NSFileManager

Inherits From:	NSObject
Conforms To:	NSObject (NSObject) NSCopying
Declared In:	Foundation/NSFileManager.h

Class Description

NSFileManager enables you to perform many generic file-system operations. With it you can:

- Create directories and files.
- Extract the contents of files (as NSData objects).
- Change your current working location in the file system.
- Copy, move, and link files and directories.
- Remove files, links, and directories.
- Determine the attributes of a file, a directory, or the file system.
- Set the attributes of a file or directory.
- Make and evaluate symbolic links.
- Determine the contents of directories.
- Compare files and directories for equality.

Besides offering a useful range of generic functionality, the NSFileManager API insulates an application from the underlying file system. An important part of this insulation is the encoding of file names (in, for example, Unicode, ISO Latin1, and ASCII). This insulating layer makes it easier to port the application between operating systems with different file systems. There is a default NSFileManager object for the file system; this object responds to all messages that request a operation on the associated file system.

The pathnames specified as arguments to NSFileManager methods can be absolute or relative to the current directory (which you can determine with **currentDirectoryPath** and set with **changeCurrentDirectoryPath:**). However, pathnames cannot include wildcard characters.

Note: On UNIX file systems (such as NEXTSTEP) an absolute pathname starts with the root directory of the file system, represented by a slash (/), and ends with the file or directory that the pathname identifies. A relative pathname is relative to the *current directory*, the directory in which you are working and in which saved files are currently stored (if no pathname is specified). Relative pathnames start with a subdirectory of the current directory—without an initial slash—and end with the name of the file or directory that the pathname identifies.

Path Utilities

NSFileManager methods are commonly used together with path-utility methods implemented as a category on NSString. These methods extract the components of a path (directory, file name, and extension), create paths from those components, “translate” path separators for the given platform, clean up paths containing symbolic links and redundant slashes, and perform similar tasks. Where your code manipulates strings that are part of file-system paths, it should use these methods. See the specification of the NSString class cluster for details.

Method Types

Getting the default manager	+ defaultManager
Directory operations	– changeCurrentDirectoryPath: – createDirectoryAtPath:attributes: – currentDirectoryPath
File operations	– copyPath:toPath:handler: – createFileAtPath:contents:attributes: – movePath:toPath:handler: – linkPath:toPath:handler: – removeFileAtPath:handler:
Getting and comparing file contents	– contentsAtPath: – contentsEqualAtPath:andPath:
Detemining access to files	– fileExistsAtPath: – fileExistsAtPath:isDirectory: – isReadableFileAtPath: – isWritableFileAtPath: – isExecutableFileAtPath: – isDeletableFileAtPath:
Getting and setting attributes	– fileAttributesAtPath:traverseLink: – fileSystemAttributesAtPath: – changeFileAttributes:atPath:
Discovering directory contents	– directoryContentsAtPath: – enumeratorAtPath: – subpathsAtPath:
Symbolic-link operations	– createSymbolicLinkAtPath:pathContent: – pathContentOfSymbolicLinkAtPath:
Converting file-system representations	– fileSystemRepresentationWithPath: – stringWithFileSystemRepresentation:length:

Class Methods

defaultManager

+ (NSFileManager *)**defaultManager**

Returns the default NSFileManager object for the file system. You invoke all NSFileManager instance methods with this object as the receiver.

Instance Methods

changeCurrentDirectoryPath:

– (BOOL)**changeCurrentDirectoryPath:**(NSString *)*path*

Changes the path of the current directory to *path* and returns YES if successful, NO if not successful. All relative pathnames refer implicitly to the current working directory. The current working directory is stored per task.

See also: – **currentDirectoryPath**, – **fileExistsAtPath:isDirectory:**, – **directoryContentsAtPath:**, – **createDirectoryAtPath:attributes:**

changeFileAttributes:atPath:

– (BOOL)**changeFileAttributes:**(NSDictionary *)*attributes* **atPath:**(NSString *)*path*

Changes the attributes of the file or directory specified by *path*. Attributes that you can change are the owner, the group, file permissions, and the modification date. As in the POSIX standard, the application must either own the file or directory or must be running as superuser for attribute changes to take effect. The method attempts to make all changes specified in *attributes* and ignores any rejection of an attempted modification. If all changes succeed, it returns YES. If any change fails, the method returns NO, but it is undefined whether any changes actually occured.

Some useful global keys for identifying object values in the *attributes* dictionary are:

Key	Value Type
NSFileModificationDate	NSDate
NSFilePosixPermissions	NSNumber

The NSFilePosixPermissions value must be initialized with the code representing the POSIX file-permissions bit pattern.

You can change single attributes or any combination of attributes; you need not specify keys for all four attributes.

See also: – **fileAttributesAtPath:traverseLink:**

contentsAtPath:

– (NSData *)**contentsAtPath:**(NSString *)*path*

Returns the contents of the file specified in *path* as an NSData object. If *path* specifies a directory, or if some other error occurs, this method returns **nil**.

See also: – **contentsEqualAtPath:andPath**:, – **createFileAtPath:contents:attributes:**

contentsEqualAtPath:andPath:

– (BOOL)**contentsEqualAtPath:**(NSString *)*path1* **andPath:**(NSString *)*path2*

Compares the file or directory specified in *path1* with that specified in *path2* and returns YES if they have the same contents. If *path1* and *path2* are directories, the contents are the list of files and subdirectories each contain; contents of subdirectories are compared. If the contents differ in any way, this method returns NO. It does not traverse symbolic links but compares the links themselves.

See also: – **contentsAtPath:**

copyPath:toPath:handler:

– (BOOL)**copyPath:**(NSString *)*source*
toPath:(NSString *)*destination*
handler:*handler*

Copies the directory or file specified in path *source* to a different location in the file system identified by pathname *destination*. If *source* is a file, the method creates a file at *destination* that holds the exact contents of the original file (this includes UNIX special files). If *source* is a directory, the method creates a new directory at *destination* and recursively populates it with duplicates of the files and directories contained in *source*, preserving all links. The file specified in *source* must exist, while *destination* must not exist prior to the operation. When a file is being copied, the *destination* path must end in a file name; there is no implicit adoption of the *source* file name. Symbolic links are not traversed but are themselves copied.

If the copy operation is successful, the method returns YES. If the operation is not successful, but the callback *handler* of **fileManager:shouldProceedAfterError:** returns YES (see below), **copyPath:toPath:handler:** also returns YES. Otherwise this copy method returns NO. The method also attempts to make the attributes of the directory or file at *destination* identical to *source*, but ignores any failure at this attempt.

The argument *handler* identifies an object that responds to the callback messages **fileManager:willProcessPath:** and **fileManager:shouldProceedAfterError:** (see "Methods Implemented by the CallbackHandler," below). This callback mechanism is similar to delegation. NSFileManager sends the first message when it begins a copy, move, remove, or link operation. It sends the second message when it encounters any error in processing. You can specify **nil** for *handler* if no object

responds to the callback messages; if you specify **nil** and an error occurs, the method automatically returns NO.

This code fragment verifies that the file to be copied exists and then copies that file to the user's **~/Library/Reports** directory:

```
NSString *source = @"/tmp/quarterly_report.rtf";
NSString *destination = [[NSHomeDirectory()
   stringByAppendingPathComponent:@"Library"]
   stringByAppendingPathComponent:@"Reports"];
NSFileManager *manager = [NSFileManager defaultManager];
if ([manager fileExistsAtPath:source])
      [manager copyPath:source toPath:destination handler:nil];
```

See also: – **linkPath:toPath:handler:**, – **movePath:toPath:handler:**, – **fileManager:shouldProceedAfterError:**, – **removeFileAtPath:handler:**, – **fileManager:willProcessPath:**

createDirectoryAtPath:attributes:

– (BOOL)**createDirectoryAtPath:**(NSString *)*path*
attributes:(NSDictionary *)*attributes*

Creates a directory (without contents) at *path* that has the specified *attributes*. Returns YES upon success or NO upon failure. The directory to be created must not exist yet at *path*. The file attributes that you can set are owner and group numbers, file permissions, and modification date. If you specify **nil** for *attribute*s, default values for these attributes are set (particularly write access for creator and read access for others). The following table lists the global constants used as keys in the *attributes* NSDictionary and the types of the associated values:

Key	Value Type
NSFileModificationDate	NSDate
NSFileOwnerAccount Number	NSNumber
NSFileGroupOwnerAccountNumber	NSNumber
NSFilePosixPermissions	NSNumber

See also: – **changeCurrentDirectoryPath:**, – **changeFileAttributes:atPath:**, **– createFileAtPath:contents:attributes:**, **– currentDirectoryPath**

createFileAtPath:contents:attributes:

– (BOOL)**createFileAtPath:**(NSString *)*path*
contents:(NSData *)*contents*
attributes:(NSDictionary *)*attributes*

Creates a file at *path* that contains *contents* and has the specified file *attributes*. Returns YES upon success or NO upon failure. The file attributes that you can set are owner and group numbers, file permissions, and modification date. If you specify **nil** for *attribute*s, the file is given a default set of attributes. The following table summarizes the the keys and types to associate with values in the NSDictionary *attributes*.

Key	Value Type
NSFileModificationDate	NSDate
NSFileOwnerAccount Number	NSNumber
NSFileGroupOwnerAccountNumber	NSNumber
NSFilePosixPermissions	NSNumber

See also: – **contentsAtPath:**, – **changeFileAttributes:atPath:**, – **fileAttributesAtPath:traverseLink:**

createSymbolicLinkAtPath:pathContent:

– (BOOL)**createSymbolicLinkAtPath:**(NSString *)*path*
pathContent:(NSString *)*otherPath*

Creates a symbolic link identified by *path* that refers to the location *otherPath* in the file system. Returns YES if the operation is successful and NO if it is not successful. The method returns NO if a file, directory, or symbolic link identical to *path* already exists.

See also: – **pathContentOfSymbolicLinkAtPath:**, – **linkPath:toPath:handler:**

currentDirectoryPath

– (NSString *)**currentDirectoryPath**

Returns the path of the program's current directory. Relative pathnames refer implictly to this directory; for example, if the current directory is **/tmp**, and the relative pathname is **reports/info.txt**, the full pathname is constructed as **/tmp/reports/info.txt**. This path is initialized to the current working directory, and can be thereafter reset with **changeCurrentDirectoryPath:**. If the application's current working directory isn't accessible, this method returns **nil**.

See also: – **createDirectoryAtPath:attributes:**

directoryContentsAtPath:

– (NSArray *)**directoryContentsAtPath:**(NSString *)*path*

Returns an array containing the filenames (including directories and symbolic links) immediately below the directory specified in *path*. The results are shallow, going no further than the next level below the specified directory. Here is sample output, generated by NSArray's **description** method, when this method is invoked with **/NextDeveloper** as *path*.

```
Demos,
Apps,
Makefiles,
OpenStepConversion,
Examples,
Headers,
2.0CompatibleHeaders,
Palettes
```

As the example shows, the results omit the path preceeding the subdirectory. This method skips "." and ".." and does not traverse symbolic links in the specified directory. It returns **nil** if the directory specified at *path* does not exist or if there is some other error in accessing it.

See also: – **currentDirectoryPath**, – **fileExistsAtPath:isDirectory:**, – **enumeratorAtPath:**, – **subpathsAtPath:**

enumeratorAtPath:

– (NSDirectoryEnumerator *)**enumeratorAtPath:**(NSString *)*path*

Returns an NSDirectoryEnumerator with which to enumerate the contents of the directory specified at path. This enumeration, which returns NSString objects, goes very deep and hence is very useful for large file-system subtrees. If the method discovers a new mount point, it traverses the mount point. It also reports any symbolic links it discovers. It returns **nil** if it cannot get the device of the linked-to file.

This code fragment enumerates the subdirectories and files under **/MyAccount/Documents** and processes all files with an extension of **.doc**:

```
NSString *file;
NSDirectoryEnumerator *enumerator = [[NSFileManager defaultManager]
   enumeratorAtPath:@"/MyAccount/Documents"];
while (file = [enumerator nextObject]) {
   if ([[file pathExtension] isEqualToString:@"doc"])
      [self scanDocument:file];
}
```

The NSDirectoryEnumerator class has methods for obtaining the attributes of the existing path and of the parent directory, and for skipping descendents of the existing path.

See also: – **currentDirectoryPath**, – **fileExistsAtPath:isDirectory:**, – **directoryContentsAtPath:**, – **subpathsAtPath:**

fileAttributesAtPath:traverseLink:

– (NSDictionary *)**fileAttributesAtPath:**(NSString *)*path* **traverseLink:**(BOOL)*flag*

Returns an NSDictionary containing various objects that represent the POSIX attributes of the file specified at *path*. You access these objects using these global constants as keys:

Key	Value Type
NSFileSize (in bytes)	NSNumber
NSFileModificationDate	NSDate
NSFileOwnerAccount Number	NSNumber
NSFileGroupOwnerAccountNumber	NSNumber
NSFileReferenceCount (number of hard links)	NSNumber
NSFileIdentifier	NSNumber
NSFileDeviceIdentifier	NSNumber
NSFilePosixPermissions	NSNumber
NSFileType	NSString

NSFileType's global strings are defined as:

- NSFileTypeDirectory
- NSFileTypeRegular
- NSFileTypeSymbolicLink
- NSFileTypeSocket
- NSFileTypeCharacterSpecial
- NSFileTypeBlockSpecial
- NSFileTypeUnknown

If *flag* is YES and *path* is a symbolic link, the attributes of the linked-to file are returned; if *flag* is NO, the attributes of the symbolic link are returned.

This piece of code gets several attributes of a file and logs them.

```
NSNumber *fsize, *refs, *owner;
NSDate *moddate;
```

```
NSDictonary *fattrs =
 [manager fileAttributesAtPath:@"/tmp/List" traverseLink:YES];

if (!fattrs) {
   NSLog(@"Path is incorrect!");
   return;
}
if (fsize = [fattrs objectForKey:NSFileSize])
   NSLog(@"File size: %d\n", [fsize intValue]);

if (refs = [fattrs objectForKey:NSFileReferenceCount])
   NSLog(@"Ref Count: %d\n", [refs intValue]);

if (moddate = [fattrs objectForKey:NSFileModificationDate])
   NSLog(@"Modif Date: %@\n", [moddate description]);
```

As a convenience, NSDictionary provides a set of methods (declared as a category in **NSFileManager.h**) for quickly and efficiently obtaining attribute information from the returned NSDictionary: **fileSize**, **fileType**, **fileModificationDate**, and **filePosixPermissions**. For example, you could rewrite the last statement in the code example above as:

```
if (moddate = [fattrs fileModificationDate])
   NSLog(@"Modif Date: %@\n", [moddate description]);
```

See also: – **changeFileAttributes:atPath:**

fileExistsAtPath:

– (BOOL)**fileExistsAtPath:**(NSString *)*path*

Returns YES if the file specified in *path* exists, or NO if it does not. The method traverses final symbolic links.

See also: – **fileExistsAtPath:isDirectory:**

fileExistsAtPath:isDirectory:

– (BOOL)**fileExistsAtPath:**(NSString *)*path* **isDirectory:**(BOOL *)*isDirectory*

Returns whether the file specified in *path* exists. If you want to determine if *path* is a directory, specify the address of a boolean variable for *isDirectory*; the method indirectly returns YES if *path* is a directory. The method traverses final symbolic links.

This example gets an NSArray that identifies the fonts in **/NextLibrary/Fonts**:

```
NSArray *subpaths;
```

```
BOOL isDir;
NSString *fontPath = @"/NextLibrary/Fonts";
NSFileManager *manager = [NSFileManager defaultManager];
if ([manager fileExistsAtPath:fontPath isDirectory:&isDir]
   && isDir)
    subpaths = [manager subpathsAtPath:fontPath];
```

See also: – **fileExistsAtPath:**

fileSystemAttributesAtPath:

– (NSDictionary *)**fileSystemAttributesAtPath:**(NSString *)*path*

Returns an NSDictionary containing objects that represent attributes of the mounted file system; *path* is any pathname within the mounted file system. You access the attribute objects in the NSDictionary using these global constants as keys:

Key	Value Type
NSFileSystemSize (in an appropriate unit, usually bytes)	NSNumber
NSFileSystemFreeSize (in an appropriate unit, usually bytes)	NSNumber
NSFileSystemNodes	NSNumber
NSFileSystemFreeNodes	NSNumber
NSFileSystemNumber	NSNumber

The following code example checks to see if there's sufficient space on the file system before adding a new file to it:

```
const char *data = [[customerRec description] cString];
NSData *contents = [NSData dataWithBytes:data length:sizeof(data)];
NSFileManager *manager = [NSFileManager defaultManager];
NSDictionary *fsattrs =
    [manager fileSystemAttributesAtPath:@"/Net/sales/misc"];
if ([[fsattrs objectForKey:NSFileSystemFreeSize] unsignedIntValue]
    > [contents length])
    [manager createFileAtPath:@"/Net/sales/misc/custrec.rtf"
         contents:contents attributes:nil];
```

See also: – **fileAttributesAtPath:traverseLink:**, – **changeFileAttributes:atPath:**

fileSystemRepresentationWithPath:

– (const char *)**fileSystemRepresentationWithPath:**(NSString *)*path*

Returns a C-string representation of *path* that properly encodes Unicode strings for use by the file system. If you need the C string beyond the scope of your autorelease pool, you should copy it. This method raises an exception upon error. Use this method if your code calls system routines that expect C-string path arguments.

See also: – **stringWithFileSystemRepresentation:length:**

isDeletableFileAtPath:

– (BOOL)**isDeletableFileAtPath:**(NSString *)*path*

Returns YES if the invoking object appears to be able to delete the directory or file specified in *path* and NO if it cannot. To be deletable, either the parent directory of *path* must be writable and its owner must be the application; if *path* is a directory, it must have no undeletable items in it. This method does not traverse symbolic links.

isExecutableFileAtPath:

– (BOOL)**isExecutableFileAtPath:**(NSString *)*path*

Returns YES if the underlying operating system appears able to execute the file specified in path and NO if it cannot. This method traverses symbolic links.

isReadableFileAtPath:

– (BOOL)**isReadableFileAtPath:**(NSString *)*path*

Returns YES if the invoking object appears able to read the file specified in *path* and NO if it cannot. This method traverses symbolic links.

isWritableFileAtPath:

– (BOOL)**isWritableFileAtPath:**(NSString *)*path*

Returns YES if the invoking object appears able to write to the file specified in *path* and NO if it cannot. This method traverses symbolic links.

linkPath:toPath:handler:

– (BOOL)**linkPath:**(NSString *)*source*
toPath:(NSString *)*destination*
handler:*handler*

If pathname *source* identifies a file, this method hard-links the directory or file specified in *destination* to it. If *source* is a directory or a symbolic link, this method copies it to *destination* instead of creating a hard link. The file, link, or directory specified in *source* must exist, while *destination* must not yet exist. The *destination* path must end in a file name; there is no implicit adoption of the *source* file name. Symbolic links in *source* are not traversed.

If the link operation is successful, **linkPath:toPath:handler:** returns YES. If the operation is not successful, but the *handler* method **fileManager:shouldProceedAfterError:** returns YES, the method also returns YES. Otherwise it returns NO.

The argument *handler* identifies an object that responds to the callback messages **fileManager:willProcessPath:** and **fileManager:shouldProceedAfterError:** (see "Methods Implemented by the Callback Handler," below). This callback mechanism is similar to delegation. NSFileManager sends the first message when it begins a copy, move, remove, or link operation. It sends the second message when it encounters any error in processing. You can specify **nil** for *handler* if no object responds to the callback messages; if you specify **nil** and an error occurs, the method automatically returns NO.

This code fragment verifies the pathname typed in a text field (**imageFileField**) and then links the file to the user's **~/Library/Images** directory:

```
NSString *imageFile = [imageFileField stringValue];
NSString *destination = [[NSHomeDirectory()
    stringByAppendingPathComponent:@"Library"]
    stringByAppendingPathComponent:@"Images"];
NSFileManager *manager = [NSFileManager defaultManager];
if ([manager fileExistsAtPath:source])
       [manager linkPath:source toPath:destination handler:self];
```

See also: – **copyPath:toPath:handler:**, – **createSymbolicLinkAtPath:pathContent:**, – **movePath:toPath:handler:**, – **fileManager:shouldProceedAfterError:,** – **removeFileAtPath:handler:**, – **fileManager:willProcessPath:**,

movePath:toPath:handler:

– (BOOL)**movePath:**(NSString *)*source*
toPath:(NSString *)*destination*
handler:*handler*

Moves the directory or file specified in path *source* to a different location in the file system identified by the pathname *destination*. If *source* is a file, the method creates a file at *destination* that holds the exact

contents of the original file (including UNIX special files) and then deletes the original file. If *source* is a directory, **movePath:toPath:handler:** creates a new directory at *destination* and recursively populates it with duplicates of the files and directories contained in *source*; it then deletes the old directory and its contents. The file specified in *source* must exist, while *destination* must not yet exist. The *destination* path must end in a file name; there is no implicit adoption of the *source* file name. Symbolic links are not traversed; however, links are preserved.

If the move operation is successful, the method returns YES. If the operation is not successful, but the *handler* method **fileManager:shouldProceedAfterError:** returns YES, **movePath:toPath:handler:** also returns YES; otherwise it returns NO. If a failure in a move operation occurs, the pre-existing path or the new path remains intact, but not both.

The argument *handler* identifies an object that responds to the callback messages **fileManager:willProcessPath:** and **fileManager:shouldProceedAfterError:** (see "Methods Implemented by the Callback Handler," below). This callback mechanism is similar to delegation. NSFileManager sends the first message when it begins a copy, move, remove, or link operation. It sends the second message when it encounters any error in processing. You can specify **nil** for *handler* if no object responds to the callback messages; if you specify **nil** and an error occurs, the method automatically returns NO.

See also: – **copyPath:toPath:handler:**, – **linkPath:toPath:handler:**, – **removeFileAtPath:handler:**, – **fileManager:shouldProceedAfterError:**, – **fileManager:willProcessPath:**

pathContentOfSymbolicLinkAtPath:

– (NSString *)**pathContentOfSymbolicLinkAtPath:**(NSString *)*cStringPath*

Returns the actual path of the directory or file that the symbolic link *cStringPath* refers to. Returns **nil** upon failure.

See also: – **createSymbolicLinkAtPath:pathContent:**

removeFileAtPath:handler:

– (BOOL)**removeFileAtPath:**(NSString *)*path* **handler:***handler*

Deletes the file, link, or directory (including, recusively, all subdirectories, files and links in the directory) identified by *path*. If the removal operation is successful, **removeFileAtPath:handler:** returns YES. If the operation is not successful, but the *handler* method **fileManager:shouldProceedAfterError:** returns YES, **removeFileAtPath:handler:** also returns YES; otherwise it returns NO.

The argument *handler* identifies an object that responds to the callback messages **fileManager:willProcessPath:** and **fileManager:shouldProceedAfterError:** (see "Methods Implemented by the Callback Handler," below). This callback mechanism is similar to delegation. NSFileManager sends the first message when it begins a copy, move, remove, or link operation. It sends

the second message when it encounters any error in processing. You can specify **nil** for *handler* if no object responds to the callback messages; if you specify **nil** and an error occurs, the method automatically returns NO.

Since the removal of directory contents is so thorough and final, be careful when using this method. Do not specify "." or ".." for *path*; this will raise the exception NSInvalidArgumentException. This method does not traverse symbolic links.

See also: **copyPath:toPath:handler:**, – **linkPath:toPath:handler:**, – **movePath:toPath:handler:**, – **fileManager:shouldProceedAfterError:**, – **fileManager:willProcessPath:**

stringWithFileSystemRepresentation:length:

– (NSString *)**stringWithFileSystemRepresentation:**(const char *)*string*
length:(unsigned int)*len*

Returns an NSString object converted from a C-string representation of a path name in the current file system (*string*). Use this method if your code receives paths as C-strings from system routines.

See also: – **fileSystemRepresentationWithPath:**

subpathsAtPath:

– (NSArray *)**subpathsAtPath:**(NSString *)*path*

Returns an NSArray that lists (as NSStrings) the contents of the directory identified by path. This list of directory contents goes very deep and hence is very useful for large file-system subtrees. The method skips "." and "..". If path is a symbolic link, **subpathsAtPath:** traverses the link. The method returns **nil** if it cannot get the device of the linked-to file.

Here is a sample fragment of what **subpathsAtPath:** returns (as the output of NSArray's **description** method) when *path* is **/NextDeveloper**:

```
Demos/AppInspector.app/Voyeur.nib/PauseH.tiff,
Demos/AppInspector.app/Voyeur.nib/data.classes,
Demos/AppInspector.app/Voyeur.nib/data.nib,
Demos/AppInspector.app/check.tiff,
Demos/AppInspector.app/checkH.tiff,
Demos/AppInspector.app/AppInspector,
Headers,
Headers/architecture,
Headers/architecture/ARCH_INCLUDE.h,
Headers/architecture/adb_bus.h,
Headers/architecture/adb_kb_codes.h,
Headers/architecture/adb_kb_map.h,
Headers/architecture/alignment.h,
```

Notice that this method reveals every element of the subtree at *path*, including the contents of file packages (such as applications, nib files, and RTFD files). This code fragment gets the contents of **/NextLibrary/Fonts** after verifying that the directory exists:

```
BOOL isDir=NO;
NSArray *subpaths;
NSString *fontPath = @"/NextLibrary/Fonts";
NSFileManager *manager = [NSFileManager defaultManager];
if ([manager fileExistsAtPath:fontPath isDirectory:&isDir] && isDir)
    subpaths = [manager subpathsAtPath:fontPath];
```

See also: **– directoryContentsAtPath:**, – **enumeratorAtPath:**

Methods Implemented by the Callback Handler (Notification)

fileManager:shouldProceedAfterError:

– (BOOL)**fileManager:**(NSFileManager *)*manager*
shouldProceedAfterError:(NSDictionary *)*errorInfo*

NSFileManager sends this message for each error it encounters when copying, moving, removing, or linking files or directories. The NSDictionary object *errorInfo* contains two or three pieces of information (all NSStrings) related to the error:

Key	Value
@"Path"	The path related to the error (usually the source path)
@"Error"	A description of the error
@"ToPath"	The destination path (not all errors)

Return YES if the operation (which is often continuous within a loop) should proceed and NO if it should not; the Boolean value is passed back to the invoker of **copyPath:toPath:handler:**, **movePath:toPath:handler:**, **removeFileAtPath:handler:** or **linkPath:toPath:handler:**. If an error occurs and your handler has not implemented this method, the invoking method automatically returns NO.

The following implementation of **fileManager:shouldProceedAfterError:** displays the error string in an attention panel and leaves it to the user whether to proceed or stop:

```
-(BOOL)fileManager:(NSFileManager *)manager
      shouldProceedAfterError:(NSDictionary *)errorDict
{
   int result;
   result = NSRunAlertPanel(@"Gumby App", @"File operation error:
      %@ with file: %@", @"Proceed", @"Stop", NULL,
      [errorDict objectForKey:@"Error"],
      [errorDict objectForKey:@"Path"]);

   if (result == NSAlertDefaultReturn)
```

```
        return YES;
      else
         return NO;
  }
```

See also: – **fileManager:willProcessPath:**

fileManager:willProcessPath:

– (void)**fileManager:**(NSFileManager *)*manager* **willProcessPath:**(NSString *)*path*

NSFileManager sends this message to the designated handler for each file or directory (identified by *path*) that it is about to copy, link, remove, or move. This notification gives you the opportunity to update your user interface or to do anything similar where the knowledge of *path* is important.

See also: – **fileManager:shouldProceedAfterError:**