- IO EXCEPTION MUST BE CATCHED
- Opened streams must be closed in the finaly bloc

Byte & Character Stream :

- Commun methods : read/write/close
- read returns int & write receives int
- Byte Stream : FileInputStream/ FileOutputStream (String path)
    - Limited to one byte character
- Character Streams : FileReader/FileWriter (String path)
    - Limited to two bytes character

BufferedStream :

- Byte & Character Stream are unbuffered I/O, This means each read or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.
- Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.
- **BufferedInputStream receives a FileInputStream construct as parameter (idem for Writer)**
- **BufferedReader receives a FileReader constructor as parameter (idem for Writer)**
- It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as flushing the buffer, we can flush a buffer using the method flush or in some case there is a autoflush in some buffered class, specified by an optional constructor argument. When autoflush is enabled, certain key events cause the buffer to be flushed

Scanner :

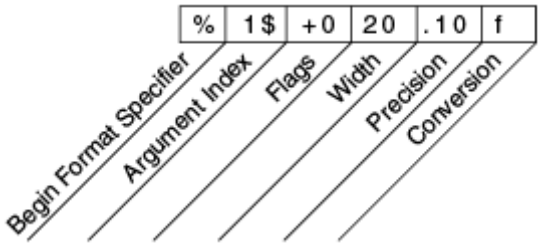
- receives buffered Stream, it’s useful for breaking down formatted input into tokens and translating individual tokens according to their data type.
- methods :
    - hasNext() or hasNextTYPE()
    - next or nextTYPE() ;
    - useLocale(Local.NAME) and useDelimiter(regex)

Formated output :

- Methods System.out.format(string,args...) (it’s simillar to C’s printf except that we use %n to return to line instead of \n.
    - In the Java programming language, the \n escape always generates the linefeed character (\u000A). Don't use \n unless you specifically want a linefeed character. To get the correct line separator for the local platform, use %n.
- some of the format specifiers :
    - d formats an integer value as a decimal value.
    - f formats a floating point value as a decimal value.
    - n outputs a platform-specific line terminator.
    - x formats an integer as a hexadecimal value.
    - s formats any value as a string.
    - tB formats an integer as a locale-specific month name.

Note: Except for `%%` and `%n`, all format specifiers must match an argument. If they don't, an exception is thrown.
Example :

- `System.out.format("%f, %1$+020.10f %n", Math.PI);`
  - Ouputs : 3.141593, +00000003.1415926536



- We can also specify < to match the same argument as the previous specifier. Thus the example could have said: System.out.format("%f, %<+020.10f %n", Math.PI);

Console :

- System.console returns an object of type Console if the console exists, else it returns Null
- we can use readLine(String prompt) and readPassword(String prompt) on the console Object

DATA Streams (Write primitive data and String values to binary file)

- all data streams classes implement either DataInput or DataOuput **interface**
- **the most widely-used classes are DataInputStream and DataOutPutStream**
- All implementations of DataInput methods use **EOFException** instead of return values.
- Constructor : `DataOutputStream(new BufferedOutputStream(new FileOutputStream(dataFile)));` (or Input)
- **Methods :**
  - Writing data : writeDouble,writeInt,writeUTF (String in a modified form of UTF-8.)
  - Reading data : ReadDouble,readInt,readUTF
- `DataStreams` uses one very bad programming technique: it uses floating point numbers to represent monetary values. In general, floating point is bad for precise values. It's particularly bad for decimal fractions, because common values (such as `0.1`) do not have a binary representation. The correct type to use for currency values is `java.math.BigDecimal`

Object Streams (serialization)

- Methods : writeObject and readObject

You might wonder what happens if two objects on the same stream both contain references to a single object. Will they both refer to a single object when they're read back? The answer is "yes." A stream can only contain one copy of an object, though it can contain any number of references to it. Thus if you explicitly write an object to a stream twice, you're really writing only the reference twice. For example, if the following code writes an object `ob` twice to a stream:

```
Object ob = new Object();
out.writeObject(ob);
out.writeObject(ob);
```

Each `writeObject` has to be matched by a `readObject`, so the code that reads the stream back will look something like this:

```
Object ob1 = in.readObject();
Object ob2 = in.readObject();
```

This results in two variables, `ob1` and `ob2`, that are references to a single object.

However, if a single object is written to two different streams, it is effectively duplicated — a single program reading both streams back will see two distinct objects.

- If readObject() doesn't return the object type expected, attempting to cast it to the correct type may throw a **ClassNotFoundException**.

source : Oracle’s java tutorial.