Using %Library.File

Version 2024.1
2024-05-02

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using %Library.File
InterSystems IRIS Data Platform Version 2024.1 2024-05-02
Copyright © 2024 InterSystems Corporation
All rights reserved.

InterSystems®, HealthShare Care Community®, HealthShare Unified Care Record®, IntegratedML®, InterSystems Caché®, InterSystems Ensemble® InterSystems HealthShare®, InterSystems IRIS®, and TrakCare are registered trademarks of InterSystems Corporation. HealthShare® CMS Solution Pack™ HealthShare® Health Connect Cloud™, InterSystems IRIS for Health™, InterSystems Supply Chain Orchestrator™, and InterSystems TotalView™ For Asset Management are trademarks of InterSystems Corporation. TrakCare is a registered trademark in Australia and the European Union.

All other brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Memorial Drive, Cambridge, MA 02142, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation. InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION, COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion and without notice, to make substitutions and modifications in the products and practices described in this document.

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

1

What Is %Library.File?

The %Library.File class (%File for short) provides an extensive API for working with files and directories. This document describes the main features of this API, for example, listing directory and drive contents; creating, copying, and deleting directories and files; setting and getting file attributes; and reading and writing files. For a canonical list of properties, methods, and queries, see the class reference.

If you specify a partial filename or directory name when using %File, most of its methods assume that you are referring to an item relative to the directory that contains the default globals database for the namespace you are working in. This directory is referred to in this document as the “default directory.” Any exceptions to this rule are noted where applicable.

Also, %File treats the file or directory name as case-sensitive only if the underlying operating system treats file and directory names as case-sensitive. That is, file or directory names are case-sensitive on Unix but not case-sensitive on Windows.

2

Query Directories and Drives

The %Library.File class provides class queries that can query drives and directories.

2.1 List the Contents of a Directory

The **FileSet()** class query lists the contents of a directory. This query accepts the following parameters, in order:

1. *directory* — Specifies the name of the directory to examine.
2. *wildcards* — Specifies the filename pattern to match, if any. For details, see the section "Wildcards" in the reference for $ZSEARCH.
3. *sortby* — Specifies how to sort the results. Use one of the following values:
   - `Name` — Name of the file (the default)
   - `Type` — Item type
   - `DateCreated` — Date and time when the file was created
   - `DateModified` — Date and time when the file was last modified
   - `Size` — File size
4. *includedirs* — Specifies how to handle directories within the given directory. If this argument is true (1), the query returns all directories before any files, and the directory names ignore the *wildcards* argument. If this argument is false (0), the *wildcards* argument applies to both files and directories. The default is 0.
5. *delimiter* — Specifies the delimiter between wildcards in the *wildcards* argument. The default is `;`

The result set returned by this query provides the following fields:

- Name — Full pathname of the item.
- Type — Type of the item: `F` indicates a file, `D` indicates a directory, and `S` indicates a symbolic link.
- Size — File size, in bytes. This field is null for directories and symbolic links.
- DateCreated — Date and time, in the format yyyy-mm-dd hh:mm:ss, when the item was created.
- DateModified — Date and time, in the format yyyy-mm-dd hh:mm:ss, when the item was last modified.
- ItemName — Short name of the item. For a file, this is the filename alone, without the directory. For a directory, this is just the last part of the directory path.

Note: Windows is the only platform that currently tracks the actual created date. Other platforms store the date of the last file status change.

Here is a simple example that uses this class query:

Class Member

```
ClassMethod ShowDir(dir As %String = "", wildcard As %String = "", sort As %String = "Name")
{
  set stmt = ##class(%SQL.Statement).%New()
  set status = stmt.%PrepareClassQuery("%File", "FileSet")
  if $$$ISERR(status) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(status) quit}

  set rset = stmt.%Execute(dir, wildcard, sort)
  if (rset.%SQLCODE '= 0) {write "%Execute failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
 quit}

  while rset.%Next()
  {
    write !, rset.%Get("Name")
    write " ", rset.%Get("Type")
    write " ", rset.%Get("Size")
  }
  if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
quit}
}
```

Assuming the method is within the User.FileTest class, running this method from the Terminal on a specified directory, filtering for log files, and sorting by file size gives something like:

```
USER>do ##class(FileTest).ShowDir("C:\InterSystems\IRIS\mgr", "*.log", "Size")

C:\InterSystems\IRIS\mgr\alerts.log F 380
C:\InterSystems\IRIS\mgr\FeatureTracker.log F 730
C:\InterSystems\IRIS\mgr\journal.log F 743
C:\InterSystems\IRIS\mgr\ensinstall.log F 12577
C:\InterSystems\IRIS\mgr\iboot.log F 40124
C:\InterSystems\IRIS\mgr\SystemMonitor.log F 483865
C:\InterSystems\IRIS\mgr\messages.log F 4554535
```

For another example, the following method examines a directory and all its subdirectories, recursively, and writes out the name of each file that it finds:

Class Member

```
ClassMethod ShowFilesInDir(directory As %String = "")
{
  set stmt = ##class(%SQL.Statement).%New()
  set status = stmt.%PrepareClassQuery("%File", "FileSet")
  if $$$ISERR(status) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(status) quit}

  set rset = stmt.%Execute(directory)
  if (rset.%SQLCODE '= 0) {write "%Execute failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
 quit}

  while rset.%Next()
  {
    set name = rset.%Get("Name")
    set type = rset.%Get("Type")

    if (type = "F") {
      write !, name
     } elseif (type = "D"){
      do ..ShowFilesInDir(name)
    }
  }
  if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
quit}
}
```

Running this method in the Terminal on the default directory gives something like:

```
USER>do ##class(FileTest).ShowFilesInDir()

C:\InterSystems\IRIS\mgr\user\IRIS.DAT
C:\InterSystems\IRIS\mgr\user\iris.lck
C:\InterSystems\IRIS\mgr\user\userenstemp\IRIS.DAT
C:\InterSystems\IRIS\mgr\user\userenstemp\iris.lck
C:\InterSystems\IRIS\mgr\user\usersecondary\IRIS.DAT
C:\InterSystems\IRIS\mgr\user\usersecondary\iris.lck
```

2.2 List the Drives or Mounted File Systems

The **DriveList()** class query lists the available drives (on Windows) or the mounted file systems (on UNIX®). This query accepts one parameter:

1. *fullyqualified* — If this argument is 1, the query includes a trailing backslash on each Windows drive name. This argument has no effect on other platforms. The default is 0.

The result set returned by this query provides one field:

- Drive — Name of a drive (on Windows) or name of a mounted file system (on UNIX®).

The following example shows how you might use this query:

```
ClassMethod ShowDrives()
{
  set stmt = ##class(%SQL.Statement).%New()
  set status = stmt.%PrepareClassQuery("%File","DriveList")
  if $$$ISERR(status) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(status) quit}

  set rset = stmt.%Execute(1)
  if (rset.%SQLCODE '= 0) {write "%Execute failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
 quit}

  while rset.%Next()
  {
    write !, rset.%Get("Drive")
  }
  if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
quit}
}
```

Again assuming the method is within the User.FileTest class, running the method in the Terminal gives something like:

```
USER>do ##class(FileTest).ShowDrives()

c:\
x:\
```

3

File and Directory Properties and Attributes

The %Library.File class provides many class methods that you can use to obtain information about files and directories, or to view or set their properties and attributes.

Note: If you specify a partial filename or directory name, most of these methods assume that you are referring to an item relative to the directory that contains the default globals database for the namespace you are working in. This directory is referred here as the "default directory." Any exceptions to this rule are noted.

Also, these methods treat the file or directory name as case-sensitive only if the underlying operating system treats file and directory names as case-sensitive. That is, file or directory names are case-sensitive on UNIX® but not case-sensitive on Windows.

3.1 Check File and Directory Existence

To find out whether a given file exists, use the **Exists()** method and specify the filename as the argument. For example:

```
USER>write ##class(%File).Exists("C:\temp\test.html")
1
```

Similarly, to find out whether a given directory exists, use the **DirectoryExists()** method, and specify the directory as the argument. For example:

```
USER>write ##class(%File).DirectoryExists("C:\temp")
1
```

As noted earlier, these methods treat the file or directory name as case-sensitive on Unix but not case-sensitive on Windows. Also, if you specify a partial filename or directory name, the method assumes that you are referring to a file or directory relative to the directory that contains the default globals database for the namespace you are working in. For example:

```
USER>write ##class(%File).Exists("iris.dat")
1
```

3.2 View and Set File and Directory Permissions

The %Library.File class provides many class methods that you can use to view or set the permissions of a file or directory.

3.2.1 See If a File or Directory is Read-Only or Writeable

Given a file or directory name, the **ReadOnly()** method returns 1 if the file or directory is read-only and 0 otherwise:

```
USER>write ##class(%File).ReadOnly("export.xml")
1
USER>write ##class(%File).ReadOnly("C:\temp")
0
```

Similarly, given a file or directory name, the **Writeable()** method returns 1 if the file or directory is writeable and 0 otherwise:

```
USER>write ##class(%File).Writeable("export.xml")
0
USER>write ##class(%File).Writeable("C:\temp")
1
```

3.2.2 Make a File or Directory Read-Only or Writeable (Windows)

To make a file or directory on Windows read-only, use the **SetReadOnly()** method, which returns a boolean value to indicate success or failure. This method takes three arguments, the second of which is omitted in Windows. The first argument is the name of the file or directory. The third argument is an output argument. If negative, it contains the error code returned by the operating system in case the method fails.

In the example below, the call to **SetReadOnly()** successfully changes the file C:\temp\testplan.pdf to read-only.

```
USER>write ##class(%File).ReadOnly("C:\temp\testplan.pdf")
0
USER>write ##class(%File).SetReadOnly("C:\temp\testplan.pdf",,.return)
1
USER>write ##class(%File).ReadOnly("C:\temp\testplan.pdf")
1
```

In the example below, the call to **SetReadOnly()** fails with Windows system error code 5, which means “Access is denied.”

```
USER>write ##class(%File).SetReadOnly("C:\",,.return)
0
USER>write return
-5
```

To make a file or directory on Windows writeable, use the **SetWriteable()** method. This method takes the same three arguments, the second of which is again omitted in Windows.

```
USER>write ##class(%File).Writeable("export.xml")
0
USER>write ##class(%File).SetWriteable("export.xml",,.return)
1
USER>write ##class(%File).Writeable("export.xml")
1
```

3.2.3 Make a File or Directory Read-Only or Writeable (Unix)

On Unix, the methods **SetReadOnly()** and **SetWriteable()** can be also be used, but their behavior is somewhat different, due to the presence of the second parameter. For more information, see **%Library.File.SetReadOnly()** or **%Library.File.SetWriteable()** in the class reference.

However in Unix, you may want to specify different permissions for owner, group, and user. For finer control of file and directory permissions, see the section View or Set File and Directory Attributes.

3.3 View and Set File and Directory Attributes

To view or set the attributes of a file or directory at a more detailed level, use the **Attributes()** and **SetAttributes()** methods of %Library.File. File attributes are represented by a sequence of bits that are expressed collectively as an integer. The meaning of the individual bits depends on the underlying operating system.

For a full listing of attribute bits, see **%Library.File.Attributes()** in the class reference.

For tips on working with strings of attribute bits, see Manipulating Bitstrings Implemented as Integers.

3.3.1 View File and Directory Attributes

The **Attributes()** method of %Library.File expects the file or directory name as the argument and returns a sequence of attribute bits expressed as an integer.

The following examples were run on a Windows system:

```
USER>write ##class(%File).Attributes("iris.dat")
32
USER>write ##class(%File).Attributes("C:\temp")
16
USER>write ##class(%File).Attributes("secret.zip")
35
```

In the first example, 32 means that iris.dat is an archive file. In the second example, 16 means that C:\temp is a directory. In the third example, more than one bit is set, and 35 indicates that secret.zip is an archive (32) that is hidden (2) and read-only (1). Adding 32 + 2 + 1 = 35.

The following example was run on a Unix system:

```
write ##class(%File).Attributes("/home")
16877
```

In this example, 16877 means that /home is a directory (16384) with read (256), write (128), and execute (64) permission for owner; read (32) and execute (8) permission for group; and read (4) and execute (1) permission for others. Adding 16384 + 256 + 128 + 64 + 32 + 8 + 4 + 1 = 16877.

3.3.2 Set File and Directory Attributes

Conversely, the **SetAttributes()** method sets a file or directory's attributes (where possible) and returns a boolean value to indicate success or failure. This method takes three arguments. The first argument is the name of the file or directory. The second argument is an integer that represents the desired attributes you would like the file or directory to have. The third argument is an output argument. If negative, it contains the error code returned by the operating system in case the method fails.

The following example, on Windows, makes the file C:\temp\protectme.txt read-only by setting the 1 bit:

```
USER>write ##class(%File).Attributes("C:\temp\protectme.txt")
32
USER>write ##class(%File).SetAttributes("C:\temp\protectme.txt",33,.return)
1
USER>write ##class(%File).Attributes("C:\temp\protectme.txt")
33
```

The following example, on Unix, changes permissions on the file myfile in the default directory from 644 to full permissions (777):

```
USER>write ##class(%File).Attributes("myfile")
33188
USER>write ##class(%File).SetAttributes("myfile",33279,.return)
1
USER>write ##class(%File).Attributes("myfile")
33279
```

The desired attribute value is calculated by adding the value for a regular file (32768) with the masks for owner (448), group (56), and others (7).

3.4 View Other File and Directory Properties

Other class methods of %Library.File allow you to examine various other properties of files and directories.

The **GetFileDateCreated()** method returns the date a file or directory was created in **$H** format:

```
USER>write $zdate(##class(%File).GetFileDateCreated("stream"))
12/09/2019
```

Note: Windows is the only platform that currently tracks the actual created date. Other platforms store the date of the last file status change.

The **GetFileDateModified()** method returns the date a file or directory was modified in **$H** format:

```
USER>write $zdate(##class(%File).GetFileDateModified("iris.dat"))
08/20/2020
```

The **GetFileSize()** method returns the size of a file, in bytes:

```
USER>write ##class(%File).GetFileSize("export.xml")
2512
```

The **GetDirectorySpace()** method returns the amount of free space and total space in a drive or directory. The space can be returned in bytes, MB (the default), or GB, depending on the value of the fourth argument, which can be 0, 1, or 2. In this example, 2 indicates that the space is returned in GB:

```
USER>set status = ##class(%File).GetDirectorySpace("C:", .FreeSpace, .TotalSpace, 2)

USER>write FreeSpace
182.87
USER>write TotalSpace
952.89
```

On Windows, if you pass a directory name to this method, the amount of space returned is for the entire drive.

For the **GetDirectorySpace()** method, any error status returned is the operating system-level error. In the example below, Windows system error code 3 indicates “The system cannot find the path specified.”

```
USER>set status = ##class(%File).GetDirectorySpace("Q:", .FreeSpace, .TotalSpace, 2)

USER>do $system.Status.DisplayError(status)

ERROR #83: Error code = 3
```

4

File and Directory Names

The %Library.File class provides several class methods that you can use to work with file and directory names. In most cases, the files and directories do not need to exist in order to use these methods.

Note: If you specify a partial filename or directory name, most of these methods assume that you are referring to an item relative to the directory that contains the default globals database for the namespace you are working in. This directory is referred here as the "default directory." Any exceptions to this rule are noted.

Also, these methods treat the file or directory name as case-sensitive only if the underlying operating system treats file and directory names as case-sensitive. That is, file or directory names are case-sensitive on UNIX® but not case-sensitive on Windows.

4.1 Get File and Directory Names

The %Library.File class provides class methods that you can use to obtain parts of file and directory names.

Given a full pathname, use **GetDirectory()** and **GetFilename()** to get the directory and the short filename respectively. For this method, partial directory names are not permitted.

```
USER>set filename = "C:\temp\samples\sample.html"

USER>write ##class(%File).GetDirectory(filename)
C:\temp\samples\
USER>write ##class(%File).GetFilename(filename)
sample.html
```

Given a filename, use **CanonicalFilename()** to get the full path from the root:

```
USER>set filename = "iris.dat"

USER>write ##class(%File).CanonicalFilename(filename)
c:\intersystems\IRIS\mgr\user\iris.dat
USER>write ##class(%File).CanonicalFilename("foo.dat")
```

If the file cannot be opened, the **CanonicalFilename()** method returns an empty string.

Given a directory name, use **ComputeFullDBDir()** to construct the canonical form of the directory name.

```
USER>write ##class(%File).ComputeFullDBDir("foodirectory")
c:\intersystems\IRIS\mgr\user\foodirectory\
```

Given a directory name, use **GetDirectoryLength()** and **GetDirectoryPiece()** to get the number of pieces in the directory and a specific piece, respectively. Pieces can be delimited by slash (/) or backslash (\), depending on the operating system.

```
USER>set dir = "C:\temp\samples"

USER>write ##class(%File).GetDirectoryLength(dir)
3
USER>write ##class(%File).GetDirectoryPiece(dir,1)
C:
```

Given a filename or directory name, use **ParentDirectoryName()** to get the parent directory.

```
USER>set dir = "stream"

USER>write ##class(%File).ParentDirectoryName(dir)
C:\InterSystems\IRIS\mgr\user\
```

4.2 Normalize File and Directory Names

The %Library.File class provides class methods that return normalized file and directory names (following the naming rules of the operating system on which the server is running). These are useful when you are creating new file and directory names by appending name pieces to existing names.

Given a filename, **NormalizeFilename()** returns the normalized filename.

Given a directory name, **NormalizeDirectory()** returns the normalized directory name.

The methods return normalized names that are appropriate for use on the underlying operating system and will attempt to normalize slash (/) or backslash (\) path delimiters.

Windows examples:

```
USER>set filename = "C:\temp//samples\myfile.txt"

USER>write ##class(%File).NormalizeFilename(filename)
C:\temp\samples\myfile.txt
USER>write ##class(%File).NormalizeDirectory("stream")
C:\InterSystems\IRIS\mgr\user\stream\
```

Unix examples:

```
USER>set filename = "/tmp//samples/myfile.txt"

USER>write ##class(%File).NormalizeFilename(filename)
/tmp/samples/myfile.txt
USER>write ##class(%File).NormalizeDirectory("stream")
/InterSystems/IRIS/mgr/user/stream/
```

Add a second argument when calling one of these methods to normalize the directory name or filename relative to a specified directory. The directory must exist.

Windows examples:

```
USER>write ##class(%File).NormalizeFilename("myfile.txt", "C:\temp\samples")
C:\temp\samples\myfile.txt
USER>write ##class(%File).NormalizeDirectory("stream", "")
C:\InterSystems\IRIS\mgr\user\stream\
```

Unix examples:

```
USER>write ##class(%File).NormalizeFilename("myfile.txt", "/tmp/samples")
/tmp/samples/myfile.txt
USER>write ##class(%File).NormalizeDirectory("stream", "")
/InterSystems/IRIS/mgr/user/stream/
```

The **SubDirectoryName()** method is similar to the two-argument form of **NormalizeDirectory()**, except the order of the arguments is reversed. Also the directory does not need to exist. Pass a `1` in the third argument to add a trailing delimiter, or a `0` to omit it (the default).

Windows examples:

```
USER>write ##class(%File).SubDirectoryName("C:\foobar", "samples")
C:\foobar\samples
USER>write ##class(%File).SubDirectoryName("", "stream", 1)
C:\InterSystems\IRIS\mgr\user\stream\
```

Unix examples:

```
USER>write ##class(%File).SubDirectoryName("/foobar", "samples")
/foobar/samples
USER>write ##class(%File).SubDirectoryName("", "stream", 1)
/InterSystems/IRIS/mgr/user/stream/
```

4.3 Handle File and Directory Names with Spaces

For file names and directory names that include spaces, use **NormalizeFilenameWithSpaces()**, which handles spaces in pathnames as appropriate to the host platform. Unlike **NormalizeFilename()** and **NormalizeDirectory()**, this method takes only one argument and cannot normalize a file or directory name relative to another directory, nor does it normalize partial file or directory names relative to the default directory.

On Windows systems, if the pathname contains spaces, and the file or directory does not exist, the method returns the pathname enclosed in double quotes. If the pathname contains spaces, and the file or directory does exist, the method returns the short form of the pathname. If the pathname does not contain spaces, the method returns the pathname unaltered.

```
USER>write ##class(%File).NormalizeFilenameWithSpaces("C:\temp\nonexistant folder")
"C:\temp\nonexistant folder"
USER>write ##class(%File).NormalizeFilenameWithSpaces("C:\temp\existant folder")
C:\temp\EXISTA~1
USER>write ##class(%File).NormalizeFilenameWithSpaces("iris.dat")
iris.dat
```

For further details, see **%Library.File.NormalizeFilenameWithSpaces()** in the class reference.

On Unix systems, if the pathname contains spaces, the method returns the pathname enclosed in double quotes. If the pathname does not contain spaces, the method returns the pathname unaltered.

```
USER>write ##class(%File).NormalizeFilenameWithSpaces("/InterSystems/my directory")
"/InterSystems/my directory"
USER>write ##class(%File).NormalizeFilenameWithSpaces("iris.dat")
iris.dat
```

4.4 Construct and Deconstruct File and Directory Names

The %Library.File class provides class methods that let you construct a filename from an array of paths or deconstruct a filename into an array of paths.

Given an array of paths, **Construct()** assembles the paths and returns the filename. The filename constructed is appropriate to the server platform. Calling this method without arguments returns the default directory.

Given a filename, **Deconstruct()** disassembles the filename and returns an array of paths. The contents of the array is appropriate to the server platform.

The following Windows example passes an array `dirs` to **Construct()**. The empty string in the last array location indicates that the filename returned should terminate in a `\`.

```
USER>zwrite dirs
dirs=4
dirs(1)="C:"
dirs(2)="Temp"
dirs(3)="samples"
dirs(4)=""
USER>write ##class(%File).Construct(dirs...)
C:\Temp\samples\
```

The following Unix example calls **Construct()** without arguments. The method returns the default directory.

```
USER>set default = ##class(%File).Construct()

USER>write default
/InterSystems/IRIS/mgr/user
```

The following Unix example calls **Deconstruct()**, which takes the paths in the variable `default` and stores them in array `defaultdir`.

```
USER>do ##class(%File).Deconstruct(default, .defaultdir)

USER>zwrite defaultdir
defaultdir=4
defaultdir(1)="InterSystems"
defaultdir(2)="IRIS"
defaultdir(3)="mgr"
defaultdir(4)="user"
```

4.5 Get the System Manager Directory

Use the **ManagerDirectory()** method to get the fully qualified name of the *installdir*/mgr directory. For example:

```
USER>write ##class(%File).ManagerDirectory()
C:\InterSystems\IRIS\mgr\
```

5

Work with Directories

The %Library.File class provides several class methods that allow you to perform various operations on directories.

Note: If you specify a partial filename or directory name, most of these methods assume that you are referring to an item relative to the directory that contains the default globals database for the namespace you are working in. This directory is referred here as the "default directory." Any exceptions to this rule are noted.

Also, these methods treat the file or directory name as case-sensitive only if the underlying operating system treats file and directory names as case-sensitive. That is, file or directory names are case-sensitive on UNIX® but not case-sensitive on Windows.

5.1 Create Directories

To create a directory, use the **CreateDirectory()** method, which returns a boolean value to indicate success or failure. This method takes two arguments. The first argument is the name of the directory to create. The second argument is an output argument. If negative, it contains the error code returned by the operating system in case the method fails.

If C:\temp already exists, the following command fails with Windows system error code 183, which means "Cannot create a file when that file already exists."

```
USER>write ##class(%File).CreateDirectory("C:\temp", .return)
0
USER>write return
-183
```

If C:\temp already exists, but C:\temp\test does not exist, the following command fails because **CreateDirectory()** creates, at most, the last directory in the given directory path. So the returned Windows system error code is 3, or "The system cannot find the path specified."

```
USER>write ##class(%File).CreateDirectory("C:\temp\test\this", .return)
0
USER>write return
-3
```

The following example succeeds with Windows system code 0, or "The operation completed successfully."

```
USER>write ##class(%File).CreateDirectory("C:\temp\test", .return)
1
USER>write return
0
```

The similar **CreateNewDir()** method creates a new directory within the specified parent directory. This method takes three arguments. The first argument is the name of the parent directory. The second argument is the name of the directory to create. The third argument is an output argument. If negative, it contains the error code returned by the operating system in case the method fails.

The first example, below, creates a directory called newdir in the parent directory C:\temp. The second example creates a new directory called newdir in the default directory.

```
USER>write ##class(%File).CreateNewDir("C:\temp", "newdir", .return)
1
USER>write ##class(%File).CreateNewDir("", "newdir", .return)
1
```

Another related method, **CreateDirectoryChain()**, creates all the directories on the given directory path (if possible).

The first example, below, creates three nested directories in the parent directory C:\temp. The second example creates three nested directories in the default directory.

```
USER>write ##class(%File).CreateDirectoryChain("C:\temp\one\two\three", .return)
1
USER>write ##class(%File).CreateDirectoryChain("one\two\three", .return)
1
```

5.2 Copy Directories

To copy a directory, use the **CopyDir()** method, which returns a boolean value to indicate success or failure.

This method takes five arguments:

1. *pSource* — Specifies the name of the source directory.
2. *pTarget* — Specifies the name of the target directory.
3. *pOverlay* — Specifies whether to overwrite the target directory, if it exists. The default is 0.
4. *pCreated* — Output argument that contains the number of files or directories created during the copy process.
5. *pDeleteBeforeCopy* — Specifies whether to delete any file that exists in the target directory before the copy is performed. The default is 0.

Partial directory names for either *pSource* or *pTarget* are calculated relative to the directory that contains the default globals database for the namespace you are working in.

Unlike the directory creation methods, **CopyDir()** does not have an output argument in which to return a system error code.

In the first example, below, the copy operation is successful, and 46 files and directories are copied from C:\temp to C:\temp2. In the second example, the copy operation is successful, and 46 files and directories are copied from C:\temp to a directory temp2 within the default directory.

```
USER>write ##class(%File).CopyDir("C:\temp", "C:\temp2", 0, .pCreated, 0)
1
USER>write pCreated
46
USER>write ##class(%File).CopyDir("C:\temp", "temp2", 0, .pCreated, 0)
1
USER>write pCreated
46
```

In the final example, below, *pOverlay* is set to 0, so the copy fails due to the target directory already existing.

```
USER>write ##class(%File).CopyDir("C:\temp", "C:\temp2", 0, .pCreated, 0)
0
USER>write pCreated
0
```

5.3 Delete Directories

To delete a non-empty directory, use the **RemoveDirectory()** method, which returns a 1 on success or 0 on failure. This method takes two arguments. The first argument is the name of the directory to remove. The second argument, which is an output argument, contains the error code returned by the operating system in case the method fails.

In the first example, below, the method succeeds. The second example fails with a Windows error code of 145, or "Directory not empty."

```
USER>write ##class(%File).RemoveDirectory("C:\temp2\newdir", .return)
1
USER>write ##class(%File).RemoveDirectory("C:\temp2", .return)
0
USER>write return
-145
```

To delete a directory, including any subdirectories, use the **RemoveDirectoryTree()** method, which returns a 1 on success or 0 on failure. Unlike the **RemoveDirectory()** method, **RemoveDirectoryTree()** does not have an output argument in which to return a system error code.

RemoveDirectoryTree() succeeds even if the directory and any subdirectories are not empty.

```
USER>write ##class(%File).RemoveDirectoryTree("C:\temp2")
1
```

5.4 Rename Directories

To rename a directory, use the **Rename()** method, which returns a 1 on success or 0 on failure. This method takes three arguments. The first argument is the name of the directory to rename, and the second is the new name. The third argument is an output argument. If negative, it contains the error code returned by the operating system in case the method fails.

Using **Rename()** to rename a directory works only if the directory is on the same file system as you are working on.

In the first example, below, the method succeeds. In the second example, C:\nodir does not exist, so the method fails with a Windows error code of 3, or "The system cannot find the path specified."

```
USER>write ##class(%File).Rename("C:\temp\oldname", "C:\temp\newname", .return)
1
USER>write ##class(%File).Rename("C:\nodir\oldname", "C:\nodir\newname", .return)
0
USER>write return
-3
```

Be careful how you specify paths when using this method, as the following example has the effect of moving C:\temp\oldname to the default directory and then renaming it newname.

```
USER>write ##class(%File).Rename("C:\temp\oldname", "newname", .return)
1
```

6

Work with Files

The %Library.File class provides several class methods that allow you to perform various operations on files. For information on manipulating the files themselves, see Work with the %File Object.

Note: If you specify a partial filename or directory name, most of these methods assume that you are referring to an item relative to the directory that contains the default globals database for the namespace you are working in. This directory is referred here as the "default directory." Any exceptions to this rule are noted.

Also, these methods treat the file or directory name as case-sensitive only if the underlying operating system treats file and directory names as case-sensitive. That is, file or directory names are case-sensitive on UNIX® but not case-sensitive on Windows.

6.1 Copy Files

To copy a file, use the **CopyFile()** method, which returns a boolean value to indicate success or failure.

This method takes four arguments:

1. *from* — Specifies the name of the source file.
2. *to* — Specifies the name of the target file.
3. *pDeleteBeforeCopy* — Specifies whether to delete the target file, if it exists, before the copy is performed. The default is 0.
4. *return* — Output argument. If negative, contains the error code returned by the operating system in case the method fails.

Examples:

The first example, below, copies the file old.txt to new.txt in the directory C:\temp. The second example copies the same file to new.txt in the default directory.

```
USER>write ##class(%File).CopyFile("C:\temp\old.txt", "C:\temp\new.txt", 0, .return)
1
USER>write ##class(%File).CopyFile("C:\temp\old.txt", "new.txt", 0, .return)
1
```

This last example fails with a Windows error code of 2, or “File not found.”

```
USER>write ##class(%File).CopyFile("foo.txt", "new.txt", 0, .return)
0
USER>write return
-2
```

6.2 Delete Files

To delete a file, use the **Delete()** method, which returns a 1 on success or 0 on failure. This method takes two arguments. The first argument is the name of the file to remove. The second argument is an output argument. If negative, it contains the error code returned by the operating system in case the method fails.

In the first example, below, the method succeeds. The second example fails with a Windows error code of 2, or “File not found.”

```
USER>write ##class(%File).Delete("C:\temp\myfile.txt", .return)
1
USER>write ##class(%File).Delete("C:\temp\myfile.txt", .return)
0
USER>write return
-2
```

To match wildcards when deleting files, use the **ComplexDelete()** method. The first argument specifies the names of the files to remove. The second argument is an output argument. If negative, it contains the error code returned by the operating system in case the method fails.

The following example deletes all files with the .out extension in the C:\temp directory.

```
USER>write ##class(%File).ComplexDelete("C:\temp\*.out", .return)
1
```

6.3 Truncate Files

To truncate a file, use the **Truncate()** method, which returns a 1 on success or 0 on failure. This method takes two arguments. The first argument is the name of the file to truncate. The second argument is an output argument. If negative, it contains the error code returned by the operating system in case the method fails.

If you truncate an existing file, the method deletes the content from the file, but does not remove it from the file system. If you truncate a file that does not exist, the method creates a new empty file.

In the first example, below, the method succeeds. The second example fails with a Windows error code of 5, or “Access is denied.”

```
USER>write ##class(%File).Truncate("C:\temp\myfile.txt", .return)
1
USER>write ##class(%File).Truncate("C:\no access.txt", .return)
0
USER>write return
-5
```

6.4 Rename Files

To rename a file, use the **Rename()** method, which returns a 1 on success or 0 on failure. This method takes three arguments. The first argument is the name of the file to rename, and the second is the new name. The third argument is an output argument. If negative, it contains the error code returned by the operating system in case the method fails.

In the first example, below, the method succeeds. The second example fails with a Windows error code of 183, or "Cannot create a file when that file already exists."

```
USER>write ##class(%File).Rename("C:\temp\oldname.txt", "C:\temp\newname.txt", .return)
1
USER>write ##class(%File).Rename("C:\temp\another.txt", "C:\temp\newname.txt", .return)
0
USER>write return
-183
```

Be careful how you specify paths when using this method, as the following example has the effect of moving C:\temp\oldname.txt to the default directory and then renaming it newname.txt.

```
USER>write ##class(%File).Rename("C:\temp\oldname.txt", "newname.txt", .return)
1
```

6.5 Compare Files

To compare two files, use the **Compare()** method, which returns a boolean value of 1 if the two files are identical and 0 otherwise. The method does not have an output argument in which to return a system error code.

In the first example, below, the two files are identical, and the method returns 1. In the second example, the two files are different, so the method returns 0.

```
USER>write ##class(%File).Compare("C:\temp\old.txt", "C:\temp\new.txt")
1
USER>write ##class(%File).Compare("C:\temp\old.txt", "C:\temp\another.txt")
0
```

If either or both file does not exist, as in the example below, the method also returns 0.

```
USER>write ##class(%File).Compare("foo.txt", "bar.txt")
0
USER>write ##class(%File).Exists("foo.txt")
0
```

6.6 Generate Temporary Files

To generate a temporary file, use the **TempFilename()** method, which returns the name of the temporary file. This method takes three arguments. The first argument is the desired file extension of the temporary file. The second is a directory in which to generate the temporary file. If not provided, the method generates the file in the OS-provided temporary directory. The third argument is an output argument. If negative, it contains the error code returned by the operating system in case the method fails.

Windows examples:

```
USER>write ##class(%File).TempFilename("txt")
C:\WINDOWS\TEMP\GATqk8a6.txt
USER>write ##class(%File).TempFilename("txt","C:\temp")
C:\temp\WpSwuLlA.txt
```

Unix examples:

```
USER>write ##class(%File).TempFilename("", "", .return)
/tmp/filsfHGzc
USER>write ##class(%File).TempFilename("tmp", "/InterSystems/temp", .return)
/InterSystems/temp/file0tnuh.tmp
USER>write ##class(%File).TempFilename("", "/tmp1", .return)

USER>write return
-2
```

In the third example, above, the directory does not exist, and the method fails with system error code 2, or “No such file or directory.”

7

Work with the %File Object

If you want to manipulate a file itself, you need to instantiate a %File object using the **%New()** method of the %Library.File class. The class also provides instance methods that allow you to work with the file.

Note: This section provides a few examples of using the %File object for illustrative purposes.

For simple reading and writing of files, use the %Stream.FileCharacter and %Stream.FileBinary classes, as these provide additional functionality, for example, automatically opening files in the correct mode.

Note: If you specify a partial filename or directory name, most of these methods assume that you are referring to an item relative to the directory that contains the default globals database for the namespace you are working in. This directory is referred here as the "default directory." Any exceptions to this rule are noted.

Also, these methods treat the file or directory name as case-sensitive only if the underlying operating system treats file and directory names as case-sensitive. That is, file or directory names are case-sensitive on UNIX® but not case-sensitive on Windows.

7.1 Create an Instance of a %File Object

To work with a file, you need to instantiate a %File object that represents that file using the **%New()** method. This file may or may not already exist on disk.

The following example instantiates a %File object for the file export.xml in the default directory.

```
set fileObj = ##class(%File).%New("export.xml")
```

7.2 Open and Close Files

Once you have instantiated a %File object, you need to open the file using the **Open()** method to read from it or write to it:

```
USER>set status = fileObj.Open()

USER>write status
1
```

Use the **Close()** method to close the file:

```
USER>do fileObj.Close()
```

7.3 Examine Properties of a %File Object

Once you have instantiated the file, you can examine the properties of the file directly.

```
USER>write fileObj.Name
export.xml
USER>write fileObj.Size
2512
USER>write $zdate(fileObj.DateCreated)
11/18/2020
USER>write $zdate(fileObj.DateModified)
11/18/2020
USER>write fileObj.LastModified
2020-11-18 14:24:38
USER>write fileObj.IsOpen
0
```

Note that LastModified is a human readable timestamp, not a date in **$H** format.

The properties Size, DateCreated, DateModified, and LastModified are calculated at the time they are accessed. Accessing these properties for a file that does not exist returns `-2`, indicating that the file could not be found.

Note: Windows is the only platform that currently tracks the actual created date. Other platforms store the date of the last file status change.

```
USER>write ##class(%File).Exists("foo.xml")
0
USER>set fooObj = ##class(%File).%New("foo.xml")

USER>write fooObj.Size
-2
```

If a file is open, you can view its canonical name, which is the full path from the root directory, by accessing the CanonicalName property.

```
USER>write fileObj.CanonicalName

USER>set status = fileObj.Open()

USER>write fileObj.IsOpen
1
USER>write fileObj.CanonicalName
c:\intersystems\IRIS\mgr\user\export.xml
```

7.4 Read from Files

To read from a file, you can open the file and then use the **Read()** method.

The following example reads the first 200 characters of messages.log.

```
USER>set messages = ##class(%File).%New(##class(%File).ManagerDirectory() _ "messages.log")

USER>set status =  messages.Open("RU")

USER>write status
1
USER>set text = messages.Read(200, .sc)

USER>write text


*** Recovery started at Mon Dec 09 16:42:01 2019
     Current default directory: c:\intersystems\IRIS\mgr
     Log file directory: .\
     WIJ file spec: c:\intersystems\IRIS\mgr\IR
USER>write sc
1
USER>do messages.Close()
```

To read an entire line from a file, use the **ReadLine()** method, which is inherited from %Library.File's parent class, %Library.AbstractStream.

The following example reads the first line of C:\temp\shakespeare.txt.

```
USER>set fileObj  = ##class(%File).%New("C:\temp\shakespeare.txt")

USER>set status =  fileObj.Open("RU")

USER>write status
1
USER>set text = fileObj.ReadLine(,.sc)

USER>write text
Shall I compare thee to a summer's day?
USER>write sc
1
USER>do fileObj.Close()
```

7.5 Write to Files

To write to a file, you can open the file and then use the **Write()** or **WriteLine()** methods.

The following example writes a line of text to a new file.

```
USER>set fileObj = ##class(%File).%New("C:\temp\newfile.txt")

USER>set status = fileObj.Open("WSN")

USER>write status
1
USER>set status = fileObj.WriteLine("Writing to a new file.")

USER>write status
1
USER>write fileObj.Size
24
```

7.6 Rewind Files

After reading from or writing to a file, you may wish to rewind the file using the **Rewind()** method so that you can perform operations from the beginning of the file.

Taking up from where the previous example left off, `fileObj` is now positioned at its end. Rewinding the file and using **WriteLine()** again has the effect of overwriting the file.

```
USER>set status = fileObj.Rewind()

USER>write status
1
USER>set status = fileObj.WriteLine("Rewriting the file from the beginning.")

USER>write status
1
USER>write fileObj.Size
40
```

Closing the file and reopening it also rewinds the file.

```
USER>do fileObj.Close()

USER>set status = fileObj.Open("RU")

USER>write status
1
USER>set text = fileObj.ReadLine(,.sc)

USER>write sc
1
USER>write text
Rewriting the file from the beginning.
```

7.7 Clear Files

To clear a file, you can open the file and then use the **Clear()** method. This removes the file from the file system.

The following example clears the file junk.xml in the default directory.

```
USER>write ##class(%File).Exists("junk.xml")
1
USER>set fileObj = ##class(%File).%New("junk.xml")

USER>set status = fileObj.Open()

USER>write status
1
USER>set status = fileObj.Clear()

USER>write status
1
USER>write ##class(%File).Exists("junk.xml")
0
```

8

%File Example

This example shows a sample class that uses several of the %Library.File methods.

In the example class User.FileTest, the **ProcessFile()** method accepts an input file and an output file and calls **SetUpInputFile()** and **SetUpOutputFile()** to open the files, one for reading and one for writing. It then reads the input file, line by line, and calls **ProcessLine()** to perform one or more substitutions on the contents of each line, writing the new contents of each line to the output file.

```
Include %occInclude

Class User.FileTest Extends %Persistent
{

/// Set up the input file
/// 1. Create a file object
/// 2. Open the file for reading
/// 3. Return a handle to the file object
ClassMethod SetUpInputFile(filename As %String) As %File
{
    Set fileObj = ##class(%File).%New(filename)
    Set status = fileObj.Open("RU")
    if $$$ISERR(status) {
        do $system.Status.DisplayError(status)
        quit $$$NULLOREF
    }
    quit fileObj
}

/// Set up the output file
/// 1. Create the directory structure for the file
/// 2. Create a file object
/// 3. Open the file for writing
/// 4. Return a handle to the file object
ClassMethod SetUpOutputFile(filename As %String) As %File
{
    set dir=##class(%File).GetDirectory(filename)
    do ##class(%File).CreateDirectoryChain(dir)
    Set fileObj = ##class(%File).%New(filename)
    Set status = fileObj.Open("WSN")
    If ($SYSTEM.Status.IsError(status)) {
        do $system.Status.DisplayError(status)
        quit $$$NULLOREF
    }
    quit fileObj
}

/// Process one line, using $REPLACE to perform a series of substitutions on the line
ClassMethod ProcessLine(line As %String = "") As %String
{
    set newline = line

    set newline = $REPLACE(newline, "Original", "Jamaican-Style")
    set newline = $REPLACE(newline, "traditional", "innovative")
    set newline = $REPLACE(newline, "orange juice", "lime juice")
    set newline = $REPLACE(newline, "orange zest", "ginger")
    set newline = $REPLACE(newline, "white sugar", "light brown sugar")

    quit newline
}
```

```
/// Process an input file, performing a series of substitutions on the content and
/// writing the new content to an output file
ClassMethod ProcessFile(inputfilename As %String = "", outputfilename As %String = "")
{
    // Make sure filenames were passed in
    if (inputfilename="") || (outputfilename="") {
        write !, "ERROR: missing file name"
        quit
    }

    // Open input file for reading
    set inputfile = ..SetUpInputFile(inputfilename)
    if (inputfile = $$$NULLOREF) quit

    // Open output file for writing
    set outputfile = ..SetUpOutputFile(outputfilename)
    if (outputfile = $$$NULLOREF) quit

    // Loop over each line in the input file
    // While not at the end of the file:
    // 1. Read a line from the file
    // 2. Call ProcessLine() to process the line
    // 3. Write the new contents of the line to the output file
    while (inputfile.AtEnd = 0) {
        set line = inputfile.ReadLine(,.status)
         if $$$ISERR(status) {
             do $system.Status.DisplayError(status)
         }
         else {
             set newline = ..ProcessLine(line)
             do outputfile.WriteLine(newline)
         }
    }

    // Close the input and output files
     do inputfile.Close()
     do outputfile.Close()
}

}
```

Call the **ProcessFile()** method as follows:

```
USER>do ##class(FileTest).ProcessFile("C:\temp\original cranberry sauce.txt",
"C:\temp\jamaican-style cranberry sauce.txt")
```

If the input file, C:\temp\original cranberry sauce.txt, contains the following:

```
Original Whole Berry Cranberry Sauce

This traditional whole berry cranberry sauce gets its distinctive flavor
from the freshly squeezed orange juice and the freshly grated orange zest.

2 tsp freshly grated orange zest
1 1/4 cups white sugar
1/4 cup freshly squeezed orange juice
3 cups cranberries (12 oz. package)

1. Grate orange zest into a bowl and set aside.
2. Combine the sugar and orange juice in a saucepan. Bring to a boil over medium-low
heat and stir until sugar is dissolved.
3. Add cranberries and cook over medium-high heat, stirring occasionally, until the
cranberries have popped.
4. Add the cranberry mixture into the bowl with the orange zest, and stir. Let cool.
5. Cover bowl and chill.
```

Then the output file, C:\temp\jamaican-style cranberry sauce.txt, will contain the following:

```
Jamaican-Style Whole Berry Cranberry Sauce

This innovative whole berry cranberry sauce gets its distinctive flavor
from the freshly squeezed lime juice and the freshly grated ginger.

2 tsp freshly grated ginger
1 1/4 cups light brown sugar
1/4 cup freshly squeezed lime juice
3 cups cranberries (12 oz. package)

1. Grate ginger into a bowl and set aside.
2. Combine the sugar and lime juice in a saucepan. Bring to a boil over medium-low
heat and stir until sugar is dissolved.
3. Add cranberries and cook over medium-high heat, stirring occasionally, until the
cranberries have popped.
4. Add the cranberry mixture into the bowl with the ginger, and stir. Let cool.
5. Cover bowl and chill.
```