Java Magazine

Coding, Java 17

Modern file input/output with Java Path API and Files helper methods

Ben Evans

September 8, 2021

When developers need to write file input or output code in Java, the situation can seem quite complex, especially because several APIs are available. It is not always obvious to newcomers which API they should use for common operations—or what the other APIs do.

This situation is because Java has had I/O support since the very first version. These earliest versions of I/O functionality emphasized portability due to Java's strong desire for platform independence. As a result, they were not always easy to work with and some key functionality was missing.

Subsequent versions of Java introduced other APIs to make developers' lives easier, but it wasn't until Java 7 arrived that Java's I/O capabilities became streamlined and easy to use.

For modern Java development, you should go straight to these modern APIs (known as NIO.2 or just the `Path` API) for simple tasks. All the other APIs—including low-level I/O handling—are still there if you need them, but for most day-to-day tasks the simple `Path` API should suffice.

There are two parts to the simple API, which lives in the package `java.nio.file`. Most importantly, there's the `Path` interface itself. There are also a bunch of helper methods to implement common file-handling tasks, which are mostly contained in the `Files` class.

Let's meet the API, starting with the fundamental abstraction, the `Path` itself.

Introducing Java's Path API

An instance of `Path` is an object that may be used to locate a file in a file system. It represents a file system location that

- Is hierarchical
- Is composed of a sequence of *path elements*
- Is system-dependent in its implementation
- May or may not correspond to an existing file

It's important to note that the `Path` API can be used to reference files that have not yet been created or that have already been deleted; hence, the last point above.

These aspects make a `Path` fundamentally different from the other common (but much older) Java abstraction for I/O: the `File` object.

In fact, `Path` is an interface not a class. This enables different operating systems and other file system providers to create different implementations of the `Path` interface. This provides a shared abstraction but also allows for system-dependent capabilities that do not have to be supported on all file systems.

All `Path` instances can be thought of as consisting of zero or more *directory names* and then one *name element*. In addition, some paths have a root component (such as `/` or `C:\`), but this is not mandatory because some instances (such as objects representing *relative paths*) do not have a root.

The name element is the element "furthest" from the root of the directory hierarchy and represents the name of the file or directory. You can intuitively think of a `Path` as consisting of the elements joined together by a delimiter. That delimiter is whatever is appropriate for the operating system.

Java 7 shipped with a `Paths` class that provides factory methods for creating `Path` objects. In Java 11 and later, these methods are also available as static methods directly on the `Path` interface.

The factories provide methods (`Paths.get ()` and `Path.of()`) for creating `Path` objects. The usual overload of those methods takes a `String` and uses the default file system provider. An alternative version takes a URI instead; this provides a hook for the ability of NIO.2 to plug in providers of custom or nonstandard file system implementations, such as network-aware file systems or zip files.

Once you have a `Path` object, you can use it to perform common operations by means of the convenience and utility methods to deal with files and file systems, many of which are contained as static methods in the `Files` class.

The helper methods

Zealous proponents of other languages sometimes complain that Java has too much boilerplate. Historically, this may have been somewhat accurate for Java I/O, but times have changed for the better. For example, in modern Java a straightforward copy operation is now as simple as the following:

```
var input = Path.of("input");
var output = Path.of("output");

try {
  Files.copy(input, output);
} catch(IOException ex) {
  ex.printStackTrace();
}
```

Copy code snippet

This is roughly the same amount of code as one would expect in a supposedly more-compact language. What's more, this code is safer, because the checked exception forces the programmer to confront the all-too-common situation that the copy might fail, rather than just permitting a silent failure to cause further downstream problems.

Warning: You will notice that the only checked exception thrown by methods in `Path` is an `IOException`. This aids the goal of coding simplicity, but it can obscure an underlying problem sometimes. You may need to write some extra exception handling if you want to deal with an explicit subtype of `IOException`.

The `Files` class is huge, and it's worth getting familiar with the methods in it. (There are also variants that make use of the older APIs, including `File`, but let's start by just using the modern API.)

Here's my suggestion: Before writing a file I/O method from scratch, check to see whether `Files` already has a method that does what you want (or something close to it).

The code below shows some of the major methods in `Files`. Their operation is generally self-explanatory. Some of the `Files` methods provide the opportunity to pass optional arguments so you can provide additional (possibly implementation-specific) behavior for the operation.

In the comments below, I am explicit about the return types for those methods where the return type is important. Some of the other methods, such as `copy()`, do not have particularly useful return types. The point of those methods is the side effects, that is, actually copying the file.

```
Path source, target;
Attributes attr;

// Creating files
//
// Example of path --> /home/ben/.profile
// Example of attributes --> rw-rw-rw-
Files.createFile(target, attr);

// Deleting files
Files.delete(target);
boolean deleted = Files.deleteIfExists(target);

// Copying/moving files
Files.copy(source, target);
Files.move(source, target);

// Utility methods to retrieve information
long size = Files.size(target);
```

```
FileTime fTime = Files.getLastModifiedTime(target);
System.out.println(fTime.to(TimeUnit.SECONDS));

Map<String, ?> attrs = Files.readAttributes(target, "*");
System.out.println(attrs);

// File metadata
boolean isDir = Files.isDirectory(target);
boolean isSym = Files.isSymbolicLink(target);

// Reading and writing
Charset cs = StandardCharsets.UTF_8;
List<String> lines = Files.readAllLines(source, cs);
byte[] b = Files.readAllBytes(source);

// You should, of course, never store a stream in a temp variable.
// This example is just to indicate the return type, and in real
// code would be used as the source of a stream pipeline.
Stream<String> tmp = Files.lines(source, cs);
```

Copy code snippet

Some of the methods above have default behavior you need to be aware of. For example, by default, a copy operation will not overwrite an existing file, so you need to specify this behavior as a *copy option*, as follows:

```
Files.copy(Path.of("input.txt"), Path.of("output.txt"),
           StandardCopyOption.REPLACE_EXISTING);
```

Copy code snippet

`StandardCopyOption` is an enum that implements an interface named `CopyOption`. This interface is implemented by an enum called `LinkOption`. The former case is for regular files, while the link form is to specify how symbolic links should be handled—provided the underlying OS supports symbolic links, of course.

This slight bit of extra complexity is used to allow `Files.copy()` and other methods to have a final, variadic argument of `CopyOption` allowing the developer to have a very flexible way of specifying the required copy behavior. A version of this pattern is seen in a couple of other places in the API.

A brief yet relevant history of Java I/O

One reason why Java is so popular is because its rich libraries offer powerful and concise APIs to solve most of your programming needs. However, there are a few areas in which older versions of Java weren't quite up to scratch. One example of this type of headache for developers was traditionally in I/O APIs. There have been several attempts to resolve this issue, leading to several different I/O APIs.

- Java 1.0: The `File`, `InputStream`, and `OutputStream` APIs
- Java 1.1: The Readers and Wrtiers API

- Java 1.4: The New I/O (NIO) API
- Java 7: The `Path` API (NIO.2)

The original `java.io.File` class, in particular, had significant limitations.

- It did not deal with filenames consistently across all platforms.
- It failed to have a unified model for file attributes.
- It was difficult to use it to traverse directories.
- It didn't allow the use of platform-specific features (such as symbolic links).

The older APIs also failed to address nonblocking operations for file systems. These limitations, and the need to provide support for modern approaches to I/O, led to the work that became NIO.2 in Java 7 (as specified in JSR 203: More New I/O APIs for the Java Platform "NIO.2"). This project had three major goals.

- A new file system interface
- An API for asynchronous (as opposed to polled, nonblocking) I/O operations on both sockets and files
- The completion of the socket-channel functionality (previously defined in JSR 51), including the addition of support for binding, option configuration, and multicast datagrams

The first goal further broke down into three subgoals.

- Full access to file attributes
- Ability to access file system–specific APIs
- A service provider interface (SPI) for pluggable file system implementations

These are recognizably the fundamental capabilities that the `Path` API delivers, and it's no surprise that the API has been enthusiastically adopted by developers. Despite this, however, it is sometimes necessary to interact with older code—or with relatively new code written by a developer who prefers the older file I/O methods. That's why familiarity with the other APIs that the JDK provides will pay dividends for the thoughtful programmer.

Fundamental to these older APIs is the `File` class—and it's there that I turn to next. (Remember: This is entirely different from the `Files` class of helper applications that work with `Path`.)

The old File class

The `File` class is the cornerstone of Java's original way to do file I/O. The abstraction represents metadata about a file system's location, that is, both files and directories.

The good news is that `File` provides an extensive set of methods for querying the file system metadata that the object represents, for example

```
// Permissions management
boolean canX = f.canExecute();
boolean canR = f.canRead();
boolean canW = f.canWrite();
```

```
boolean ok;
ok = f.setReadOnly();
ok = f.setExecutable(true);
ok = f.setReadable(true);
ok = f.setWritable(false);

// Different views of the file's name
File absF = f.getAbsoluteFile();
File canF = f.getCanonicalFile();
String absName = f.getAbsolutePath();
String canName = f.getCanonicalPath();
String name = f.getName();
String pName = getParent();
URI fileURI = f.toURI(); // Create URI for File path

// File metadata
boolean exists = f.exists();
boolean isAbs = f.isAbsolute();
boolean isDir = f.isDirectory();
boolean isFile = f.isFile();
boolean isHidden = f.isHidden();
long modTime = f.lastModified(); // millis since epoch
boolean updateOK = f.setLastModified(updateTime); // millis
long fileLen = f.length();

// File management operations
boolean renamed = f.renameTo(destFile);
boolean deleted = f.delete();

// Create won't overwrite existing file
boolean createdOK = f.createNewFile();

// Temporary file handling
File tmp = File.createTempFile("my-tmp", ".tmp");
tmp.deleteOnExit();

// Directory handling
boolean createdDir = dir.mkdir();
String[] fileNames = dir.list();
File[] files = dir.listFiles();
```

Copy code snippet

The bad news: Despite the large number of methods in the `File` class, some basic functionality is not, and never has been, provided directly. Famously, `File` does not provide a way to read the contents of a file directly.

Not only that but the way `File` abstracts over files and directories can be cumbersome to deal with, and leads to code like the following:

```
// Get a file object to represent the user's home directory
var homedir = new File(System.getProperty("user.home"));

// Create an object to represent a config file that is present in homedir
```

```
var f = new File(homedir, "app.conf");

// Check that the file exists, really is a file, and is readable
if (f.exists() && f.isFile() && f.canRead()) {

  // Create a file object for a new configuration directory
  var configdir = new File(f, ".configdir");

  configdir.mkdir();

  // Finally, move the config file to its new home
  f.renameTo(new File(configdir, ".config"));
}
```

Copy code snippet

This demonstrates some of the problems with the `File` abstraction, specifically that it is very general and requires a lot of work to interrogate a `File` object. In the example, to determine what a `File` object represents and its capabilities takes far too much code.

As noted above, by itself, the `File` class does not provide all needed functionality for I/O. Instead, it must be paired with the I/O stream abstraction (which is not to be confused with the Java streams that were introduced in Java 8 to facilitate a more functional approach to handling collections).

The I/O stream API was present in the very first version of Java as a way of dealing with sequential streams of bytes from disks or other sources.

The core of this API is a pair of abstract classes, `InputStream` and `OutputStream`. These are very widely used, and all subclasses must provide a method that returns or else returns the next byte of input. It should be no surprise that the standard input and output streams, `System.in` and `System.out` (which are Java's representations of `STDIN` and `STDOUT`), are streams of this type.

The major drawback of this API is that using the streams is often inconvenient and usually requires developers to use byte arrays and other low-level abstractions.

For example, to count all the times that lowercase *a* (ASCII value 97) appears in the book *Alice in Wonderland* requires code like the following:

```
try (var is = new FileInputStream("/Users/ben/alice.txt")) {
  var buf = new byte[4096];
  int len, count = 0;
  while ((len = is.read(buf)) > 0) {
    for (var i = 0; i < len; i = i + 1)
      if (buf[i] == 97) {
        count = count + 1;
      }
  }
  System.out.println("'a's seen: "+ count);
} catch (IOException e) {
  e.printStackTrace();
}
```

Copy code snippet

This takes you far too far down the byte-handling rabbit hole and uses some C-style tricks that should be left in the past. By contrast, the corresponding code using the `Path` API is far more declarative and flexible.

```
var alice = Path.of("/Users/ben/alice.txt");
try {
  long count = Files.lines(alice)
                    .flatMap(l -> Stream.of(l.split("")))
                    .filter(s -> s.equals("a"))
                    .count();
  System.out.println("'a's seen: "+ count);
} catch (IOException e) {
  e.printStackTrace();
}
```

Copy code snippet

It should now be no surprise that I can't recommend using the old, low-level APIs. Instead, the more modern APIs based on `Path` will be a far better fit most of the time. However, there are circumstances in which using a blend of the modern APIs and some of the older alternatives is required.

Bridging the File and Path APIs

Here's an example that shows the easy interoperation between `Path` and `File` objects. First, notice how the `Path` factory methods can be used to access the same path in different ways.

```
var p = Path.of("/Users/ben/cluster.txt");
var p2 = Path.of(new URI("file:///Users/ben/cluster.txt"));
System.out.println(p2.equals(p));
```

Copy code snippet

Unsurprisingly, this prints `true`. Let's now convert a path to a `File` object and use it normally.

```
File f = p.toFile();
System.out.println(f.isDirectory());
```

Copy code snippet

The addition of a `toPath()` method to `File` allows you to go in the other direction.

```
Path p3 = f.toPath();
System.out.println(p3.equals(p));
```

Copy code snippet

This pair of methods allows effortless motion between the two APIs and allows for easy refactoring of code based on `File` so it uses `Path` instead.

You can use other bridge methods in the new `Files` helper class to access the older I/O APIs such as `FileInputStream`.

```
var inputFile = new File("input");
try (var in = new FileInputStream(inputFile)) {
  Files.copy(in, Paths.get("output"));
} catch(IOException ex) {
  ex.printStackTrace();
}
```

Copy code snippet

There are also convenience factory methods for creating readers, writers, and file streams.

```
BufferedReader br = Files.newBufferedReader(target, cs);
BufferedWriter bwr = Files.newBufferedWriter(target, cs);

InputStream is = Files.newInputStream(target);
OutputStream os = Files.newOutputStream(target);
```

Copy code snippet

You can use these methods to write code like the following, which uses specified `Path` locations:

```
var logFile = Path.of("/tmp/my.log");
try (var writer =
       Files.newBufferedWriter(logFile, StandardCharsets.UTF_8,
                               StandardOpenOption.WRITE)) {
  writer.write("Hello World!");
  // ...
} catch (IOException e) {
```

```
    // ...
}
```

Copy code snippet

This provides a simple bridge to older code that uses the Readers and Writers API.

Extensibility with FileSystem and FileStore

The `Path` API was designed with extensibility in mind for future types of file I/O. For example, the `OpenOption` and `CopyOption` interfaces can be extended to provide additional options for opening and copying files or filelike entities. Regarding this extensibility, three of the most important types in the `java.nio.file` package are

- `Path`: The concept of a directory path on the file system
- `FileSystem`: Interfaces with file systems
- `FileStore`: Deals with the underlying devices, partitions, and so on

You have already seen how the fact that `Path` is an interface lends itself to extensibility.

These are the two other classes.

- `FileSystem` is essentially a factory for creating objects to access files and other objects in the file system, whether that file system is the default file system or an alternative file system retrieved by its URI. It has associated helper methods in `FileSystems` that create new file systems via a service provider interface (SPI) mechanism. A default `FileSystem` object is always available, and it corresponds to the local file system on the machine where the JVM was started.
- An instance of `FileStore` handles the low-level concrete details for how to interact with the bits and bytes of a `FileSystem`. This class is not often needed by the end user; instead, the higher-level abstractions suffice for almost all applications.

Conclusion

You should use the `Path` API for file I/O wherever possible. Remember that you can bridge the older APIs if necessary —and it's smart to be aware of the lower-level components and implementation details just in case.

Dig deeper

- Working and unit testing with temporary files in Java
- The Java NIO.2 file system in JDK 7
- Java documentation for Path

Ben Evans

Ben Evans (@kittylyst) is a Java Champion and Principal Engineer at New Relic. He has written five books on programming, including *Optimizing Java* (O'Reilly). Previously he was a founder of jClarity (acquired by Microsoft) and a member of the Java SE/EE Executive Committee.

Previous Post

Quiz yourself: Service types and service providers in Java modules

Mikalai Zaikin | 2 min read

Next Post

The art of long-term support and what LTS means for the Java ecosystem

Donald Smith | 4 min read