Theory: Files

23 minutes 9 / 9 problems solved

Start practicing

It is often the case that a program needs to process and store data located outside the codebase: configuration settings, some dataset for processing, logs of execution and so on. The simplest way to store data is to use files that are supported by all modern operating systems. You can consider a file as a collection of data that is stored on a disk or another device, and that can be manipulated as a single unit when addressed by its name. Files can be organized into directories that act as folders for other files and directories.

In this topic, we will learn how to work with files directly from a Java program.

2 required topics
Objects In project 13
Running programs on your computer In project 2
4 dependent topics
Managing files In project
Writing files In project
Reading files In project
File hierarchies In project

§1. The File class

There is a class called `File` in the `java.io` package. An object of this class represents an existing or non-existing file or a directory. The class can be used to manipulate files and directories: creating, removing, accessing properties and more.

The simplest way to create a file object is to pass a string path to its constructor. The valid format of the string depends on the operating system:

- Windows uses backslashes for paths (`'\'`),
- Linux, OS X, Android and other UNIX-like systems use the forward slash (`'/'`).

You should keep this difference in mind while working with files.

If your operating system is Windows, do not forget to use the escape character `'\'`.

Let's create two objects of the `File` class for different platforms.

```
File fileOnUnix = new File("/home/username/Documents");    // a directory o
n a UNIX-like system
   File fileOnWin = new File("D:\\Materials\\java-
materials.pdf"); // a file on Windows
```

The code will work even if a file or a directory does not actually exist in your file system. It does not create a new file or directory. It just represents "a virtual" file or directory that exists already or may be created in the future.

To display the character for separating the path to a file in your case, you can print the following:

```
System.out.println(File.separator); // '/' - for Linux
```

Objects of the `File` class are immutable; that is, once created, the abstract pathname represented by an object will never change.

§2. Absolute and relative path

You've already seen examples of files described by an **absolute path**. Simply, a path is **absolute** if it starts with the root element of the file system. It has the complete information about the file location including the type of the operating system.

It is considered bad practice to locate a file using its absolute path inside your programs, since you will lose the ability to reuse your program on different platforms. Another problem is that you cannot transfer the file

along with the specified directory, you will have to change the code that accesses it.

A **relative path** is a path that doesn't include the root element of the file system. This always starts from your **working directory**. This directory is represented by a `.` (dot). A relative path is not complete and needs to be combined with the current directory path in order to reach the requested file.

Here is an example with a file inside the images directory which is in your working directory:

```
File fileOnUnix = new File("./images/picture.jpg");
File fileOnWin = new File("./images/picture.jpg");
```

As you can see, both paths look exactly the same, which provides platform independence. Interestingly, the dot character can be skipped, so the path `images/picture.jpg` is also correct.

In order to construct platform-independent programs, it is a common convention to use relative paths whenever possible. You can also transfer the working directory that contains `images/picture.jpg` without any code modifications.

To access the parent directory, just write `..` (double dot). So, `../picture.jpg` is a file placed in the parent directory of the working directory. The relative path `images/../images/picture.jpg` means the parent directory of `images`, then the `images` folder again. And `picture.jpg` is the file inside `images` folder. In general `images/../images/picture.jpg` and `images/picture.jpg` are the same paths.

§3. Basic methods

An instance of `File` has a list of methods. Take a look at some of them:

- `String getPath()` returns the string path to this file or directory;
- `String getName()` returns the name of this file or directory (just the last name of the path)
- `boolean isDirectory()` returns `true` if it is a directory and exists, otherwise, `false`;
- `boolean isFile()` returns `true` if it is a file that exists (not a directory), otherwise, `false`;
- `boolean exists()` returns `true` if this file or directory actually exists in your file system, otherwise, `false`;
- `String getParent()` returns the string path to the parent directory that contains this file or directory.

The list is not complete, but for now, we will focus on these ones. For other methods, see here.

Let's create an instance of an existing file and print out some info about it.

```
File file = new File("/home/username/Documents/javamaterials.pdf");

System.out.println("File name: " + file.getName());
System.out.println("File path: " + file.getPath());
System.out.println("Is file: " + file.isFile());
System.out.println("Is directory: " + file.isDirectory());
System.out.println("Exists: " + file.exists());
System.out.println("Parent path: " + file.getParent());
```

As we expect, the code prints the following:

```
File name: javamaterials.pdf
File path: /home/username/Documents/javamaterials.pdf
Is file: true
Is directory: false
Exists: true
Parent path: /home/username/Documents
```

Suppose now we have an instance that represents a non-existing file and prints the details about it:

```
File name: javamaterials1.pdf
File path: /home/art/Documents/javamaterials1.pdf
Is file: false
Is directory: false
Exists: false
Parent path: /home/art/Documents
```

The file does not exist, so the application does not know its type.

There is also a group of methods `canRead()`, `canWrite()`, `canExecute()` to test whether the application can **read/modify/execute** the file denoted by the path. It is recommended to use these methods, otherwise, you can encounter file access errors when your user does not have enough permissions to perform an operation with a file.

We believe the `File` class provides a very clear API to process files and directories on different platforms.