Managing Files Attributes

File and File Store Attributes

A file system's metadata is typically referred to as its file attributes. The `Files` class includes methods that can be used to obtain a single attribute of a file, or to set an attribute.

Methods	Comments
`size(Path)`	Returns the size of the specified file in bytes.
`isDirectory(Path, LinkOption)`	Returns true if the specified `Path` locates a file that is a directory.
`isRegularFile(Path, LinkOption...)`	Returns true if the specified `Path` locates a file that is a regular file.
`isSymbolicLink(Path)`	Returns true if the specified `Path` locates a file that is a symbolic link.
`isHidden(Path)`	Returns true if the specified `Path` locates a file that is considered hidden by the file system.
`getLastModifiedTime(Path, LinkOption...)` `setLastModifiedTime(Path, FileTime)`	Returns or sets the specified file's last modified time.
`getOwner(Path, LinkOption...)` `setOwner(Path, UserPrincipal)`	Returns or sets the owner of the file.

Methods	Comments
`getPosixFilePermissions(Path, LinkOption...)` `setPosixFilePermissions(Path, Set<PosixFilePermission>)`	Returns or sets a file's POSIX file permissions.
`getAttribute(Path, String, LinkOption...)` `setAttribute(Path, String, Object, LinkOption...)`	Returns or sets the value of a file attribute.

If a program needs multiple file attributes around the same time, it can be inefficient to use methods that retrieve a single attribute. Repeatedly accessing the file system to retrieve a single attribute can adversely affect performance. For this reason, the `Files` class provides two `readAttributes()` methods to fetch a file's attributes in one bulk operation.

Methods	Comments
`readAttributes(Path, String, LinkOption...)`	Reads a file's attributes as a bulk operation. The `String` parameter identifies the attributes to be read.
`readAttributes(Path, Class<A>, LinkOption...)`	Reads a file's attributes as a bulk operation. The `Class<A>` parameter is the type of attributes requested and the method returns an object of that class.

Before showing examples of the `readAttributes()` methods, it should be mentioned that different file systems have different notions about which attributes should be tracked. For this reason, related file attributes are grouped together into views. A view maps to a particular file system implementation, such as POSIX or DOS, or to a common functionality, such as file ownership.

The supported views are as follows:

- `BasicFileAttributeView` – Provides a view of basic attributes that are required to be supported by all file system implementations.
- `DosFileAttributeView` – Extends the basic attribute view with the standard four bits supported on file systems that support the DOS attributes.
- `PosixFileAttributeView` – Extends the basic attribute view with attributes supported on file systems that support the POSIX family of standards, such as UNIX. These attributes include file owner, group owner, and the nine related access permissions.
- `FileOwnerAttributeView` – Supported by any file system implementation that supports the concept of a file owner.

- `AclFileAttributeView` – Supports reading or updating a file's Access Control Lists (ACL). The NFSv4 ACL model is supported. Any ACL model, such as the Windows ACL model, that has a well-defined mapping to the NFSv4 model might also be supported.
- `UserDefinedFileAttributeView` – Enables support of metadata that is user defined. This view can be mapped to any extension mechanisms that a system supports. In the Solaris OS, for example, you can use this view to store the MIME type of a file.

A specific file system implementation might support only the basic file attribute view, or it may support several of these file attribute views. A file system implementation might support other attribute views not included in this API.

In most instances, you should not have to deal directly with any of the `FileAttributeView` interfaces. (If you do need to work directly with the `FileAttributeView`, you can access it via the `getFileAttributeView(Path, Class<V>, LinkOption...)` method.)

The `readAttributes()` methods use generics and can be used to read the attributes for any of the file attributes views. The examples in the rest of this page use the `readAttributes()` methods.

Basic File Attributes

As mentioned previously, to read the basic attributes of a file, you can use one of the `Files.readAttributes()` methods, which reads all the basic attributes in one bulk operation. This is far more efficient than accessing the file system separately to read each individual attribute. The varargs argument currently supports the `LinkOption` enum, `NOFOLLOW_LINKS`. Use this option when you do not want symbolic links to be followed.

> *A word about time stamps: The set of basic attributes includes three time stamps:* `creationTime`, `lastModifiedTime`, *and* `lastAccessTime`. *Any of these time stamps might not be supported in a particular implementation, in which case the corresponding accessor method returns an implementation-specific value. When supported, the time stamp is returned as an* `FileTime` *object.*

The following code snippet reads and prints the basic file attributes for a given file and uses the methods in the `BasicFileAttributes` class.

```
Path file = ...;
BasicFileAttributes attr = Files.readAttributes(file, BasicFileAttributes.class);

System.out.println("creationTime: " + attr.creationTime());
System.out.println("lastAccessTime: " + attr.lastAccessTime());
```

```
System.out.println("lastModifiedTime: " + attr.lastModifiedTime());

System.out.println("isDirectory: " + attr.isDirectory());
System.out.println("isOther: " + attr.isOther());
System.out.println("isRegularFile: " + attr.isRegularFile());
System.out.println("isSymbolicLink: " + attr.isSymbolicLink());
System.out.println("size: " + attr.size());
```

In addition to the accessor methods shown in this example, there is a `fileKey()` method that returns either an object that uniquely identifies the file or `null` if no file key is available.

Setting Time Stamps

The following code snippet sets the last modified time in milliseconds:

```
Path file = ...;
BasicFileAttributes attr =
    Files.readAttributes(file, BasicFileAttributes.class);
long currentTime = System.currentTimeMillis();
FileTime ft = FileTime.fromMillis(currentTime);
Files.setLastModifiedTime(file, ft);
```

DOS File Attributes

DOS file attributes are also supported on file systems other than DOS, such as Samba. The following snippet uses the methods of the `DosFileAttributes` class.

```
Path file = ...;
try {
    DosFileAttributes attr =
        Files.readAttributes(file, DosFileAttributes.class);
    System.out.println("isReadOnly is " + attr.isReadOnly());
    System.out.println("isHidden is " + attr.isHidden());
    System.out.println("isArchive is " + attr.isArchive());
    System.out.println("isSystem is " + attr.isSystem());
} catch (UnsupportedOperationException x) {
    System.err.println("DOS file" +
        " attributes not supported:" + x);
}
```

However, you can set a DOS attribute using the `setAttribute(Path, String, Object, LinkOption...)` method, as follows:

```
Path file = ...;
Files.setAttribute(file, "dos:hidden", true);
```

POSIX File Permissions

POSIX is an acronym for Portable Operating System Interface for UNIX and is a set of IEEE and ISO standards designed to ensure interoperability among different flavors of UNIX. If a program conforms to these POSIX standards, it should be easily ported to other POSIX-compliant operating systems.

Besides file owner and group owner, POSIX supports nine file permissions: *read*, *write*, and *execute* permissions for the file owner, members of the same group, and "everyone else."

The following code snippet reads the POSIX file attributes for a given file and prints them to standard output. The code uses the methods in the `PosixFileAttributes` class.

```
Path file = ...;
PosixFileAttributes attr =
    Files.readAttributes(file, PosixFileAttributes.class);
System.out.format("%s %s %s%n",
    attr.owner().getName(),
    attr.group().getName(),
    PosixFilePermissions.toString(attr.permissions()));
```

The `PosixFilePermissions` helper class provides several useful methods, as follows:

- The `toString()` method, used in the previous code snippet, converts the file permissions to a string (for example, `rw-r--r--`).
- The `fromString()` method accepts a string representing the file permissions and constructs a `Set` of file permissions.
- The `asFileAttribute()` method accepts a `Set` of file permissions and constructs a file attribute that can be passed to the `Files.createFile()` or `Files.createDirectory()` method.

The following code snippet reads the attributes from one file and creates a new file, assigning the attributes from the original file to the new file:

```
Path sourceFile = ...;
Path newFile = ...;
PosixFileAttributes attrs =
    Files.readAttributes(sourceFile, PosixFileAttributes.class);
```

```
FileAttribute<Set<PosixFilePermission>> attr =
    PosixFilePermissions.asFileAttribute(attrs.permissions());
Files.createFile(file, attr);
```

The `asFileAttribute()` method wraps the permissions as a `FileAttribute`. The code then attempts to create a new file with those permissions. Note that the *umask* also applies, so the new file might be more secure than the permissions that were requested.

To set a file's permissions to values represented as a hard-coded string, you can use the following code:

```
Path file = ...;
Set<PosixFilePermission> perms =
    PosixFilePermissions.fromString("rw-------");
FileAttribute<Set<PosixFilePermission>> attr =
    PosixFilePermissions.asFileAttribute(perms);
Files.setPosixFilePermissions(file, perms);
```

Setting a File or Group Owner

To translate a name into an object you can store as a file owner or a group owner, you can use the `UserPrincipalLookupService` service. This service looks up a name or group name as a string and returns a `UserPrincipal` object representing that string. You can obtain the user principal look-up service for the default file system by using the `FileSystem.getUserPrincipalLookupService()` method.

The following code snippet shows how to set the file owner by using the `setOwner()` method:

```
Path file = ...;
UserPrincipal owner = file.getFileSystem().getUserPrincipalLookupService()
        .lookupPrincipalByName("sally");
Files.setOwner(file, owner);
```

There is no special-purpose method in the `Files` class for setting a group owner. However, a safe way to do so directly is through the POSIX file attribute view, as follows:

```
Path file = ...;
GroupPrincipal group =
    file.getFileSystem().getUserPrincipalLookupService()
        .lookupPrincipalByGroupName("green");
Files.getFileAttributeView(file, PosixFileAttributeView.class)
        .setGroup(group);
```

User-Defined File Attributes

If the file attributes supported by your file system implementation are not sufficient for your needs, you can use the `UserDefinedAttributeView` to create and track your own file attributes.

Some implementations map this concept to features like NTFS Alternative Data Streams and extended attributes on file systems such as ext3 and ZFS. Most implementations impose restrictions on the size of the value, for example, ext3 limits the size to 4 kilobytes.

A file's MIME type can be stored as a user-defined attribute by using this code snippet:

```
Path file = ...;
UserDefinedFileAttributeView view =
    Files.getFileAttributeView(file, UserDefinedFileAttributeView.class);
view.write("user.mimetype",
           Charset.defaultCharset().encode("text/html");
```

To read the MIME type attribute, you would use this code snippet:

```
Path file = ...;
UserDefinedFileAttributeView view =
    Files.getFileAttributeView(file,UserDefinedFileAttributeView.class);
String name = "user.mimetype";
ByteBuffer buf = ByteBuffer.allocate(view.size(name));
view.read(name, buf);
buf.flip();
String value = Charset.defaultCharset().decode(buf).toString();
```

Note: In Linux, you might have to enable extended attributes for user-defined attributes to work. If you receive an `UnsupportedOperationException` when trying to access the user-defined attribute view, you need to remount the file system. The following command remounts the root partition with extended attributes for the ext3 file system. If this command does not work for your flavor of Linux, consult the documentation.

```
$ sudo mount -o remount,user_xattr /
```

If you want to make the change permanent, add an entry to `/etc/fstab`.

File Store Attributes

You can use the `FileStore` class to learn information about a file store, such as how much space is available. The `getFileStore(Path)` method fetches the file store for the specified file.

The following code snippet prints the space usage for the file store where a particular file resides:

```
Path file = ...;
FileStore store = Files.getFileStore(file);

long total = store.getTotalSpace() / 1024;
long used = (store.getTotalSpace() -
             store.getUnallocatedSpace()) / 1024;
long avail = store.getUsableSpace() / 1024;
```

Determining MIME Type

To determine the MIME type of a file, you might find the `probeContentType(Path)` method useful. For example:

```
try {
    String type = Files.probeContentType(filename);
    if (type == null) {
        System.err.format("'%s' has an" + " unknown filetype.%n", filename);
    } else if (!type.equals("text/plain") {
        System.err.format("'%s' is not" + " a plain text file.%n", filename);
        continue;
    }
} catch (IOException x) {
    System.err.println(x);
}
```

Note that this method returns null if the content type cannot be determined.

The implementation of this method is highly platform specific and is not infallible. The content type is determined by the platform's default file type detector. For example, if the detector determines a file's content type to be `application/x-java` based on the `.class` extension, it might be fooled.

You can provide a custom `FileTypeDetector` if the default is not sufficient for your needs.

Last update: *January 25, 2023*