

VisualAge[®] C++ Professional for AIX[®]



IBM[®] Open Class[™] : File Systems

Version 5.0

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page v.

Edition Notice

This edition applies to Version 5.0 of IBM VisualAge C++ and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

© Copyright International Business Machines Corporation 1998, 2000. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	v	Path Names and Path Name Parsers	10
Programming Interface Information	vii	Work with Path Names	11
Trademarks and Service Marks	vii	Data Accessors	13
Industry Standards	viii	Access File Contents	14
About This Book.	ix	File System Iterators	15
File Systems.	1	Access Directory and Volume Contents	15
File System Entities	1	File System Movers and Copiers	17
Instantiate the File System Classes	3	Copy and Move Files and Directories.	18
Create and Delete Files and Directories	5	Thread Safety.	20
Get and Set Information about File System Entities	7	Customize File System Operations.	20
		File System Exceptions	22

Notices

Note to U.S. Government Users Restricted Rights -- use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd.
1150 Eglinton Avenue East
Toronto, Ontario M3C 1H7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

® (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. ® Copyright IBM Corp. 1998, 2000. All rights reserved.

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interface allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

The following terms are trademarks of the International Business Machines Corporation in the United States, or other countries, or both:

AIX
AS/400
DB2
CICS
C Set ++
IBM
Network Station
Object Connection
OS/2
OS/390
OS/400
Open Class
Operating System/2
Operating System/400
PowerPC 403
PowerPC 601
PowerPC 603
PowerPC 604
Presentation Manager
RS/6000
S/390
SAA
Systems Application Architecture
TeamConnection
VisualAge
WebSphere
Workplace Shell

Lotus, Lotus Notes, and Domino are trademarks or registered trademarks of the Lotus Development Corporation in the United States, or other countries, or both.

Tivoli Management Environment, TME 10, and Tivoli Module Designer are trademarks of Tivoli Systems Inc. in the United States, or other countries, or both.

Encina and DCE Encina Lightweight Client are trademarks of Transarc Corporation in the United States, or other countries, or both.

Microsoft, Win32, Windows, Windows NT, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

UNIX is a registered trademark in the U.S. and other countries licensed exclusively through X/Open Company Limited.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks or registered trademarks of Intel Corporation in the United States, or other countries, or both.

C-bus is a registered trademark of Corollary, Inc.

PC Direct is a registered trademark of Ziff Communicatoins Company and is used by IBM Corporation under license

Other company, product, and service names, which may be denoted by a double asterisk(**), may be trademarks or service marks of others.

Industry Standards

VisualAge C++ Professional for AIX, Version 5.0 supports the following standards:

- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899-1990 [1992]). This standard has officially replaced American National standard for Information Systems-Programming Language C (X3.159-1989) and is technically equivalent to the ANSI C standard. VisualAge C++ supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The IBM Systems Application Architecture (SAA) C Level 2 language definition.
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The ISO/IEC 9945-1:1990/IEEE POSIX 1003.-1990 standard.
- The X/Open Common Applications Environment Specifications, System Interfaces and Headers, Issue 4.

About This Book

The information in this PDF document is also available in the online help.

To find this information, or any topics listed in this document as Related Concepts, Related Tasks, or Related References, simply type the topic title into the search bar in the top frame of your browser in the online help.

For some topics, the suggested references may already be contained in this document. In such cases, there is a cross-reference to the page on which the related topic appears.

File Systems

The IBM Open Class File System Framework gives you access to all file system objects such as volumes, directories, files, file contents and operations. Its interface provides classes that represent physical file system objects and allows you to manipulate those objects.

Some typical tasks you can perform using the File System Framework are:

- Creating, moving, copying, deleting, locating, and otherwise manipulating file system objects
- Manipulating and parsing path names in a portable manner
- Accessing the attributes of files, directories, and volumes
- Accessing the contents of volume, directory, and file objects
- Iterating through directories and volumes

File System Framework functions are organized into six broad categories of classes:

- File System Entities
- Path Names and Path Name Parsers
- Data Accessors
- File System Iterators
- File System Movers and Copiers
- File System Exceptions

RELATED CONCEPTS

"File System Entities"

"Path Names and Path Name Parsers" on page 10

"Data Accessors" on page 13

"File System Iterators" on page 15

"File System Movers and Copiers" on page 17

"Thread Safety" on page 20

"File System Exceptions" on page 22

RELATED TASKS

"Instantiate the File System Classes" on page 3

"Create and Delete Files and Directories" on page 5

"Access File Contents" on page 14

"Access Directory and Volume Contents" on page 15

"Copy and Move Files and Directories" on page 18

"Customize File System Operations" on page 20

"Get and Set Information about File System Entities" on page 7

"Work with Path Names" on page 11

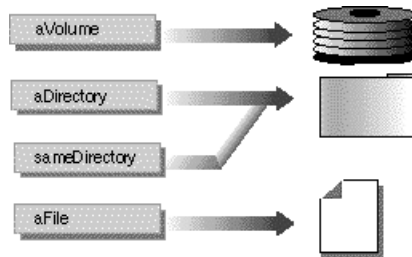
File System Entities

Overview of File System Entities

The File System Framework enables you to use surrogate instances called file system entities to access physical objects that reside on hardware or virtual devices. These surrogate objects are instances of `IFileSystemEntity` and its subclasses: `IFile`, `IDirectory` and `IVolume`.

- IFileSystemEntity is the base class for all file system entities.
- IFile provides a surrogate object for a file.
- IDirectory provides a surrogate object for a file system directory.
- IVolume provides a surrogate object for a file system volume.

The following figure shows the relationship of surrogate instances to physical objects:



Multiple surrogate objects can refer to the same physical object.

The IFileSystemEntity class defines the common protocol for file system entities. IFile, IDirectory and IVolume derive from IFileSystemEntity and implement protocols specific to each kind of entity.

Files

An IFile instance is a surrogate entity that maps to a physical file. You locate and create IFile instances using the IFile constructors and IDirectory's lookUp and createFile methods. You can also delete the physical file entity by using a deleteSelf method.

Directories

An IDirectory instance is a surrogate entity that maps to a physical directory. With it, you can look up an entity within a directory, create a new directory, create a new file, delete the contents of a directory, and delete the directory itself.

Volumes

IVolume derives from IDirectory because it represents the root directory of a volume. You can treat an IVolume instance much as you would treat an IDirectory instance. IVolume also has several methods you can use to access attributes that are specific to volumes.

Attributes

Types of attributes for a file system object include:

- Name
- Creation date and time
- Modification date and time

In addition, IFile has a size attribute which you access using the size and setSize functions.

IFileSystemEntity implements functions for accessing the attributes that are common to all file system objects. IDirectory, IVolume, and IFile also implement accessors for attributes that are specific to their corresponding physical file system objects. For example, IFile instances manage end-of-file information stored as an

attribute. They also manage storage of the file name, creation date, modification date, file size, and entity type (which in this case is always file).

RELATED CONCEPTS

“File Systems” on page 1

RELATED TASKS

“Instantiate the File System Classes”

“Create and Delete Files and Directories” on page 5

“Access File Contents” on page 14

“Access Directory and Volume Contents” on page 15

“Copy and Move Files and Directories” on page 18

“Get and Set Information about File System Entities” on page 7

“Work with Path Names” on page 11

Instantiate the File System Classes

Instantiate a File System Entity from a Host-Specific Path Name

If you know the host system’s path name for the object, you can use an entity constructor that takes the path name as an argument. For example, the following code constructs an `IFile` for the file for which the host-specific path name is specified by `argv[1]`:

```
IFile myFile(IHostPathName(argv[1]));
```

If your path name is in the format for a different operating system, you can specify an object to use to parse the path:

```
IFile myFile(argv[1], IFile::kMustExist,  
             IUnixPathNameParser());
```

The code in the above example will fail if there is no file at the location specified by `argv[1]`.

The following code will create a directory named `xyz` in the subdirectory named `sub`:

```
IDirectory newDir = dir.createDirectory("sub/xyz",  
                                       IUnixPathNameParser());
```

If `sub` does not exist, `createDirectory` will automatically create it.

If you do not specify a parser to use for a host-specific path that is not in the current host’s format, one of the following will occur when the path is used:

- If the path is illegal for the current host’s format, you will get an `IInvalidName` exception.
- If the path happens to be legal in the host’s format, it might be interpreted differently than you expect. For example, if you create a file using the path “foo\bar” on a UNIX[®] system, the constructor may create one directory called “foo\bar” since the backslash (`\`) is a valid character in a UNIX file or directory name.

Instantiate a File System Entity from a Portable Path Name

If you already have an `IPathName` object, you can use it to construct an entity instance that represents the physical object at the location specified by the path. You simply use `IPathName` in the entity instance’s constructor:

```
IPathName aFullPath = ....;  
IFile myFile(aFullPath);
```

If you have an `IDirectory` instance for the directory in which you wish to create an entity, you can also use the `IDirectory` methods `createFile` and `createDirectory`.

To create a file named "abc", you do this:

```
IDirectory aDirectory = ....;
IFile newFile = aDirectory.createFile("abc");
```

The following code shows how to specify a subdirectory relative to the current directory. In this case the subdirectory is named "subdirectory":

```
// use path name relative to current directory
IFile myFile(IPathName("subdirectory"));
```

If you have a partial path name relative to a known directory, you can use `IDirectory::lookup`:

```
IPathName aPartialPath = ....;
IFile myFile = aDirectory.lookup(aPartialPath);
```

Instantiate a File System Entity from Another Instance of the Same Entity Class

You can make a copy of a file system entity by using the entity class' `copy` constructor. In each of the following examples, both `originalEntity` and `duplicateEntity` will point to the same physical object on disk:

```
IFile originalEntity(pathName);
IFile duplicateEntity(originalEntity);
IDirectory originalEntity(pathName);
IDirectory duplicateEntity(originalEntity);
IVolume originalEntity(hostPathName);
IVolume duplicateEntity(originalEntity);
IFileSystemEntity originalEntity(pathName);
IFileSystemEntity duplicateEntity(originalEntity);
```

Use an Instance of `IFileSystemEntity` to Represent a File, Directory or Volume

Instantiate and use subclasses of `IFileSystemEntity`, such as `IFile`, when you need to manipulate a specific type of file system object. If you only need to represent file system entities generically and the functions provided by `IFileSystemEntity` are sufficient, you can use an instance of `IFileSystemEntity` to represent a file, directory or volume:

```
IFileSystemEntity fileEntity(filePathName);
IFileSystemEntity directoryEntity(dirPathName);
IFileSystemEntity volumeEntity(volHostPathName);
```

If you try to manipulate an `IFileSystemEntity` instance in a way that is inconsistent with the type of physical object which the entity represents, you will get an exception. For example:

```
IFileSystemEntity myEntity = IFile(filename);
...
IDirectory myDirectory = myEntity; // WRONG: myEntity
// is not a directory!
// An exception of type
// IEntityMismatch is thrown.
```

RELATED CONCEPTS

"File Systems" on page 1

"File System Entities" on page 1

"Path Names and Path Name Parsers" on page 10

"File System Exceptions" on page 22

RELATED TASKS

“Create and Delete Files and Directories”
“Access File Contents” on page 14
“Access Directory and Volume Contents” on page 15
“Copy and Move Files and Directories” on page 18
“Customize File System Operations” on page 20
“Get and Set Information about File System Entities” on page 7
“Work with Path Names” on page 11

Create and Delete Files and Directories

IFile and IDirectory inherit creation options from IFileSystemEntity. You can use these options to specify creation behavior when instantiating an IFile or IDirectory object.

To automatically create a file or directory if it does not already exist, use the kCreateIfNeeded option:

```
IFile myFile(pathName, IFile::kCreateIfNeeded);  
IDirectory myDir(pathName, IDirectory::kCreateIfNeeded);
```

To create a new file or directory only where one does not already exist, use the kCreateOnly option (**Note:** this will throw an IAlreadyExists exception if the file already exists):

```
IFile myNewFile(pathName, IFile::kCreateOnly);  
IDirectory myNewDir(pathName, IDirectory::kCreateOnly);
```

To replace an existing file or directory, or create a new one if it does not already exist, use the kReplaceExisting option:

```
IFile myNewFile(pathName, IFile::kReplaceExisting);  
IDirectory myNewDir(pathName, IDirectory::kReplaceExisting);
```

Create a File or Directory and Intermediate Subdirectories

Except for kMustExist, the entity creation options will also cause any missing intermediate directories to be created. The following code will create the file file.txt if it does not already exist. It will also create the intermediate directories abc and xyz if needed:

```
IFile myFile(IHostPathName("abc/xyz/file.txt"),  
            IFile::kCreateIfNeeded, IUnixPathNameParser());
```

The following code will create a directory named xyz in the subdirectory named sub. If sub does not exist, createDirectory will automatically create it too:

```
IDirectory newDir = dir.createDirectory("sub/xyz",  
                                       IDirectory::kCreateOnly, IUnixPathNameParser());
```

Delete a File or Directory

IFile and IDirectory inherit the deleteSelf member function from IFileSystemEntity.

To delete a physical file, call its deleteSelf member function:

```
IFile fileToKill(pathName);  
fileToKill.deleteSelf();
```

To delete a physical directory, call its deleteSelf member function. By default, deleteSelf will only delete the directory if it is entirely empty. If the directory is not empty, deleteSelf will throw the exception IMustBeEmpty.

```
IDirectory dirToKill(pathName);  
dirToKill.deleteSelf();
```

If the directory is not empty and you want to delete it and all of its contents, pass true to `deleteSelf`, overriding the default value (false) of `deleteSelf`'s `deleteChildren` parameter.

```
dirToKill.deleteSelf(true); // Deletes all contents too!
```

If any of the entities in the directory or any of their children cannot be deleted, `deleteSelf` deletes as many as it can and then throws the first exception that it encountered.

Delete all Directory Contents

To recursively delete all the contents of a directory, but not the directory itself, call `deleteAllContents`. This method attempts to delete each member of the directory recursively.

```
aDirectory.deleteAllContents();
```

If any of the entities in the directory or any of their children cannot be deleted, `deleteAllContents` deletes as many as it can and then throws the first exception that it encountered.

Create Temporary Files and Directories

To create a directory or a file for storing temporary data, use `IDirectory::createTemporary` or `IFile::createTemporary`. These functions will create a new directory or a new file in the system-specific temporary directory. There are two versions of each of these functions: one to specify the name of the temporary entity, the other to generate the name automatically.

```
// create a 1K file with given name
IFile tempFile(IFile::createTemporary(fileName, 1024));
// create a zero-length file with given name
IFile tempFile(IFile::createTemporary(fileName));
// create a zero-length file with unique name
IFile tempFile(IFile::createTemporary());
// create a directory with given name
IDirectory tempDir(IDirectory::createTemporary(dirName));
// create a directory with unique name
IDirectory tempDir(IDirectory::createTemporary());
```

Temporary directories and files created in this way are not automatically deleted when their `IDirectory` or `IFile` instance goes out of scope. Nor are temporary files automatically deleted when they are closed. These entities are only “temporary” in the sense that they live in the system’s temporary directory and that they are not guaranteed to survive a system reboot. You are still responsible for deleting a temporary `IDirectory` or `IFile` when you are finished using it.

You can call the temporary function on a file or directory entity to determine if the entity was constructed using the `createTemporary` function.

RELATED CONCEPTS

“File Systems” on page 1

“File System Entities” on page 1

“Path Names and Path Name Parsers” on page 10

“File System Exceptions” on page 22

RELATED TASKS

“Instantiate the File System Classes” on page 3

“Access File Contents” on page 14

“Access Directory and Volume Contents” on page 15
“Copy and Move Files and Directories” on page 18
“Customize File System Operations” on page 20
“Get and Set Information about File System Entities”
“Work with Path Names” on page 11

Get and Set Information about File System Entities

Get a Path Name from a File System Entity

To get the full path name from an `IFileSystemEntity` object, call its `path` function:

```
IPathName pathName = anEntity.path(); // get full path of entity
```

To get the full host-specific path name for the entity, call the `hostPath` function. You can pass in an optional path name parser to use in constructing the host-specific path name. For example, if you specifically want a host-specific path name in Windows[®] format, pass in an instance of `IWin32PathNameParser`. If you do not explicitly specify a path name parser, you will get the parser that matches the path format of the host operating system.

```
IHostPathName hostPathName(anEntity.hostPath());  
cout << "Path name in host path format is " << hostPathName;  
IHostPathName winPathName(anEntity.hostPath(IWin32PathNameParser()));  
cout << "Path name in Windows path format is " << winPathName;
```

Access the Attributes of Files, Directories and Volumes

An attribute common to all types of entities is the entity name. To get and set the name of an entity call the `name` and `setName` functions:

```
IFilename entityName = anEntity.name(); // get the name  
anEntity.setName(IFilename("newname")); // set the name
```

Note that `IFilename` is a typedef for `IText`.

Another attribute common to all types of entities is the time of the most recent modification to the physical entity. You can retrieve the modification timestamp of the physical entity by calling `modificationTime`:

```
ITimeStamp modTime = anEntity.modificationTime();
```

You can also determine if a single file system entity object was created with either the `IFile::createTemporary` function or `IVolume::createTemporary` function. This does not check to see whether the physical object resides in any particular system temporary directory; it returns true if the entity instance was specifically created using one of those two static functions.

```
if (anEntity.temporary()) {  
    // it's a temporary entity  
}
```

Rename a File System Entity

To rename an entity, call the `setName` function:

```
anEntity.setName(newName);
```

You can also rename an entity by moving it.

Compare File System Entities

`IFileSystemEntity`'s equality (`==`) and inequality operators (`!=`) test whether two entity objects point to the same physical object on the disk. Two objects that refer to the same physical object will compare equal, even if they were constructed from different path names:

```

// construct a pathname relative to current dir
IHostPathName name1("data\\settings.zzz");
// construct an absolute pathname
IHostPathName name2("/program/data/settings.zzz");
// set the current directory
IDirectory::setCurrent(IDirectory("/program",
    IUnixPathNameParser());
// Compare the file entities
IFile file1(name1, IFile::kMustExist, IWin32PathNameParser());
IFile file2(name2, IFile::kMustExist, IUnixPathNameParser());
if (file1 != file2) {
    // major problem
}

```

Test the Identity of a File System Entity

You can test whether an entity is an instance of a particular class with the function `IFileSystemEntity::isA`. Pass an `IFileKind` representing the desired class, and `isA` returns true if the entity belongs to that class. Each entity class declares a constant named `kKind` that represents the class type.

As an example, the following code will print out the names of all of the directories that reside inside `myDirectory`.

```

for (IDirectoryIterator anEntity(myDirectory); anEntity; anEntity++) {
    if (anEntity.isA(IDirectory::kKind)) {
        cout << anEntity->name() << endl;
    }
}

```

Test the Validity of a File System Entity

You can call a file system entity's `valid` function to determine if an entity instance represents a real file system object:

```

if (IFile(IPathName("example")).valid()) {
    cout << "the file exists";
}

```

The `valid` function returns false after you call `deleteSelf` or any other function that deletes the physical object or invalidates the entity:

```

IFile aFile(IPathName("example"));
aFile.deleteSelf();
// the following statement will print "invalid"
cout << "'example' is " <<
    (aFile.valid() ? "valid" : "invalid");

```

The `valid` function also returns false for entity instances created using the default constructor, because they do not refer to any physical object until assigned a good value:

```

IFile someFile;
// The following statement will print "invalid"
// because someFile doesn't refer to a physical object.
cout << "someFile is " <<
    (someFile.valid() ? "valid" : "invalid");
someFile = IFile(IPathName("goodfile"));
// The following statement will print "valid" if the
// file 'goodfile' exists.
cout << "someFile is " <<
    (someFile.valid() ? "valid" : "invalid");

```

However, the `valid` function will still return true if something else, such as another process, deletes the physical object to which the entity instance points. If your program is running in a multithreaded situation where there is a possibility that

another thread or process has deleted a file, call the `connected` function to determine if an entity's physical file system object still exists:

```
if (aDirectory.connected()) {  
    // directory is still here  
}
```

The `connected` function goes out to the disk or network to verify that the entity still exists. It can be considerably slower than calling the `valid` function.

Determine the State of a Volume

`IVolume` provides several state access functions which you can use to get information about a volume. Use the `online`, `remote` and `removable` functions to find out whether a volume is online or offline, whether it is remote or local, and whether or not it is a removable-media volume:

```
IVolume volume(IHostPathName("C:"),  
    IWin32PathNameParser());  
IVolume::EState online = volume.online();  
IVolume::EState remote = volume.remote();  
IVolume::EState removable = volume.removable();
```

These functions return one of three values. `IVolume::EState` defines three values: `kFalse`, `kTrue` and `kUnknown`. You can use the state value returned by any of the above functions as if the state value was boolean, but the unknown status will be treated as true. This is the more conservative assumption — if you don't know whether a volume is remote, your code will assume that it is.

Determine the Amount of Free Space on a Volume

Call the `freeSpace` function to determine the amount of space available on a particular volume:

```
IFileSize freeSpace = aVolume.freeSpace();
```

`IFileSize` is a class that represents a 64-bit signed integer. You can perform all the standard mathematical operations on an `IFileSize` object, as if it were an integer value.

You can also find out the total storage space for a particular volume by calling the `totalSpace` function. For example, you could calculate the percentage of total storage space used on a device:

```
IFileSize percentUsed =  
    aVolume.freeSpace() / aVolume.totalSpace();
```

RELATED CONCEPTS

- "File Systems" on page 1
- "File System Entities" on page 1
- "Path Names and Path Name Parsers" on page 10
- "File System Exceptions" on page 22

RELATED TASKS

- "Instantiate the File System Classes" on page 3
- "Create and Delete Files and Directories" on page 5
- "Access File Contents" on page 14
- "Access Directory and Volume Contents" on page 15
- "Copy and Move Files and Directories" on page 18
- "Customize File System Operations" on page 20
- "Work with Path Names" on page 11

Path Names and Path Name Parsers

Overview of Path Names and Path Name Parsers

A path name specifies the physical location of an object on a mounted volume. The File System Framework provides classes which enable you to specify path names in a portable manner, or in a host-specific way. You use path names to:

- Create a file system entity
- Retrieve a file system entity based on a path name
- Get a path name based on the file system entity
- Store and manipulate path names across platforms in a portable manner

The classes associated with path names and path name parsers are as follows:




- ICompoundName
- IPathName
- ICompoundNameParser
- IUnixPathNameParser
- IWin32PathNameParser

You can obtain a file system entity by providing the path name of the entity you want, and you can get the path name of an entity by providing the file system entity itself. The file system gives you two ways to use path names with file system entities:

- You can store path names as strings in the format used by a particular operating system.
- You can store and manipulate path names in a portable manner.

Host-Specific Path Names

Host-specific paths are simple strings that store path names in the format used by a particular operating system. Some examples of host-specific path names are as follows:

Operating System	Example Path Name
	/tmp/file.txt files/new/readme.txt
 	C:\temp\file.txt files\new\readme.txt

Path Name Parsers

Host-specific path names are not portable. For example, if you try to use a Windows-style path name on a UNIX system, you will get an exception. The File System Framework provides a way to avoid this problem. All methods that take a host-specific path as a parameter also take an optional ICompoundNameParser parameter, which you can use to specify a parser for the path name. If you do not explicitly specify a parser, you will get the parser that matches the path format of the host operating system:

The path name parsers provided in this release of Open Class are as follows

Class	Description
IUnixPathNameParser	A concrete class that parses and formats UNIX-style file system path names.
IWin32PathNameParser	A concrete class that parses and formats Windows-style file system path names.

Portable Path Names

Although host-specific path names are easy to use, you sometimes need to store or manipulate a path name in a portable manner. For example, a text document might need to include the relative path to an embedded graphic file. The class `IPathName` provides this portable functionality.

`IPathName`, derived from `ICompoundName`, represents full or partial paths to file system entities. It allows you to specify file system entities via their absolute or relative path names.

You can also construct an `IPathName` from a `IDirectory` instance and a partial path relative to that directory. The partial path can be specified either as an `IHostPathName` or as another `IPathName`.

RELATED CONCEPTS

"File Systems" on page 1

RELATED TASKS

"Instantiate the File System Classes" on page 3

"Work with Path Names"

"Create and Delete Files and Directories" on page 5

"Access File Contents" on page 14

"Access Directory and Volume Contents" on page 15

"Copy and Move Files and Directories" on page 18

"Get and Set Information about File System Entities" on page 7

Work with Path Names

Instantiate Host-Specific Path Names and Path Name Parsers

Use instances of `IHostPathName`, together with an appropriate path name parser, to represent file, directory or volume path names in the path name format of a particular operating system. The following example constructs an `IHostPathName` object from a UNIX path name and then constructs an `IDirectory` object using that host-specific path name:

```
IHostPathName unixPath("/usr/local/bin");  
IDirectory localBin(unixPath, IUnixPathNameParser());
```

Instantiate a Portable Path Name from a Host-Specific Path Name

Although host-specific path names are easy to use, you sometimes need to store or manipulate a path name in a portable manner. The class `IPathName`, derived from `ICompoundName`, provides this portable functionality. `IPathName` represents full or partial paths to file system entities and allows you to specify file system entities via their absolute or relative path names.

You can instantiate `IPathName` directly from a host-specific path name by specifying an path name parser or by using the system's default parser:

```
// specify a parser
IPathName aPath("abc/xyz", IUnixPathNameParser());
// use the default parser
IPathName aPath("abc/xyz");
```

You can also construct an IPathName from a IDirectory instance and a partial path relative to that directory. The partial path can be specified either as an IHostPathName or as another IPathName:

```
// partial path is IHostPathName
IPathName aPath(aDirectory, "abc/xyz", IUnixPathNameParser());
// partial path is IPathName
IPathName aPathName("abc/xyz", IUnixPathNameParser());
IPathName aPath(aDirectory, aPathName);
```

Append and Retrieve Path Name Components

To append all the components from one path onto the end of another, use the append function. After this code executes, xyz will contain the path one/two/three/four/five:

```
IPathName xyz("one/two/three", IUnixPathNameParser());
IPathName abc("four/five");
xyz.append(abc);
```

To append a single component to a path name, use the appendComponent function. After this code executes, abc will contain the path one/two/three:

```
IPathName abc(IHostPathName("one/two"),
    IUnixPathNameParser());
abc.appendComponent(IText("three"));
```

To retrieve the individual components of a path name, use the functions componentAt and lastComponent:

```
IPathName abc("one/two/three", IUnixPathNameParser());
// will return "two"
IText compTwo = abc.componentAt(2);
// will return "three"
IText lastComp = abc.lastComponent();
```

Indices start at one, just as in the Collections classes.

RELATED CONCEPTS

- “File Systems” on page 1
- “File System Entities” on page 1
- “Path Names and Path Name Parsers” on page 10
- “File System Exceptions” on page 22

RELATED TASKS

- “Instantiate the File System Classes” on page 3
- “Create and Delete Files and Directories” on page 5
- “Access File Contents” on page 14
- “Access Directory and Volume Contents” on page 15
- “Copy and Move Files and Directories” on page 18
- “Customize File System Operations” on page 20
- “Get and Set Information about File System Entities” on page 7

Data Accessors

Data access can be provided through the non-ANSI I/O Stream Library or through the ANSI C++ Stream classes. To use the ANSI C++ Stream classes you must define the `__IOC_ANSI_STREAM` macro. By default, IBM Open Class Streaming classes uses the non-ANSI I/O Stream Library.

`IFileIOStream` is a stream class that can be constructed from an `IFile`. It allows `iostream`-based code to interoperate with the file system API. To perform I/O operations, you use the functions inherited from `iostream`. Use `IFileIOStream` when you need compatibility with the standard `iostream` classes or access to legacy data that is stored in files. You use this class almost exactly the same way you would the standard `fstream` class, except that the constructor accepts an `IFile` argument.

`IDataStream` is the Open Class stream for C++ objects. Use `IDataStream` to store persistent objects, Open Class documents, and other object-based data.

The constructors for both of the stream classes take two `int` arguments, `modeForMe` and `modeForOthers`, that determine how the file is opened. The first argument specifies the type of access that you want to have to the stream yourself, plus any special behavior that you want when the file is opened. The second argument specifies the access that you want to allow for others who try to open the file when you already have it open.

The `int` values for both of these arguments are constructed by combining the `open_mode` constants from the `ios` class or the `EOpenMode` constants from the `IFile` class. The constants that specify access levels are as follows:

Constant	Description
0	Allows no access to the file. This is the default value of the <code>modeForOthers</code> argument.
in	Allows read access. The file can be read but no
out	Allows writing to the file.

There are a few other open-mode constants that specify special behavior when you open or write to the file. These constants have no effect if they are added into the `modeForOthers` argument.

Constant	Description
app	Causes a seek to the end immediately after the open, but not before each write.
ate	Causes a seek to the end immediately after the open, but not before each write.
trunc	Truncates an existing stream when opening it.

Both `IDataStream` and `IFileIOStream` keep track of the end of the file, also called end of stream. As you write out a file stream, the end-of-file marker moves forward. To move the stream marker more efficiently, use `IFileIOStream::seek` to move the marker forward to the position you want.

When you move the end-of-file marker forward, the results will depend on the host file system. Because your code might be used on different file systems, you need to code as though you risk running out of file space each time you move the end-of-file marker.

RELATED CONCEPTS

“File Systems” on page 1
IBM Open Class Streaming Classes
Data Streams
I/O Stream Classes

RELATED TASKS

Instantiate a Data Stream

Access File Contents

Set Stream Access Permissions

This example opens an existing file for shared reading and streams in the contents. The second `IFile::in` in the constructor’s argument list allows other processes to open the file for concurrent reading.

```
IFile theFile(pathName);
IDataStream* stream =
    theFile.createStream(IFile::in, IFile::in);
ISomeDataType* fileContents = NULL;
::readObject(fileContents, *stream);
delete stream;
```

Access File Contents Using C++ Streams

`IFileIOStream` allows you to access the data within a file and derives from the standard library class `iostream`, providing interoperability with C++ `iostreams`.

You can create an `IFileIOStream` using an `IFile` instance to specify the file whose contents you want to stream:

```
IFile aFile(pathName, IFile::kCreateIfNeeded);
IFileIOStream aStream(aFile);
aStream << "I'm writing this string to the stream." << endl;
```

You can also use a full or partial path name that specifies the file:

```
IFileIOStream aStream(hostPathName, modeForMe, modeForOthers,
    IUnixPathNameParser());
```

Access File Contents Using Open Class Streams

`IDataStream` is the standard Open Class stream for C++ objects. To access a file’s contents with an `IDataStream` object, call the file entity’s `createStream` function. The `createStream` function returns a pointer to an `IDataStream`. You are responsible for deleting the pointer when you are finished with the stream:

```
IFile aFile(pathName);
IDataStream* stream = aFile.createStream(IFile::in +
    IFile::out + IFile::trunc);
::writeObject(fileContents);
cout << "streaming out completed." << endl;
delete stream;
```

A file remains open until the stream’s `close` function is called or until the `IFileIOStream` instance is destroyed. If you allocate one of these instances on the heap, you take responsibility for destroying the instance (and thus closing the file) when you are done.

RELATED CONCEPTS

- “File System Entities” on page 1
- “Path Names and Path Name Parsers” on page 10
- “Data Accessors” on page 13
- “File System Exceptions” on page 22

RELATED TASKS

- “Instantiate the File System Classes” on page 3
- “Create and Delete Files and Directories” on page 5
- “Access Directory and Volume Contents”
- “Copy and Move Files and Directories” on page 18
- “Customize File System Operations” on page 20
- “Get and Set Information about File System Entities” on page 7
- “Work with Path Names” on page 11

File System Iterators

The File System Framework provides three classes that you can use for iterating through directories and volumes:

- `IDirectoryIterator` iterates over the files and directories contained within a single directory or an entire directory hierarchy.
- `IRootDirectoryIterator` iterates over all topmost directories on all mounted volumes.
- `IVolumeIterator` iterates over all the local disks and network volumes that are currently mounted.

► OS/2

WIN

`IVolumeIterator` returns an `IVolume` for each disk connected to the system and for each connected network drive. Additionally, because each volume is represented as a root directory, `IRootDirectoryIterator` will return the same entities as `IVolumeIterator`.

► AIX

There is only one root directory (“/”) on UNIX systems.

RELATED CONCEPTS

- “File Systems” on page 1

RELATED TASKS

- “Access Directory and Volume Contents”
- “Work with Path Names” on page 11

Access Directory and Volume Contents

Look Up Entities with the `IDirectory::lookUp` Function

`IDirectory` provides the lookup function `lookUp`. This function will search in the directory for an entity with a specified name. If you pass a simple file name to `lookUp`, it will look for an entity with that name in the physical directory represented by the `IDirectory` object:

```
File myFile = aDirectory.lookUp("filename");
```

If you know the name of an entity and the directory in which it is located, use the method `IDirectory::lookUp` to find it. For example, to find a directory named `abc` inside a directory that is specified by the variable `aDirectory`, use this code:

```
IDirectory myDirectory = aDirectory.lookUp("abc");
```

If you pass in a partial path, `lookUp` will look for an entity in the specified subdirectory of the `IDirectory`. The following code will look for a subdirectory named `direct` and then for a file inside it named `filename`.

```
IFile aFile = dir.lookUp("direct/filename",  
    IUnixPathNameParser());
```

Look Up Entities with a Directory Iterator

Another way to find an entity is to search through an entire directory hierarchy or through all of the entities in a particular directory. This type of search is performed with *iterators*.

The following example searches through the directory tree under `aDirectory` for a file named "filename". Passing `true` in the second argument to the `IDirectoryIterator` makes the iteration recursive.

```
IDirectory aDirectory(...);  
...  
IFile myFile;  
for (IDirectoryIterator anEntity(aDirectory, true);  
    anEntity;  
    anEntity++)  
{  
    if (anEntity->isA(IFile::kKind) &&  
        anEntity->name() == "filename")  
    {  
        myFile = *anEntity;  
        break;  
    }  
}
```

Iterate over the Contents of a Directory

`IDirectoryIterator` iterates over the contents of a single directory or an entire directory tree. The `bool` recursive argument to the constructor determines whether or not the iteration is recursive. Because directories can contain other directories as well as files, the results of iteration are often heterogeneous. The following simple example will print the names of all files in the current directory:

```
for (IDirectoryIterator anEntity(IDirectory::current());  
    anEntity;  
    anEntity++)  
{  
    if (anEntity->isA(IFile::kKind)) {  
        cout << anEntity->name() << endl;  
    }  
}
```

Iterate over the Contents of a Volume

`IVolume` derives from `IDirectory`, so you can iterate over all the directories on a particular volume by creating an `IDirectoryIterator` object to which you pass an `IVolume` object during construction. Passing `true` in the second argument to the `IDirectoryIterator` constructor makes the iteration recursive.

```
IVolume theVolume(IHostPathName("/"),  
    IUnixPathNameParser());  
for (IDirectoryIterator anEntity(theVolume, true);  
    anEntity;
```

```

    anEntity++)
{
    processEntity(*anEntity);
}

```

Iterate over all Mounted Volumes

You can iterate over all of the volumes mounted on a system. Instantiate and use an `IVolumeIterator`:

```

for (IVolumeIterator aVolume; aVolume; aVolume++) {
    processEntity(*aVolume);
}

```

Iterate over all Root Directories

You can also search through all of the topmost, or “parentless” directories, on the system. These directories are called root directories. `IRootDirectoryIterator` iterates over all root directories on all mounted volumes and returns an `IDirectory` for the top of each separate directory hierarchy on the system. For example, on a UNIX host there is only one root directory (named “/”) for the entire system.

OS/2

WIN

On a Windows or an OS/2[®] system, there is a one-to-one correspondence between root directories and volumes, so volume iterators and root-directory iterators return the same entities.

The following code will call the external function `processEntity` for every root directory on the system.

```

for (IRootDirectoryIterator aRoot; aRoot; aRoot++) {
    processEntity(*aRoot);
}

```

RELATED CONCEPTS

[“File Systems” on page 1](#)
[“File System Entities” on page 1](#)
[“Path Names and Path Name Parsers” on page 10](#)
[“File System Iterators” on page 15](#)
[“File System Exceptions” on page 22](#)

RELATED TASKS

[“Instantiate the File System Classes” on page 3](#)
[“Create and Delete Files and Directories” on page 5](#)
[“Access File Contents” on page 14](#)
[“Copy and Move Files and Directories” on page 18](#)
[“Customize File System Operations” on page 20](#)
[“Get and Set Information about File System Entities” on page 7](#)
[“Work with Path Names” on page 11](#)

File System Movers and Copiers

`IFileSystemEntity`'s `moveTo` and `copyTo` Functions

The `IFileSystemEntity` class provides functions that copy or move a file or directory into a specified destination directory:

- The `copyTo` function copies an entity from its current location to another location

- The `moveTo` function moves an entity from its current location to another location

These functions are not meant to be customized.

Mover and Copier Classes

The File System Framework provides the following classes which you can subclass if you need customized copy or move behavior:

- `IFileOperation` is an abstract base class for classes that operate recursively over a tree of file system entities. It is the parent of `IFileSystemMover` and `IFileSystemCopier`.
- `IFileSystemMover` moves files and directories from one `IDirectory` instance to another, recursively moving all of an entity's children.
- `IFileSystemCopier` copies files and directories between `IDirectory` instances, recursively copying all of an entity's children.

These classes provide the infrastructure for progress indicators, custom failure handling, and resolution of file-naming conflicts. You can instantiate and use `IFileSystemMover` and `IFileSystemCopier` directly, or subclass them in order to implement progress reporting, name-conflict resolution and failure handling. You can derive from `IFileSystemCopier` or `IFileSystemMover` to handle failures, resolve name conflicts, or report progress. You can also create an entirely new file operation by subclassing from `IFileOperation`. For example, you could create a subclass of `IFileOperation` that adds up the total size of all files in a directory tree, or searches through all of the files for a particular piece of data.

RELATED CONCEPTS

"File Systems" on page 1

RELATED TASKS

"Copy and Move Files and Directories"

"Customize File System Operations" on page 20

Copy and Move Files and Directories

Copy a File or Directory Using `IFileSystemEntity::copyTo`

`IFileSystemEntity` provides the `copyTo` function, inherited by `IFile` and `IDirectory`, which you can use to copy an entity from one parent directory to another:

```
IDirectory destinationDir = ...;
aFile.copyTo(destinationDir);
aDirectory.copyTo(destinationDir);
```

You can specify a new name for the copy:

```
aFile.copyTo(destinationDir, newName);
aDirectory.copyTo(destinationDir, IFilename(""));
```

If the new name you provide is empty, the original entity's name is used.

If you are copying a directory tree, you can control the behavior when an error occurs. You have two choices for error handling, as defined by the `IFileSystemEntity::EFailureAction` enum:

- `kStop` causes the exception to be thrown immediately. This is the default.
- `kContinue` attempts to complete the operation on all remaining entities, and then throws the exception encountered.

For example:

```
aDirectory.copyTo(destinationDir, IFileName(""), IDirectory::kContinue);
```

Since files do not have children, the `kContinue` flag has no effect if an error occurs while copying a file.

Move a File or Directory Using `IFFileSystemEntity::moveTo`

`IFFileSystemEntity` provides the `moveTo` function, inherited by `IFile` and `IDirectory`, which you can use to move an entity from one parent directory to another:

```
IDirectory destinationDir = ...;
aFile.moveTo(destinationDir);
aDirectory.moveTo(destinationDir);
```

If you want to move a physical entity to a different name in the same directory (rename it), you can use the `setName` function.

You can specify a new name for the directory at the same time you move it:

```
aFile.moveTo(destinationDir, newName);
aDirectory.moveTo(destinationDir, IFileName(""));
```

If the new name you provide is empty, the original entity's name is used.

If you are moving a directory tree, you can control the behavior when an error occurs in the same manner as `IFFileSystemEntity::copyTo`. For example:

```
aDirectory.moveTo(destinationDir, IFileName(""), IDirectory::kContinue);
```

Since files do not have children, the `kContinue` flag has no effect if an error occurs while moving a file.

When you call `moveTo` on an entity instance, the instance will be updated to point to the entity's new physical location on disk. However, any other instances that pointed to the original physical entity will not be updated.

Use the `IFFileSystemCopier` and `IFFileSystemMover` Classes Directly

You can use the `IFFileSystemCopier` and the `IFFileSystemMover` utility classes to copy and move file system entities. Simply instantiate the appropriate object, specifying whether to stop or keep going when a failure occurs, and then call its `move` or `copy` function:

```
IFFileSystemCopier copier(IFFileSystemCopier::kContinue);
copier.copy(aFile, destinationDir);

IFFileSystemMover mover(IFFileSystemMover::kStop);
mover.move(aFile, destinationDir, newName);
mover.move(aDir, destinationDir);
```

RELATED CONCEPTS

"File Systems" on page 1

"File System Entities" on page 1

"File System Movers and Copiers" on page 17 "File System Exceptions" on page 22

"File System Movers and Copiers" on page 17

RELATED TASKS

"Instantiate the File System Classes" on page 3

"Create and Delete Files and Directories" on page 5

"Access File Contents" on page 14

"Access Directory and Volume Contents" on page 15

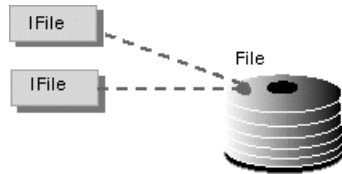
“Customize File System Operations”

“Get and Set Information about File System Entities” on page 7

“Work with Path Names” on page 11

Thread Safety

A surrogate instance (a file system entity) is only a handle to a physical entity. Multiple file system entities can point to a single physical entity.



The classes that comprise the File System interface are not multithread-safe. You cannot share individual file system entity instances. Even though the entity instances themselves are not thread-safe, however, the underlying physical objects they represent usually are. You can make copies of an instance and simultaneously use the original and each copy in a different thread.

Copies of file system instances are very inexpensive to construct, and the File System interface is designed to ensure that the implementation of copy semantics is efficient.

Note that if the files themselves are not thread-safe—for example, if there are two unsynchronized threads writing to a file at once—making copies of the entity instances will not help.

RELATED CONCEPTS

“File Systems” on page 1

Open Class Threading Model

RELATED TASKS

“Instantiate the File System Classes” on page 3

“Access File Contents” on page 14

Customize File System Operations

Support Progress Indicators

You will typically customize move or copy operations rather than create your own file operation. See [Create Your Own File Operation](#) below for information about subclassing `IFileOperation` to create an entirely new file system operation.

You can support progress indicators by subclassing the appropriate file system operation class and overriding `reportProgress`. Return true if the operation should continue or false if it should terminate. In the following example, the file system copier `IMyCopier` prints a message to cout each time `reportProgress` is called. The definitions of trivial functions such as constructors and destructors have been omitted.

```
class IMyCopier : public IFileSystemCopier {
public:
    IMyCopier(EFailureAction action,
```

```

        unsigned int tellMeTimes);
protected:
    virtual bool reportProgress(double fractionDone,
                               const IFileSystemEntity& currentEntity);
};
bool
IMyCopier::reportProgress(
    double fractionDone,
    const IFileSystemEntity& currentEntity)
{
    cout << fractionDone << "% complete\n";
    return !terminatedByUser();
}

```

The `tellMeTimes` parameter to the `IMyCopier` constructor is passed to the base class `IFileSystemCopier`. This parameter specifies the number of times that `reportProgress` should be called during the operation. For example, if you want to display a “% copied” indicator, pass 100 in this parameter. If you have a progress indicator that is 210 pixels wide, pass 210 in this parameter.

Resolve File Name Conflicts

You can implement file name-conflict resolution during move or copy operations by subclassing a file operation class, such as `IFileSystemCopier` or `IFileSystemMover`, and overriding `renameNeeded`. The following example is a custom `IFileSystemMover` subclass, `IMyMover`, that calls the external function `GetNewName` to prompt the user for a new file name. It then returns true to indicate that a new name was entered and the file operation should continue, or false to indicate that a new name was not entered and the operation must stop. The definitions of trivial functions such as constructors and destructors have been omitted.

```

class IMyMover : public IFileSystemMover {
public:
    IMyMover(EFailureAction action = kContinue,
             unsigned int tellMeTimes = 0);
protected:
    virtual bool renameNeeded(IFileName& modifyThisName,
                              const IFileSystemEntity& currentEntity);
};
bool
IMyMover::renameNeeded(
    IFileName& modifyThisName,
    const IFileSystemEntity& currentEntity)
{
    bool gotNewName = GetNewName(currentEntity.name(),
                                  modifyThisName);
    return gotNewName;
}

```

Handle Exceptions and Other Failures

Any exceptions other than name-conflict errors are passed back to the file operation object for handling. The default implementation rethrows the exception. You can subclass a file operation class and override `handleFailure` to handle any exceptions that occur during the execution of your file operation:

```

class IMyCopier : public IFileSystemCopier {
public:
    IMyCopier(EFailureAction action = kStop,
              unsigned int tellMeTimes = 0);
protected:
    virtual bool handleFailure(IException& reason,
                               const IFileSystemEntity& currentEntity);
};
bool

```

```

IMyCopier::handleFailure(
    IException& reason,
    const IFileSystemEntity& currentEntity)
{
    // display an alert window
    return errorHandledSuccessfully;
}

```

Create Your Own File System Operation

You will not normally need to create an entirely new type of file system operation, but it is easy to do. Subclass IFileOperation and override the pure virtual function doOneEntity:

```

class IMyOperation : public IFileOperation {
public:
    IMyOperation(EFailureAction action,
        unsigned int tellMeTimes);
protected:
    virtual IFileSystemEntity
        doOneEntity(const IFileSystemEntity& entity,
            IDirectory& target,
            const IFileName& newName);
};
IFileSystemEntity
IMyOperation::doOneEntity(
    const IFileSystemEntity& entity,
    IDirectory& target,
    const IFileName& newName)
{
    // perform operation here and return appropriate
    // IFileSystemEntity
}

```

RELATED CONCEPTS

["File Systems" on page 1](#)
["File System Entities" on page 1](#)
["File System Movers and Copiers" on page 17](#)
["File System Exceptions"](#)

RELATED TASKS

["Instantiate the File System Classes" on page 3](#)
["Create and Delete Files and Directories" on page 5](#)
["Access File Contents" on page 14](#)
["Access Directory and Volume Contents" on page 15](#)
["Copy and Move Files and Directories" on page 18](#)
["Get and Set Information about File System Entities" on page 7](#)
["Work with Path Names" on page 11](#)

File System Exceptions

All file system exceptions derive directly or indirectly from IException. All of the standard IException behavior will work with these classes:

Class	Description
IEntityInUse	Indicates an attempt to move, delete, or otherwise manipulate an entity that is currently in use.

Class	Description
IEntityInvalid	Indicates an attempt to perform an operation on a file system entity that does not point to a valid on-disk entity.
IEntityTypeMismatch	Indicates an attempt to assign the wrong type of file system entity to another entity subclass.
IFileOperationCancelled	Indicates that the user canceled a file system operation.
IVolumeOffline	Indicates an attempt to perform an operation on a volume that is off line.

RELATED CONCEPTS

“File Systems” on page 1
 Exceptions in the IBM Open Class

RELATED TASKS

“Instantiate the File System Classes” on page 3
 “Create and Delete Files and Directories” on page 5
 “Access File Contents” on page 14
 “Access Directory and Volume Contents” on page 15
 “Copy and Move Files and Directories” on page 18