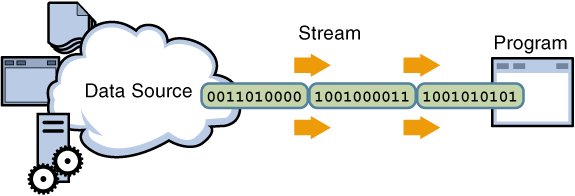
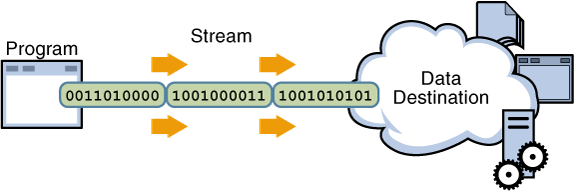
**Input Output: java.io package**

* The package **java.io** is used for using the classes suitable for taking input: from program, disks, objects, etc. and producing output: to program, disks, objects, printers, etc.
* I/O operations are performed through **streams**.
  + **Input streams**: represent sources that you can read data from and
  + A stream is a sequence of data. A program uses an *input stream* to read data from a source, one item at a time



Reading information into a program (Fig. Oracle Java Tutorial Site)

* + **Output streams**: represent sources that you can write data to.
  + A program uses an *output stream* to write data to a destination, one item at time



Writing information from a program (Fig. Oracle Java Tutorial Site)

* These streams are subdivided into different types like:
  + **Byte-oriented streams**
    - **Byte streams** in Java are used for handling I/O of bytes. Byte streams are generally suitable for reading and writing binary data like **audio**, **video**, **images** etc. However the most basic way of reading and writing is using byte streams
  + **Character-oriented streams**
    - Java has also defined the **character streams** that are used for input and outputs of character types.
* **Byte Stream Classes**
  + There are two top-level abstract classes called **InputStream** and **OutputStream** for byte streams I/O.
  + Each of these classes has many concrete subclasses that are used to handle different I/O like disks, memory buffers, files, etc. To use stream classes you must use **java.io** package. The below is the list of some byte stream classes.

**Stream Class**  **Meaning**

BufferedInputStream Buffered input stream

BufferedOutputStream Buffered output stream

ByteArrayInputStream Input stream that reads from a byte array

ByteArrayOutputStream Output stream that writes to a byte array

DataInputStream An input stream that contains methods for reading the Java standard data types

DataOutputStream An output stream that contains methods for writing the Java standard data types

FileInputStream Input stream that reads from a file

FileOutputStream Output stream that writes to a file

FilterInputStream Implements **InputStream**

FilterOutputStream Implements **OutputStream**

InputStream Abstract class that describes stream input

OutputStream Abstract class that describes stream output

PipedInputStream Input pipe

PipedOutputStream Output pipe

PrintStream Output stream that contains **print( )** and **println( )**

PushbackInputStream Input stream that supports one-byte which returns a byte to the input stream

RandomAccessFile Supports random access file I/O

SequenceInputStream Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

* **Character Stream Classes**
  + The two top-level abstract classes **Reader** and **Writer** define the character streams.
  + These classes handle Unicode character streams. Java uses concrete subclasses of the above two classes for character I/O handling. The below is the list of some character stream classes.

**Stream Class**  **Meaning**

BufferedReader Buffered input character stream

BufferedWriter Buffered output character stream

CharArrayReader Input stream that reads from a character array

CharArrayWriter Output stream that writes to a character array

FileReader Input stream that reads from a file

FileWriter Output stream that writes to a file

FilterReader Filtered reader

FilterWriter Filtered writer

InputStreamReader Input stream that translates bytes to characters

LineNumberReader Input stream that counts lines

OutputStreamWriter Output stream that translates characters to bytes

PipedReader Input pipe

PipedWriter Output pipe

PrintWriter Output stream that contains **print( )** and **println( )**

PushbackReader Input stream that allows characters to be returned to the input stream

Reader Abstract class that describes character stream input

StringReader Input stream that reads from a string

StringWriter Output stream that writes to a string

Writer Abstract class that describes character stream output

* **Using FileInputStream and FileOutputStream Classes**
  + These two classes are used for reading and writing files. These classes create byte streams linked to files
  + We use the following commonly used constructors for file reading and writing purposes.

*FileInputStream(String fileName) throws FileNotFoundException*

* + Creates a **FileInputStream** by opening a connection to an actual file, the file named by the path name **fileName** in the file system.
  + If the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading then a **FileNotFoundException** is thrown.

*FileOutputStream(String filename, boolean append) throws FileNotFoundException*

* + Creates an output file stream to write to the file with the specified name
  + If the second argument is true, then bytes will be written to the end of the file rather than the beginning.
  + If the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason then a **FileNotFoundException** is thrown.
  + After we use the file we must close the file by calling close method defined in both the classes above. Its definition format is:

*void close() throws IOException*

* + Reading from the file can be done by using read method defined in **FileInputStream** class whose definition syntax is:

*int read() throws IOException*

* + The above method reads each byte from the file and returns the byte as an integer value. -1 is returned if end of file is obtained.
  + Writing to the file is done by using the write method defined in **FileOutputStream** class whose syntax is:

*void write(int byteval) throws IOException*

* + Here though byteval is an integer only the low-order eight bits (byte) are written.

**Program**

import java.io.\*;

public class FileReadWrite {

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

FileInputStream in = null;

FileOutputStream out = null;

try {

in = new FileInputStream("file1.txt");

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

int c;

while ((c = in.read()) != -1) {

out.write(c);

}

} finally {

if (in != null) {

in.close();

}

if (out != null) {

out.close();

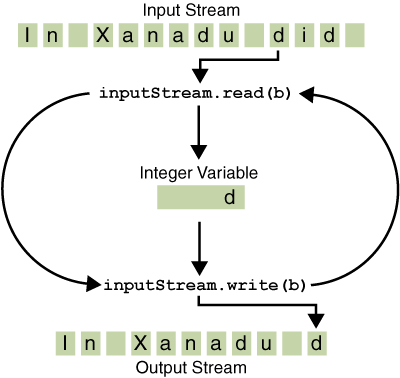
}

}

}

}

* This spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time, as shown in the following figure.



Simple byte stream input and output.

* Notice that read() returns an int value.
* If the input is a stream of bytes, why doesn't read() return a byte value?
* Using a int as a return type allows read() to use -1 to indicate that it's reached the end of the stream.

### Always Close Streams

* Closing a stream when it's no longer needed is very important — so important that FileReadWrite  uses a finally block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.
* One possible error is that FileReadWrite  was unable to open one or both files. When that happens, the stream variable corresponding to the file never changes from its initial null value. That's why FileReadWrite  makes sure that each stream variable contains an object reference before invoking close.

### When Not to Use Byte Streams

* FileReadWrite  seems like a normal program, but it actually represents a kind of low-level I/O that you should avoid. Since xanadu.txt contains character data, the best approach is to use [character streams](file:///E:\DWIT\tutorial\tutorial\essential\io\charstreams.html). There are also streams for more complicated data types. Byte streams should only be used for the most primitive I/O.
* So why talk about byte streams? Because all other stream types are built on byte streams.
* **Using FileReader and FileWriter Classes**
  + These two classes are used for reading and writing files.
  + These classes create character streams linked to files.
  + **FileReader** is the direct descendent of **InputStreamReader** class that acts as the bridge between byte stream and character stream.
  + Similarly, **FileWriter** is the direct descendent of **OutputStreamReader** class that acts as the bridge between byte stream and character stream.

*FileReader(String fileName) throws FileNotFoundException*

* + Creates a **FileReader** by opening a connection to an actual file, the file named by the path name **fileName** in the file system. If the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading then a **FileNotFoundException** is thrown.

*FileWriter(String filename, boolean append) throws IOException*

* + Creates an output file stream to write to the file with the specified name
  + If the second argument is true, then character will be written to the end of the file rather than the beginning. If the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason then an **IOException** is thrown.
  + After we use the file we must close the file by calling close method defined in both the classes above. Its definition format is:

*void close() throws IOException*

* + Reading from the file can be done by using read method defined in **FileReader** class whose definition syntax is:

*int read() throws IOException*

* + The above method reads each character from the file and returns the character as an integer value. -1 is returned if end of file is obtained.
  + Writing to the file is done by using the write method defined in **FileWriter** class whose syntax is:

*void write(int c) throws IOException*

* + c is an integer representing the character to be written.
  + **Program**

Change the class name **FileInputStream** to **FileReader** and **FileOutputStream** to **FileWriter** in the above program and run.

* + Using Classes for Line Reading and Writing
  + A line is terminated by newline character (“\n”) or some system uses carriage return character followed by newline character (“\r\n”) or carriage return character (“\r”) only. To perform the line based I/O we can use two classes **BufferedReader** and **PrintWriter**.
* **Buffer 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.

inputStream = new BufferedReader(new FileReader("file1.txt"));

outputStream = new BufferedWriter(new FileWriter("file2.txt"));

* + There are four buffered stream classes used to wrap unbuffered streams:
    - [BufferedInputStream](http://java.sun.com/javase/6/docs/api/java/io/BufferedInputStream.html) and [BufferedOutputStream](http://java.sun.com/javase/6/docs/api/java/io/BufferedOutputStream.html) create buffered byte streams
    - While [BufferedReader](http://java.sun.com/javase/6/docs/api/java/io/BufferedReader.html) and [BufferedWriter](http://java.sun.com/javase/6/docs/api/java/io/BufferedWriter.html) 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.
* 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.
* **BufferedReader**
  + This class is inherited from **Reader** class.
  + It reads the text from the character input stream using the character buffer that helps efficient reading of characters, arrays, and lines.
  + The buffering causes the input streams to be stored in temporary memory location so that the costly operation “read” that would be otherwise done from the file by converting bytes to the character is efficiently done through the buffer.

*BufferedReader(Reader in)*

The example of using the constructor is as given below:

*BufferedReader in = new BufferedReader(new FileReader("my.txt"));*

Some methods related to this class are given below:

* *int read()throws IOException*: Reads a single character. This method overrides read method in the class **Reader**.
* *String readLine() throws IOException*: Reads a line of text and returns the string with line content without line termination character. Null is returned if end of stream is met.
* *void close() throws* [*IOException*](file:///C:\Program Files\Java\docs\api\java\io\IOException.html): Close the stream.
* *long skip(long n) throws* [*IOException*](file:///C:\Program Files\Java\docs\api\java\io\IOException.html): Skip **n** number of characters and returns the number of characters actually skipped.
* **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** API breaks input into individual tokens associated with bits of data.
  + The **formatting** API assembles data into nicely-formatted, human readable form.
* **Scanner**
  + Objects of type [Scanner](http://java.sun.com/javase/6/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)

import java.io.\*;

import java.util.Scanner;

public class ScanDemo {

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

Scanner s = null;

try {

s = new Scanner(new BufferedReader(new FileReader("file1.txt")));

while (s.hasNext()) {

System.out.println(s.next());

}

} finally {

if (s != null) {

s.close();

}

}

}

}

* + - This program invokes Scanner's close method when it is done with the scanner object.
    - Even though a scanner is not a stream, you need to close it to indicate that you're done with its underlying stream.
    - To use a different token separator, invoke useDelimiter(), specifying a regular expression. For example, suppose you wanted the token separator to be a comma, optionally followed by white space. You would invoke,

s.useDelimiter(",\\s\*");

* **Formatting**
  + Stream objects that implement formatting are instances of either [PrintWriter](http://java.sun.com/javase/6/docs/api/java/io/PrintWriter.html), a character stream class, and [PrintStream](http://java.sun.com/javase/6/docs/api/java/io/PrintStream.html), a byte stream class.
  + Note:  The only PrintStream objects you are likely to need are [System.out](#out) and [System.err](#err).
  + When you need to create a formatted output stream, instantiate PrintWriter, not PrintStream.
* **PrintWriter**
  + This class is inherited from **Writer** class. It writes formatted texts to the output stream. Some generally used constructors of this class are:
* *PrintWriter(*[*OutputStream*](file:///C:\Program Files\Java\docs\api\java\io\OutputStream.html)*out)*: Creates a new **PrintWriter** with automatic line flushing using the **OutputStream** provided.
* *PrintWriter(*[*String*](file:///C:\Program Files\Java\docs\api\java\lang\String.html)*fileName) throws* [*FileNotFoundException*](file:///C:\Program Files\Java\docs\api\java\io\FileNotFoundException.html): Creates a new **PrintWriter** with automatic line flushing using the file provided. If the file is already there it becomes file with zero size, otherwise new file is created and the output will be written to the file and is buffered.

Here are some methods related to this class

* *void write(int c)*: Writes a single character.
* *void write(String s)* : Writes a string.
* *void print(****TYPE****b)* : Print **TYPE** value (this is notation for us to understand, the actual text in place of TYPE would boolean, int, long, float, double, char, char[], String, or Object ).
* *void println()*: Terminate the current line using line separator string. Line separator is not necessarily a single newline character.
* *void println(****TYPE****x)*: Print **TYPE** value and then terminate the line (this is notation for us to understand, the actual text in place of TYPE would boolean, int, long, float, double, char, char[], String, or Object ).

**Program**

import java.io.\*;

class LineIO {

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

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

PrintWriter pOut = new PrintWriter(System.out, true);//line flushing true

String str[] = new String[100];

System.out.println("Enter lines of text.");

System.out.println("Enter 'stop' to quit.");

for(int i=0; i<100; i++) {

str[i] = br.readLine();

if(str[i].equals("stop")) break;

}

System.out.println("\nHere is the output:");

for(int i=0; i<100; i++) {

if(str[i].equals("stop"))

break;

pOut.println(str[i]);

}

}

}

Examples of reading console input using BufferedReader

**Reading Characters**

import java.io.\*;

class BRRead {

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

char c;

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

System.out.print("Enter a character:");

c = (char)br.read();

System.out.print("Input character is " + c);

}

}

**Reading Strings**

import java.io.\*;

class BRReadLines {

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

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

System.out.print("a = ");

int a = Integer.parseInt(br.readLine());

System.out.print("b = ");

int b = Integer.parseInt(br.readLine());

System.out.println("Sum = " + (a + b));

}

}

**File IO**

* The [File](http://java.sun.com/javase/6/docs/api/java/io/File.html) class makes it easier to write platform-independent code that examines and manipulates files.
* Why create a File object for a file that doesn't exist? A program can use the object to parse a file name. Also, the file can be created by passing the File object to the constructor of some classes, such as FileWriter.
* If the file *does* exist, a program can examine its attributes and perform various operations on the file, such as renaming it, deleting it, or changing its permissions.

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

* It's worth mentioning that [File.compareTo()](#compareTo(java.io.File))would not consider a and b to be the same. Even though they refer to the same file, the names used to construct them is different.
* The delete method deletes the file immediately, while the deleteOnExit method deletes the file when the virtual machine terminates.
* The setLastModified sets the modification date/time for the file.
* The renameTo() method renames the file. Note that the file name string behind the File object remains unchanged, so the File object will not refer to the renamed file.

### Working With Directories

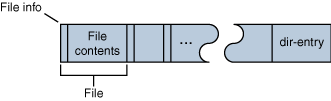
* File has some useful methods for working with directories.
* The mkdir method creates a directory. The mkdirs method does the same thing, after first creating any parent directories that don't yet exist.
* The list and listFiles methods list the contents of a directory. The list method returns an array of String file names, while listFiles returns an array ofFile objects.

### Static Methods

* File contains some useful static methods.
* The createTempFile method creates a new file with an unique name, and returns a File object referring to it
* The listRoots returns a list of file system root names. On Microsoft Windows, this will be the root directories of mounted drives, such as a:\ and c:\. On UNIX and Linux systems, this will be the root directory, /.

**Random Access File**

* *Random access files* permit nonsequential, or random, access to a file's contents.
* Consider the archive format known as ZIP. A ZIP archive contains files and is typically compressed to save space. It also contain a directory entry at the end that indicates where the various files contained within the ZIP archive begin, as shown in the following figure.



A ZIP archive.

* Suppose that you want to extract a specific file from a ZIP archive. If you use a sequential access stream, you have to:

1. Open the ZIP archive.
2. Search through the ZIP archive until you located the file you wanted to extract.
3. Extract the file.
4. Close the ZIP archive.

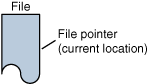
* Using this procedure, on average, you'd have to read half the ZIP archive before finding the file that you wanted to extract. You can extract the same file from the ZIP archive more efficiently by using the seek feature of a random access file and following these steps:
* Open the ZIP archive.
* Seek to the directory entry and locate the entry for the file you want to extract from the ZIP archive.
* Seek (backward) within the ZIP archive to the position of the file to extract.
* Extract the file.
* Close the ZIP archive.
* This algorithm is more efficient because you read only the directory entry and the file that you want to extract.
* The [java.io.RandomAccessFile](http://java.sun.com/javase/6/docs/api/java/io/RandomAccessFile.html) class implements both the DataInput and DataOutput interfaces and therefore can be used for both reading and writing.RandomAccessFile is similar to FileInputStream and FileOutputStream in that you specify a file on the native file system to open when you create it.
* When you create a RandomAccessFile, you must indicate whether you will be just reading the file or also writing to it. (You have to be able to read a file in order to write it.) The following code creates a RandomAccessFile to read the file named farrago.txt:

new RandomAccessFile("file1.txt", "r");

* And this one opens the same file for both reading and writing:

new RandomAccessFile("xanadu.txt", "rw");

* After the file has been opened, you can use the common read or write methods defined in the DataInput and DataOutput interfaces to perform I/O on the file.
* RandomAccessFile supports the notion of a *file pointer*. The file pointer indicates the current location in the file. When the file is first created, the file pointer is set to 0, indicating the beginning of the file. Calls to the read and write methods adjust the file pointer by the number of bytes read or written.



A ZIP file has the notion of a current file pointer.

* In addition to the normal file I/O methods that implicitly move the file pointer when the operation occurs, RandomAccessFile contains three methods for explicitly manipulating the file pointer.
* int skipBytes(int) — Moves the file pointer forward the specified number of bytes
* void seek(long) — Positions the file pointer just before the specified byte
* long getFilePointer() — Returns the current byte location of the file pointer

**LAB:**

* Write a program to create a text file in the path c:\java\abc.txt and check whether that file is exists. Using the command exists(), isDirectory(), isFile(), getName() and getAbsolutePath().
* Write a program to rename the given file, after renaming the file delete the renamed file. (Accept the file name using command line arguments.)
* Write a program to create a directory and check whether the directory is created.