**Java IO Tutorial**

Ref - http://docs.oracle.com/javase/tutorial/essential/io/index.html

# Byte Streams

Programs use byte streams to perform input and output of 8-bit bytes. All byte stream classes are descended from [InputStream](http://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html" \t "_blank) and [OutputStream](http://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html" \t "_blank).

There are many byte stream classes such as  [FileInputStream](http://docs.oracle.com/javase/8/docs/api/java/io/FileInputStream.html" \t "_blank) and [FileOutputStream](http://docs.oracle.com/javase/8/docs/api/java/io/FileOutputStream.html" \t "_blank). Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

## When Not to Use Byte Streams

Byte streams should only be used for the most primitive I/O.

# Character Streams

Character streams are often "wrappers" for byte streams. The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes. FileReader, for example, uses FileInputStream, while FileWriter uses FileOutputStream.

There are two general-purpose byte-to-character "bridge" streams: [InputStreamReader](http://docs.oracle.com/javase/8/docs/api/java/io/InputStreamReader.html" \t "_blank) and [OutputStreamWriter](http://docs.oracle.com/javase/8/docs/api/java/io/OutputStreamWriter.html" \t "_blank). Use them to create character streams when there are no prepackaged character stream classes that meet your needs.

## Line-Oriented I/O

Character I/O usually occurs in bigger units than single characters. One common unit is the line: a string of characters with a line terminator at the end. A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n").

# Buffered Streams

Most of the examples we've seen so far use *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.

To reduce this kind of overhead, the Java platform implements *buffered* I/O streams. 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.

A program can convert an unbuffered stream into a buffered stream using the wrapping idiom we've used several times now, where the unbuffered stream object is passed to the constructor for a buffered stream class. Here's how you might modify the constructor invocations in the CopyCharacters example to use buffered I/O:

inputStream = new BufferedReader(new FileReader("xanadu.txt"));

outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));

There are four buffered stream classes used to wrap unbuffered streams: [BufferedInputStream](http://docs.oracle.com/javase/8/docs/api/java/io/BufferedInputStream.html" \t "_blank) and [BufferedOutputStream](http://docs.oracle.com/javase/8/docs/api/java/io/BufferedOutputStream.html" \t "_blank) create buffered byte streams, while [BufferedReader](http://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html" \t "_blank)and [BufferedWriter](http://docs.oracle.com/javase/8/docs/api/java/io/BufferedWriter.html" \t "_blank) create buffered character streams.

## Flushing Buffered Streams

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.

Some buffered output classes support *autoflush*, specified by an optional constructor argument. When autoflush is enabled, certain key events cause the buffer to be flushed. For example, an autoflush PrintWriter object flushes the buffer on every invocation of println or format. See [Formatting](http://docs.oracle.com/javase/tutorial/essential/io/formatting.html) for more on these methods.

To flush a stream manually, invoke its flush method. The flush method is valid on any output stream, but has no effect unless the stream is buffered.

# Scanning and Formatting

Programming I/O often involves translating to and from the neatly formatted data humans like to work with. To assist you with these chores, the Java platform provides two APIs. The [scanner](http://docs.oracle.com/javase/tutorial/essential/io/scanning.html) API breaks input into individual tokens associated with bits of data. The [formatting](http://docs.oracle.com/javase/tutorial/essential/io/formatting.html) API assembles data into nicely formatted, human-readable form.

# Scanning

Objects of type [Scanner](http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html) are useful for breaking down formatted input into tokens and translating individual tokens according to their data type.

## Breaking Input into Tokens

By default, a scanner uses white space to separate tokens. (White space characters include blanks, tabs, and line terminators. For the full list, refer to the documentation for[Character.isWhitespace](http://docs.oracle.com/javase/8/docs/api/java/lang/Character.html#isWhitespace-char-).)

# I/O from the Command Line

## Standard Streams

Standard Streams are a feature of many operating systems. By default, they read input from the keyboard and write output to the display. They also support I/O on files and between programs, but that feature is controlled by the command line interpreter, not the program.

The Java platform supports three Standard Streams: Standard Input, accessed through System.in; Standard Output, accessed through System.out; and Standard Error, accessed through System.err. These objects are defined automatically and do not need to be opened. Standard Output and Standard Error are both for output; having error output separately allows the user to divert regular output to a file and still be able to read error messages.

You might expect the Standard Streams to be character streams, but, for historical reasons, they are byte streams. System.out and System.err are defined as [PrintStream](http://docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html" \t "_blank)objects. Although it is technically a byte stream, PrintStream utilizes an internal character stream object to emulate many of the features of character streams.

By contrast, System.in is a byte stream with no character stream features. To use Standard Input as a character stream, wrap System.in in InputStreamReader.

InputStreamReader cin = new InputStreamReader(System.in);

## The Console

A more advanced alternative to the Standard Streams is the Console. This is a single, predefined object of type [Console](http://docs.oracle.com/javase/8/docs/api/java/io/Console.html) that has most of the features provided by the Standard Streams, and others besides. The Console is particularly useful for secure password entry. The Console object also provides input and output streams that are true character streams, through its reader and writer methods.

Before a program can use the Console, it must attempt to retrieve the Console object by invoking System.console(). If the Console object is available, this method returns it. IfSystem.console returns NULL, then Console operations are not permitted, either because the OS doesn't support them or because the program was launched in a noninteractive environment.

The Console object supports secure password entry through its readPassword method. This method helps secure password entry in two ways. First, it suppresses echoing, so the password is not visible on the user's screen. Second, readPassword returns a character array, not a String, so the password can be overwritten, removing it from memory as soon as it is no longer needed.

# Data Streams

Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values. All data streams implement either the [DataInput](http://docs.oracle.com/javase/8/docs/api/java/io/DataInput.html) interface or the [DataOutput](http://docs.oracle.com/javase/8/docs/api/java/io/DataOutput.html" \t "_blank) interface.

# Object Streams

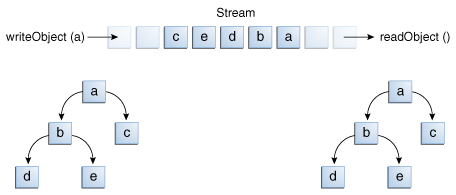
Just as data streams support I/O of primitive data types, object streams support I/O of objects. Most, but not all, standard classes support serialization of their objects. Those that do implement the marker interface [Serializable](http://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html" \t "_blank).

The object stream classes are [ObjectInputStream](http://docs.oracle.com/javase/8/docs/api/java/io/ObjectInputStream.html" \t "_blank) and [ObjectOutputStream](http://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutputStream.html" \t "_blank). These classes implement [ObjectInput](http://docs.oracle.com/javase/8/docs/api/java/io/ObjectInput.html" \t "_blank) and [ObjectOutput](http://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutput.html" \t "_blank), which are subinterfaces of DataInput andDataOutput. That means that all the primitive data I/O methods covered in [Data Streams](http://docs.oracle.com/javase/tutorial/essential/io/datastreams.html) are also implemented in object streams. So an object stream can contain a mixture of primitive and object values.

## Output and Input of Complex Objects

The writeObject and readObject methods are simple to use, but they contain some very sophisticated object management logic. This isn't important for a class like Calendar, which just encapsulates primitive values. But many objects contain references to other objects. If readObject is to reconstitute an object from a stream, it has to be able to reconstitute all of the objects the original object referred to. These additional objects might have their own references, and so on. In this situation, writeObject traverses the entire web of object references and writes all objects in that web onto the stream. Thus a single invocation of writeObject can cause a large number of objects to be written to the stream.

This is demonstrated in the following figure, where writeObject is invoked to write a single object named **a**. This object contains references to objects **b** and **c**, while **b**contains references to **d** and **e**. Invoking writeobject(a) writes not just **a**, but all the objects necessary to reconstitute **a**, so the other four objects in this web are written also. When **a** is read back by readObject, the other four objects are read back as well, and all the original object references are preserved.

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.