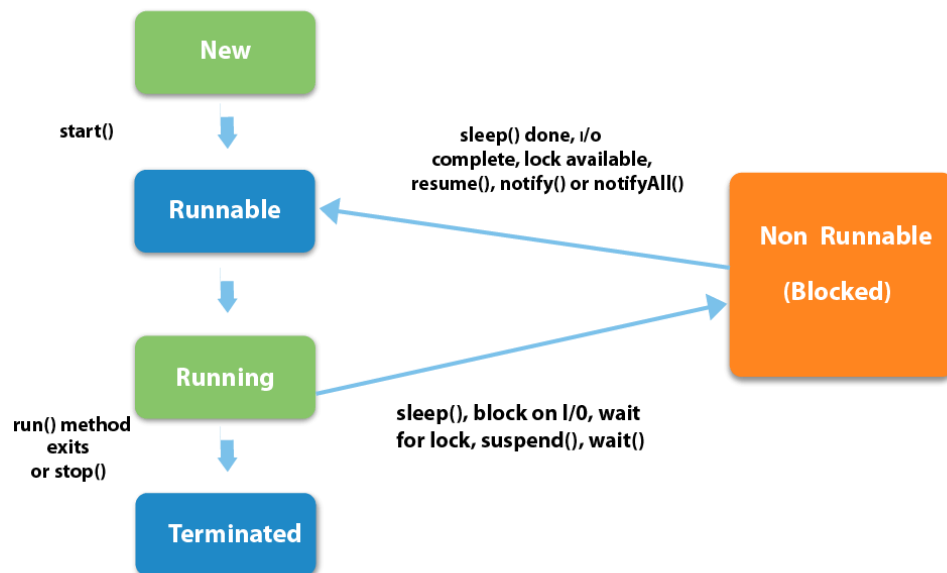
1.1 Introduction to Multithreading:

- Java provides **built-in support for multithreaded programming.**
- A multithreaded program **contains two or more parts (thread) that can run concurrently.**
- Multithreading is a specialized form of multitasking.
- There are two distinct types of multitasking: **process-based** and **thread-based.**
- A process is, in essence, a program that is executing.
- Process-based multitasking is the feature that allows your computer to run two or more programs concurrently. A program is the smallest unit of code that can be dispatched by the scheduler.
 - **Example:** Run the Java compiler at the same time that you are using a text editor.
- 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.
 - **Example:** A text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.
- Multitasking threads require less overhead than multitasking processes.
- **Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly.**
- **Threads are lightweight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost.**
- Multithreaded multitasking is under the control of Java.
- Advantages of Multithreading:
 - It **doesn't block the user** because threads are independent and we can perform multiple operations at the same time.
 - We **can perform many operations together, so it saves time.**
 - Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.



1.2 The Java Thread Model:

- The Java run-time system depends on threads for many things.
- All the class libraries are designed with multithreading in mind.
- In a single-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 **one thread can pause without stopping other parts of your program.** For example, Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause.
- 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.



- The following are the Java Thread states as per javadoc.
 - New:** The thread is in new state if you create an instance of Thread class but before the invocation of `start()` method.
 - Runnable:** The thread is in runnable state after invocation of `start()` method, but the thread scheduler has not selected it to be the running thread.
 - Running:** The thread is in running state if the thread scheduler has selected it.
 - Non-Runnable (Blocked):** This is the state when the thread is still alive, but is currently not eligible to run.
 - Terminated:** A thread is in terminated or dead state when its `run()` method exits.

1.3 Thread Priorities:

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.
- Thread priorities are integers that specify the relative priority of one thread to another.
- A thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch*.
- The rules that determine when a context switch takes place are simple:
 - 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 pre-empted (no matter what it is doing) by a higher-priority thread.

1.4 Synchronization & Messaging:

- If we need multiple threads to communicate and share a common data, we need some way to ensure that they don't conflict with each other.
- That is, you must prevent one thread from writing data while another thread is reading it.
- For this purpose, Java 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.

- 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.

1.5 The Thread Class and the Runnable Interface:

- Java's multithreading system is built upon the Thread class, its methods, and interface, Runnable.
- To create a new thread, your program should **either extend Thread class or implement the Runnable interface**.
- The Thread class defines several methods that help manage threads. The most common ones are:
 1. **public void run():** is used to perform action for a thread.
 2. **public void start():** starts the execution of the thread. JVM calls the run() method on the thread.
 3. **public void sleep(long milliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
 4. **public void join():** waits for a thread to die.
 5. **public void join(long milliseconds):** waits for a thread to die for the specified milliseconds.
 6. **public int getPriority():** returns the priority of the thread.
 7. **public int setPriority(int priority):** changes the priority of the thread.
 8. **public String getName():** returns the name of the thread.
 9. **public void setName(String name):** changes the name of the thread.
 10. **public Thread currentThread():** returns the reference of currently executing thread.
 11. **public int getId():** returns the id of the thread.
 12. **public Thread.State getState():** returns the state of the thread.
 13. **public boolean isAlive():** tests if the thread is alive.
 14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
 15. **public void suspend():** is used to suspend the thread(deprecated).
 16. **public void resume():** is used to resume the suspended thread(deprecated).
 17. **public void stop():** is used to stop the thread(deprecated).
 18. **public boolean isDaemon():** tests if the thread is a daemon thread.
 19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
 20. **public void interrupt():** interrupts the thread.
 21. **public boolean isInterrupted():** tests if the thread has been interrupted.
 22. **public static boolean interrupted():** returns the interrupted flag either true or false.

1.6 The Main Thread:

- **Main thread of our program is executed when our program begins.**
- 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 get the reference of a thread we need to call the the method currentThread(), which is a public static member of Thread class. Its general form is shown here:

static Thread currentThread()

Example:

```
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); //in milliseconds
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Main thread interrupted");
        }
    }
}

```

OUTPUT:

```

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

```

- When we print a thread reference variable it displays, in order: **the name of the thread**, its **priority**, and the **name of its group**.
- By default, the name of the main thread is *main*. Its priority is 5, which is the default value, and *main* is also the name of the group of threads to which this thread belongs.
- A thread group is a data structure that controls the state of a collection of threads as a whole.
- The general form of sleep() method is as follows:
static void sleep(long milliseconds) throws InterruptedException
- The number of milliseconds to suspend is specified in milliseconds. This method may throw an InterruptedException. This would happen if some other thread wanted to interrupt this sleeping one.
- The sleep() method has a second form, shown below, which allows you to specify the period in terms of milliseconds and nanoseconds:
static void sleep(long milliseconds, int nanoseconds) throws InterruptedException
- We can set the name of a thread by using **setName()** and obtain the name of a thread by calling **getName()**
- The signature of these methods are shown below:
final void setName(String threadName)
final String getName()
- Here, *threadName* specifies the name of the thread.

1.7 Creating a Thread:

- We can create a thread either by:
 - Implementing the **Runnable** interface or by
 - Extending the **Thread** class.

Implementing Runnable interface

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

- Inside run(), we will define the code that constitutes the new thread.
- The run() establishes the entry point for another, concurrent thread of execution within our program. This thread will end when run() returns.
- Once you create a class that implements Runnable, you will instantiate an object of type Thread from within that class.
- Thread defines several constructors. The used one has the following signature:
Thread(Runnable threadOb, String threadName)
- In this constructor, threadOb is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName.
- After the new thread is created, it will not start running until you call its start() method, which is declared within Thread. In essence, start() executes a call to run().
- The start() method is shown here:

```
void start( )
```

Example:

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

    // This is the entry point for the second thread
    public void run()
    {
        try
        {
            for(int i = 5; i > 0; i--)
            {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

class ThreadDemo
{
    public static void main(String args[])
    {
        new NewThread(); // create a new thread
        try
        {
            for(int i = 5; i > 0; i--)
            {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Main thread interrupted.");
        }
    }
}
```

```

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

```

OUTPUT:

```

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.

```

- Passing **this** as the first argument to the constructor, Thread will be used to call the start() method which in turn calls run() method on this object.
- The output may vary based on processor speed and task load.
- In a multithreaded program, often the main thread must be the last thread to finish running.

Extending Thread

- We can create a thread by creating a new class that extends Thread, and then to create an instance of that class.
- The extending class must override the run() method, which is the entry point for the new thread.
- It must also call start() to begin execution of the new thread.

Example:

```

class NewThread extends Thread
{
    NewThread()
    {
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start(); // Start the thread
    }

    // This is the entry point for the second thread.
    public void run()
    {
        try
        {
            for(int i = 5; i > 0; i--)
            {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

class ExtendThread
{

```

```

public static void main(String args[])
{
    new NewThread(); // create a new thread
    try
    {
        for(int i = 5; i > 0; i--)
        {
            System.out.println("Main Thread: " + i);
            Thread.sleep(1000);
        }
    }
    catch (InterruptedException e)
    {
        System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
}
}

```

OUTPUT:

```

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.

```

NOTE:

- If you will not be overriding any of Thread's other methods, it is probably best simply to implement Runnable.
- **Can we start a thread twice?** No. After starting a thread, it can never be started again. If you does so, an `IllegalThreadStateException` is thrown. In such case, thread will run once but for second time, it will throw exception. The following program throw an exception.

```

public class TestThreadTwice1 extends Thread
{
    public void run()
    {
        System.out.println("running...");
    }
    public static void main(String args[])
    {
        TestThreadTwice1 t1=new TestThreadTwice1();
        t1.start();
        t1.start();
    }
}

```

1.8 Creating Multiple Thread:

- We can spawn as many threads as we need. For example, the following program creates three child threads:

Example:

```

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.");
    }
}
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.");
    }
}

```

OUTPUT:

```

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
Three: 3
Two: 3
One: 2

```


Three: 2
 Two: 2
 One: 1
 Three: 1
 Two: 1
 One exiting.
 Two exiting.
 Three exiting.
 Main thread exiting.

1.9 Using `isAlive()` and `join()`:

- There are two ways exist to determine whether a thread has finished or not.
 - We can call `isAlive()` on the thread which returns true if the thread upon which it is called is still running else returns false. This method is defined by Thread, and its general form is shown here:


```
final boolean isAlive()
```
 - The method that we use commonly to wait for a thread to finish is called `join()`, whose general form is shown here:


```
public void join() throws InterruptedException
public void join(long milliseconds) throws InterruptedException
```

 - This method waits until the thread on which it is called terminates. i.e. The `join()` method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

Example:

```
class NewThread implements Runnable
{
    String name;
    Thread t;
    NewThread(String threadname)
    {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // 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.");
    }
}

```

OUTPUT:

```

New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
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
Two exiting.
Three exiting.

```

One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.

1.10 ThreadGroup:

- Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.
- Java thread group is implemented by *java.lang.ThreadGroup* class.
- A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.
- A thread is allowed to access information about its own thread group only.
- There are only two constructors of ThreadGroup class.

ThreadGroup(String name): creates a thread group with given name.

ThreadGroup(ThreadGroup parent, String name): creates a thread group with given parent group and name.

- The following code shows a code to group multiple threads.

```
ThreadGroup tg1 = new ThreadGroup("Group A");  
Thread t1 = new Thread(tg1, new MyRunnable(), "one");  
Thread t2 = new Thread(tg1, new MyRunnable(), "two");  
Thread t3 = new Thread(tg1, new MyRunnable(), "three");
```
- Now all 3 threads belong to one group. Here, **tg1** is the thread group name, **MyRunnable** is the class that implements Runnable interface and "one", "two" and "three" are the thread names.
- Now we can interrupt all threads by a single line of code only.

```
Thread.currentThread().getThreadGroup().interrupt();
```

- **ThreadGroup Example:**

```
public class ThreadGroupDemo implements Runnable  
{  
    public void run()  
    {  
        System.out.println(Thread.currentThread().getName());  
    }  
    public static void main(String[] args)  
    {  
        ThreadGroupDemo runnable = new ThreadGroupDemo();  
        ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");  
  
        Thread t1 = new Thread(tg1, runnable, "one");  
        t1.start();  
        Thread t2 = new Thread(tg1, runnable, "two");  
        t2.start();  
        Thread t3 = new Thread(tg1, runnable, "three");  
        t3.start();  
  
        System.out.println("Thread Group Name: "+tg1.getName());  
        tg1.list();  
  
    }  
}
```

OUTPUT:

one
two
three

```

Thread Group Name: Parent ThreadGroup
java.lang.ThreadGroup[name=Parent ThreadGroup,maxpri=10]
  Thread[one,5,Parent ThreadGroup]
  Thread[two,5,Parent ThreadGroup]
  Thread[three,5,Parent ThreadGroup]

```

1.11 Thread Priorities:

- Each thread has a priority. Priorities are represented by a number between **1 and 10**. In most cases, Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.
- In theory, **higher-priority threads get more CPU time** than lower-priority threads.
- To set a thread's priority, use the *setPriority()* method, which is a member of Thread.
- The general form of *setPriority()* method is as follows:
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 class.
- You can obtain the current priority setting by calling the *getPriority()* method of Thread, shown below:
final int getPriority()
- The following example demonstrates two threads at different priorities, which do not run on a preemptive platform in the same way as they run on a non-preemptive platform.
- One thread is set two levels above the normal priority, as defined by Thread.NORM_PRIORITY, and the other is set to two levels below it. The threads are started and allowed to run for ten seconds. Each thread executes a loop, counting the number of iterations. After ten seconds, the main thread stops both threads. The number of times that each thread made it through the loop is then displayed.

Example:

```

class clicker implements Runnable
{
    long click = 0;
    Thread t;
    private volatile boolean running = true;
    public clicker(int p)
    {
        t = new Thread(this);
        t.setPriority(p);
    }
    public void run()
    {
        while (running)
            click++;
    }
    public void stop()
    {
        running = false;
    }
    public void start()
    {
        t.start();
    }
}
class 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);
        lo.start();
        hi.start();
        try
        {
            Thread.sleep(10000);
        }
        catch (InterruptedException e)
        {
            System.out.println("Main thread interrupted.");
        }
        lo.stop();
        hi.stop();
        // wait for child threads to terminate.
        try
        {
            hi.t.join();
            lo.t.join();
        }
        catch (InterruptedException e)
        {
            System.out.println("InterruptedException caught");
        }
        System.out.println("Low-priority thread: " + lo.click);
        System.out.println("High-priority thread: " + hi.click);
    }
}

```

OUTPUT:

Low-priority thread: 4408112
High-priority thread: 589626904

- The exact output produced by this program depends on the speed of the CPU and the number of other tasks running in the system.
- Note that running is preceded by the keyword volatile. **Volatile Keyword is applicable to variables. volatile keyword in Java guarantees that value of the volatile variable will always be read from main memory and not from Thread's local cache.**
- volatile is used here to ensure that the value of “running” is examined each time the following loop iterates:

```

while (running)
{
    click++;
}

```

- Without the use of volatile, Java is free to optimize the loop in such a way that a local copy of running is created.
- The use of volatile prevents this optimization, telling Java that “running” may change in ways not directly apparent in the immediate code.

1.12 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 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 re-enter the same monitor if it so desires.
- Java implements synchronization through language elements, most of the complexity associated with synchronization has been eliminated.
- We can synchronize our code in either of two ways.
 - Using Synchronized Methods
 - Using synchronized statements

1.12.1 Using Synchronized Methods:

- To understand the need for synchronization, let's understand the following example that does not use it.

Example:

```
// This program is not synchronized.
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");
        // wait for threads to end
        try
        {

```

```

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

```

OUTPUT:

```

[Hello[Synchronized[World]
]
]

```

- 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, on the same object, at the same time. This is known as a **race condition**.
- In most situations, a race condition is more subtle and less predictable, because we 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, we must serialize access to `call()`. We 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`, as shown below:

```

class Callme
{
    synchronized void call(String msg)
    {
        ...
    }
}

```

- After `synchronized` has been added to `call()`, the output of the program is as follows:

```

[Hello]
[Synchronized]
[World]

```
- Once a thread enters any `synchronized` method on an instance, no other thread can enter any other `synchronized` method on the same instance.

1.12.2 Using Synchronized Statements:

- `Synchronized` methods will not work in all cases.
- Suppose we want to synchronize access to objects of a class that was not designed for multithreaded access and we do not have access to the source code.
- We can simply put calls to the methods defined by this class 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.
- An alternative version of the preceding example, using a `synchronized` block within the `run()` method is shown below:

Example:

```

class Callme
{
    void call(String msg)
    {
        System.out.print "[" + msg);
        try
        {
            Thread.sleep(1000);
        }
        catch (InterruptedException e)
        {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}
class Caller implements Runnable
{
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s)
    {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }

    // synchronize calls to call()
    public void run()
    {
        synchronized(target)
        {
            // synchronized block
            target.call(msg);
        }
    }
}
class Synch1
{
    public static void main(String args[])
    {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "world");
        // wait for threads to end
        try
        {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        }
        catch(InterruptedException e)
        {
            System.out.println("Interrupted");
        }
    }
}

```

OUTPUT:

```

[Hello]
[Synchronized]
[World]

```


- Here, the call() method is not modified by synchronized. Instead, the synchronized statement is used inside Caller's run() method which causes each thread waits for the prior one to finish before proceeding.

1.13 Inter thread Communication:

- Threads also provide a secondary benefit which do away 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.
- Polling can be avoided by using Java's interprocess communication mechanism via the wait(), notify(), and notifyAll() methods implemented as final methods in Object class.
- The rules for using these methods are actually quite simple:
 - **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. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.
 - **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.)
- Sun recommends that calls to wait() should take place within a loop that checks the condition on which the thread is waiting.
- Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?
 - It is because they are related to lock and object has a lock.
- Difference between wait() and sleep():

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

- To begin, consider the following sample program that incorrectly implements a simple form of the producer/consumer problem.
- It consists of four classes:
 - Q, the queue that we're trying to synchronize;

- Producer, the threaded object that is producing queue entries;
- Consumer, the threaded object that is consuming queue entries; and
- PC, the tiny class that creates the single Q, Producer, and Consumer.

Example:

```

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;
        while(true)
        {
            q.put(i++);
        }
    }
}
class Consumer implements Runnable
{
    Q q;
    Consumer(Q q)
    {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run()
    {
        while(true)
        {
            q.get();
        }
    }
}
class PC
{
    public static void main(String args[])
    {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);
        System.out.println("Press Control-C to stop.");
    }
}

```

OUTPUT:

```

Put: 0
Put: 1

```

```

Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Press Control-C to stop.
Put: 7
Got: 7
Got: 7
Got: 7
Got: 7
...
...

```

- In the above example after the producer produced the item the consumer is consuming the same item repeatedly. Sometime the producer is keep on producing the items without waiting for the consumer to consume.
- The proper way to write this program in Java is to use wait() and notify() to signal in both directions, as shown below:

Example 1:

```

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;
    }
    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();
    }
}

```

```

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

```

OUTPUT:

```

Press Control-C to stop.
Put: 0
Got: 0
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
...
...

```

Example 2:

```

class Customer
{

```

```

        int amount=10000;

        synchronized void withdraw(int amount)
        {
            System.out.println("going to withdraw...");
            if(this.amount<amount)
            {
                System.out.println("Less balance; waiting for deposit...");
                try
                {
                    wait();
                }
                catch(Exception e){}
            }
            this.amount-=amount;
            System.out.println("withdraw completed...");
        }

        synchronized void deposit(int amount)
        {
            System.out.println("going to deposit...");
            this.amount+=amount;
            System.out.println("deposit completed... ");
            notify();
        }
    }
    class MyThread1 extends Thread
    {
        Customer c;
        MyThread1(Customer c)
        {
            this.c=c;
            start();
        }

        public void run()
        {
            c.withdraw(15000);
        }
    }
    class MyThread2 extends Thread
    {
        Customer c;
        MyThread2(Customer c)
        {
            this.c=c;
            start();
        }

        public void run()
        {
            c.deposit(10000);
        }
    }
    class TestThread
    {
        public static void main(String args[])
        {
            final Customer c=new Customer();
            new MyThread1(c);
            new MyThread2(c);
        }
    }
}

```

OUTPUT:

```

going to withdraw...
Less balance; waiting for deposit...

```

going to deposit...
deposit completed...
withdraw completed...

1.15 Interrupting a Thread:

- If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException.
- If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true.
- The three methods provided by the Thread class for interrupting a thread
 - public void interrupt():** breaks out the sleeping or waiting state throwing InterruptedException
 - public static boolean interrupted():** returns the interrupted flag either true or false.
 - public boolean isInterrupted():** returns the interrupted flag after that it sets the flag to false if it is true.
- In this example, after interrupting the thread, we handle the exception, so it will break out the sleeping but will not stop working..

Example 1:

```
class TestInterruptingThread2 extends Thread
{
    public void run()
    {
        try
        {
            Thread.sleep(1000);
            System.out.println("task");
        }
        catch(InterruptedException e)
        {
            System.out.println("Exception handled "+e);
        }
        System.out.println("thread is running...");
    }

    public static void main(String args[])
    {
        TestInterruptingThread2 t1=new TestInterruptingThread2();
        t1.start();
        t1.interrupt();
    }
}
```

OUTPUT:

Exception handled java.lang.InterruptedException: sleep interrupted
thread is running...

- In this example, Thread.interrupted() method call is demonstrated.

Example 2:

```
public class TestInterruptingThread2 extends Thread
{
    public void run()
    {
        for(int i=1;i<=2;i++)
        {
            if(Thread.interrupted())
            {
                System.out.println("code for interrupted thread");
            }
            else
            {

```

```

        System.out.println("code for normal thread");
    }
}

public static void main(String args[])
{
    TestInterruptingThread2 t1=new TestInterruptingThread2();
    TestInterruptingThread2 t2=new TestInterruptingThread2();
    t1.start();
    t1.interrupt();
    t2.start();
}
}

```

OUTPUT:

```

code for interrupted thread
code for normal thread
code for normal thread
code for normal thread

```

1.16 Daemon Threads:

- A daemon thread is a thread that does not prevent the JVM from exiting when the program finishes but the thread is still running.
- Daemon thread in java is a service provider thread that provides services to the user thread.
- Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.
- It is a low priority thread.
- There are many java daemon threads running automatically e.g. gc, finalizer etc.
- The **java.lang.Thread** class provides two methods for java daemon thread.
 - **public void setDaemon(boolean status):** is used to mark the current thread as daemon thread or user thread.
 - **public boolean isDaemon():** is used to check that current is daemon.

Example:

```

public class TestDaemonThread1 extends Thread
{
    public void run()
    {
        if(Thread.currentThread().isDaemon())
        {
            System.out.println("daemon thread work");
        }
        else
        {
            System.out.println("user thread work");
        }
    }
    public static void main(String[] args)
    {
        TestDaemonThread1 t1=new TestDaemonThread1();
        TestDaemonThread1 t2=new TestDaemonThread1();
        TestDaemonThread1 t3=new TestDaemonThread1();

        t1.setDaemon(true);//now t1 is daemon thread

        t1.start();//starting threads
        t2.start();
        t3.start();
    }
}

```

OUTPUT:

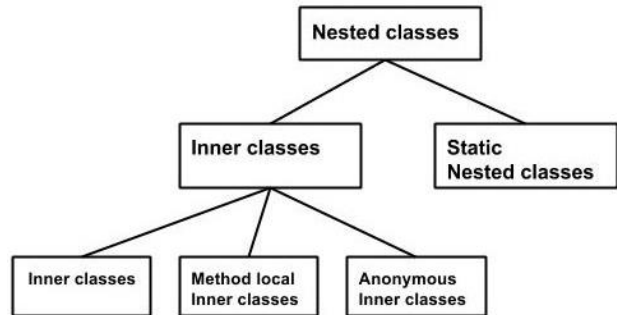
daemon thread work
user thread work
user thread work

NOTE: If you want to make a user thread as Daemon, make it daemon before it is started otherwise it will throw **IllegalThreadStateException**.

2. NESTED AND INNER CLASS

2.1 Introduction to Inner class:

- If we define a **class within another class** such classes are known as *nested classes*.
- If class B is defined within class A, then B does not exist independently of A.
- **A nested class has access to the members, including private members, of the class in which it is nested.**
- However, **the enclosing class does not have access to the members of the nested class.**



- **Advantages of inner class:**
 - Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.
 - Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
 - **Code Optimization:** It requires less code to write.
- There are two types of nested classes:
 - *Static Nested class*
 - *Non-static Nested class*
 - *Inner class*
 - *Method local Inner class*
 - *Anonymous Inner class*

Type	Description
Member Inner Class	A non-static class that is created inside a class but outside a method is called member inner class.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

2.2 Static Nested Class:

- A static nested class is one that has the **static** modifier applied.
- A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members.
- Just like static members, a static nested class does not have access to the instance variables and methods of the outer class.
- A static nested class must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Static nested classes are rarely used.

Example:

```
public class Outer
```



```

{
    static class Nested_Demo
    {
        public void my_method()
        {
            System.out.println("This is my nested class");
        }
    }
    public static void main(String args[])
    {
        Outer.Nested_Demo nested = new Outer.Nested_Demo();
        nested.my_method();
    }
}

```

OUTPUT:

This is my nested class

2.3 Non-Static Nested Class (Member Inner Class):

- An inner class is a non-static nested class. It is the most important type of nested class.
- It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

Example:

```

class Outer
{
    int outer_x = 100;
    void test()
    {
        Inner inner = new Inner();
        inner.display();
    }
    class Inner
    {
        void display()
        {
            System.out.println("display: outer_x = " + outer_x);
        }
    }
}
class InnerClassDemo
{
    public static void main(String args[])
    {
        Outer outer = new Outer();
        outer.test();
    }
}

```

OUTPUT:

display: outer_x = 100

- The java compiler creates two class files in case of inner class. The class file name of inner class is "Outer\$Inner".
- If you want to instantiate inner class, you must have to create the instance of outer class. In such case, instance of inner class is created inside the instance of outer class.
- The Java compiler generates an error message **if any code outside of class Outer attempts to instantiate class Inner.**
- An inner class has access to all of the members of its enclosing class, but the reverse is not true.
- Members of the inner class are known only within the scope of the inner class and may not be used by the outer class.

Example:

```

class Outer
{
    int outer_x = 100;
    void test()
    {
        Inner inner = new Inner();
        inner.display();
    }

    // this is an inner class
    class Inner
    {
        int y = 10; // y is local to Inner
        void display()
        {
            System.out.println("display: outer_x = " + outer_x);
        }
    }
    void showy()
    {
        System.out.println(y); // error, y not known here!
    }
}
class InnerClassDemo
{
    public static void main(String args[])
    {
        Outer outer = new Outer();
        outer.test();
    }
}

```

2.4 Method Local Inner Class:

- It is possible to define inner classes within any block scope. For example, we can define a nested class within the block defined by a method or even within the body of a for loop, as this next program shows.
- A class created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Example:

```

class Outer
{
    int outer_x = 100;
    void test()
    {
        for(int i=0; i<10; i++)
        {
            class Inner
            {
                void display()
                {
                    System.out.println("display: outer_x = " + outer_x);
                }
            }
            Inner inner = new Inner();
            inner.display();
        }
    }
}
class InnerClassDemo
{
    public static void main(String args[])
    {
        Outer outer = new Outer();
        outer.test();
    }
}

```

```
}
```

OUTPUT:

```
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
```

- **Rules:**

- Local variable can't be private, public or protected.
- Local inner class cannot be invoked from outside the method.
- Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in local inner class.

2.5 Anonymous Inner Class:

- Anonymous inner classes are declared without any name at all.
- In case of anonymous inner classes, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface.
- An anonymous inner class can be useful when making an instance of an object with certain “extras” such as overloading methods of a class or interface, without having to actually subclass a class.
- Anonymous inner classes are useful in writing implementation classes for listener interfaces in graphics programming.
- Syntax:

```
// Test can be interface, abstract/concrete class
Test t = new Test()
{
    // data members and methods
    public void test_method()
    {
        .....
        .....
    }
};
```

- To understand anonymous inner class, let us take the following simple program:

```
interface Age
{
    int x = 21;
    void getAge();
}
class AnonymousDemo
{
    public static void main(String[] args)
    {
        // MyClass is implementation class of Age interface
        MyClass obj = new MyClass();

        // calling getAge() method implemented at MyClass
        obj.getAge();
    }
}
```

```
// MyClass implement the methods of Age Interface
class MyClass implements Age
{
    @Override
    public void getAge()
    {
        // printing the age
        System.out.print("Age is "+x);
    }
}
```

- There is no need to write a separate class MyClass. Instead, directly copy the code of MyClass into this parameter, as shown below:

```
Age oj1 = new Age()
{
    @Override
    public void getAge()
    {
        System.out.print("Age is "+x);
    }
};
```

Here, an object to Age is not created but an object of MyClass is created and copied in the entire class code as shown above. This is possible only with anonymous inner class. Anonymous inner class version of the above Program is given below:

```
interface Age
{
    int x = 21;
    void getAge();
}
class AnonymousDemo
{
    public static void main(String[] args)
    {
        // MyClass is hidden inner class of Age interface
        // whose name is not written but an object to it
        // is created.
        Age oj1 = new Age()
        {
            @Override
            public void getAge()
            {
                // printing age
                System.out.print("Age is "+x);
            }
        };
        oj1.getAge();
    }
}
```

- Anonymous classes are created in two ways:

a) Java anonymous inner class using class

Example:

```
class Demo
{
    void show()
    {
        System.out.println("I am in show method of super class");
    }
}
class Flavor1Demo
```

```

{
    // An anonymous class with Demo as base class
    static Demo d = new Demo()
    {
        void show()
        {
            super.show();
            System.out.println("I am in Flavor1Demo class");
        }
    };
    public static void main(String[] args)
    {
        d.show();
    }
}

```

OUTPUT:

I am in show method of super class

I am in Flavor1Demo class

- In the above code, we have two class Demo and Flavor1Demo. Here demo act as super class and anonymous class acts as a subclass, both classes have a method show(). In anonymous class show() method is overridden.

- **Internal working of the following code:**

```

static Demo d = new Demo()
{
    void show()
    {
        super.show();
        System.out.println("I am in Flavor1Demo class");
    }
};

```

1. A class is created but its name is decided by the compiler which extends the Demo class and provides the implementation of the show() method.
 2. An object of Anonymous class is created that is referred by the reference variable d of Demo type.
- The internal class generated by the compiler is as follows:

```

static class Flavor1Demo$1 extends Demo
{
    Flavor1Demo$1(){}
    void show ()
    {
        super.show();
        System.out.println("I am in Flavor1Demo class");
    }
}

```

b) Java anonymous inner class using interface

Example:

```

interface Hello
{
    void show();
}
class Flavor2Demo
{
    // An anonymous class that implements Hello interface
    static Hello h = new Hello()
    {
        public void show()
        {
            System.out.println("I am in anonymous class");
        }
    };
}

```

```

        public static void main(String[] args)
        {
            h.show();
        }
    }

```

OUTPUT:

I am in anonymous class

- In above code we create an object of anonymous inner class but this anonymous inner class is an implementer of the interface Hello.
- Any anonymous inner class can implement only one interface at one time. It can either extend either a class or implement an interface at a time.

3. IO STREAMS

3.1 Introduction:

- A **stream** is a **logical entity 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 they are linked to differ.
- A **stream is a sequence of data**. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.
- In Java, three streams are created for us automatically. All these streams are attached with the console.
 1. **System.out**: standard output stream
 2. **System.in**: standard input stream
 3. **System.err**: standard error stream
- The I/O classes defined by java.io are listed below:

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	

- The I/O interfaces defined by java.io are listed below:

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

3.2 File:

- The File class does not operate on streams. It directly deals with files and the file system.
- 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**.
- The following constructors can be used to create File objects:

File(String directoryPath)

File(String directoryPath, String filename)

File(File dirObj, String filename)

Example:

```
File f1 = new File("/");
File f2 = new File("/", "autoexec.bat");
File f3 = new File(f1, "autoexec.bat");
```

- The following example demonstrates several of the File methods:

```
import java.io.File;
class FileDemo
{
    public static void main(String args[])
    {
        File f1 = new File("\\JavaPrograms\\A5.java");
        System.out.println("File Name: " + f1.getName());
        System.out.println("Path: " + f1.getPath());
        System.out.println("Abs Path: " + f1.getAbsolutePath());
        System.out.println("Parent: " + f1.getParent());
        System.out.println(f1.exists() ? "exists" : "does not exist");
        System.out.println(f1.canWrite() ? "is writeable" : "is not writeable");
        System.out.println(f1.canRead() ? "is readable" : "is not readable");
        System.out.println("is " + (f1.isDirectory() ? "" : "not") + " a directory");
        System.out.println(f1.isFile() ? "is normal file" : "might be a named pipe");
        System.out.println(f1.isAbsolute() ? "is absolute" : "is not absolute");
        System.out.println("File last modified: " + f1.lastModified());
        System.out.println("File size: " + f1.length() + "Bytes");
    }
}
```

Output:

```

File Name: A5.java
Path: \JavaPrograms\A5.java
Abs Path: H:\JavaPrograms\A5.java
Parent: \JavaPrograms
exists
is writeable
is readable
is not a directory
is normal file
is not absolute
File last modified: 1535033284000
File size: 314 Bytes

```

- Besides the methods used in the above program, File class also includes following two useful utility methods:

boolean renameTo(File newName): Renames the invoking File object to the name newName

boolean delete(): deletes the invoking File object. We can also use delete() to delete a directory if the directory is empty.

3.3 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 one is shown below:

String[] list()

- The list of files is returned in an array of String objects.
- The following program shows how to use list() to examine the contents of a directory:

```

import java.io.File;
class DirList
{
    public static void main(String args[])
    {
        String dirname = "H:\\\\ADA";
        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");
    }
}
}

```

Output:

```

H:\JavaPrograms>java DirList
Directory of H:\ADA
Syllabus.docx is a file
EBooks is a directory
Exam&Assignment is a directory
LectureNoteSPP is a directory
StudyMaterialFromNet is a directory

```

- The second form of list() includes only those files that match a certain filename pattern, or filter.
- The second form of list(), is shown below:

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.
- The following program displays only .docx files of the directory H:\JAVA-2018.

```

//save this as OnlyExt.java
import java.io.*;
public class OnlyExt implements FilenameFilter
{
    String ext;
    public OnlyExt(String ext)
    {
        this.ext = "." + ext;
    }
    public boolean accept(File dir, String name)
    {
        return name.endsWith(ext);
    }
}
//save this as DirListOnly.java
// Directory of .docx files.
import java.io.*;
class DirListOnly
{
    public static void main(String args[])
    {
        String dirname = "H:\\\\JAVA-2018";
        File f1 = new File(dirname);
        FilenameFilter only = new OnlyExt("docx");
        String s[] = f1.list(only);
    }
}

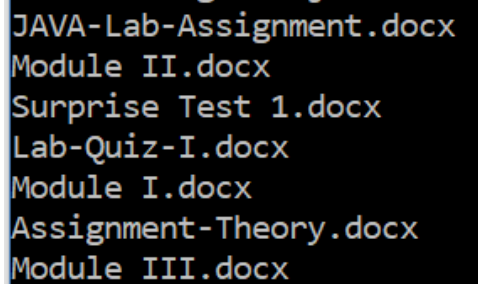
```

```

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

```

Output:



```

JAVA-Lab-Assignment.docx
Module II.docx
Surprise Test 1.docx
Lab-Quiz-I.docx
Module I.docx
Assignment-Theory.docx
Module III.docx

```

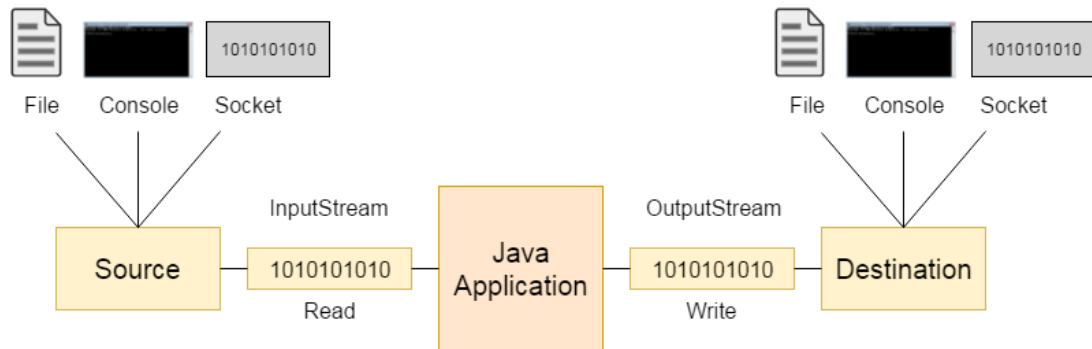
- There is a variation to the `list()` method, called `listFiles()`, which return the file list as an array of `File` objects instead of strings..
- The signatures for `listFiles()` are:
 - `File[] listFiles()`**
 - `File[] listFiles(FilenameFilter FFObj)`**
- **Creating Directories:**
 - **The `mkdir()` method creates a directory**, returns true on success and false on failure. Failure may be due to 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 `makedirs()` method. It creates both a directory and all the parents of the directory.

3.4 The Stream Classes:

- **Java's stream-based I/O is built upon four abstract classes:** `InputStream`, `OutputStream`, `Reader`, and `Writer`.
- `InputStream` and `OutputStream` are designed for byte streams. `Reader` and `Writer` are designed for character streams.
- **Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit Unicode.**
- In general, we 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.

3.4.1 The Byte Streams:

- **Byte stream classes are topped by `InputStream` and `OutputStream`.**
- **`InputStream` is an abstract class that defines Java's model of streaming byte input.** Java application uses an **input stream** to read data from a source; it may be a file, an array, peripheral device or socket. It **implements the `Closeable` interface**. Most of the methods in this class will throw an **`IOException`** on error conditions.
- **`OutputStream` is an abstract class that defines streaming byte output.** It **implements the `Closeable` and `Flushable` interfaces**. Most of the methods in this class return void and throw an **`IOException`** in the case of errors.
- The working of Java `OutputStream` and `InputStream` by the figure given below:



- The methods defined by `InputStream` and `OutputStream` are shown in the following table:

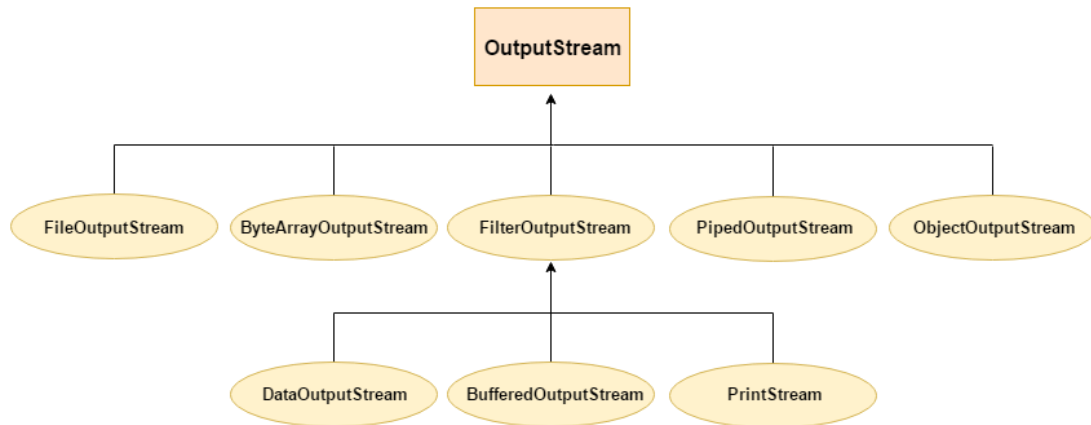
Method	Description
<code>int available()</code>	Returns the number of bytes of input currently available for reading.
<code>void close()</code>	Closes the input source. Further read attempts will generate an IOException .
<code>void mark(int numBytes)</code>	Places a mark at the current point in the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

The Methods Defined by `InputStream`

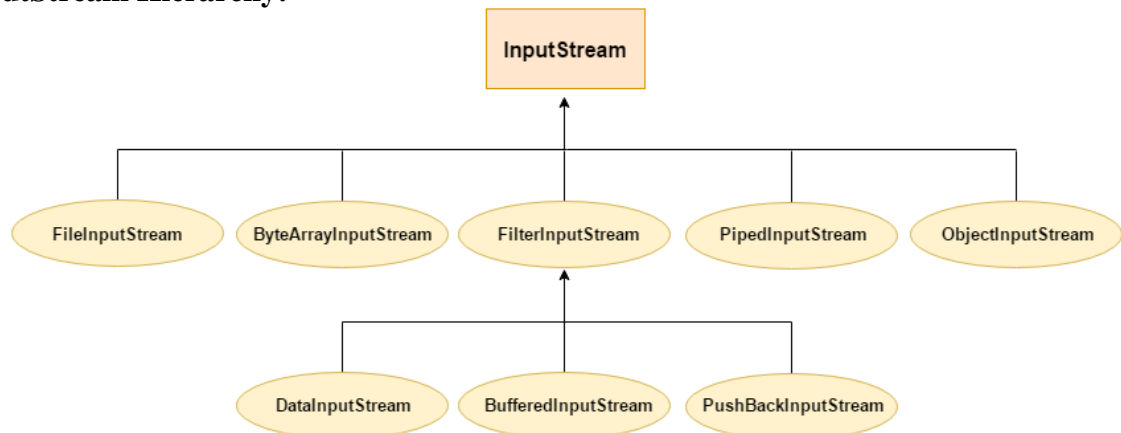
Method	Description
<code>void close()</code>	Closes the output stream. Further write attempts will generate an IOException .
<code>void flush()</code>	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
<code>void write(int b)</code>	Writes a single byte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

The Methods Defined by `OutputStream`

- OutputStream Hierarchy:**



- **InputStream Hierarchy:**



1. Java FileOutputStream Class

- Java **FileOutputStream** is an output stream used for writing data to a file.
- **To write primitive values into a file**, we use FileOutputStream class.
- We can write byte-oriented as well as character-oriented data through FileOutputStream class. However, for character-oriented data, it is preferred to use FileWriter than FileOutputStream.
- Its most common constructors is shown below:

FileOutputStream(String filepath)

Example:

```

import java.io.FileOutputStream;
public class FileOutputStreamExample
{
    public static void main(String args[])
    {
        try
        {
            FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
            fout.write(65);
            String s="welcome to silicon.";
            byte b[]=s.getBytes();
            //converting string into byte array
            fout.write(b);
            fout.close();
            System.out.println("success...");
        }
        catch(Exception e){System.out.println(e);}
    }
}
  
```

Output:

success...

Contents of file testout.txt:

AWelcome to Silicon.

2. Java FileInputStream Class

- Java **FileInputStream** class obtains input bytes from a file.
- It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. We can also read character-stream data. However, for reading streams of characters, it is recommended to use `FileReader` class.
- Its two most common constructors are shown here:

FileInputStream(String filepath)

FileInputStream(File fileObj)

Example:

```
import java.io.FileInputStream;
public class DataStreamExample
{
    public static void main(String args[])
    {
        try
        {
            FileInputStream fin=new FileInputStream("D:\\testout.txt");
            int i=0;
            while((i=fin.read())!=-1)
            {
                System.out.print((char)i);
            }
            fin.close();
        }
        catch(Exception e){System.out.println(e);}
    }
}
```

Output:

AWelcome to Silicon.

Example (Coping a text file into another):

```
import java.io.*;
class CopyFile
{
    public static void main(String args[]) throws IOException
    {
        int i;
        FileInputStream fin;
        FileOutputStream fout;

        // open input file
        try
        {
            fin = new FileInputStream(args[0]);
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Input File Not Found");
            return;
        }

        // open output file
        try
        {
            fout = new FileOutputStream(args[1]);
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Error Opening Output File");
            return;
        }
    }
}
```

```

// Copy File
try
{
    do
    {
        i = fin.read();
        if(i != -1)
            fout.write(i);
    }
    while(i != -1);
}
catch(IOException e)
{
    System.out.println("File Error");
}
fin.close();
fout.close();
}

```

Run: java CopyFile Sample.java copy.txt

Output:

Copies the content of Sample.java file into copy.txt file.

3. Java BufferedOutputStream Class

- Java **BufferedOutputStream** class is **used for buffering an output stream**.
- It internally uses buffer to store data. It **adds more efficiency than to write data directly into a stream**. So, it **makes the performance fast**.
- The two most common constructors are shown here:

BufferedOutputStream(OutputStream os): It creates the new buffered output stream which is used for writing the data to the specified output stream.

BufferedOutputStream(OutputStream os, int size): It creates the new buffered output stream which is used for writing the data to the specified output stream with a specified buffer size.

Example:

```

import java.io.*;
public class BufferedOutputStreamExample
{
    public static void main(String args[])throws Exception
    {
        FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
        BufferedOutputStream bout=new BufferedOutputStream(fout);
        String s="welcome to Silicon.";
        byte b[]=s.getBytes();
        bout.write(b);
        bout.flush();    // It flushes the buffered output stream.
        bout.close();
        fout.close();
        System.out.println("success");
    }
}

```

Output:

success...

Contents of file testout.txt:

Welcome to Silicon.

Note: The flush() flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.

4. Java BufferedInputStream Class

- Java **BufferedInputStream** class is used to **read information from stream**. It **internally uses buffer mechanism to make the performance fast**.
- The important points about BufferedInputStream are:

- When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
- When a `BufferedInputStream` is created, an internal buffer array is created.
- The two most common constructors are shown here:
 - `BufferedInputStream(InputStream IS)`:** It creates the `BufferedInputStream` and saves it argument, the input stream `IS`, for later use.
 - `BufferedInputStream(InputStream IS, int size)`:** It creates the `BufferedInputStream` with a specified buffer size and saves it argument, the input stream `IS`, for later use.

Example:

```
import java.io.*;
public class BufferedInputStreamExample
{
    public static void main(String args[])
    {
        try
        {
            FileInputStream fin=new FileInputStream("D:\\testout.txt");
            BufferedInputStream bin=new BufferedInputStream(fin);
            int i;
            while((i=bin.read())!=-1)
            {
                System.out.print((char)i);
            }
            bin.close();
            fin.close();
        }
        catch(Exception e){System.out.println(e);}
    }
}
```

Output:

Welcome to Silicon.

5. Java RandomAccessFile Class

- This class is used for reading and writing to random access file.
- A random access file behaves like a large array of bytes. There is a cursor implied to the array called file pointer, by moving the cursor we do the read write operations.
- If end-of-file is reached before the desired number of byte has been read than `EOFException` is thrown. It is a type of `IOException`.
- The constructor of this class are as follows:

Constructor	Description
<code>RandomAccessFile(File file, String mode)</code>	Creates a random access file stream to read from, and optionally to write to, the file specified by the <code>File</code> argument.
<code>RandomAccessFile(String name, String mode)</code>	Creates a random access file stream to read from, and optionally to write to, a file with the specified name.

- Some of the important methods of this class are as follows:

Modifier and Type	Method	Description
void	<code>close()</code>	It closes this random access file stream and releases any system resources associated with the stream.

void	seek(long pos)	It sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.
void	writeDouble(double v)	It converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.
void	writeFloat(float v)	It converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.
void	write(int b)	It writes the specified byte to this file.
int	read()	It reads a byte of data from this file.
long	length()	It returns the length of this file.

- The following program both reads and writes into the file copy.txt randomly.

Example:

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomAccessFileExample
{
    static final String FILEPATH ="Copy.TXT";
    public static void main(String[] args)
    {
        try
        {
            System.out.println(new String(readFromFile(FILEPATH, 0, 18)));
            writeToFile(FILEPATH, "I LOVE MY COUNTRY AND MY PEOPLE", 31);
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
    private static byte[] readFromFile(String filePath, int position, int
                                    size) throws IOException
    {
        RandomAccessFile file = new RandomAccessFile(filePath, "r");
        file.seek(position);
        byte[] bytes = new byte[size];
        file.read(bytes);
        file.close();
        return bytes;
    }
    private static void writeToFile(String filePath, String data, int
                                    position) throws IOException
    {
        RandomAccessFile file = new RandomAccessFile(filePath, "rw");
        file.seek(position);
        file.write(data.getBytes());
        file.close();
    }
}
```

Content of copy.txt before run:

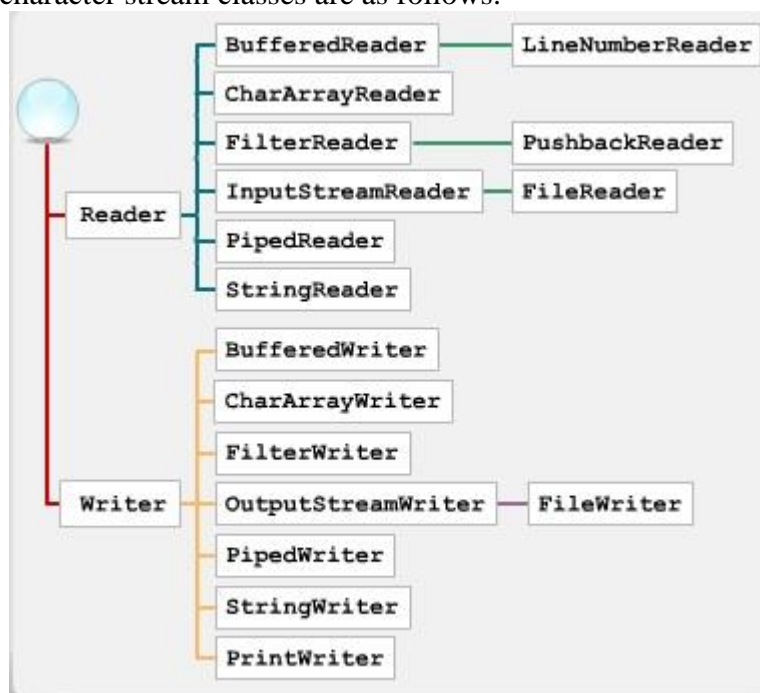
This class is used for reading and writing to random access file. This is used as input/output file.

Content of copy.txt after run:

This class is used for reading **I LOVE MY COUNTRY AND MY PEOPLE**le.
This is used as input/output file.

3.4.2 The Character Streams:

- In Java, **characters are stored using Unicode conventions**. Character stream automatically allows us to read/write data character by character. For example FileReader and FileWriter are character streams used to read from source and write to destination.
- Java Byte streams are used to perform input and output of 8-bit bytes, whereas **Java Character streams are used to perform input and output for 16-bit unicode**.
- Byte streams cannot work directly with Unicode characters.
- The hierarchy of character stream classes are as follows:



- At the top of the character stream hierarchies are the Reader and Writer abstract classes.
- Though there are many classes related to character streams but the most frequently used classes are, FileReader and FileWriter.
- Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.
- **When to use Character Stream over Byte Stream?**
 - In Java, characters are stored using Unicode conventions. Character stream is useful when we want to process text files. These text files can be processed character by character. A character size is typically 16 bits.
- **When to use Byte Stream over Character Stream?**
 - Byte oriented reads byte by byte. A byte stream is suitable for processing raw data like binary files.

1. 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.

2. 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.

3. FileReader Class

- The `FileReader` class creates a `Reader` that we can use to read the contents of a file.
- The two most common **constructors** are shown here:
 - `FileReader(String filePath)`:** It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws `FileNotFoundException`.
 - `FileReader(File fileObj)`:** It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws `FileNotFoundException`.
- **Methods** of `FileReader` class:
 - `int read()`:** It is used to return a character in ASCII form. It returns -1 at the end of file.
 - `void close()`:** It is used to close the `FileReader` class.

Example:

```
import java.io.FileReader;
public class FileReaderExample
{
    public static void main(String args[])throws Exception
    {
        FileReader fr=new FileReader("D:\\testout.txt");
        int i;
        while((i=fr.read())!=-1)
            System.out.print((char)i);
        fr.close();
    }
}
```

Content of testout.txt before run:

Welcome to Java.

Output:

Welcome to Java.

4. FileWriter Class

- `FileWriter` creates a `Writer` that you can use to write to a file.
- The two most common **constructors** are shown here:
 - `FileWriter (String filePath)`:** Creates a new file. It gets file name in string..
 - `FileWriter (File fileObj)`:** Creates a new file. It gets file name in `File` object.
 - `FileWriter (String filePath, boolean append)`:** Creates a new file. It gets file name in string. If `append` is true, then output is appended to the end of the file.
 - `FileWriter (File fileObj, boolean append)`:** Creates a new file. It gets file name in `File` object. If `append` is true, then output is appended to the end of the file.
- **Methods** of `FileWriter` class:
 - `void write(String text)`:** It is used to write the string into `FileWriter`.
 - `void write(char c)`:** It is used to write the char into `FileWriter`.
 - `void write(char[] c)`:** It is used to write char array into `FileWriter`.
 - `void flush()`:** It is used to flushes the data of `FileWriter`.
 - `void close()`:** It is used to close the `FileWriter`.

Example:

```
import java.io.FileWriter;
public class FileWriterExample
{
    public static void main(String args[])
    {
        try
        {
            FileWriter fw=new FileWriter("D:\\testout.txt");
            fw.write("welcome to Java");
            fw.close();
        }
        catch(Exception e){System.out.println(e);}
        System.out.println("Success...");
    }
}
```

Content of testout.txt after run:

Welcome to Java.

Output:

Success . . .

5. **BufferedReader Class**

- BufferedReader reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
- Java BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method. It makes the performance fast. It inherits Reader class.
- The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.
- It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders.
- The two most common **constructors** are shown here:

BufferedReader(Reader rd): It is used to create a buffered character input stream that uses the default size for an input buffer.

BufferedReader(Reader rd, int size): It is used to create a buffered character input stream that uses the specified size for an input buffer.

- **Methods** of BufferedReader class:

Method	Description
int read()	It is used for reading a single character.
int read(char[] cbuf, int off, int len)	It is used for reading characters into a portion of an array.
boolean markSupported()	It is used to test the input stream support for the mark and reset method.
String readLine()	It is used for reading a line of text.
boolean ready()	It is used to test whether the input stream is ready to be read.
long skip(long n)	It is used for skipping the characters.
void reset()	It repositions the stream at a position the mark method was last called on this input stream.
void mark(int readAheadLimit)	It is used for marking the present position in a stream.
void close()	It closes the input stream and releases any of the system resources associated with the stream.

- In the following program we are reading the data from the text file testout.txt using Java BufferedReader class.

Example:

```
import java.io.*;
public class BufferedReaderExample
{
    public static void main(String args[])throws Exception
    {
        FileReader fr=new FileReader("D:\\testout.txt");
        BufferedReader br=new BufferedReader(fr);

        int i;
        while((i=br.read())!=-1)
        {
            System.out.print((char)i);
        }
        br.close();
    }
}
```

```

        fr.close();
    }
}

```

Content of testout.txt:

Welcome to Java.

Output:

Welcome to Java.

- In the following program we are reading data from console by InputStreamReader and BufferedReader.
- An InputStreamReader is a bridge from byte streams to character streams. It reads bytes and decodes them into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.
- In this example, we are connecting the BufferedReader stream with the InputStreamReader stream for reading the line by line data from the keyboard.

Example:

```

import java.io.*;
public class BufferedReaderExample
{
    public static void main(String args[])throws Exception
    {
        InputStreamReader r=new InputStreamReader(System.in);
        BufferedReader br=new BufferedReader(r);
        System.out.println("Enter your name");
        String name=br.readLine();
        System.out.println("welcome "+name);
    }
}

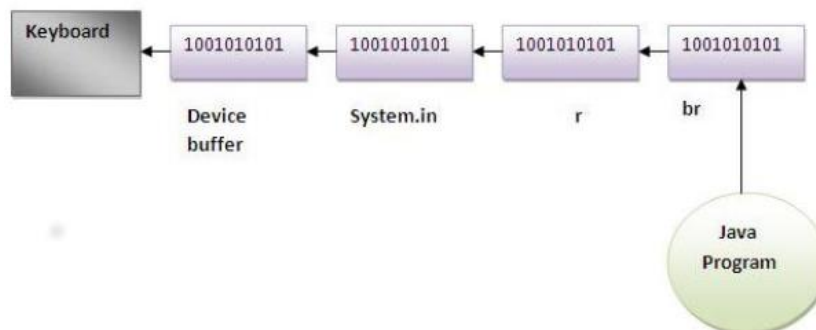
```

Output:

Enter your name

Ipsita Panda

Welcome Ipsita Panda



6. BufferedWriter Class

- Java BufferedWriter class is used to provide buffering for Writer instances.
- It makes the performance fast. It inherits Writer class.
- The buffering characters are used for providing the efficient writing of single arrays, characters, and strings.
- The two most common **constructors** are shown here:

BufferedWriter(Writer wrt): It is used to create a buffered character output stream that uses the default size for an output buffer.

BufferedWriter(Writer wrt, int size): It is used to create a buffered character output stream that uses the specified size for an output buffer.

- **Methods of BufferedWriter class:**

Method	Description
void newLine()	It is used to add a new line by writing a line separator.

void write(int c)	It is used to write a single character.
void write(char[] cbuf, int off, int len)	It is used to write a portion of an array of characters.
void write(String s, int off, int len)	It is used to write a portion of a string.
void flush()	It is used to flushes the input stream.
void close()	It is used to closes the input stream

- In the following program we are writing the data to a text file testout.txt using Java BufferedWriter.

Example:

```
import java.io.*;
public class BufferedWriterExample
{
    public static void main(String[] args) throws Exception
    {
        FileWriter writer = new FileWriter("D:\\testout.txt");
        BufferedWriter buffer = new BufferedWriter(writer);
        buffer.write("welcome to Java.");
        buffer.close();
        System.out.println("Success");
    }
}
```

Output:

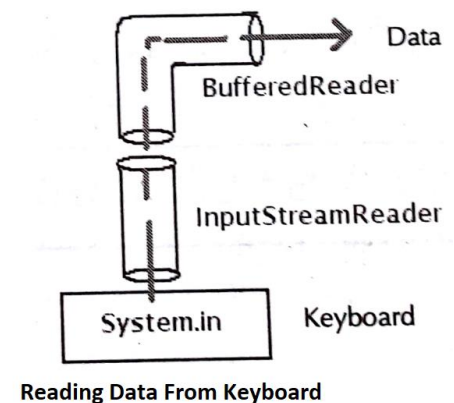
Success

Content of testout.txt:

Welcome to Java.

3.5. Taking Input from keyboard: In Java, there are different ways for reading input from the user.

1. **Using BufferedReader Class:** This is the Java classical method to take input, introduced in JDK1.0. This method is used by wrapping the **System.in** (standard input stream) in an **InputStreamReader** which is wrapped in a **BufferedReader**, we can read input from the user in the command line.
 - A stream is required to accept input from the keyboard. A stream (just like water pipe) represents flow of data from one place to another. It is always required if we want to move data from one place to another.
 - There are two types of streams: Input stream receives the data and output stream sends the data. All these streams are available in **java.io** package.
 - In the **System** class of **java.lang** package we have the following members which represents different Stream objects:
 - **System.in:** Represents **InputStream** object, which by default represents standard input device keyboard.
 - **System.out:** Represents **PrintStream** object, which by default represents standard output device monitor. This is used to display normal messages.
 - **System.err:** Represents **PrintStream** object, which by default represents standard output device monitor. This is used to display error messages.
 - To accept data from keyboard (**System.in**):



- **Connect the keyboard to an input stream object.** Here we can use **InputStreamReader** that can read the data from the keyboard.

```
InputStreamReader obj = new InputStreamReader(System.in);
```
- **Connect the InputStreamReader to BufferedReader** (which is another input type of stream) with methods to read data properly.

```
BufferedReader br = new BufferedReader(obj);
```
- The above two steps can be combined to

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```
- Accepting a single character:

```
char ch = (char)br.read(); // read() method reads a single character and returns its ASCII number.
```
- Accepting a string:

```
String str = br.readLine(); // readLine() method reads and returns a string.
```
- Accepting an integer: Accept the integer as a string and convert it into integer.

```
int n = Integer.parseInt(br.readLine()); // parseInt() is a static method of Integer class.
```
- Accepting a float: Accept the float as a string and convert it into float.

```
float f = Float.parseFloat(br.readLine()); // parseFloat() is a static method of Float class.
```
- Accepting a double value: Accept the double as a string and convert it into double.

```
double d = Double.parseDouble(br.readLine()); // parseDouble() is a static method of Double class.
```
- Accepting a byte value: Accept the byte as a string and convert it into byte.

```
byte b = Byte.parseByte(br.readLine()); // parseByte() is a static method of Byte class.
```
- Accepting a short value: Accept the short as a string and convert it into short.

```
short s = Short.parseShort(br.readLine()); // parseShort() is a static method of Short class.
```
- Accepting a long value: Accept the long as a string and convert it into long.

```
long l = Long.parseLong(br.readLine()); // parseLong() is a static method of Long class.
```
- Accepting a boolean value: Accept the long as a string and convert it into long.

```
boolean x = Boolean.parseBoolean(br.readLine()); // parseBoolean() is a static method of Boolean class.
```

NOTE: Byte, Short, Integer, Long, Float, Double, and Boolean are called wrapper classes available in java.lang package.

Example:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class First
{
    public static void main(String[] args) throws IOException
    {
        //Enter data using BufferReader
        BufferedReader reader = new BufferedReader
            (new InputStreamReader(System.in));

        // Reading data using readLine
        String name = reader.readLine();
```

```

        // Printing the read line
        System.out.println(name);
    }
}

```

Input: Silicon

Output: Silicon

2. **Using Console instance:** The **java.io.Console** class which provides convenient methods for reading input and writing output to the standard input (keyboard) and output streams (display) in command-line (console) programs. Java introduced the Console class (since Java 6) which is more convenient to work with the standard input/output streams than the traditional approach (using System.in and System.out).

- We obtain the unique instance of the Console class via the static method console() of the System class like this:

```
Console console = System.console();
```

- However this method returns null if the program is running in non-interactive environment such as inside an IDE, so we need to check to ensure that the Console is actually available for use:

```

if (console == null)
{
    System.out.println("Console is not supported");
    System.exit(1);
}

```

- We can use the following methods for dealing with the standard input/output streams:

- printf(String format, Object... args): works same as System.out.printf()
- readLine() : works same as BufferedReader.readLine()
- readLine(String fmt, Object... args): prints a formatted string and reads input
- readPassword(): reads a password to a char array
- readPassword(String fmt, Object... args): prints a formatted string and reads a password to a char array. This method allows a program to read a password from the user securely. First, it suppresses echoing, so the password is not visible on the user's screen. Second, readPassword() returns a character array, not a String, so the password can be overwritten, removing it from memory as soon as it is no longer needed.

- The following program demonstrates how to use the Console class to read input data from the user and print output:

Example:

```

import java.io.*;
import java.util.*;

public class ConsoleInputOutputExample
{
    public static void main(String[] args) throws IOException
    {
        Console console = System.console();

        if (console == null)
        {
            System.out.println("Console is not supported");
            System.exit(1);
        }

        String name = console.readLine("what's your name? ");
    }
}

```



```

        String age = console.readLine("How old are you? ");
        String city = console.readLine("Where do you live? ");

        console.format("%s, a %s year-old man is living in
                        %s", name, age, city);
    }
}

```

Input:

What's your name? John
 How old are you? 40
 Where do you live? California

Output:

John, a 40 year-old man is living in California

- The following program demonstrates using the Console class to read login information from the user:

Example:

```

import java.io.*;
import java.util.*;

public class ConsoleLoginExample
{
    public static void main(String[] args) throws IOException
    {
        Console console = System.console();

        if (console == null)
        {
            .out.println("Console is not supported");
            System.exit(1);
        }

        console.printf("Welcome to Java Expert Program!\n");
        String name = console.readLine("Enter your name: ");
        char[] password = console.readPassword("Enter your
                                                password: ");

        char[] correctPassword = {'n', 'i', 'm', 'd', 'a'};

        if (Arrays.equals(password, correctPassword))
            console.printf("Thanks %s, you are logged
                           in.\n", name);
        else
            console.printf("Sorry, you are denied.\n");
        Arrays.fill(password, ' ');
        Arrays.fill(correctPassword, ' ');
    }
}

```

Input:

Welcome to Java Expert Program!
 Enter your name: John
 Enter your password:

Output:

Thanks John, you are logged in.