**Java File Class**

The File class is an abstract representation of file and directory pathname. A pathname can be either absolute or relative.

The File class have several methods for working with directories and files such as creating new directories or files, deleting and renaming directories or files, listing the contents of a directory etc.

Fields

|  |  |  |  |
| --- | --- | --- | --- |
| **Modifier** | **Type** | **Field** | **Description** |
| static | String | pathSeparator | It is system-dependent path-separator character, represented as a [string](https://www.javatpoint.com/java-string) for convenience. |
| static | char | pathSeparatorChar | It is system-dependent path-separator character. |
| static | String | separator | It is system-dependent default name-separator character, represented as a string for convenience. |
| static | char | separatorChar | It is system-dependent default name-separator character. |

[Constructors](https://www.javatpoint.com/java-constructor)

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| File(File parent, String child) | It creates a new File instance from a parent abstract pathname and a child pathname string. |
| File(String pathname) | It creates a new File instance by converting the given pathname string into an abstract pathname. |
| File(String parent, String child) | It creates a new File instance from a parent pathname string and a child pathname string. |
| File(URI uri) | It creates a new File instance by converting the given file: URI into an abstract pathname. |

Useful Methods

|  |  |  |
| --- | --- | --- |
| **Modifier and Type** | **Method** | **Description** |
| static File | createTempFile(String prefix, String suffix) | It creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name. |
| boolean | createNewFile() | It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. |
| boolean | canWrite() | It tests whether the application can modify the file denoted by this abstract pathname.String[] |
| boolean | canExecute() | It tests whether the application can execute the file denoted by this abstract pathname. |
| boolean | canRead() | It tests whether the application can read the file denoted by this abstract pathname. |
| boolean | isAbsolute() | It tests whether this abstract pathname is absolute. |
| boolean | isDirectory() | It tests whether the file denoted by this abstract pathname is a directory. |
| boolean | isFile() | It tests whether the file denoted by this abstract pathname is a normal file. |
| String | getName() | It returns the name of the file or directory denoted by this abstract pathname. |
| String | getParent() | It returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory. |
| Path | toPath() | It returns a java.nio.file.Path object constructed from the this abstract path. |
| URI | toURI() | It constructs a file: URI that represents this abstract pathname. |
| File[] | listFiles() | It returns an [array](https://www.javatpoint.com/array-in-java) of abstract pathnames denoting the files in the directory denoted by this abstract pathname |
| long | getFreeSpace() | It returns the number of unallocated bytes in the partition named by this abstract path name. |
| String[] | list(FilenameFilter filter) | It returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter. |
| boolean | mkdir() | It creates the directory named by this abstract pathname. |

Java File Example 1

1. **import** java.io.\*;
2. **public** **class** FileDemo {
3. **public** **static** **void** main(String[] args) {
5. **try** {
6. File file = **new** File("javaFile123.txt");
7. **if** (file.createNewFile()) {
8. System.out.println("New File is created!");
9. } **else** {
10. System.out.println("File already exists.");
11. }
12. } **catch** (IOException e) {
13. e.printStackTrace();
14. }
16. }
17. }

Output:

New File is created!

Java File Example 2

1. **import** java.io.\*;
2. **public** **class** FileDemo2 {
3. **public** **static** **void** main(String[] args) {
5. String path = "";
6. **boolean** bool = **false**;
7. **try** {
8. // createing  new files
9. File file  = **new** File("testFile1.txt");
10. file.createNewFile();
11. System.out.println(file);
12. // createing new canonical from file object
13. File file2 = file.getCanonicalFile();
14. // returns true if the file exists
15. System.out.println(file2);
16. bool = file2.exists();
17. // returns absolute pathname
18. path = file2.getAbsolutePath();
19. System.out.println(bool);
20. // if file exists
21. **if** (bool) {
22. // prints
23. System.out.print(path + " Exists? " + bool);
24. }
25. } **catch** (Exception e) {
26. // if any error occurs
27. e.printStackTrace();
28. }
29. }
30. }

Output:

testFile1.txt

/home/Work/Project/File/testFile1.txt

true

/home/Work/Project/File/testFile1.txt Exists? true

Java File Example 3

1. **import** java.io.\*;
2. **public** **class** FileExample {
3. **public** **static** **void** main(String[] args) {
4. File f=**new** File("/Users/sonoojaiswal/Documents");
5. String filenames[]=f.list();
6. **for**(String filename:filenames){
7. System.out.println(filename);
8. }
9. }
10. }

Output:

"info.properties"

"info.properties".rtf

.DS\_Store

.localized

Alok news

apache-tomcat-9.0.0.M19

apache-tomcat-9.0.0.M19.tar

bestreturn\_org.rtf

BIODATA.pages

BIODATA.pdf

BIODATA.png

struts2jars.zip

workspace

Java File Example 4

1. **import** java.io.\*;
2. **public** **class** FileExample {
3. **public** **static** **void** main(String[] args) {
4. File dir=**new** File("/Users/sonoojaiswal/Documents");
5. File files[]=dir.listFiles();
6. **for**(File file:files){
7. System.out.println(file.getName()+" Can Write: "+file.canWrite()+"
8. Is Hidden: "+file.isHidden()+" Length: "+file.length()+" bytes");
9. }
10. }
11. }

Output:

"info.properties" Can Write: true Is Hidden: false Length: 15 bytes

"info.properties".rtf Can Write: true Is Hidden: false Length: 385 bytes

.DS\_Store Can Write: true Is Hidden: true Length: 36868 bytes

.localized Can Write: true Is Hidden: true Length: 0 bytes

Alok news Can Write: true Is Hidden: false Length: 850 bytes

apache-tomcat-9.0.0.M19 Can Write: true Is Hidden: false Length: 476 bytes

apache-tomcat-9.0.0.M19.tar Can Write: true Is Hidden: false Length: 13711360 bytes

bestreturn\_org.rtf Can Write: true Is Hidden: false Length: 389 bytes

BIODATA.pages Can Write: true Is Hidden: false Length: 707985 bytes

BIODATA.pdf Can Write: true Is Hidden: false Length: 69681 bytes

BIODATA.png Can Write: true Is Hidden: false Length: 282125 bytes

workspace Can Write: true Is Hidden: false Length: 1972 bytes

**How to read and write Java object to a file**

Java object [Serialization](https://en.wikipedia.org/wiki/Serialization) is an API provided by Java Library stack as a means to serialize Java objects. Serialization is a process to convert objects into a writable byte stream. Once converted into a byte-stream, these objects can be written to a file. The reverse process of this is called de-serialization.

A Java object is serializable if its class or any of its superclasses implement either the java.io.Serializable interface or its subinterface, java.io.Externalizable.

1. Java Object

Person.java

package com.mkyong;

import java.io.Serializable;

public class Person implements Serializable {

private static final long serialVersionUID = 1L;

private String name;

private int age;

private String gender;

Person() {

};

Person(String name, int age, String gender) {

this.name = name;

this.age = age;

this.gender = gender;

}

@Override

public String toString() {

return "Name:" + name + "\nAge: " + age + "\nGender: " + gender;

}

}

2. Writing and Reading objects in Java

The objects can be converted into byte-stream using java.io.ObjectOutputStream. In order to enable writing of objects into a file using ObjectOutputStream, it is mandatory that the concerned class implements Serializable interface as shown in the class definition below.

Reading objects in Java are similar to writing object using ObjectOutputStreamObjectInputStream. Below example shows the complete cycle of writing objects and reading objects in Java.

On reading objects, the ObjectInputStream directly tries to map all the attributes into the class into which we try to cast the read object. If it is unable to map the respective object exactly then it throws a ClassNotFound exception.

Let us now understand the writing and reading process using an example. We are using the Person class shown above as an object.

WriterReader.java

package com.mkyong;

package com.mkyong;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

public class WriterReader {

public static void main(String[] args) {

Person p1 = new Person("John", 30, "Male");

Person p2 = new Person("Rachel", 25, "Female");

try {

FileOutputStream f = new FileOutputStream(new File("myObjects.txt"));

ObjectOutputStream o = new ObjectOutputStream(f);

// Write objects to file

o.writeObject(p1);

o.writeObject(p2);

o.close();

f.close();

FileInputStream fi = new FileInputStream(new File("myObjects.txt"));

ObjectInputStream oi = new ObjectInputStream(fi);

// Read objects

Person pr1 = (Person) oi.readObject();

Person pr2 = (Person) oi.readObject();

System.out.println(pr1.toString());

System.out.println(pr2.toString());

oi.close();

fi.close();

} catch (FileNotFoundException e) {

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

} catch (IOException e) {

System.out.println("Error initializing stream");

} catch (ClassNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

On executing the above code, we get below output:

Name:John

Age: 30

Gender: Male

Name:Rachel

Age: 25

Gender: Female

**Java.io package**

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

* **InPutStream** − The InputStream is used to read data from a source.
* **OutPutStream** − The OutputStream is used for writing data to a destination.



Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one −

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file −

**Example**

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();

}

}

}

}

Now let's have a file **input.txt** with the following content −

This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

$javac CopyFile.java

$java CopyFile

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file −

**Example**

import java.io.\*;

public class CopyFile {

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

FileReader in = null;

FileWriter out = null;

try {

in = new FileReader("input.txt");

out = new FileWriter("output.txt");

int c;

while ((c = in.read()) != -1) {

out.write(c);

}

}finally {

if (in != null) {

in.close();

}

if (out != null) {

out.close();

}

}

}

}

Now let's have a file **input.txt** with the following content −

This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

$javac CopyFile.java

$java CopyFile

Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams −

* **Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
* **Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
* **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

Following is a simple program, which creates **InputStreamReader** to read standard input stream until the user types a "q" −

**Example**

[Live Demo](http://tpcg.io/lVH2u1)

import java.io.\*;

public class ReadConsole {

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

InputStreamReader cin = null;

try {

cin = new InputStreamReader(System.in);

System.out.println("Enter characters, 'q' to quit.");

char c;

do {

c = (char) cin.read();

System.out.print(c);

} while(c != 'q');

}finally {

if (cin != null) {

cin.close();

}

}

}

}

Let's keep the above code in ReadConsole.java file and try to compile and execute it as shown in the following program. This program continues to read and output the same character until we press 'q' −

$javac ReadConsole.java

$java ReadConsole

Enter characters, 'q' to quit.

1

1

e

e

q

q

Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed in this tutorial.

FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file −

InputStream f = new FileInputStream("C:/java/hello");

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows −

File f = new File("C:/java/hello");

InputStream f = new FileInputStream(f);

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public void close() throws IOException{}**  This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException. |
| 2 | **protected void finalize()throws IOException {}**  This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException. |
| 3 | **public int read(int r)throws IOException{}**  This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file. |
| 4 | **public int read(byte[] r) throws IOException{}**  This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned. |
| 5 | **public int available() throws IOException{}**  Gives the number of bytes that can be read from this file input stream. Returns an int. |

There are other important input streams available, for more detail you can refer to the following links −

* [ByteArrayInputStream](https://www.tutorialspoint.com/java/java_bytearrayinputstream.htm)
* [DataInputStream](https://www.tutorialspoint.com/java/java_datainputstream.htm)

FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file −

OutputStream f = new FileOutputStream("C:/java/hello")

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows −

File f = new File("C:/java/hello");

OutputStream f = new FileOutputStream(f);

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public void close() throws IOException{}**  This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException. |
| 2 | **protected void finalize()throws IOException {}**  This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException. |
| 3 | **public void write(int w)throws IOException{}**  This methods writes the specified byte to the output stream. |
| 4 | **public void write(byte[] w)**  Writes w.length bytes from the mentioned byte array to the OutputStream. |

There are other important output streams available, for more detail you can refer to the following links −

* [ByteArrayOutputStream](https://www.tutorialspoint.com/java/java_bytearrayoutputstream.htm)
* [DataOutputStream](https://www.tutorialspoint.com/java/java_dataoutputstream.htm)

**Example**

Following is the example to demonstrate InputStream and OutputStream −

import java.io.\*;

public class fileStreamTest {

public static void main(String args[]) {

try {

byte bWrite [] = {11,21,3,40,5};

OutputStream os = new FileOutputStream("test.txt");

for(int x = 0; x < bWrite.length ; x++) {

os.write( bWrite[x] ); // writes the bytes

}

os.close();

InputStream is = new FileInputStream("test.txt");

int size = is.available();

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

System.out.print((char)is.read() + " ");

}

is.close();

} catch (IOException e) {

System.out.print("Exception");

}

}

}

The above code would create file test.txt and would write given numbers in binary format. Same would be the output on the stdout screen.

File Navigation and I/O

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

* [File Class](https://www.tutorialspoint.com/java/java_file_class.htm)
* [FileReader Class](https://www.tutorialspoint.com/java/java_filereader_class.htm)
* [FileWriter Class](https://www.tutorialspoint.com/java/java_filewriter_class.htm)

Directories in Java

A directory is a File which can contain a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail, check a list of all the methods which you can call on File object and what are related to directories.

Creating Directories

There are two useful **File** utility methods, which can be used to create directories −

* The **mkdir( )** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
* The **mkdirs()** method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory −

**Example**

import java.io.File;

public class CreateDir {

public static void main(String args[]) {

String dirname = "/tmp/user/java/bin";

File d = new File(dirname);

// Create directory now.

d.mkdirs();

}

}

Compile and execute the above code to create "/tmp/user/java/bin".

**Note** − Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

Listing Directories

You can use **list( )** method provided by **File** object to list down all the files and directories available in a directory as follows −

**Example**

import java.io.File;

public class ReadDir {

public static void main(String[] args) {

File file = null;

String[] paths;

try {

// create new file object

file = new File("/tmp");

// array of files and directory

paths = file.list();

// for each name in the path array

for(String path:paths) {

// prints filename and directory name

System.out.println(path);

}

} catch (Exception e) {

// if any error occurs

e.printStackTrace();

}

}

}

This will produce the following result based on the directories and files available in your **/tmp** directory −

**Output**

test1.txt

test2.txt

ReadDir.java

ReadDir.class