Java.io.File Class in Java

* The File class is Java’s representation of a file or directory path name.
* The File class contains several methods for working with the path name, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.
* It is an abstract representation of file and directory pathnames.
* A pathname, whether abstract or in string form can be either absolute or relative. The parent of an abstract pathname may be obtained by invoking the getParent() method of this class.
* First of all, we should create the File class object by passing the filename or directory name to it.
* Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.
* A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as **access permissions.**

**How to create a File Object?**  
A File object is created by passing in a String that represents the name of a file, or a String or another File object. For example,

* File a = new File("/usr/local/bin/geeks");
* defines an abstract file name for the geeks file in directory /usr/local/bin. This is an absolute abstract file name.

**Constructors**

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

**Methods**

1. **boolean canExecute() :**Tests whether the application can execute the file denoted by this abstract pathname.
2. **boolean canRead()**: Tests whether the application can read the file denoted by this abstract pathname.
3. **boolean canWrite() :**Tests whether the application can modify the file denoted by this abstract pathname.
4. **int compareTo(File pathname) :**Compares two abstract pathnames lexicographically.
5. **boolean createNewFile() :**Atomically creates a new, empty file named by this abstract pathname .
6. **static File createTempFile(String prefix, String suffix) :**Creates an empty file in the default temporary-file directory.
7. **boolean delete() :**Deletes the file or directory denoted by this abstract pathname.
8. **boolean equals(Object obj) :**Tests this abstract pathname for equality with the given object.
9. **boolean exists()**: Tests whether the file or directory denoted by this abstract pathname exists.
10. **String getAbsolutePath() :**Returns the absolute pathname string of this abstract pathname.
11. **long getFreeSpace() :**Returns the number of unallocated bytes in the partition .
12. **String getName() :**Returns the name of the file or directory denoted by this abstract pathname.
13. **String getParent() :**Returns the pathname string of this abstract pathname’s parent.
14. **File getParentFile() :**Returns the abstract pathname of this abstract pathname’s parent.
15. **String getPath() :**Converts this abstract pathname into a pathname string.
16. **boolean isDirectory() :**Tests whether the file denoted by this pathname is a directory.
17. **boolean isFile() :**Tests whether the file denoted by this abstract pathname is a normal file.
18. **boolean isHidden() :**Tests whether the file named by this abstract pathname is a hidden file.
19. **long length() :**Returns the length of the file denoted by this abstract pathname.
20. **String[] list() :**Returns an array of strings naming the files and directories in the directory .
21. **File[] listFiles() :**Returns an array of abstract pathnames denoting the files in the directory.
22. **boolean mkdir() :**Creates the directory named by this abstract pathname.
23. **boolean renameTo(File dest) :**Renames the file denoted by this abstract pathname.
24. **boolean setExecutable(boolean executable) :**A convenience method to set the owner’s execute permission.
25. **boolean setReadable(boolean readable) :**A convenience method to set the owner’s read permission.
26. **boolean setReadable(boolean readable, boolean ownerOnly) :**Sets the owner’s or everybody’s read permission.
27. **boolean setReadOnly() :**Marks the file or directory named so that only read operations are allowed.
28. **boolean setWritable(boolean writable)**: A convenience method to set the owner’s write permission.
29. **String toString() :**Returns the pathname string of this abstract pathname.
30. **URI toURI() :**Constructs a file URI that represents this abstract pathname.

**Program 1:**Program to check if a file or directory physically exist or not.

import java.io.File;

// Displaying file property

class fileProperty

{

    public static void main(String[] args) {

        //accept file name or directory name through command line args

        String fname =args[0];

        //pass the filename or directory name to File object

        File f = new File(fname);

        //apply File class methods on File object

        System.out.println("File name :"+f.getName());

        System.out.println("Path: "+f.getPath());

        System.out.println("Absolute path:" +f.getAbsolutePath());

        System.out.println("Parent:"+f.getParent());

        System.out.println("Exists :"+f.exists());

        if(f.exists())

        {

            System.out.println("Is writeable:"+f.canWrite());

            System.out.println("Is readable"+f.canRead());

            System.out.println("Is a directory:"+f.isDirectory());

            System.out.println("File Size in bytes "+f.length());

        }

    }

}

File name :file.txt

Path: file.txt

Absolute path:C:\Users\akki\IdeaProjects\codewriting\src\file.txt

Parent:null

Exists :true

Is writeable:true

Is readabletrue

Is a directory:false

File Size in bytes 20

Here we will accept a directory name from the keyboard and then display all the contents of the directory .For this purpose, list() method can be used as:

String arr[]=f.list();

class Contents

{

    public static void main(String[] args) throws IOException {

String path=System.getProperty("user.dir");

        //create File object with dirpath and dname

        File f = new File(path);

        //if directory exists,then

        if(f.exists())

        {

            //get the contents into arr[]

            //now arr[i] represent either a File or Directory

            String arr[]=f.list();

            //find no. of entries in the directory

            int n=arr.length;

            //displaying the entries

            for (int i = 0; i < n ; i++) {

                System.out.println(arr[i]);

                //create File object with the entry and test

                //if it is a file or directory

                File f1=new File(arr[i]);

                if(f1.isFile())

                    System.out.println(": is a file");

                if(f1.isDirectory())

                    System.out.println(": is a directory");

            }

            System.out.println("No of entries in this directory "+n);

        }

        else

            System.out.println("Directory not found");

    }

}

# Different ways of Reading a text file in Java

There are multiple ways of writing and reading a text file. this is required while dealing with many applications.

There are several ways to read a plain text file in Java e.g. you can use [FileReader](http://www.geeksforgeeks.org/file-handling-java-using-filewriter-filereader/), [BufferedReader](http://www.geeksforgeeks.org/java-io-bufferedreader-class-java/)or [Scanner](http://www.geeksforgeeks.org/scanner-class-in-java/) to read a text file. Every utility provides something special e.g. BufferedReader provides buffering of data for fast reading, and Scanner provides parsing ability.

We can also use both BufferReader and Scanner to read a text file line by line in Java. Then Java SE 8 introduces another Stream class **java.util.stream.Stream** which provides a lazy and more efficient way to read a file

**Here are some of the many ways of reading files:**

1. **Using BufferedReader:**This method reads text from a character-input stream. It does buffering for efficient reading of characters, arrays, and lines.  
   The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.

In general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders. For example,

BufferedReader in = new BufferedReader(Reader in, int size);

// Java Program to illustrate reading from FileReader

// using BufferedReader

import java.io.\*;

public class ReadFromFile2

{

  public static void main(String[] args)throws Exception

  {

  // We need to provide file path as the parameter:

  // double backquote is to avoid compiler interpret words

  // like \test as \t (ie. as a escape sequence)

  File file = new File("C:\\Users\\pankaj\\Desktop\\test.txt");

  BufferedReader br = new BufferedReader(new FileReader(file));

  String st;

  while ((st = br.readLine()) != null)

    System.out.println(st);

  }

}

**2. Using FileReader class:** Convenience class for reading character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate.  
Constructors defined in this class are:

// Creates a new FileReader, given the

// File to read from.

FileReader(File file)

// Creates a new FileReader, given the

// FileDescriptor to read from.

FileReader(FileDescriptor fd)

// Creates a new FileReader, given the

// name of the file to read from.

FileReader(String fileName)

|  |
| --- |
| // Java Program to illustrate reading from  // FileReader using FileReader  import java.io.\*;  public class ReadingFromFile  {    public static void main(String[] args) throws Exception    {      // pass the path to the file as a parameter      FileReader fr =        new FileReader("C:\\Users\\pankaj\\Desktop\\test.txt");        int i;      while ((i=fr.read()) != -1)        System.out.print((char) i);    }  } |

Copying file using FileStreams in Java

We can copy a file from one location to another using FileInputStream and FileOutputStream classes in Java.  
For this we have to import some specific classes of java.io package. So for instance let us include the entire package with statement import java.io.\*;

The main logic of copying file is to read the file associated to FileInputStream variable and write the read contents into the file associated with FileOutputStream variable.

**Methods used in the program**

1. **int read()**; Reads a byte of data. Present in FileInputStream. Other versions of this method : int read(byte[] bytearray) and int read(byte[] bytearray, int offset, int length)
2. **void write(int b)** : Writes a byte of data. Present in FileOutputStream. Other versions of this method : void write(byte[] bytearray) and void write(byte[] bytearray, int offset, int length);

|  |
| --- |
| /\* Program to copy a src file to destination.     The name of src file and dest file must be     provided using command line arguments where     args[0] is the name of source file and     args[1] is name of destination file \*/    import java.io.\*;  class src2dest  {      public static void main(String args[])      throws FileNotFoundException,IOException      {          /\* If file doesnot exist FileInputStream throws             FileNotFoundException and read() write() throws             IOException if I/O error occurs \*/          FileInputStream fis = new FileInputStream(args[0]);            /\* assuming that the file exists and need not to be             checked \*/          FileOutputStream fos = new FileOutputStream(args[1]);            int b;          while  ((b=fis.read()) != -1)              fos.write(b);            /\* read() will readonly next int so we used while             loop here in order to read upto end of file and             keep writing the read int into dest file \*/          fis.close();          fos.close();      }  } |

File Permissions in Java

Java provides a number of method calls to check and change the permission of a file, such as a read-only file can be changed to have permissions to write. File permissions are required to be changed when the user want to restrict the operations permissible on a file. For example, a file permission can be changed from write to read-only because the user no longer want to edit the file.

**Checking the current file permissions**

A file can be in any combination of following permissible permissions:

* **Executable:** Tests whether the application can execute the file denoted by this abstract path name.  
  Syntax:
* **public boolean canExecute()**
* **Returns:** true if and only if the abstract path name

exists and the application is allowed to execute the file

* **Readable:**Tests whether the application can read the file denoted by this abstract path name.  
  Syntax:
* **public boolean canRead()**
* **Returns:** true if and only if the file specified by this

abstract path name exists and can be read by the application; false otherwise

* **Writable:** Tests whether the application can modify the file denoted by this abstract path name.  
  Syntax:
* **public boolean canWrite()**
* **Returns:** true if and only if the file system actually
* contains a file denoted by this abstract path name and

the application is allowed to write to the file; false otherwise.

For example, a file can be readable and writable but not executable. Here’s Java program to get the current permissions associated with a file.

|  |
| --- |
| // Java program to check the current file permissions.  import java.io.\*;    public class Test  {      public static void main(String[] args)      {          // creating a file instance          File file = new File("C:\\Users\\Mayank\\Desktop\\1.txt");            // check if the file exists          boolean exists = file.exists();          if(exists == true)          {              // printing the permissions associated with the file              System.out.println("Executable: " + file.canExecute());              System.out.println("Readable: " + file.canRead());              System.out.println("Writable: "+ file.canWrite());          }          else          {              System.out.println("File not found.");          }      }  } |

**Changing file permissions**

A file can have any combinations of the following permissions:

* Executable
* Readable
* Writable

Here are methods to change the permissions associated with a file:

* **setExecutable**A convenience method to set the owner’s execute permission for this abstract path name.
* **public boolean setExecutable(boolean executable)**
* **Description:**
* **Parameters:** executable - If true, sets the access
* permission to allow execute operations;
* if false to disallow execute operations

**Returns:** true if and only if the operation succeeded.

The operation will fail if the user does not have permission to change the access permissions of this abstract path name. If executable is false and the underlying file system does not implement an execute permission, then the operation will fail.

* **setReadable:** A convenience method to set the owner’s read permission for this abstract path name.
* **public boolean setReadable(boolean readable)**
* **Parameters:** readable - If true, sets the access permission to
* allow read operations; if false to disallow read operations

**Returns:** true if and only if the operation succeeded.

The operation will fail if the user does not have permission to change the access permissions of this abstract path name. If readable is false and the underlying file system does not implement a read permission, then the operation will fail.

* **setWritable :** A convenience method to set the owner’s write permission for this abstract path name.
* **public boolean setWritable(boolean writable)**
* **Parameters:** writable - If true, sets the access permission
* to allow write operations; if false to disallow write operations

**Returns:** true if and only if the operation succeeded.

The operation will fail if the user does not have permission to change the access permissions of this abstract path name.

|  |
| --- |
| // Java program to change the file permissions  import java.io.\*;    public class Test  {      public static void main(String[] args)      {          // creating a new file instance          File file = new File("C:\\Users\\Mayank\\Desktop\\1.txt");            // check if file exists          boolean exists = file.exists();          if(exists == true)          {              // changing the file permissions              file.setExecutable(true);              file.setReadable(true);              file.setWritable(false);              System.out.println("File permissions changed.");                // printing the permissions associated with the file currently              System.out.println("Executable: " + file.canExecute());              System.out.println("Readable: " + file.canRead());              System.out.println("Writable: "+ file.canWrite());            }          else          {              System.out.println("File not found.");          }      }  } |

Moving a file from one directory to another using Java

Java provides functions to move files between directories. Two ways to achieve this are described here. The first method utilizes Files package for moving while the other method first copies the file to destination and then deletes the original copy from the source.

1. **Using Files.Path move() method:** Renaming and moving the file permanently to a new location.  
   **Syntax:**
2. **public static Path move(Path source, Path target, CopyOption..options)**
3. **throws IOException**
4. **Parameters:**
5. **source** - the path to the file to move
6. **target** - the path to the target file
7. (may be associated with a different provider to the source path)
8. **options** - options specifying how the move should be done

**Returns:** the path to the target file

|  |
| --- |
| // Java program to illustrate renaming and  // moving a file permanently to a new loaction  import java.io.\*;  import java.nio.file.Files;  import java.nio.file.\*;    public class Test  {      public static void main(String[] args) throws IOException      {          Path temp = Files.move          (Paths.get("C:\\Users\\Mayank\\Desktop\\44.txt"),          Paths.get("C:\\Users\\Mayank\\Desktop\\dest\\445.txt"));            if(temp != null)          {              System.out.println("File renamed and moved successfully");          }          else          {              System.out.println("Failed to move the file");          }      }  } |

**2. Using Java.io.File.renameTo() and Java.io.File.delete() methods:** Copying the file and deleting the original file using these two methods.  
**Syntax of renameTo():**

**public boolean renameTo(File dest)**

**Description:** Renames the file denoted by this abstract path name.

**Parameters:** dest - The new abstract path name for the named file

**Returns:** true if and only if the renaming succeeded; false otherwise

**Syntax of delete():**

**public boolean delete()**

**Description:** Deletes the file or directory

denoted by this abstract path name.

**Returns:** true if and only if the file or

directory is successfully deleted; false otherwise

|  |
| --- |
| // Java program to illustrate Copying the file  // and deleting the original file  import java.io.\*;    public class Test  {      public static void main(String[] args)      {          File file = new File("C:\\Users\\Mayank\\Desktop\\1.txt");            // renaming the file and moving it to a new location          if(file.renameTo             (new File("C:\\Users\\Mayank\\Desktop\\dest\\newFile.txt")))          {              // if file copied successfully then delete the original file              file.delete();              System.out.println("File moved successfully");          }          else          {              System.out.println("Failed to move the file");          }          }  } |