Text & Binary I/O

CSGE601021 Dasar-Dasar Pemrograman 2

Fakultas Ilmu Komputer Universitas Indonesia

Text IO

The `File` Class

- The File class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion. The filename is a string. The File class is a wrapper class for the file name and its directory path.

java.io.File	
`+File(pathname: String)`	Creates a `File` object for the specified path name. The path name may be a directory or a file.
`+File(parent: String, child: String)`	Creates a `File` object for the child under the directory parent. The child may be a file name or a subdirectory.
`+File(parent: File, child: String)`	Creates a `File` object for the child under the directory parent. The parent is a `File` object. In the preceding constructor, the parent is a string.
`+exists(): boolean`	Returns true if the file or the directory represented by the `File` object exists.
`+canRead(): boolean`	Returns true if the file represented by the `File` object exists and can be read.
`+canWrite(): boolean`	Returns true if the file represented by the `File` object exists and can be written.
`+isDirectory(): boolean`	Returns true if the `File` object represents a directory.
`+isFile(): boolean`	Returns true if the `File` object represents a file.
`+isAbsolute(): boolean`	Returns true if the `File` object is created using an absolute path name.
`+isHidden(): boolean`	Returns true if the file represented in the `File` object is hidden. The exact definition of *hidden* is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period(.) character.
`+getAbsolutePath(): String`	Returns the complete absolute file or directory name represented by the `File` object.
`+getCanonicalPath(): String`	Returns the same as `getAbsolutePath()` except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).
`+getName(): String`	Returns the last name of the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getName()` returns `test.dat`.
`+getPath(): String`	Returns the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getPath()` returns `c:\book\test.dat`.
`+getParent(): String`	Returns the complete parent directory of the current directory or the file represented by the `File` object. For example, new `File("c:\\book\\test.dat").getParent()` returns `c:\book`.
`+lastModified(): long`	Returns the time that the file was last modified.
`+length(): long`	Returns the size of the file, or 0 if it does not exist or if it is a directory.
`+listFile(): File[]`	Returns the files under the directory for a directory `File` object.
`+delete(): boolean`	Deletes the file or directory represented by this `File` object.The method returns true if the deletion succeeds.
`+renameTo(dest: File): boolean`	Renames the file or directory represented by this `File` object to the specified name represented in dest. The method returns true if the operation succeeds.
`+mkdir(): boolean`	Creates a directory represented in this `File` object. Returns true if the the directory is created successfully.
`+mkdirs(): boolean`	Same as `mkdir()` except that it creates directory along with its parent directories if the parent directories do not exist.

Exploring File Properties

- Write a program that demonstrates how to create files in a platform-independent way and use the methods in the File class to obtain their properties. The following figures show a sample run of the program on Windows and on Unix

Command Prompt

```
C:\book>java TestFileClass
Does it exist? true
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
What is its absolute path? C:\book\.\image\us.gif
What is its canonical path? C:\book\image\us.gif
What is its name? us.gif
What is its path? .\image\us.gif
When was it last modified? Sat May 08 14:00:34 EDT 1999
What is the path separator? ;
What is the name separator? \

C:\book>
```

Command Prompt - telnet panda

```
$ pwd
/home/liang/book
$ java TestFileClass
Does it exist? true
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
What is its absolute path? /home/liang/book/./image/us.gif
What is its canonical path? /home/liang/book/image/us.gif
What is its name? us.gif
What is its path? ./image/us.gif
When was it last modified? Wed Jan 23 11:00:14 EST 2002
What is the path separator? :
What is the name separator? /
$
```

Exploring File Properties (2)

```
public class TestFileClass {
  public static void main(String[] args) {
    java.io.File file = new java.io.File("image/us.gif");
    System.out.println("Does it exist? " + file.exists());
    System.out.println("The file has " + file.length() + " bytes");
    System.out.println("Can it be read? " + file.canRead());
    System.out.println("Can it be written? " + file.canWrite());
    System.out.println("Is it a directory? " + file.isDirectory());
    System.out.println("Is it a file? " + file.isFile());
    System.out.println("Is it absolute? " + file.isAbsolute());
    System.out.println("Is it hidden? " + file.isHidden());
    System.out.println("Absolute path is " + file.getAbsolutePath());
    System.out.println("Last modified on " +
                        new java.util.Date(file.lastModified()));
  }
}
```

Text I/O

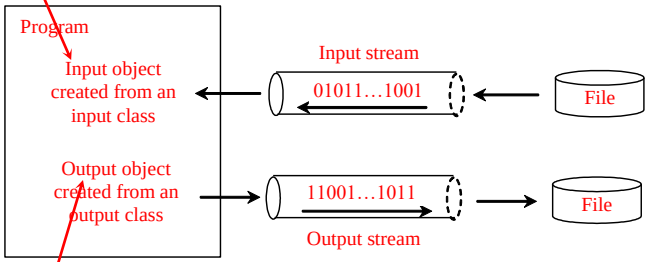
- A File object **encapsulates** the properties of a file or a path, but **does not** contain the methods for reading/writing data from/to a file.
- In order to perform I/O, **you need to create objects using appropriate Java I/O** classes.
- The objects contain the methods for reading/writing data from/to a file.
- You already know one class already for input ….

Scanner

How is I/O Handled in Java?

A File object encapsulates the properties of a file or a path, but does not contain the methods for reading/writing data from/to a file. In order to perform I/O, you need to create objects using appropriate Java I/O classes.

```
Scanner input = new Scanner(new File("temp.txt"));
System.out.println(input.nextLine());
```




```
PrintWriter output = new PrintWriter("temp.txt");
output.println("Java 101");
output.close();
```

Reading Data Using Scanner

java.util.Scanner	
+Scanner(source: File)	Creates a Scanner object to read data from the specified file.
+Scanner(source: String)	Creates a Scanner object to read data from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has another token in its input.
+next(): String	Returns next token as a string.
+nextByte(): byte	Returns next token as a byte.
+nextShort(): short	Returns next token as a short.
+nextInt(): int	Returns next token as an int.
+nextLong(): long	Returns next token as a long.
+nextFloat(): float	Returns next token as a float.
+nextDouble(): double	Returns next token as a double.
+useDelimiter(pattern: String): Scanner	Sets this scanner's delimiting pattern.

Writing Data Using PrintWriter

java.io.PrintWriter	
+PrintWriter(filename: String)	Creates a PrintWriter for the specified file.
+print(s: String): void	Writes a string.
+print(c: char): void	Writes a character.
+print(cArray: char[]): void	Writes an array of character.
+print(i: int): void	Writes an int value.
+print(l: long): void	Writes a long value.
+print(f: float): void	Writes a float value.
+print(d: double): void	Writes a double value.
+print(b: boolean): void	Writes a boolean value.
Also contains the overloaded println methods.	A println method acts like a print method; additionally it prints a line separator. The line separator string is defined by the system. It is \r\n on Windows and \n on Unix.
Also contains the overloaded printf methods.	The printf method was introduced in §3.6, “Formatting Console Output and Strings.”

Try-with-resources

- **Programmers often forget to close the file.**
- JDK 7 provides the followings new try-with-resources syntax that automatically closes the files.

```
try (declare and create resources) {
  Use the resource to process the file;
}
```

Example

```
public class WriteDataWithAutoClose {
  public static void main(String[] args) throws Exception {
    java.io.File file = new java.io.File("scores.txt");
    if (file.exists()) {
      System.out.println("File already exists");
      System.exit(0);
    }

    try (
      // Create a file
      java.io.PrintWriter output = new java.io.PrintWriter(file);
    ) {
      // Write formatted output to the file
      output.print("John T Smith ");
      output.println(90);
      output.print("Eric K Jones ");
      output.println(85);
    }
  }
}
```

Example: Replacing Text

- Write a class named `ReplaceText` that replaces a string in a text file with a new string. The filename and strings are passed as command-line arguments as follows:

```
java ReplaceText sourceFile targetFile oldString newString
```

- For example, invoking

```
java ReplaceText FormatString.java t.txt StringBuilder StringBuffer
```

  replaces all the occurrences of StringBuilder by StringBuffer in FormatString.java and saves the new file in t.txt.

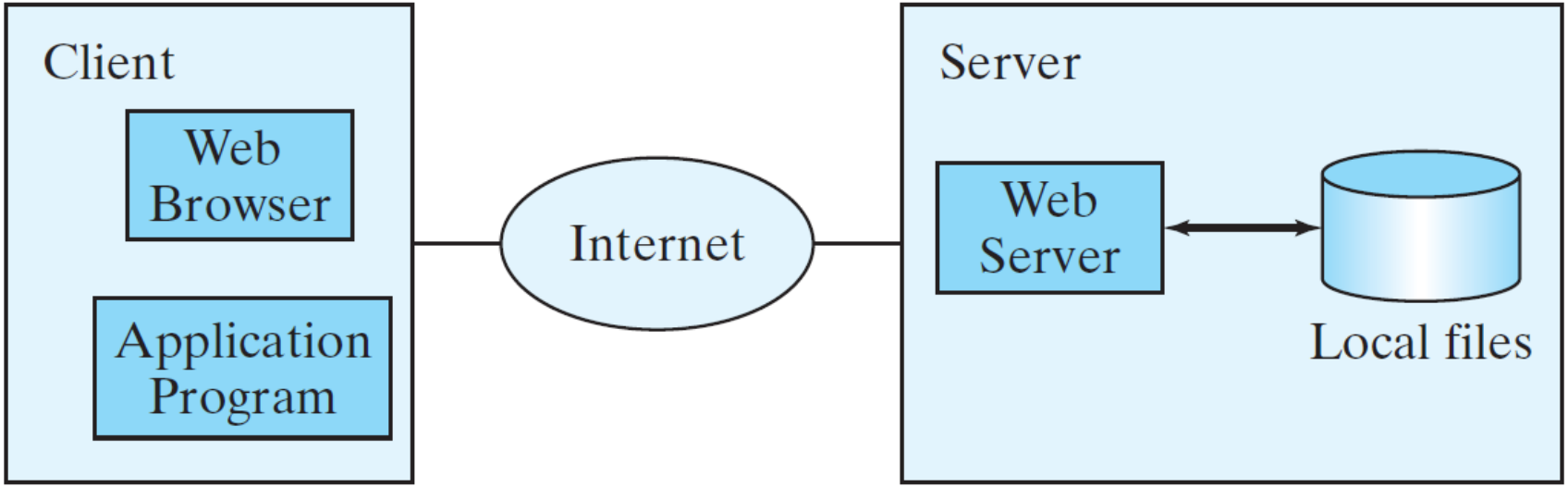
https://liveexample.pearsoncmg.com/html/ReplaceText.html

Source Code: Replacing Text

```
public class ReplaceText {
  public static void main(String[] args) throws Exception {
    // Check command line parameter usage
    if (args.length != 4) {
      System.out.println(
        "Usage: java ReplaceText sourceFile targetFile oldStr newStr");
      System.exit(1);
    }
    // Check if source file exists
    File sourceFile = new File(args[0]);
    if (!sourceFile.exists()) {
      System.out.println("Source file " + args[0] + " does not exist");
      System.exit(2);
    }
    // Check if target file exists
    File targetFile = new File(args[1]);
    if (targetFile.exists()) {
      System.out.println("Target file " + args[1] + " already exists");
      System.exit(3);
    }
    try (
      // Create input and output files
      Scanner input = new Scanner(sourceFile);
      PrintWriter output = new PrintWriter(targetFile);
    ) {
      while (input.hasNext()) {
        String s1 = input.nextLine();
        String s2 = s1.replaceAll(args[2], args[3]);
        output.println(s2);
      }
    }
  }
}
```

Reading Data from the Web

- Just like you can read data from a file on your computer, you can read data from a file on the Web.



Reading Data from the Web (2)

- Create a URL

```
URL url = new URL("www.google.com/index.html");
```

- After a URL object is created, you can use the `openStream()` method defined in the URL class to open an input stream and use this stream to create a Scanner object

```
Scanner input = new Scanner(url.openStream());
```

https://liveexample.pearsoncmg.com/html/ReadFileFromURL.html

Source Code

```java
public class ReadFileFromURL {
  public static void main(String[] args) {
    System.out.print("Enter a URL: ");
    String URLString = new Scanner(System.in).next();

    try {
      java.net.URL url = new java.net.URL(URLString);
      int count = 0;
      Scanner input = new Scanner(url.openStream());
      while (input.hasNext()) {
        String line = input.nextLine();
        count += line.length();
      }

      System.out.println("The file size is " + count + " characters");
    }
    catch (java.net.MalformedURLException ex) {
      System.out.println("Invalid URL");
    }
    catch (java.io.IOException ex) {
      System.out.println("IO Errors");
    }
  }
}
```

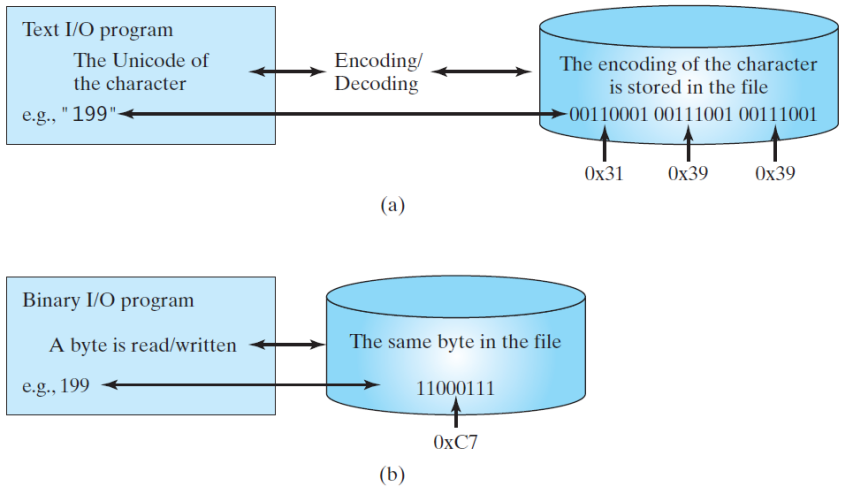
Binary IO

Text File vs. Binary File

- Data stored in a text file are represented in human-readable form. Data stored in a binary file are represented in binary form.
  - You cannot read binary files. Binary files are designed to be read by programs. For example, the Java source programs are stored in text files and can be read by a text editor, but the Java classes are stored in binary files and are read by the JVM.
  - The advantage of binary files is that they are more efficient to process than text files.
- Although it is not technically precise and correct, you can imagine that a text file consists of a sequence of characters and a binary file consists of a sequence of bits.
  - For example, the decimal integer 199 is stored as the sequence of three characters: '1', '9', '9' in a text file and the same integer is stored as a byte-type value C7 in a binary file, because decimal 199 equals to hex C7.

Text File vs. Binary File

Text I/O requires encoding and decoding. The JVM converts a Unicode to a file specific encoding when writing a character and coverts a file specific encoding to a Unicode when reading a character. Binary I/O does not require conversions. When you write a byte to a file, the original byte is copied into the file. When you read a byte from a file, the exact byte in the file is returned.



Text File vs. Binary File

aLeapp_out.txt

파일 편집 보기

```
Info: 189 plugins loaded.
Procesing started. Please wait. This may take a few minutes...

--------------------------------------------------------------------------------
ALEAPP v3.1.6: Android Logs, Events, and Protobuf Parser
Objective: Triage Android Full System Extractions.
By: Alexis Brignoni | @AlexisBrignoni | abrignoni.com
By: Yogesh Khatri   | @SwiftForensics | swiftforensics.com
Building files listing...
File listing complete - 3826 files
Artifact categories to parse: 189
File/Directory selected: ₩₩?₩C:₩Users₩madeh₩AppData₩Local₩Temp₩Autopsy₩avd_au

--------------------------------------------------------------------------------

Accounts_ce [accounts_ce] artifact started
Accounts_ce [accounts_ce] artifact completed
```

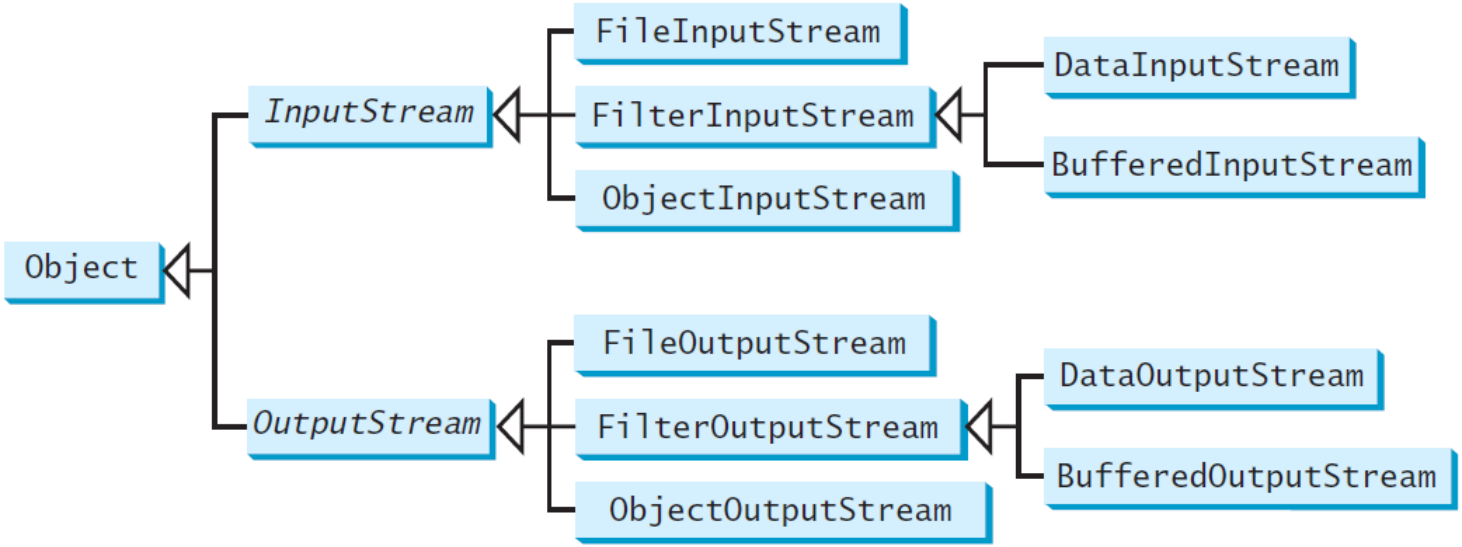
vs

abp.jar

파일 편집 보기

```
?□o?<쀤□?뽈I??욷?□_/?□?'Y?P?mp??構[?J??KL*뵲? 불L뇟9WVu.똔??휸]??4?
톗□n澾□K??N랠괼J?X]FT?츙BTKk*뜯X뒷□녥앍곳o□
墓復_?<?□□녦Y?D?뚶Xm陪L표?8쥒□??作?落唯{      ?□쇌□
9팵:z?c□?M쵤□y
Ko虧뒙1??봙□怠<?
x???=s:?뫥퍊?片?□?혉c&엄??□□@□?k<뙮^??휁뷶펟B?꿸/^ Y,□?R졢#/?qQ???늑        |썅[?훰□퐾???□??□퇓?鋤
*웗.'닸i □f(鼇引3'險O몂浸鷺?PK □?뒝@□ ? PK□□□ □□□ □Q□S            H   org/eclipse/jdt/internal/jarinjar
?□□O?+Je쳳?O?턞洌dS□?햪??%廣□□zR□y홎?y{□y□캤
□□홷줄?c?
?□□_*#닳?q-□A쒚∠?利?□쌇u뒎w"5□u쀰(2?□?첵□9뉲h□*쌅].U;??□?=C? 9??□34㎜J놰Is횾h랐
욟2□□
쮤1 룪□H%.&?죏?□?I흇>?虔b?%□9     □h2u瞭?? □□깹K&:□kW?※?헠?%?뺇]??? u?□쌶?□징=?꿨b?쐊퓝>O셆멷
org/eclipse/jdt/internal/jarinjarloader/JarRsrcLoader.class죥        X□?~□X쁢□sIB□M
쩩□?5Qc뀀□□콣?  ??잚?쓶?柏츔쉅{□{h듐??믎?굘wmko{j蜓쾶펠내걋老?毬3??앾?□□?G□□캚T*(□87e□F?i?zd멕?
K`EhOl팔□□쩽갤?>?)?□□쥸?65?K묶㈜?稽`뮌??i?싲d□쭬뷴□濟?-?H?S퓞??쭸□삡_$j?4$G쮯□?bO□8?□=뫔
w?a??넼TG:1□李쨉??,?똡$륿? 뛙q□?E??c@w變5?h
籃캝?붿□픁쓍宬Y잰□팾?힂□      ?쯁F3?컺딁쾩b???□mmN?Q?[V?艸?짧□?슏??部b□렜%톕V여彷Z헜?□곎뺃趨□'
l□픁□棉□?輓<?{~귩?!S째????f□?%5岬???ㅣ?쨄찰멽?p굼 덬 褙냰冊?봇{□□h?□堧□뮲$G쌺쓍g'?F꿸 ₩똵?N]?켎E
n긡?=□%쫼□匿;빳_D.d?△J        悚꼚?Q?땓□{??m쮹抑4?ΛEBE퓔d붰?$X□T
b(쀤8蚌G뿆톡꺝□₩稽0G쯂$?븕붮?袂?덫?뿛]'???□6h?뻰列e??+졺□쥊?뤼팏핚El?ow【|桑J좉1₩?오l
座x9^At□印W刀?:쿺p뿛?贓%□)?튪?□.z1:,e%L혂겦3?瀯m뱭□?□땋f늕?[??楷?□□K?뷀l?□i□즅?즆x??□o%B?(+)z?n
G?#坐턧kKG旿xk  ?□店-l?툸wRi쎋섅w灌륌
```

Binary I/O Classes



InputStream

The value returned is a byte as an int type.

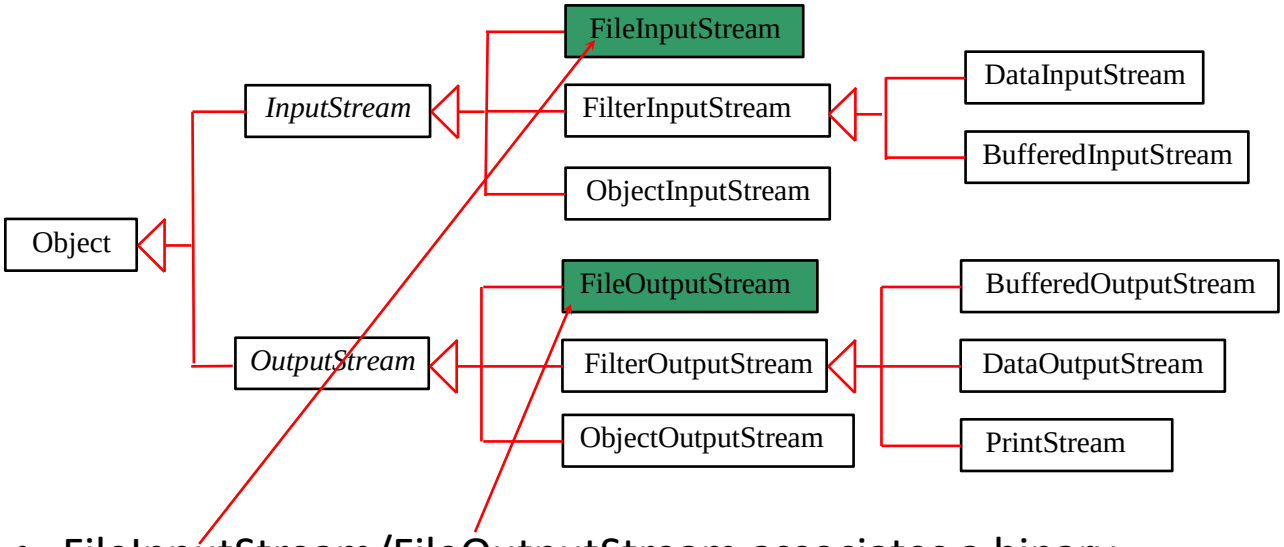
java.io.InputStream	
+read(): int	Reads the next byte of data from the input stream. The value byte is returned as an int value in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value –1 is returned.
+read(b: byte[]): int	Reads up to b.length bytes into array b from the input stream and returns the actual number of bytes read. Returns -1 at the end of the stream.
+read(b: byte[], off: int, len: int): int	Reads bytes from the input stream and stores into b[off], b[off+1], …, b[off+len-1]. The actual number of bytes read is returned. Returns -1 at the end of the stream.
+available(): int	Returns the number of bytes that can be read from the input stream.
+close(): void	Closes this input stream and releases any system resources associated with the stream.
+skip(n: long): long	Skips over and discards n bytes of data from this input stream. The actual number of bytes skipped is returned.
+markSupported(): boolean	Tests if this input stream supports the mark and reset methods.
+mark(readlimit: int): void	Marks the current position in this input stream.
+reset(): void	Repositions this stream to the position at the time the mark method was last called on this input stream.

OutputStream

The value is a byte as an int type.

java.io.OutputStream	
+write(int b): void	Writes the specified byte to this output stream. The parameter b is an int value. (byte)b is written to the output stream.
+write(b: byte[]): void	Writes all the bytes in array b to the output stream.
+write(b: byte[], off: int, len: int): void	Writes b[off], b[off+1], …, b[off+len-1] into the output stream.
+close(): void	Closes this output stream and releases any system resources associated with the stream.
+flush(): void	Flushes this output stream and forces any buffered output bytes to be written out.

FileInputStream/FileOutputStream



- FileInputStream/FileOutputStream associates a binary input/output stream with an external file.
- All the methods in FileInputStream/FileOuptputStream are inherited from its superclasses.

FileInputStream

To construct a FileInputStream, use the following constructors:

public FileInputStream(String filename)

public FileInputStream(File file)

A java.io.FileNotFoundException would occur if you attempt to create a FileInputStream with a nonexistent file.

FileOutputStream

To construct a FileOutputStream, use the following constructors:

public FileOutputStream(String filename)

public FileOutputStream(File file)

public FileOutputStream(String filename, boolean append)

public FileOutputStream(File file, boolean append)

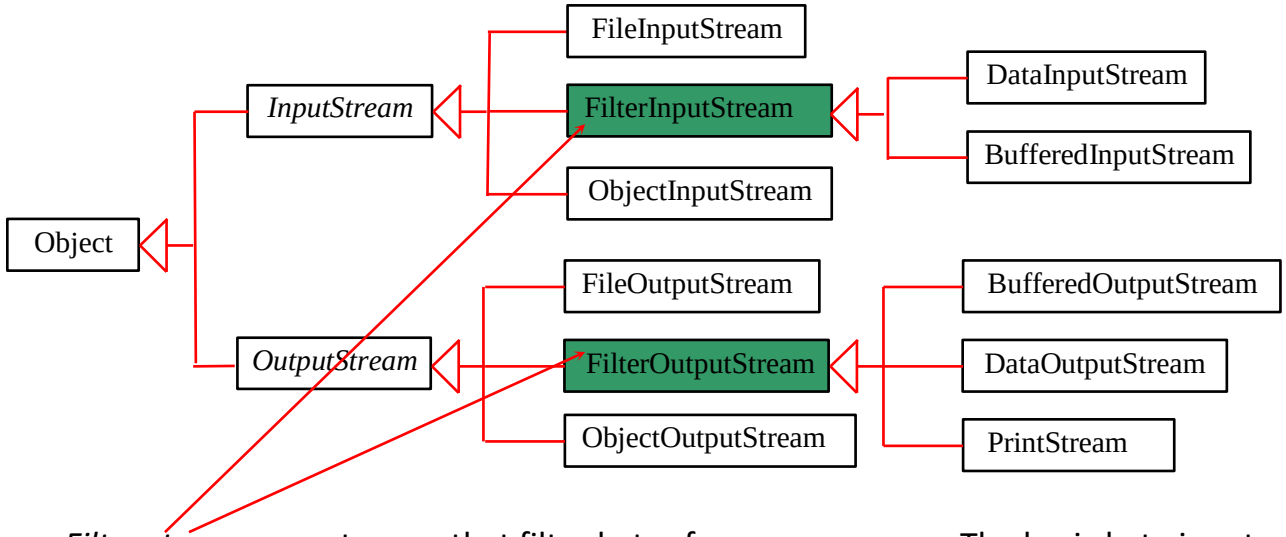
- If the file does not exist, a new file would be created. If the file already exists, the first two constructors would delete the current contents in the file.
- To retain the current content and append new data into the file, use the last two constructors by passing true to the append parameter.

```java
import java.io.*;

public class TestFileStream {
  public static void main(String[] args) throws IOException {
    try (
      // Create an output stream to the file
      FileOutputStream output = new FileOutputStream("temp.dat");
    ) {
      // Output values to the file
      for (int i = 1; i <= 10; i++)
        output.write(i);
    }

    try (
      // Create an input stream for the file
      FileInputStream input = new FileInputStream("temp.dat");
    ) {
      // Read values from the file
      int value;
      while ((value = input.read()) != -1)
        System.out.print(value + " ");
    }
  }
}
```

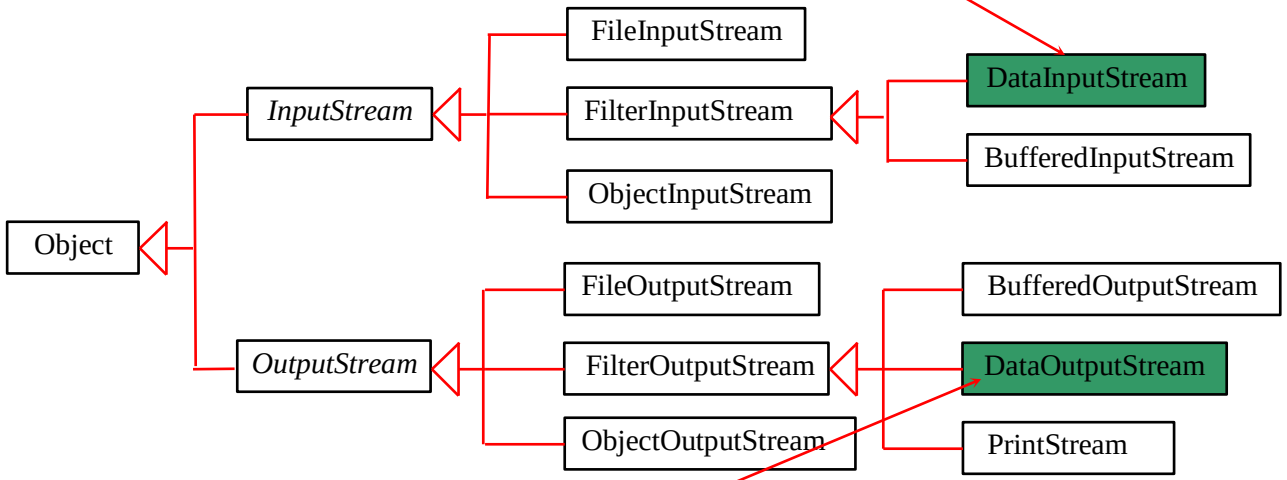
FilterInputStream/FilterOutputStream



- *Filter streams* are streams that filter bytes for some purpose. The basic byte input stream provides a read method that can only be used for reading bytes. If you want to read integers, doubles, or strings, you need a filter class to wrap the byte input stream.
- Using a filter class enables you to read integers, doubles, and strings instead of bytes and characters. FilterInputStream and FilterOutputStream are the base classes for filtering data. When you need to process primitive numeric types, use DatInputStream and DataOutputStream to filter bytes.

DataInputStream/DataOutputStream

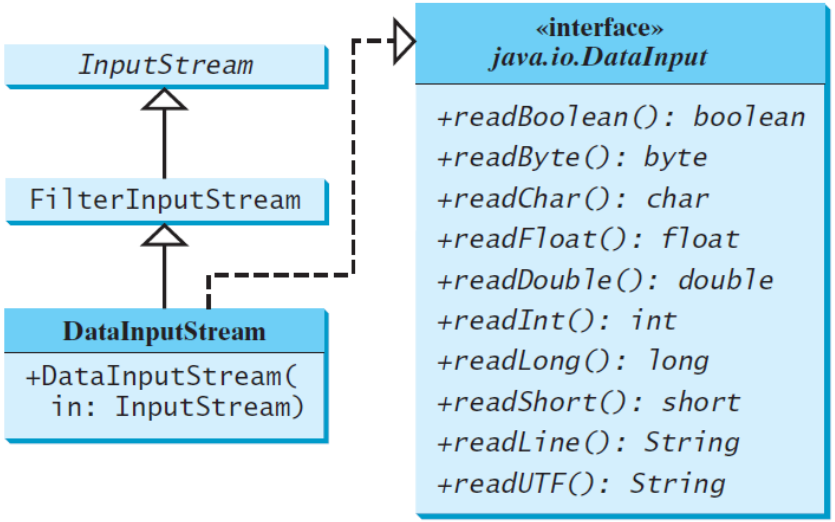
DataInputStream reads bytes from the stream and converts them into appropriate primitive type values or strings.



DataOutputStream converts primitive type values or strings into bytes and output the bytes to the stream.

DataInputStream

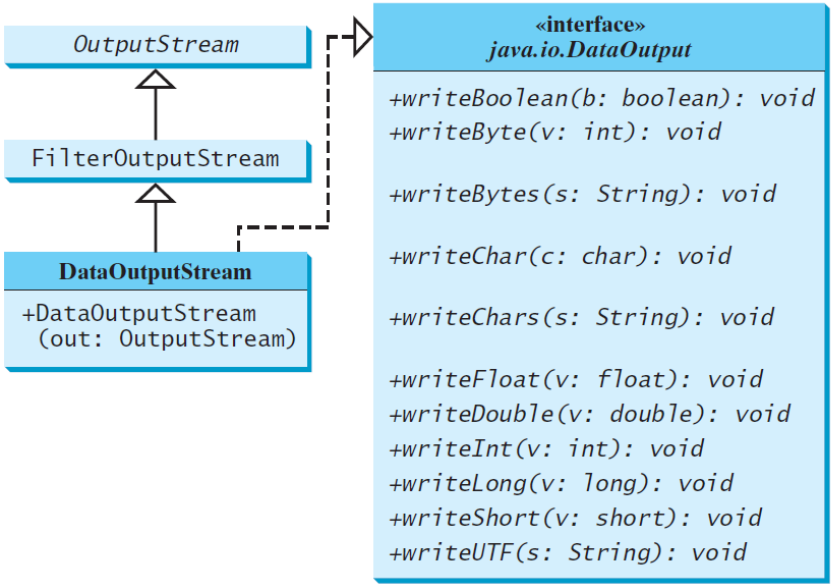
DataInputStream extends FilterInputStream and implements the DataInput interface.



Reads a `Boolean` from the input stream.	
Reads a `byte` from the input stream.	
Reads a character from the input stream.	
Reads a `float` from the input stream.	
Reads a `double` from the input stream.	
Reads an `int` from the input stream.	
Reads a `long` from the input stream.	
Reads a `short` from the input stream.	
Reads a `line` of characters from input.	
Reads a `string` in UTF format.	

DataOutputStream

DataOutputStream extends FilterOutputStream and implements the DataOutput interface.



Writes a Boolean to the output stream.

Writes the eight low-order bits of the argument `v` to the output stream.

Writes the lower byte of the characters in a string to the output stream.

Writes a character (composed of 2 bytes) to the output stream.

Writes every character in the string `s` to the output stream, in order, 2 bytes per character.

Writes a `float` value to the output stream.

Writes a `double` value to the output stream.

Writes an `int` value to the output stream.

Writes a `long` value to the output stream.

Writes a `short` value to the output stream.

Writes `s` string in UTF format.

Characters and Strings in Binary I/O

A Unicode consists of two bytes (16 bits).

- The writeChar(char c) method writes the Unicode of character c to the output.
- The writeChars(String s) method writes the Unicode for each character in the string s to the output.

Why UTF-8? What is UTF-8?

UTF-8 is a coding scheme that allows systems to operate with both ASCII and Unicode efficiently. Most operating systems use ASCII. **Java uses Unicode**.

- The ASCII character set is a subset of the Unicode character set. Since most applications need only the ASCII character set, it is a waste to represent an 8-bit ASCII character as a 16-bit Unicode character.
- The UTF-8 is an alternative scheme that stores a character using 1, 2, or 3 bytes. ASCII values (less than 0x7F) are coded in one byte. Unicode values less than 0x7FF are coded in two bytes. Other Unicode values are coded in three bytes.

Using DataInputStream/DataOutputStream

Data streams are used as wrappers on existing input and output streams to filter data in the original stream. They are created using the following constructors:

public DataInputStream(InputStream instream)

public DataOutputStream(OutputStream outstream)

The statements given below create data streams. The first statement creates an input stream for file **in.dat**; the second statement creates an output stream for file **out.dat**.

DataInputStream infile =

new DataInputStream(new FileInputStream("in.dat"));

DataOutputStream outfile =

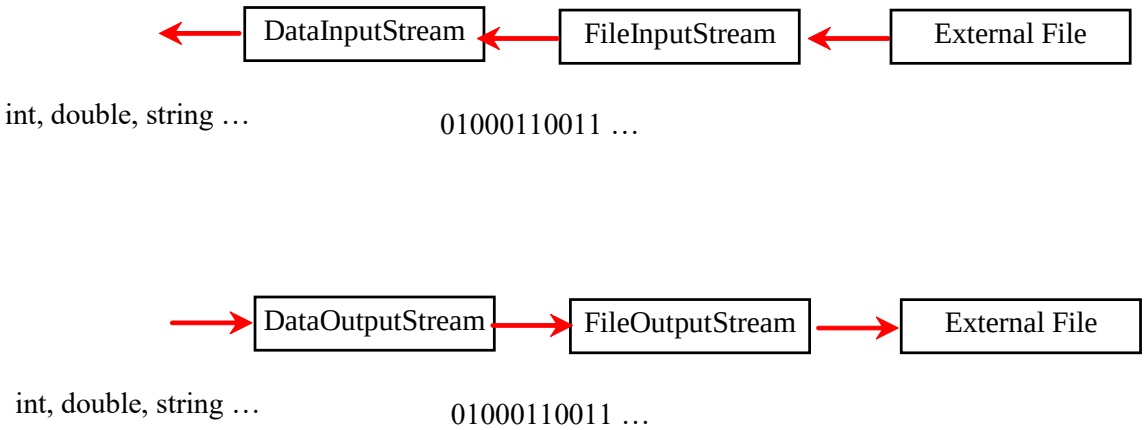
new DataOutputStream(new FileOutputStream("out.dat"));

```
import java.io.*;

public class TestDataStream {
  public static void main(String[] args) throws IOException {
    try ( // Create an output stream for file temp.dat
      DataOutputStream output =
        new DataOutputStream(new FileOutputStream("temp.dat"));
    ) {
      // Write student test scores to the file
      output.writeUTF("Liam");
      output.writeDouble(85.5);
      output.writeUTF("Susan");
      output.writeDouble(185.5);
      output.writeUTF("Chandra");
      output.writeDouble(105.25);
    }

    try ( // Create an input stream for file temp.dat
      DataInputStream input =
        new DataInputStream(new FileInputStream("temp.dat"));
    ) {
      // Read student test scores from the file
      System.out.println(input.readUTF() + " " + input.readDouble());
      System.out.println(input.readUTF() + " " + input.readDouble());
      System.out.println(input.readUTF() + " " + input.readDouble());
    }
  }
}
```

Concept of pipe line



- **DataInputStream** filters data from an input stream into appropriate primitive-type values or strings.
- **DataOutputStream** converts primitive-type values or strings into bytes and outputs the bytes to an output stream.
- You can view DataInputStream/FileInputStream and DataOutputStream/FileOutputStream working in a pipe line as shown above.

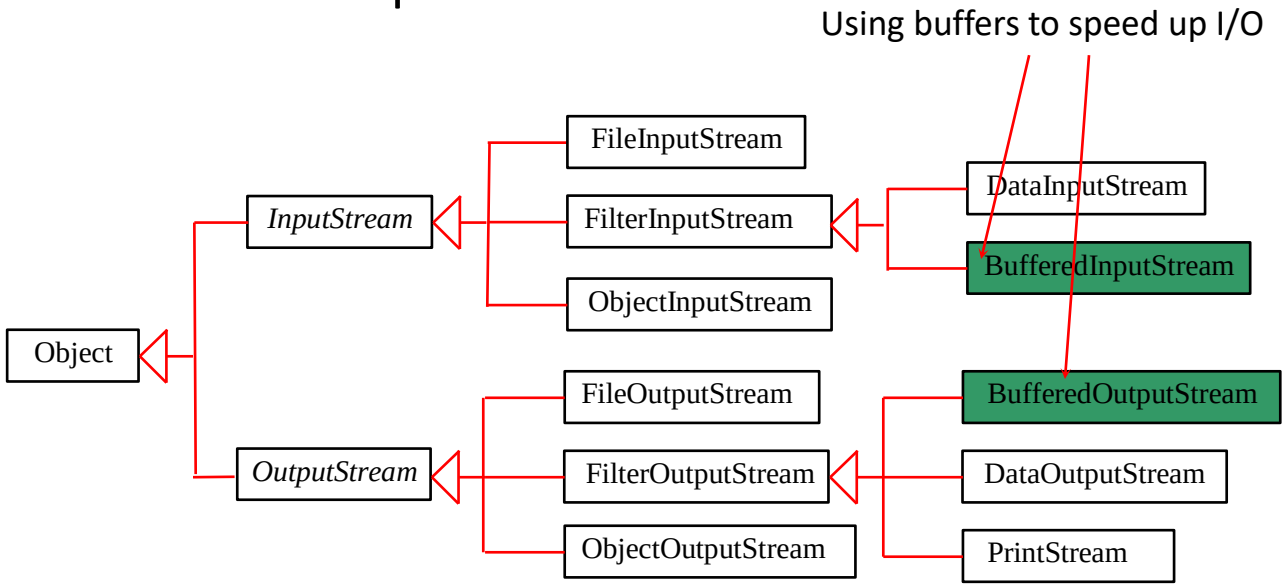
Order and Format

CAUTION: You have to read the data in the same order and same format in which they are stored. For example, since names are written in UTF-8 using writeUTF, you must read names using readUTF.

Checking End of File

TIP: If you keep reading data at the end of a stream, an EOFException would occur. So how do you check the end of a file? You can use input.available() to check it. input.available() == 0 indicates that it is the end of a file.

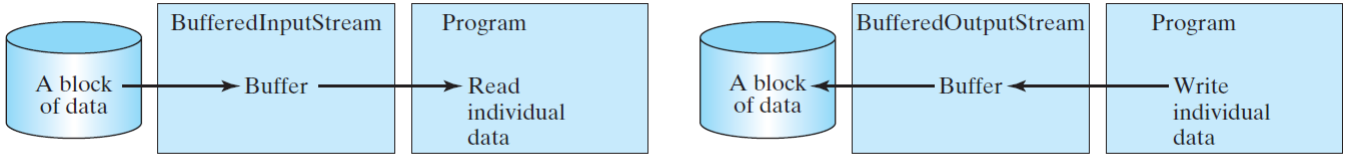
BufferedInputStream/ BufferedOutputStream



BufferedInputStream/BufferedOutputStream does not contain new methods. All the methods BufferedInputStream/BufferedOutputStream are inherited from the InputStream/OutputStream classes.

Constructing BufferedInputStream/BufferedOutputStream

- **BufferedInputStream**/**BufferedOutputStream** can be used to speed up input and output by reducing the number of disk reads and writes.
- Using **BufferedInputStream**, the whole block of data on the disk is read into the buffer in the memory once. The individual data are then delivered to your program from the buffer
- Using **BufferedOutputStream**, the individual data are first written to the buffer in the memory. When the buffer is full, all data in the buffer are written to the disk once



(a) (b)

```
// Create a BufferedInputStream
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int bufferSize)

// Create a BufferedOutputStream
public BufferedOutputStream(OutputStream out)
public BufferedOutputStream(OutputStreamr out, int bufferSize)
```

.:END:.