Understanding the Main Java I/O Concepts

Introducing the Java I/O API

The I/O in "Java I/O" stands for Input / Output. The Java I/O API gives all the tools your application needs to access inforı

With the Java I/O API, you can read and write files, as well as get and send data over a network using different protocols.

Understanding Java I/O, Java NIO and Java NIO2

The Java I/O API was created in the mid-90s along with the first versions of the JDK.

In 2002, with Java SE 1.4, Java NIO was released, with new classes, concepts and features. NIO stands for Non-blocking

In 2011, with Java SE 7, Java NIO2 was released, with more classes and concepts. It also brought new patterns for Java I

This Java I/O tutorial covers the three parts of the API: Java I/O, Java NIO, and Java NIO2.

Accessing a File

There are two main concepts in Java I/O: locating the resource you need to access (it can be a file or a network resource)

There are two ways to access files in the Java I/O API: the first one uses the `File` class and the second one uses the `Pat`

The `File` class was introduced in Java SE 1.0: it represents the legacy way to access files. You can see this class as a wr

Starting with Java SE 7, the `Path` interface was introduced, as part of the Java NIO2 API. The role of this interface is to fix

- Many methods of the `File` class do not throw exceptions when they fail, making it impossible to obtain a useful error
- The rename method doesn't work consistently across platforms.
- There is no real support for symbolic links.
- More support for metadata is desired, such as file permissions, file owner, and other security attributes.
- Accessing file metadata is inefficient.
- Many of the `File` methods do not scale. Requesting a large directory listing over a server can result in a hang. Large c
- It is not possible to write reliable code that can recursively walk a file tree and respond appropriately if there are circul

All these issues are fixed by the `Path` interface. So using the `File` class is not recommended anymore.

This tutorial has a section on how you can refactor your old-style `File` code to using the `Path` interface.

Understanding I/O Streams

An *I/O Stream* represents an input source or an output destination. A stream can represent many different kinds of source

Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and obje

No matter how they work internally, all streams present the same simple model to programs that use them: a stream is a

The data source and data destination can be anything that holds, generates, or consumes data. Obviously this includes di

I/O streams are a different concept than the streams from the Stream API introduced in Java SE 8. Even if the name is the

The Java I/O API defines two kinds of content for a resource:

- character content, think of a text file, a XML or JSON document,
- and byte content, think of an image or a video.

It also defines two operations on this content: reading and writing.

Following this, the Java I/O API defines four base classes, that are abstract, each modeling a type of I/O stream and a spe

	Reading	**Writing**
Streams of character	`Reader`	`Writer`
Streams of bytes	`InputStream`	`OutputStream`

All byte streams are descended from `InputStream` and `OutputStream`, there are many of them. Some of them are coverec

All character stream classes are descended from `Reader` and `Writer`.

Last update: *January 25, 2023*

Back to Tutorial List