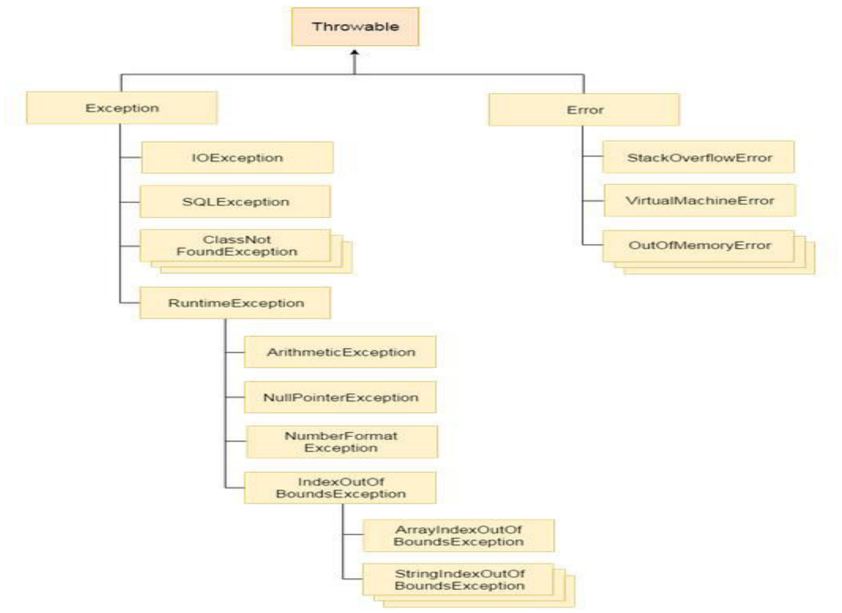
Unit -3

Exception handling in JAVA

Exception Handling in Java

- It is the mechanism to accomplish normal flow of the application to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc…
- In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Hierarchy of Java Exception classes



Types of Exception

- There are mainly two types of exceptions: checked and unchecked exception. The sun microsystem says there are three types of exceptions:
- Checked Exception
- Unchecked Exception
- Error

Difference between checked and unchecked exceptions

1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Common exceptions

- 1. 50/0 → Arithmetic exception
- 2. String S = null, S.o.p(s.length) → ________________________
- 3. int a[5];

  a[6] = 20; → __________________

4. String s= “HELLLO”;

S.o.p(Interger.parseInt(S)); → _________

Java Exception Handling Keywords

- There are 5 keywords used in java exception handling.
- try -> try block is used to enclose the code that might throw an exception. It must be used within the method
- Catch -> Java try block must be followed by either catch or finally block. catch block is used to handle the Exception. multiple catch block with a single try is also possible
- Finally-> **Java finally block**is always executed whether exception is handled or not. **Java finally block** follows try or catch **block**.

- Throw -> The **throw** keyword in **Java** is used to explicitly **throw** an exception from a method or any **block** of code. We can **throw** either checked or unchecked exception. The **throw** keyword is mainly used to **throw** custom exceptions. When a **throw** statement is encountered and executed, execution of the current method is stopped and returned to the caller
- Throws ->Whereas the **throws keyword** is used to declare that a method may **throw** one or some exceptions

Syntax

java try-catch

- try{
- //code that may throw exception
- }catch(Exception_class_Name ref){}

try-finally block

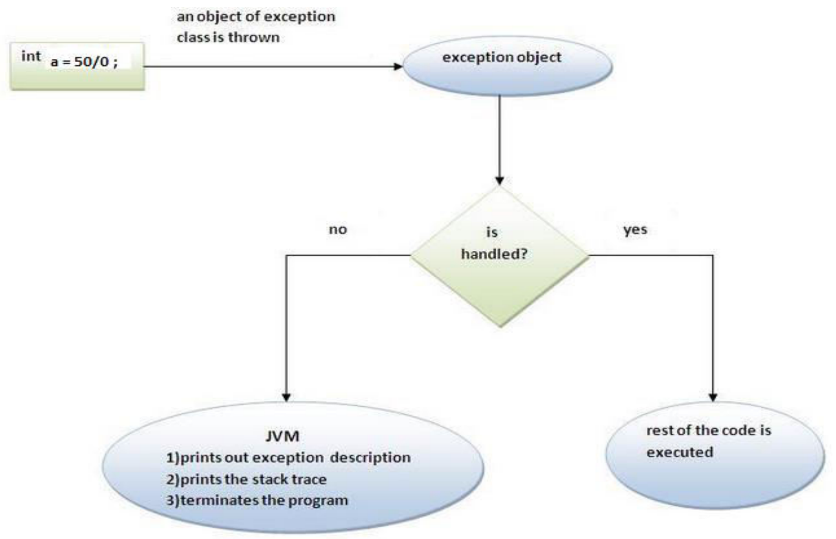
- try{
- //code that may throw exception
- }finally{}

Example program for arithmetic exception

```
// program with out exception
class exp1
{
public static void main(String
   ar[])
{
int a = 50/0;
System.out.println("the value
   of a "+a);
}
}
```

```
// program using exception
class exp1
{
public static void main(String
ar[])
{
try
{
int a = 50/0;
System.out.println("the value
of a "+a);
}catch(Exception
e){System.out.println(e);}
System.out.println("lines of
coding continues");
```

How exception is handled using try-catch



int a = 50/0 ;
an object of exception class is thrown
exception object
is handled?
no
yes
JVM
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

Multiple catch block

```
class exp2
{
public static void main(String ar[])
{
try
{
String s = "helo";
int a[] = new int[5];
a[5]= Integer.parseInt(s);
} catch (NumberFormatException  e )
{
System.out.println("\nNumber  format:"+e);
}
catch(ArrayIndexOutOfBoundsException  e)
{
System.out.println("\nArray  index: "+e);
}
catch(Exception  e){}
System.out.println("\nprogram  continues...");
}
}
```

Conclusions from multiple catch

1. **At a time only one Exception is occured and at a time only one catch block is executed.**
2. **All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception . (Get the answer for this conclusion)**

Nested try-catch block

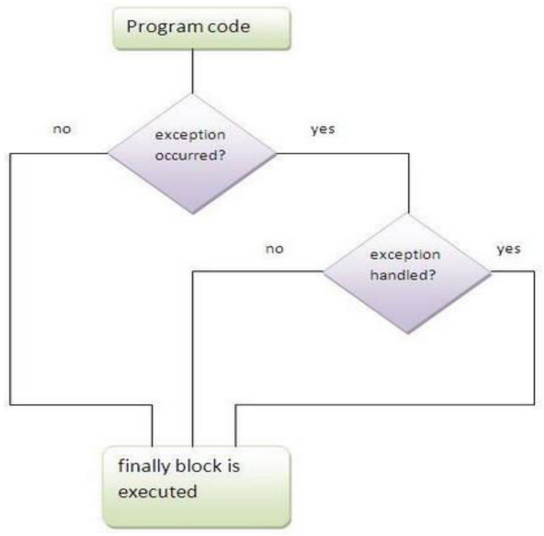
- Some cases where certain set of code causes some error and another set of code causes another error then we are in need of nested try-catch block..(Example program exp3.java)

```
• class exp3
• {
• public static void main(String ar[])
• {
• try
• {
• try
• {
• String s = "helo";
• int a  = Integer.parseInt(s);
• } catch (NumberFormatException
  e ){
• System.out.println("\nNumber
  format:"+e);
• }
try
{
int a[] = new int[5];
a[5]= 100;
}catch(ArrayIndexOutOfBoundsException
e)
{
System.out.println("\nArray index: "+e);
}
}catch(Exception e)
{
System.out.println("General
exeception.....");
}
System.out.println("\nprogram
continues...");
}
}
```

Finally block

- **Java finally block** is a **block** that is used to execute important code such as closing connection, stream etc. **Java finally blockis** always executed whether exception is handled or not. **Java finally block** follows try or catch **block**.

Finally block



```
// program for finally to be executed even if
// exception is not present
class finally1
{
public static void main(String args[])
{
try
{
int a=10,b=10,c;
c = a/b;
System.out.println(c);
}catch(Exception e)
{
System.out.println(e);
}
 finally
{
System.out.println("This block will execute even if the
exception is not present");
}
 System.out.println("program continues..");     }     }
```

```
// program for finally to be executed if  exception present
class finally2
{
public static void main(String args[])
{
try
{
int a=10,b=0,c;
c = a/b;
System.out.println(c);
}catch(ArrayIndexOutOfBoundsException e)
{
System.out.println(e);
}
 finally
{
System.out.println("This block will execute even if the
exception is present but not handled");
}
 System.out.println("program continues..");     }     }
```

- class finally2
- {
- public static void main(String args[])
- {
- try
- {
- int a[] = new int[10];
- a[23]=45;
- }
- finally
- {
- int a=10, b=0;
- int c = a/b;
- System.out.println(c);
- System.out.println("This block will execute even if the exception is present but not
- handled");
- }
- System.out.println("program continues..");
- }

Try-catch inside finally block

```
•       class finally2
•       {
•       public static void main(String args[])
•       {
•       try
•       {
•       int a[] = new int[10];
•       a[23]=45;
•       } catch(Exception e) {System.out.println(e);}


•        finally
•       {
•       try
•       {
•       int a=10, b=0;
•       int c = a/b;
•       System.out.println(c);
•       System.out.println("This block will execute even if the exception is present but not

•       handled");
•       }catch(Exception e){System.out.println(e);}

•       }

•       System.out.println("program continues..");
•            }
•          }
```

User-defined Exception

- If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

```
•       import java.util.*;
•          class Incometax extends Exception
•       {
•           Incometax(String s)
•       {
•            super(s);
•           }
•          }

•          class userdefinedexception
•       {
•
•             static void check(int salary) throws Incometax
•       {
•               if(salary <100000)
•                throw new Incometax("No need to pay tax");
•               else
•                System.out.println("Pay your tax");
•            }
•
•             public static void main(String args[])
•       {
•              int n;
•                          Scanner s = new Scanner(System.in);
•                          System.out.println("Enter your annual salary");
•                          n = s.nextInt();
•                try{
•                check(n);
•                }catch(Exception e){System.out.println("Exception caught: "+e);}
•
•                System.out.println("sucess...");
•            }
•          }
```

Advantage of Exception Handling

- **to maintain the normal flow of the application**.
- Ie. Suppose there is 6lines of coding statements in our program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

Practise Programs- Exception Handling

1. Write a class to sort the given set of n integers in descending order. Include a try block to locate the array index out of bound exception and catch it.

2. Write a program to calculate the grade of a student. Enter marks for minimum 5 subjects. While entering marks if the mark is negative throw NegativeMarkException and if the mark is greater than 100 throw OutofRangeException.

3. Write a Java Program to create a class Employee containing EmpCode, name, age, experience. Create a constructor to initialize the data members. Create a display( ) method to display the details. Create a check( )method to check for the following conditions:

- Length of EmpCode should be exactly 6
- Maximum characters of name is 30
- Age cannot be less than 18 ormore than 58
- Experience cannot exceed age-18

4. Write a java program to get the age of the user, if the age of the user is less than 18, give a message that " you are not eligible for voting"", else allow the user to provide their vote.

Thank you

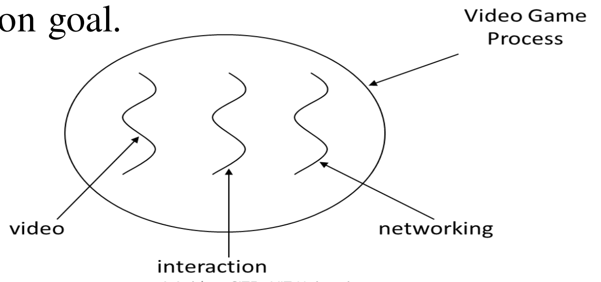
MULTITHREADING
UNIT - 3

Topics to be covered

- Life cycle of a Thread
- Creating Thread
- Thread Scheduler
- Sleeping a Thread
- Calling run() method

Thread

- Individual and separate unit of execution that is part of a process (Process- is a program in execution, Thread – dispatch able unit of work)
  - multiple threads can work together to accomplish a common goal.



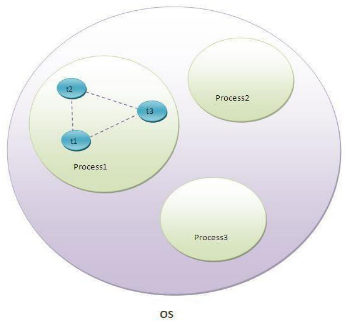
Multitasking

- Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:
- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)
  - Threads share the same address space.
  - Thread is lightweight.
  - Cost of communication between the thread is low.

Thread in java

- 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. It shares a common memory area.



Note: At a time one thread is executed only.

Life cycle of a Thread (Thread States)

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

- New
- Runnable
- Running
- Non-Runnable (Blocked)
- Terminated

1) New:

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable:

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

Creating thread

- There are two ways to create a thread:
  - –By extending Thread class
  - –By implementing Runnable interface.

Thread class

- Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Commonly used **Constructors** of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used **methods** of Thread class:

- **public void run()**: is used to perform action for a thread.
- **public void start()**: starts the execution of the thread. JVM calls the run() method on the thread.
- **public void sleep(long miliseconds)**: Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- **public int getPriority():** returns the priority of the thread.
- **public int setPriority(int priority):** changes the priority of the thread.

- **public String getName():** returns the name of the thread.
- **public void setName(String name):** changes the name of the thread.
- **public Thread currentThread():** returns the reference of currently executing thread.
- **public int getId():** returns the id of the thread.
- **public Thread.State getState():** returns the state of the thread.
- **public boolean isAlive():** tests if the thread is alive.

Starting a thread

- **start() method** of Thread class is used to start a newly created thread.

It performs following tasks:

- A new thread starts(with new call stack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

Sample program

```
Class sample extends Thread
{
public void run()
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Sample  t1=new sample();
t1.start();
 }
}
```

Thread Scheduler in java

- **Thread scheduler** in java is the part of the JVM that decides which thread should run.
- There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.
- Only one thread at a time can run in a single process.
- The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Sleep() in java

- The sleep() method of Thread class is used to sleep a thread for the specified amount of time.
- Syntax:
- public static void sleep(long miliseconds)throws InterruptedException
- public static void sleep(long miliseconds, int nanos)throws InterruptedException

```
class threadtest
{
public static void main(String arg[])
{
 thread1 t1 = new thread1();
thread2 t2 = new thread2();
thread3 t3 = new thread3();
System.err.println("Starting
    thread");
t1.start();
t2.start();
t3.start();
System.out.println("threads
    started");
}
}
```

```
class thread1 extends Thread
```

```
• class thread2 extends Thread
```

```
class thread3 extends Thread
{
public void run()
{
try
{
System.out.println("thread3");
sleep(3000);
System.out.println(" Thread3 is
running now... ");
}catch(Exception e) {
System.out.println(e.toString());
}
}
}
```

Can we start a thread twice??

No. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

What if we call run() method directly instead start() method?

- Each thread starts in a separate call stack.
- Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

Instead start() ; if run() is used?

```
class sample extends Thread
{
public void run()
{
System.out.println(" thread running");
/*try
{
sleep(1000);
System.out.println("thread running now");
}catch(Exception e){}*/
for(int i=1;i<5;i++){

try{Thread.sleep(500);}catch(InterruptedExce
ption e){System.out.println(e);}
   System.out.println(i);
 }

}
```

```
public static void main(String ar[])
{
sample s = new sample();
sample s2 = new sample();
s.start();
s2.start();
//s.run();
//s2.run();
}
}
```

Thread priority

- Each thread have 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 pre-emptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

```
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.

```
class threadtest
{
public static void main(String arg[])
{
thread1 t1 = new thread1();
thread2 t2 = new thread2();
thread3 t3 = new thread3();
System.err.println("Starting
thread");
t1.setPriority(Thread.MAX_PRIORI
TY);
t2.setPriority(Thread.MIN_PRIORIT
Y);
t1.start();
t2.start();
t3.start();
System.out.println("threads
started");
}
}
```

```
class thread1 extends Thread
{
int time;
public void run()
{
try
{
System.out.println("thread1");
sleep(1000);
}catch(Exception e) {
System.out.println(e.toString());
}
}
}
```

Creating thread using Runnable interface

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

```
//Program using interface
 class runnable implements Runnable
{
   public void run()
{
   System.out.println("thread is
running...");
   }

public static void main(String a[])
{
   runnable m1=new runnable();
   Thread t1 =new Thread(m1);
  t1.start();
  Thread t2 = new Thread(m1);
 t2.start();
   }
  }
```

```
//program using class
Class sample extends Thread
{
public void run()
{
System.out.println("thread is r
unning...");
}
public static void main(String a
rgs[])
{
Sample  t1=new sample();
t1.start();
 }
}
```

Reasons for implementing a Runnable interface

- There are two reasons for implementing a Runnable interface preferable to extending the Thread Class:

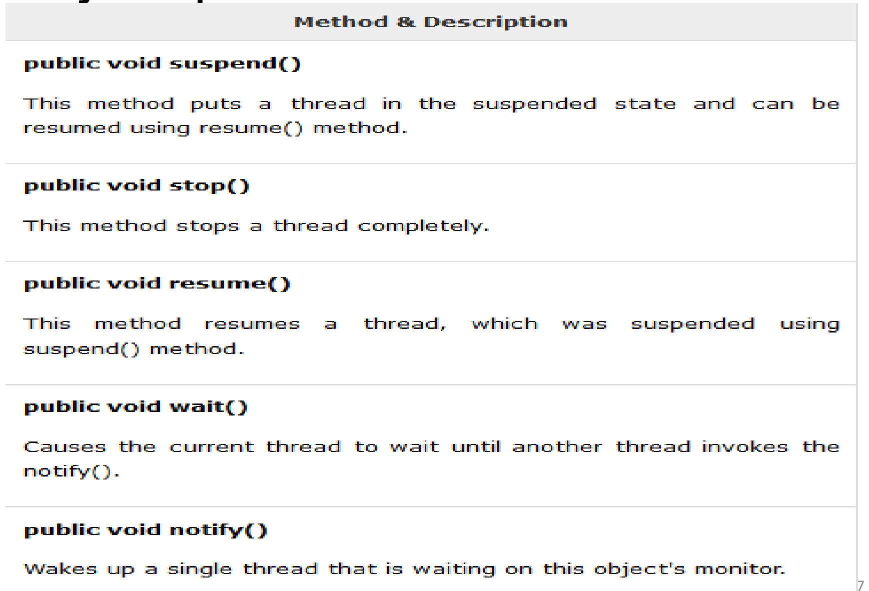
1. If you extend the Thread Class, that means that subclass cannot extend any other Class, but if you implement Runnable interface then you can do this.

2. The class implementing the Runnable interface can avoid the full overhead of Thread class which can be excessive.

performing single task by multiple threads

```
class tasking implements Runnable{
  public void run(){
  System.out.println("task one");
  }

  public static void main(String args[]){
  Thread t1 =new Thread(new tasking());//passing anonymous object
                                              //of tasking class
  Thread t2 =new Thread(new tasking());

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

   }
  }
```

Major operations on thread control



Method & Description
public void suspend() This method puts a thread in the suspended state and can be resumed using resume() method.
public void stop() This method stops a thread completely.
public void resume() This method resumes a thread, which was suspended using suspend() method.
public void wait() Causes the current thread to wait until another thread invokes the notify().
public void notify() Wakes up a single thread that is waiting on this object's monitor.

Thread 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.
- Methods:

public void join()throws InterruptedException

public void join(long milliseconds)throws InterruptedException

```
class join1 extends
Thread{
    public void run(){
     for(int i=1;i<=5;i++){
      try{
       Thread.sleep(500);
      }catch(Exception
e){System.out.println(e);}

System.out.println(Threa
d().getName()+i);
     }
    }
```

```
 public static void main(String
args[]){
   join1 t1=new join1();
   join1 t2=new join1();
   join1 t3=new join1();
   t1.start();
   try{
    t1.join();
   }catch(Exception
e){System.out.println(e);}

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

Synchronized in Java

- Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.
- So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.

```
synchronized(sync_object)

 {
// Access shared variables and other
 // shared resources
 }
```

Example program – threadsync.java

```
• class multi
• {
• synchronized void multiplication(int n)
• {
• for(int i=1;i<=5;i++){
•         System.out.println(n*i);
•         try{
•          Thread.sleep(400);
•         }catch(Exception e)

• {System.out.println(e);}
•       }
• }
• }
• class threadsync
• {
• public static void main(String aa[])
• {
• multi m = new multi();
• thread1 t1 = new thread1(m);
• thread2 t2 = new thread2(m);
• t1.start();
• t2.start();
• }
• }
```

```
class thread1 extends Thread
{
multi m;
thread1(multi t)
{
this.m=t;
}

public void run()
{
m.multiplication(5);
}
}
class thread2 extends Thread
{
multi m;
thread2(multi t)
{
this.m=t;
}

public void run()
{
m.multiplication(10);
}
}
```

Dead lock in java

- 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. – deadlock.java

```
• class deadlock
• {
• public static void main(String aa[])
• {
• final String a="a";
• final String b ="b";

• Thread t1 = new Thread()
• {
• public void run()
• {
• synchronized(a)
• {
• System.out.println("thread1: locked a");
• try
• {
• Thread.sleep(100);
• }catch(Exception e) { }
• synchronized(b)
• {
• System.out.println("thread1: locked b");
• }
• }
• }
• };
```

```
Thread t2 = new Thread()
{
 public void run()
 {
      synchronized (b) {
        System.out.println("Thread 2: locked 
b");

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

        synchronized (a) {
          System.out.println("Thread 2: locked 
a");
        }
       }
      }
    };
t1.start();
t2.start();
}
}
```

Inter thread Communication

- **wait()** -causes current thread to **wait** until another thread invokes the notify**()** method or the notifyAll**()** method for this object
- public void notify() -Wakes up a single **thread** that is waiting on this object's monitor.
- public void notifyAll() -Wakes up all the threads that called wait( ) on the same object.

```
class student extends Thread
{
int amount=1000;

 void withdraw(int amount)
{
synchronized(this)
{
System.out.println("Withdrawing...");

if( this.amount<amount)
{
System.out.println("Low balance....");

try
{
wait();
}catch(Exception e){}
}//if ends
this.amount=this.amount-amount;
System.out.println("withdraw completed");
}
}
synchronized void deposit(int amount)
{
   System.out.println("going to deposit...");
   this.amount=this.amount+amount;
   System.out.println("deposit completed... ");
   notify();
   }
   }  // student ends
```

```
class interth{
    public static void main(String args[]){
   student  s=new student();
    new Thread(){
    public void run(){s.withdraw(1500);}
    }.start();
 new Thread(){
  public void run(){s.deposit(1000);}
  }.start();

 }}
```

Thank you

Java IO Packages

Introduction

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

Stream

- A stream is a sequence of data. In Java a stream is composed of bytes. It's called a stream because it's like a stream of water that continues to flow.
- In java, 3 streams are created for us automatically. All these streams are attached with console.
- **1) System.out:** standard output stream
- **2) System.in:** standard input stream
- **3) System.err:** standard error stream

Output Stream& Input stream

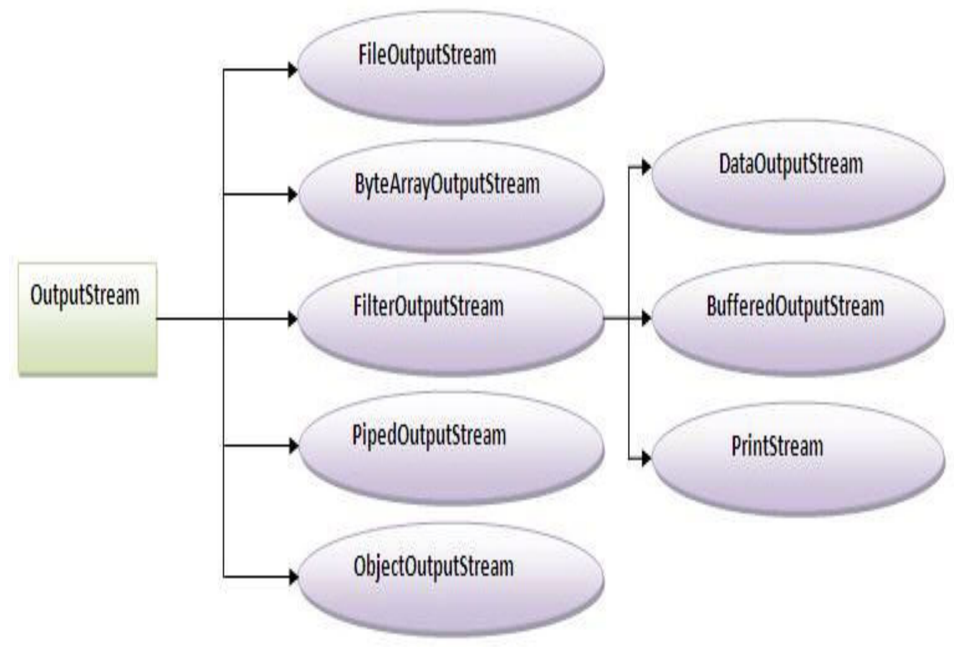
- Java application uses an output stream to write data to a destination, it may be a file, an array, peripheral device or socket.
- Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.

OutputStream class

- OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream

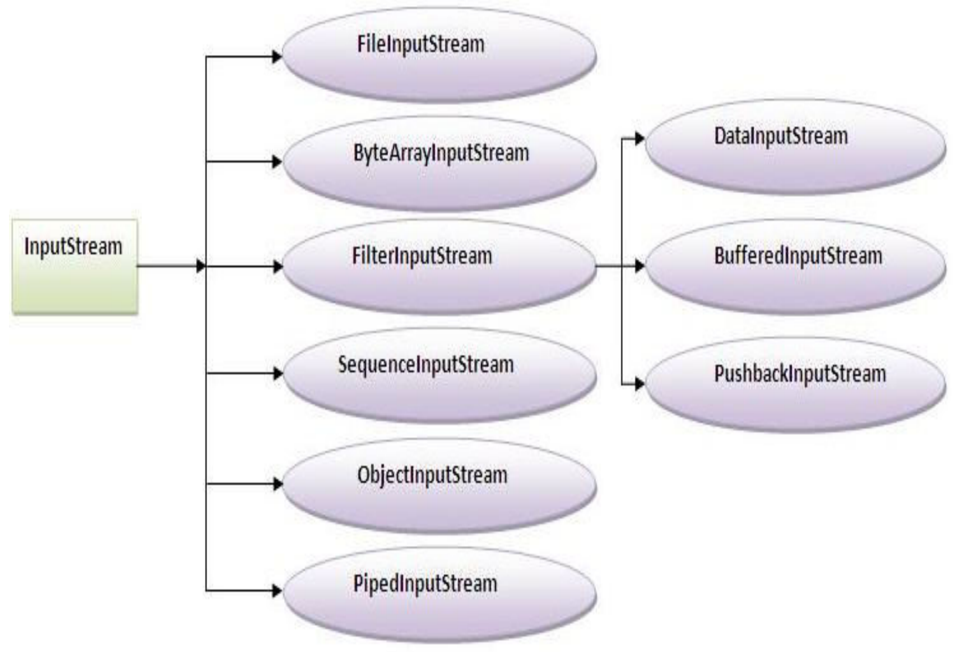
Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.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



InputStream Class

- InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

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



Java File Class

- The File class is an abstract representation of file and directory pathname. A pathname can be either absolute or relative.
- The File class have several methods for working with directories and files such as creating new directories or files, deleting and renaming directories or files, listing the contents of a directory etc.

File Constructor

Constructor	Description
File(File parent, String child)	It creates a new File instance from a parent abstract pathname and a child pathname string.
File(String pathname)	It creates a new File instance by converting the given pathname string into an abstract pathname.
File(String parent, String child)	It creates a new File instance from a parent pathname string and a child pathname string.
File(URI uri)	It creates a new File instance by converting the given file: URI into an abstract pathname.

Methods of File Class

Modifier and Type	Method	Description
static File	createTempFile(String prefix, String suffix)	It creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
boolean	createNewFile()	It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
boolean	canWrite()	It tests whether the application can modify the file denoted by this abstract pathname.String[]
boolean	canExecute()	It tests whether the application can execute the file denoted by this abstract pathname.

boolean	canRead()	It tests whether the application can read the file denoted by this abstract pathname.
boolean	isAbsolute()	It tests whether this abstract pathname is absolute.
boolean	isDirectory()	It tests whether the file denoted by this abstract pathname is a directory.
boolean	isFile()	It tests whether the file denoted by this abstract pathname is a normal file.
String	getName()	It returns the name of the file or directory denoted by this abstract pathname.
String	getParent()	It returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.

Path	toPath()	It returns a java.io.file.Path object constructed from the this abstract path.
URI	toURI()	It constructs a file: URI that represents this abstract pathname.
File[]	listFiles()	It returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname
long	getFreeSpace()	It returns the number of unallocated bytes in the partition named by this abstract path name.
String[]	list(FilenameFilter filter)	It returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
boolean	mkdir()	It creates the directory named by this abstract pathname.

```
import java.io.*;
  class File1
{
    public static void main(String[] args) {

      try {
        File f = new File("firstfile.txt");
        if (f.createNewFile()) {
          System.out.println("New File is created!");
        } else {
          System.out.println("File already exists.");
        }
      } catch (IOException e) {
        e.printStackTrace();
      }

    }
```

```
// to list all the file in the directoryjav
import java.io.*;
 class File2
 {
public static void main(String[] args)
 {
   File f=new File("C:/users/admin/desktop/java/");
   String filenames[]=f.list();
   for(String filename:filenames){
      System.out.println(filename);
   }
}
}
```

Java FileWriter and FileReader

- Java FileWriter and FileReader classes are used to write and read data from text files. These are character-oriented classes, used for file handling in java.

Java FileWriter class

- Java FileWriter class is used to write character-oriented data to the file.

Constructors of FileWriter class:

- 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 of FileWriter

Methods	Description
1) public void write(String text)	writes the string into FileWriter.
2) public void write(char c)	writes the char into FileWriter.
3) public void write(char[] c)	writes char array into FileWriter.
4) public void flush()	flushes the data of FileWriter.
5) public void close()	closes FileWriter.

Java File Reader class

Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class.

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.

Method	**Description**
1) public int read()	returns a character in ASCII form. It returns -1 at the end of file.
2) public void close()	closes FileReader.

Write To a File

- We used the FileWriter class together with its write() method to write some text to the file.

```
import java.io.FileWriter;
import java.io.IOException;

public class writefile
{
  public static void main(String[] args) {
    try
{
      FileWriter  fw = new FileWriter("one.txt");
      fw.write("First file program to write something on it");
      fw.close();
      System.out.println("Successfully wrote to the file.");
    } catch (Exception e) {
      System.out.println("An error occurred.");
        }
  }
}
```

Read a File

```
import java.io.*;

import java.util.Scanner; // Import the Scanner class to read text files

public class readfile
 {
  public static void main(String[] args) {
    try {
      File f = new File("one.txt");
      Scanner s  = new Scanner(f);
      while (s.hasNextLine()) {
        String data = s.nextLine();
        System.out.println(data);
      }
      s.close();
    } catch (FileNotFoundException e) {
      System.out.println("An error occurred.");

    }
  }
}
```

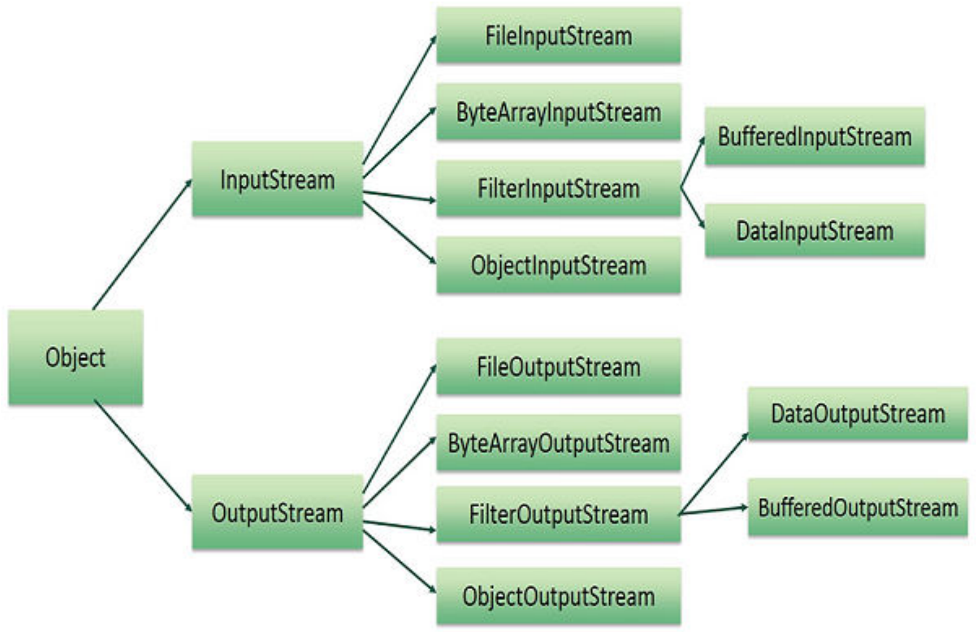
- There are many available classes in the Java API that can be used to read and write files in Java: FileReader, BufferedReader, Files, Scanner, FileInputStream, FileWriter, BufferedWriter, FileOutputStream

Implementation of …

- 1. Byte stream
- 2. Character stream
- 3. Buffered stream
- 4. Data stream
- 5. Object Stream
- 6. File stream

Byte stream in java

- Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file −



File input/output stream

- Java FileInputStream class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data. But, for reading streams of characters, it is recommended to use _______________class.
- Java FileOutputStream is an output stream used for writing data to a __________

FileOutputStream Methods

Method	Description
protected void finalize()	It is used 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[] ary, int off, int len)	It is used to write **len** 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.
FileChannel getChannel()	It is used to return the file channel object associated with the file output stream.
FileDescriptor getFD()	It is used to return the file descriptor associated with the stream.
void close()	It is used to closes the file output stream.

Sample program using FileOutputStream

```
import java.io.*;
class Simplewrite
{
public static void main(String ar[])
{
try
{
FileOutputStream f = new FileOutputStream (“abc.txt");
//f.write(45);
String s = "This the Fileoutputstream program";
byte b[] = s.getBytes(); // convert the string in to byte array
f.write(b);
f.close();
}catch(Exception e)
{
System.out.println(e);
}
}
}
```

Note: to check the output see that a file called abc.txt is created in the path where you executed this program

Java FileInputStream class methods

Method	Description
int available()	It is used to return the estimated number of bytes that can be read from the input stream.
int read()	It is used to read the byte of data from the input stream.
int read(byte[] b)	It is used to read up to **b.length** bytes of data from the input stream.
int read(byte[] b, int off, int len)	It is used to read up to **len** bytes of data from the input stream.
long skip(long x)	It is used to skip over and discards x bytes of data from the input stream.
FileChannel getChannel()	It is used to return the unique FileChannel object associated with the file input stream.
FileDescriptor getFD()	It is used to return the FileDescriptor object.
protected void finalize()	It is used to ensure that the close method is call when there is no more reference to the file input stream.
void close()	It is used to closes the stream.

Example of FileInputStream class

Java FileInputStream class obtains input bytes from a file FileOutputStream

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

```
import java.io.*;
public class filestream {

  public static void main(String args[]) {

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

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

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

ByteArray-Stream

- The ByteArrayInputStream class allows a buffer in the memory to be used as an InputStream. The input source is a byte array.
- The ByteArrayOutputStream class stream creates a buffer in memory and all the data sent to the stream is stored in the buffer.

Byte Array Input stream

- **public int read()**
- This method reads the next byte of data from the InputStream. Returns an int as the next byte of data. If it is the end of the file, then it returns -1.
- **public int read(byte[] r, int off, int len)**
- This method reads upto **len** number of bytes starting from **off** from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned.
- **public int available()**
- Gives the number of bytes that can be read from this file input stream. Returns an int that gives the number of bytes to be read.
- **public void mark(int read)**
- This sets the current marked position in the stream. The parameter gives the maximum limit of bytes that can be read before the marked position becomes invalid.
- **public long skip(long n)**
- Skips 'n' number of bytes from the stream. This returns the actual number of bytes skipped.

ByteArray Output Stream

- **public void reset()**
- This method resets the number of valid bytes of the byte array output stream to zero, so all the accumulated output in the stream will be discarded.
- **public byte[] toByteArray()**
- This method creates a newly allocated Byte array. Its size would be the current size of the output stream and the contents of the buffer will be copied into it. Returns the current contents of the output stream as a byte array.
- **public String toString()**
- Converts the buffer content into a string. Translation will be done according to the default character encoding. Returns the String translated from the buffer's content.
- **public void write(int w)**
- Writes the specified array to the output stream.
- **public void write(byte []b, int of, int len)**
- Writes len number of bytes starting from offset off to the stream.
- 6 **public void writeTo(OutputStream outSt)**
- Writes the entire content of this Stream to the specified stream argument.

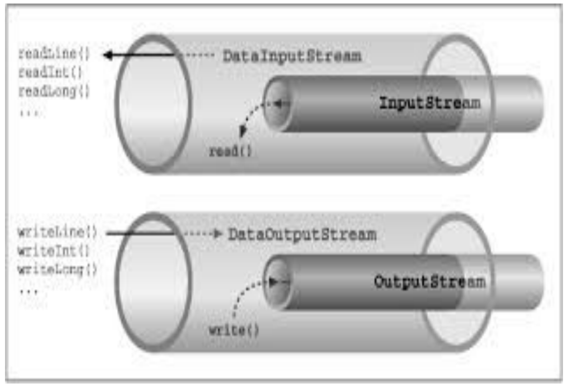
```
import java.io.*;
public class bytestream {

  public static void main(String args[])throws IOException {
    ByteArrayOutputStream b = new ByteArrayOutputStream(12);

    while( b.size()!= 10 ) {
      // Gets the inputs from the user
      b.write("hello".getBytes());
    }
    byte b1 [] = b.toByteArray();
    System.out.println("Print the content");

    for(int x = 0 ; x < b1.length; x++) {
      // printing the characters
      System.out.print((char)b1[x]  + "   ");
    }
    System.out.println("   ");

    int c;
    ByteArrayInputStream bInput = new ByteArrayInputStream(b1);
    System.out.println("Converting characters to Upper case " );

    for(int y = 0 ; y < 1; y++) {
      while(( c = bInput.read())!= -1) {
        System.out.println(Character.toUpperCase((char)c));
      }
```

FilterOutputStream Class

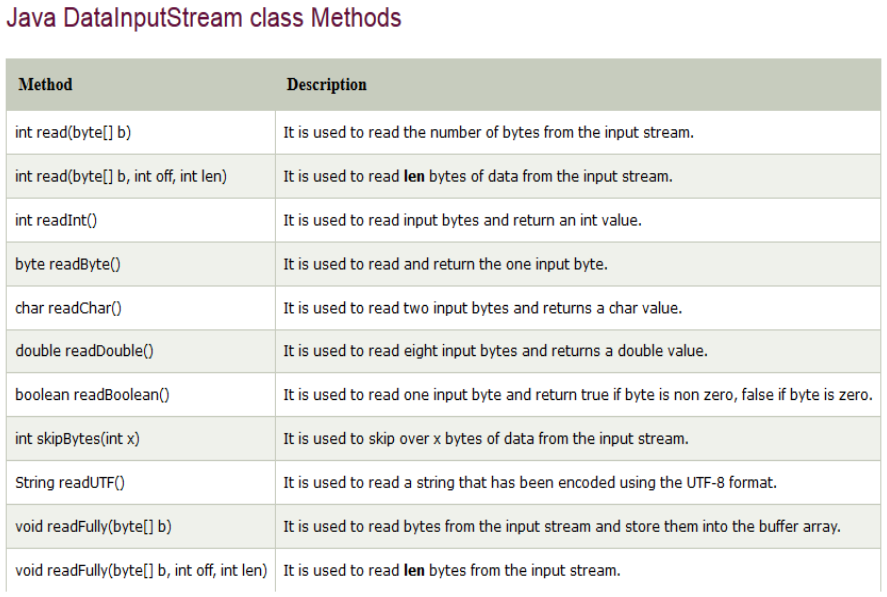
- Java FilterOutputStream class implements the OutputStream class. It provides different sub classes such as BufferedOutputStream and DataOutputStream to provide additional functiona
- Java FilterInputStream class implements the InputStream. It contains different sub classes as BufferedInputStream, DataInputStream for providing additional functionality. So it is less used individually.lity. So it is less used individually.

Data Input/output 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.



DataInputStream Class



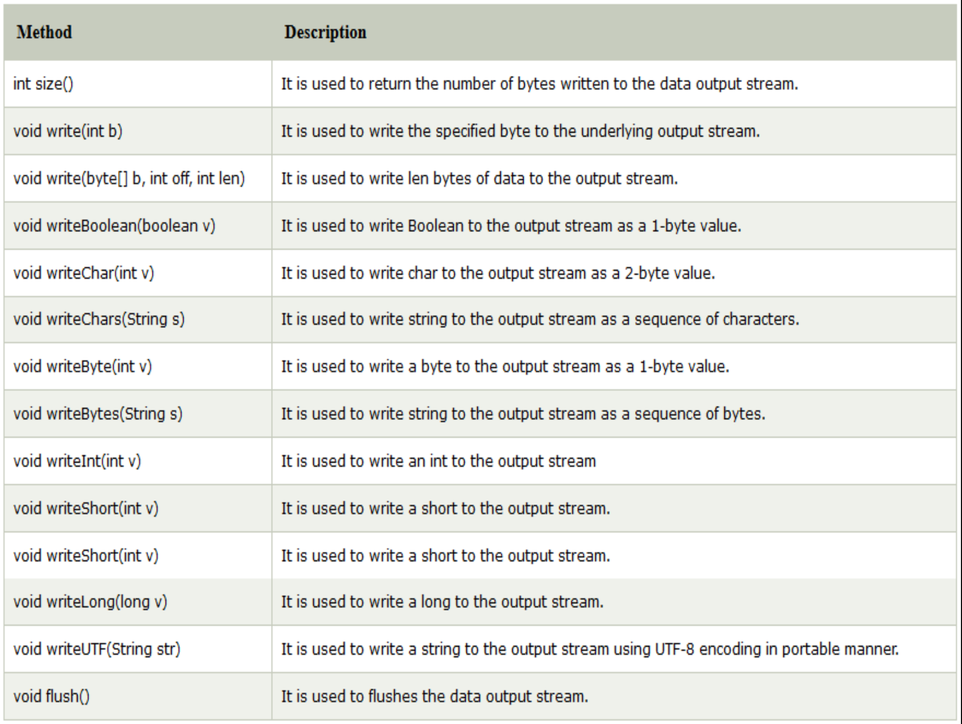
Java DataInputStream class Methods

Method	Description
int read(byte[] b)	It is used to read the number of bytes from the input stream.
int read(byte[] b, int off, int len)	It is used to read **len** bytes of data from the input stream.
int readInt()	It is used to read input bytes and return an int value.
byte readByte()	It is used to read and return the one input byte.
char readChar()	It is used to read two input bytes and returns a char value.
double readDouble()	It is used to read eight input bytes and returns a double value.
boolean readBoolean()	It is used to read one input byte and return true if byte is non zero, false if byte is zero.
int skipBytes(int x)	It is used to skip over x bytes of data from the input stream.
String readUTF()	It is used to read a string that has been encoded using the UTF-8 format.
void readFully(byte[] b)	It is used to read bytes from the input stream and store them into the buffer array.
void readFully(byte[] b, int off, int len)	It is used to read **len** bytes from the input stream.

```
import java.io.*;
class Datain
 {
  public static void main(String[] args) throws IOException
 {
    FileInputStream in = new FileInputStream("sample.txt");
    DataInputStream di = new DataInputStream(in);
    int count = in.available();
    byte[] ary = new byte[count];
    in.read(ary);
    for (byte bt : ary)
 {
     char k = (char) bt;
     System.out.print(k);
    }
  }
}
```

Java DataOutputStream Class

- Java DataOutputStream class allows an application to write primitive Java data types to the output stream in a machine-independent way.
- Java application generally uses the data output stream to write data that can later be read by a data input stream.



Method	Description
int size()	It is used to return the number of bytes written to the data output stream.
void write(int b)	It is used to write the specified byte to the underlying output stream.
void write(byte[] b, int off, int len)	It is used to write len bytes of data to the output stream.
void writeBoolean(boolean v)	It is used to write Boolean to the output stream as a 1-byte value.
void writeChar(int v)	It is used to write char to the output stream as a 2-byte value.
void writeChars(String s)	It is used to write string to the output stream as a sequence of characters.
void writeByte(int v)	It is used to write a byte to the output stream as a 1-byte value.
void writeBytes(String s)	It is used to write string to the output stream as a sequence of bytes.
void writeInt(int v)	It is used to write an int to the output stream
void writeShort(int v)	It is used to write a short to the output stream.
void writeShort(int v)	It is used to write a short to the output stream.
void writeLong(long v)	It is used to write a long to the output stream.
void writeUTF(String str)	It is used to write a string to the output stream using UTF-8 encoding in portable manner.
void flush()	It is used to flushes the data output stream.

```
import java.io.*;
public class Dataout
 {
   public static void main(String[] args) throws IOException
{
FileOutputStream f = new FileOutputStream("Sample.txt");
DataOutputStream data = new DataOutputStream(f);
      data.writeChars("This is Dataoutputstream

programming");
      data.flush();
      data.close();
      System.out.println("Success...");
   }
}
```

Buffered Stream

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

The important points about BufferedInputStream are:

- When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
- When a BufferedInputStream is created, an internal buffer array is created.

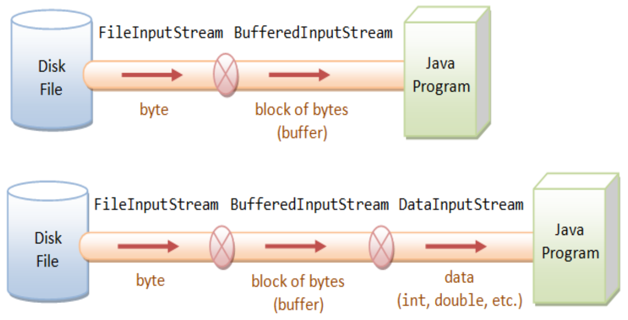
- **BufferedOutputStream**-Java BufferedOutputStream class is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

Java BufferedOutputStream Class

- Java BufferedOutputStream class is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

Java BufferedOutputStream class constructors

- **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.



- 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.

BufferedOutputStream Example

```
import java.io.*;
public class BOS
{
public static void main(String args[])throws Exception
{
    FileOutputStream f=new FileOutputStream("BOS.txt");
    BufferedOutputStream bout=new BufferedOutputStream(f);
    String s="Welcome to Buffered Reader Programming.";
    byte b[]=s.getBytes();
    bout.write(b);
    bout.flush();
    bout.close();
    f.close();
    System.out.println("success");
}
}
```

BufferedInputStream

- Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.
- The important points about BufferedInputStream are:
- When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
- When a BufferedInputStream is created, an internal buffer array is created.

- 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 of Buffered Input Stream

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.

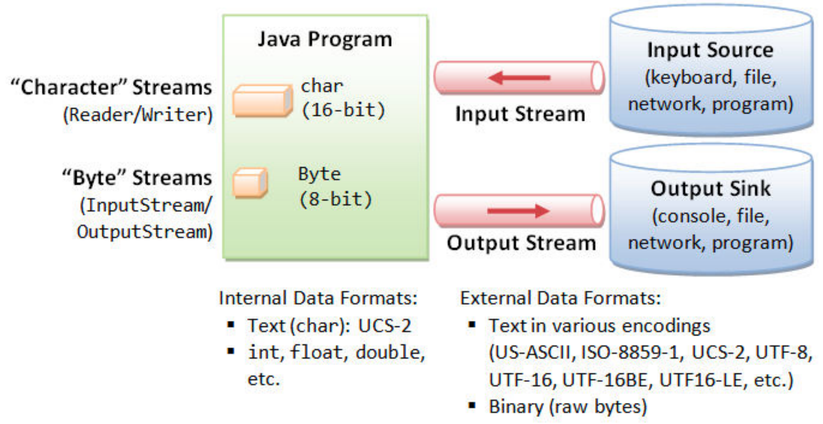
Mark and reset method in BufferedInputStream

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

public class bufferedinput{
  public static void main(String[] args) throws Exception {
    InputStream iStream = null;
    BufferedInputStream bis = null;

    try {
      // read from file c:/test.txt to input stream
      iStream = new FileInputStream("c:/users/admin/
     desktop/java/one.txt");

      // input stream converted to buffered input stream
      bis = new BufferedInputStream(iStream);

      // read and print characters one by one
      System.out.println("Char : "+(char)bis.read());
      System.out.println("Char : "+(char)bis.read());
 // mark is set on the input stream
        bis.mark(0);
        System.out.println("Char : "+
(char)bis.read());
        System.out.println("reset() invoked");

        // reset is called
        bis.reset();

        // read and print characters
        System.out.println("char : "+
(char)bis.read());
        System.out.println("char : "+
(char)bis.read());

     } catch(Exception e)
{System.out.println(e);}
        }
}
```

```
import java.io.*;
public class BufferedInputStreamExample{
 public static void main(String args[]){
  try{
    FileInputStream fin=new FileInputStream("c:/users/admin/desktop/java/
one.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);}
 }
}
```

Character streams

- Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.



Java FileWriter and FileReader

- Java FileWriter and FileReader classes are used to write and read data from text files. These are character-oriented classes, used for file handling in java.
- Java has suggested not to use the FileInputStream and FileOutputStream classes if you have to read and write the textual information.

Character stream example

```
• import java.io.*;
• public class charstream {

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

• IOException {
•     FileReader in = null;
•     FileWriter out = null;

•     try {
•       in = new
  FileReader("Sample.txt");
•       out = new
  FileWriter("Sample2.txt");
•
```

```
int c;
     while ((c = in.read()) !=
-1) {
       out.write(c);
     }
   }finally {
     if (in != null) {
       in.close();
     }
     if (out != null) {
       out.close();
     }
   }
  }
}
```

Buffered Reader/Writer class

- **Java.io.BufferedReader** class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
- Following are the important points about BufferedReader −
- The buffer size may be specified, or the default size may be used.
- Each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream.

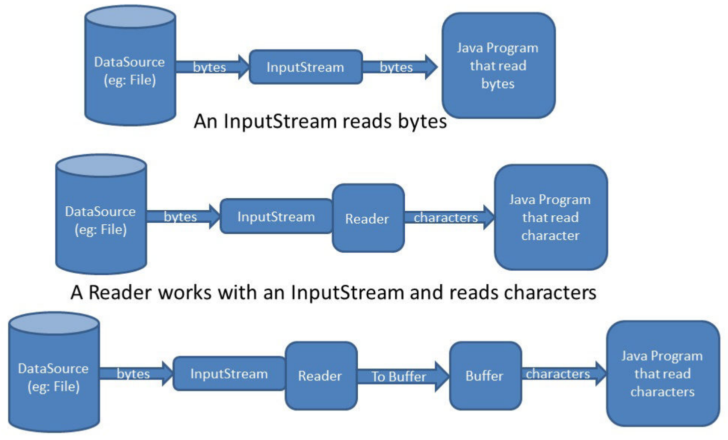
- The **Java.io.BufferedWriter** class writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.

Following are the important points about BufferedWriter −

- The buffer size may be specified, or the default size may be used.
- A Writer sends its output immediately to the underlying character or byte stream.

Java 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.
- **Java BufferedReader class constructors**
- BufferedReader(Reader rd) - It is used to create a buffered character input stream that uses the default size for an input buffer.
- BufferedReader(Reader rd, int size)- It is used to create a buffered character input stream that uses the specified size for an input buffer.

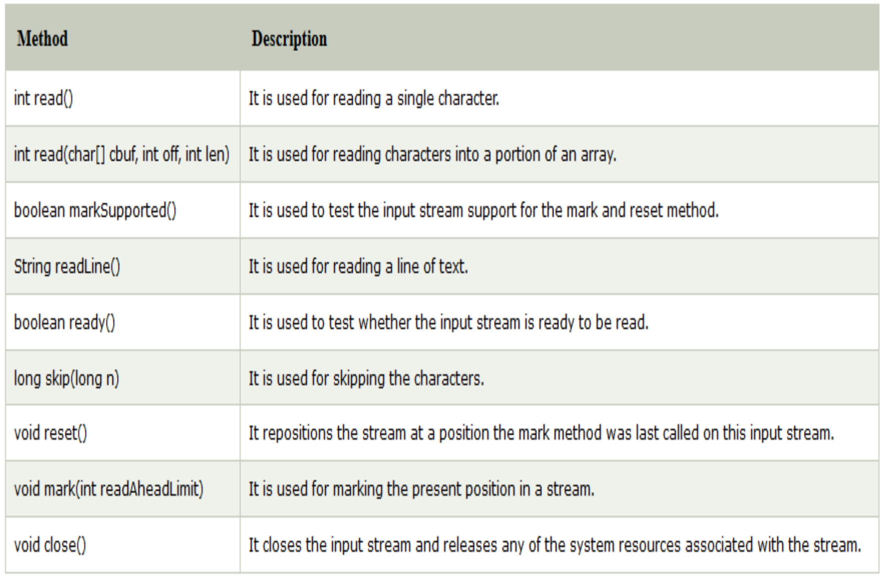


An InputStream reads bytes

A Reader works with an InputStream and reads characters

A BufferedReader pre-fetches characters in to a buffer for performance

Java BufferedReader class methods



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

```
• import java.io.*;
• public class BuffRead
• {
• public static void main(String args[])throws Exception
• {
•   InputStreamReader r=new InputStreamReader (System.in);
•   BufferedReader br=new BufferedReader(r);
•    System.out.println("Enter your Regno and  name");
•    String name=br.readLine();
•    int no = Integer.parseInt(br.readLine());
•    System.out.println(" Hello "+name + " " +no);
• }
• }
```

Buffered Writer

```
import java.io.*;
public class bufferedwriter
 {
public static void main(String[] args) throws Exception
{
    FileWriter writer = new FileWriter

("c://users/admin/desktop/java/one.txt");
    BufferedWriter buffer = new BufferedWriter(writer);
    buffer.write("Welcome to java File handling

programming.");
    buffer.close();
    System.out.println("Success");
    }
}
```

- Similarly go through the rest all Classes such as
- CharArrayReader
- CharArrayWriter
- PrintStream
- PrintWriter
- OutputStreamWriter
- InputStreamReader
- PushbackInputStream
- PushbackReader
- StringWriter
- StringReader
- PipedWriter
- PipedReader
- FilterWriter
- FilterReader

Serialization and Deserialization in Java

- **Serialization in Java** is a mechanism of *writing the state of an object into a byte stream*.
- It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.
- The reverse operation of serialization is called *deserialization*.
- **Advantages of Java Serialization**
- It is mainly used to travel object's state on the network (which is known as marshaling).

java.io.Serializable interface

- Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that objects of these classes may get the certain capability.

```
import java.io.Serializable;
public class Student implements Serializable{
 int id;
 String name;
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}
```

```
import java.io.*;
class Student implements Serializable{
int id;
String name;
public Student(int id, String name) {
 this.id = id;
 this.name = name;
 }
}
   class serial
{
    public static void main(String args[])throws Exception
{
     Student s1 =new Student(211,"ravi");
    FileOutputStream fout=new FileOutputStream("f.txt");
     ObjectOutputStream out=new ObjectOutputStream(fout);
     out.writeObject(s1);
     out.flush();
     System.out.println("success");
    }
   }
```

Deserialization in java

- Deserialization is the process of reconstructing the object from the serialized state.It is the reverse operation of serialization.

```
import java.io.*;
class Student implements Serializable{
 int id;
 String name;
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}
   class deserial
{
    public static void main(String args[])throws Exception
{
     ObjectInputStream in=new ObjectInputStream(new

      FileInputStream("f.txt"));
     Student s=(Student)in.readObject();
     System.out.println(s.id+" "+s.name);
      in.close();
    }
   }
```