Files and Folders

Introduction

The File API provides an interface with the server's OS File system. It allows you to handle files and folders as JavaScript objects using specific classes. Also, it provides support for reading and writing text streams and binary streams.

BinaryStream Class

Properties and methods in the BinaryStream class allow you to handle and parse binary streams (i.e., streams of bytes).

To create a *BinaryStream* object, you need to execute the **BinaryStream( )** constructor method (from the Application class).

A *BinaryStream* object provides sequential access to binary data streams, mapped on binary files stored on disk.

Note that Wakanda also proposes support for *TextStream* objects for which you can define a *charSet* parameter.

BinaryStream( )

BinaryStream **BinaryStream**(String | File *file*, String *readMode*)

Parameter	Type	Description
file	String, File	Binary file to reference
readMode	String	Streaming action: "Write" for writing data, "Read" for reading data
Returns	BinaryStream	New BinaryStream object

Description

The **BinaryStream( )** method creates a new *BinaryStream* object. *BinaryStream* objects are handled using the various properties and methods of the **BinaryStream Class**.

In the *file* parameter, pass the path of, or a reference to, the binary file to write or to read. It can be one of the following forms:

- an absolute path (using the "/" separator) or a URL, including the file name
- a valid *File* object

Once the file is referenced, you can start writing or reading the data, depending on the value you passed in the *readMode* parameter:

- If you passed "Write", the file is opened in write mode.
- If you passed "Read", the file is opened in read mode.

changeByteOrder( )

void **changeByteOrder**( )

Description

The **changeByteOrder( )** method indicates that the next reading of structured values in the *BinaryStream* object requires a byte swap.

This method is useful when you want to read structured data (like long or real values -- actually any data type other that bytes) and the file used for the streaming was created on a platform where the byte ordering is different. The byte swap is necessary for example between PowerPC-based and Intel-based computers.

close( )

void **close**( )

Description

The **close( )** method closes the file referenced in the *BinaryStream* object.

The referenced file is opened when you execute the method and remains open until you call **close( )**.

flush( )

void **flush**( )

Description

The **flush( )** method saves the buffer contents to the disk file referenced in the *BinaryStream* object.

When you do several method calls to write data into a *BinaryStream* object, for optimization reasons the data is stored in a buffer that is saved to disk when the stream is closed. This method allows you to store the buffer during the process without having to close the stream.

getByte( )

Number **getByte**( )

Returns	Number	Byte at the cursor location

Description

The **getByte( )** method returns a number representing the next byte from the *BinaryStream* object and moves the cursor to the next position in the stream.

getLong( )

Number **getLong**( )

Returns	Number	Long value in the stream

Description

The **getLong( )** method returns the next long number (if present) from the *BinaryStream* object and moves the cursor to the next position in the stream.

Long numbers are coded on four (4) bytes. Note that the size and byte ordering of long numbers may vary from one OS to another.

getLong64( )

Number **getLong64**( )

Returns	Number	Next Long64 value in the stream

Description

The **getLong64( )** method returns the next long64 number (if present) from the *BinaryStream* object and moves the cursor to the next position in the stream.

Long64 numbers are coded on eight (8) bytes. Note that the size and byte ordering of long numbers may vary from one OS to another.

getPos( )

Number **getPos**( )

Returns	Number	Position of the cursor in the stream

Description

The **getPos( )** method returns the current position of the cursor in the *BinaryStream* object.

getReal( )

Number **getReal**( )

Returns	Number	Next real value in stream

Description

The **getReal( )** method returns the next real (if present) from the *BinaryStream* object and moves the cursor to the next

position in the stream.

getSize( )

Number **getSize**( )

Returns	Number	Current size of the stream (in bytes)

Description

The **getSize( )** method returns the size of the stream.

When you are reading the stream, this method returns the size of the file referenced in the *BinaryStream* object.

When you are writing the stream, this method returns the current size of the stream in memory.

getString( )

String **getString**

Returns	String	Next string value in the stream

Description

The **getString( )** method returns the next string (if present) from the *BinaryStream* object and moves the cursor to the next position in the stream.

getWord( )

Number **getWord**( )

Returns	Number	Next binary integer

Description

The **getWord( )** method returns the next word, i.e., a binary integer (if present) from the *BinaryStream* object, and moves the cursor to the next position in the stream.

Word numbers are coded on two (2) bytes. Note that the size and byte ordering of integers may vary from one OS to another.

isByteSwapping( )

Boolean **isByteSwapping**( )

Returns	Boolean	True if the stream needs byte-swapping

Description

The **isByteSwapping( )** method returns *True* if the current data reading in the *BinaryStream* object is in byte swap mode.

The byte swap mode is activated using the method.

putByte( )

void **putByte**(Number *byteValue*)

Parameter	**Type**	**Description**
byteValue	Number	Byte value to write into the stream

Description

The **putByte( )** method writes the byte value you passed as the parameter in the *BinaryStream* object at the current cursor location. The cursor is then moved to the next position in the stream.

putLong( )

void **putLong**(Number *longValue*)

Parameter	Type	Description
longValue	Number	Long value to write into the stream

Description

The **putLong( )** method writes the long value you passed as the parameter in the *BinaryStream* object at the current cursor location. The cursor is then moved to the next position in the stream.

putLong64( )

void **putLong64**(Number *long64Value*)

Parameter	Type	Description
long64Value	Number	Long64 value to write into the stream

Description

The **putLong64( )** method writes the long64 value you passed as the parameter in the *BinaryStream* object at the current cursor location. The cursor is then moved to the next position in the stream.

putReal( )

void **putReal**(Number *realValue*)

Parameter	Type	Description
realValue	Number	Real value to write into the stream

Description

The **putReal( )** method writes the real value you passed as the parameter in the *BinaryStream* object at the current cursor location. The cursor is then moved to the next position in the stream.

putString( )

void **putString**(String *stringValue*)

Parameter	Type	Description
stringValue	String	String value to write into the stream

Description

The **putString( )** method writes the string value you passed as the parameter in the *BinaryStream* object at the current cursor location. The cursor is then moved to the next position in the stream.

putWord( )

void **putWord**(Number *wordValue*)

Parameter	Type	Description
wordValue	Number	Word (integer) value to write into the stream

Description

The **putWord( )** method writes the byte word (i.e., an integer value) you passed as the parameter in the *BinaryStream* object at the current cursor location. The cursor is then moved to the next position in the stream.

setPos( )

void **setPos**(Number *offset*)

Parameter	Type	Description
offset	Number	New cursor position in the stream

Description

The **setPos( )** method moves the stream cursor to the position you passed in *offset* in the *BinaryStream* object.

File Class

The File class provides properties and methods that allow you to create and manipulate server-side *File* objects.

The basic way to create a *File* object is to execute the **File( )** method (from the Application class).

Note: The Wakanda File object implementation is close to that of the W3C File Object.

File objects contain references to disk files that may or may not actually exist on disk. For example, when you execute the **File( )** method to create a new file, a valid *File* object is created but nothing is actually stored on disk until you call the **create( )** method.
You should be aware that some methods will work correctly with the *File* object, even if the referenced file does not exist on disk. Some other methods require that the referenced file actually exists on disk. If you call any of them and the file does not exist, a "file not found" error is generated.

creationDate

Description

The **creationDate** property returns the creation date for the *File* object.

This property can be modified.

exists

Description

The **exists** property returns *true* if the file referenced in the *File* object already exists at the defined path. Testing this property before creating or renaming files will help prevent errors.

Example

See the example for the **TextStream( )** method.

extension

Description

The **extension** property returns the file name extension of the *File* object.

The extension string is returned without the "." character at the beginning.

The file referenced in the *File* object does not need to already exist on disk.

modificationDate

Description

The **modificationDate** property returns the last modification date for the *File* object.

This property can be modified.

name

Description

The **name** property gets or sets the short name of the *File* object without the path information. The file name is handled with its extension. If you do not want to manage the extension, use the **nameNoExt** property.

If you read the file name, the referenced file does not need to already exist on disk.
If you change the file name using this property, the referenced file must already exist. Note that it is renamed on disk but the **name**property is not updated in the *File* object.

Example

This property allows you to read or write the file name on disk:

```
var myFile = File ("c:/temp/invoices.txt");
var theName = myFile.name; // theName contains "invoices.txt"
myFile.name = theName + "_old"; // the file is renamed "invoices.txt_old" on disk
    //but myFile.name still contains "invoices.txt"
```

nameNoExt

Description

The **nameNoExt** property returns the short name of the *File* object without its path information or extension.

If you want to get the file name with its extension, use the **name** property.

parent

Description

The **parent** property returns a new *Folder* object containing the parent folder of the *File* or *Folder* object. The file or folder referenced in the object does not need to already exist on disk.

path

Description

The **path** property returns the full path of the *File* or *Folder* object, including the file or folder name itself.

The file or folder referenced in the object does not necessarily need to exist on disk. The returned path is expressed using the Posix syntax (folders are separated with "/").

readOnly

Description

The **readOnly** property gets or sets the read-only status of the *File* or *Folder* object.

The property value is *true* if the referenced file or folder is in read-only mode and *false* if it is in read/write mode.

The referenced file or folder must already exist on disk and you must have the appropriate access rights to change its status.

size

Description

The **size** property returns the size of the *File* object expressed in bytes. The file referenced in the *File* object must already exist on disk when the property is read. Otherwise, an error is returned.

visible

Description

The **visible** property gets or sets the visibility status of the *File* or *Folder* object. The referenced file or folder must already exist on disk and, if you want to change the property value, you must have the appropriate access rights.

The property value is *true* if the referenced file or folder is visible, and *false* if it is not visible.

copyTo( )

void **copyTo**(File | String *destination* [, Boolean | String *overwrite*])

Parameter	Type	Description
destination	File, String	Destination file
overwrite	Boolean, String	True or "Overwrite" to override existing file if any, otherwise False or "KeepExisting"

Description

The **copyTo( )** method copies the file referenced in the *File* object (the source object) into the specified *destination*. The file referenced in the *File* source object must already exist, otherwise the method returns a "File not found" error.

In the *destination* parameter, you can pass a string containing an absolute path, a URL, or a *File* object.

By default, a "File already exists" error will occur if there is a file with the same name as the source file at the defined *destination*. You can change this behavior by using the *overwrite* parameter:

- If you pass *True* or the "OverWrite" string in *overwrite*, the method will delete and override the existing file

without any error.
- If you pass *False* or the "KeepExisting" string in *overwrite* (or omit the parameter), the existing file is left untouched and an error is generated. By default, the file is not overwritten.

Example

The following example duplicates a file in its own folder:

```
var myFile = File ("c:/Documents/Invoice.txt") ; // get the File object and
myFile.copyTo ( "c:/Documents/Invoice_copy.txt") ; // duplicate it to the same location
```

Example

The following example duplicates a file and overwrites the copy if it already exists:

```
var aCopy = File ("c:/Documents/Invoice_copy.txt") ;
if (aCopy.exists)
    {
        var myFile = File ("c:/Documents/Invoice2011.txt") ;
        myFile.copyTo ( "c:/Documents/Invoice_copy.txt" , "OverWrite") ;
    }
```

create( )

Boolean **create**( )

Returns	Boolean	True if the file has been created, otherwise False.

Description

The **create( )** method stores the file referenced in the *File* on disk. The path of the file has been defined in the **File( )** method (constructor).

The **create( )** method returns *True* if the file is created successfully. In all other cases, it returns *False*.

File( )

File **File**(String | Folder *path* [, String *fileName*])

Parameter	Type	Description
path	String, Folder	Path of the file to reference
fileName	String	Name of the file to reference
Returns	File	New File object

Description

The **File( )** method creates a new *File* object. *File* objects are handled using the various properties and methods of the **File Class**.

In the *path* parameter, pass the file's path in one of the following forms:

- an absolute path (using the "/" separator) or a URL, including the file name -- in both cases, the *fileName* parameter must be omitted
- a *Folder* object -- in this case, the *fileName* parameter must be passed

If necessary, pass the file name in the *fileName* parameter.

Note that this method only creates an object that references a file and does not create the file on disk. You can work with *File* objects referencing files that may or may not exist. If you want to create the referenced file, you need to execute the **create( )** method.

Example

This example creates a new blank datastore on disk using the current datastore model:

```
    // get a reference to the current datastore model file
var currentFolder = ds.getModelFolder().path
var currentModel = File(currentFolder+ ds.getName() + ".waModel");
            // only works if the datastore has the same name as the model file
```

```
if (currentModel.exists)     // if the model actually exists
{
    var dataFile = File(currentFolder+ "newData.waData");     // create a reference to the
    var myDS = createDataStore(currentModel, dataFile);     // create and reference the da
}
```

getFreeSpace( )

Number **getFreeSpace**([Boolean | String *quotas*])

Parameter	Type	Description
quotas	Boolean, String	true or "NoQuotas" = consider the whole volume (default), false or "WithQuotas" = consider only the allowed size for the quota
Returns	Number	Free space in bytes on the volume

Description

The **getFreeSpace( )** method returns the size of the free space (expressed in bytes) available on the volume where the *File* or *Folder* object is stored. The referenced file or folder must exist on disk.

If system disk quotas have been activated on the volume where the *File* or *Folder* is stored, you can get the free space on the whole volume or only on your disk quota size, depending on the *quotas* parameter:

- If you pass *true* or the "NoQuotas" string in *quotas* (or omit the parameter), the method will take the whole volume into account. This is the action by default.
- If you pass *false* or the "WithQuotas" string in *quotas*, the method will return only the free space of the disk quota.

getURL( )

String **getURL**([Boolean | String *encoding*])

Parameter	Type	Description
encoding	Boolean, String	true or "Encoded" = encode the URL, false or "Not Encoded" = do not encode the URL (default)
Returns	String	URL of the referenced file

Description

The **getURL( )** method returns the absolute URL of the *File* or *Folder* object.

By default, the returned URL characters are not encoded. You can set the encoding mode by using the *encoding* parameter:

- If you pass *true* or the "Encoded" string in *encoding*, the URL will be encoded.
- If you pass *false* or the "Not Encoded" string in *encoding* (or omit the parameter), the URL is not encoded. This is the default action.

Example

The following example gets the URL from the path of a file:

```
var myfile = File ("c:/wk/web/img/logo.png");
var url = myfile.getURL(); // returns "file:///c:/wk/web/img/logo.png"
```

getVolumeSize( )

Number **getVolumeSize**([Boolean | String *quotas*])

Parameter	Type	Description
quotas	Boolean, String	true or "NoQuotas" = consider the whole volume (default), false or "WithQuotas" = consider only the allowed size for the quota
Returns	Number	Size in bytes of the volume where the file is stored

Description

The **getVolumeSize( )** method returns the size (expressed in bytes) of the volume where the *File* or *Folder* object is stored. The referenced file or folder must exist on disk.

If system disk quotas have been activated on the volume where the file or folder is stored, you can get the total size of the volume or only your disk quota size, depending on the value you pass in the *quotas* parameter:

- If you pass *true* or the "NoQuotas" string in *quotas* (or omit the parameter), this method returns the whole size of the disk. This is the default action.
- If you pass *false* or the "WithQuotas" string in *quotas*, this method returns the size of the disk quota.

isFile( )

Boolean **isFile**(String *path*)

Parameter	Type	Description
path	String	Posix path
Returns	Boolean	True if path corresponds to an existing path, False otherwise

Description

The **isFile( )** class method can be used with the **File( )** constructor to know if *path* corresponds to a file on disk.

Pass in *path* a Posix string containing an absolute or relative path to a file on the disk. If the path corresponds to a file on the disk, **isFile( )** returns True. If the path corresponds to a non-existing file or a folder, **isFile( )** returns False.

Example

We want to know if "reports.proj" is a file:

```
var myFile = File.isFile("C:/wakanda/projects/reports.proj");
```

moveTo( )

void **moveTo**(File | String *destination* [, Boolean | String *overwrite*])

Parameter	Type	Description
destination	File, String	Destination file
overwrite	Boolean, String	True or "Overwrite" to override existing file if any, otherwise false or "KeepExisting"

Description

The **moveTo( )** method moves the file referenced in the *File* object (the source object) to the specified *destination*. The file referenced in the *File* source object must already exist, otherwise the method returns a "File not found" error.

In the *destination* parameter, you can pass a string containing an absolute path, a URL, or a *File* object.

By default, a "File already exists" error will occur if there is a file with the same name as the source file at the defined *destination*. You can change this behavior using the *overwrite* parameter:

- If you pass *true* or the "OverWrite" string in *overwrite*, the method will delete and overwrite the existing file without any error.
- If you pass *false* or the "KeepExisting" string in *overwrite* (or omit the parameter), the existing file is left untouched and an error is generated. This is the default action.

Notes:

- You can use this method to rename a file on disk.
- Using this method, you can move a file from and to any folder on the same volume. If you want to move a file between two distinct volumes, use **copyTo( )** to “move” the file then delete the original copy of the file using the **remove( )** method.

next( )

Boolean **next**( )

Returns	Boolean	True if there is a next file in the iteration. Otherwise, it is false.

Description

The **next( )** method works with the file iterator: it puts the file pointer on the next file within an iteration of files, for example, in a **for** loop.

The method returns *true* if it has been executed correctly and *false* otherwise. More specifically, the method returns *false* when it reaches the end of a file collection, i.e., the files stored within a *Folder* object.

Example

See example for the **valid( )** method.

remove( )

Boolean **remove**( )

Returns	Boolean	True if the file was removed successfully. Otherwise, it returns false.

Description

Note: You can also call this method using the drop( ) name.

The **remove( )** method removes the file or folder referenced in the *File* or *Folder* object from the storage volume. The file or folder referenced in the object must already exist on disk, otherwise this method returns a "File not found" error. In the case of folders, this method deletes the folder as well as all of its contents.

The **remove( )** method returns *true* if the file or folder is removed successfully. In all other cases, it returns *false*.

setName ( )

Boolean **setName** (String *newName*)

Parameter	Type	Description
newName	String	New name for the disk file
Returns	Boolean	True if the file has been renamed successfully. Otherwise, it returns false.

Description

The **setName ( )** method allows you to rename a file on disk referenced in the *File* object. For this method to work, the file must already exist on disk and the application should have appropriate write permissions.

In the *newName* parameter, pass the new name of the file with its extension. The string must comply with the OS file naming rules.

If the method has been completed successfully, it returns *true*. If a problem occurred (file not found, no write permission, etc.), the method returns *false*.

Note that this method will rename the file on disk, but not the file name referenced in the *File* object.

valid( )

Boolean **valid**( )

Returns	Boolean	True if a current file reference exists. Otherwise, it returns false.

Description

The **valid( )** method works with the file iterator: it checks the validity of the pointer to the current *File* object within an iteration of files, for example, in a **for** loop.

The method returns *true* if the pointer is valid and *false* otherwise.

Example

In this example, we change all the file names in a folder (those at the top level of the folder) to uppercase letters:

```
var myFolder = Folder("c:/Wakanda/Files/");  // get a reference to the folder
for (var afile = myFolder.firstFile; afile.valid(); afile.next())
        // start at the first file and check whether it is valid in the folder
        // then move to the next file
    {
        afile.name = afile.name.toUpperCase();
```

```
}
```

Example

See example for the **firstFile** property (Folder class).

Folder Class

The Folder class provides properties and methods properties and methods that allow you to create and manipulate server-side *Folder* objects.

The basic way to create a *Folder* object is to execute the **Folder( )** method (method from the Application class).

Folder objects contain references to folders that may or may not actually exist on disk. For example, when you execute the **Folder( )** method to create a new folder, a valid *Folder* object is created but nothing is actually stored on disk until you call the **create( )** method.
You should be aware of the fact that some methods will work correctly with a *Folder* object, even if the referenced folder does not exist on disk. Some other methods require that the referenced folder actually exists on disk: if they are called when the folder does not exist, a 'file not found' error is generated.

creationDate

Description

The **creationDate** property returns the creation date for the folder referenced in the *Folder* object.

This value can be modified.

Example

This example sorts a folder list by date:

```
function sortFolderListByCreationDate(folderA, folderB) {
    return (folderA.creationDate > folderB.creationDate) ? 1 : -1;
}
```

files

Description

The **files** property returns the list of the files located in the *Folder* object. The property returns an array of *File* objects.

Example

This function returns true if all the files in a folder are visible:

```
function testAllFilesAreVisible(folder) {
    return folder.files.every(
        function (file) {
            return file.visible;
        }
    );
}
```

firstFile

Description

The **firstFile** property returns a new *File* object referencing the file found in the first position of the *Folder* object. This property is useful when looping files in a folder.

If the folder does not contain any files, the property returns a *File* object with all the property values set to null.

Note: The first file in a folder is selected internally by the file iterator; no specific order can be applied.

Example

If you want to register each first-level file in a specific folder in an array to display a choice list, you can write the following:

```
var folder = Folder("c:/wakanda/");
var result2 = [];
var fileIter = folder.firstFile;  // initialize the iteration
while (fileIter.valid())  // while there are files to process
{
    result2.push(fileIter.path);
    fileIter.next();  // increment the iteration
}
```

```
result2;   // display the object in the Wakanda Code Editor
```

Note: The result of this example is exactly the same as the result of example 1 for the forEachFile( ) method. The only advantage of this example is that it allows you to add tests to stop the iteration at any time.

firstFolder

Description

The **firstFolder** property returns a new *Folder* object containing the subfolder found in the first position of the *Folder* object. This property is useful when you loop the subfolders in a folder.

If the folder does not contain any subfolders, the property returns a *Folder* object with all the property values set to null.

Note: The first subfolder in a folder is selected internally by the iterator; no specific order can be applied.

folders

Description

The **folders** property returns the list of the subfolders located in the *Folder* object. The property returns an array of *Folder* objects.

modificationDate

Description

The **modificationDate** property returns the last modification date for the folder referenced in the *Folder* object.

This value can be modified.

name

Description

The **name** property gets or sets the name of the *Folder* object without the path information.

When you read this property, the referenced folder does not need to already exist on disk.
If you change the folder name using this property, the referenced folder must already exist. Note that it is renamed on disk but the **name** property is not updated in the *Folder* object.

Example

This function returns 1 if the name of folderA is greater than folderB in alphabetical order. Otherwise, it returns -1.

```
function sortFolderListByName(folderA, folderB) {
    return (folderA.name > folderB.name) ? 1 : -1;
}
```

parent

Description

The **parent** property returns a new *Folder* object containing the parent folder of the *File* or *Folder* object. The file or folder referenced in the object does not need to already exist on disk.

path

Description

The **path** property returns the full path of the *File* or *Folder* object, including the file or folder name itself.

The file or folder referenced in the object does not necessarily need to exist on disk. The returned path is expressed using the Posix syntax (folders are separated with "/").

readOnly

Description

The **readOnly** property gets or sets the read-only status of the *File* or *Folder* object.

The property value is *true* if the referenced file or folder is in read-only mode and *false* if it is in read/write mode.

The referenced file or folder must already exist on disk and you must have the appropriate access rights to change its status.

visible

Description

The **visible** property gets or sets the visibility status of the *File* or *Folder* object. The referenced file or folder must already exist on disk and, if you want to change the property value, you must have the appropriate access rights.

The property value is *true* if the referenced file or folder is visible, and *false* if it is not visible.

nameNoExt

Description

The **nameNoExt** property returns the name of the *Folder* object without path information and without the main extension (if any).

If you want to get the full name of the folder with its extension, use the **name** property.

extension

Description

The **extension** property returns the folder name extension of the *Folder* object.

If the folder name does not have an extension, the property value is *undefined*.

exists

Description

The **exists** property returns *true* if the folder referenced in the *Folder* object already exists at the defined path.

create( )

Boolean **create**

Returns	Boolean	True if the folder has been created. Otherwise, it returns false.

Description

The **create( )** method creates the folder referenced in the *Folder* object on disk. The path of the folder is defined in the **Folder( )** method.

The **create( )** method returns *true* if the file is created successfully. Otherwise, it returns *false*.

If a folder already exists at the defined path (with or without any contents), the method does nothing but returns *true*. If the path contains a hierarchy of non-existing subfolders, they are created if necessary.

Folder( )

Folder **Folder**(String *path*)

Parameter	Type	Description
path	String	Path of the folder to reference
Returns	Folder	New Folder object

Description

The **Folder( )** method creates a new object of type *Folder*. *Folder* objects are handled using the various properties and methods of the **Folder Class**.

In the *path* parameter, pass the path of the folder to reference. It can be:

- an absolute path in Posix syntax (using the "/" separator) or
- a URL.

Note that this method only creates an object that references a folder and does not create anything on disk. You can handle *Folder* objects referencing folders that may or may not exist. If you want to create the referenced folder, you need to execute the **create( )** method.

Example

This example creates an "Archives" subfolder in the "Wakanda" folder:

```
var newFolder = Folder ("c:/Wakanda/Archives/");
var isOK = newFolder.create();
```

forEachFile( )

void **forEachFile**(Function *callbackFn* [, Object *thisArg*])

Parameter	Type	Description
callbackFn	Function	Handler function to invoke for each file in the folder
thisArg	Object	Token object that will be bound to 'this' in the handler

Description

Note: You can also call this method using its alias ***each( )****.*

The **forEachFile( )** method executes the *callbackFn* function once for each file present at the first level of the *Folder* object in ascending order. Only first level files are taken into account (files in subfolders are left untouched). If you want to process all the files in a folder at every level of recursivity, you may want to consider using the **parse( )** method.

The *callbackFn* function accepts three parameters: *theFile*, *iterator*, and *theFolder*.

- The first parameter, *theFile*, is the file currently being processed. When the function is executed, this parameter receives the file on which it is iterating. You can then perform any type of operation on the file.
- The second (optional) parameter, *iterator*, is the iterator. When the function is executed, this parameter receives the position of the file currently being processed in the folder. You can use it, for example, to display a counter.
- The third (optional) parameter, *theFolder*, receives the parent *Folder* being processed.

If a *thisArg* parameter is provided, it will be used as the **this** value each time the *callbackFn* is called. If it is not provided, **undefined** is used instead.

If existing files in the folder are modified, their values as passed to the callback will be the value when **forEachFile( )** visits them. Files that are deleted after the call to **forEachFile( )** begins and before being visited are not visited.

Example

In this example, we register each first level file in a specific folder in an array, for example to display a choice list by using the **forEachFile( )** method:

```
var folder = Folder("c:/wakanda/");
 var result = [];
 folder.forEachFile(function(file)
 {
     result.push(file.path); // store the file path
 });
 result; // display the object in the Wakanda Code Editor
```

Note: The result for this example is exactly the same as the example of the firstFile method. The forEachFile( ) method simplifies the iteration.

Example

In this example, fill an array with all the files matching a specific criteria in the 'Wakanda' folder.

```
var list = [];
var myFolder = Folder('c:/Wakanda/');
myFolder.forEachFile(
    function (file) {
        if ((file.modificationDate + this.ms) > +(new Date()))
        {
            list.push(file);
        }
    },
```

```
    {ms: 42} // thisArg parameter
);
```

forEachFolder( )

void **forEachFolder**(Function *callbackFn* [, Object *thisArg*])

Parameter	Type	Description
callbackFn	Function	Handler function to call for each subfolder
thisArg	Object	Token object that is bound to 'this' in the handler

Description

The **forEachFolder( )** method executes the *callbackFn* function once for each subfolder present at the first level of the *Folder* object in ascending order. Only first level subfolders are taken into account (nested subfolders are left untouched).

The *callbackFn* function accepts three parameters: *theSubfolder*, *iterator*, and *theFolder*.

- The first parameter, *theSubfolder*, is the subfolder currently being processed. When the function is executed, this parameter receives the folder on which it iterates. You can then perform any type of operation on the subfolder.
- The second (optional) parameter, *iterator*, is the iterator. When the function is executed, this parameter receives the position of the subfolder currently being processed in the parent folder. You can use it, for example, to display a counter.
- The third (optional) parameter, *theFolder*, receives the parent *Folder* being processed.

If a *thisArg* parameter is provided, it will be used as the **this** value each time the *callbackFn* is called. If it is not provided, **undefined** is used instead.

If existing subfolders in the folder are modified, their values as passed to callback will be the values at the time **forEachFolder( )** visits them. Folders that are deleted after the call to **forEachFolder( )** begins and before being visited are not visited.

getFreeSpace( )

Number **getFreeSpace**([Boolean | String *quotas*])

Parameter	Type	Description
quotas	Boolean, String	true or "NoQuotas" = consider the whole volume (default), false or "WithQuotas" = consider only the allowed size for the quota
Returns	Number	Free space in bytes on the volume where the folder is stored

Description

The **getFreeSpace( )** method returns the size of the free space (expressed in bytes) available on the volume where the *File* or *Folder* object is stored. The referenced file or folder must exist on disk.

If system disk quotas have been activated on the volume where the *File* or *Folder* is stored, you can get the free space on the whole volume or only on your disk quota size, depending on the *quotas* parameter:

- If you pass *true* or the "NoQuotas" string in *quotas* (or omit the parameter), the method will take the whole volume into account. This is the action by default.
- If you pass *false* or the "WithQuotas" string in *quotas*, the method will return only the free space of the disk quota.

getURL( )

String **getURL**([Boolean | String *encoding*])

Parameter	Type	Description
encoding	Boolean, String	true or "Encoded" = encode the URL, false or "Not Encoded" = do not encode the URL (default)
Returns	String	URL of the referenced folder

Description

The **getURL( )** method returns the absolute URL of the *File* or *Folder* object.

By default, the returned URL characters are not encoded. You can set the encoding mode by using the *encoding*

parameter:

- If you pass *true* or the "Encoded" string in *encoding*, the URL will be encoded.
- If you pass *false* or the "Not Encoded" string in *encoding* (or omit the parameter), the URL is not encoded. This is the default action.

Example

The example below returns the URL from the path of a folder:

```
var myfolder = Folder("c:/wk/web/img/");
var url = myfolder.getURL(); // will return "file:///c:/wk/web/img/"
```

getVolumeSize( )

Number **getVolumeSize**([Boolean | String *quotas*])

Parameter	Type	Description
quotas	Boolean, String	true or "NoQuotas" = consider the whole volume (default), false or "WithQuotas" = consider only the allowed size for the quota
Returns	Number	Size in bytes of the volume where the folder is stored

Description

The **getVolumeSize( )** method returns the size (expressed in bytes) of the volume where the *File* or *Folder* object is stored. The referenced file or folder must exist on disk.

If system disk quotas have been activated on the volume where the file or folder is stored, you can get the total size of the volume or only your disk quota size, depending on the value you pass in the *quotas* parameter:

- If you pass *true* or the "NoQuotas" string in *quotas* (or omit the parameter), this method returns the whole size of the disk. This is the default action.
- If you pass *false* or the "WithQuotas" string in *quotas*, this method returns the size of the disk quota.

isFolder( )

Boolean **isFolder**(String *path*)

Parameter	Type	Description
path	String	Posix path
Returns	Boolean	True if the path corresponds to a folder, False otherwise

Description

The **isFolder( )** class method can be used with the **Folder( )** constructor to know if *path* corresponds to a folder on disk.

Pass in *path* a Posix string containing an absolute or relative path to a folder on the disk. If the path corresponds to a folder, **isFolder( )** returns True. If the path corresponds to a non-existing folder or a file, **isFolder( )** returns False.

Example

We want to know if "reports.proj" is a folder:

```
var myFolder = Folder.isFolder("C:/wakanda/projects/reports.proj");
```

next( )

Boolean **next**

Returns	Boolean	True if there is a next subfolder in the iteration. Otherwise, it returns false.

Description

The **next( )** method works with the subfolder iterator. It puts the folder pointer on the next subfolder in an iteration of subfolders, for example in a **for** loop.

The method returns *true* if it has been executed correctly and *false* otherwise. This method returns *false* when it reaches

the end of a folder collection, i.e., the subfolders stored in a *Folder* object.

parse( )

void **parse**(Function *callbackFn* [, Object *thisArg*])

Parameter	Type	Description
callbackFn	Function	Handler function to call for each file and subfolder
thisArg	Object	Token object that will be bound to 'this' in the handler

Description

The **parse( )** method executes the *callbackFn* function once for each file or subfolder present in the *Folder* object in ascending order. This method also executes the *callbackFn* function on subfolders in the subfolders recursively.

The *callbackFn* function accepts three parameters: *item*, *iterator*, and *theFolder*.

- The first parameter, *item*, represents the file currently being processed. When the function is executed, this parameter receives the file on which it iterates. You can then perform any type of operation on the current item.
- The second (optional) parameter, *iterator*, is the iterator. When the function is executed, this parameter receives the position of the file currently being processed in the parent folder. You can use it, for example, to display a counter.
- The third (optional) parameter, *theFolder*, receives the parent *Folder* being processed.

If a *thisArg* parameter is provided, it will be used as the **this** value each time the *callbackFn* is called. If it is not provided, **undefined** is used instead.

If existing elements (files or subfolders) of the folder are modified, their values as passed to callback will be the values at the time **parse( )** visits them. Items that are deleted after the call to **parse( )** begins and before being visited are not visited.

Example

In a "Messages" folder containing several text files stored in several subfolders, this example loads and concatenates the contents of each file into a single text variable. It also handles errors that may occur.

```
var folder = Folder("c:/Messages/");
var result = [];
var textResult = "";
folder.parse(function(file, position) // for each file in the folder
{
    var ext = file.extension.toLowerCase(); // get .TXT or.Text
    if (ext == "text" || ext == "txt")
    {
        result.push(file.path); // put the file in an array
        try  // error handling block
        {
            var text = loadText(file);  // we already know that files are of type Text
            if (text != null)  // do not take empty files
                textResult += "pos# "+position+" : "+text + "\n";  // write the position a
        }
        catch (err)
        {
            textResult += "*** " + file.path+" could not be loaded *** \n";
        }
    }
});
textResult; // display the text variable in the Wakanda Code Editor
```

Example

In this example, we count the files with the read-only and read/write status in a folder (at all sublevels):

```
var folder = Folder("c:/wakanda/");
var result = { readOnlyCount: 0, readWriteCount: 0 } // initialize the count to zero
folder.parse(function(file) // retrieve the 'file' parameter
{
    if (file.readOnly)
        result.readOnlyCount++;
    else
        result.readWriteCount++;
```

```
});
result; // display the resulting object in the Wakanda Code Editor
```

remove( )

Boolean **remove**( )

Returns	Boolean	True if the folder was removed successfully. Otherwise, it returns false.

Description

Note: You can also call this method using the drop( ) name.

The **remove( )** method removes the file or folder referenced in the *File* or *Folder* object from the storage volume. The file or folder referenced in the object must already exist on disk, otherwise this method returns a "File not found" error. In the case of folders, this method deletes the folder as well as all of its contents.

The **remove( )** method returns *true* if the file or folder is removed successfully. In all other cases, it returns *false*.

removeContent( )

Boolean **removeContent**( )

Returns	Boolean	True if the folder's contents were removed successfully. Otherwise, it returns false.

Description

Note: You can also call this method by using the dropContent( ) name.

The **removeContent( )** method removes the contents of the folder referenced in the *Folder* object from the storage volume. The folder referenced in the object must already exist on disk, otherwise the method returns a "File not found" error.
Only the contents of the folder is removed (files and subfolders), but the folder itself is left untouched.

The **removeContent( )** method returns *true* if the folder's content are successfully removed. Otherwise, it returns *false*.

setName( )

void **setName**(String *newName*)

Parameter	**Type**	**Description**
newName	String	New name for the folder on disk

Description

The **setName( )** method allows you to rename the folder referenced in the *Folder* object on disk. For the method to work, the folder must already exist on disk and the application should have appropriate write permissions to do so.

In the *newName* parameter, pass the folder's new name. The string must comply with the OS file naming rules.

If the method was successful, it returns *true*. If a problem occurred (a folder with the same name already exists, no write permission, etc.), the method returns *false*.

Note that this method will rename the folder on the disk, but not the folder name referenced in the *Folder* object.

valid( )

Boolean **valid**( )

Returns	Boolean	True if a current folder exists. Otherwise, it returns false.

Description

The **valid( )** method works with the subfolder iterator. It checks the validity of the pointer to the current folder within an iteration of folders, for example in a **for** loop.

This method returns *true* if the pointer is valid and *false* if it is not.

TextStream Class

TextStream class methods allow you to handle and parse text-based streams mapped to disk files. Unlike the **BinaryStream Class** methods, TextStream methods support a *charSet* parameter for encoding/decoding streams.

To create a *TextStream* object, you need to execute the **TextStream( )** constructor method (method from the Application class).

Note: If you need to load all the contents of a text file at once, you might also consider using the loadText( ) from the Application class.

close( )

void **close**

Description

The **close( )** method closes the file referenced in the *TextStream* object.

The referenced file is opened when you execute the method and stays open until you call **close( )**.

end( )

Boolean **end**( )

Returns Boolean true when the cursor is beyond the last character of the file (in buffer)

Description

The **end( )** method returns *true* if the cursor position is after the last character of the file referenced in the *TextStream* object.

As long as the cursor is located within the file, the **end( )** method returns *false*.

This method is useful when you want to read a large file. For example, you can read a file line by line (by using the method with an empty string) until **end( )** returns *true*.

flush( )

void **flush**( )

Description

The **flush( )** method saves the contents of the buffer to the disk file referenced in the *TextStream* object.

When you execute several method calls to write data into a *TextStream* object, for optimization reasons the data is stored in a buffer that is saved to disk when the stream is closed. This method allows you to save the buffer at any time during the process without having to close the stream.

getPos( )

Number **getPos**( )

Returns Number Position of the cursor in the stream

Description

The **getPos( )** method returns the current position of the cursor in the *TextStream* object.

getSize( )

Number **getSize**( )

Returns Number Current size of the stream (in bytes)

Description

The **getSize( )** method returns the current size of the stream.

When you are reading the stream, this method returns the size of the file referenced in the *TextStream* object.

When you are writing the stream, this method returns the current size of the stream in memory.

read( )

String **read**([Number | String *numBytesOrDelimiter*])

Parameter	Type	Description
numBytesOrDelimiter	Number, String	Number of bytes to read or Character at which to stop reading
Returns	String	Received data

Description

The **read( )** method reads characters from the file referenced in the *TextStream* object.

The characters read are returned as a string value.

- To read a particular number of characters, pass this number in *numBytesOrDelimiter*. This number of characters will be returned and the internal cursor position will be updated. You can read up to 2GB of text.
  Note: The value you pass actually represents the number of bytes. Characters are usually encoded on one byte, but certain characters are encoded on two bytes, like accented characters. The correspondence between the number of characters/number of bytes may vary.
- To read the whole contents of the file, pass 0 in *numBytesOrDelimiter* or omit the parameter.
- To read data until a particular character is encountered, pass that character in *numBytesOrDelimiter*. This method will return all the characters in the file from the current cursor position until it finds the delimiter string (which is not returned). In this case, if the delimiter string is not found, the **read( )** method will read to the end of the file.

If you pass an empty string ("") in *numBytesOrDelimiter*, this method will return one line of text -- it returns all the characters until it finds an end-of-line character (*carriage return* or *line feed*).

When reading a file, the first **read( )** method begins at the beginning of the file. Reading subsequent data begins at the character following the last byte read.

When attempting to read past the end of the file, **read( )** will return the data read up to that point and the **end( )** method will return *true*. Then, the next **read( )** will return an empty string.

setPos( )

void **setPos**(Number *offset*)

Parameter	Type	Description
offset	Number	New cursor position in the stream

Description

The **setPos( )** method moves the stream cursor to the *offset* position in the *TextStream* object.

TextStream( )

TextStream **TextStream**(String | File *file* , String *readMode* [, Number *charset*])

Parameter	Type	Description
file	String, File	Binary text file to reference
readMode	String	Streaming action: "Write" to write data, "Read" to read data, and "Overwrite" to replace the file with new data.
charset	Number	Character set of the text. By default, the value is 7, which is UTF-8.
Returns	TextStream	New TextStream object

Description

The **TextStream( )** method creates a new *TextStream* object. *TextStream* objects are handled using the various properties and methods of the **TextStream Class**.

In the *file* parameter, pass the path of the text file or a reference to it. The value can be either:

- an absolute path (using the "/" separator) or a URL, including the file name or
- a valid *File* object

Once the file is referenced, you can start writing or reading the stream data depending on the value you passed in the *readMode* parameter:

- If you passed "Write", the file is opened in write mode.
- If you passed "Read", the file is opened in read mode.
- If you passed "Overwrite", the file is replaced by the data you will write.

The *charSet* parameter is optional. It can be used to indicate a charset that is different from the default one (UTF-8). This parameter takes an integer as a value. Setting a charset overrides the default charset unless a BOM is detected in the text in which case the BOM's charset is used. Here is a list of the most common accepted values:

- -2 - ANSI
- 0 - Unknown
- 1 - UTF-16 Big Endian
- 2 - UTF-16 Little Endian
- 3 - UTF-32 Big Endian
- 4 - UTF-32 Little Endian
- 5 - UTF-32 Raw Big Endian
- 6 - UTF-32 Raw Little Endian
- 7 - UTF-8
- 8 - UTF-7
- 9 - ASCII
- 10 - EBCDIC
- 11 - IBM code page 437
- 100 - Mac OS Roman
- 101 - Windows Roman
- 102 - Mac OS Central Europe
- 103 - Windows Central Europe
- 104 - Mac OS Cyrillic
- 105 - Windows Cyrillic
- 106 - Mac OS Greek
- 107 - Windows Greek
- 108 - Mac OS Turkish
- 109 - Windows Turkish
- 110 - Mac OS Arabic
- 111 - Windows Arabic
- 112 - Mac OS Hebrew
- 113 - Windows Hebrew
- 114 - Mac OS Baltic
- 115 - Windows Baltic
- 116 - Mac OS Simplified Chinese
- 117 - Windows Simplified Chinese
- 118 - Mac OS Traditional Chinese
- 119 - Windows Traditional Chinese
- 120 - Mac OS Japanese
- 1000 - Shift-JIS (Japan, Mac/Win)
- 1001 - JIS (Japan, ISO-2022-JP, for emails)
- 1002 - BIG5, Chinese (Traditional)
- 1003 - EUC-KR, Korean
- 1004 - KOI8-R, Cyrillic
- 1005 - ISO 8859-1, Western Europe
- 1006 - ISO 8859-2, Central/Eastern Europe (CP1250)
- 1007 - ISO 8859-3, Southern Europe
- 1008 - ISO 8859-4, Baltic/Northern Europe
- 1009 - ISO 8859-5, Cyrillic
- 1010 - ISO 8859-6, Arab
- 1011 - ISO 8859-7, Greek
- 1012 - ISO 8859-8, Hebrew
- 1013 - ISO 8859-9, Turkish
- 1014 - ISO 8859-10, Nordic and Baltic languages (not available on Windows)
- 1015 - ISO 8859-13, Baltic Rim countries (not available on Windows)
- 1016 - GB2312, Chinese (Simplified)
- 1017 - GB2312-80, Chinese (Simplified)
- 1018 - ISO 8859-15, ISO-Latin-9
- 1019 - Windows-31J (code page 932)

Example

We want to implement a Log function that we could call to create new log files and append messages at any moment.

Using text streams is very useful in this case:

```
function Log(file)  // Constructor function definition
{
    var log =
    {
        appendToLog: function (myMessage)   // append function
        {
            var file = this.logFile;
            if (file != null)
            {
                if (!file.exists)  // if the file does not exist
                    file.create();  // create it
                var stream = TextStream(file, "write");  // open the stream in write mode
                stream.write(myMessage+"\n"); // append the message to the end of stream
                stream.close(); // do not forget to close the stream
            }
        },

        init: function(file)  // to initialize the log
        {
            this.logFile = file;
            if (file.exists)
                file.remove();
            file.create();
        },

        set: function(file)  // to create the log file
        {
            if (typeof file == "string") // only text files can be created
                file = File(file);
            this.logFile = file;
        },

        logFile: null
    }

    log.set(file);

    return log;
}
```

We can then create any log file we want and add messages in a very simple way, for example:

```
var log = new Log("c:/wakanda/mylog.txt"); // Creates a log file
var log2 = new Log("c:/wakanda/mylog2.txt"); // Creates another log file
log.appendToLog("*** First log file header***");
log2.appendToLog("*** Second log file header***");
log.appendToLog("First log entry in log1");
log2.appendToLog("First log entry in log2");
```

write( )

void **write**(String *text*)

Parameter	Type	Description
text	String	Data of type Text to write in the stream

Description

The **write( )** method writes the data you passed in the *text* parameter in the *TextStream* object. The text is written at the current position of the cursor in the stream. The cursor position is updated afterwards.