Trail: Essential Classes
Lesson: Basic I/O
Section: File I/O (Featuring NIO.2)

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

What Is a Path? (And Other File System Facts)

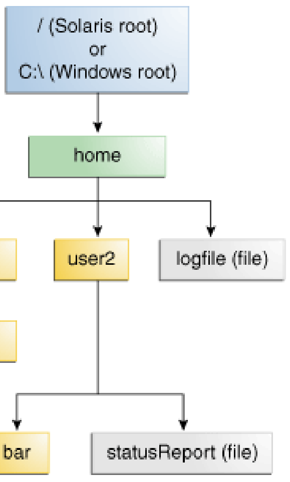
A file system stores and organizes files on some form of media, generally one or more hard drives, in such a way that they can be easily retrieved. Most file systems in use today store the files in a tree (or *hierarchical*) structure. At the top of the tree is one (or more) root nodes. Under the root node, there are files and directories (*folders* in Microsoft Windows). Each directory can contain files and subdirectories, which in turn can contain files and subdirectories, and so on, potentially to an almost limitless depth.

This section covers the following:

- What Is a Path?
- Relative or Absolute?
- Symbolic Links

What Is a Path?

The following figure shows a sample directory tree containing a single root node. Microsoft Windows supports multiple root nodes. Each root node maps to a volume, such as `C:\` or `D:\`. The Solaris OS supports a single root node, which is denoted by the slash character, `/`.



Sample Directory Structure

A file is identified by its path through the file system, beginning from the root node. For example, the `statusReport` file in the previous figure is described by the following notation in the Solaris OS:

```
/home/sally/statusReport
```

In Microsoft Windows, `statusReport` is described by the following notation:

```
C:\home\sally\statusReport
```

The character used to separate the directory names (also called the *delimiter*) is specific to the file system: The Solaris OS uses the forward slash (`/`), and Microsoft Windows uses the backslash slash (`\`).

Relative or Absolute?

A path is either *relative* or *absolute*. An absolute path always contains the root element and the complete directory list required to locate the file. For example, `/home/sally/statusReport` is an absolute path. All of the information needed to locate the file is contained in the path string.

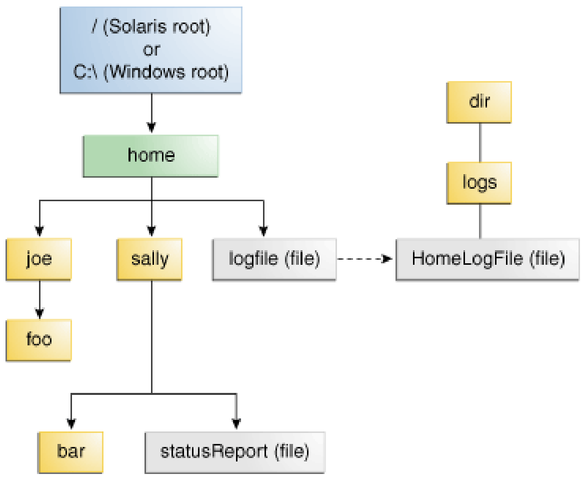
A relative path needs to be combined with another path in order to access a file. For example, `joe/foo` is a relative path. Without more information, a program cannot reliably locate the `joe/foo` directory in the file system.

Symbolic Links

File system objects are most typically directories or files. Everyone is familiar with these objects. But some file systems also support the notion of symbolic links. A symbolic link is also referred to as a *symlink* or a *soft link*.

A *symbolic link* is a special file that serves as a reference to another file. For the most part, symbolic links are transparent to applications, and operations on symbolic links are automatically redirected to the target of the link. (The file or directory being pointed to is called the *target* of the link.) Exceptions are when a symbolic link is deleted, or renamed in which case the link itself is deleted, or renamed and not the target of the link.

In the following figure, `logFile` appears to be a regular file to the user, but it is actually a symbolic link to `dir/logs/HomeLogFile`. `HomeLogFile` is the target of the link.



Example of a Symbolic Link.

A symbolic link is usually transparent to the user. Reading or writing to a symbolic link is the same as reading or writing to any other file or directory.

The phrase *resolving a link* means to substitute the actual location in the file system for the symbolic link. In the example, resolving `logFile` yields `dir/logs/HomeLogFile`.

In real-world scenarios, most file systems make liberal use of symbolic links. Occasionally, a carelessly created symbolic link can cause a circular reference. A circular reference occurs when the target of a link points back to the original link. The circular reference might be indirect: directory `a` points to directory `b`, which points to directory `c`, which contains a subdirectory pointing back to directory `a`. Circular references can cause havoc when a program is recursively walking a directory structure. However, this scenario has been accounted for and will not cause your program to loop infinitely.

The next page discusses the heart of file I/O support in the Java programming language, the `Path` class.

Previous page: File I/O (Featuring NIO.2)
Next page: The Path Class