Java IO and file Handling

Prof. Anita Agrawal

BITS Pilani- K.K.Birla Goa Campus

Email: aagrawal@goa.bits-pilani.ac.in

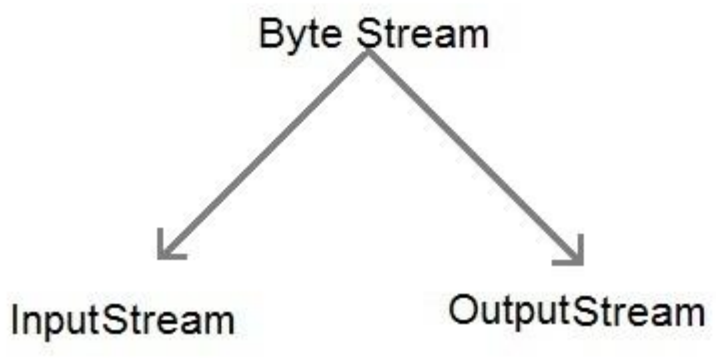
Introduction

- **Java I/O** (Input and Output) is used *to process the input* and *produce the output*.
- Java uses the concept of a **stream** to make I/O operations fast.
- A stream is linked to a physical layer by java I/O system to make input and output operations.
- Java encapsulates Stream under **java.io** package
- The java.io package contains all the classes required for input and output operations.
- File handling in Java is performed by Java I/O

Types of Streams

- Two types of streams:
  - Byte Stream
  - Character Stream
- Byte stream:

  Defined by using two abstract classes at the top of hierarchy, they are InputStream and OutputStream.



Byte Oriented streams and character-oriented streams

- Byte-oriented streams.
  - Intended for general-purpose input and output.
  - Data may be primitive data types or raw bytes.
- Character-oriented streams.
  - Intended for character data.
  - Data is transformed from/to 16 bit Java char used inside programs to the UTF format used externally.

Byte Stream

- Create binary files.
- A binary file essentially contains the memory image of the data. That is, it stores bits as they are in memory.
- Binary files are faster to read and write because no translation need take place.

InputStream

- Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.
- It is an abstract class.
- It is the superclass of all classes representing an input stream of bytes.

Useful methods of input stream

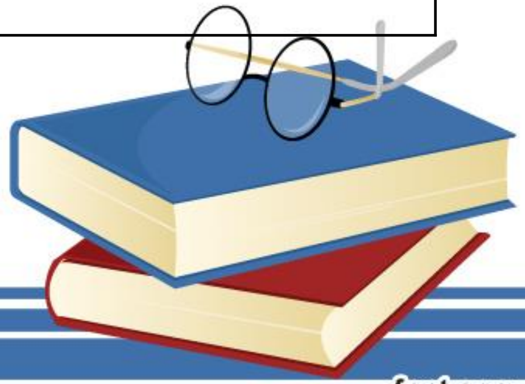
Method	Description
1) public abstract int read()	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()	is used to close the current input stream.

OutputStream class

- Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.
- It is an abstract class.
- It is the superclass of all classes representing an output stream of bytes.
- An output stream accepts output bytes and sends them to some sink.

Important methods of output stream

Method	Description
1) public void write(int)	is used to write a byte to the current output stream.
2) public void write(int[ ])	is used to write an array of byte to the current output stream.
3) public void flush()	flushes the current output stream.
4) public void close()	is used to close the current output stream.



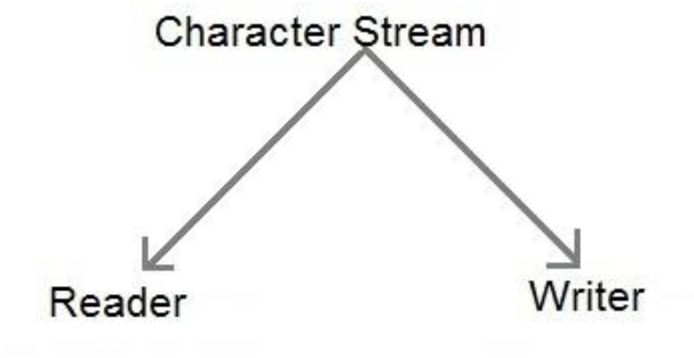
Important Byte Stream classes

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain print() and println() method

- Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream.**

Character Stream

- Character stream is also defined by using two abstract classes at the top of hierarchy, they are Reader and Writer.

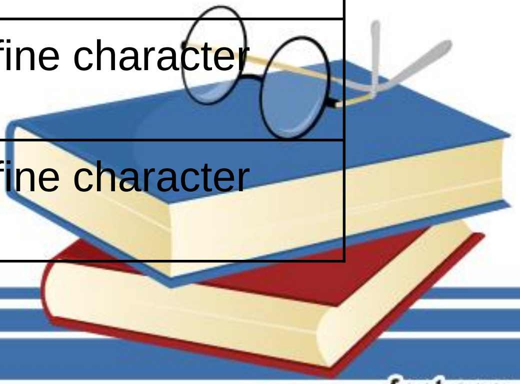


Character Streams

- **Character** streams are used to perform input and output for 16-bit unicode
- Character streams create text files.
- These are files designed to be read with a text editor.
- Java automatically converts its internal unicode characters to the local machine representation (ASCII in our case).
- These two abstract classes have several concrete classes that handle unicode character.

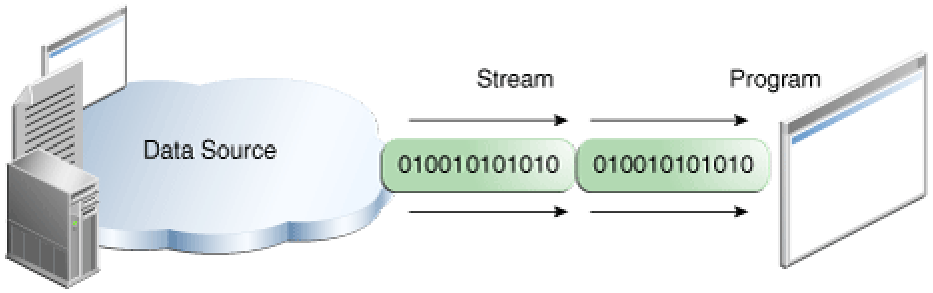
Some important character stream classes

Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain print() and println() method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output

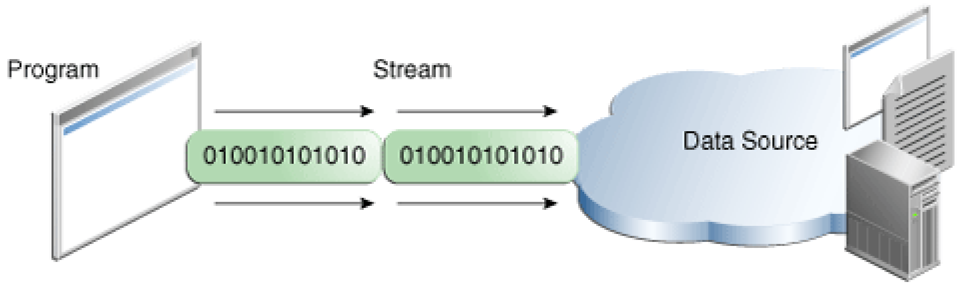


- Though there are many classes related to character streams but the most frequently used classes are, **Reader** and **Writer.**

Reading information into a program



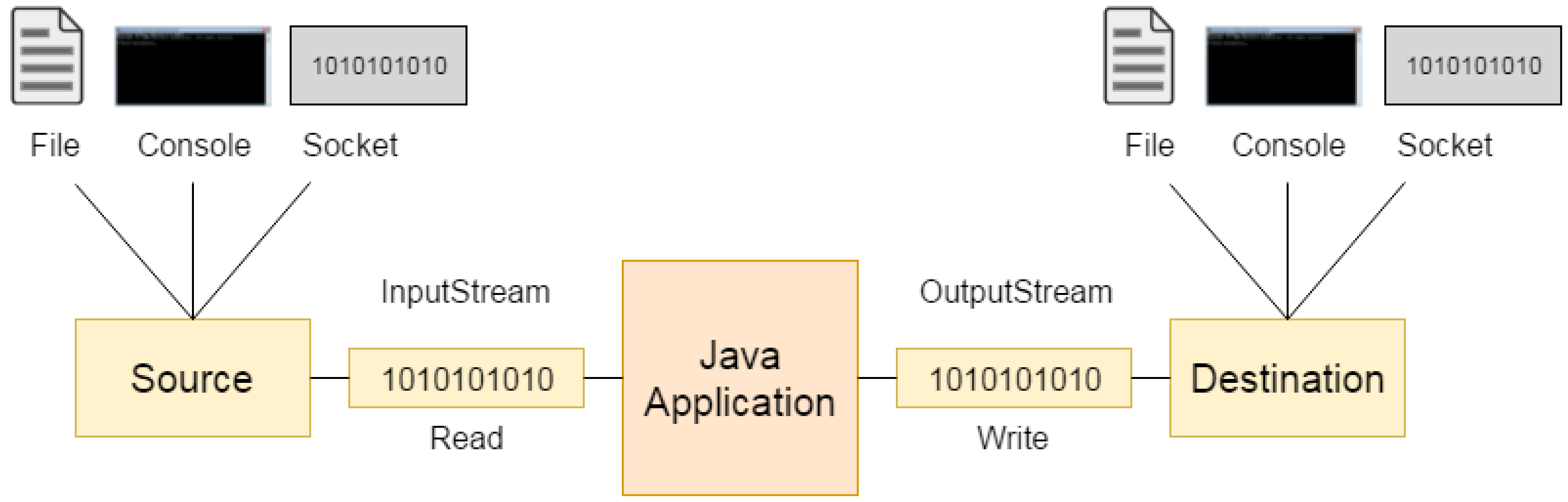
Writing information from a program



No matter how they work internally, all streams present the same simple model to programs that use them

The data source and data destination pictured above can be anything that holds, generates, or consumes data. This includes disk files, another program, a peripheral device, a network socket, or an array.

Working of Java input and output stream



Reading and Writing a file

- To open a file, you simply create an object of one of these classes, viz:
- FileInputStream
- FileOutputStream
- Specifying the file name as an argument to the constructor
  - FileInputStream(String fileName)
  - FileOutputStream(String fileName)

- For input stream:
  - If the file **does not exist**, then **FileNotFoundException** is thrown
- For output stream:
  - If the file cannot be **opened** or **created**, then **FileNotFoundException** is thrown
- **FileNotFoundException** is a subclass of **IOException**
  - So catching an **IOException** exception will do the job
- When an output file is opened, any pre-existing file by the same name is destroyed

Reading from a file

- To read from a file
  - Use **read( )** method defined within **FileInputStream**
    - Int read( )
- Each time **read()** is called
  - A single byte is read from the file and returns the byte as an integer value
  - **read( )** returns –1 when the end of the file is encountered
  - **read( )**can throw an **IOException**

Read1.java, Read2.java

- **NOTE:**
  - Resource declared in the **try** statement is implicitly **final**
  - You can’t assign to the resource after it has been created
  - We can manage more than one resource within a single **try** statement with a semicolon

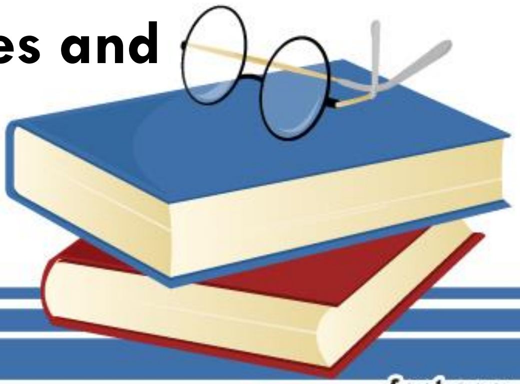
Read8.java

Closing a file

- When you are done with a file, you must close the file
  - Closing a file releases the system resources allocated to the file
  - Allowing them to be used by another file
- The traditional approach to close the file is by calling the **close( )** method
  - void close( )
  - The second is to use the **try**-with-resources statement
  - Automatically closes a file when it is no longer needed
  - No explicit call to **close( )** is executed
  - Available for JDK 7 or later

Automatically closing a file revisited…..

- Explicit calls to **close( )** are made once a file is no longer needed
- JDK 7 added a new feature that offers another way
  - This feature, sometimes referred to as *automatic resource management*(**ARM**)
  - Simply, an expanded version of the **try** statement
- Try-with-resources
  - try (resource-specification) {// use the resource}
- *resource-specification* is a statement that **declares and initializes** a resource, such as a file stream



Automatically closing a file

try(FileInputStream fr = new FileInputStream(args[0])) {

- **try** declares a **FileInputStream** called **fr**
- **fr** is then assigned a reference to the file opened by its constructor
- The variable **fr** is local to the **try** block being created when the **try** is entered
- When the **try** is left, the stream associated with **fr** is **automatically** closed