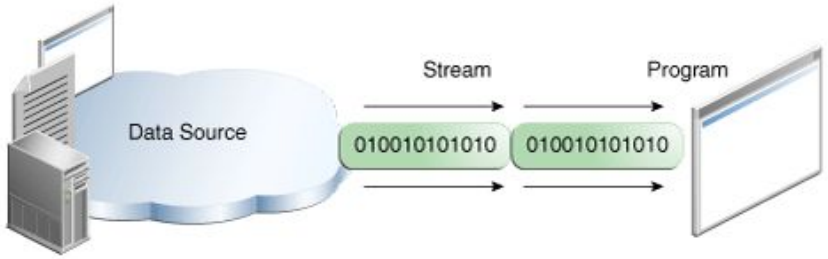
CSE-2102
Object Oriented Programming

Dr. Muhammad Ibrahim
Assistant Professor
Dept. of Computer Science and Engineering
University of Dhaka

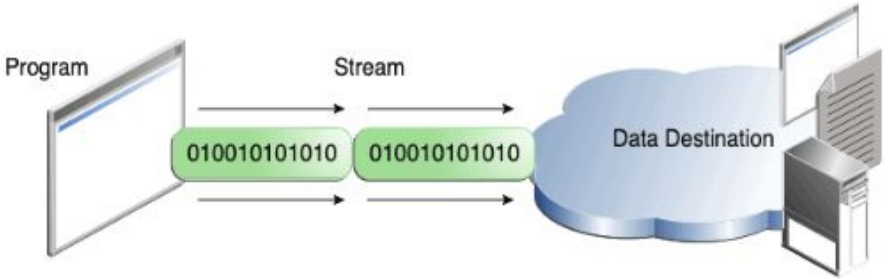
I/O

Input and Output (I/O)

- So far we haven't used console I/O much because real-life Java programs don't use them much as they mostly run graphical user interfaces (GUI)
  - E.g.: Swing, AWT, JavaFX, Web applications
- Three types of I/O
  - Console
  - File
  - Network



Reading information into a program.



Writing information from a program.

Image: https://docs.oracle.com/javase/tutorial/essential/io/streams.html

Stream

- Java performs I/O operations through streams.
- A stream is an abstraction that either consumes information or produces it.
- A stream is linked to a physical device by Java's I/O system.
- Importantly, all streams behave in the same manner, although the physical devices to which they are connected may differ.
  - Thus the same I/O classes and methods can be used for different types of devices thereby hiding physical details from programmers.
- `java.io` package contains classes that facilitate streams.
- There are two types of streams:
  - Byte stream
  - Character stream
    - Recall that a byte and a character in Java consists of 1 and 2 bytes respectively.

Byte Stream and Character Stream

- Byte streams are used to read and write binary data.
- Character streams are used to read and write Unicode.
  - In some cases character streams are more efficient than byte streams.
- At a lower level of Java, all streams are byte oriented, character streams just add a layer on top of that.

Byte Stream Hierarchy

`InputStream`
(Abstract class)

An important (abstract) method: `read()`

Some important subclasses:
`BufferedInputStream`
`DataInputStream`
`FileInputStream`

`OutputStream`
(Abstract class)

An important (abstract) method: `write()`

Some important subclasses:
`BufferedOutputStream`
`DataOutputStream`
`FileOutputStream`
`PrintStream`

All of these classes are in `java.io` package

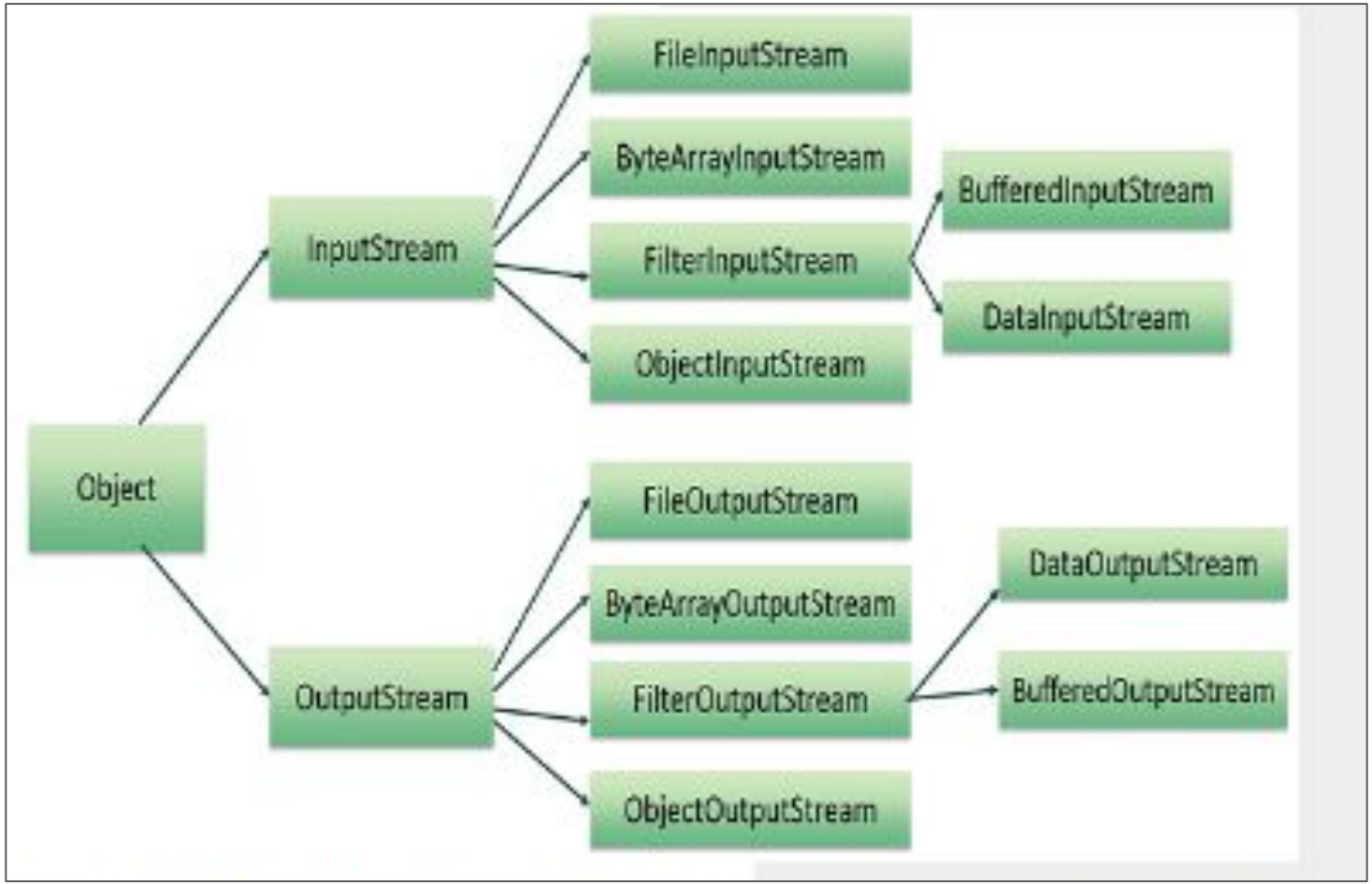


Image: https://www.tutorialspoint.com/java/java_files_io.htm

Character Stream Hierarchy

`Reader`
(Abstract class)

An important (abstract) method: `read()`

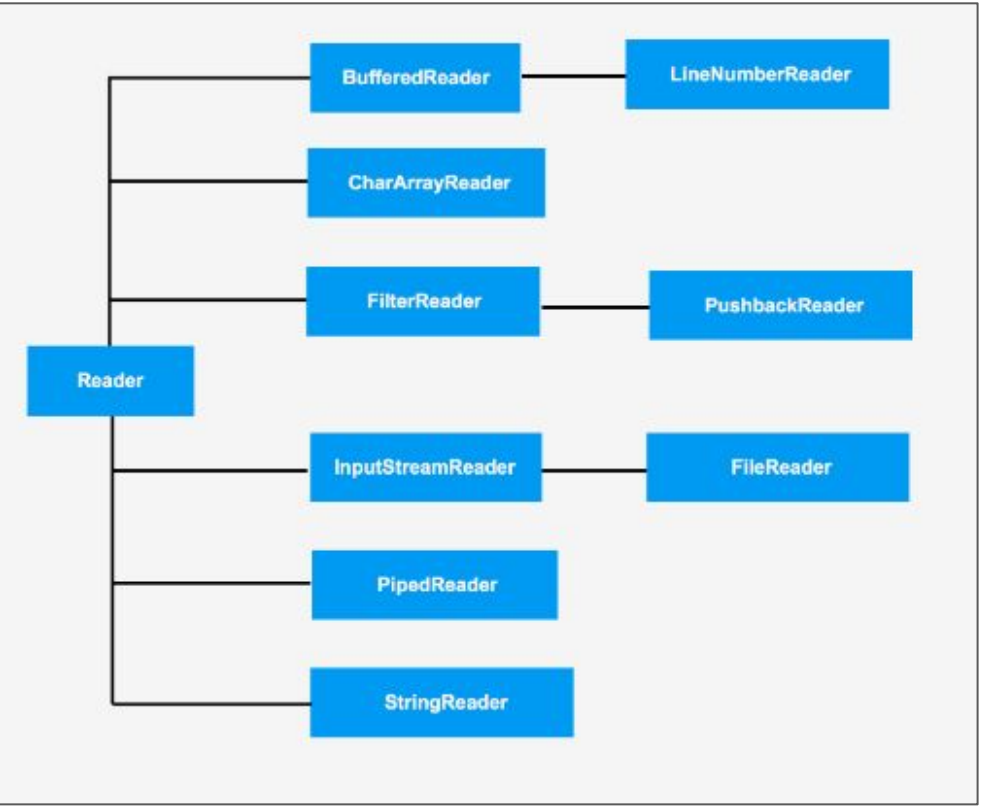
Some important subclasses:
`BufferedReader`
`FileReader`
`InputStreamReader`

`Writer`
(Abstract class)

An important (abstract) method: `write()`

Some important subclasses:
`BufferedWriter`
`FileWriter`
`OutputStreamWriter`
`PrintWriter`

All of these classes are in `java.io` package



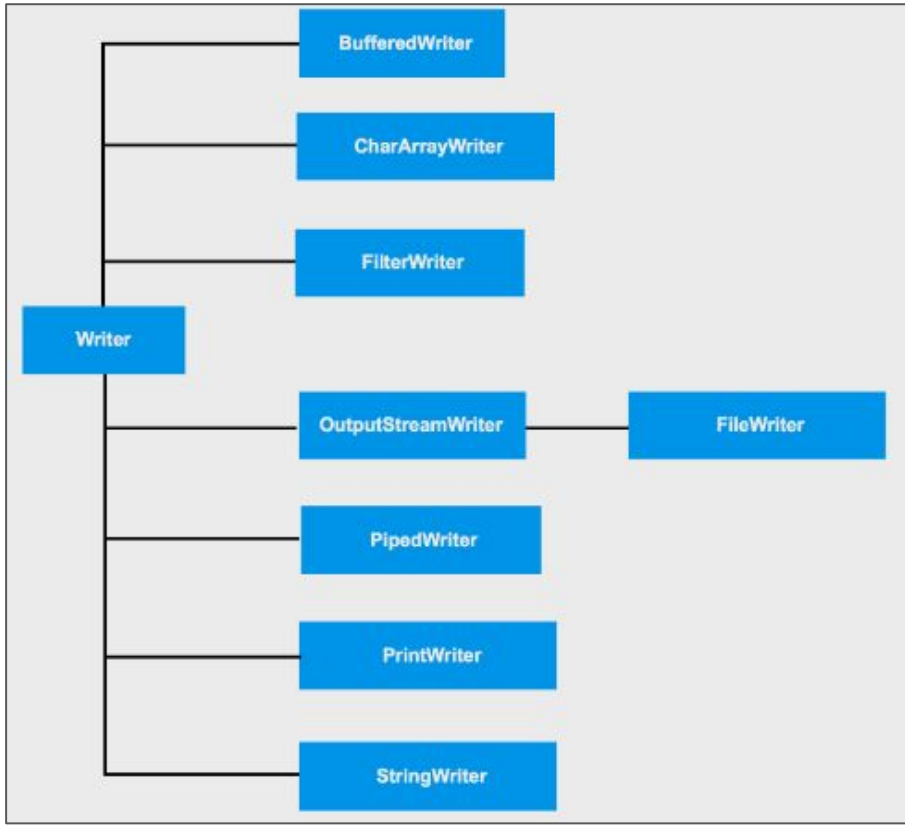


Image: https://javagyansite.com/2020/03/22/character-stream-hierarchy-in-java/

The Predefined Streams

- Recall that all Java programs by default import `java.lang` package.
- This package contains a class called `System` that encapsulates several runtime environmental details.
- `System` class contains three predefined stream variables: `in`, `out` and `err`.
  - These are public, static and final (`InputStream in; PrintStream out; PrintStream err;`)
    - So can be used without `System`'s object.
- `System.out` is the standard output stream
  - By default, this is console output
  - This is an object of type `PrintStream` (byte stream), and `System.err` is also such an object.
- `System.in` is the standard input stream
  - By default, this is console input
  - This is an object of type `InputStream` (byte stream)

Reading Console Input: Characters

- `BufferedReader br = new BufferedReader(new InputStreamReader(System.in));`
  - Character stream
  - `BufferedReader` wraps a `Reader` object (or one of its subclasses)

    ```
    BufferedReader (Reader r)
    ```
  - `InputStreamReader` links `System.in` stream.

    ```
    InputStreamReader (InputStream in)
    ```
  - `System.in` is line buffered, i.e., no input is passed to your program until your press ENTER
- `int read() throws IOException`
  - Reads characters and returns equivalent integer (Unicode or ASCII)

Reading Console Input: Characters

```
   void characters () throws IOException {
       BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
       int i;
       do{
         i = in.read();
         System.out.println((char)i);
       }while (i != 'q');
   }
```

Reading Console Input: Strings

- `BufferedReader br = new BufferedReader(new InputStreamReader(System.in));`
- `String readLine() throws IOException`

Reading Console Input: Strings

```
void strings() throws IOException {
     BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
     String str;
     System.out.println("Enter lines of text.");
     System.out.println("Enter 'stop' to quit.");
     do {
         str = br.readLine();
         System.out.println(str);
     } while(!str.equals("stop"));
 }
```

Writing Console Output

1. `print()` and `println()` methods
   a. Defined in `PrintStream` class
      i. So they are referenced by `System.out`
   b. Byte stream
2. `void write(int num_of_bytes)`
   a. Although the type of parameter is int, only the low order 8 bits are written (in console)
   b. Byte stream
3. PrintWriter class: `PrintWriter(OutputStream os, boolean flushing_on)`
   a. Keeping flushing_on true makes automatic flushing of output stream every time a println() method (or the like) is called.
   b. For real world programs, this mechanism is suggested
   c. Character stream, so internationalizing your program
   d. Supports print() and println() methods

Topics

- Writing to Console
- File I/O
  - Byte stream
  - Character stream
- Strings
  - Built-in methods
  - StringBuffer and StringBuilder classes

Writing Console Output

1. `print()` and `println()` methods
   a. Defined in `PrintStream` class
      i. So they are referenced by System.out
   b. Byte stream
2. `void write(int num_of_bytes)`
   a. Although the type of parameter is int, only the low order 8 bits are written (in console)
   b. Byte stream
3. `PrintWriter` class: `PrintWriter(OutputStream os, boolean flushing_on)`
   a. Keeping flushing_on true makes automatic flushing of output stream every time a `println()` method (or the like) is called.
   b. For real world programs, this mechanism is suggested
   c. Character stream, so internationalizing your program
   d. Supports `print()` and `println()` methods

Files

- The `File` class (see example)

Reading and Writing Files

Byte stream: FileInputStream and FileOutputStream classes

Character stream: FileReader and FileWriter classes

Closing a file: `void close () throws IOException`

File I/O with Byte Stream

- FileInputStream(String fileName) throws FileNotFoundException
- FileOutputStream(String fileName) throws FileNotFoundException

Reading from file: int read( ) throws IOException

- When it is called, it reads a single byte from the file and returns the byte as an integer value.
- read( ) returns –1 when the end of the file is encountered.

Let’s see an example …

Reading File Using Byte Stream

```
int i=0;

FileInputStream fin=null;

fin = new FileInputStream(filename);
do {
   i = fin.read();
   if(i != -1) System.out.print((char) i);
} while(i != -1);

fin.close();
```

You can play around with the optimal setting of try-catch block (which is omitted here)

Digression: Using finally Block

```
try {
    do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
    } while(i != -1);
} catch(IOException e) {
    System.out.println("Error Reading File");
} finally {
// Close file on the way out of the try block.
    try {
        fin.close();
    } catch(IOException e) {
        System.out.println("Error Closing File");
    }
}
```

Writing to a File Using Byte Stream

```
FileOutputStream fout = new
FileOutputStream("C:\\a.txt");

do {

   i = fin.read();

   if(i != -1) fout.write(i);

} while(i != -1);
```

File I/O with Character Stream

- Input
  - Classes: BufferedReader, FileReader, File
  - Methods: readLine(), close()
- Output
  - Classes: BufferedWriter, FileWriter, File
  - Methods: write(), close()
- Let’s see an example …

Strings

- Recall that strings are implemented as objects of type String
  - String str = new String();
- Java's strings are immutable
  - Each time we need modification on an existing string, a new String object is created
  - Why you wonder? Because fixed, immutable strings can be implemented more efficiently than mutable ones
  - However, a variable that refers to a String object can refer to different String objects.
- If we need to change strings, there are two other classes for this: StringBuffer and StringBuilder.
  - Their strings can be changed
- String, StringBuffer and StringBuilder are in java.lang package
  - All three are final classes, so cannot be inherited
  - All three implement an interface named CharSequence

String Constructors

- String s = new String();
  - Empty string
- String(char chars[ ])
  - char chars[] = { 'a', 'b', 'c' }; String s = new String(chars);
- String(char chars[ ], int startIndex, int numChars)
  - char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' }; String s = new String(chars, 2, 3);
- String(String strObj)
  - char c[] = {'a', 'b', 'c'', 'd'}; String s1 = new String(); String s2 = new String(s1);
- String(byte chrs[ ])
  - Because an ASCII code takes 1 byte
- String(byte chrs[ ], int startIndex, int numChars)

Example: String Constructors for Byte

```
class String_constructors {
   public static void main(String args[]) {
      byte ascii[] = {65, 66, 67, 68, 69, 70 };
      String s1 = new String(ascii);
      System.out.println(s1);
      String s2 = new String(ascii, 2, 3);
      System.out.println(s2);
   }
}
```

This program generates the following output:

```
ABCDEF
CDE
```

More String Constructors

- `String(StringBuffer str_buf_obj)`
- `String(StringBuilder str_build_obj)`
- `String(int codePoints[ ], int startIndex, int numChars)`

String Operations

- String length: int length()
  - char chars[] = { 'a', 'b', 'c' }; String s = new String(chars);
    System.out.println(s.length());
- String literals: `String s2 = "abc";`
  - A string object is created in the background for “abc”
  - “abc”.length is also valid
- Concatenation:
  - `String str = “ghi”;  String str2 = “abc” + “def” + str;`
  - With other data types: `String s = "four: " + 2 + 2;`
    `System.out.println(s);`
    - Output: four: 22 (NOT: “four: 4”)
    - Why? Because “four” + 2 results in a string “four: 2”, then + 2 is executed on it.
    - `String s = "four: " + (2 + 2);`, however, produces “four: 4” as output

String Operations (Contd.)

- Description of an object: override the toString() method
  - String toString() { return “Description of this string”; }
- Extracting a character: char charAt(int position)
  - char ch;   ch = "abc".charAt(1);
- Extracting multiple characters:
  - void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
- Convert to character array: char[ ] toCharArray()

String Operations (Contd.)

- String comparison:
  - boolean equals(Object str)
  - boolean equalsIgnoreCase(String str)
- Pattern matching:
  - boolean startsWith(String str): "Foobar".startsWith("Foo")
    - boolean startsWith(String str, int startIndex): "Foobar".startsWith("bar", 3)
  - boolean endsWith(String str): "Foobar".endsWith("bar")
- equals( ) Versus ==
  - == checks if two variables are referring to the same object
  - Example is in the next slide
- Comparing to know less or greater:
  - int compareTo(String str)
  - int compareToIgnoreCase(String str)

Example of equals() Versus ==

```
class EqualsNotEqualTo {
    public static void main(String args[]) {
        String s1 = "Hello";
        String s2 = new String(s1);
        System.out.println(s1 + " equals " + s2 + " -> " +
        s1.equals(s2));
        System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
    }
}
```

Output:

```
Hello equals Hello -> true
Hello == Hello -> false
```

More Built-in Methods

- A lot of other built-in methods; for your self-study
  - valueOf
  - trim
  - replace
  - concat
  - substring
  - indexOf and lastIndexOf

StringBuffer Class

- Supports modification of strings
- To insert characters inside a string, space is automatically increased
- Defines four constructors:
  - StringBuffer( ): reserves room for 16 characters
  - StringBuffer(int size): reserves room for size amount of characters
  - StringBuffer(String str): initializes with str and reserves 16 more spaces
  - StringBuffer(CharSequence chars):
- A lot of built-in methods; for your self-study

Example of StringBuffer

```
class StringBufferDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        System.out.println("buffer = " + sb);
        System.out.println("length = " + sb.length());
        System.out.println("capacity = " + sb.capacity());
    }
}
```

Output:

```
buffer = Hello
length = 5
capacity = 21
```

More Built-in Methods

- Modifying a string
  - void setCharAt(int where, char ch)
  - StringBuffer insert(int index, String str)
  - StringBuffer insert(int index, char ch)
  - StringBuffer insert(int index, Object obj)
  - delete() deleteAt()
  - replace()
  - reverse()
  - append()
- Difference between the methods of StringBuffer class that modify a string and that of String class: the former modify the actual string whereas the latter keep the string unchanged but creates a new string to modify.

StringBuilder Class

- StringBuilder is quite similar to StringBuffer
  - Sometimes StringBuilder is slightly faster
- Up to the programmer’s choice as to which one to use