The stream method helps to **sequentially access** a file. There are two types of streams in Java- Byte Stream and Character Stream.

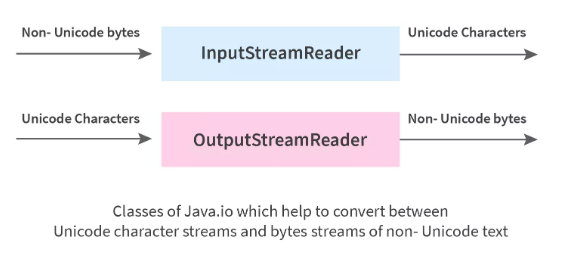
Byte streams in Java are used to perform input and output operations of 8-bit bytes while the Character stream is used to perform input and output operations for 16-bits Unicode.

Character streams are useful in reading or writing text files which are processed character by character. Byte Streams are useful to read/write data from raw binary files.

An I/O (Input/Output) stream is used to represent an **input source** or an **output destination**. It can represent many kinds of sources and destinations like disc files (systems that manage data on permanent storage devices eg. hard disc or magnetic disc) or devices.

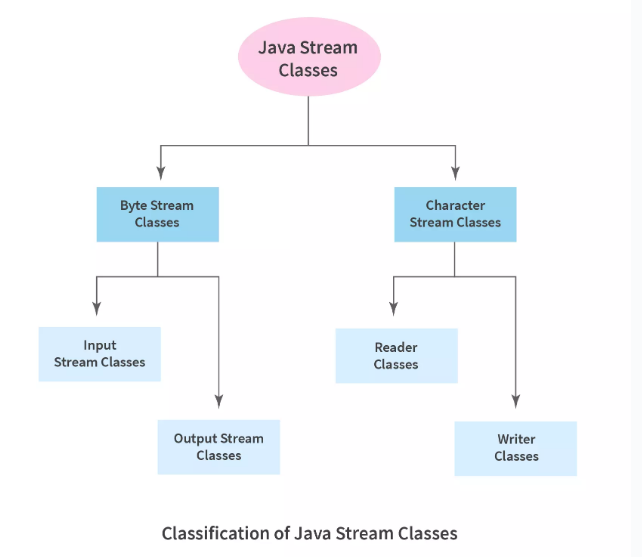
The stream method helps to **sequentially access** a file. Some streams simply pass on the data while some manipulate and transform the data in a useful way. For example, some streams copy the contents of the file to another and do not modify them or some streams perform manipulations on them like adding or filtering data etc. Streams support many kinds of data including bytes, primitive data types, characters and objects.

The java.io package contains classes that allow the user to convert between Unicode character streams and byte streams of non-Unicode text.



Let us understand some terminologies associated with the same.

* **Stream-** It is a sequence of data/objects that supports various methods. They are used to read or write data. Eg- files, input-output devices etc.
* **Input Stream Reader-** It reads data from a source, one item at a time. An InputStreamReader class is a bridge from byte streams to character streams. It reads bytes and decodes them into characters.
* **Output Stream Writer-** It writes data to a destination, one item at a time. An OutputStreamWriter class is a bridge from character streams to byte streams. Characters written to it are encoded into bytes.
* **Unicode** is an international character encoding standard by which each letter, digit or symbol is assigned a unique numeric value across all platforms.
* **Non-Unicode text** are modules or character encodings that do not support Unicode standards. It only supports English language representation.

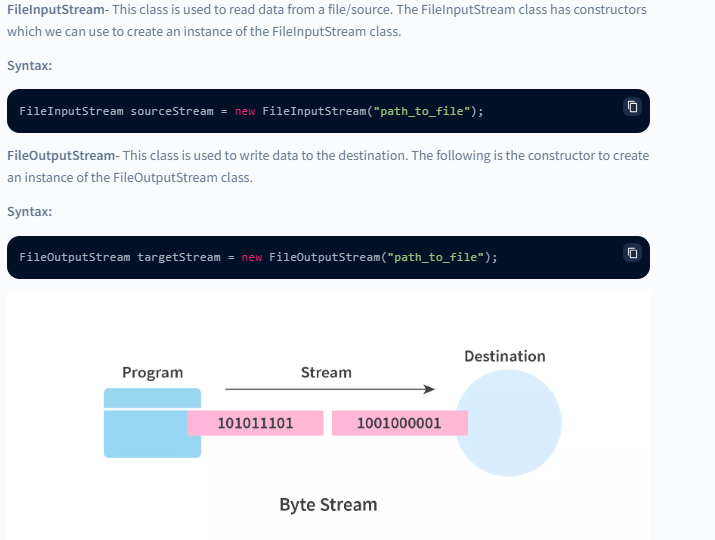


**What is Byte Stream in Java?**

Byte streams are used to perform **input** and **output** of **8-bit bytes**. They are used to read bytes from the input stream and write bytes to the output stream. Mostly, they are used to read or write raw binary data.

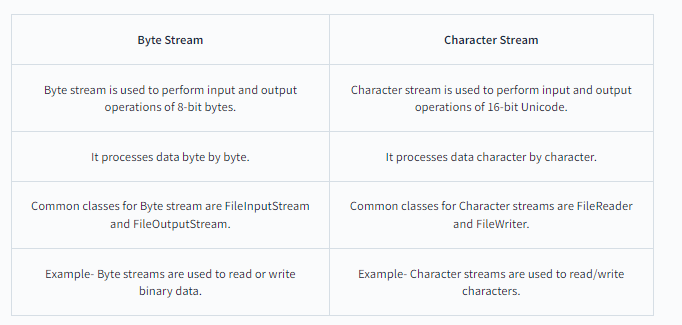
**In Java, the byte streams have a 3 phase mechanism:**

* **Split-** The input data source is split into a stream by a spliterator. Java Spliterator interface is an internal iterator that breaks the stream into smaller parts for traversing over them.
* **Apply-** The elements in the stream are processed.
* **Combine-** After the elements are processed, they are again combined together to create a single result.



The above image shows an example of the byte stream. From a program, the bytes are being transferred in the form of a stream to the destination.





**When to use Byte Stream over Character Stream?**

Byte streams are used to process raw data like binary files. If we have a file that contains binary data, then it will be appropriate to use Byte stream. They can be used to read/write data of 8-bit bytes.

Example- Byte streams are used to read or write binary data.

**Benefits of Byte Stream**

* Byte stream provides a convenient way to handle the input and output of bytes. If your file is too large, byte stream handles data in chunks rather than having the entire data altogether in the memory.
* They are useful when we want to read/write binary data.

**Benefits of Character Stream**

* Character stream provides convenient means to handle character-based inputs and outputs.
* Since they use Unicode, Character streams can be internationalized. Internationalization is the process of preparing an application that supports linguistic, regional, cultural or political-specific data.
* In some cases, character streams are more efficient than byte streams especially when the file contains characters.
* Character streams automatically translate the internal format of the file (the content of the file) to and from the local character set without extra effort by the programmer.
* In short, character streams make it easy to write programs that are not dependent upon a specific character encoding, which becomes easy to internationalize.

The **java.io** package is used to handle input and output operations. Java IO has various classes that handle input and output sources. A stream is a sequence of data.

Java input stream classes can be used to read data from input sources such as **keyboard** or a **file**. Similarly output stream classes can be used to write data on a **display** or a **file** again.

We can also perform **File Handling** using **Java IO API**.

The **java.io** package is used to handle input and output operations. Java IO has various classes that handle input and output sources. A stream is a sequence of data.

Java input stream classes can be used to read data from input sources such as **keyboard** or a **file**. Similarly output stream classes can be used to write data on a **display** or a **file** again.

We can also perform **File Handling** using **Java IO API**.

**What is Java IO?**

The **java.io** package consists of input and output streams used to read and write data to files or other input and output sources.

There are 3 categories of classes in java.io package:

* Input Streams.
* Output Streams.
* Error Streams.

Java supports three streams that are automatically attached with the console.

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

### Input Streams

As we know input source consists of data that needs to be read in order to extract information from it. Input Streams help us to read data from the input source. It is an abstract class that provides a programming interface for all input streams.

Input streams are opened implicitly as soon as it is created. To close the input stream, we use a close() method on the source object.

### Output Streams

The output of the executed program has to be stored in a file for further use. Output streams help us to write data to a output source(may be file). Similarly like input streams output streams are also abstract classes that provides a programming interface for all output streams.

The output stream is opened as soon as it is created and explicitly closed by using the close() method.

### Error Streams

Error streams are the same as output streams. In some ide’s error is displayed in different colors (other than the color of output color). It gives output on the console the same as output streams.

## Why We Need IO Streams in Java?

In day-to-day work, we do not enter the input into the programs manually. Also, the result of the program needs to be stored somewhere for further use.

So, IO streams in Java provide us with input and output streams that help us to extract data from the files and write the data into the files. Normally, we can create, delete, and edit files using Java.io.

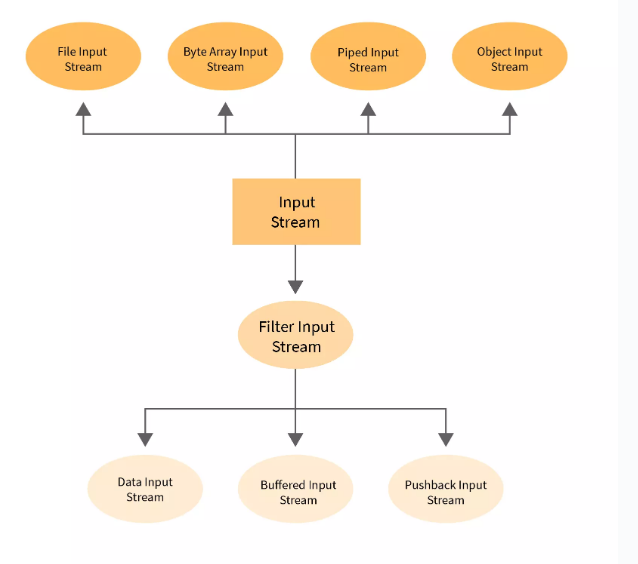
In short, all the file manipulation is done using Java IO streams. Java IO streams also handle user input functionality.

## Types of Streams in Java

Depending on the types of operations, streams are divided into 2 primary classes.

### Input Stream

It is an abstract superclass of the **java.io** package and is used to read the data from an input source. In other words, reading data from files or from a keyboard, etc. We can create an object of the input stream class using the new keyword. The input stream class has several types of constructors.



**BufferedInputStream**

BufferedInputStream is buffered.

"BufferedInputStream reads bytes from another InputStream (Eg - FileInputStream). So, BufferedInputStream is wrapper formed on FileInputStream.

**FileInputStream fis = new FileInputStream(""c:/myFile.txt"");**

**BufferedInputStream bis = new BufferedInputStream(fis);"**

when BufferedInputStream.read() is called mostly data is read from the buffer. When data is not available available in buffer a call is made to read system file and lot of bytes are kept in buffer.

BufferedInputStream.readLine() method reads whole line and keep it in buffer.A BufferedInputStream enables another input stream to buffer the input and supports the mark and reset methods.

An internal buffer array is created when the BufferedInputStream is created. As bytes from the stream are read or skipped, the internal buffer is refilled as necessary from the contained input stream, many bytes at a time. BufferedInputStream is much faster as compared to FileInputStream.

Example -

As we discussed above that when BufferedInputStream.read() is called mostly data is read from the buffer. A BufferedInputStream reads from FileInputStream, will request lot of data from the FileInputStream (128 bytes or so… not exact figure). Thus only 2 calls will be made for reading 256 bytes from file.

Another Example - Real world Example - You must have seen youtube videos where video is buffered before you actually start watching it, buffering overall improves your video watching experience.

The **java.io.InputStream.mark(int readlimit)** method marks the current position in this input stream. A subsequent invocation to the reset() method reposition the stream to the recently marked position.

**readlimit** − The maximum number of bytes that can be read before the mark position becomes invalid.

Eg - Let's say we are reading 'abcd'

read 'a',

mark() 'b',

read 'b',

read 'c',

call reset()

now we will start reading from 'b'

When the BufferedInputStream is created, an internal buffer array is created. As bytes from the stream are read or skipped, the internal buffer is refilled as necessary from the contained input stream, many bytes at a time. The mark operation remembers a point in the input stream and the reset operation causes all the bytes read since the most recent mark operation to be reread before new bytes are taken from the contained input stream.

**FileInputStream**

FileInputStream is not buffered. "FileInputStream reads bytes from a file.

**FileInputStream fis = new FileInputStream(""c:/myFile.txt"");"**

**Every time FileInputStream.read() is called a call is made to read a system file.**

**FileInputStream.read()**

reads 1 byte (8-bit) at a time. A FileInputStream obtains input bytes from a file in a file system.

And does not supports mark and reset methods. FileInputStream is slower as compared to BufferedInputStream.

Example -

As we discussed in point above that every time FileInputStream.read() is called a call is made to read a system file.

A FileInputStream will make 256 calls for reading 256 bytes from file.

No buffering will make your videos watching experience a nightmare.

Character Stream

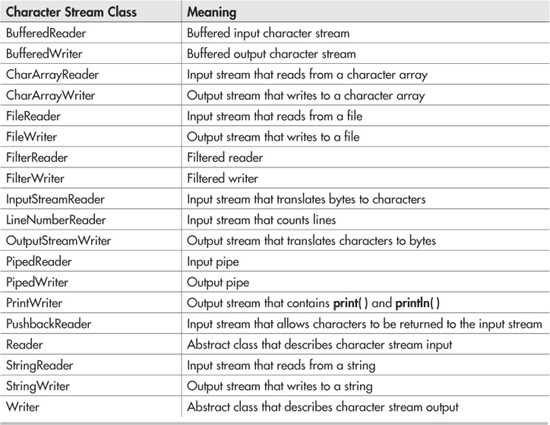
In java, when the IO stream manages 16-bit Unicode characters, it is called a character stream. The unicode set is basically a type of character set where each character corresponds to a specific numeric value within the given character set, and every programming language has a character set.

In java, the character stream is a 16 bits carrier. The character stream in java allows us to transmit 16 bits of data.

The java character stream is defined by two abstract classes, **Reader** and **Writer**. The Reader class used for character stream based input operations, and the Writer class used for charater stream based output operations.

Character streams are defined by using two class hierarchies topped by these two abstract classes: **Reader** and **Writer**. **Reader** is used for input, and **Writer** is used for output. Concrete classes derived from **Reader** and **Writer** operate on Unicode character streams.

From **Reader** and **Writer** are derived several concrete subclasses that handle various I/O situations. In general, the character-based classes parallel the byte-based classes.



Java FileWriter and FileReader classes are used to write and read data from text files (they are [Character Stream](https://www.geeksforgeeks.org/character-stream-vs-byte-stream-java/) classes). It is recommended **not** to use the FileInputStream and FileOutputStream classes if you have to read and write any textual information as these are Byte stream classes.

**FileWriter**  
FileWriter is useful to create a file writing characters into it.

* This class inherits from the OutputStream class.
* The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream.
* FileWriter is meant for writing streams of characters. For writing streams of raw bytes, consider using a FileOutputStream.
* FileWriter creates the output file if it is not present already.

**Constructors:**

* **FileWriter(File file) –** Constructs a FileWriter object given a File object.
* **FileWriter (File file, boolean append) –** constructs a FileWriter object given a File object.
* **FileWriter (FileDescriptor fd) –** constructs a FileWriter object associated with a file descriptor.
* **FileWriter (String fileName) –** constructs a FileWriter object given a file name.
* **FileWriter (String fileName, Boolean append) –** Constructs a FileWriter object given a file name with a Boolean indicating
* **public void write (int c) throws IOException –** Writes a single character.
* **public void write (char [] stir) throws IOException –** Writes an array of characters.
* **public void write(String str)throws IOException –** Writes a string.
* **public void write(String str,** **int off,** **int len)throws IOException –** Writes a portion of a string. Here off is offset from which to start writing characters and len is the number of characters to write.
* **public void flush() throws IOException** flushes the stream
* **public void close() throws IOException** flushes the stream first and then closes the writer.

[BufferedReader](https://www.geeksforgeeks.org/java-io-bufferedreader-class-java/) and FileReader both classes are used to read data from a given character stream. Both of them have some pros and some cons. In this article, we will discuss the differences between them. Though the most important difference is in the way they work, but we will discuss the other details also.

**What is a Buffer?**

A buffer is a small portion of the device memory storage used to temporarily store some amount of data. Usually, Buffers use the RAM of the device to store the temporary data, and hence, accessing data from the buffer is much faster than accessing the same amount of data from the hard drive.

Differences between BufferedReader and FileReader are illustrated over and discussed over major parameters taken into consideration as follows:

1. Usage
2. Efficiency
3. Speed
4. Reading lines

**1. Usage**

FileReader is used to read a file from a disk drive whereas BufferedReader is not bound to only reading files. It can be used to read data from any character stream. FileReader class provides two constructors:

* FileReader(File file): It takes a File object that represents a file in your disk and creates a new FileReader instance.
* FileReader(FileDescriptor fd) : Creates a new FileReader, given the FileDescriptor to read from.
* FileReader(String fileName): Takes the name of the file as the only parameter and creates a new FileReader instance to read the file.

BufferedReader class provides two constructors:

* BufferedReader(Reader rd): It uses a Reader to read data from the character input stream and creates a default sized input buffer.
* BufferedReader(Reader rd, int size): Takes two parameters:
  + First: A Reader that is used to read the input stream data
  + Second: The size of the input buffer. It creates a new BufferedReader with the given sized input buffer.

As seen, BufferedReader accepts any type of [Reader](https://www.geeksforgeeks.org/java-io-reader-class-java/)([StringReader](https://www.geeksforgeeks.org/java-io-stringreader-class-java/), FileReader, etc.) and hence capable of reading from any character input stream. Whereas, FileReader is capable of reading characters from files only. Usually, we wrap a FileReader with BufferedReader to read characters from files.

**2. Efficiency**

BufferedReader is much more efficient than FileReader in terms of performance. FileReader directly reads the data from the character stream that originates from a file. Each time, it reads a character, it directly accesses the disk drive and every time it takes some time for the disk drive to position the read head correctly which makes it very inefficient.

Whereas BufferedReader creates an input buffer and allows the input to be read from the hard drive in large chunks of data rather than a byte at a time, resulting in a huge improvement in performance. The default buffer size is 8Kb( which is enough in most cases) though it can be customized. BufferedReader reads lots of data at a time and stores it in the created buffer memory. When [java.io.BufferedReader#read()](https://www.geeksforgeeks.org/bufferedreader-read-method-in-java-with-examples/) is called, it reads the data from the memory buffer. When data is not available in the buffer, it makes a corresponding read request of the underlying character stream and loads lots of data into the created buffer. As a result, we do not have to access the hard drive directly when reading each character, we can read from the buffer memory which is fast and much more efficient.

**Reading Lines**

In most cases, you would like to read a line at a time rather than reading a character at a time and only the BufferedReader provides a *[readLine()](https://www.geeksforgeeks.org/bufferedreader-readline-method-in-java-with-examples/)* method that reads a whole line at a time. Simply, the given Reader(FileReader in this case) reads the characters and stores them in the buffer. When the [java.io.BufferedReader#readLine()](https://www.geeksforgeeks.org/bufferedreader-readline-method-in-java-with-examples/) method is called, characters of a line stored in the buffer, are returned as a String. It saves lots of time and hence is faster than FileReader#read() method. Note that, BufferedReader is able to read a whole line at a time only because it uses a buffer memory, it can store the characters of a line in the buffer and read all the characters together directly from the buffer.

**Serialization**

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

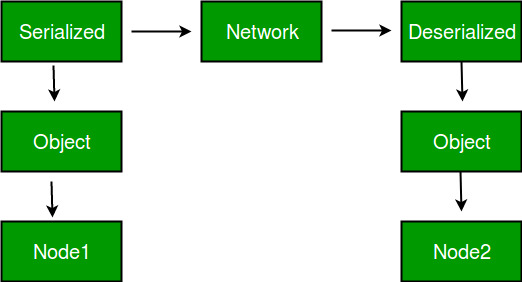
Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform. To make a Java object serializable we implement the **java.io.Serializable** interface. The ObjectOutputStream class contains **writeObject()** method for serializing an Object.



**Advantages of Serialization**

1. To save/persist state of an object.
2. To travel an object across a network.



Only the objects of those classes can be serialized which are implementing **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 certain capability. Other examples of marker interfaces are:- Cloneable and Remote.

**Points to remember**

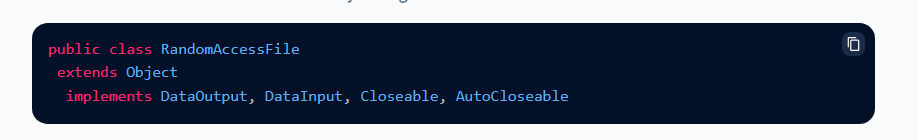
1. If a parent class has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true. 2. Only non-static data members are saved via Serialization process.   
3. Static data members and transient data members are not saved via Serialization process. So, if you don’t want to save value of a non-static data member then make it transient.   
4. Constructor of object is never called when an object is deserialized.   
5. Associated objects must be implementing Serializable interface. Example

**SerialVersionUID** The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and receiver of a serialized object have loaded classes for that object which are compatible with respect to serialization. If the receiver has loaded a class for the object that has different UID than that of corresponding sender’s class, the Deserialization will result in an **InvalidClassException**.

When we want a class to be serializable and all the other classes that we use in the class should also be serializable

**RandomAccessFile in java** is a class that lets the user read and write to a file at the same time. It extends the Object class and implements DataInput and DataOutput interfaces. These **interfaces** facilitate the conversion of primitive type data to a sequence of bytes and bytes to specified type data. It is used to read and write data to a file simultaneously. The file acts as a large array of bytes stored in the file system.

Methods of this class usually throw EOFException i.e. **End of File Exception** if the end of the file is reached before the desired number of bytes are read. It is a type of IOException.



DataOutput and DataInput interfaces provide the functionality to convert data from any of the Java primitive data types to a series of bytes, writing these bytes to a binary stream and vice versa.

**Closeable** and **AutoCloseable** interfaces are used to release the resources at the end of the execution of the program.

### File Pointer in RandomAccessFile Class

A file behaves like a large array of bytes, thus we use a cursor or index for the array called the file pointer. Data is read byte by byte starting at the file pointer and the write operation is performed starting at the file pointer too. File pointer can be moved or retrieved using seek() and getFilePointer() methods.

### File Pointer in RandomAccessFile Class

A file behaves like a large array of bytes, thus we use a cursor or index for the array called the file pointer. Data is read byte by byte starting at the file pointer and the write operation is performed starting at the file pointer too. File pointer can be moved or retrieved using seek() and getFilePointer() methods.

### Constructors of RandomAccessFile Class

**The RandomAccessFile class in Java has the following two constructors :**

1. **RandomAccessFile(File fileObj, String mode) :**  
   It creates a random access file stream to read, write, or execute the specified File Object.
2. **RandomAccessFile(String fileName, String mode) :**  
   It creates a random access file stream to read, write, or execute the file with the specified name.



In the example above, we are creating an object of the RandomAcessFile class. It takes two parameters the file along with its location and the mode in which we wish to open the file. The mode can read, write or execute. We are moving the pointer 4 bytes ahead using the seek() method. We read 12 bytes of data from the file pointer into bytes and print it in the output by converting it into the string.

**Conclusion**

* Random access file in java is used to read and write files in java programs. It is different from the File class as it allows one to perform reading and writing operations simultaneously.
* It extends Object class and implements **DataInput** and **DataOutput interfaces**.
* **File pointer** acts as an index or cursor for the file where the file is a large array of bytes.
* RandomAccessFile class provides various useful inbuilt methods to perform operations on the file.
* Some of the important methods are seek(), getFilePointer(), length(), read(), write(), readFully(), etc.

Path API over File API

JAVA NIO