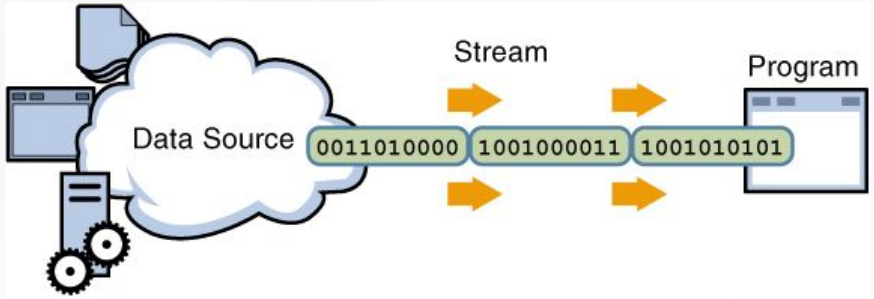
IO Programming



Outline

- ✔ File class
- ✔ Stream
- ✔ Byte Stream
- ✔ Character Stream

File class

- Java **`File`** class represents the files and directory pathnames in an **abstract manner**. This class is used for **creation of files** and **directories**, **file searching**, **file deletion** etc.
- The File object represents the actual file/directory on the disk. Below given is the list of constructors to create a File object.
- Constructors :

Sr.	Constructor
1	File(String pathname) Creates a new File instance by converting the given pathname string into an abstract pathname.
2	File(String parent, String child) Creates a new File instance from a parent pathname string and a child pathname string.
3	File(URI uri) Creates a new File instance by converting the given file: URI into an abstract pathname.

Methods of File Class

Sr.	Method
1	public boolean isAbsolute() Tests whether this abstract pathname is absolute. Returns true if this abstract pathname is absolute, false otherwise
2	public String getAbsolutePath() Returns the absolute pathname string of this abstract pathname.
3	public boolean canRead() Tests whether the application can read the file denoted by this abstract pathname. Returns true if and only if the file specified by this abstract pathname exists and can be read by the application; false otherwise.
4	public boolean canWrite() Tests whether the application can modify to the file denoted by this abstract pathname. Returns true if and only if the file system actually contains a file denoted by this abstract pathname and the application is allowed to write to the file; false otherwise.
5	public boolean exists() Tests whether the file or directory denoted by this abstract pathname exists. Returns true if and only if the file or directory denoted by this abstract pathname exists; false otherwise

Methods of File Class (Cont.)

Sr.	Method
6	public boolean isDirectory() Tests whether the file denoted by this abstract pathname is a directory. Returns true if and only if the file denoted by this abstract pathname exists and is a directory; false otherwise.
7	public boolean isFile() Tests whether the file denoted by this abstract pathname is a normal file. A file is normal if it is not a directory and, in addition, satisfies other system-dependent criteria
8	public long lastModified() Returns the time that the file denoted by this abstract pathname was last modified. Returns a long value representing the time the file was last modified, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970).
9	public long length() Returns the length of the file denoted by this abstract pathname.
10	public boolean delete() Deletes the file or directory.
11	public String[] list() Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.

File Class Example

```
import java.io.File;
class FileDemo {
    public static void main(String args[]) {
        File f1 = new File("FileDemo.java");
        System.out.println("File Name: " + f1.getName());
        System.out.println("Path: " + f1.getPath());
        System.out.println("Abs Path: " + f1.getAbsolutePath());
        System.out.println("Parent: " + f1.getParent());
        System.out.println(f1.exists() ? "exists" : "does not exist");
        System.out.println(f1.canWrite() ? "is writeable" : "is not writeable");
        System.out.println (f1.canRead () ? "is readable" : "is not readable");
        System.out.println ("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
        System.out.println(f1.isFile() ? "is normal file" : "might be a named pipe");
        System.out.println(f1.isAbsolute() ? "is absolute" : "is not absolute");
        System.out.println("File last modified: " + f1.lastModified());
        System.out.println("File size: " + f1.length() + " Bytes");
    }
}
```

Stream

- A stream can be defined as a sequence of data.
- All streams represent an input source and an output destination.
- There are two kinds of Streams
  - **Byte Stream**
  - **Character Stream**
- The **java.io** package contains all the classes required for input-output operations.
- The stream in the java.io package supports all the datatype including primitive.

Byte Streams

- Byte streams provide a convenient means for handling input and output of bytes.
- Byte streams are used, for example, when reading or writing binary data.

FileOutputStream

- Java `FileOutputStream` is an output stream for writing data to a file.
- `FileOutputStream` will create the file before opening it for output.
- On opening a read only file, it will throw an exception.

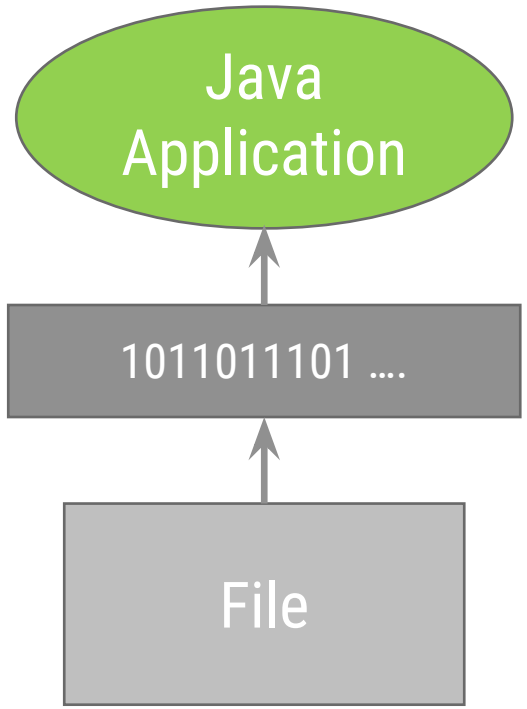
Sr.	Method
1	**void write(byte[] b)** This method writes b.length bytes from the specified byte array to this file output stream.
2	**void write(byte[] b, int off, int len)** This method writes len bytes from the specified byte array starting at offset off to this file output stream.
3	**void write(int b)** This method writes the specified byte to this file output stream.
4	**void close()** This method closes this file output stream and releases any system resources associated with this stream.

FileOutputStream Example

```
class FileOutDemo {
    public static void main(String args[]) {
        try {
            FileOutputStream fout = new FileOutputStream("abc.txt");
            String s = "Sourav Ganguly is my favorite player";
            byte b[] = s.getBytes();
            fout.write(b);
            fout.close();
            System.out.println("Success...");
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

FileInputStream

- `FileInputStream` class is used to read bytes from a file.
- It should be used to read byte-oriented data for example to read image, audio, video etc.



Sr.	Method
1	**public int read()**
2	**public int read(byte[] b)** b - the buffer into which the data is read. Returns: the total number of bytes read into the buffer, or -1.
3	**public int read(byte[] b, int off, int len)** b - the buffer into which the data is read. off - the start offset in the destination array b len - the maximum number of bytes read. Returns: the total number of bytes read into the buffer, or -1
4	**public long skip(long n)** n - the number of bytes to be skipped. Returns: the actual number of bytes skipped.
5	**public int available()** an estimate of the number of remaining bytes that can be read
6	**public void close()** Closes this file input stream and releases any system resources associated.

FileInputStream Example

```
class SimpleRead {
    public static void main(String args[]) {
        try {
            FileInputStream fin = new FileInputStream("abc.txt");
            int i = 0;
            while ((i = fin.read()) != -1) {
                System.out.println((char) i);
            }
            fin.close();
        } catch (Exception e) {
             System.out.println(e);
        }
    }
}
```

Example of Byte Streams

```
import java.io.*;
public class CopyFile {
    public static void main(String args[]) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    } }
```

Character Streams

- Character Streams provide a convenient means for handling input and output of characters.
- Internationalization is possible as it uses Unicode.
- For character streams we have two base classes
  - Reader
  - Writer

Reader

- The Java `Reader` class is the base class of all Reader's in the IO API.
- Subclasses include a `FileReader`, `BufferedReader`, `InputStreamReader`, `StringReader` and several others.
- Here is a simple Java IO Reader example:

```
Reader reader = new FileReader("c:\\data\\myfile.txt");
int data = reader.read();
while (data != -1) {
    char dataChar = (char) data;
    data = reader.read();
}
```

- Combining Readers with InputStream

```
Reader reader = new InputStreamReader("c:\\data\\myfile.txt");
```

Writer

- The Java `Writer` class is the base class of all Writers in the I-O API.
- Subclasses include `BufferedWriter`, `PrintWriter`, `StringWriter` and several others.
- Here is a simple Java IO Writer example:

```
Writer writer = new FileWriter("c:\\data\\file-output.txt");
writer.write("Hello World Writer");
writer.close();
```

- Combining Readers With OutputStreams

```
Writer writer = new OutputStreamWriter("c:\\data\\file-output.txt");
```

BufferedReader

- The `java.io.BufferedReader` class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
- Following are the important points about `BufferedReader`:
  - The buffer size may be specified, or the default size may be used.
  - Each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream.
- Constructors :

Sr.	Constructor
1	BufferedReader(Reader in) This creates a buffering character-input stream that uses a default-sized input buffer.
2	BufferedReader(Reader in, int sz) This creates a buffering character-input stream that uses an input buffer of the specified size.

BufferedReader (Methods)

Sr.	Methods
1	void close() This method closes the stream and releases any system resources associated with it.
2	int read() This method reads a single character.
3	int read(char[] cbuf, int off, int len) This method reads characters into a portion of an array.
4	String readLine() This method reads a line of text.
5	void reset() This method resets the stream.
6	long skip(long n) This method skips characters.

BufferedReader – Example

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

class BufferedReaderDemo {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("input.txt");
        BufferedReader br = new BufferedReader(fr);
        char c[] = new char[20];
        br.skip(8);
        if (br.ready()) {
            System.out.println(br.readLine());
            br.read(c);
            for (int i = 0; i < 20; i++) {
                System.out.print(c[i]);
            }
        }
    }
}
```