**The Stream Classes**

Java’s stream-based I/O is built upon four abstract classes: **InputStream**, **OutputStream**, **Reader**, and **Writer**. They are used to create several concrete stream subclasses. Although your programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream classes.

**InputStream** and **OutputStream** are designed for ***byte*** streams. **Reader** and **Writer** are designed for ***character*** streams. The byte stream classes and the character stream classes form separate hierarchies. In general, you should use the character stream classes when working with characters or strings and use the byte stream classes when working with bytes or other binary objects.

**The Byte Streams**

The byte stream classes provide a rich environment for handling byte-oriented I/O. A byte stream can be used with any type of object, including binary data. This versatility makes byte streams important to many types of programs. Since the byte stream classes are topped by **InputStream** and **OutputStream**, our discussion begins with them.

**InputStream**

**InputStream** is an abstract class that defines Java’s model of streaming byte input. It implements the **AutoCloseable** and **Closeable** interfaces. Most of the methods in this class will throw an **IOException** when an I/O error occurs.

The Java InputStream class is the base class (superclass) of all input streams in the [**Java IO API**](https://jenkov.com/tutorials/java-io/index.html). Each subclass of InputStream typically has a very specific use, but can be used as an InputStream. The InputStream subclasses are:

* [**ByteArrayInputStream**](https://jenkov.com/tutorials/java-io/bytearrayinputstream.html)
* [**FileInputStream**](https://jenkov.com/tutorials/java-io/fileinputstream.html)
* [**PipedInputStream**](https://jenkov.com/tutorials/java-io/pipedinputstream.html)
* [**BufferedInputStream**](https://jenkov.com/tutorials/java-io/bufferedinputstream.html)
* [**FilterInputStream**](https://jenkov.com/tutorials/java-io/filterinputstream.html)
* [**PushbackInputStream**](https://jenkov.com/tutorials/java-io/pushbackinputstream.html)
* [**DataInputStream**](https://jenkov.com/tutorials/java-io/datainputstream.html)
* [**ObjectInputStream**](https://jenkov.com/tutorials/java-io/objectinputstream.html)
* [**SequenceInputStream**](https://jenkov.com/tutorials/java-io/sequenceinputstream.html)

**InputStreams and Sources**

A Java InputStream is typically connected to some data source, like a file, network connection, pipe etc. This is also explained in more detail in the [**Java IO Overview**](https://jenkov.com/tutorials/java-io/overview.html) text.

|  |  |
| --- | --- |
| **Method** | **Description** |
| int available( ) | Returns the number of bytes of input currently available for reading. |
| void close( ) | Closes the input source. Further read attempts will generate an **IOException**. |
| void mark(int *numBytes*) | Places a mark at the current point in the input stream that will remain valid until *numBytes* bytes are read. |
| boolean markSupported( ) | Returns **true** if **mark( )** / **reset( )** are supported by the invoking stream. |
| static InputStream nullInputStream( ) | Returns an open, but null input stream, which is a stream that contains no data. Thus, the stream is always at the end of the stream and no input can be obtained. The stream can, however, be closed. |
| int read( ) | Returns an integer representation of the next available byte of input. –1 is returned when an attempt is made to read at the end of the stream. |
| int read(byte[ ] *buffer*) | Attempts to read up to *buffer.length* bytes into *buffer* and returns the actual number of bytes that were successfully read. –1 is returned when an attempt is made to read at the end of the stream. |
| int read(byte[ ] *buffer*, int *offset*, int *numBytes*) | Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when an attempt is made to read at the end of the stream. |
| byte[ ] readAllBytes( ) | Beginning at the current position, reads to the end of the stream, returning a byte array that holds the input. |
| byte[ ] readNBytes(int *numBytes*) | Attempts to read *numBytes* bytes, returning the result in a byte array. If the end of the stream is reached before *numBytes* bytes have been read, then the returned array will contain less than *numBytes* bytes. |
| int readNBytes(byte[ ] *buffer*, int *offset*, int *numBytes*) | Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. |
| void reset( ) | Resets the input pointer to the previously set mark. |
| long skip(long *numBytes*) | Ignores (that is, skips) *numBytes* bytes of input, returning the number of bytes actually ignored. |
| void skipNBytes(long *numBytes*) | Ignores (that is, skips) *numBytes* of input. Throws **EOFException** if the end of the stream is reached before *numBytes* are skipped, or **IOException** if an I/O error occurs. |
| long transferTo(OutputStream *strm*) | Copies the bytes in the invoking stream into *strm*, returning the number of bytes copied. |

Methods of InputStream.

**mark() and reset()**

The InputStream class has two methods called mark() and reset() which subclasses of InputStream may or may not support.

If an InputStream subclass supports the **mark**() and **reset**() methods, then that subclass should override the markSupported() to return true. If the **markSupported**() method returns false then mark() and reset() are not supported.

The **mark**() sets a mark internally in the InputStream which marks the point in the stream to which data has been read so far. The code using the InputStream can then continue reading data from it. If the code using the InputStream wants to go back to the point in the stream where the mark was set, the code calls **reset**() on the InputStream. The InputStream then "**rewinds**" and go back to the mark, and starts returning (reading) data from that point again. This will of course result in some data being returned more than once from the InputStream.

The methods **mark**() and **reset**() methods are typically used when implementing parsers. Sometimes a parser may need to read ahead in the InputStream and if the parser doesn't find what it expected, it may need to rewind back and try to match the read data against something else.

**FileInputStream**

The **FileInputStream** class creates an **InputStream** that you can use to read bytes from a file. Two commonly used constructors are shown here: Either can throw a **FileNotFoundException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.

FileInputStream(String filePath)

FileInputStream(File fileObj)

The following example creates two **FileInputStream**s that use the same file and each of the two constructors:

FileInputStream f0 = new FileInputStream("/autoexec.bat")

File f = new File("/autoexec.bat");

FileInputStream f1 = new FileInputStream(f);

Although the first constructor is probably more commonly used, the second allows you to closely examine the file using the **File** methods, before attaching it to an input stream.

When a **FileInputStream** is created, it is also opened for reading. **FileInputStream** overrides several of the methods in the abstract class **InputStream**. The **mark( )** and **reset( )** methods are not overridden, and any attempt to use **reset( )** on a **FileInputStream** will generate an **IOException**.

Example:

import java.io.FileInputStream;

import java.io.IOException;

class FileInputStreamDemo {

public static void main(String[] args) {

int size;

// Use try-with-resources to close the stream.

try (FileInputStream f = new FileInputStream(

"D:\\Khawja\_Ghouse\\Study\_Material\\java\\workspace\\Java\_IO\\src\\com\\skg\\file\\demo.txt")) {

System.out.println("Total Available Bytes: " + (size = f.available()));

int n = size / 40;

System.out.println("First " + n + " bytes of the file one read() at a time");

for (int i = 0; i < n; i++) {

System.out.print((char) f.read());

}

System.out.println("\nStill Available: " + f.available());

System.out.println("Reading the next " + n + " with one read(b[])");

// As the length of bytes we are giving f.read(b) that many byte read will be

// completed.

byte[] b = new byte[200];

if (f.read(b) != 200) {

System.err.println("couldn’t read " + n + " bytes.");

}

System.out.println(new String(b, 0, 200));

System.out.println("\nStill Available: " + (size = f.available()));

System.out.println("Skipping half of remaining bytes with skip()");

f.skip(size / 2);

System.out.println("Still Available: " + f.available());

System.out.println("Reading " + n / 2 + " into the end of array");

if (f.read(b, n / 2, n / 2) != n / 2) {

System.err.println("couldn’t read " + n / 2 + " bytes.");

}

System.out.println(new String(b, 0, b.length));

System.out.println("\nStill Available: " + f.available());

} catch (IOException e) {

System.out.println("I/O Error: " + e);

}

}

}

**OutputStream**

**OutputStream** is an abstract class that defines streaming byte output. It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces. Most of the methods defined by this class return **void** and throw an **IOException** in the case of I/O errors. Table 22-2 shows the methods in **OutputStream**.

**Java OutputStream Subclasses**

Here are some of the well-known subclasses of the Java *OutputStream* class:

* [**ByteArrayOutputStream**](https://jenkov.com/tutorials/java-io/bytearrayoutputstream.html)
* **FileOutputStream**
* [**PipedOutputStream**](https://jenkov.com/tutorials/java-io/pipedoutputstream.html)
* [**BufferedOutputStream**](https://jenkov.com/tutorials/java-io/bufferedoutputstream.html)
* [**FilterOutputStream**](https://jenkov.com/tutorials/java-io/filteroutputstream.html)
* [**DataOutputStream**](https://jenkov.com/tutorials/java-io/dataoutputstream.html)
* [**PrintStream**](https://jenkov.com/tutorials/java-io/printstream.html)
* [**ObjectOutputStream**](https://jenkov.com/tutorials/java-io/objectoutputstream.html)

**OutputStream's and Destinations**

A Java OutputStream is typically connected to some data destination - as mentioned in the [**Java IO Overview**](https://jenkov.com/tutorials/java-io/overview.html), like a file, network connection, pipe, memory buffer etc. This is also explained in more detail in the [**Java IO Overview**](https://jenkov.com/tutorials/java-io/overview.html) text. The OutputStream's data destination is where all data written to the OutputStream will eventually end.

|  |  |
| --- | --- |
| **Method** | **Description** |
| void close( ) | Closes the output stream. Further write attempts will generate an **IOException**. |
| void flush( ) | Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers. |
| static OutputStream nullOutputStream( ) | Returns an open, but null output stream, which is a stream to which no output is actually written. Thus, its output methods can be called but don’t actually produce output. The stream can, however, be closed. |
| void write(int *b*) | Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write( )** with an expression without having to cast it back to **byte**. |
| void write(byte[ ] *buffer*) | Writes a complete array of bytes to an output stream. |
| void write(byte[ ] *buffer*, int *offset*, int *numBytes*) | Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*]. |

**FileOutputStream**

**FileOutputStream** creates an **OutputStream** that you can use to write bytes to a file. It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces. Four of its constructors are shown here:

FileOutputStream(String *filePath*) FileOutputStream(File *fileObj*) FileOutputStream(String *filePath*, boolean *append*) FileOutputStream(File *fileObj*, boolean *append*)

They can throw a **FileNotFoundException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. If *append* is **true**, the file is opened in append mode.

The creation of a **FileOutputStream** is not dependent on the file already existing. **FileOutputStream** will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an exception will be thrown.

The following example creates a sample buffer of bytes by first making a **String** and then using the **getBytes( )** method to extract the byte array equivalent. It then creates three files. The first, **file1.txt**, will contain every other byte from the sample. The second, **file2.txt**, will contain the entire set of bytes. The third and last, **file3.txt**, will contain only the last quarter.

import java.io.FileOutputStream;

import java.io.IOException;

public class FileOutputStreamDemo {

public static void main(String[] args) {

String source = "Now is the time for all good men\n" + " to come to the aid of their country\n"

+ " and pay their due taxes.";

byte[] buf = source.getBytes();

FileOutputStream f0 = null;

FileOutputStream f1 = null;

FileOutputStream f2 = null;

try {

f0 = new FileOutputStream("file1.txt");

f1 = new FileOutputStream("file2.txt");

f2 = new FileOutputStream("file3.txt");

// write to first file

for (int i = 0; i < buf.length; i += 2)

f0.write(buf[i]);

// write to second file

f1.write(buf);

// write to third file

f2.write(buf, buf.length - buf.length / 4, buf.length / 4);

} catch (IOException e) {

System.out.println("An I/O Error Occurred");

} finally {

try {

if (f0 != null)

f0.close();

} catch (IOException e) {

System.out.println("Error Closing file1.txt");

}

try {

if (f1 != null)

f1.close();

} catch (IOException e) {

System.out.println("Error Closing file2.txt");

}

try {

if (f2 != null)

f2.close();

} catch (IOException e) {

System.out.println("Error Closing file3.txt");

}

}

}

}

The Java OutputStream's flush() method flushes all data written to the OutputStream to the underlying data destination. For instance, if the OutputStream is a FileOutputStream then bytes written to the FileOutputStream may not have been fully written to disk yet. The data might be buffered in OS memory somewhere, even if your Java code has written it to the FileOutputStream. By calling flush() you can assure that any buffered data will be flushed (written) to disk (or network, or whatever else the destination of your OutputStream has). Here is an example of flushing data written to a Java OutputStream by calling its flush() method: f1.flush();

Here are the contents of each file after running this program. First, **file1.txt**:

Nwi h iefralgo e t oet h i ftercuty n a hi u ae.

Next, **file2.txt**:

Now is the time for all good men to come to the aid of their country and pay their due taxes.

Finally, **file3.txt**:

nd pay their due taxes.

1. //Demonstrate FileOutputStream.
2. //This version uses try-with-resources.
3. import java.io.\*;
5. class FileOutputStreamDemo {
6. public static void main(String[] args) {
7. String source = "Now is the time for all good men\n" + " to come to the aid of their country\n"
8. + " and pay their due taxes.";
9. byte[] buf = source.getBytes();
11. //Use try-with-resources to close the files.
12. try (FileOutputStream f0 = new FileOutputStream("file1.txt");
13. FileOutputStream f1 = new FileOutputStream("file2.txt");
14. FileOutputStream f2 = new FileOutputStream("file3.txt")) {
15. //write to first file
16. for (int i = 0; i < buf.length; i += 2)
17. f0.write(buf[i]);
18. //write to second file
19. f1.write(buf);
20. //write to third file
21. f2.write(buf, buf.length - buf.length / 4, buf.length / 4);
22. } catch (IOException e) {
23. System.out.println("An I/O Error Occurred");
24. }
25. }

**ByteArrayInputStream and ByteArrayOutputStream**

**ByteArrayInputStream** is an implementation of an input stream that uses a byte array as the source. This class has two constructors, each of which requires a byte array to provide the data source:

**ByteArrayInputStream**(byte[ ] *array*) ByteArrayInputStream(byte[ ] *array*, int *start*, int *numBytes*)

Here, *array* is the input source. The second constructor creates an **InputStream** from a subset of the byte array that begins with the character at the index specified by *start* and is *numBytes* long.

The **close( )** method has no effect on a **ByteArrayInputStream**. Therefore, it is not necessary to call **close( )** on a **ByteArrayInputStream**, but doing so is not an error.

The following example creates a pair of **ByteArrayInputStream**s, initializing them with the byte representation of the alphabet:

//Demonstrate ByteArrayInputStream.

import java.io.\*;

class ByteArrayInputStreamDemo {

public static void main(String[] args) {

String tmp = "abcdefghijklmnopqrstuvwxyz";

byte[] b = tmp.getBytes();

ByteArrayInputStream input1 = new ByteArrayInputStream(b);

ByteArrayInputStream input2 = new ByteArrayInputStream(b, 0, 3);

}

}

**ByteArrayOutputStream**

ByteArrayOutputStream is an implementation of an output stream that uses a byte array as the destination. ByteArrayOutputStream has two constructors, shown here:

ByteArrayOutputStream( )

ByteArrayOutputStream(int numBytes)

In the first form, a buffer of 32 bytes is created. In the second, a buffer is created with a size equal to that specified by *numBytes*. The buffer is held in the protected buf field of ByteArrayOutputStream. The buffer size will be increased automatically, if needed. The number of bytes held by the buffer is contained in the protected count field of ByteArrayOutputStream.

The close( ) method has no effect on a ByteArrayOutputStream. Therefore, it is not necessary to call close( ) on a ByteArrayOutputStream, but doing so is not an error.

package com.skg.file;

import java.io.ByteArrayOutputStream;

import java.io.IOException;

public class ByteArrayOutputStreamDemo {

public static void main(String[] args) {

ByteArrayOutputStream f = new ByteArrayOutputStream();

String s = "This should end up in the array";

byte[] buf = s.getBytes();

try {

f.write(buf);

} catch (IOException e) {

System.out.println("Error Writing to Buffer");

return;

}

System.out.println("Buffer as a string");

System.out.println(f.toString());

System.out.println("Into array");

byte[] b = f.toByteArray();

for (int i = 0; i < b.length; i++)

System.out.print((char) b[i]);

System.out.println("\nTo an OutputStream()");

}

}

Output:

Buffer as a string

This should end up in the array

Into array

This should end up in the array

To an OutputStream()

**BufferedInputStream**

The Java **BufferedInputStream** class, java.io.BufferedInputStream, provides transparent reading of chunks of bytes and buffering for a [**Java InputStream**](https://jenkov.com/java-io/inputstream.html), including any subclasses of InputStream. Reading larger chunks of bytes and buffering them can speed up IO quite a bit. Rather than read one byte at a time from the network or disk, the BufferedInputStream reads a larger block at a time into an internal buffer. When you read a byte from the Java BufferedInputStream you are therefore reading it from its internal buffer. When the buffer is fully read, the BufferedInputStream reads another larger block of data into the buffer. This is typically much faster than reading a single byte at a time from an InputStream, especially for disk access and larger data amounts.

Java BufferedInputStream Example : To add buffering to an InputStream simply wrap it in a BufferedInputStream. Here is how that looks:

BufferedInputStream bufferedInputStream = new BufferedInputStream(

new FileInputStream("c:\\data\\input-file.txt"));

As you can see, using a BufferedInputStream to add buffering to a non-buffered InputStream is pretty easy. The BufferedInputStream creates a byte array internally, and attempts to fill the array by calling the InputStream.read(byte[]) methods on the underlying InputStream.

Setting Buffer Size of a BufferedInputStream

You can set the buffer size to use internally by the Java BufferedInputStream. You provide the buffer size as a parameter to the BufferedInputStream constructor, like this:

int bufferSize = 8 \* 1024;

BufferedInputStream bufferedInputStream = new BufferedInputStream(

new FileInputStream("c:\\data\\input-file.txt"),

bufferSize

);

This example sets the internal buffer used by the BufferedInputStream to 8 KB. It is best to use buffer sizes that are multiples of 1024 bytes. That works best with most built-in buffering in hard disks etc.

Except for adding buffering to your input streams, BufferedInputStream behaves exactly like an [**InputStream**](https://jenkov.com/tutorials/java-io/inputstream.html).

**mark() and reset()**

An interesting aspect to note about the BufferedInputStream is that is supports the mark() and reset() methods inherited from the InputStream. Not all InputStream subclasses support these methods. In general, you can call the markSupported() method to find out if mark() and reset() are supported on a given InputStream or not, **but the BufferedInputStream supports them.**

**Closing a BufferedInputStream**

When you are finished reading data from a Java BufferedInputStream you must close it. You close a BufferedInputStream by calling the ***close***() method inherited from InputStream. Closing a Java BufferedInputStream will also close the InputStream from which the BufferedInputStream is reading and buffering data.

**BufferedOutputStream**

The Java BufferedOutputStream class, java.io.BufferedOutputStream, is used to capture bytes written to the BufferedOutputStream in a buffer, and write the whole buffer in one batch to an underlying [**Java OutputStream**](https://jenkov.com/tutorials/java-io/outputstream.html) for increased performance. Buffering can speed up IO quite a bit, especially when writing data to disk access or network.

Creating a BufferedOutPutStream

To add buffering to your OutputStream's simply wrap them in a BufferedOutputStream. Here is an example of wrapping an OutputStream in a BufferedOutputStream:

package com.skg.file;

import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class BufferedIOStreamDemo {

public static void main(String[] args) {

File file = new File("file1.txt");

FileInputStream fileInputStream = null;

BufferedInputStream bufferedInputStream = null;

FileOutputStream fileOutputStream = null;

BufferedOutputStream bufferedOutputStream = null;

String DataToWrite = "Give some random data …………";

try {

fileInputStream = new FileInputStream(file);

bufferedInputStream = new BufferedInputStream(fileInputStream);

// Create buffer

byte[] buffer = new byte[1024];

byte[] bufferOutput = DataToWrite.getBytes();

int bytesRead = 0;

// Now single bufferReader read will read 1024 bytes of data.

while ((bytesRead = bufferedInputStream.read(buffer)) != -1) {

System.out.println("--------------------------------");

System.out.println(new String(buffer, 0, bytesRead));

}

fileOutputStream = new FileOutputStream("bufferedIOdemo.txt");

// We can provide the fixed Buffer Size.

bufferedOutputStream = new BufferedOutputStream(bufferedOutputStream, bufferOutput.length + 1);

bufferedOutputStream.write(bufferOutput);

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

if (fileInputStream != null) {

fileInputStream.close();

}

if (bufferedInputStream != null) {

bufferedInputStream.close();

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

Once you are done writing data to a Java BufferedOutputStream you should close it. You close an BufferedOutputStream by calling its close() method. Closing a BufferedOutputStream also closes the underlying OutputStream that the BufferedOutputStream is reading data from. Here is an example of closing a Java BufferedOutputStream:

**PushbackInputStream**

The PushbackInputStream is intended to be used when you parse data from an InputStream. Sometimes you need to read ahead a few bytes to see what is coming, before you can determine how to interpret the current byte. The PushbackInputStream allows you to do that. Well, actually it allows you to push the read bytes back into the stream. These bytes will then be read again the next time you call read().

The Java PushbackInputStream is a subclass of the [**Java InputStream**](https://jenkov.com/tutorials/java-io/inputstream.html) so it inherits its public methods - read(), close() etc. The PushbackInputStream is similar to the [**Java PushbackReader**](https://jenkov.com/tutorials/java-io/pushbackreader.html), except the PushbackInputStream reads raw bytes, and the PushbackReader reads characters (text).

Example:

PushbackInputStream input = new PushbackInputStream(

new FileInputStream("c:\\data\\input.txt"));

int data = input.read();

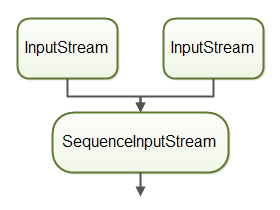
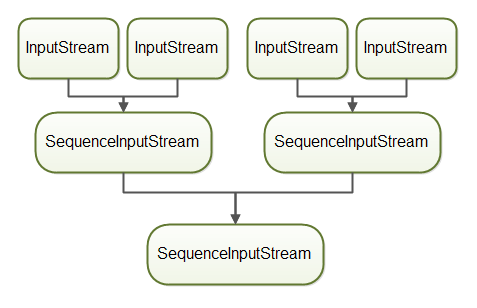
input.unread(data);

Closing a PushbackInputStream is done by calling its close() method. Here is how closing a PushbackInputStream looks:

pushbackInputStream.close();

**SequenceInputStream:**

The Java SequenceInputStream combines two or more other InputStream's into one. First the SequenceInputStream will read all bytes from the first InputStream, then all bytes from the second InputStream. That is the reason it is called a *SequenceInputStream*, since the InputStream instances are read in sequence.

Example:

InputStream input1 = new FileInputStream("c:\\data\\file1.txt");

InputStream input2 = new FileInputStream("c:\\data\\file2.txt");

SequenceInputStream sequenceInputStream =

new SequenceInputStream(input1, input2);

int data = sequenceInputStream.read();

while(data != -1){

System.out.println(data);

data = sequenceInputStream.read();

}

Closing a SequenceInputStream is done by calling its close() method. Here is how closing a SequenceInputStream looks:

sequenceInputStream.close();

**DataInputStream:**

The Java DataInputStream class, java.io.DataInputStream, enables you to read Java primitives (int, float, long etc.) from an InputStream instead of only raw bytes. You wrap an InputStream in a DataInputStream and then you can read Java primitives via ' the DataInputStream. That is why it is called DataInputStream - because it reads data (numbers) instead of just bytes.

DataInputStream dataInputStream = **new** DataInputStream(

**new** FileInputStream("binary.data"));

**int** aByte = input.read();

**int** anInt = input.readInt();

**float** aFloat = input.readFloat();

**double** aDouble = input.readDouble();

//etc.

input.close();

**DataOutputStream:**

The Java DataOutputStream class enables you to write Java primitives to OutputStream's instead of only bytes. You wrap an OutputStream in a DataOutputStream and then you can write primitives to it. That is why it is called a *DataOutputStream* - because you can write int, long, float and double values to the OutputStream, and not just raw bytes.

import java.io.\*;

public class DataOutputStreamExample {

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

DataOutputStream dataOutputStream =

new DataOutputStream(

new FileOutputStream("data/data.bin"));

dataOutputStream.writeInt(123);

dataOutputStream.writeFloat(123.45F);

dataOutputStream.writeLong(789);

dataOutputStream.close(); // Closing Data output stream

DataInputStream dataInputStream =

new DataInputStream(

new FileInputStream("data/data.bin"));

int int123 = dataInputStream.readInt();

float float12345 = dataInputStream.readFloat();

long long789 = dataInputStream.readLong();

dataInputStream.close();// Closing Data InputSteam

System.out.println("int123 = " + int123);

System.out.println("float12345 = " + float12345);

System.out.println("long789 = " + long789);

}

}

**PrintStream:**

The Java PrintStream class (java.io.PrintStream) enables you to write formatted data to an underlying OutputStream. The PrintStream class can format primitive types like int, long etc. formatted as text, rather than as their byte values. That is why it is called a *PrintStream*, because it formats the primitive values as text - like they would look when printed to the screen (or printed to paper).

PrintStream Example:

PrintStream printStream = new PrintStream(outputStream);

printStream.print(true);

printStream.print((int) 123);

printStream.print((float) 123.456);

printStream.close();

This example first creates a PrintStream which is connected to an OutputStream. Second, the example prints three primitivate values to the PrintStream. Third, the example closes the PrintStream.

**System.out and System.err are PrintStreams**

You may be familiar with these two well-known PrintStream instances in Java: System.out and System.err . If you have every used any of these two streams, you have already used a PrintStream.

**Printf()**

The Java PrintStream class contains the powerful format() and printf() methods (they do exactly the same, but the name "printf" is more familiar to C-programmers). These methods allow you to mix text and data in very advanced ways, using a formatting string.

Here is a simple Java printf() example:

PrintStream printStream = new PrintStream(outputStream);

printStream.printf(Locale.UK, "Text + data: %1$", 123);

printStream.close();

**ObjectInputStream:**

The Java ObjectInputStream class (java.io.ObjectInputStream) enables you to read Java objects from an InputStream instead of just raw bytes. You wrap an InputStream in a ObjectInputStream and then you can read objects from it. Of course the bytes read must represent a valid, serialized Java object. Otherwise reading objects will fail.

Normally you will use the ObjectInputStream to read objects written (serialized) by a [**Java ObjectOutputStream**](https://jenkov.com/tutorials/java-io/objectoutputstream.html) . You will see an example of that later.

Example of ObjectInputStream:

ObjectInputStream objectInputStream =

new ObjectInputStream(new FileInputStream("object.data"));

MyClass object = (MyClass) objectInputStream.readObject();

//etc.

objectInputStream.close();

For this ObjectInputStream example to work the object you read must be an instance of MyClass, and must have been **serialized** into the file "object.data" via an ObjectOutputStream. Before you can serialize and de-serialize objects the class of the object must implement java.io.Serializable.

import java.io.\*;

public class ObjectInputStreamExample {

public static class Person implements Serializable {

public String name = null;

public int age = 0;

}

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

ObjectOutputStream objectOutputStream =

new ObjectOutputStream(new FileOutputStream("data/person.bin"));

Person person = new Person();

person.name = "Khawja Ghouse";

person.age = 25;

objectOutputStream.writeObject(person);

objectOutputStream.close();

ObjectInputStream objectInputStream =

new ObjectInputStream(new FileInputStream("data/person.bin"));

Person personRead = (Person) objectInputStream.readObject();

objectInputStream.close();

System.out.println(personRead.name);

System.out.println(personRead.age);

}

}

Output:

Khawja Ghouse

25