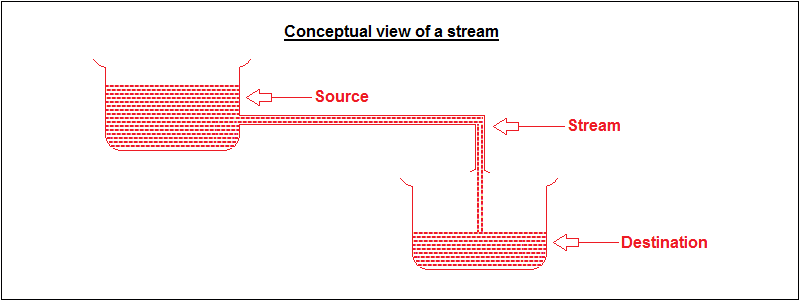
**Stream Classes**

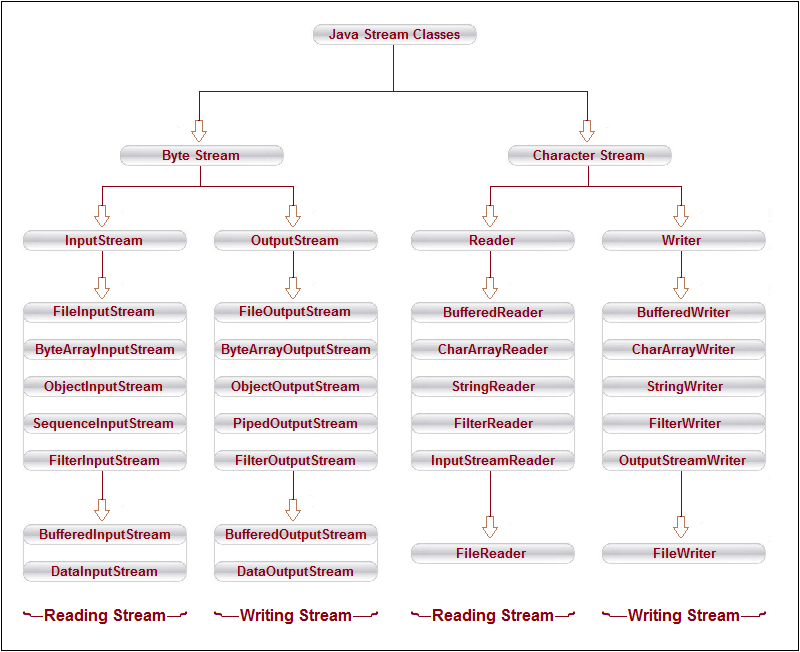
In Java, a **stream** is a **path** along which the **data flows**. Every stream has a **source** and a **destination**. We can build a complex file processing sequence using a series of simple stream operations. Two fundamental types of streams are **Writing streams** and **Reading streams**. While an **Writing streams** writes data into a **source**(file) , an **Reading streams** is used to read data from a **source**(file).

[](http://hajsoftutorial.com/java/wp-content/uploads/2015/10/j.gif)

**Types of Streams**

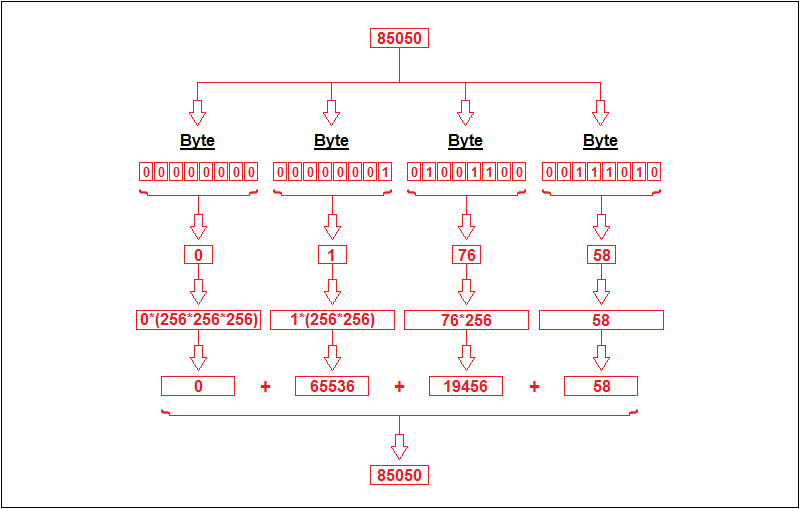
The java.io package contains a large number of stream classes that provide capabilities for processing all types of data. These classes may be categorized into two groups based on the data type on which they operate.

* **Byte stream classes**
* **Character stream classes**

[](http://hajsoftutorial.com/java/wp-content/uploads/2015/10/gu-13.png)

**Text and Binary Formats of Data**

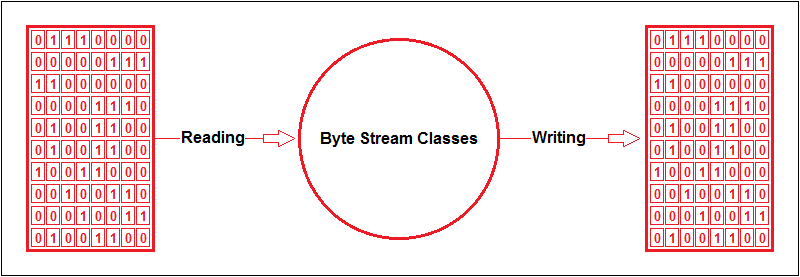
There are two fundamentally different ways to store data. They are **text format** and **binary format**. Data items are stored in human-readable form in the text format. For example, the data item **85,050** is stored as the sequence of the five characters **‘8’ ‘5’ ‘0’ ‘5’ ‘0’**. Data items are represented in bytes in the case of binary format. As a byte consists of **8** bits, it can represent one of **256** values. Since the data item **85,050** can be represented as the sequence of the four bytes **0, 1, 76, 58**.

[](http://hajsoftutorial.com/java/wp-content/uploads/2015/10/15.png)

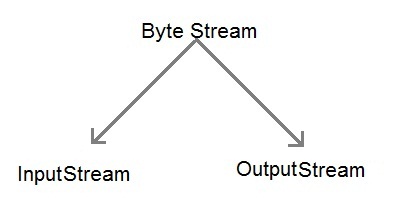
 If data items are available in **text format**, we have to use the **Character stream classes** to process the input and output. If the data items are made available in **binary format**, we have to use the **Byte stream classes**. While the text format is convenient for humans, storage of data in binary format is more compact and more efficient.

**Byte Stream Classes**

Byte stream classes have been designed to provide functional features for creating and manipulating streams and files for reading and writing bytes. Java provides two kinds of byte stream classes: **input stream classes** and **output stream classes**.

[](http://hajsoftutorial.com/java/wp-content/uploads/2015/10/Untitled3.png)

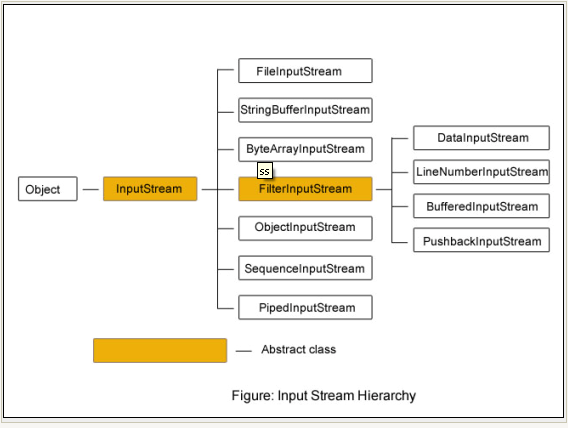
Byte stream is defined by using two abstract class at the top of hierarchy, they are InputStream and OutputStream.



These two abstract classes have several concrete classes that handle various devices such as disk files, network connection etc.

1. **Input Stream Classes**

Input stream classes that are used to read bytes include a super class known as **Inputstream** and a number of subclasses for supporting various input-related functions. The super class InputStream is an **abstract** class, and, therefore, we cannot create instances of this class. Rather, we must use the subclasses that inherit from this class.

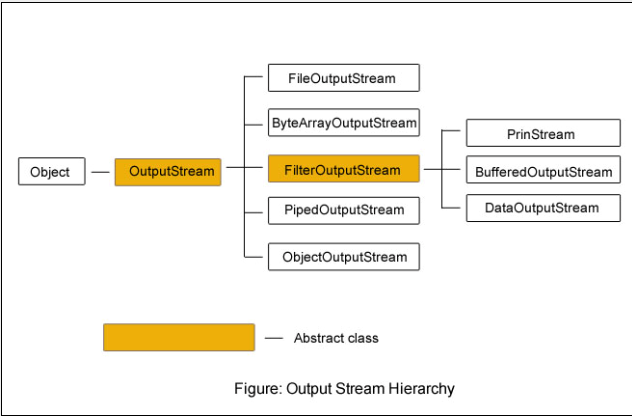


**Methods declared in InputStream and implemented in all lsubclasses of InputStream**

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| int | [**available**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayInputStream.html#available())()  Returns the number of remaining bytes that can be read (or skipped over) from this input  stream. |
| void | [**close**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayInputStream.html#close())()  Closing a ByteArrayInputStream has no effect. |
| void | [**mark**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayInputStream.html#mark(int))(int readAheadLimit)  Set the current marked position in the stream. |
| boolean | [**markSupported**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayInputStream.html#markSupported())()  Tests if this InputStream supports mark/reset. |
| int | [**read**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayInputStream.html#read())()  Reads the next byte of data from this input stream. |
| int | [**read**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayInputStream.html#read(byte[],%20int,%20int))(byte[] b, int off, int len)  Reads up to len bytes of data into an array of bytes from this input stream. |
| void | [**reset**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayInputStream.html#reset())()  Resets the buffer to the marked position. |
| long | [**skip**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayInputStream.html#skip(long))(long n)  Skips n bytes of input from this input stream. |

1. **Output Stream Classes**

Output stream classes are derived from the base class **Outputstream** like InputStream, the OutputStream is an **abstract** class and therefore we cannot instantiate it. The several subclasses of the OutputStream can be used for performing the output operations.



**Methods declared in OutputStream and implemented in all lsubclasses of OutputStream**

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| void | [**close**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayOutputStream.html#close())()  Closing a ByteArrayOutputStream has no effect. |
| void | [**reset**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayOutputStream.html#reset())()  Resets the count field of this byte array output stream to zero, so that all currently  accumulated output in the output stream is discarded. |
| int | [**size**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayOutputStream.html#size())()  Returns the current size of the buffer. |
| byte[] | [**toByteArray**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayOutputStream.html#toByteArray())()  Creates a newly allocated byte array. |
| [**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) | [**toString**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayOutputStream.html#toString())()  Converts the buffer's contents into a string decoding bytes using the platform's  default character set. |
| [**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) | [**toString**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayOutputStream.html#toString(int))(int hibyte)  **Deprecated.**  *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred*  *way to do this is via the toString(String enc) method, which takes an*  *encoding-name argument, or the toString() method, which uses the platform's*  *default character encoding.* |
| [**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) | [**toString**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayOutputStream.html#toString(java.lang.String))([**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) charsetName)  Converts the buffer's contents into a string by decoding the bytes using the specified  [**charsetName**](https://docs.oracle.com/javase/7/docs/api/java/nio/charset/Charset.html). |
| void | [**write**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayOutputStream.html#write(byte[],%20int,%20int))(byte[] b, int off, int len)  Writes len bytes from the specified byte array starting at offset off to this byte array  output stream. |
| void | [**write**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayOutputStream.html#write(int))(int b)  Writes the specified byte to this byte array output stream. |
| void | [**writeTo**](https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayOutputStream.html#writeTo(java.io.OutputStream))([**OutputStream**](https://docs.oracle.com/javase/7/docs/api/java/io/OutputStream.html) out)  Writes the complete contents of this byte array output stream to the specified  output stream argument, as if by calling the output stream's write method using  out.write(buf, 0, count). |

**Some important Byte stream classes.**

|  |  |
| --- | --- |
| **Stream class** | **Description** |
| **BufferedInputStream** | Used for Buffered Input Stream. |
| **BufferedOutputStream** | Used for Buffered Output Stream. |
| **DataInputStream** | Contains method for reading java standard datatype  **DataInputStream** increases performance with its **readInt()** and **readLine()**methods etc. |
| **DataOutputStream** | An output stream that contain method for writing java standard data type |
| **FileInputStream** | Input stream that reads from a file |
| **FileOutputStream** | Output stream that write to a file. |
| **InputStream** | Abstract class that describe stream input. |
| **OutputStream** | Abstract class that describe stream output. |
| **PrintStream** | Output Stream that contain print() and println() method |
| **LineNumberInputStream** | adds line numbers in the destination file that do not exist in the source file |

## Working with Filtered Streams

Filtered streams process data as its being written to or read from the stream. The java.io package implements these filtered streams:

FilteredInputStream

* DataInputStream
* BufferedInputStream
* LineNumberInputStream
* PushbackInputStream

FilteredOutputStream

* PrintStream
* DataOutputStream
* BufferedOutputStream

**1. DataInputStream and DataOutputStream**

[DataInputStream and DataOutputStream](http://way2java.com/io/datainputstream-and-dataoutputstream-%e2%80%93-file-copying/) are high-level as they are the subclasses of **FilterInputStream** and **FilterOutputStream**. These streams' extra functionality is that they can read or write integers, doubles and lines at a time, instead of byte by byte. This increases the performance to some extent; instead of reading and writing byte by byte with FileInputStream and FileOutputStream.

To use a filtered input (output) stream, you attach the filtered stream to another input (output) stream. For example, you can attach a DataInputStream to the standard input stream like this:

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

String input;

while ((input = dis.readLine()) != null) {

. . . // do something interesting here

}

You might do this so that you can use the more convenient readXXX() methods, such as readLine(), implemented by DataInputStream.

**2. Giving Line Numbers – LineNumberInputStream**

**LineNumberInputStream** is a subclass of **FilterInputStream** stream. We know each filter stream adds some extra functionality. The [LineNumberInputStream](http://way2java.com/io/giving-line-numbers-linenumberinputstream/) does the extra work of **adding line numbers** that do not exist in the source.

**3. Using Buffering to Enhance the Performance**

Buffering of streams increases the performance to higher extents of few thousand times. For buffering, two streams exist in **java.io** package in byte streams category – [**BufferedInputStream** and **BufferedOutputStream**](http://way2java.com/io/bufferedinputstream-and-bufferedoutputstream/). These high-level classes are subclasses of **FilterInputStream** and **FilterOutputStream** and their functionality is to increase the performance with buffer. These streams give an implicit system-defined buffer (generally, 2048 bytes) into which data is read and written. The buffer decreases the number of transfers between source file context area and destination file context area and thereby performance increases.

**4. PushbackInputStream – Pushing unwanted extra character**

Java permits to unread the unwanted bytes in a stream with the class PushbackInputStream. This is an unusual filter stream and it pushes back a character into the input stream. It is used to check a sequence of characters that meets with a condition or not. It can check the delimiter of a line like \r (carriage-return) or \n (life-feed) etc. If the sequence is broken out, it uses unread() method to push a byte back to the stream. Internally, the DataInputStream uses a PushbackInputStream to check for a line-feed (\n) sequence is crossed or not. For a program, you can go through [PushbackReader](http://way2java.com/io/pushbackreader-%e2%80%93-pushing-out-character/).

#### Using Filtered Streams

To use a filtered input (output) stream, you attach the filtered stream to another input (output) stream. For example, you can attach a DataInputStream to the standard input stream like this:

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

String input;

while ((input = dis.readLine()) != null) {

. . . // do something interesting here

}

You might do this so that you can use the more convenient readXXX() methods, such as readLine(), implemented by DataInputStream.

#### [Using DataInputStream and DataInputStream](https://www.cs.princeton.edu/courses/archive/spr96/cs333/java/tutorial/java/io/dataIO.html)

This page provides and explains an example of using DataInputStream and DataOutputStream, two filtered streams that can read and write primitive Java data types.

These classes define several key methods. Two most important are

1. read() : reads byte of data.
2. write() : Writes byte of data.

### Reading and writing in a file

Till now, we have been reading the data entered by a user using the keyboard. Now, we will see how to read and write data in a file.

### Writing data in a file

**class Test**

**{**

**public static void main( String args[])**

**{**

**FileOutputStream fo=new FileOutputStream("prog.txt");**

**String s1="Welcome to Codesdope";**

**byte b1[]=s1.getBytes(); *//converting string into byte array***

**fo.write(b1);**

**fo.close();**

**}**

**}**

Here, **byte b1[ ]=s1.getBytes();** is converting string(character array) into byte array.

Then by writing **fo.write(b1);**, we are writing the data in a file named **prog.txt** because fo is the object of the FileOutputStream class.

### Reading data from a file

**class Test1**

**{**

**public static void main( String args[])**

**{**

**FileInputStream fi=new FileInputStream("prog.txt");**

**int n=0;**

**while((n=fi.read())!=-1){**

**System.out.println((char)n);**

**}**

**fin.close();**

**}**

**}**

In the above code, by writing **FileInputStream fi=new FileInputStream("prog.txt");**, we are creating an object **fi** of the class **FileInputStream**. Thus, the data of the file gets stored in the object fi.

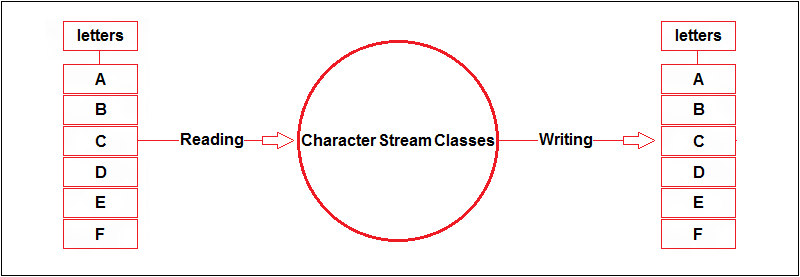
We are assigning **n = fi.read()** i.e. we are assigning the characters in the value of fi to the integer variable n. Note that we chose here an integer variable because the function read() returns an integer value.

**while((n=fi.read())!=-1) -** This while loop will continue till all the characters in the value of fi have been read.

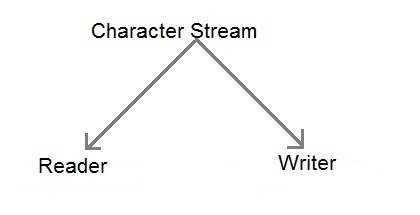
**System.out.println((char)n); -** Finally, the characters are printed. Since n is an integer, we are first converting it to a character by typecasting it.

**Character Stream Classes**

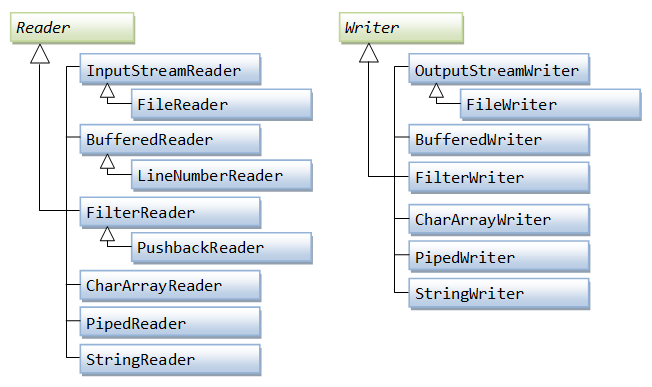
Character streams can be used to read and write 16-bit Unicode characters. Like byte streams, there are two kinds of character stream classes, namely, **reader stream classes** and **writer tream classes**.

[](http://hajsoftutorial.com/java/wp-content/uploads/2015/10/Untitled41.png)

 Character stream is also defined by using two abstract class at the top of hierarchy, they are Reader and Writer.



These two abstract classes have several concrete classes that handle unicode character.



1. **Reader Stream Classes**

Reader stream classes that are used to read characters include a super class known as **Reader** and a number of subclasses for supporting various input-related functions. Reader stream classes are functionally very similar to the input stream classes, except input streams use bytes as their fundamental unit of information, while reader streams use characters. The Reader class contains methods that are identical to those available in the InputStream class, except Reader is designed to handle characters. Therefore, reader classes can perform all the functions implemented by the input stream classes.

1. **Writer Stream Classes**

Like output stream classes, the writer stream classes are designed to perform all output operations on files. Only difference is that while output stream classes are designed to write bytes, the writer stream are designed to write character. The **Writer** class is an **abstract** class which acts as a base class for all the other writer stream classes. This base class provides support for all output operations by defining methods that are identical to those in Outputstream class.

**Some important Charcter stream classes.**

|  |  |
| --- | --- |
| **Stream class** | **Description** |
| **BufferedReader** | Handles buffered input stream. |
| **BufferedWriter** | Handles buffered output stream. |
| **FileReader** | Input stream that reads from file. |
| **FileWriter** | Output stream that writes to file. |
| **InputStreamReader** | Input stream that translate byte to character |
| **OutputStreamReader** | Output stream that translate character to byte. |
| **PrintWriter** | Output Stream that contain print() and println() method. |
| **Reader** | Abstract class that define character stream input |
| **Writer** | Abstract class that define character stream output |

### The FilterWriter is a base class for implementing your own filtering Writer's. Basically it just overrides all methods in Writer.

### Like with FilterOutputStream, I see no sensible purpose for this class. I cannot see that this class actually adds or changes any behaviour in Writer except that it takes a Writer in its constructor. If you choose to extend this class you might as well extend the Writer class directly, and avoid the extra class in the hierarchy.

The FilterReader is a base class for implementing your own filtering readers. Basically it just overrides all methods in Reader.

Like with FilterInputStream, I see no sensible purpose for this class. I cannot see that this class actually adds or changes any behaviour in Reader except that it takes a Reader in its constructor. If you choose to extend this class you might as well extend the Reader class directly, and avoid the extra class in the hierarchy.

The Java PushbackReader works much like the [**PushbackInputStream**](http://tutorials.jenkov.com/java-io/pushbackinputstream.html) except that the PushbackReader works on characters, whereas the PushbackInputStream works on bytes.

he Java LineNumberReader class (java.io.LineNumberReader is a BufferedReader that keeps track of line numbers of the read characters. Line numbering begins at 0. Whenever the LineNumberReader encounters a line terminator in the characters returned by the wrapped Reader, the line number is incremented.

You can get the current line number from the LineNumberReader by calling the getLineNumber() method. You can also set the current line number, should you need to, by calling the setLineNumber() method.

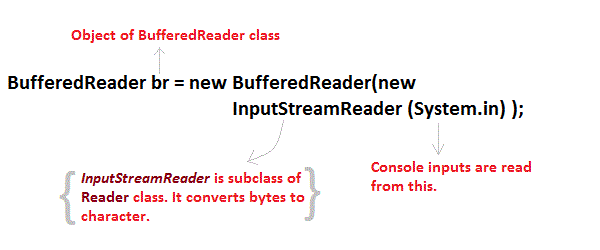
### Reading Console Input/ Taking input from keyboard

All the Stream classes come under java.io package.

**Import Java.io.\*;**

Note that here we wrote \* because we want to include all the classes of java.util package.

.To take input from a user, we use **BufferedReader** class by creating an object of it. For that, we have to write the following code.



Now, let's understand this code word by word.

**BufferedReader** - This is a class that is used for taking character input.

**b** - object of BufferedReader class

**InputStreamReader** - It converts bytes to characters.

**System.in** - It is input stream. User inputs are read from this.

Thus, we are taking user input from **System.in** which is converted from bytes to characters by the class **InputStreamReader**. This value is stored in the object **b** of the class **BufferedReader**.

1. **Reading Characters**

read() method is used with BufferedReader object to read characters. As this function returns integer type value has we need to use typecasting to convert it into **char** type.

*int* **read()** throws **IOException**

->Below is a simple example explaining character input.

class CharRead

{

public static void main( String args[])

{

BufferedReader br = new Bufferedreader(new InputstreamReader(System.in));

System.out.println(“enter a character”);

char c = **(char)br.read()**; //Reading character

System.out.println(“c is”+c);

}

}

**Output:**

enter a character

y

c is y

1. **Reading Strings**

To read string we have to use readLine() function with BufferedReader class's object.

*String* **readLine()** throws **IOException**

**Program to take String input from Keyboard in Java**

import java.io.\*;

class MyInput

{

public static void main(String[] args)

{

String text;

**InputStreamReader** isr = new **InputStreamReader(System.in)**;

**BufferedReader** br = new **BufferedReader(isr)**;

System.out.println(“enter a line of text”);

text = br.readLine(); //Reading String

System.out.println(“text is:“+text);

}

}

**Output:**

enter a line of text

hi h r u

text is: hi h r u

**Program to read from a file using BufferedReader class**

import java. Io \*;

class ReadTest

{

public static void main(String[] args)

{

try

{

File fl = new File("d:/myfile.txt");

BufferedReader br = new BufferedReader(new FileReader(fl)) ;

String str;

while((str=br.readLine())!=null)

{

System.out.println(str);

}

br.close();

fl.close();

}

catch(IOException e)

{ e.printStackTrace(); }

}

}

**Program to write to a File using FileWriter class**

import java. Io \*;

class WriteTest

{

public static void main(String[] args)

{

try

{

File fl = new File("d:/myfile.txt");

String str="Write this string to my file";

FileWriter fw = new FileWriter(fl) ;

fw.write(str);

fw.close();

fl.close();

}

catch (IOException e)

{ e.printStackTrace(); }

}

}

In addition to the stream classes, java.io contains these other classes:

**File**

Represents a file on the native file system. You can create a File object for a file on the native file system and then query the object for information about that file (such as its full pathname).

**FileDescriptor**

Represents a file handle (or descriptor) to an open file or an open socket. You will not typically use this class.

**RandomAccessFile**

Represents a random access file.

**StreamTokenizer**

Breaks the contents of a stream into tokens. Tokens are the smallest unit recognized by a text-parsing algorithm (such as words, symbols, and so on). A StreamTokenizer object can be used to parse any text file. For example, you could use it to parse a Java source file into variable names operators and so on, or an HTML file into HTML tags, words and such.

java.io defines three interfaces:

**DataInput** and **DataOutput**

These two interfaces describe streams that can read and write primitive Java types in machine-independent format. DataInputStream, DataOutputStream, and RandomAccessFile implement these interfaces.

**FilenameFilter**

The list method in the File class uses a FilenameFilter to determine which files in a directory to list. The FilenameFilter accepts or rejects files based on their names. You could use FilenameFilter to implement simple regular expression style file search patterns such as foo.\*.

# Java IO Tutorial - Java Random Access Files

Using a random access file, we can read from a file as well as write to the file.

Reading and writing using the file input and output streams are a sequential process.

Using a random access file, we can read or write at any position within the file.

An object of the RandomAccessFile class can do the random file access. We can read/write bytes and all primitive types values to a file.

RandomAccessFile can work with strings using its readUTF() and writeUTF() methods.

The RandomAccessFile class is not in the class hierarchy of the InputStream and OutputStream classes.

The RandomAccessFileclass treats the file as an array of Bytes And you can write your data in any position of the Array. To do that, it uses a pointer that holds the current position (you can think of that pointer like a cursor in a text editor…).

|  |  |
| --- | --- |
| [Constructor](https://www.javatpoint.com/java-constructor) | **Description** |
| RandomAccessFile(File file, [String](https://www.javatpoint.com/java-string)mode) | Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument. |
| RandomAccessFile(String name, String mode) | Creates a random access file stream to read from, and optionally to write to, a file with the specified name. |

## Mode

A random access file can be created in four different access modes. The access mode value is a string. They are listed as follows:

|  |  |
| --- | --- |
| **Mode** | **Meaning** |
| "r" | The file is opened in a read-only mode. |
| "rw" | The file is opened in a read-write mode. The file is created if it does not exist. |
| "rws" | The file is opened in a read-write mode. Any modifications to the file's content and its metadata are written to the storage device immediately. |
| "rwd" | The file is opened in a read-write mode. Any modifications to the file's content are written to the storage device immediately. |

## Read and Write

We create an instance of the RandomAccessFile class by specifying the file name and the access mode.

RandomAccessFile raf = new RandomAccessFile("randomtest.txt", "rw");

A random access file has a file pointer that moves forward when we read data from it or write data to it.

The file pointer is a cursor where our next read or write will start.

Its value indicates the distance of the cursor from the beginning of the file in bytes.

**getFilePointer()** : to get the current position of the pointer

When we create an object of the RandomAccessFile class, the file pointer is set to zero.

**seek(**int**)** :We can set the file pointer at a specific location in the file using the seek() method.

**length()** :The length() method of a RandomAccessFile returns the current length of the file.

**setLength():** We can extend or truncate a file by using its setLength() method.

**read(byte[] b):** to reads up to b.length bytes of data from the file into an array of bytes

**write(byte[] b):** to write b.length bytes from the specified byte array to the file, starting at the current file pointer

|  |
| --- |
|  |

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

1. **import** java.io.IOException;
2. **import** java.io.RandomAccessFile;
3. **public** **class** RandomAccessFileExample {
4. **static** **final** String FILEPATH ="myFile.TXT";
5. **public** **static** **void** main(String[] args) {
6. **try** {
7. System.out.println(**new** String(readFromFile(FILEPATH, 0, 18)));
8. writeToFile(FILEPATH, "I love my country and my people", 31);
9. } **catch** (IOException e) {
10. e.printStackTrace();
11. }
12. }
13. **private** **static** **byte**[] readFromFile(String filePath, **int** position, **int** size)
14. **throws** IOException {
15. RandomAccessFile file = **new** RandomAccessFile(filePath, "r");
16. file.seek(position);
17. **byte**[] bytes = **new** **byte**[size];
18. file.read(bytes);
19. file.close();
20. **return** bytes;
21. }
22. **private** **static** **void** writeToFile(String filePath, String data, **int** position)
23. **throws** IOException {
24. RandomAccessFile file = **new** RandomAccessFile(filePath, "rw");
25. file.seek(position);
26. file.write(data.getBytes());
27. file.close();
28. }
29. }