**IO Streams**

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

Stream

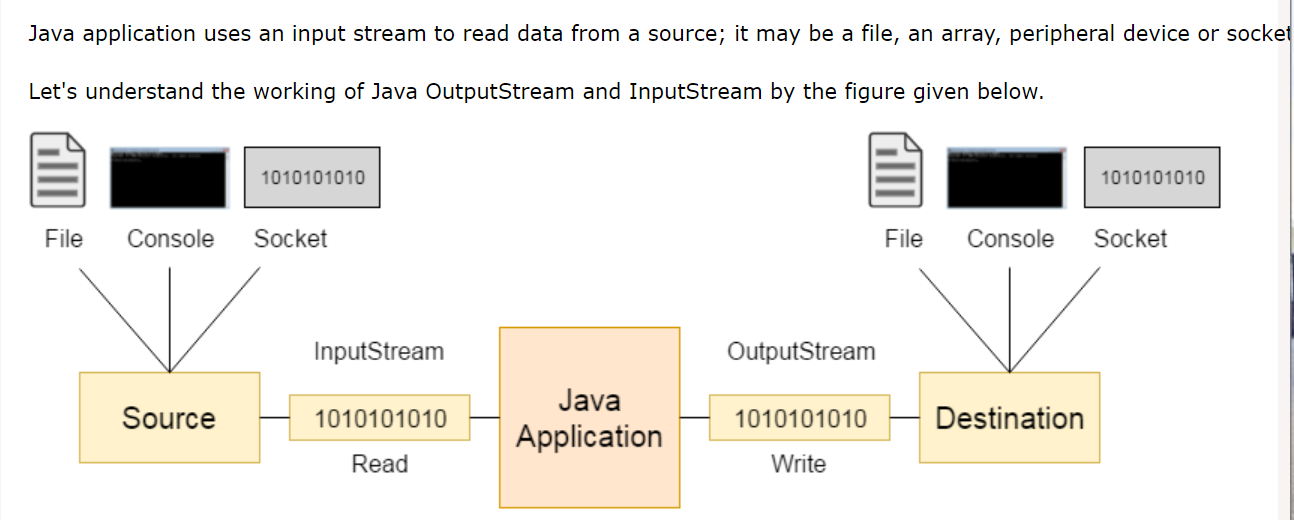
A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

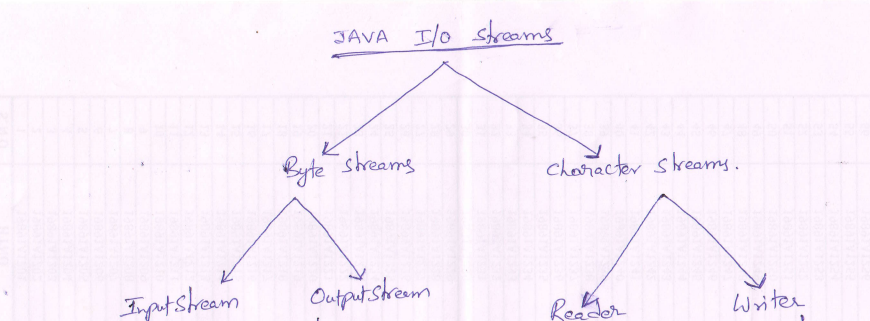
In Java, 3 streams are created for us automatically. All these streams are attached with the console.

**1) System.out:**standard output stream

**2) System.in:**standard input stream

**3) System.err:**standard error stream





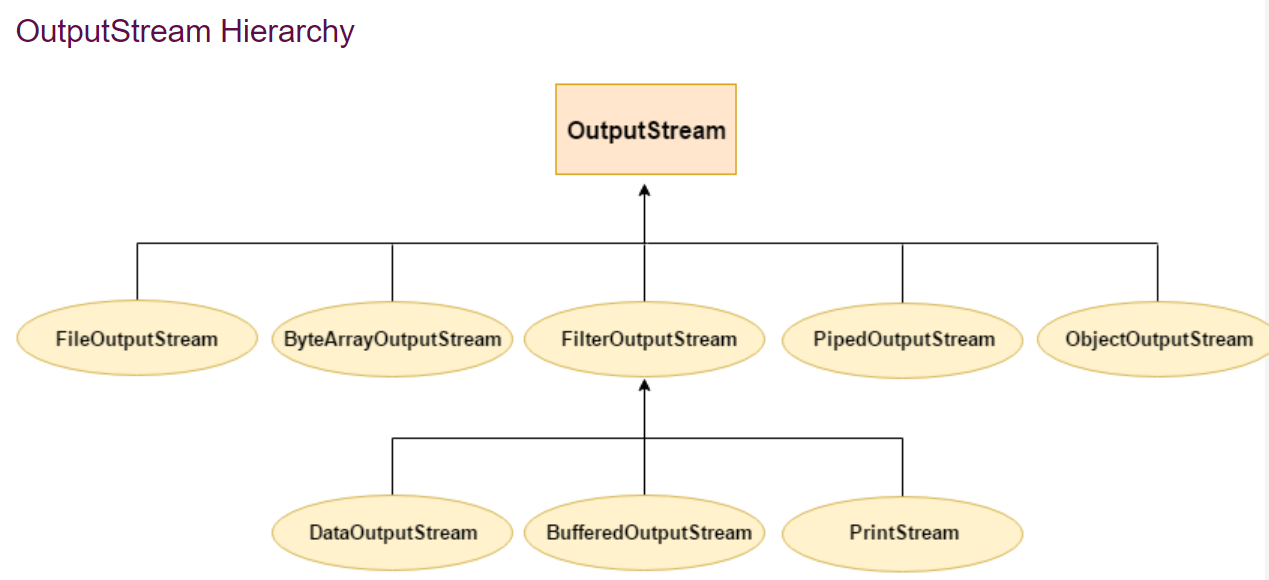
**Byte Streams**

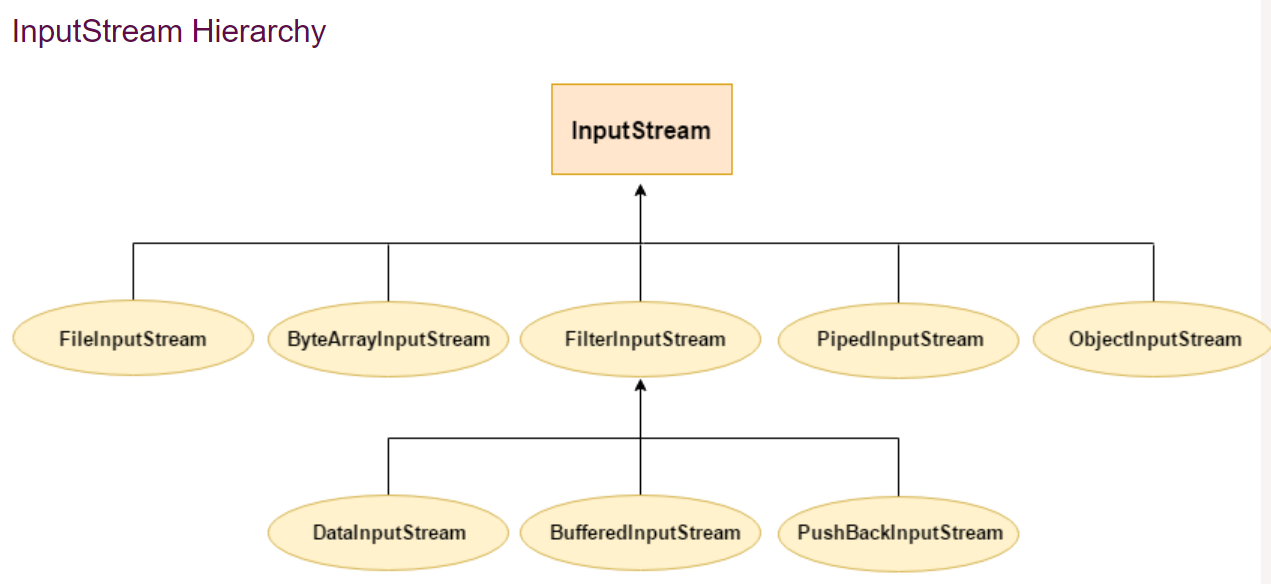
OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

InputStream

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



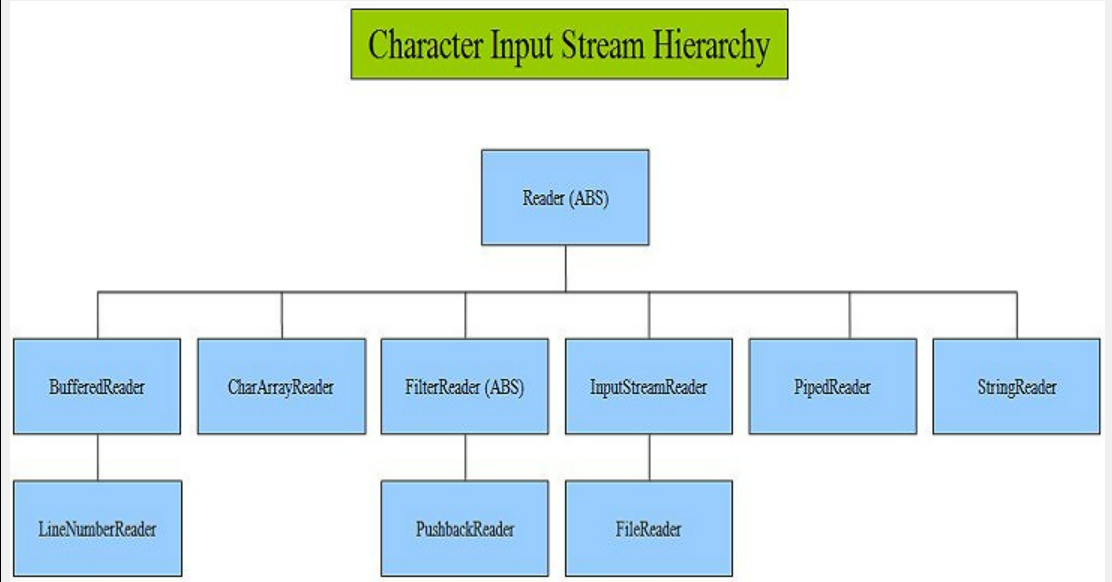


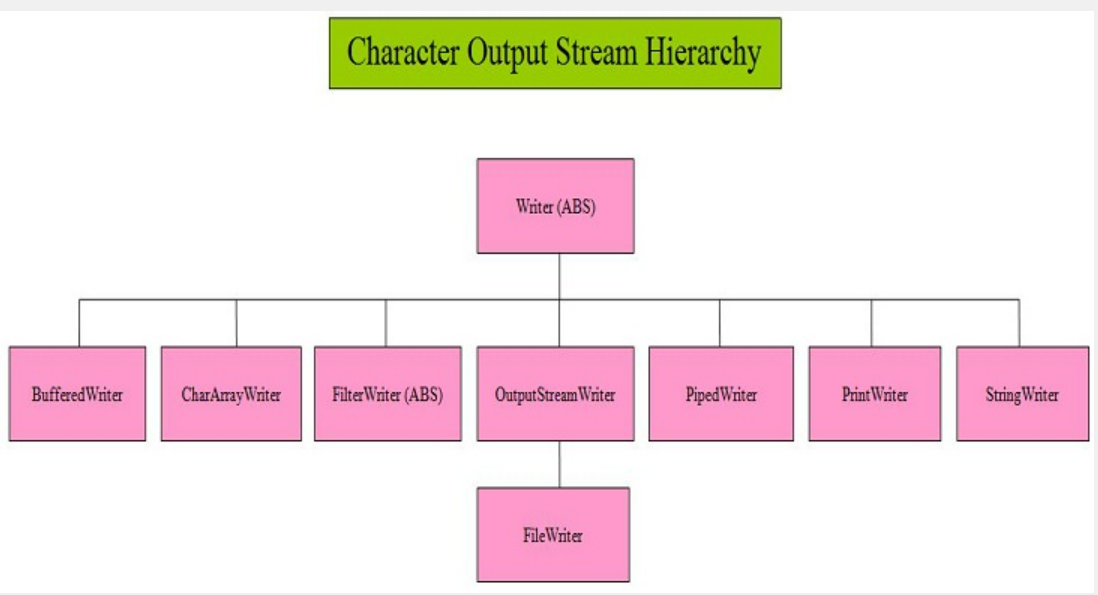
**Character Streams**

Character streams are defined within two class hierarchies, one for input and one for output:

* The Writer class is the *abstract superclass* of all character output streams
* The Reader class is the *abstract superclass* of all character input streams

These classes define the characteristics that are common to character input and character output streams, which are implemented in the concrete subclasses of each hierarchy.





**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.

|  |  |  |  |
| --- | --- | --- | --- |
| **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.nio.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](https://www.javatpoint.com/array-in-java) 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. |

**Java –RandomAccessFile**

This [class](https://www.javatpoint.com/object-class) is used for reading and writing to random access file. A random access file behaves like a large [array](https://www.javatpoint.com/array-in-java) of bytes. There is a cursor implied to the array called file [pointer](https://www.javatpoint.com/c-pointers), by moving the cursor we do the read write operations. If end-of-file is reached before the desired number of byte has been read than EOFException is [thrown](https://www.javatpoint.com/throw-keyword). It is a type of IOException.

* This class is not derived from either InputStream or OutputStream.
* Implemented DataInput&DataOutput interfaces.
* RandomAccessFile class has implemented all the standard input & output methods which we can use for read/write operations with Random access files.
* We can position the file pointer at a required position for read/write operations.

**Access modes:**

* r - for read
* w - for write
* rw - for read and write

**Positioning File Pointer:**

void seek(long newpos) throws IOException

* It moves the file pointer by number of bytes specified by the argument ‘newpos’ from the beginning of the file.

**Java Console Class**

The Java Console class is used to get input from console. It provides methods to read texts and passwords.

If you read password using Console class, it will not be displayed to the user.

The java.io.Console class is attached with system console internally. The Console class is introduced since 1.5

## **Java Console class declaration**

Let's see the declaration for Java.io.Console class:

**public** **final** **class** Console **extends** Object **implements** Flushable

## **How to get the object of Console**

System class provides a static method console() that returns the [singleton](https://www.javatpoint.com/singleton-design-pattern-in-java) instance of Console class.

**public** **static** Console console(){}

Let's see the code to get the instance of Console class.

Console c=System.console();

**Singleton Class in Java:**

A singleton class is a class that can have only one object (an instance of the class) at a time.  
After first time, if we try to instantiate the Singleton class, the new variable also points to the first instance created. So whatever modifications we do to any variable inside the class through any instance, it affects the variable of the single instance created and is visible if we access that variable through any variable of that class type defined.

**Serialization**

Process of writing state of an object to a byte stream. Saving the object in a persistant storage.

Only the objects that implement serializable interface can be serialized.

The serializable interface has no methods, it only indicates that the objects of this can

be serialized.

If a class is Serializable, all of its sub classes are also serializable.

Transient attributes will not be serialized.

Static attributes also will not be serialized.

If the object to be serialized has references to other objects, which in turn have

reference to still more objects. This set of objects and the relation ships among them

form a directed graph.

There may also be circular references within this object graph ie object x may contain

reference to object y, and y may have references to back to  x.

Objects may also contain references to themselves.

The object serialization and deserialization facilities have been designed to work

correctly in these scenarios.

If you attempt to serialize an object at the top of the object graph, all of the

referenced objects are recursively located and serialized.

At deserialization all these objects and their references are corectly restored.

It is legal to serialize an object of type that has a super type that does not implement

Serializable interface.

At the time of deserialization jre invokes the constructors of those base classes which

do not implement Serializable interface.

An object Serialized on one JVM can be successfully deserialized on another JVM

**Autoboxing & Unboxing**

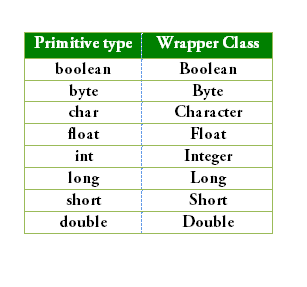
**Autoboxing:**Converting a primitive value into an object of the corresponding [wrapper class](https://www.geeksforgeeks.org/wrapper-classes-java/) is called autoboxing. For example, converting int to [Integer class](https://www.geeksforgeeks.org/wrapper-classes-java/). The Java compiler applies autoboxing when a primitive value is:

* Passed as a parameter to a method that **expects an object** of the corresponding wrapper class.
* Assigned to a variable of the corresponding **wrapper class**.

**Unboxing:** Converting an object of a wrapper type to its corresponding primitive value is called unboxing. For example conversion of [Integer](https://www.geeksforgeeks.org/wrapper-classes-java/) to int. The Java compiler applies unboxing when an object of a wrapper class is:

* Passed as a parameter to a method that **expects a value** of the corresponding primitive type.
* Assigned to a variable of the corresponding **primitive type**.

The following table lists the primitive types and their corresponding wrapper classes, which are used by the Java compiler for autoboxing and unboxing:



Ex:

int i=10; //i is variable of type int

Integer j = new Integer(10); //j is a object of type Integer

**Generics in Java**

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

**Syntax** to use generic collection

1. ClassOrInterface<Type>

**Example** to use Generics in java

1. ArrayList<String>

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

**1) Type-safety:** We can hold only a single type of objects in generics. It doesn?t allow to store other objects.

Without Generics, we can store any type of objects.

1. List list = **new** ArrayList();
2. list.add(10);
3. list.add("10");
4. With Generics, it is required to specify the type of object we need to store.
5. List<Integer> list = **new** ArrayList<Integer>();
6. list.add(10);
7. list.add("10");// compile-time error

**2) Type casting is not required:** There is no need to typecast the object.

Before Generics, we need to type cast.

1. List list = **new** ArrayList();
2. list.add("hello");
3. String s = (String) list.get(0);//typecasting
4. After Generics, we don't need to typecast the object.
5. List<String> list = **new** ArrayList<String>();
6. list.add("hello");
7. String s = list.get(0);

**3) Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

1. List<String> list = **new** ArrayList<String>();
2. list.add("hello");
3. list.add(32);//Compile Time Error