4.1 Multithreading in Java

Java provides built-in support for *multithreaded programming*. *Thread* is basically a lightweight sub-process, a smallest unit of processing, that can run concurrently with the other parts (other threads) of the same process. **The process of executing multiple threads simultaneously is known as multithreading.**

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread*, and each thread defines a separate path of execution. Thus, *multithreading* is a specialized form of multitasking.

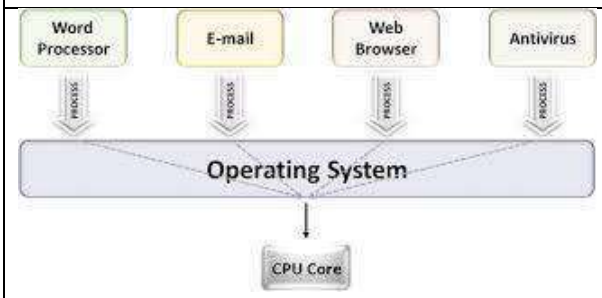
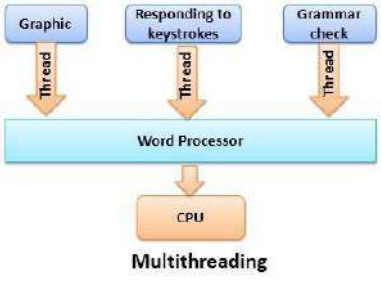
Advantages of Java Multithreading

1. It is mostly used in games, animation etc
2. It doesn't block the user because threads are independent and you can perform multiple operations at same time.
3. You can perform many operations together so it saves time.
4. Threads are independent so it doesn't affect other threads if exception occur in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

1. Process-based (Multiprocessing)
2. Thread-based (Multithreading)

Process-based	Thread-based
1. A *process* is a program that is executing	1. The thread is the smallest unit of dispatchable code.
2. It allows your computer to run two or more programs concurrently.	2. It allows a single program can perform two or more tasks simultaneously
3. Processes are heavy weight	3. Threads are light weight
4. They require own separate address and memory spaces.	4. They share common address and memory spaces.
5. Context switching is costlier.	5. Context switching is lower in cost.
6. Inter-process communication is expensive and limited	6. Inter-thread communication is inexpensive
7. For example, it enables you to listen the music at the same time that you are writing to a text editor and download content from internet.	7. For 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
	

4.2 The Java Thread Model

The Java language and its run-time system was designed keeping in mind about multithreading. The run-time system depend upon multithreading. Java provides asynchronous thread environment, this helps to increase the utilization of CPU.

Multithreading is best in all cases in contrast with single-thread model. Single-thread system uses an approach of event loop with polling. According to this approach a single thread in the system runs in an infinite loop. Polling is the mechanism, that selects a single event from the event queue to choose what to do next. As the event is selected, then event loop forwards the control to the corresponding required event handler. Nothing else can be happened, until the event handler returns. Because of this CPU time is wasted. Here, only one part of the complete program is dominating the whole system, and preventing the system to execute or start any other process. In single-thread model one thread blocks all other threads until its execution completes. On other waiting or idle thread can start and acquire the resource which is not in use by the current thread. This causes the wastage of resources.

Java's multithreading provides benefit in this area by eliminating the loop and polling mechanism, one thread can be paused without stopping the other parts of the program. If any thread is paused or blocked, still other threads continue to run.

As the process has several states, similarly a thread exists in several states. A thread can be in the following states:

Ready to run (New): First time as soon as it gets CPU time.

Running: Under execution.

Suspended: Temporarily not active or under execution.

Blocked: Waiting for resources.

Resumed: Suspended thread resumed, and start from where it left off.

Terminated: Halts the execution immediately and never resumes.

Java thread model can be defined in the following three sections:

Thread Priorities

Each thread has its own priority in Java. Thread priority is an absolute integer value. Thread priority decides only when a thread switches from one running thread to next, called *context switching.* Priority does increase the running time of the thread or gives faster execution.

Synchronization

Java supports an asynchronous multithreading, any number of thread can run simultaneously without disturbing other to access individual resources at different instant of time or shareable resources. But some time it may be possible that shareable resources are used by at least two threads or more than two threads, one has to write at the same time, or one has to write and other thread is in the middle of reading it. For such type of situations and circumstances Java implements synchronization model called *monitor*. The monitor was first defined by C.A.R. Hoare. You can consider the monitor as a box, in which only one thread can reside. As a thread enter in monitor, all other threads have to wait until that thread exits

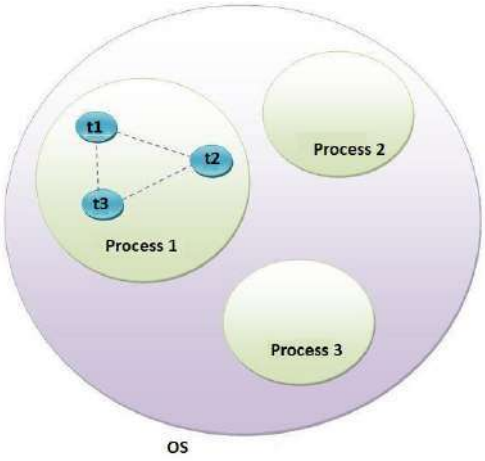
from the monitor. In such a way, a monitor protects the shareable resources used by it being manipulated by other waiting threads at the same instant of time. Java provides a simple methodology to implement synchronization.

Messaging

A program is a collection of more than one thread. Threads can communicate with each other. Java supports messaging between the threads with lost-cost by *Java's Messaging System*. It provides methods to all objects for inter-thread communication. As a thread exits from synchronization state, it notifies all the waiting threads.

4.3 What is a Thread?

- ✓ A Thread is a lightweight sub process, a smallest unit of processing.
- ✓ It is a separate path of execution.
- ✓ Threads are independent, if there occurs exception in one thread, it doesn't affect other threads.
- ✓ They share common memory area.



As shown in the above figure, Thread is executed inside the process. There is a context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

Types of Thread

There are two types of threads. They are:

1. Daemon or System defined Thread (Example: Garbage Collector)
2. User defined Thread (Created by Programmer)

Thread Life Cycle

A Thread can be in one of the five states. According to ***SUN***, there is only 4 states in thread life cycle in java. NEW, RUNNABLE, NON-RUNNABLE and TERMINATED. There is no running state.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)

5. Terminated

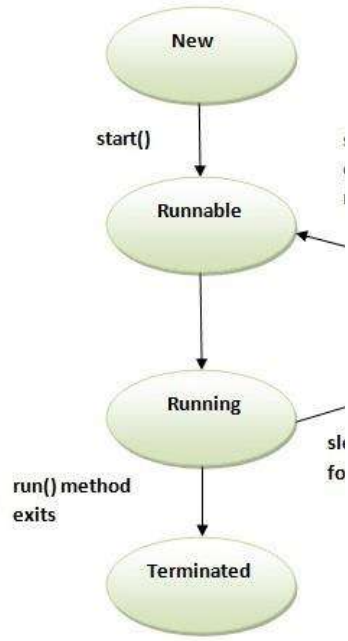


Fig. Thread states

New

The *Thread* is in new state if you create an instance of Thread class but before the invocation of *start()* method. When a thread is in the newborn state, calling any method other than *start ()* method causes an *IllegalThreadStateException*.

Runnable

The *Thread* is in runnable state after invocation of *start()* method. A thread in the runnable state is ready for execution but is not being executed currently. Once a thread is in the runnable state, it gets all the resources of the system (as, for example, access to the CPU) and moves on to the running state.

Running

After the runnable state, if the thread gets CPU access, it moves into the running state. The thread will be in the running state unless one of the following things will occur:

- ✓ It dies (that is, the run () method exits).
- ✓ It gets blocked to the input/output for some reason.
- ✓ It calls sleep ().
- ✓ It calls wait ().
- ✓ It calls yield ().
- ✓ It is preempted by a thread of higher priority.
- ✓ Its quantum (time slice) expires.

Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run. A thread can enter the blocked state when one of the following five conditions occurs:

- ✓ When sleep () is called,
- ✓ When suspend () is called,
- ✓ When wait () is called,

- ✓ The thread calls an operation (I/O Resources),
- ✓ The thread is waiting for monitor.

Terminated

A thread is in terminated or dead state when its run() method exits or *stop ()* method is invoked. A thread in the dead state cannot be executed further.

4.4 Thread class

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Constructors of Thread class

1. Thread()
2. Thread(String name)
3. Thread(Runnable r)
4. Thread(Runnable r, String name)

Where *name* is thread name and *r* is instance of a class which implements Runnable interface.

Methods

These are all ***public*** methods of a Thread class

Method name	Description
void run()	Used to perform action for a thread.
void start()	Starts the execution of the thread.
void sleep(long milliseconds)	Causes the currently executing thread to sleep for the specified number of milliseconds.
void join()	Waits for a thread to die.
void join(long milliseconds)	Waits for a thread to die for the specified milliseconds.
int getPriority()	Returns the priority of the thread.
int setPriority(int priority)	Changes the priority of the thread.
String getName()	Returns the name of the thread.
void setName(String name)	Changes the name of the thread.
Thread currentThread()	Returns the reference of currently executing thread.
public int getId()	Returns the id of the thread.
Thread.State getState()	Returns the state of the thread.
boolean isAlive()	Tests if the thread is alive.
void yield()	Causes the currently executing thread object to temporarily pause and allow other threads to execute.
void suspend()	Used to suspend the thread (depricated).
void resume()	Used to resume the suspended thread (depricated).
void stop()	Used to stop the thread (depricated).
boolean isDaemon()	Tests if the thread is a daemon thread.
void setDaemon (boolean b)	Marks the thread as daemon or user thread.
void interrupt()	Interrupts the thread.
boolean isInterrupted()	Tests if the thread has been interrupted.
static boolean interrupted()	Tests if the current thread has been interrupted.

Runnable interface

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. public void run(): is used to perform action for a thread.

4.5 The Main Thread

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

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

If you want to return the reference of the *main thread*, invoke a method called *currentThread()* which returns the reference of thread in which it is called. Its general form is shown here:

static Thread currentThread( )

Example

```
class MainThread
{
        public static void main (String ar[] )
        {
                Thread t = Thread.currentThread();
                System.out.println("Current Thread: " + t);
                t.setName("Main Thread");
                System.out.println("After name change: " + t);
                try
                {
                        for(int i=1;i<=5;i++)
                        {
                                System.out.print(i + "\t");
                                Thread.sleep(500);
                        }
                }
                catch(InterruptedException ie)
                {
                        System.out.println(ie);
                }
        }
}
```

Output

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

4.6 How to create Thread

There are two ways to create a thread.

1. By extending Thread class
2. By implementing Runnable interface.

4.6.1 Implementing Runnable

The easiest way to create a thread is to create a class that implements the Runnable interface. After implementing runnable interface , the class needs to implement the run() method, which is of form,

public void run()

- ✓ The code that your thread will execute is placed inside *run()* method.
- ✓ *run()* method can call other methods.
- ✓ Thread will end when *run()* method terminates.

Example for creating thread by implementing Runnable interface

```
class MyThread implements Runnable
{
        String name;
        Thread t;
        MyThread(String n)
        {
                name = n;
                t = new Thread(this, name);
                t.start();
        }
        public void run()
        {
                System.out.println(name+ " thread is running...");
        }
        public static void main(String args[])
        {
                new MyThread("Child");
        }
}
```

Output

Child thread is running...

4.6.2 Extending Thread class

This is another way to create a thread by extending a Thread class. The extending class must override *run()* method which is the entry point of new thread.

Example for creating thread by extending Thread class

```
class MyThread extends Thread
{
        String name;
        MyThread(String n)
        {
                super(n);
                name = n;
        }
        public void run()
        {
```

```
		System.out.println(name+" thread is running...");
	}
	public static void main(String args[])
	{
		MyThread t=new MyThread("Child");
		t.start();
	}
}
```

Output

My First thread is running...

Creating Multiple Threads

Your program can spawn as many threads as it needs. For example, the following program creates three child threads:

Example program for creating multiple threads using Thread class

```
class MultiT  extends Thread
{
	String str;
	int time;
	MultiT(String s, int t)
	{
		str=s;
		time=t;
	}
	public void run()
	{
		for(;;)
		{
			System.out.println(str);
			try
			{
				Thread.sleep(time);
			}
			catch(InterruptedException e)
			{
				System.out.println("InterruptedException");
			}
		}
	}
	public static void main(String[] args)
	{
		System.out.println("Press Ctrl-C to stop.");
		MultiT t1= new MultiT("Good Morning",1000);
		MultiT t2= new MultiT("Hello",2000);
		MultiT t3= new MultiT("Welcome",3000);
```

```
                t1.start();
                t2.start();
                t3.start();
        }
}
```

Output

```
Press Ctrl-C to stop.
Welcome
Hello
Good Morning
Hello
Good Morning
Welcome
Good Morning
Hello
Good Morning
```

4.6.3 Example program for creating multiple threads by using Runnable interface

```
class MultiR  implements Runnable
{
        String str;
        int time;
        Thread t;
        MultiR(String s, int tm)
        {
                str=s;
                time=tm;
                t = new Thread(this);
                t.start();
        }
        public void run()
        {
                for(;;)
                {
                        System.out.println(str);
                        try
                        {
                                Thread.sleep(time);
                        }
                        catch(InterruptedException e)
                        {
                                System.out.println("InterruptedException");
                        }
                }
        }
```

```
        public static void main(String[] args)
        {
                System.out.println("Press Ctrl-C to stop.");
                new MultiR("Good Morning",1000);
                new MultiR("Hello",2000);
                new MultiR("Welcome",3000);
        }
}
```

Output

```
Welcome
Hello
Press Ctrl-C to stop.
Good Morning
Hello
Good Morning
Welcome
Good Morning
Hello
Good Morning
```

4.7 Using isAlive( ) and join( )

Sometimes one thread needs to know when other thread is terminating. In java, **isAlive()** and **join()**are two different methods that are used to check whether a thread has finished its execution or not.

isAlive()

The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise. This method is defined by **Thread**, and its general form is:

final boolean isAlive( )

join()

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. Its general form is:

public void join()throws InterruptedException

public void join(long milliseconds)throws InterruptedException

Example program on join() and isAlive() methods

```
class JoinAlive extends Thread
{
        String name;
        JoinAlive(String n)
        {
                super(n);
                name = n;
        }
        public void run()
        {
```

```
                System.out.println(name+" is running...");
                for(int i=1;i<=5;i++)
                {
                        try
                        {
                           Thread.sleep(500);
                        }
                        catch(Exception e)
                        {
                                System.out.println(e);
                        }
                        System.out.print(i+"  ");
                }
        }
        public static void main(String args[])
        {
                JoinAlive t1=new JoinAlive("T1");
                JoinAlive t2=new JoinAlive("T2");
                JoinAlive t3=new JoinAlive("T3");
                t1.start();
                try
                {
                        t1.join();
                }
                catch(InterruptedException e)
                {
                        System.out.println(e);
                }
                t2.start();
                t3.start();
                System.out.println("\nT1 is alive:"+t1.isAlive());
                System.out.println("T2 is alive:"+t2.isAlive());
                System.out.println("T3 is alive:"+t3.isAlive());
        }
}
```

Output

```
T1 is running...
1  2  3  4  5
T1 is alive: false
T3 is running...
T2 is alive: true
T2 is running...
T3 is alive: true
1  1  2  2  3  3  4  4  5  5
```

4.8 Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**. Its general form is:

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. Default priority of a thread is 5.

Thread class defined 3 levels:

public static int MIN_PRIORITY
public static int NORM_PRIORITY
public static int MAX_PRIORITY

The value of MIN_PRIORITY is 1, the value of MAX_PRIORITY is 10 and the value of NORM_PRIORITY is 5.

One can obtain the current priority of a thread by calling the *getPriority( )* method, shown here:

final int getPriority( )

Example of priority of a Thread

```
class TPriorities extends Thread {
        public void run(){}
        public static void main(String args[])
        {
                TPriorities t1=new TPriorities();
                TPriorities t2=new TPriorities();
                TPriorities t3=new TPriorities();
                t1.setPriority(Thread.MIN_PRIORITY);
                t2.setPriority(Thread.NORM_PRIORITY+1);
                t3.setPriority(Thread.MAX_PRIORITY);
                t1.start();
                t2.start();
                t3.start();
                System.out.println("T1 Priority : "+t1.getPriority());
                System.out.println("T2 Priority : "+t2.getPriority());
                System.out.println("T3 Priority : "+t3.getPriority());
        }
}
```

Output: T1 Priority : 1
T2 Priority : 6
T3 Priority : 10

4.9 Daemon Thread in java

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.

Daemon thread is a low priority thread (in context of JVM) that runs in background to perform tasks such as garbage collection (gc, finalizer) etc.,

You can see all the detail by typing the *jconsole* in the command prompt. The *jconsole* tool provides information about the loaded classes, memory usage, running threads etc.

Properties

- It is a low priority thread.
- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- JVM terminates daemon thread when all user threads finish their execution
- Daemon thread cannot prevent the JVM from exiting when all the user threads finish their execution.

Methods

void setDaemon (boolean status): This method is used to mark the current thread as daemon thread. Its general form is:

public final void setDaemon(boolean status)

if status is true, marks this thread as a daemon thread. If you call the setDaemon() method after starting the thread, it would throw **IllegalThreadStateException**.

boolean isDaemon(): This method is used to check that current is daemon. It returns true if the thread is Daemon else it returns false. Its general form is:

public final boolean isDaemon()

Java program to demonstrate the usage of setDaemon() and isDaemon() methods.

```
class DaemonThread extends Thread
{
   public void run()
   {
      // Checking whether the thread is Daemon or not
      if(Thread.currentThread().isDaemon())
            System.out.println("Daemon thread working...");
      else
            System.out.println("User thread is working...");
   }
   public static void main(String[] args)
   {

      DaemonThread t1 = new DaemonThread();
      DaemonThread t2 = new DaemonThread();
```

```
        t1.setDaemon(true);       // Setting user thread t1 to Daemon
        t1.start();
        t2.start();
    }
}
```

Output

Daemon thread working...
User thread working…

4.10 Synchronization

Synchronization is a process that involves coordinating the execution of multiple threads to ensure a desired outcome without corrupting the shared data and preventing any occurrence of deadlocks and race conditions.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

The synchronization is mainly used to:

- ✓ To prevent thread interference.
- ✓ To prevent consistency problem.

> **Note:** A *race condition* is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

Synchronization is built around an internal entity known as the ***lock*** or ***monitor***. A *monitor* is an object that is used as a mutually exclusive lock. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

Limitations

1. **Concurrency Limitations:** Java synchronization does not allow concurrent reads.
2. **Decreases Efficiency:** Java synchronized method run very slowly and can degrade the performance,

Types of Synchronization: There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Thread Synchronization

There are two types of thread synchronization

1. Mutual exclusive
2. Inter-thread communication.

4.10.1 Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by two ways in java:

1. *by synchronized method*
2. *by synchronized block*

synchronized method

The *synchronized* keyword in java creates a block of code referred to as critical section. If you declare any method as *synchronized*, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource.

```
synchronized public return-type method_name( parameter-list)
{
        //statement to be synchronized
}
```

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Example of java synchronized method

```
class Table
{
        synchronized void printTable(int n)   //synchronized method
        {
                for(int i=1;i<=10;i++)
                {
                        System.out.println(n+"x"+i+"="+(n*i));
                        try
                        {
                            Thread.sleep(400);
                      }catch(InterruptedException e)
                        {
                                System.out.println(e);
                        }
                }
        }
}
class MyThread1 extends Thread
{
        Table t;
        MyThread1(Table t)
        {
                this.t=t;
        }
        public void run()
        {
                t.printTable(2);
        }
}

class MyThread2 extends Thread
{
```

```
        Table t;
        MyThread2(Table t)
        {
                this.t=t;
        }
        public void run()
        {
                t.printTable(3);
        }
}

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

Output

```
2x1=2                    3x1=3
2x2=4                    3x2=6
2x3=6                    3x3=9
2x4=8                    3x4=12
2x5=10                   3x5=15
2x6=12                   3x6=18
2x7=14                   3x7=21
2x8=16                   3x8=24
2x9=18                   3x9=27
2x10=20                  3x10=30
```

synchronized block

Synchronized block can be used to perform synchronization on any specific resource of the method. Its general form is:

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

Here, *object* is a reference to the *object* being synchronized. A *synchronized block* ensures that a call to a synchronized method that is a member of *object's* class occurs only after the current thread has successfully entered *object's* monitor.

Suppose you have 100 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block. If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Example of java synchronized block

```
class Table
{
	void printTable(int n)
	{
		for(int i=1;i<=10;i++)
		{
			System.out.println(n+"x"+i+"="+(n*i));
			try
			{
			    Thread.sleep(400);
			}catch(InterruptedException e)
			{
				System.out.println(e);
			}
		}
	}
}

class MyThread1 extends Thread
{
	Table t;
	MyThread1(Table t)
	{
		this.t=t;
	}
	public void run()
	{
		synchronized(t)		//synchronized block
		{
			t.printTable(2);
		}
	}
}
class MyThread2 extends Thread
{
	Table t;
	MyThread2(Table t)
	{
		this.t=t;
	}
	public void run()
	{
		synchronized(t)		//synchronized block
		{
			t.printTable(4);
		}
	}
}
public class Sync {
	public static void main(String args[])
```

```
        {
                Table tbl = new Table();        //only one object
                MyThread1 t1=new MyThread1(tbl);
                MyThread2 t2=new MyThread2(tbl);
                t1.start();
                t2.start();
        }
}
```

Output

```
2x1=2                   4x1=4
2x2=4                   4x2=8
2x3=6                   4x3=12
2x4=8                   4x4=16
2x5=10                  4x5=20
2x6=12                  4x6=24
2x7=14                  4x7=28
2x8=16                  4x8=32
2x9=18                  4x9=36
2x10=20                 4x10=40
```

4.10.2 Inter-thread communication

It is all about allowing synchronized threads to communicate with each other. *Inter-thread communication* is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of **Object class**:

1. wait()
2. notify()
3. notifyAll()

wait() method

It tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls *notify( )* or *notifyAll( )*, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception. Its general form is:

public final void wait()throws InterruptedException
public final void wait(long timeout)throws InterruptedException

notify() method

It wakes up a thread that called wait( ) on the same object. Its general form is:

public final void notify( )

notifyAll( ) method

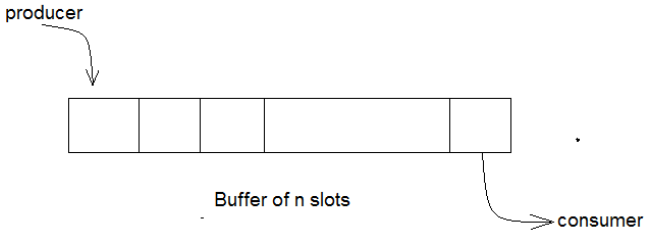
It wakes up all the threads that called *wait( )* on the same object. One of the threads will be granted access. Its general form is:

public final void notifyAll( )

Producer-Consumer solution using threads in Java

In computing, the producer–consumer problem (a.k.a bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data, one piece at a time.



Problem

To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Solution

The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again.

In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Source code for Producer – Consumer Problem

```
class Q
{
        int n;
        boolean valueSet = false;
        synchronized int get()
        {
                while(!valueSet)
                  try
                  {
                          wait();
                  }
                  catch(InterruptedException e){}
                  System.out.println("Got: "+n);
                  valueSet=false;
                  notify();
                  return n;
        }
        synchronized void put(int n)
        {
                while(valueSet)
                  try
                  {
                    wait();
                  }
                  catch(InterruptedException e){}
                  this.n = n;
                  valueSet=true;
```

```
                System.out.println("Put: "+n);
                notify();
        }
}

class Producer implements Runnable
{
        Q q;
        Thread t;
        Producer(Q q)
        {
                this.q = q;
                t= new Thread (this,"Producer");
                t.start();
        }
        public void run()
        {
                int i=0;
                while(true)
                {
                        q.put(i++);
                }
        }
}
class Consumer implements Runnable
{
        Q q;
        Thread t;
        Consumer(Q q)
        {
                this.q = q;
                t = new Thread(this,"Consumer");
                t.start();
        }
        public void run()
        {
                while(true)
                {
                        q.get();
                }
        }
}
class PCProblem
{
        public static void main(String[] args)
        {
                Q q  = new Q();
                new Producer(q);
                new Consumer(q);
                System.out.println("Press Ctrl - C to stop.");
        }
}
```

Output

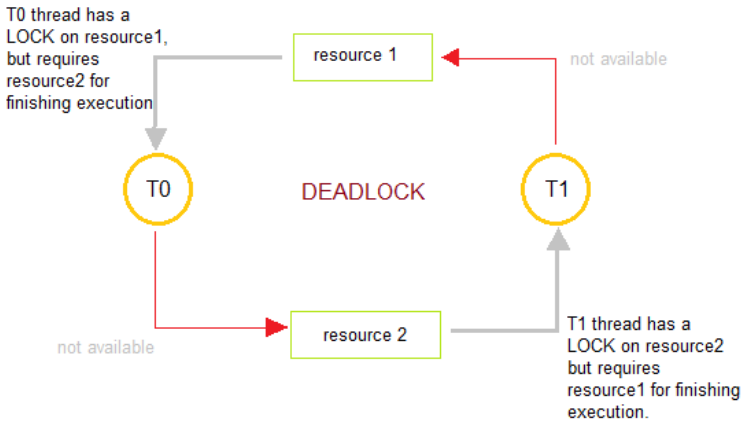
```
Put: 0
Press Ctrl - C to stop.
Got: 0
Put: 1
Got: 1
Put: 2
Got: 2
```

4.11 What is a Deadlock?

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

Consider an example, in operating systems when there are two or more processes hold some resources and wait for resources held by other(s).

For example, in the below diagram, Thread-0 is holding Resource 1 and waiting for resource 2 which is acquired by Thread-1, and Thread-1 is waiting for resource 1.



Deadlock can arise if following four conditions hold simultaneously.

1. *Mutual Exclusion:* Only one process can use at a time.
2. *Hold and Wait:* A process is holding at least one resource and waiting for resources.
3. *No Preemption:* A resource cannot be taken from a process unless the process releases the resource.
4. *Circular Wait:* A set of processes are waiting for each other in circular form.

Java program that illustrates deadlock

```
public class Deadlock
{
        public static void main(String a[])
        {
                String r1 = "resource1";
                String r2 = "resource2";
                Thread t1 = new Thread()
                {
                     public void run()
                    {
                            synchronized(r1)
                            {
                                    try
```

```
                                {
                                        Thread.sleep(50);
                                }catch(InterruptedException e){}
                                System.out.println("Thread one locked "+r1);
                                synchronized(r2)
                                {
                                    System.out.println("Thread one locked "+r2);
                                }
                            }
                        }
                    };
                    Thread t2 = new Thread()
                    {
                        public void run()
                        {
                            synchronized(r2)
                            {
                                try
                                {
                                    Thread.sleep(50);
                                 }catch(InterruptedException e){}
                                 System.out.println("Thread two locked "+r2);
                                 synchronized(r1)
                                {
                                    System.out.println("Thread two locked "+r1);
                                }
                            }
                        }
                    };
                    t1.start();
                    t2.start();
                }
            }
```

Output

Thread one locked resource1
Thread two locked resource2

We can also detect deadlock by running this program on *cmd*. We have to collect Thread Dump. If we are using Windows and Java 8, command is *jcmd $PID Thread.print.* We can get PID by running *jps* command. Thread dump for above program is below:

JAVA I/O

Introduction

- ✓ Java I/O (Input and Output) is used *to process the input* and *produce the output*.
- ✓ Java uses the concept of *stream* to make I/O operations fast.
- ✓ The *java.io* package contains all the classes required for input and output operations.
- ✓ We can perform file handling in java by Java I/O API.

4.12 What is a stream?

A stream is a sequence of data. In Java a stream is composed of bytes. In Java, three streams are created for us automatically. All these streams are attached with console. They are:

1. System.out: **standard output stream**
2. System.in: **standard input stream**
3. System.err: **standard error stream**

Types of Streams

In general, streams are classified into two types known as *character streams* and the *byte streams*. The *java.io* package provides two sets of class hierarchies to handle character and byte streams for reading and writing:

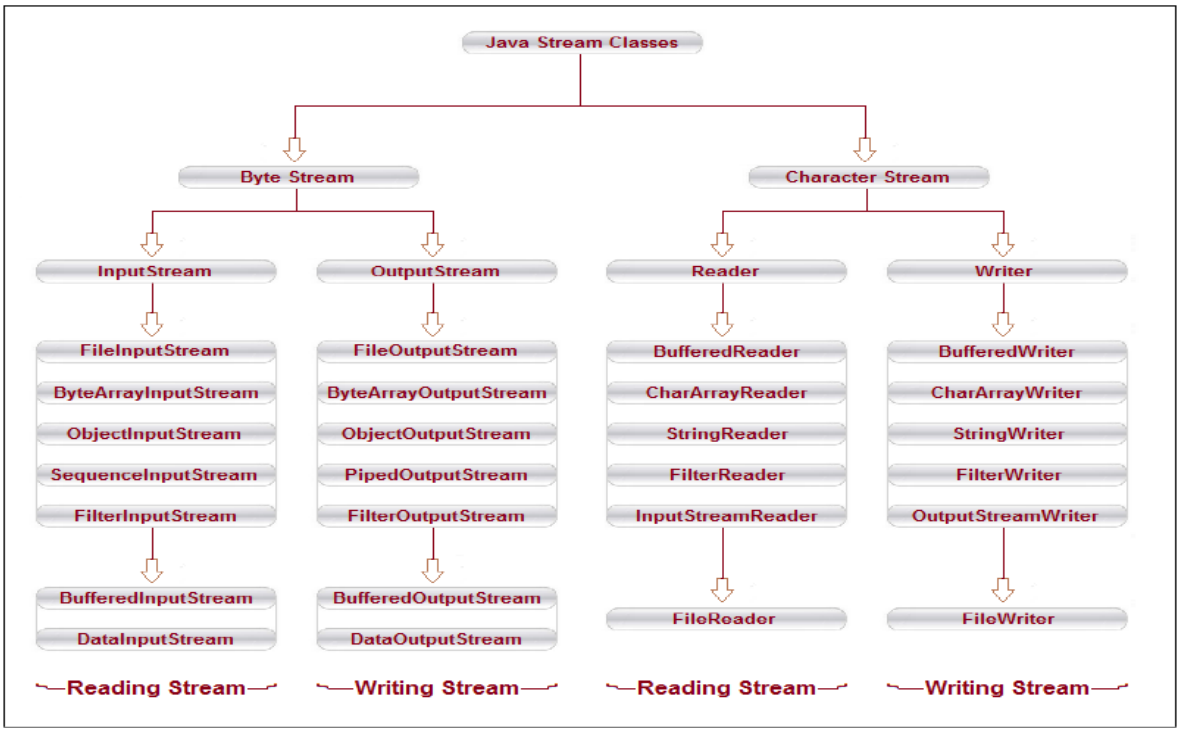
1. *Input Stream and Output Stream* classes are operated on bytes for reading and writing, respectively.
2. *Reader and Writer* classes are operated on *characters* for reading and writing, respectively.
3. There are two other classes that are useful for handling input and output. These are *File* class and *Random Access File* class.

Description of four classes in the package *java.io*

Name of class	Description
Reader, Writer	Supports read/write 16-bit Unicode character. Used for only character data
InputStream, OutputStream	Define basic methods for input and output streams of data. These classes are used only for binary data.
File	Enables creating, deleting and renaming files, navigating through the file system, testing file existence and finding information about files.
Random Access File	Enables the program to read/write from/to any location in the file, not just the beginning/end of the file, is the case as in the usual sequential access. The file works as a random-access disk file.

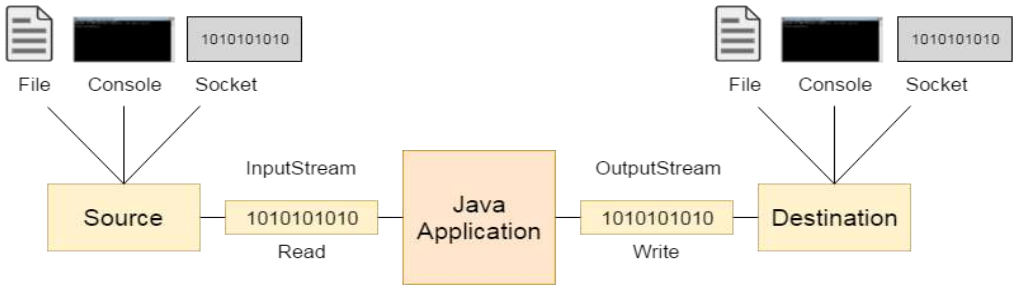
The *java.io* package contains a large number of stream classes that provide capabilities for processing all types of data. These classes may be categorized into two groups based on the data type on which they operate.

1. Byte stream classes
2. Character stream classes



OutputStream

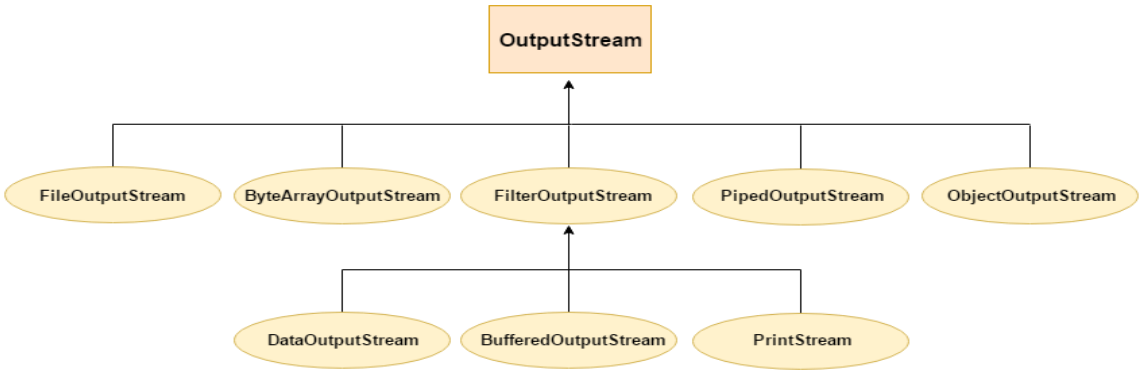
- Java application uses an output stream to write data to a destination, it may be a *file*, an *array*, *peripheral device* or *socket*.
- *OutputStream* class is an abstract class. It is the super class of all classes representing an output stream of bytes.



Methods

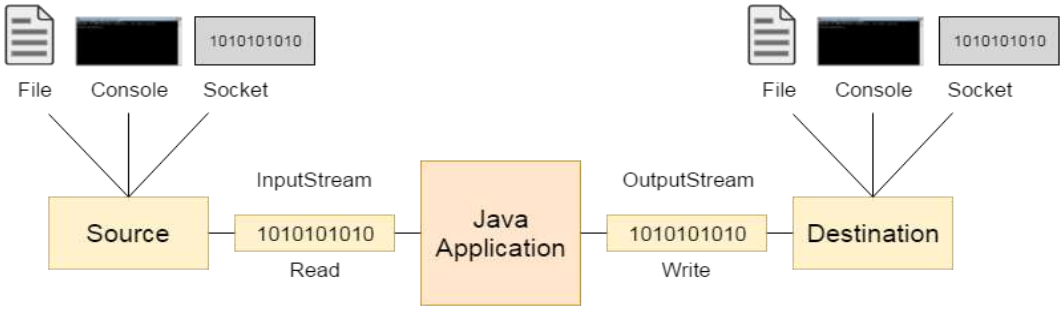
Method	Description
public void write(int) throws IOException	It is used to write a byte to the current output stream.
public void write(byte[]) throws IOException	It is used to write an array of byte to the current output stream.
public void flush() throws IOException	It flushes the current output stream.
public void close() throws IOException	It is used to close the current output stream.

OutputStream Hierarchy



InputStream

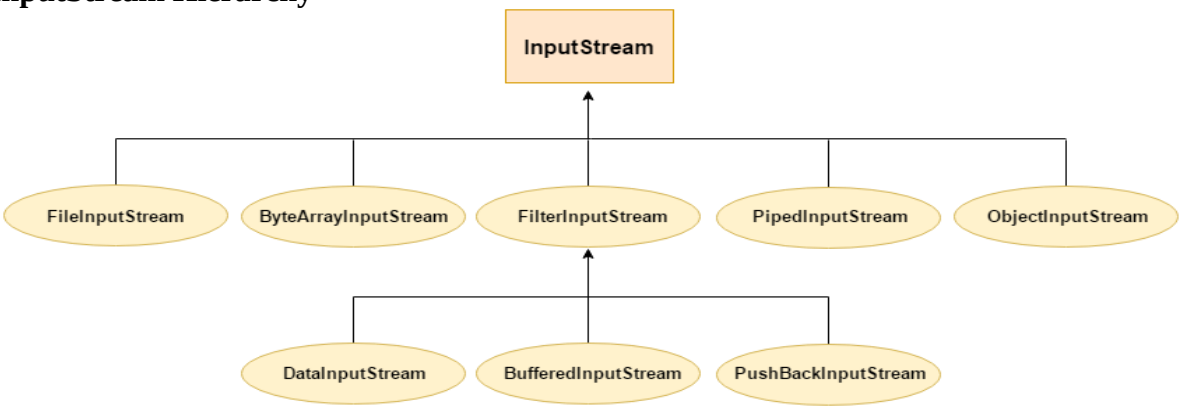
Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket. *InputStream* class is an abstract class. It is the super class of all classes representing an input stream of bytes.



Methods

Method	Description
public abstract int read() throws IOException	It reads the next byte of data from the input stream. It returns -1 at the end of file.
public int available() throws IOException	It returns an estimate of the number of bytes that can be read from the current input stream.
public void close() throws IOException	It is used to close the current input stream.

InputStream Hierarchy



4.13 FileOutputStream Class

- ✓ Java *FileOutputStream* is an output stream used for writing data to a file.
- ✓ If you have to write primitive values into a file, use FileOutputStream class.
- ✓ You can write byte-oriented as well as character-oriented data through FileOutputStream class.

Declaration

public class FileOutputStream extends OutputStream

Methods

Method	Description
protected void finalize()	It is sued to clean up the connection with the file output stream.
void write(byte[] ary)	It is used to write ary.length bytes from the byte array to the file output stream.
void write(byte[] array, int off, int length)	It is used to write length bytes from the byte array starting at offset off to the file output stream.
void write(int b)	It is used to write the specified byte to the file output stream.
void close()	It is used to closes the file output stream.

Java program that illustrates writing data to the file

```
import java.io.*;
public class FileOutputSEx {
   public static void main(String args[]){
        try
        {
             FileOutputStream fout=new FileOutputStream("test.txt");
             String s="Welcome to Files.";
             byte b[]=s.getBytes(); //converting string into byte array
             fout.write(b);
             fout.close();
        }
        catch(Exception e)
        {
                        System.out.println(e);
         }
    }
}
```

FileInputStream Class

- ✓ Java *FileIntputStream* is an input stream used for reading data from a file.
- ✓ You can read byte-oriented as well as character-oriented data through FileInputStream class.

Reading data from file

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

4.14 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.
- ✓ For adding the buffer in an OutputStream, use the BufferedOutputStream class.

Declration

public class BufferedOutputStream extends FilterOutputStream

Constructors

Constructor	Description
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.

Methods

Method	Description
void write(int b)	It writes the specified byte to the buffered output stream.
void write(byte[] b, int off, int len)	It write the bytes from the specified byte-input stream into a specified byte array, starting with the given offset
void flush()	It flushes the buffered output stream.

Java program that illustrates writing data to the file

```
import java.io.*;
public class BufferedOSEx
{
        public static void main(String args[]) throws Exception
        {
                FileOutputStream fout=new FileOutputStream("test.txt");
                BufferedOutputStream bout=new BufferedOutputStream(fout);
                String s="Welcome to VVIT.";
                byte b[]=s.getBytes();
                bout.write(b);
                bout.flush();
                bout.close();
```

```
                fout.close();
        }
}
```

BufferedInputStream Class

- ✓ Java BufferedInputStream class is used to read information from stream.
- ✓ It internally uses buffer mechanism to make the performance fast.

Declaration

public class BufferedInputStream extends FilterInputStream

Constructors

Constructor	Description
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.

Methods

Method	Description
int available()	It returns an estimate number of bytes that can be read from the input stream without blocking by the next invocation method for the input stream.
int read()	It read the next byte of data from the input stream.
int read(byte[] b, int off, int ln)	It read the bytes from the specified byte-input stream into a specified byte array, starting with the given offset.
void close()	It closes the input stream and releases any of the system resources associated with the stream.
void reset()	It repositions the stream at a position the mark method was last called on this input stream.
void mark(int readlimit)	It sees the general contract of the mark method for the input stream.
long skip(long x)	It skips over and discards x bytes of data from the input stream.
boolean markSupported()	It tests for the input stream to support the mark and reset methods.

Java program that illustrates reading data from file.

```
import java.io.*;
public class BufferedISEx
{
        public static void main(String args[])
        {
                try {
                        FileInputStream fin=new FileInputStream("test.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 VVIT.

4.15 Writer

- ✓ It is an abstract class for writing to character streams.
- ✓ The methods that a subclass must implement are *write(char[], int, int), flush(), and close()*.
- ✓ Most subclasses will override some of the methods defined here to provide higher efficiency, functionality or both.

Constructor

Modifier	Constructor	Description
protected	Writer()	It creates a new character-stream writer whose critical sections will synchronize on the writer itself.
protected	Writer(Object lock)	It creates a new character-stream writer whose critical sections will synchronize on the given object.

Methods

Method	Description
Writer append(char c)	It appends the specified character to this writer.
Writer append(CharSequence csq)	It appends the specified character sequence to this writer
Writer append(CharSequence csq, int start, int end)	It appends a subsequence of the specified character sequence to this writer.
abstract void close()	It closes the stream, flushing it first.
abstract void flush()	It flushes the stream.
void write(char[] cbuf)	It writes an array of characters.
abstract void write(char[] cbuf, int off, int len)	It writes a portion of an array of characters.
void write(int c)	It writes a single character.
void write(String str)	It writes a string.
void write(String str, int off, int len)	It writes a portion of a string.

Java program that illustrates writing data to the file

```
import java.io.*;
public class WriterEx
{
   public static void main(String[] args)
   {
      try {
         Writer w = new FileWriter("test.txt");
         String str = "I am studying b.tech";
         w.write(str);
         w.close();
        }
         catch (IOException e)
```

```
        {
                e.printStackTrace();
        }
    }
}
```

Output.txt contains:

I am studying b.tech

Java Reader

- ✓ Java Reader is an abstract class for reading character streams.
- ✓ The only methods that a subclass must implement are read(char[], int, int) and close().
- ✓ Most subclasses, however, will override some of the methods to provide higher efficiency, additional functionality, or both.
- ✓ Some of the implementation classes are BufferedReader, CharArrayReader, FilterReader, InputStreamReader, PipedReader, StringReader

Fields

Modifier and Type	Field	Description
protected Object	lock	The object used to synchronize operations on this stream.

Constructor

Constructor	Description
protected Reader()	It creates a new character-stream reader whose critical sections will synchronize on the reader itself.
protected Reader(Object lock)	It creates a new character-stream reader whose critical sections will synchronize on the given object.

Methods

Method	Description
abstract void close()	It closes the stream and releases any system resources associated with it.
void mark(int readAheadLimit)	It marks the present position in the stream.
boolean markSupported()	It tells whether this stream supports the mark() operation.
int read()	It reads a single character.
int read(char[] cbuf)	It reads characters into an array.
abstract int read(char[] cbuf, int off, int len)	It reads characters into a portion of an array.
int read(CharBuffer target)	It attempts to read characters into the specified character buffer.
boolean ready()	It tells whether this stream is ready to be read.
void reset()	It resets the stream.
long skip(long n)	It skips characters.

Java program that illustrates writing data to the file

```
import java.io.*;
public class ReaderEx
{
        public static void main(String[] args)
        {
                try {
```

```
                        Reader reader = new FileReader("Output.txt");
                        int data = reader.read();
                        while (data != -1)
                        {
                                System.out.print((char) data);
                                data = reader.read();
                        }
                        reader.close();
                } catch (Exception ex)
                {
                        System.out.println(ex.getMessage());
                }
        }
}
```

Output

I am studying b.tech

4.16 FileWriter Class

- ✓ Java FileWriter class is used to write character-oriented data to a file.
- ✓ It is character-oriented class which is used for file handling in java.
- ✓ Unlike FileOutputStream class, you don't need to convert string into byte array because it provides method to write string directly.

Declaration

public class FileWriter extends OutputStreamWriter

Constructors

Constructor	Description
FileWriter(String file)	Creates a new file. It gets file name in string.
FileWriter(File file)	Creates a new file. It gets file name in File object.

Methods

Method	Description
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.

Java program that illustrates writing data to the file

```
import java.io.*;
public class FileWEx
{
        public static void main(String args[])
        {
                try
                {
                        FileWriter fw=new FileWriter("test.txt");
                        fw.write("Welcome to VVIT.");
                        fw.close();
                }
                catch(Exception e)
                {
```

```
                    System.out.println(e);
                }
        }
}
```

FileReader Class

- ✓ Java *FileReader* class is used to read data from the file.
- ✓ It returns data in byte format like FileInputStream class.
- ✓ It is character-oriented class which is used for file handling in java.

Declaration

public class FileReader extends InputStreamReader

Constructors

Constructor	Description
FileReader(String file)	It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws *FileNotFoundException*.
FileReader(File file)	It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws *FileNotFoundException.*

Methods

Method	Description
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.

Java program that illustrates reading data from the file

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

Output

Welcome to VVIT.

4.17 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.

Declaration

public class BufferedWriter extends Writer

Constructors

Constructor	Description
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

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

Java program that illustrates writing data to the file

```
import java.io.*;
public class BufferWEx
{
        public static void main(String[] args) throws Exception
        {
           FileWriter writer = new FileWriter("test.txt");
           BufferedWriter bw = new BufferedWriter(writer);
           bw.write("Welcome to VVIT.");
           bw.close();
        }
}
```

BufferedReader Class

- ✓ 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.

Declaration

public class BufferedReader extends Reader

Constructors

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

Methods

Method	Description
int read()	It is used for reading a single character.
int read(char[] cbuf, int off,	It is used for reading characters into a portion of an array.

int len)	
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.

Java program that illustrates reading data from the file

```
import java.io.*;
public class BufferedREx
{
   public static void main(String args[])throws Exception
   {
        FileReader fr=new FileReader("test.txt");
        BufferedReader br=new BufferedReader(fr);
        int i;
        while((i=br.read())!=-1)
           {
         System.out.print((char)i);
           }
        br.close();
        fr.close();
   }
}
```

Output

Welcome to VVIT.

Java program that illustrates reading data from console by InputStreamReader and BufferedReader

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

Output

Enter your name: Vijay

Welcome Vijay

4.18 PrintWriter class

Java PrintWriter class is the implementation of Writer class. It is used to print the formatted representation of objects to the text-output stream.

Declaration

public class PrintWriter extends Writer

Methods

Method	Description
void println(boolean x)	It is used to print the boolean value.
void println(char[] x)	It is used to print an array of characters.
void println(int x)	It is used to print an integer.
PrintWriter append(char c)	It is used to append the specified character to the writer.
PrintWriter append(CharSequence ch)	It is used to append the specified character sequence to the writer.
PrintWriter append(CharSequence ch, int start, int end)	It is used to append a subsequence of specified character to the writer.
boolean checkError()	It is used to flushes the stream and check its error state.
protected void setError()	It is used to indicate that an error occurs.
protected void clearError()	It is used to clear the error state of a stream.
PrintWriter format(String format, Object... args)	It is used to write a formatted string to the writer using specified arguments and format string.
void print(Object obj)	It is used to print an object.
void flush()	It is used to flushes the stream.
void close()	It is used to close the stream.

Java program that illustrates write data to console and file by PrintWriter class

```
import java.io.*;
public class PrintWEx
{
   public static void main(String[] args) throws Exception
   {
        //write data to Console using PrintWriter
        PrintWriter pw = new PrintWriter(System.out);
        pw.write("Welcome to VVIT.");
        pw.flush();
        pw.close();
        //Write data to File using PrintWriter
        PrintWriter pw1 =null;
        pw1 = new PrintWriter(new File("test.txt"));
        pw1.write("We provide better education.");
        pw1.flush();
        pw1.close();
   }
}
```

Output: Welcome to VVIT.

Test.txt

We provide world class education.

4.19 Random Access File

Java also allows you to access the contents of a file in random order i.e. data items can be read and written in any order. This is especially useful in direct access applications such as banking systems, airline reservation systems, Automatic Teller Machine (ATM) etc. where the desired information must be located immediately.

Random access files (or direct access files) are analogous to arrays, where each element is accessed directly by means of its index number. Java provides *java.io.RandomAccessFile* class that enables you to perform random access file input and output operations.

When a data file is opened for random read and write access, an internal file pointer is set at the beginning of the file. When you read or write data to the file, the file pointer moves forward to the next data item.

For example, when reading an int value using readInt() , 4 bytes are read from the file and the file pointer moves 4 bytes ahead from the previous file pointer position. Similarly, when reading a double value using readDouble () , 8 byte are read from the file pointer and the file pointer moves 8 bytes ahead from the previous file pointer position.

```
import java.io.*;
class RFileEx
{
    public static void main(String[] args)
   {
        try
        {

           RandomAccessFile file = new RandomAccessFile("f.txt","rw");
           file.setLength(0);
           for(int i=0;i<50;i++)
               file.writeInt(i);
               System.out.println("Length of File After Writing Data is : "+file.length());
               file.seek(0);
               System.out.println("First Number is : "+file.readInt());
               file.seek(1*4);
               System.out.println("Second Number is : "+file.readInt());
               file.writeInt(101);
               file.seek(file.length());
               file.writeInt(50);
               System.out.println("Current Length of File is : "+file.length());
        }
               catch(Exception e) {  e.printStackTrace();  }
   }
}
```

Output:

```
Length of File After Writing Data is : 200
First Number is : 0
Second Number is : 1
Current Length of File is : 204
```