### Topics:

**Unit 4Multithreading**

**MultiThreading**: java.lang.Thread, the main thread, Creation of new thread, Thread priority, Multithreading, Synchronization, Suspending and resuming threads, Communication between threads, Input/Output: reading and writing data, java.io.package.

## Introduction

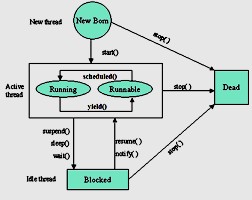
Unlike many other programming languages, java supports multithreaded programming. The multithreaded program contains two or more ***parts*** that can run concurrently. These parts are called “**Threads**”. Each Thread defines separate path of execution. Thus, Multithreading is specialized form of **Multitasking**.

However, there are two distinct types of multitasking: **process based and thread-based**. It is important to understand the difference between the two. For most readers, process-based multitasking is the more familiar form. A ***process*** is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In process based multitasking, a program is the smallest unit of code that can be dispatched by the ***scheduler***.

In a *thread-based* multitasking environment, ***the thread is the smallest unit of dispatchable code***. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the **“big picture,”** and thread-based multitasking handles the **details**. Multitasking threads require less overhead than multitasking processes. Processes are heavy weight tasks that require their own separate address space.

### Difference between Multiprocessing and Multithreading

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

**Life Cycle of a Thread**

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

* Newborn state
* Runnable State
* Running State
* Blocked state
* Dead state

#### Fig 1: Life Cycle of Thread

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

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

* + Schedule it for running using the start() method.
  + Kill it using stop() method.

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

**Running state**: Running state means that the processor has given its time to the thread for it execution. The thread runs until it ***relinquishes*** the control or it is preempted by the other higher priority thread. As shown in the Fig 1. a running thread can be preempted using the suspen(), or wait(), or sleep() methods.

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

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

#### The methods of the Thread class are as follow:

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 miliseconds):** 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 miliseconds):** waits for a thread to die for the specified miliseconds.
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(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
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():** tests if the current thread has been interrupted.

## Multithreading in Java

Every program that we have been writing has at least one thread, that is, the "**main**" thread. Whenever a program starts executing, the JVM is responsible for creating the main thread and calling "main()" method. Along with this main thread, some other threads are also running to carryout the tasks such as "**finalization**" and "**garbage collection**". The thread can either die naturally or be forced to die.

* Thread dies naturally when it exits from the "run()" method.
* Thread can be forced to die by calling "interrupt()" method.

#### java.lang.Thread package

Creation of Thread in java is very simple task. There is a class called "Thread", which belongs to the "java.lang.Thread" package. This package contains one interface also called "**Runnable**". Both these contain a common method called "**run()**" which is the heart of the thread. The run() method would have the following syntax:

#### Syntax:

**public void run()**

**{**

**//statement for implementing the thread.**

**}**

**Thread Constructors:**

* Thread ()-without arguments, default constructor
* Thread(String str)- Thread contains name given as argument

## The Main Thread

Every java program has a thread called "main" thread. When the program execution starts, the JVM creates "main" Thread and calls the "main()" method from within that thread. Along with this JVM also creates other threads for the purpose of the Housekeeping task such as "garbage" collection. The "main" thread Spawns the other Threads. These spawned threads are called "Child Threads". The main thread is always is the last thread to finish execution. We, as Programmer can also take control of the main thread, using the method "**currentThread()**". The main thread can be controlled by this method. We can also change the name of the Thread using the method "**setName(String name)**".

#### Example Program:

class MainThread

{

#### MainThread.java

public static void main(String args[])

{

Thread t=Thread.currentThread(); System.out.println("Name of the Thread is:"+t); t.setName("KSR");

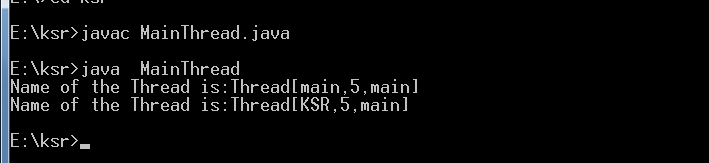
System.out.println("Name of the Thread is:"+t);

}

}

**Note: If we want name of the Thread alone, we can write t.getName().**

#### Output:



**Creation of Threads**

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

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

#### Creating Thread using The Thread Class

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

1. Declare the class as extending the Thread class.
2. Override the "**run()**" method that is responsible for running the thread.
3. Create a thread and call the "**start()**" method to instantiate the Thread Execution.

#### Declaring the class

TheThread class can be declared as follows: class MyThread extends Thread

{

}

#### Overriding the method run()

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

{

}

#### Starting the new Thread

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

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

**Example program:**

import java.io.\*; import java.lang.\*;

**ThreadTest.java**

class A extends Thread

{

public void run()

{

for(int i=1;i<=5;i++)

{

System.out.println("From Thread A :i="+i);

}

System.out.println("Exit from Thread A");

}

}

class B extends Thread

{

public void run()

{

for(int j=1;j<=5;j++)

{

System.out.println("From Thread B :j="+j);

}

System.out.println("Exit from Thread B");

}

}

class C extends Thread

{

public void run()

{

for(int k=1;k<=5;k++)

{

System.out.println("From Thread C :k="+k);

}

System.out.println("Exit from Thread C");

}

}

class ThreadTest

{

public static void main(String args[])

{

System.out.println("main thread started"); A a=new A();

a.start();

B b=new B(); b.start();

C c=new C(); c.start();

System.out.println("main thread ended");

}

}

**output: First Run**

|  |  |
| --- | --- |
| **First Run** | **Second Run** |
|  |  |

**Note:** If you observe here, first and second Runs produce different Results. This depends on the Processor availability, speed of the processor, Scheduling algorithms and priority of the threads. If you run this in two different systems they produce different outputs.

#### Creating the Thread using Runnable Interface

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

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

**Example program:**

**Runnable.java**

|  |  |
| --- | --- |
| class A implements Runnable  {  public void run()  {  for(int i=1;i<=5;i++)  {  System.out.println("A's i="+i);  }  }  }  class B implements Runnable  {  public void run()  {  for(int i=1;i<=5;i++)  {  System.out.println("B's i="+i);  }  }  }  class RunnableTest  { public static void main(String args[])  {  A a=new A(); B b=new B();  Thread t1=new Thread(a); //providing runnable object as argument Thread t2=new Thread(b);  t1.start();  t2.start();  }  **}** | |
| **First Run** | **Second Run** |
|  |  |

**Output: Threads A and B execution by running the above program two times. (You may see a different sequence of Output, every time you run this program)**

**Which way of creating Thread is better?**

**Answer:** We may get one question, that is which way of creating thread is better. The Answer is, if you want use others methods of the thread, then use Thread super class. If you are interested in only run() method use Runnable interface.

#### Advantage of the Multithreading

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

**When a Thread is ended: *isAlive() and join()* methods**

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

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

**final boolean isAlive();**

This method returns the either "**TRUE**" or "**FALSE**" . It returns "TRUE" if the thread is alive, returns "FALSE" otherwise.

#### General form of run() method

**public void join();**

While **isAlive( )** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join( ).** This method forces the currently running thread to finish it task first.

This method also has another version as follow:

#### Public void run(milliseconds);

Here, time milliseconds are used to specify the deadline within which the thread has to finish its task, context-switching is done to transfer control otherwise.

#### Example program: displaying two tables ( 2 and 3 table)



|  |  |
| --- | --- |
| **Without join() method** | **With join() method** |
| **class A implements Runnable**  **{**  **public void run()**  **{**  **for(int i=1;i<=10;i++)**  **{**  **System.out.println("2\*"+i+"="+(2\*i));**  **}**  **}**  **}**  **class B implements Runnable**  **{**  **public void run()**  **{**  **for(int j=1;j<=10;j++)**  **{**  **System.out.println("3\*"+j+"="+(3\*j));**  **}**  **}**  **}**  **class JoinTest**  **{**  **public static void main(String args[])**  **{**  **A a=new A(); B b=new B();**  **Thread t1=new Thread(a);**  **//providing runnable object as argument**  **Thread t2=new Thread(b); t1.start();**  **t2.start();**  **}**  **}** | **class A implements Runnable**  **{**  **public void run()**  **{**  **for(int i=1;i<=10;i++)**  **{**  **System.out.println("2\*"+i+"="+(2\*i));**  **}**  **}**  **}**  **class B implements Runnable**  **{**  **public void run()**  **{**  **for(int j=1;j<=10;j++)**  **{**  **System.out.println("3\*"+j+"="+(3\*j));**  **}**  **}**  **}**  **class JoinTest**  **{**  **public static void main(String args[])**  **{**  **A a=new A(); B b=new B();**  **Thread t1=new Thread(a); //providing runnable object as argument**  **Thread t2=new Thread(b);**  **Thread t3=new Thread(); //new thread t3 is created, but note starte**  **//calling isAlive() method System.out.println("Thread t3 is live :"+t3.isAlive()); t1.start();**  **//calling join() method try**  **{**  **t1.join();**  **}**  **catch(InterruptedException ie)**  **{**  **}**  **t2.start();**  **System.out.println("Thread t2 is live :"+t2.isAlive());**  **}**  **}** |
| **Without join() output** | **With join() output** |

|  |  |
| --- | --- |
|  |  |

**Daemon Thread**

Daemon thread is a low priority thread that runs in background to perform tasks such as garbage collection.

#### Properties:

* They cannot prevent the JVM from exiting when all the user threads finish their execution.
* JVM terminates itself when all user threads finish their execution
* If JVM finds running daemon thread, it terminates the thread and after that shutdown itself. JVM does not care whether Daemon thread is running or not.
* It is an utmost low priority thread.
* ***When is a user thread is finished, the JVM automatically terminates the Daemon threads.***

#### Methods:

1. **void setDaemon(boolean status):** This method is used to mark the current thread as daemon thread or user thread. For example if I have a user thread tU then tU.setDaemon(true) would make it Daemon thread. On the other hand if I have a Daemon thread tD then by calling tD.setDaemon(false) would make it user thread.

#### Syntax:

**public final void setDaemon(boolean on)**

#### boolean isDaemon():

This method is used to check that current is daemon. It returns true if the thread is Daemon else it returns false.

#### Syntax:

**public final boolean isDaemon() returns: true or false**

#### DTest.java

|  |  |
| --- | --- |
| **class DT extends Thread** | |
| **{** | |
| **public void run()** | |
| **{** | |
| **for(int i=1;i<=1000;i++)** | |
| **System.out.println("Daemon i="+i);** | |
| **}** | |
| **}** | |
| **class UD extends Thread** | |
| **{** | |
| **public void run()**  **{**  **for (int j=1;j<=5;j++) System.out.println("User j="+j);**  **}**  **}**  **class DTest**  **{**  **public static void main(String args[])**  **{**  **DT d=new DT(); d.setDaemon(true); UD u=new UD(); d.start();**  **u.start();** | |
| **}** | **}** |

**Output:**

**The Thread Priorities**

Thread priorities are used by the thread ***scheduler*** to decide when and which thread should be allowed to run. In theory, **higher-priority** threads get more CPU time than **lower- priority** threads. In practice, the amount of CPU time that a thread gets often depends on **several factors** besides its priority. (For example, how an operating system implements ***multitasking*** can affect the relative availability of CPU time.) A higher-priority thread can also **preempts** (stops) a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread **resumes** (from sleeping or waiting on I/O, for example), it will preempt the lower priority thread.

To set a thread’s priority, use the **setPriority( )** method, which is a member of **Thread**.

This is its general form:

final void setPriority(int *level*)

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must bewithin the range **MIN\_PRIORITY** and **MAX\_PRIORITY**. Currently, these values are 1 and10, respectively. To return a thread to default priority, specify **NORM\_PRIORITY**, which iscurrently 5. These priorities are defined as **static final** variables within **Thread**.

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

#### final int getPriority( )

**Example Program: PTest.java**

class PTest

{

public static void main(String args[])

{

class PThread1 extends Thread

{

public void run()

{

System.out.println(" Child 1 is started");

}

}

class PThread2 extends Thread

{

public void run()

{

System.out.println(" Child 2 is started");

}

}

class PThread3 extends Thread

{

public void run()

{

System.out.println(" Child 3 is started");

}

}

**//setting the priorities to the thread using the setPriority() method**

PThread1 pt1=new PThread1(); pt1.setPriority(1);

PThread2 pt2=new PThread2(); pt2.setPriority(9);

PThread3 pt3=new PThread3(); pt3.setPriority(6);

pt1.start();

pt2.start();

pt3.start();

**//getting the priority**

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

}

}

**Note:** Of course, the exact output produced by this program depends on the speed of your CPU and the number of other tasks running in the system. When this same program is run under a non preemptive system, different results will be obtained.

**Synchronization**

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called ***synchronization****.*

Key to synchronization is the concept of the **monitor** (also called a ***semaphore***). A *monitor* is an object that is used as a mutually exclusive lock, or ***mutex****.* Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have ***entered*** the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread ***exits*** the monitor. These other threads are said to be ***waiting*** for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

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

#### Example without the synchronization

class Table

{

void printTable(int n)

{

//method not synchronized for(int i=1;i<=5;i++)

{

try

{

}

System.out.println(n+"\*"+i+"="+(n\*i));

Thread.sleep(400);

catch(InterruptedException ie)

{

System.out.println("The Exception is :"+ie);

}

}

//end of the printTable() method

}

}

class MyThread1 extends Thread

{

Table t; MyThread1(Table mt)

{

t=mt;

}

public void run()

{

t.printTable(5);

}

} //end of the Thread1

class MyThread2 extends Thread

{

Table t; MyThread2(Table mt)

{

t=mt;

}

public void run()

{

t.printTable(100);

}

} //end of Thread2

class TestSyn

{

public static void main(String args[])

{

Table tab = new Table();//only one object MyThread1 t1=new MyThread1(tab); MyThread2 t2=new MyThread2(tab); t1.start();

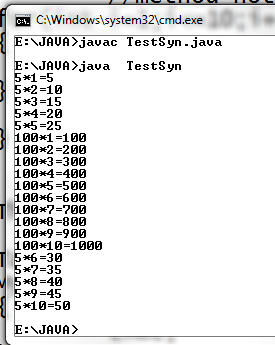
t2.start();

}

}

**The output for the above program will be as follow:**

In the above output, it can be observed that both the threads are simultaneously accessing the Table object to print the table.



**Using the Java synchronized method**

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

**The general form of the synchronized method is: synchronized** type method\_name(para\_list)

{

//body of the method

}

where synchronized is the keyword, method contains the type, and method\_name represents the name of the method, and para\_list indicate the list of the parameters.

**Example using the synchronized method**

class Table

{

synchronized void printTable(int n)

{

//method not synchronized for(int i=1;i<=10;i++)

{

System.out.println(n+"\*"+i+"="+(n\*i));

}

//end of the printTable() method

}

}

class MyThread1 extends Thread

{ Table t; MyThread1(Table mt)

{

t=mt;

}

public void run()

{

t.printTable(5);

}

} //end of the Thread1

class MyThread2 extends Thread

{ Table t; MyThread2(Table mt)

{

t=mt;

}

public void run()

{

t.printTable(100);

}

} //end of Thread2 class TestSyn

{

public static void main(String args[])

{

Table tab = new Table();//only one object

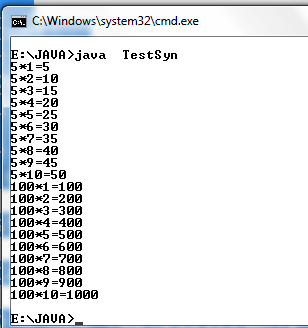
MyThread1 t1=new MyThread1(tab); MyThread2 t2=new MyThread2(tab); t1.start();

t2.start();

}

}

#### Output:



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

printTable() method.

1. This way of communications between the threads competing for same resource is called **implicit communication.**
2. This has one disadvantage due to polling. The polling wastes the CPU time. To save the CPU time, it is preferred to go to the **inter-thread communication.**

**Note:**

**Inter-Thread Communication**

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

Java addresses this polling problem, using via **wait()**, **notify()**, and **notifyAll()** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context.

* **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.
* **notify( )** wakes up a thread that called **wait( )** on the same object.
* **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. One of the threads will be granted access.

#### These methods are declared within Object, as shown here:

final void wait( ) throws InterruptedException final void notify( )

final void notifyAll( )

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

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

#### Example program for producer and consumer problem: class Q

{

int n;

boolean valueSet = false;***//flag***

#### synchronized int get()

{

while(!valueSet) try {

wait();

}

catch(InterruptedException e)

{

System.out.println("InterruptedException caught");

}

System.out.println("Got: " + n); valueSet = false;

notify(); return n;

#### } //end of the get() method synchronized void put(int n)

{

while(valueSet) try {

wait();

}

catch(InterruptedException e)

{

System.out.println("InterruptedException caught");

}

this.n = n; valueSet = true;

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

#### }//end of the put method

} **//end of the class Q**

class Producer implements Runnable

{

Q q; Producer(Q q)

{

this.q = q;

new Thread(this, "Producer").start();

}

**public void run()**

{

int i = 0;

while(true)

{

q.put(i++);

}

}

}**//end of Producer**

class Consumer implements Runnable

{

Q q;

Consumer(Q q)

{

this.q = q;

new Thread(this, "Consumer").start();

}

public void run()

{

while(true)

{

q.get();

}

}

}**//end of Consumer**

class PCFixed

{

public static void main(String args[])

{

Q q = new Q(); new Producer(q); new Consumer(q);

System.out.println("Press Control-C to stop.");

}

}

#### Suspending, Blocking and StoppingThreads

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

#### Thread.stop();

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

Blocking Thread

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

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

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

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

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

**Example program:**

The following program demonstrates these methods:

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

TestThread.java

//(1) declare a class

class AThread extends Thread

{

//(2) override run() method public void run()

{

try{

for(int i=1;i<=10;i++)

{

System.out.println("A Thread i ="+i); Thread.sleep(1000);

}

}

catch(Exception e)

{

}

}

}

//(1) declare a class

class BThread extends Thread

{

//(2) override run() method public void run()

{

try{

for(int j=1;j<=10;j++)

{

System.out.println("B Thread j ="+j); Thread.sleep(1000);

}

}

catch(Exception e)

{

}

}

}

class TestThread

{

public static void main(String args[])

{

//(3) create object from above class and call start() method AThread a=new AThread();

BThread b=new BThread();

//call start() method a.start();

b.start();

try

{

Thread.sleep(1000); System.out.println("Thread a is suspending"); **a.suspend(); //suspending** Thread.sleep(1000); System.out.println("Thread a is Resuming"); **a.resume(); // resuming**

**b.stop(); //stoping**

}

catch(Exception e)

{

}

}

}

**Thread Exceptions**

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

It’s general form will be as follows try

{

Thread.sleep(1000);

}

cathc(Exception e)

{ --------

}

## Deadlock

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

X

Thread1

Thread2

Y

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

Example program:

public class TestDead

{

#### TestDead.java

public static void main(String[] args)

{

final String resource1 = "John Gardner"; final String resource2 = "James Gosling";

// t1 tries to lock resource1 then resource2

Thread t1 = new Thread()

{

public void run()

{ //locking the resource synchronized (resource1)

{

System.out.println("Thread 1: locked resource 1");

try {

}

Thread.sleep(100);

catch (Exception e)

{

System.out.println(e);

}

synchronized (resource2)

{

System.out.println("Thread 1: locked resource 2");

}

}

} //end of run()

}; //end of t1

// t2 tries to lock resource2 then resource1 Thread t2 = new Thread()

{

public void run()

{

synchronized (resource2)

{

System.out.println("Thread 2: locked resource 2"); try { Thread.sleep(100);} catch (Exception e) {} synchronized (resource1)

{

System.out.println("Thread 2: locked resource 1");

}

}

}//end of run()

}; //end of t2

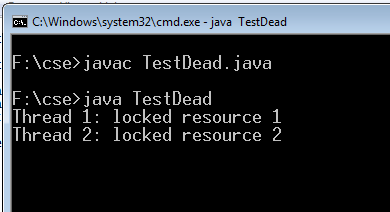
t1.start();

t2.start();

}

}

## Output:



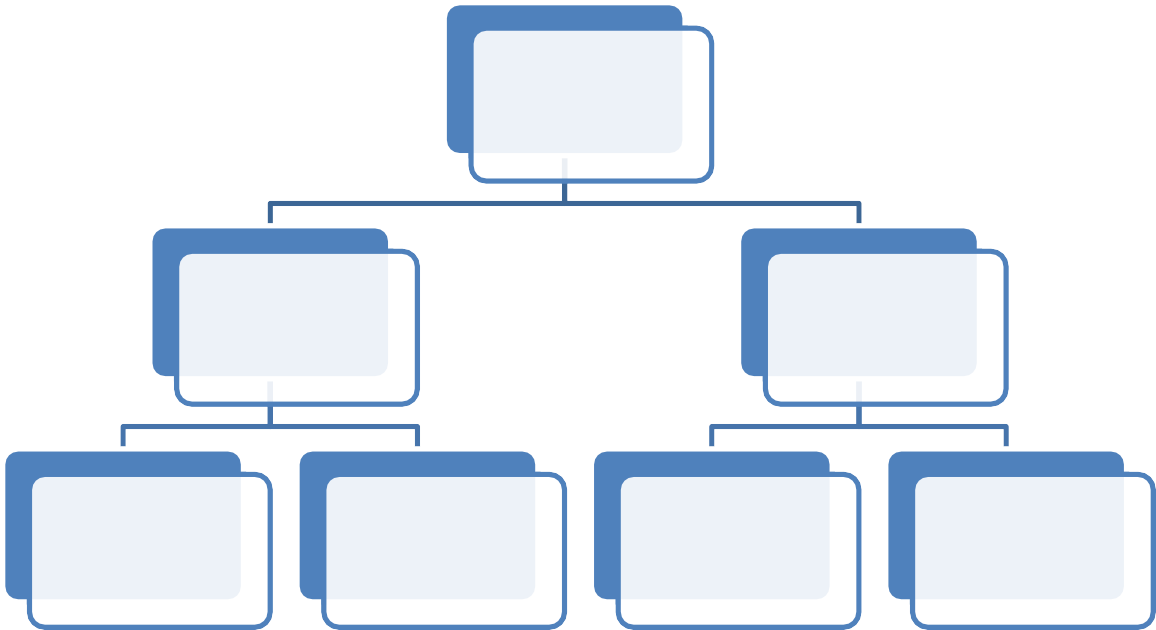
**Input/Output: reading and writing data, java.io.package**

The two most important parts of the computer are: ***input and output***. The input is one that is processed to generate output. There two predefined package provided by java to support both input and output operations. There are two predefined packages in java that contain classes to perform I/O operations. These are **java.io.\***, and **java.nio.\***. The **java.io.\*** used to perform reading and writing to console and reading and writing to the Files. The **java.nio.\***, contains all the classes of java.io.\*, and also contain the classes to perform advanced operations such as buffering, memory mapping, character encoding and decoding etc;.

The java.io.\* package provides separate classes for reading and writing data. The java I/O facility is based on streams. A stream is a continuous flow of data. In java streams for both

types of data have been defined: **byte streams** and **character streams**. The byte stream classes deal with reading and writing of ***bytes*** with sockets and files. The character stream classes deal with reading and writing of ***characters*** with sockets and files.

The java.io, provides two top level byte stream abstract classes: java.io.InputStream and OutputStream classes. It also provides two other top level classes for Character stream abstract classess: java.io.Reader and java.io.Writer. The subclasses of these classes are actually used for reading and writing data.



**java.io**

**package**

**byte stream**

**character**

**stream**

**java.io.InputStream**

**java.io.OutputStream**

**java.io.Reader**

**java.io.Writer**

**Stream**

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

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



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

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

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

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

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

**CopyFile.java**

import java.io.\*; public class CopyFile

{

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

{

FileInputStream in=null; FileOutputStream out=null;

try

{

in=new FileInputStream("CopyFile.java"); out=new FileOutputStream("output.txt"); int c;

while((c =in.read())!=-1)

{

out.write(c);

}

} //end of try finally

{

if(in!=null)

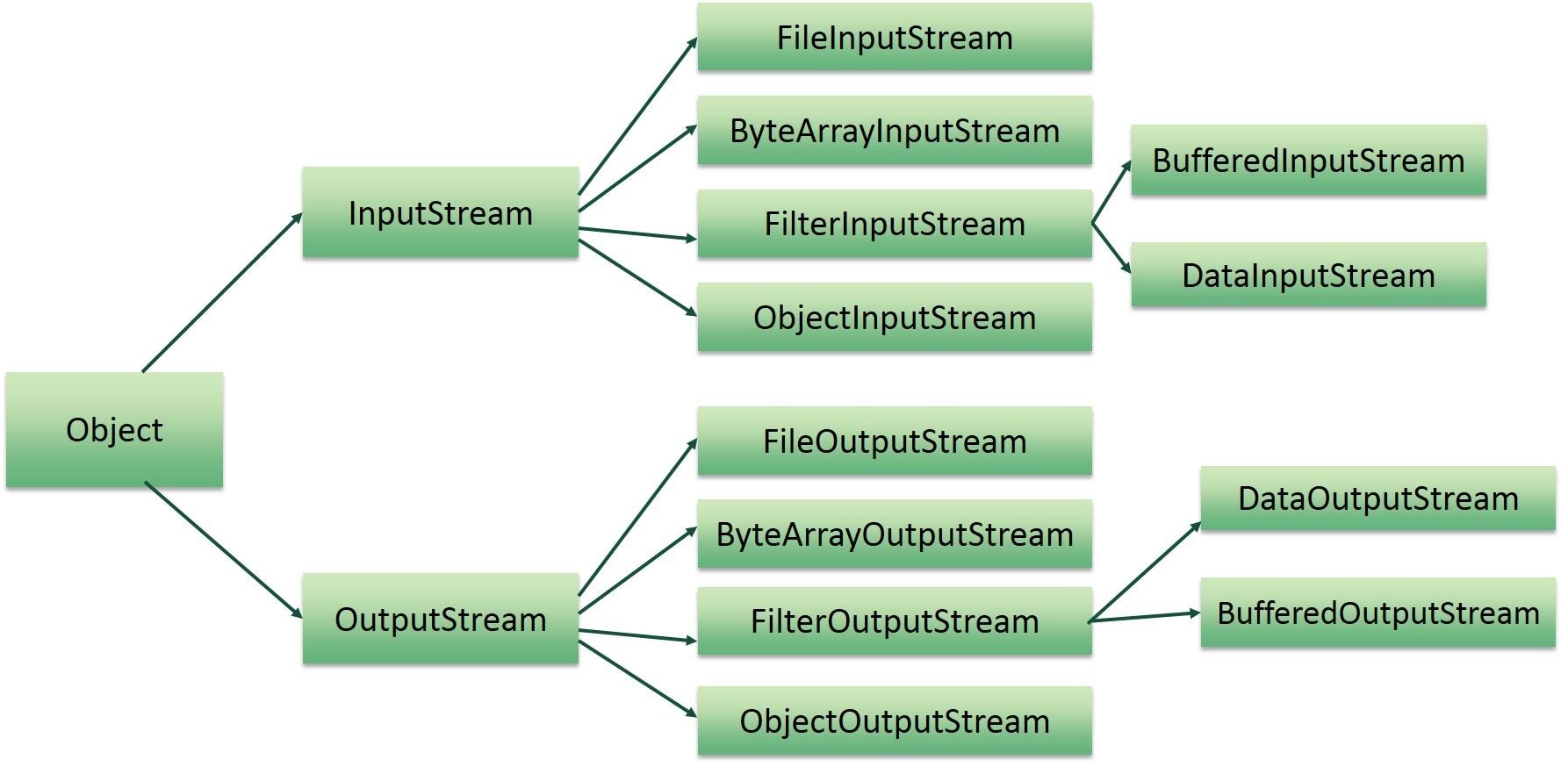
{

in.close(); //used to close the file input stream

}

if(out!=null)

{



**Example program using Input and Output Streams**

out.close();//used to close the file output stream

}

}//end of finally

}//end of main

} //end of class

#### java.io.File class

The j**ava.io.File** class is very important to know. This is not belongs to either Byte Stream or Character Stream classes. This is mainly used to test the properties of the file, such as is it file, or is it executable, is it Directory etc,;

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

|  |  |
| --- | --- |
| **Method** | **Description** |
| [**boolean canExecute()**](http://www.tutorialspoint.com/java/io/file_canexecute.htm) | This method tests whether the application can execute the file  denoted by this abstract pathname. |
| [**boolean canRead()**](http://www.tutorialspoint.com/java/io/file_canread.htm) | This method tests whether the application can read the file denoted by  this abstract pathname. |
| [**boolean canWrite()**](http://www.tutorialspoint.com/java/io/file_canwrite.htm) | This method tests whether the application can modify the file denoted  by this abstract pathname. |
| [**boolean exists()**](http://www.tutorialspoint.com/java/io/file_exists.htm) | This method tests whether the file or directory denoted by this  abstract pathname exists. |
| [**String getName()**](http://www.tutorialspoint.com/java/io/file_getname.htm) | This method returns the name of the file or directory denoted by this  abstract pathname. |
| [**String getParent()**](http://www.tutorialspoint.com/java/io/file_getparent.htm) | This method returns the pathname string of this abstract pathname's  parent, or null if this pathname does not name a parent directory. |
| [**String getPath()**](http://www.tutorialspoint.com/java/io/file_getpath.htm) | This method converts this abstract pathname into a pathname string. |
| [**String[] list()**](http://www.tutorialspoint.com/java/io/file_list.htm) | This method returns an array of strings naming the files and  directories in the directory denoted by this abstract pathname. |
| [**boolean isDirectory()**](http://www.tutorialspoint.com/java/io/file_isdirectory.htm) | This method tests whether the file denoted by this abstract pathname  is a directory. |
| [**boolean isFile()**](http://www.tutorialspoint.com/java/io/file_isfile.htm) | This method tests whether the file denoted by this abstract pathname  is a normal file. |

**Example program:**

import java.io.File; class FileDemo

{

public static void main(String args[])

{

#### FileDemo.java

File f=new File(args[0]);

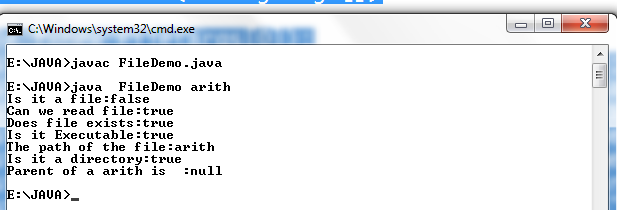
System.out.println("Is it a file:"+f.isFile()); System.out.println("Can we read file:"+f.canRead()); System.out.println("Does file exists:"+f.exists()); System.out.println("Is it Executable:"+f.canExecute()); System.out.println("The path of the file:"+f.getPath()); System.out.println("Is it a directory:"+f.isDirectory());

System.out.println("Parent of a "+f.getName()+" is :"+f.getParent());

}

}

#### Output:



**Byte Streams**

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

,**FileInputStream** ,**FileOutputStream, DatInputStream and DataOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file:

## Reading and Writing using the Byte Stream classes ( FileInputStream and FileOutputStream)

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

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

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

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public void close() throws IOException{}** |

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

#### Methods of FileOutputStream are as follow:

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

The Constructor will be as follow:

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

List of Methods of FileOutputStream are as Follow:

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public void close() throws IOException{}**This method closes the file output stream. |

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

Example Program:

**Example program using FileInputStream and FileOutputOutput Streams**

**CopyFile.java**

import java.io.\*; public class CopyFile

{

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

{

FileInputStream in=null; FileOutputStream out=null;

try

{

in=new FileInputStream("CopyFile.java"); out=new FileOutputStream("output.txt"); int c;

while((c =in.read())!=-1)

{

out.write(c);

}

} //end of try finally

{

if(in!=null)

{

in.close(); //used to close the file input stream

}

if(out!=null)

{

out.close();//used to close the file output stream

}

}//end of finally

}//end of main

} //end of class

#### Using the DataInputStream and DataOutputStream classes of Byte Stream

A **data input stream** enable an application read primitive Java data types from an underlying input stream in a machine-independent way(instead of raw bytes). That is why it is called DataInputStream – because it reads data (numbers) instead of just bytes.

A **data output stream** lets an application write primitive Java data types to an output stream in a portable way. An application can then use a data input stream to read the data back in

#### DataInputStream

|  |  |
| --- | --- |
| **Sl. No** | **class methods** |
| **1** | **int read(byte[] b) :** Reads some number of bytes from the contained input stream and  stores them into the buffer array b. |
| **2** | **int read(byte[] b, int off, int len) :** Reads up to len bytes of data from the contained input  stream into an array of bytes. |
| **3** | **boolean readBoolean() :** Reads one input byte and returns true if that byte is nonzero, false  if that byte is zero. |
| **4** | **char readChar() :** Reads two input bytes and returns a char value. |
| **5** | **double readDouble() :** Reads eight input bytes and returns a double value. |
| **6** | **int readInt() :** Reads four input bytes and returns an int value. |
| **7** | **String readLine():** Reads a string from the input string. |

**DataOutputStream**

|  |  |
| --- | --- |
| **Sl.No** | **class methods** |
| **1** | **void flush() :** Flushes this data output stream. |
| **2** | **int size () :** Returns the current value of the counter written, the number of bytes written to  this data output stream so far. |
| **3** | **void write (byte[] b, int off, int len) :** Writes len bytes from the specified byte array starting at  offset off to the underlying output stream. |
| **4** | **void write (int b) :** Writes the specified byte (the low eight bits of the argument b) to the  underlying output stream. |
| **5** | **void writeBoolean (boolean v) :** Writes a boolean to the underlying output stream as a 1-byte  value. |
| **6** | **void writeByte (int v) :** Writes out a byte to the underlying output stream as a 1-byte value. |
| **7** | **void writeChar (int v) :** Writes a char to the underlying output stream as a 2-byte value, high  byte first. |
| **8** | **void writeDouble (double v) :** Converts the double argument to a long using the  doubleToLongBits method in class Double, and then writes that long value to the underlying |

|  |  |
| --- | --- |
|  | output stream as an 8-byte quantity, high byte first. |
| **9** | **void writeFloat (float v):** Converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the underlying output stream as a 4-  byte quantity, MSB first. |
| **10** | **void writeInt (int v) :** Writes an int to the underlying output stream as four bytes, high byte  first. |

Example Program using the **DataInputStream and DataOutputStream**

import java.io.\*; class DIOTest

{

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

{

try

{

#### DataInputStream dis=new DataInputStream(System.in);

int a,b,c; float avg;

System.out.println("Enter values from the Keyboard"); a=Integer.parseInt(dis.readLine()); b=Integer.parseInt(dis.readLine()); c=Integer.parseInt(dis.readLine());

avg=(a+b+c)/4; System.out.println("The avg is :"+avg);

#### DataOutputStream dos=new DataOutputStream(new FileOutputStream("rao.txt"));

dos.writeChar('A'); //writes one character in the output stream

}

catch(FileNotFoundException e)

{

}

}

}

# Reading and Writing Using Character Streams

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

### Methods of FileReader class

|  |  |
| --- | --- |
| **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. |

**Methods of FileWriter class**

|  |  |
| --- | --- |
| **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. |

#### Example program:FileReadDemo.java

FileReadDemo.java

import java.io.\*; class FileReadDemo

{

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

{

File f=new File("FileReadDemo.java"); if(f.exists())

{

FileReader fr=new FileReader(f);

FileWriter fw=new FileWriter("FileWrite.txt"); int n;

String str=""; while((n=fr.read())!=-1)

{

System.out.print((char)n); str=str+(char)n;

}

fw.write(str); //writing to the FileWriter fw.close(); //closing is very important

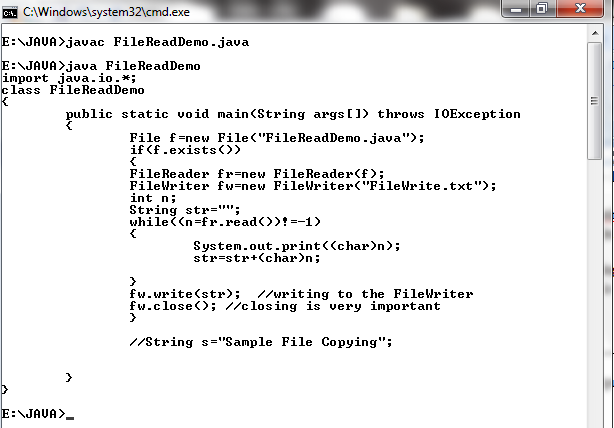
}

}

}

**Output:**

The "FileWrite.txt" contains the output. Go to the Current directory and open the “FileWrite.txt”.



**Reading and Writing Using the Console (Scanner class)**

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

#### ScannerDemo.java

import java.util.\*; class ScannerDemo

{

public static void main(String args[])

{

Scanner s=new Scanner(System.in); String name;

int age;

System.out.println("Enter your name:"); name=s.nextLine(); System.out.println("enter your age;"); age=s.nextInt();

System.out.println("Your name is:"+name); System.out.println("Your Age is:"+age);

}

}

-----\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*End of 4th Unit\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*-------