21.6 Managing File Attributes

Useful metadata is associated with directory entries in a file system—for example, the file permissions that indicate whether the entry is readable or writable, or whether it is a symbolic link, and its size. Such metadata in the file system is often referred to as *file attributes*. Managing file attributes is a separate concern from the data that is stored in files.

There are basically two approaches provided by the NIO.2 API for managing file attributes:

- Accessing *individual file attributes* associated with a directory entry in the file system (p. 1321)
- Accessing a *set of file attributes* associated with a directory entry in the file system as a *bulk operation* (p. 1328)

Accessing Individual File Attributes

The `Files` class provides a myriad of static methods to access individual file attributes of a directory entry. It is a good idea to consult the code in Example 21.4 as we take a closer look at the relevant methods in this subsection. Since methods in the `Files` class can throw an `IOException`, the `main()` method specifies a `throws` clause with this exception.

Example 21.4 *Accessing Individual Attributes*

```
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.util.*;

import static java.lang.System.out;
import static java.nio.file.attribute.PosixFilePermission.*;

public class IndividualFileAttributes {

  public static void main(String[] args) throws IOException {

    Path fPath = Path.of("project", "src", "pkg", "Main.java");
    out.println("File: " + fPath);

    out.println("Accessing Individual File Attributes:");
    out.println("size file (bytes): " + Files.size(fPath));               // (1)
    out.println("isDirectory:       " + Files.isDirectory(fPath));       // (2)
    out.println("isRegularFile:     " + Files.isRegularFile(fPath));     // (3)
    out.println("isSymbolicLink:    " + Files.isSymbolicLink(fPath));    // (4)
    out.println();

    out.println("isReadable:        " + Files.isReadable(fPath));        // (5)
    out.println("isWritable:        " + Files.isWritable(fPath));        // (6)
```

```
        out.println("isExecutable:      " + Files.isExecutable(fPath));        // (7)
        out.println("isHidden:          " + Files.isHidden(fPath));            // (8)
        out.println();

        out.println("getLastModifiedTime: " + Files.getLastModifiedTime(fPath));// (9)
        out.println("getOwner:            " + Files.getOwner(fPath));          // (10)
        out.println();

        // Get the POSIX file permissions for the directory entry:
        Set<PosixFilePermission> filePermissions
            = Files.getPosixFilePermissions(fPath);                            // (11)
        out.println("getPosixFilePermissions (set): " + filePermissions);      // (12)
        out.println("getPosixFilePermissions (string): "
                  + PosixFilePermissions.toString(filePermissions));           // (13)

        // Get the group of the directory entry:
        out.println("getAttribute-group:  " + Files.getAttribute(fPath,        // (14)
                                                                 "posix:group"));
        out.println();

        // Update last modified time for the directory entry.                     (15)
        long currentTime = System.currentTimeMillis();
        FileTime timestamp = FileTime.fromMillis(currentTime);
        Files.setLastModifiedTime(fPath, timestamp);

        // Set new owner for the directory entry.                                 (16)
        FileSystem fs = fPath.getFileSystem();   // File system that created the path.
        UserPrincipalLookupService upls
            = fs.getUserPrincipalLookupService();// Obtain service to look up user.
        UserPrincipal user = upls.lookupPrincipalByName("khalid"); // User lookup.
        Files.setOwner(fPath, user);                                // Set user.

        // Set POSIX file permissions for the directory entry:                    (17)
        Set<PosixFilePermission> newfilePermissions
            = EnumSet.of(OWNER_READ, OWNER_WRITE, GROUP_READ, GROUP_WRITE);   // (18a)
        //Set<PosixFilePermission> newfilePermissions
        //  = PosixFilePermissions.fromString("rw-rw----");                   // (18b)
        Files.setPosixFilePermissions(fPath, newfilePermissions);             // (19)
        filePermissions = Files.getPosixFilePermissions(fPath);
        out.println("getPosixFilePermissions (set): " + filePermissions);
        out.println("getPosixFilePermissions (string): "
              + PosixFilePermissions.toString(filePermissions));

        // Setting the value of a file attribute by its attribute name.
        Files.setAttribute(fPath, "lastAccessTime", timestamp);               // (20)
    }
}
```

Possible output from the program:

```
File: project/src/pkg/Main.java
Accessing Individual File Attributes:
size file (bytes): 13
```

```
isDirectory:       false
isRegularFile:     true
isSymbolicLink:    false

isReadable:        true
isWritable:        true
isExecutable:      false
isHidden:          false

getLastModifiedTime: 2021-08-06T10:28:47.416033Z
getOwner:            khalid

getPosixFilePermissions (set): [OTHERS_READ, OWNER_WRITE, OWNER_READ, GROUP_READ]
getPosixFilePermissions (string): rw-r--r--
getAttribute-group:  admin

getPosixFilePermissions (set): [GROUP_WRITE, OWNER_WRITE, OWNER_READ, GROUP_READ]
getPosixFilePermissions (string): rw-rw----
```

Determining the File Size

The method `size()` in the `Files` class is called at (1) in Example 21.4 to determine the size of the file denoted by the `Path` object. If the `Path` object denotes a directory, the method returns the size of the directory file and *not* the size of entries in the directory.

```
static long size(Path path) throws IOException
```

Returns the size of a file (in bytes). The size of files that are not regular files is unspecified, as it is implementation specific.

Determining the Kind of Directory Entry

The following methods in the `Files` class are called at (2), (3), and (4) in Example 21.4 to determine what kind of directory entry is denoted by the `Path` object.

```
static boolean isDirectory(Path path, LinkOption... options)
static boolean isRegularFile(Path path, LinkOption... options)
static boolean isSymbolicLink(Path path)
```

Return `true` if the directory entry is a directory, a regular file, or a symbolic link, respectively. They return `false` if the directory entry does not exist, or is not of the expected kind, or it is not possible to determine what kind of directory entry it is. In the first two methods, symbolic links are followed by default, unless the constant `LinkOption.NOFOLLOW_LINKS` is specified.

Determining File Accessibility

The methods in the `Files` class shown below are called at (5), (6), (7), and (8) in Example 21.4, respectively, to determine accessibility of the directory entry denoted by the `Path` object.

```
static boolean isReadable(Path path)
static boolean isWritable(Path path)
static boolean isExecutable(Path path)
```

Test whether a file is readable, writable, or executable, respectively. The file must exist and the JVM must have the appropriate privileges to access the file.

Note that the result returned by these method is immediately outdated. The outcome of subsequent attempts to access the file is not guaranteed, as concurrently running threads might change the conditions after the method returns.

```
static boolean isHidden(Path path) throws IOException
```

Determines whether or not a file is considered *hidden*. The exact definition of a hidden file is platform specific. On Unix platforms, files whose name begins with a period character (`.`) are considered to be hidden.

Timestamp for Last Modification Time

Three different timestamps are associated with a directory entry, whose purpose is evident from their names: *last modified time*, *last access time*, and *creation time*. The timestamps are represented by the `java.nio.file.attribute.FileTime` class that provides the following methods for interoperability with `Instant` objects and with `long` values in milliseconds.

```
class java.nio.file.attribute.FileTime

static FileTime from(Instant instant)
static FileTime fromMillis(long value)
```

These static methods create a `FileTime` object representing the same point of time value on the timeline as the specified `Instant` object, or a `FileTime` object from the `long value` that specifies the number of milliseconds since the epoch (`1970-01-01T00:00:00Z`), respectively.

```
Instant toInstant()
long toMillis()
```

These instance methods convert this `FileTime` object to an `Instant` or to a `long` value in milliseconds from the epoch, respectively.

The `Files` class only provides static methods to read and update the last modified time of a directory entry. In Example 21.4, the statement at (9) prints the last modified time of the directory entry. The code at (15) sets the last modified time of the directory entry to the current time.

```
static FileTime getLastModifiedTime(Path path, LinkOption... options)
                                    throws IOException
static Path     setLastModifiedTime(Path path, FileTime time)
                                    throws IOException
```

Return or update the timestamp for the last modified time attribute of a directory entry, respectively. The timestamp is represented by the class `java.nio.file.attribute.FileTime`.

The first method follows symbolic links by default, unless the constant `LinkOption.NOFOLLOW_LINKS` is specified.

Accessing the Owner

The `Files` class only provides static methods to get and set the *owner* of a directory entry (i.e., one with a user account and appropriate access permissions). In Example 21.4, the statement at (10) prints the name of the owner of the directory entry. The code at (16) executes the necessary steps to obtain a user that can be set as the owner of the directory entry. This involves querying the file system to obtain the user look service and using the service to look up the user by name. We leave it to the reader to discover the exciting details from the API of the classes involved.

```
static UserPrincipal getOwner(Path path, LinkOption... options)
static Path setOwner(Path path, UserPrincipal owner)
```

Return or update the owner of a file, respectively.

Handling File Permissions

For a directory entry, POSIX-based file systems (*Portable Operating System Interface*) typically define *read*, *write*, and *execute* permissions for the *owner*, the *group* that the owner belongs to, and for *others*. In Java, these nine permissions are represented by the enum type `PosixFilePermission` (Table 21.10).

A human-readable form of file permissions affords interoperability with the enum type `PosixFilePermission`. This form is specified as a string of nine characters, where characters are interpreted as three permission groups of three characters. From the start of the string, the first permission group, the second permission group, and the third permission group specify the permissions for the owner, the group, and others, respectively. Each permission group is defined by the following pattern:

```
(r|-)(w|-)(x|-)
```

that is comprised of three groupings, where each grouping `(`*a*`|`*b*`)` is interpreted as either *a* or *b*. For example, `rwx` and `---` are valid permissions groups, but `w_w` and `xwr` are not. The characters `r`, `w`, and `x` stand for read, write, and execute permissions, respectively, and the character `-` indicates that the permission corresponding to the position of the character `-` is not set.

The set of file permissions created by the following statement:

```
Set<PosixFilePermission> permSet1
      = EnumSet.of(OWNER_READ, OWNER_WRITE, GROUP_READ, OTHERS_READ);
```

is equivalent to the permissions in the string `"rw-r--r--"`.

The utility class `PosixFilePermissions` provides methods for converting between the two forms of specifying file permissions.

Table 21.10 *POSIX File Permissions*

Enum type `java.nio.file.attribute.PosixFilePermission`	Description
`OWNER_EXECUTE`	Execute/search permission, owner
`OWNER_READ`	Read permission, owner
`OWNER_WRITE`	Write permission, owner
`GROUP_EXECUTE`	Execute/search permission, group
`GROUP_READ`	Read permission, group
`GROUP_WRITE`	Write permission, group
`OTHERS_EXECUTE`	Execute/search permission, others
`OTHERS_READ`	Read permission, others
`OTHERS_WRITE`	Write permission, others

Following are methods from the utility class `java.nio.file.attribute.PosixFilePermissions`:

```
static Set<PosixFilePermission> fromString(String permStr)
```

Returns the set of permissions corresponding to a given `String` representation. The `permStr` parameter is a `String` representing the permissions, as explained earlier.

```
static String toString(Set<PosixFilePermission> perms)
```

Returns the `String` representation of a set of permissions.

```
static FileAttribute<Set<PosixFilePermission>>
       asFileAttribute(Set<PosixFilePermission> perms)
```

Creates a `FileAttribute`, encapsulating a copy of the given file permissions, suitable for passing to methods that create files and directories (p. 1339).

The `getPosixFilePermissions()` and `setPosixFilePermissions()` methods of the `Files` class can be used to retrieve and update file permissions of a directory entry, as shown at (11) and (19), respectively. The methods `toString()` and `fromString()` of the `PosixFilePermissions` class at (13) and (18b) convert between a set of `PosixFilePermission` and a string representation of file permissions, respectively. Note that (18a) and (18b) define the same set of file permissions.

```
// Get the POSIX file permissions for the directory entry:
Set<PosixFilePermission> filePermissions
    = Files.getPosixFilePermissions(fPath);                              // (11)
out.println("getPosixFilePermissions (set):    " + filePermissions);    // (12)
out.println("getPosixFilePermissions (string): "
          + PosixFilePermissions.toString(filePermissions));             // (13)
...
// Set POSIX file permissions for the directory entry:                      (17)
Set<PosixFilePermission> newfilePermissions
        = EnumSet.of(OWNER_READ, OWNER_WRITE, GROUP_READ, OTHERS_READ);// (18a)
//Set<PosixFilePermission> newfilePermissions
//       = PosixFilePermissions.fromString("rw-r--r--");                 // (18b)
Files.setPosixFilePermissions(fPath, newfilePermissions);                // (19)
```

The following methods from the utility class `java.nio.file.Files` can be used for retrieving and updating the POSIX-specific file permissions of a directory entry:

```
static Set<PosixFilePermission>
       getPosixFilePermissions(Path path, LinkOption... options)
                               throws IOException
```

Returns POSIX permissions of a directory entry as a set of enum type `PosixFilePermission`. By default, symbolic links are followed, unless the constant `LinkOption.NOFOLLOW_LINKS` is specified.

```
static Path setPosixFilePermissions(Path path,
                                    Set<PosixFilePermission> perms)
                                    throws IOException
```

Sets the POSIX permissions of a directory entry, given by the parameter `perms`.

Accessing File Attributes through View and Attribute Names

The `getAttribute()` and `setAttribute()` methods of the `Files` class are general methods that can be used to read and update any file attribute by its name. These methods are useful when the `Files` class does not provide a specialized method for a particular file attribute.

The statement at (14) in Example 21.4 prints the group of the directory entry. The full attribute name of the group attribute is specified as `"posix:group"`, as the group attribute can be accessed via the POSIX file attribute view (p. 1336).

The statement at (20) in Example 21.4 sets the last access time of the directory entry to the current time. The attribute named `"lastAccessTime"` can be accessed via the basic file attribute view that is implied by default (p. 1334). The value of this file attribute is the `FileTime` object denoted by the `timestamp` reference.

```
static Object getAttribute(Path path, String attribute,
                           LinkOption... options) throws IOException
static Path   setAttribute(Path path, String attribute, Object value,
                           LinkOption... options) throws IOException
```

Read and set, respectively, the value of a file `attribute` of a directory entry denoted by the given `Path` object. By default, symbolic links are followed, unless the constant `LinkOption.NOFOLLOW_LINKS` is specified.

The `attribute` parameter has the general format:

`view-name:attribute-name`

where the *`view-name`* can be omitted. Typical view names are `"basic"`, `"posix"`, and `"dos"`. Omitting the *`view-name`* defaults to `"basic"`. The *`attribute-name`* is the name of the file attribute—for example, `"lastModifiedTime"` or `"lastAccessTime"`. Attribute views are discussed in detail later (p. 1328).

The second method sets the file `attribute` to the `value` parameter.

Bulk Operations to Retrieve File Attributes

Accessing file attributes individually raises two concerns:

- Accessing an individual file attribute incurs a cost, and frequent such accesses can adversely impact performance.
- File attributes are very much file-system-specific, and therefore not conducive to generalizing over different file systems.

To address these concerns, the NIO.2 API provides *bulk operations* to retrieve file attributes—thus avoiding retrieval of individual file attributes and only handling file-system-specific attributes.

The following three approaches can be used to access file attributes of a directory entry in a file system:

- The methods of the `Files` class can be used to access individual file attributes of a directory entry denoted by a `Path`, as discussed earlier (p. 1321).
- A *bulk operation* using the `readAttributes()` method of the `Files` class can be used to retrieve a *set of file attributes* into a *read-only object* that implements *a file attributes interface*. This read-only object acts as a repository for file attribute values pertaining to the directory entry whose `Path` object was specified in the call to the `readAttributes()` method. The interface `BasicFileAttributes`, and its subinterfaces `PosixFileAttributes` and `DosFileAttributes`, define methods to read the retrieved file attribute values (Table 21.11, p. 1330).

  In the code below at (1), the set of file attributes to read for the directory entry denoted by the `path` parameter is determined by the runtime object `BasicFileAttributes.class`.

```
BasicFileAttributes bfa = Files.readAttributes(path,                  // (1)
                                               BasicFileAttributes.class);
```

The retrieved values of the file attributes are accessible by querying the `BasicFileAttributes` object that is returned by the `readAttributes()` method (p. 1330).

- A *bulk operation* using the `getFileAttributeView()` method of the `Files` class can be used to retrieve *a set of file attributes* into an *updatable file attribute view* that acts as a repository object. This object can be used to read or update selected file attributes pertaining to the directory entry whose `Path` object was specified in the call to the `getFileAttributeView()` method. The interface `BasicFileAttributeView`, and its subinterfaces `PosixFileAttributeView` and `DosFileAttributeView`, define methods to read and update the retrieved file attributes (Table 21.12, p. 1334).

  In the code below at (6), the set of file attributes to retrieve for the directory entry denoted by the `path` parameter is determined by the runtime object `BasicFileAttributeView.class`.

```
BasicFileAttributeView bfaView = Files.getFileAttributeView(path,      // (6)
                                                 BasicFileAttributeView.class);
```

  The retrieved file attributes can be read and updated by querying the `BasicFileAttributeView` object that is returned by the `getFileAttributeView()` method (p. 1334).

Details of the API for the `readAttributes()` and the `getFileAttributeView()` methods in the `Files` class are given below. By default, these methods follow symbolic links, unless the constant `LinkOption.NOFOLLOW_LINKS` is specified.

```
static <A extends BasicFileAttributes> A
       readAttributes(Path path, Class<A> type, LinkOption... options)
       throws IOException
```

Reads a *set of read-only file attributes* as a *bulk operation* for the directory entry denoted by the `path` parameter. The file attributes to retrieve are determined by the type parameter `A`. The parameter `type` is the `Class<A>` of the file attributes required to retrieve. The type parameter `A` is typically the interface `BasicFileAttributes`, or one of its subinterfaces `PosixFileAttributes` or `DosFileAttributes` (Table 21.11, p. 1330).

```
static <V extends FileAttributeView> V
       getFileAttributeView(Path path, Class<V> type, LinkOption... options)
```

Reads a *set of file attributes* of a file as a *bulk operation*. It returns a *file attribute view* of a given type, which can be used to *read or update* the retrieved file attribute values. The file attributes to retrieve are determined by the type parameter `V`. The parameter `type` is the `Class<V>` of the file attributes required to retrieve. The type parameter `V` is typically the interface `BasicFileAttributeView`, or one of its subinterfaces `PosixFileAttributeView` or `DosFileAttributeView`—that are all subinterfaces of the `FileAttributeView` interface (Table 21.12, p. 1334).

```
static Map<String,Object> readAttributes(Path path, String attributes,
                                         LinkOption... options)
                                         throws IOException
```

Reads a *set of file attributes* as a *bulk operation* for the directory entry denoted by the `path` parameter. It returns a map of file attribute names and their values.

The `attributes` parameter of type `String` has the general format:

`view-name:attribute-list`

where the *`view-name`* can be omitted, and *`attribute-list`* is a comma-separated list of attribute names. Omitting the *`view-name`* defaults to `"basic"`. For example, `"*"` will read all `BasicFileAttributes`, and `"lastModifiedTime,lastAccessTime"` will read only last modified time and last access time attributes.

Table 21.11 summarizes the interfaces that provide read-only access to file attribute values. The `BasicFileAttributes` interface defines the basic set of file attributes that are common to many file systems. Its two subinterfaces `PosixFileAttributes` and `DosFileAttributes` define *additional* file attributes associated with POSIX-based and DOS-based file systems, respectively. An object of the appropriate file attributes interface pertaining to a specific directory entry is returned by the `Files.readAttributes()` method.

Table 21.11 *Interfaces for Read-Only Access to File Attributes*

Read-only file attributes interfaces in the `java.nio.file.attribute` package.	**Note that when an object of the read-only file attributes interface is created, it is opened on a specific directory entry in the file system. The object provides information about file attributes associated with this directory entry. The methods in these interfaces do not throw a checked exception.**
`BasicFileAttributes`	Provides *read-only* access to a basic set of file attributes that are common to many file systems (p. 1330).
`PosixFileAttributes extends BasicFileAttributes`	In addition to the basic set of file attributes, provides *read-only* access to file attributes associated with POSIX-based file systems (p. 1332).
`DosFileAttributes extends BasicFileAttributes`	In addition to the basic set of file attributes, provides *read-only* access to file attributes associated with DOS/Windows-based file systems (p. 1333).

The `BasicFileAttributes` Interface

The methods of the `java.nio.file.attribute.BasicFileAttributes` interface reflect the basic set of file attributes that are common to most file systems.

```
interface java.nio.file.attribute.BasicFileAttributes
```

```
long size()
```

Returns the size of the directory entry (in bytes).

```
boolean isDirectory()
boolean isRegularFile()
boolean isSymbolicLink()
boolean isOther()
```

Determine whether the directory entry is of a specific kind.

```
FileTime lastModifiedTime()
FileTime lastAccessTime()
FileTime creationTime()
```

Return the appropriate timestamp for the directory entry.

The method `printBasicFileAttributes()` of the utility class `FileUtils`, shown below, prints the values of the basic file attributes by calling the relevant methods on the `BasicFileAttributes` object that is passed as a parameter.

```
// Declared in utility class FileUtils.
public static void printBasicFileAttributes(BasicFileAttributes bfa) {
  out.println("Printing basic file attributes:");
  out.println("lastModifiedTime: " + bfa.lastModifiedTime());
  out.println("lastAccessTime:   " + bfa.lastAccessTime());
  out.println("creationTime:     " + bfa.creationTime());

  out.println("size:             " + bfa.size());
  out.println("isDirectory:      " + bfa.isDirectory());
  out.println("isRegularFile:    " + bfa.isRegularFile());
  out.println("isSymbolicLink:   " + bfa.isSymbolicLink());
  out.println("isOther:          " + bfa.isOther());
  out.println();
}
```

The code below obtains a `BasicFileAttributes` object at (1) that pertains to the file denoted by the `path` reference. The `printBasicFileAttributes()` method is called at (2) with this `BasicFileAttributes` object as the parameter.

```
Path path = Path.of("project", "src", "pkg", "Main.java");
out.println("File: " + path);
BasicFileAttributes bfa = Files.readAttributes(path,                    // (1)
                                               BasicFileAttributes.class);
FileUtils.printBasicFileAttributes(bfa);                                // (2)
```

Possible output from the code:

```
File: project/src/pkg/Main.java
Printing basic file attributes:
lastModifiedTime: 2021-07-23T10:15:34.854Z
lastAccessTime:   2021-07-23T10:16:33.166281Z
creationTime:     2021-07-20T23:03:58Z
size:             116
isDirectory:      false
```

```
isRegularFile:    true
isSymbolicLink:   false
isOther:          false
```

Note that the basic file attributes in the `BasicFileAttributes` object are read-only, as the `BasicFileAttributes` interface does not provide any set methods. However, values of updatable file attributes can be changed by appropriate set methods of the `Files` class (p. 1321).

The `PosixFileAttributes` Interface

As the `PosixFileAttributes` interface is a subinterface of the `BasicFileAttributes` interface, a `PosixFileAttributes` object has both the basic set of file attributes and the POSIX-specific file attributes.

```
interface java.nio.file.attribute.PosixFileAttributes
          extends BasicFileAttributes
```

```
UserPrincipal owner()
GroupPrincipal group()
```

Return the owner or the group of the directory entry, represented by the interface `java.nio.file.attribute.UserPrincipal` and its subinterface `java.nio.file.attribute.GroupPrincipal`, respectively.

```
Set<PosixFilePermission> permissions()
```

Returns a set with a copy of the POSIX permissions for the directory entry. Permissions are defined by the enum type `PosixFilePermission` discussed earlier in this chapter (Table 21.10, p. 1326).

The methods of the subinterface `PosixFileAttributes` augment the basic set of file attributes with the following POSIX-specific attributes: owner, group, and file permissions. The method `printPosixFileAttributes()` in the utility class `FileUtils`, shown below, prints the values of the POSIX-specific file attributes by calling the relevant methods on the `PosixFileAttributes` object that is passed as a parameter.

```
// Declared in the utility class FileUtils.
public static void printPosixFileAttributes(PosixFileAttributes pfa) {
  out.println("Printing POSIX-specific file attributes:");
  UserPrincipal user = pfa.owner();
  GroupPrincipal group = pfa.group();
  Set<PosixFilePermission> permissions = pfa.permissions();
  String perms = PosixFilePermissions.toString(permissions);

  out.println("owner:            " + user);
  out.println("group:            " + group);
  out.println("permissions:      " + perms);
  out.println();
}
```

The code below obtains a `PosixFileAttributes` object at (3) that pertains to the file denoted by the `path` reference. Both the `printBasicFileAttributes()` method and the `printPosixFileAttributes()` method are called at (4) and (5), respectively, with

this PosixFileAttributes object as the parameter. The call to the printBasicFileAttributes() method at (4) will print the basic file attributes in the PosixFileAttributes object.

```
Path path = Path.of("project", "src", "pkg", "Main.java");
out.println("File: " + path);
PosixFileAttributes pfa = Files.readAttributes(path,                    // (3)
                                               PosixFileAttributes.class);
FileUtils.printBasicFileAttributes(pfa);                                // (4)
FileUtils.printPosixFileAttributes(pfa);                                // (5)
```

Possible output from the code:

```
File: project/src/pkg/Main.java
Printing basic file attributes:
...
Printing POSIX-specific file attributes:
owner:            javadude
group:            admin
permissions:      rw-r--r--
```

Note that both the basic and the POSIX-specific file attributes in the PosixFileAttributes object are read-only, as the PosixFileAttributes interface does not provide any set methods. However, values of updatable file attributes can be changed by appropriate set methods of the Files class (p. 1321).

The *DosFileAttributes* Interface

As the DosFileAttributes interface is a subinterface of the BasicFileAttributes interface, a DosFileAttributes object has both the basic set of file attributes and the DOS-specific file attributes. Its usage is analogous to the PosixFileAttributes interface discussed earlier (p. 1332).

```
interface java.nio.file.attribute.DosFileAttributes
          extends BasicFileAttributes

boolean isReadOnly()
boolean isSystem()
boolean isArchive()
boolean isHidden()
```

Determine whether the file entry is of a specific kind.

File Attribute Views

Table 21.12 summarizes the *file attribute views* that different interfaces provide for *readable or updatable* access to file attributes, in contrast to the file attributes interfaces in Table 21.11, p. 1330, that allow read-only access. The BasicFileAttributeView interface allows access to the basic set of file attributes that are common to many file systems. Its two subinterfaces, PosixFileAttributeView and DosFileAttributeView, additionally allow access to file attributes associated with POSIX-based and DOS-based file systems, respectively.

An object of the appropriate view interface pertaining to a specific directory entry is returned by the `Files.getFileAttributeView()` method. All file interface views provide a `readAttributes()` method that returns the read-only file attributes object associated with the view.

Table 21.12 *Selected File Attribute Views*

Updatable file attribute view interfaces in the `java.nio.file.attribute` package.	**Note that when the view is created, it is opened on a specific directory entry in the file system. The view provides information about file attributes associated with this directory entry.**
`AttributeView`	Can read or update non-opaque values associated with directory entries in a file system.
`FileAttributeView` `extends AttributeView`	Can read or update file attributes.
`FileOwnerAttributeView` `extends FileAttributeView`	Can read or update the owner.

`BasicFileAttributeView`
`extends FileAttributeView`
 	
Corresponding read-only file attributes interface:
`BasicFileAttributes`
	Can read or update a basic set of file attributes.
	Can obtain a read-only `BasicFileAttributes` object via the view.
	Can set a timestamp for when the directory entry was last modified, last accessed, and created.
	(p. 1334)

`PosixFileAttributeView`
`extends BasicFileAttributeView,`
`FileOwnerAttributeView`
 	
Corresponding read-only file attributes interface:
`PosixFileAttributes`
	Can read or update POSIX file attributes.
	Can obtain a read-only `PosixFileAttributes` object via the view.
	Can set group and file permissions, and update the owner.
	(p. 1336)

`DosFileAttributeView`
`extends BasicFileAttributeView`
 	
Corresponding read-only file attributes interface:
`DosFileAttributes`
	Can read or update DOS file attributes.
	Can obtain a read-only `DosFileAttributes` object via the view.
	Can set archive, hidden, read-only, and system attributes.
	(p. 1338)

The `BasicFileAttributeView` Interface

The `java.nio.file.attribute.BasicFileAttributeView` interface defines a file attribute view for the basic set of file attributes.

```
interface BasicFileAttributeView extends FileAttributeView
```

`String name()`

Returns the name of the attribute view, which in this case is the string `"basic"`.

`BasicFileAttributes readAttributes()`

Reads the basic file attributes as a bulk operation. The `BasicFileAttributes` object can be used to read the values of the basic file attributes (p. 1330). This method is analogous to the `readAttributes()` method of the `Files` class (p. 1328).

```
void setTimes(FileTime lastModifiedTime,
              FileTime lastAccessTime,
              FileTime createTime)
```

Updates any or all timestamps for the file's last modified time, last access time, and creation time attributes. If any parameter has the value `null`, the corresponding timestamp is not changed. Note that apart from the `Files.setLastModified()` method, there are no methods in the `Files` class for the last access and creation times for a directory entry.

The code below obtains a `BasicFileAttributeView` object at (6) that pertains to the file denoted by the `path` reference. A `BasicFileAttributes` object is obtained at (7), providing read-only access to the basic file attributes, whose values are printed by calling the `printBasicFileAttributes()` method. The last modified time of the directory entry is explicitly read by calling the `lastModifiedTime()` method of the `BasicFileAttributes` object.

```
Path path = Path.of("project", "src", "pkg", "Main.java");
out.println("File: " + path);

BasicFileAttributeView bfaView = Files.getFileAttributeView(path,      // (6)
                                          BasicFileAttributeView.class);
System.out.printf("Using view: %s%n", bfaView.name());

// Reading the basic set of file attributes:                            (7)
BasicFileAttributes bfa2 = bfaView.readAttributes();
FileUtils.printBasicFileAttributes(bfa2);
FileTime currentLastModifiedTime = bfa2.lastModifiedTime();

// Updating timestamp for last modified time using view:                (8)
long newLMTinMillis = currentLastModifiedTime.toMillis() + 15*60*1000L;
FileTime newLastModifiedTime = FileTime.fromMillis(newLMTinMillis);
bfaView.setTimes(newLastModifiedTime, null, null);

// Reading the updated last modified time:                              (9)
out.println("updated lastModifiedTime (incorrect): "
                                       + bfa2.lastModifiedTime());   // (10)
out.println("updated lastModifiedTime: "
                                  + Files.getLastModifiedTime(path)); // (11)
out.println("updated lastModifiedTime: " + Files.getAttribute(path,    // (12)
                                                  "basic:lastModifiedTime"));
```

Possible output from the code:

```
File: project/src/pkg/Main.java
Using view: basic
Printing basic file attributes:
...
lastModifiedTime: 2021-07-26T15:15:46.813Z
...
updated lastModifiedTime (incorrect): 2021-07-26T15:15:46.813Z
updated lastModifiedTime: 2021-07-26T15:30:46.813Z
updated lastModifiedTime: 2021-07-26T15:30:46.813Z
```

The `BasicFileAttributeView` object allows the last modified, last access, and creation times of the directory entry to be updated by calling the `setTimes()` method. The code at (9) shows how the last modified time of the directory entry can be updated to a new value via the view. A `FileTime` object is created representing the new last modified time by first converting the current last modified time to milliseconds and incrementing it by 15 minutes. In the call to the `setTimes()` method, only the last modified time is specified. The other timestamps are specified as `null`, indicating that they should not be changed.

In order to verify the new last modified time, we might be tempted to use the current `BasicFileAttributes` object associated with the view, but its copies of the file attribute values are not updatable. We can create a new `BasicFileAttributes` object that reflects the new values of the file attributes, or alternately use the `getLastModifiedTime()` or the `getAttribute()` methods of the `Files` class, as shown at (11) and (12), respectively.

The `PosixFileAttributeView` Interface

As the `PosixFileAttributeView` interface is a subinterface of the `BasicFileAttributeView` interface, it allows both the basic set of file attributes and the POSIX-specific file attributes to be read and updated.

```
interface PosixFileAttributeView
          extends BasicFileAttributeView, FileOwnerAttributeView
```

`String name()`

Returns the name of the attribute view, which in this case is the string `"posix"`.

`PosixFileAttributes readAttributes()`

Retrieves the basic and POSIX-specific file attributes as a bulk operation into a `PosixFileAttributes` object whose methods can be used to read the values of these file attributes (p. 1332). This method is analogous to the `readAttributes()` method of the `Files` class (p. 1328).

`void setGroup(GroupPrincipal group) throws IOException`

Updates the group of the directory entry. Note that there is no analogous method in the `Files` class for handling the group.

```
void setPermissions(Set<PosixFilePermission> perms)
```

Updates the file permissions. This method is analogous to the `Files.setPosixFilePermissions()` method (p. 1325).

The `PosixFileAttributeView` interface extends the `java.nio.file.attribute.FileOwnerAttributeView` interface that defines the methods for reading and updating the owner of the directory entry. See also analogous methods relating to ownership in the `Files` class (p. 1325).

```
interface java.nio.file.attribute.FileOwnerAttributeView
          extends FileAttributeView

UserPrincipal getOwner() throws IOException
void          setOwner(UserPrincipal owner) throws IOException
```

Return or update the owner of a directory entry, respectively. These methods are analogous to the methods in the `Files` class (p. 1325).

A `PosixFileAttributeView` object can thus read both the basic set of file attributes and the POSIX-specific file attributes, and can update the owner, group, file permissions, and timestamps for the last modified, last access, and creation times for a directory entry.

The code below obtains a `PosixFileAttributeView` object at (13) that pertains to the file denoted by the `path` reference. The associated `PosixFileAttributes` object is obtained at (14), providing read-only access to the basic file attributes and the POSIX-specific file attributes, whose values are printed by calling the methods `printBasicFileAttributes()` and `printPosixFileAttributes()` in the utility class `FileUtils`, respectively.

```
Path path = Path.of("project", "src", "pkg", "Main.java");
out.println("File: " + path);

PosixFileAttributeView pfaView = Files.getFileAttributeView(path,      // (13)
                                        PosixFileAttributeView.class);
System.out.printf("Using view: %s%n", pfaView.name());

// Reading the basic + POSIX set of file attributes:                   // (14)
PosixFileAttributes pfa2 = pfaView.readAttributes();
FileUtils.printBasicFileAttributes(pfa2);
FileUtils.printPosixFileAttributes(pfa2);

// Updating owner and group file attributes using view.                // (15)
FileSystem fs = path.getFileSystem();
UserPrincipalLookupService upls = fs.getUserPrincipalLookupService();
UserPrincipal newUser = upls.lookupPrincipalByName("javadude");
GroupPrincipal newGroup = upls.lookupPrincipalByGroupName("admin");
pfaView.setOwner(newUser);
pfaView.setGroup(newGroup);

//Updating file permissions using view.                               // (16)
Set<PosixFilePermission> newPerms = PosixFilePermissions.fromString("r--r--r--");
pfaView.setPermissions(newPerms);
```

```
//Updating last access time using view.                                    // (17)
FileTime currentAccessTime = pfa2.lastAccessTime();
long newLATinMillis = currentAccessTime.toMillis() + 10*60*1000L;
FileTime newLastAccessTime = FileTime.fromMillis(newLATinMillis);
pfaView.setTimes(null, newLastAccessTime, null);

// Reading the updated file attributes:                                     // (18)
pfa2 = pfaView.readAttributes();
FileUtils.printBasicFileAttributes(pfa2);
FileUtils.printPosixFileAttributes(pfa2);
```

The code from (15) to (17) shows how the `PosixFileAttributeView` object can be used to update various file attributes. Keep in mind that this view inherits from the `BasicFileAttributeView` and the `FileOwnerAttributeView` interfaces.

The code at (15) updates the owner and the group of the directory entry via the view. An owner and a group are looked up in the appropriate lookup services, and updated by the `setOwner()` and `setGroup()` methods of the `PosixFileAttributeView` interface. See also corresponding methods in the `Files` class (p. 1325).

The code at (16) updates the file permissions of the directory entry via the view, analogous to the `Files.setPosixFilePermissions()` method (p. 1325). File permissions are set to read-only for the owner, the group, and other users.

The code at (17) updates only the last access time of the directory entry via the view, analogous to updating the last modified time via the `BasicFileAttributeView` object (p. 1334).

Updated file attribute values can be read using the appropriate methods of the `Files` class, or by obtaining a new `PosixFileAttributes` object, as shown at (18).

The `DosFileAttributeView` Interface

As the `DosFileAttributeView` interface is a subinterface of the `BasicFileAttributeView` interface, it allows both the basic set of file attributes and the DOS-specific file attributes to be read and updated. Its usage is analogous to the `PosixFileAttributeView` interface discussed earlier (p. 1336).

```
interface DosFileAttributeView extends BasicFileAttributeView
```

`String name()`

Returns the name of the attribute view, which in this case is the string `"dos"`.

`DosFileAttributes readAttributes()`

Reads the basic file attributes as a bulk operation. The `DosFileAttributes` object can be used to read the values of the basic and DOS-specific file attributes (p. 1333). This method is analogous to the `readAttributes()` method of the `Files` class (p. 1328)

```
void setReadOnly(boolean value)
void setSystem(boolean value)
void setArchive(boolean value)
void setHidden(boolean value)
```

Update the value of the appropriate attribute. These methods are all implementation specific.