|  |  |
| --- | --- |
| Week5 | Java File Handling |

variables and arrays are used to store data inside Java programs. This approach creates two problems.

1. Data stored in a program is lost when the program is terminated. It means storage is temporary.
2. Large volumes of data are difficult to handle using variables and arrays.

To overcome these problems, we use the concept of files to store secondary storage devices called hard.

A **file** is a collection of related records located in a specific area on the disk

A **directory in Java** is a file within a file system that contains several files and other directories.

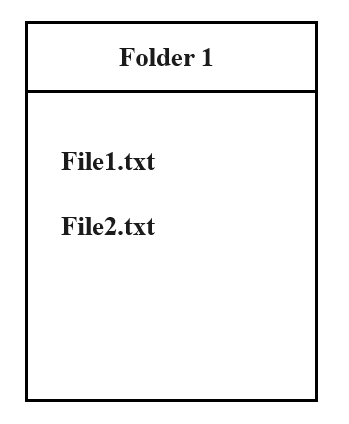
A file system is a hierarchy of files and directories, starting from a root directory.

In Java, with the help of File Class, we can work with files

File Class is inside the java.io package

|  |  |
| --- | --- |
| **File class Constructors** | **File class methods** |
| **1. File(String pathName):** This constructor creates a File object with the specified pathname. The pathname may be a file or a directory. It converts the given pathname string into an abstract pathname.  **2. File(String directoryPath, String filename):** This constructor creates a new File instance from a directory with the specified directory path and filename.  **3. File(File dirObject, String filename):** This constructor creates a new File instance from a File object dirObject that specifies a directory and a file having the name filename.  **4. File(URI uriObj):** uriObj is a URI object that describes a file. | 1. boolean exists():  2. boolean canExecute():  3. boolean canRead():  4. boolean canWrite():  5. boolean isAbsolute():  6. boolean isDirectory():  7. boolean isFile():  8. boolean isHidden():  9. boolean createNewFile():  10. boolean delete(): |

**WAP to create the following scenario & perform File operations**



How to set file permissions in java?

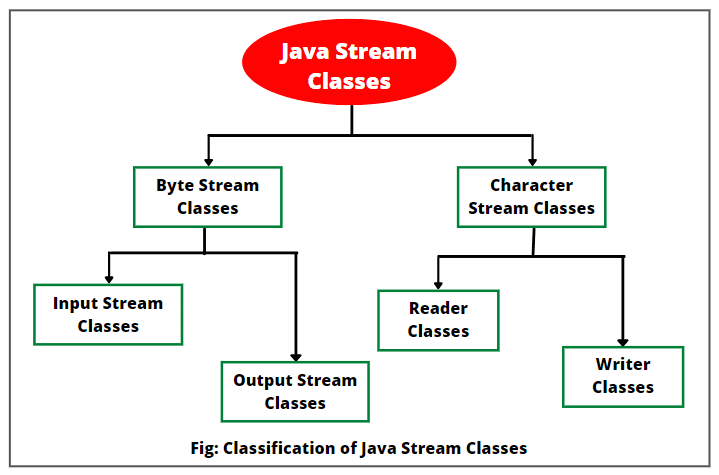
File scriptFile = new File("/test.sh");  
System.out.println("Current file permissions:");  
System.out.println("Can Execute "+scriptFile.canExecute());  
System.out.println("Can Read"+scriptFile.canRead());  
System.out.println("Can Write"+scriptFile.canWrite());  
scriptFile.setExecutable(true);  
scriptFile.setReadable(true);  
scriptFile.setWritable(true);

Java's IO package mostly concerns itself with the reading of raw data from a source and writing of raw data to a destination. The most typical sources and destinations of data are these:

* **Files**, Files are a common [source or destination](https://jenkov.com/tutorials/java-io/overview.html#inputoutput) of data in Java applications
* **Pipes**, Pipes in Java IO provides the ability for two threads running in the same JVM to communicate.
* **Network Connections**,  networking API which makes it easy to communicate via TCP/IP sockets or UDP sockets over the internet.
* **In-memory Buffers**, Byte and char arrays are often used in Java to temporarily store data internally in an application.
* **System.in, System.out, System.error**, Java has 3 streams commonly used to provide input to, and output from Java applications.

### **Streams in Java**

* In Java, a sequence of data is known as a **stream**.
* This concept is used to perform I/O operations on a file.
* There are two types of streams :



|  |  |
| --- | --- |
| **Byte streams** | **Character streams** |
| **Byte streams in Java** are designed to provide a convenient way for handling the input and output of bytes (i.e., units of 8-bits data).  Binary files are those files that are machine readable. For example, executable files, image files, and files in low-level file formats such as .zip, .class, .obj, and .exe.  Byte streams that are used for reading are called **input streams** and for writing are called **output streams**. They are represented by the abstract classes of InputStream and OutputStream in Java. | **Character streams in Java** are designed for handling the input and output of characters. They use 16-bit Unicode characters.  Character streams are **more efficient** than byte streams. They are mainly used for reading or writing to character or text-based I/O such as text files, text documents, XML, and HTML files.  Character streams that are used for reading are called **readers** and for writing are called **writers**. They are represented by the abstract classes of Reader and Writer in Java. |

|  |  |
| --- | --- |
| **Byte streams** | **Character streams** |
|  |  |
| **InputStream methods**  **1. int read():** int value in the range 0 to 255.  end of the stream is reached, the value –1 is returned.  If this method encounters an I/O error, it will throw an IOException.  **2. int read(byte[ ] b):**  reads and stores into the array of bytes b.  **3. int read(byte[ ] b, int n, int m):** m bytes of data from nth byte into an array b.  **4. int available():** returns an estimate of the number of bytes that can be read  **5. void close():** The close() method closes the input stream and releases any system resources associated with it.  **6. long skip(long n):** This method skips over n bytes of data from this input stream.  **7. void reset():** The reset() method is used to go back to the beginning of the stream.  **OutputStream methods**  **1. void write(int b):** The write() method writes the specified byte to the output stream. It accepts an int value as an input parameter. It throws an IOException if an I/O error occurs (e.g. output stream has been closed).  **2. void write(byte[ ] b):** This method writes all the specified bytes in the array b to the output stream.  **3. void write(byte[ ] b, int n, int m):** It writes m bytes from array b starting from nth byte to the output stream.  **4. void close():** It closes the output stream and releases any system resources associated with this stream.  **5. void flush():** It flushes the output stream and forces any buffered output bytes to be written out. | **Reader class methods** **1.int read()**It reads the next character from the input stream.  **2. int read(char[] cbuffer)**It reads a chunk of charaters from the input stream and store them in its byte array, cbuffer.  **3. int read(char[] cbuf, int off, int len)**It reads charaters into a portion of an array.  **4. int read(CharBuffer target)**It reads charaters into the specified character buffer.  **5. String readLine()**It reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.  **6. boolean ready()**It tells whether the stream is ready to be read.  **7. void close()**It closes the input stream and also frees any resources connected with this input stream. **Writer class methods** **1.void flush()**It flushes the output steam by forcing out buffered bytes to be written out.  **2.void write(char[] cbuf)**It writes a whole array(cbuf) to the output stream.  **3.void write(char[] cbuf, int off, int len)**It writes a portion of an array of characters.  **4.void write(int c)**It writes single character.  **5.void write(String str)**It writes a string.  **6.void write(String str, int off, int len)**It writes a portion of a string.  **7.Writer append(char c)**It appends the specified character to the writer.  **8.Writer append(CharSequence csq)**It appends the specified character sequence to the writer  **9.Writer append(CharSequence csq, int start, int end)**It appends a subsequence of the specified character sequence to the writer.  **10.void close()**It closes the output stream and also frees any resources connected with this output stream. |

**ByteArrayInputStream** class can turn a byte array into an InputStream.

**ByteArrayOutputStream**

**FilterInputStream** no special behaviour,It’s a base class for subclasses

**FilterOutputStream**

**ObjectInputStream** enables to read Java objects from an InputStream instead of raw bytes.

**ObjectOutputStream**

**Pipes** are communication channels between threads inside the same JVM.

**PipedInputStream** to read the contents of a pipe as a stream of bytes.

**PipedInputStream**

**SequenceInputStream** combines two or more other InputStream's into one.

**StringBufferInputStream** create an input stream in which the bytes read are supplied by the contents of a string.

**FileInputStream**, makes it possible to read the contents of a file as a stream of bytes.

**FileOutputStream**

**BufferedInputStream** class, provides transparent reading of chunks of bytes and buffering for a Java InputStream.

**DataInputStream** class enables you to read Java primitives (int, float, long etc.) from an InputStream instead of only raw bytes.

**PushbackInputStream** is intended to be used when you parse data from an InputStream. Sometimes you need to read ahead a few bytes to see what is coming, before you can determine how to interpret the current byte.

**InputStreamReader** interprets the bytes of an InputStream as text instead of numerical data.

**CharArrayReader** class enables you to read the contents of a char array as a character stream.

**FileReader** class, makes it possible to read the contents of a file as a stream of characters.

**PipedReader** class makes it possible to read the contents of a pipe as a stream of characters

**BufferedReader**, provides buffering for your Java Reader instances. Buffering can speed up IO quite a bit. Rather than read one character at a time from the underlying Reader, the Java BufferedReader reads a larger block (array) at a time. This is typically much faster, especially for disk access and larger data amounts.

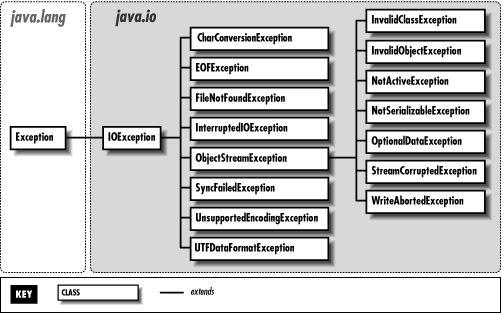
**FilterReader** is a base class for implementing your own filtering readers.

**PushbackReader** is intended to be used when you parse data from a [Reader](https://jenkov.com/tutorials/java-io/reader.html). Sometimes you need to read ahead a few characters to see what is coming, before you can determine how to interpret the current character

**LineNumberReader** is a BufferedReader that keeps track of line numbers of the read characters. Line numbering begins at 0. Whenever the LineNumberReader encounters a line terminator in the characters returned by the wrapped Reader, the line number is incremented.

**StringReader** class enables you to turn an ordinary String into a Reader.

#### **The exception classes in the java.io package**



**INTERVIEW QUESTIONS:**

Q.What is the difference between Serializable and Externalizable interface in Java?

Serializable interface exists in java.io package and forms core of java serialization mechanism. It doesn't have any method and also called Marker Interface in Java. When your class implements java.io.Serializable interface it becomes Serializable in Java and gives compiler an indication that use Java Serialization mechanism to serialize this object.  
Externalizable provides us writeExternal() and readExternal() method which gives us flexibility to control java serialization mechanism instead of relying on Java's default serialization.

Q.While serializing you want some of the members not to serialize? How do you achieve it?

If you don't want any field to be part of object's state then declare it either static or transient based on your need and it will not be included during Java serialization process.

Can we transfer a Serialized object vie network?

Yes you can transfer a Serialized object via network because java serialized object remains in form of bytes which can be transmitter via network. You can also store serialized object in Disk or database as Blob.

Q.What is RandomAccessFile?

It is a special class from java.io package which is neither a input stream nor a output stream (because it can do both). It is directly a subclass of Object class. Generally, a stream does only one purpose of either reading or writing; but RandomAccessFile can do both reading from a file and writing to a file. All the methods of DataInputStream and DataOutStream exist in RandomAccessFile.

Q. Tell something about BufferedWriter ? What are flush() and close() used for ?

Ans. A Buffer is a temporary storage area for data. The BufferedWriter class is an output stream.It is an abstract class that creates a buffered character-output stream.

Flush() is used to clear all the data characters stored in the buffer and clear the buffer.

Close() is used to closes the character output stream.

Q. What is Scanner class used for ? when was it introduced in Java ?

Ans. Scanner class introduced in Java 1.5 for reading Data Stream from the imput device. Previously we used to write code to read a input using DataInputStream. After reading the stream , we can convert into respective data type using in.next() as String ,in.nextInt() as integer, in.nextDouble() as Double etc

**RandomAccessFile**

RandomAccessFile(String path,String mode)

modes :

"r" , "rw"

public long getFilePointer()

public void skipBytes(long n)

public void seek(long n) moves the cursor position to the specified byte

The RandomAccessFile class in Java serves as a powerful tool for performing both read and write operations at any position within a file. Unlike other file access classes in Java like FileInputStream and FileOutputStream, which allow sequential access to files, RandomAccessFile allows direct access to any part of the file's contents.

Java RandomAccessFile provides the facility to read and write data to a file. RandomAccessFile works with file as large [array of bytes](https://www.digitalocean.com/community/tutorials/string-byte-array-java) stored in the file system and a cursor using which we can move the file pointer position.

RandomAccessFile class is part of [Java IO](https://www.digitalocean.com/community/tutorials/java-io-tutorial). While creating the instance of RandomAccessFile in java, we need to provide the mode to open the file.

* If the mode (in string) is **“r”,** the file can be read-only, but not written.
* If it is **“rw”,** the file is opened in read-write mode.
* If it is **“rws**”, the file is opened for read-write and every change to the file’s data will be immediately written to the physical device.
* **“rwd”:** same as “rw”, plus any changes to the file’s content, but not its metadata take effect immediately.

To get the current file pointer, you can call **getFilePointer()** method and to set the file pointer index, you can call  **seek(int position)** method moves the file pointer to a specified location.

|  |  |
| --- | --- |
| RandomAccessFile raf = new RandomAccessFile("file.txt", "r");  raf.seek(1);  byte[] bytes = new byte[5];  raf.read(bytes);  raf.close();  System.out.println(new String(bytes)); | In the first line, we are creating RandomAccessFile instance for the file in read-only mode. Then in the second line, we are moving the file pointer to index 1. We have created a byte array of length 5, so when we are calling read(bytes) method, then 5 bytes are read from file to the byte array. Finally, we are closing the RandomAccessFile resource and printing the byte array to console. |