# file

The File plugin provides functionality to work with files.

Main features are:

- Creating temporary files
- Reading from and writing to the filesystem
- Retrieving information on files and directories on the file system
- Streaming files between the Smart Client and the Server (and vise versa)
  The stream methods are not directly useful for the WebClient, because this is really streaming of files between a client and a server, and the client code of the WebClient is already running on the server, so you are the streaming data withing the same process. The streaming for the webclientis handled by the browser code.

When using the File plugin, it's important to take into account the differences and compatibility between different clients:

- Interacting with the file system through the plugin in a Smart Client happens client-side, so on the machine where the Smart Client is launched. On all other clients (Web, Headless & Batchprocessor), the operations are performed on the Server.
- All showXxxxDialog(...) functions interact with the user through a UI. These function can only be used in Clients that provide a UI, like the Smart and Web Client.
- The showXxxxDialog(...) functions, when used in the Web Client, have certain limitations due to being operated in a browser. Browser security (currently) limits interaction with the local file system, except for single file select operations initiated by the user clicking a button.

Many functions work only in the smart client or only on the server in the Headless or WebClient, some work in the WebClient's browser and that is mentioned in the doc (WebClient enabled) most of the time an extra parameter or some certain state must be set like a callback function, because in the WebClient the function will not wait for a result and then return the result, but will go on, and then the callback is called when it is done. (like showFileOpenDialog)
You have no control directly of the real files in a a WebClient, so the files on the pc of the browser. You can't directly target them to load into the client, or with saving you can't specify the target directory where it is getting saved. This is all fully controlled by the browser.
When using the streaming features of the File plugin, make sure that you set the directory "servoy.FileServerService.defaultFolder" of the File server plugin settings in the admin page. Don't reply on the default behavior, specify your own writable directory.

Because the File plugin works with 3 different kind of files (LocalFiles on disk, RemoteFiles on the server when streaming is used and Uploaded files from the webclient when the browser uploads a file), not all things that you call on a JSFile object will give you data that you expect, for example a Uploaded file that you get back in the callback method of a showFileOpenDialog in the webclient does not represent an actual File object on the server. It is a binary data object (blob) that is only in memory, so getParentFile() or getPath() will return null in these cases, you don't get access or a reference to the file path or directory of the browsers pc. Also setBytes() will result in an exception. Best things to use is the getBytes() method if you want to get the bytes and store that, that will work for all 3 types. If you want to store the bytes that a webclient uploaded in another file the quickest way to do that is to use:

```
var fileOnServer = plugins.file.convertToJSFile('/path/to/file/on/server');

plugins.file.writeFile(fileOnServer,uploadedFile.getBytes());
```

## Return Types

JSFile JSProgressMonitor

## Server Property Summary

servoy.FileServerService.defaultFolder

## Method Summary

| | |
|---|---|
| Boolean | appendToTXTFile(file, text)<br>Appends a string given in parameter to a file, using default platform encoding. |
| Boolean | appendToTXTFile(file, text, encoding)<br>Appends a string given in parameter to a file, using the specified encoding. |
| Boolean | appendToTXTFile(file, text)<br>Appends a string given in parameter to a file, using default platform encoding. |
| Boolean | appendToTXTFile(file, text, encoding) |
| JSFile | convertToJSFile(file)<br>Returns a JSFile instance corresponding to an alternative representation of a file (for example a string). |
| JSFile | convertToRemoteJSFile(path)<br>Convenience return to get a JSFile representation of a server file based on its path. |
| Boolean | copyFile(source, destination)<br>Copies the source file to the destination file. |
| Boolean | copyFolder(source, destination)<br>Copies the sourcefolder to the destination folder, recursively. |

| | |
|---|---|
| JSFile | createFile(targetFile)<br>Creates a JSFile instance. |
| Boolean | createFolder(destination)<br>Creates a folder on disk. |
| JSFile | createTempFile(prefix, suffix)<br>Creates a temporary file on disk. |
| Boolean | deleteFile(destination)<br>Removes a file from disk. |
| Boolean | deleteFolder(destination, showWarning)<br>Deletes a folder from disk recursively. |
| String | getDefaultUploadLocation()<br>Returns the default upload location path of the server. |
| JSFile | getDesktopFolder()<br>Returns a JSFile instance that corresponds to the Desktop folder of the currently logged in user. |
| JSFile[] | getDiskList()<br>Returns an Array of JSFile instances correponding to the file system root folders. |
| Number | getFileSize(path)<br>Returns the size of the specified file. |
| JSFile[] | getFolderContents(targetFolder)<br>Returns an array of JSFile instances corresponding to content of the specified folder. |
| JSFile[] | getFolderContents(targetFolder, fileFilter)<br>Returns an array of JSFile instances corresponding to content of the specified folder. |
| JSFile[] | getFolderContents(targetFolder, fileFilter, fileOption)<br>Returns an array of JSFile instances corresponding to content of the specified folder. |
| JSFile[] | getFolderContents(targetFolder, fileFilter, fileOption, visibleOption)<br>Returns an array of JSFile instances corresponding to content of the specified folder. |
| JSFile[] | getFolderContents(targetFolder, fileFilter, fileOption, visibleOption, lockedOption)<br>Returns an array of JSFile instances corresponding to content of the specified folder. |
| JSFile[] | getFolderContents(targetFolder)<br>Returns an array of JSFile instances corresponding to content of the specified folder. |
| JSFile[] | getFolderContents(targetFolder, fileFilter)<br>Returns an array of JSFile instances corresponding to content of the specified folder. |
| JSFile[] | getFolderContents(targetFolder, fileFilter, fileOption)<br>Returns an array of JSFile instances corresponding to content of the specified folder. |
| JSFile[] | getFolderContents(targetFolder, fileFilter, fileOption, visibleOption)<br>Returns an array of JSFile instances corresponding to content of the specified folder. |
| JSFile[] | getFolderContents(targetFolder, fileFilter, fileOption, visibleOption, lockedOption)<br>Returns an array of JSFile instances corresponding to content of the specified folder. |
| JSFile | getHomeFolder()<br>Returns a JSFile instance corresponding to the home folder of the logged in used. |
| Date | getModificationDate(path)<br>Returns the modification date of a file. |
| JSFile[] | getRemoteFolderContents(targetFolder)<br>Returns an array of JSFile instances corresponding to content of the specified folder on the server side. |
| JSFile[] | getRemoteFolderContents(targetFolder, fileFilter)<br>Returns an array of JSFile instances corresponding to content of the specified folder on the server side. |
| JSFile[] | getRemoteFolderContents(targetFolder, fileFilter, fileOption)<br>Returns an array of JSFile instances corresponding to content of the specified folder on the server side. |
| JSFile[] | getRemoteFolderContents(targetFolder, fileFilter, fileOption, visibleOption)<br>Returns an array of JSFile instances corresponding to content of the specified folder on the server side. |
| JSFile[] | getRemoteFolderContents(targetFolder, fileFilter, fileOption, visibleOption, lockedOption)<br>Returns an array of JSFile instances corresponding to content of the specified folder on the server side. |
| JSFile[] | getRemoteFolderContents(targetFolder)<br>Returns an array of JSFile instances corresponding to content of the specified folder on the server side. |
| JSFile[] | getRemoteFolderContents(targetFolder, fileFilter)<br>Returns an array of JSFile instances corresponding to content of the specified folder on the server side. |
| JSFile[] | getRemoteFolderContents(targetFolder, fileFilter, fileOption)<br>Returns an array of JSFile instances corresponding to content of the specified folder on the server side. |
| JSFile[] | getRemoteFolderContents(targetFolder, fileFilter, fileOption, visibleOption)<br>Returns an array of JSFile instances corresponding to content of the specified folder on the server side. |
| JSFile[] | getRemoteFolderContents(targetFolder, fileFilter, fileOption, visibleOption, lockedOption)<br>Returns an array of JSFile instances corresponding to content of the specified folder on the server side. |
| Boolean | moveFile(source, destination)<br>Moves the file from the source to the destination place. |
| Boolean | openFile(file)<br>Launches the associated application to open the file. |
| byte[] | readFile()<br>Reads all or part of the content from a binary file. |
| byte[] | readFile(file)<br>Reads all or part of the content from a binary file. |
| byte[] | readFile(file, size)<br>Reads all or part of the content from a binary file. |
| byte[] | readFile(file)<br>Reads all or part of the content from a binary file. |
| byte[] | readFile(file, size)<br>Reads all or part of the content from a binary file. |

| | |
|---|---|
| String | readTXTFile()<br>Read all content from a text file. |
| String | readTXTFile(file)<br>Read all content from a text file. |
| String | readTXTFile(file, charsetname)<br>Read all content from a text file. |
| String | readTXTFile(file)<br>Read all content from a text file. |
| String | readTXTFile(file, charsetname)<br>Read all content from a text file. |
| JSFile | showDirectorySelectDialog()<br>Shows a directory selector dialog. |
| JSFile | showDirectorySelectDialog(directory)<br>Shows a directory selector dialog. |
| JSFile | showDirectorySelectDialog(directory, title)<br>Shows a directory selector dialog. |
| JSFile | showDirectorySelectDialog(directory)<br>Shows a directory selector dialog. |
| JSFile | showDirectorySelectDialog(directory, title)<br>Shows a directory selector dialog. |
| Object | showFileOpenDialog()<br>Shows a file open dialog. |
| Object | showFileOpenDialog(selectionMode)<br>Shows a file open dialog. |
| Object | showFileOpenDialog(selectionMode, startDirectory)<br>Shows a file open dialog. |
| Object | showFileOpenDialog(selectionMode, startDirectory, multiselect)<br>Shows a file open dialog. |
| Object | showFileOpenDialog(selectionMode, startDirectory, multiselect, filter)<br>Shows a file open dialog. |
| Object | showFileOpenDialog(selectionMode, startDirectory, multiselect, filter, callbackfunction)<br>Shows a file open dialog. |
| Object | showFileOpenDialog(selectionMode, startDirectory, multiselect, filter, callbackfunction, title)<br>Shows a file open dialog. |
| Object | showFileOpenDialog(selectionMode, startDirectory, multiselect, callbackfunction)<br>Shows a file open dialog. |
| Object | showFileOpenDialog(selectionMode, startDirectory, callbackfunction)<br>Shows a file open dialog. |
| Object | showFileOpenDialog(selectionMode, startDirectory)<br>Shows a file open dialog. |
| Object | showFileOpenDialog(selectionMode, startDirectory, multiselect)<br>Shows a file open dialog. |
| Object | showFileOpenDialog(selectionMode, startDirectory, multiselect, filter)<br>Shows a file open dialog. |
| Object | showFileOpenDialog(selectionMode, startDirectory, multiselect, filter, callbackfunction)<br>Shows a file open dialog. |
| Object | showFileOpenDialog(selectionMode, startDirectory, multiselect, filter, callbackfunction, title)<br>Shows a file open dialog. |
| Object | showFileOpenDialog(selectionMode, startDirectory, multiselect, callbackfunction)<br>Shows a file open dialog. |
| Object | showFileOpenDialog(selectionMode, startDirectory, callbackfunction)<br>Shows a file open dialog. |
| Object | showFileOpenDialog(selectionMode, callbackfunction)<br>Shows a file open dialog. |
| Object | showFileOpenDialog(callbackfunction)<br>Shows a file open dialog. |
| JSFile | showFileSaveDialog()<br>Shows a file save dialog. |
| JSFile | showFileSaveDialog(fileNameDir)<br>Shows a file save dialog. |
| JSFile | showFileSaveDialog(fileNameDir, title)<br>Shows a file save dialog. |
| JSFile | showFileSaveDialog(fileNameDir)<br>Shows a file save dialog. |
| JSFile | showFileSaveDialog(fileNameDir, title)<br>Shows a file save dialog. |
| JSProgressMonitor | streamFilesFromServer(files, serverFiles)<br>Stream 1 or more files from the server to the client. |
| JSProgressMonitor | streamFilesFromServer(files, serverFiles, callback)<br>Stream 1 or more files from the server to the client, the callback method is invoked after every file, with as argument the filename that was transfered. |
| JSProgressMonitor | streamFilesToServer(files)<br>Overloaded method, only defines file(s) to be streamed |
| JSProgressMonitor | streamFilesToServer(files, serverFiles)<br>Overloaded method, defines file(s) to be streamed and a callback function |

| | |
|---|---|
| JSProgressMonitor | streamFilesToServer(files, serverFiles, callback)<br>Overloaded method, defines file(s) to be streamed, a callback function and file name(s) to use on the server |
| JSProgressMonitor | streamFilesToServer(files, callback)<br>Overloaded method, defines file(s) to be streamed and a callback function |
| Boolean | writeFile(file, data)<br>Writes data into a binary file. |
| Boolean | writeFile(file, data, mimeType)<br>Writes data into a binary file. |
| Boolean | writeFile(file, data)<br>Writes data into a binary file. |
| Boolean | writeFile(file, data, mimeType)<br>Writes data into a binary file. |
| Boolean | writeTXTFile(file, text_data)<br>Writes data into a text file. |
| Boolean | writeTXTFile(file, text_data, charsetname)<br>Writes data into a text file. |
| Boolean | writeTXTFile(file, text_data, charsetname, mimeType)<br>Writes data into a text file. |
| Boolean | writeTXTFile(file, text_data)<br>Writes data into a text file. |
| Boolean | writeTXTFile(file, text_data, charsetname)<br>Writes data into a text file. |
| Boolean | writeTXTFile(file, text_data, charsetname, mimeType)<br>Writes data into a text file. |
| Boolean | writeXMLFile(file, xml_data)<br>Writes data into an XML file. |
| Boolean | writeXMLFile(file, xml_data, encoding)<br>Writes data into an XML file. |
| Boolean | writeXMLFile(file, xml_data)<br>Writes data into an XML file. |
| Boolean | writeXMLFile(file, xml_data, encoding)<br>Writes data into an XML file. |

## Server Property Details

### servoy.FileServerService.defaultFolder

## Method Details

### appendToTXTFile

Boolean **appendToTXTFile** (file, text)

Appends a string given in parameter to a file, using default platform encoding.

**Since**

Servoy 5.2

**Parameters**

{JSFile} file - a local JSFile
{String} text - the text to append to the file

**Returns**

Boolean - true if appending worked

**Sample**

```
// append some text to a text file:
        var ok = plugins.file.appendToTXTFile('myTextFile.txt', '\nMy fantastic new line of text\n');
```

### appendToTXTFile

Boolean **appendToTXTFile** (file, text, encoding)

Appends a string given in parameter to a file, using the specified encoding.

**Since**

Servoy 5.2

**Parameters**

{JSFile} file - a local JSFile
{String} text - the text to append to the file
{String} encoding - the encoding to use

**Returns**

Boolean - true if appending worked

**Sample**

```
// append some text to a text file:
	var ok = plugins.file.appendToTXTFile('myTextFile.txt', '\nMy fantastic new line of text\n');
```

### appendToTXTFile

Boolean **appendToTXTFile** (file, text)

Appends a string given in parameter to a file, using default platform encoding.

**Since**

Servoy 5.2

**Parameters**

{String} file - the file path as a String
{String} text - the text to append to the file

**Returns**

Boolean - true if appending worked

**Sample**

```
// append some text to a text file:
	var ok = plugins.file.appendToTXTFile('myTextFile.txt', '\nMy fantastic new line of text\n');
```

### appendToTXTFile

Boolean **appendToTXTFile** (file, text, encoding)

**Parameters**

{String} file - the file path as a String
{String} text - the text to append to the file
{String} encoding - the encoding to use

**Returns**

Boolean

**Sample**

```
// append some text to a text file:
	var ok = plugins.file.appendToTXTFile('myTextFile.txt', '\nMy fantastic new line of text\n');
```

### convertToJSFile

JSFile **convertToJSFile** (file)

Returns a JSFile instance corresponding to an alternative representation of a file (for example a string).

**Parameters**

{Object} file

**Returns**

JSFile - JSFile

**Sample**

```
var f = plugins.file.convertToJSFile("story.txt");
if (f.canRead())
	application.output("File can be read.");
```

### convertToRemoteJSFile

JSFile **convertToRemoteJSFile** (path)

Convenience return to get a JSFile representation of a server file based on its path.

**Since**

Servoy 5.2

**Parameters**

{String} path - the path representing a file on the server (should start with "/")

**Returns**

JSFile - the JSFile

**Sample**

```
var f = plugins.file.convertToRemoteJSFile('/story.txt');
if (f && f.canRead())
	application.output('File can be read.');
```

### copyFile

Boolean **copyFile** (source, destination)

Copies the source file to the destination file. Returns true if the copy succeeds, false if any error occurs.

**Parameters**

{Object} source
{Object} destination

**Returns**

Boolean

**Sample**

```
// Copy based on file names.
if (!plugins.file.copyFile("story.txt", "story.txt.copy"))
	application.output("Copy failed.");
// Copy based on JSFile instances.
var f = plugins.file.createFile("story.txt");
var fcopy = plugins.file.createFile("story.txt.copy2");
if (!plugins.file.copyFile(f, fcopy))
	application.output("Copy failed.");
```

### copyFolder

Boolean **copyFolder** (source, destination)

Copies the sourcefolder to the destination folder, recursively. Returns true if the copy succeeds, false if any error occurs.

**Parameters**

{Object} source
{Object} destination

**Returns**

Boolean - success boolean

**Sample**

```
// Copy folder based on names.
if (!plugins.file.copyFolder("stories", "stories_copy"))
	application.output("Folder copy failed.");
// Copy folder based on JSFile instances.
var d = plugins.file.createFile("stories");
var dcopy = plugins.file.createFile("stories_copy_2");
if (!plugins.file.copyFolder(d, dcopy))
	application.output("Folder copy failed.");
```

### createFile

JSFile **createFile** (targetFile)

Creates a JSFile instance. Does not create the file on disk.

**Parameters**

{Object} targetFile

**Returns**

JSFile

**Sample**

```
// Create the JSFile instance based on the file name.
var f = plugins.file.createFile("newfile.txt");
// Create the file on disk.
if (!f.createNewFile())
        application.output("The file could not be created.");
```

### createFolder

Boolean **createFolder** (destination)

Creates a folder on disk. Returns true if the folder is successfully created, false if any error occurs.

**Parameters**

{Object} destination

**Returns**

Boolean

**Sample**

```
var d = plugins.file.convertToJSFile("newfolder");
if (!plugins.file.createFolder(d))
        application.output("Folder could not be created.");
```

### createTempFile

JSFile **createTempFile** (prefix, suffix)

Creates a temporary file on disk. A prefix and an extension are specified and they will be part of the file name.

**Parameters**

{String} prefix
{String} suffix

**Returns**

JSFile

**Sample**

```
var tempFile = plugins.file.createTempFile('myfile','.txt');
application.output('Temporary file created as: ' + tempFile.getAbsolutePath());
plugins.file.writeTXTFile(tempFile, 'abcdefg');
```

### deleteFile

Boolean **deleteFile** (destination)

Removes a file from disk. Returns true on success, false otherwise.

**Parameters**

{Object} destination

**Returns**

Boolean

**Sample**

```
if (plugins.file.deleteFile('story.txt'))
        application.output('File deleted.');
```

### deleteFolder

Boolean **deleteFolder** (destination, showWarning)

Deletes a folder from disk recursively. Returns true on success, false otherwise. If the second parameter is set to true, then a warning will be issued to the user before actually removing the folder.

**Parameters**

{Object} destination
{Boolean} showWarning

**Returns**

Boolean

**Sample**

```
if (plugins.file.deleteFolder('stories', true))
        application.output('Folder deleted.');
```

### getDefaultUploadLocation

String **getDefaultUploadLocation** ()

Returns the default upload location path of the server.

**Returns**

String - the location as canonical path

**Sample**

```
// get the (server-side) default upload location path:
var serverPath = plugins.file.getDefaultUploadLocation();
```

### getDesktopFolder

JSFile **getDesktopFolder** ()

Returns a JSFile instance that corresponds to the Desktop folder of the currently logged in user.

**Returns**

JSFile

**Sample**

```
var d = plugins.file.getDesktopFolder();
application.output('desktop folder is: ' + d.getAbsolutePath());
```

### getDiskList

JSFile[] **getDiskList** ()

Returns an Array of JSFile instances correponding to the file system root folders.

**Returns**

JSFile[]

**Sample**

```
var roots = plugins.file.getDiskList();
for (var i = 0; i < roots.length; i++)
        application.output(roots[i].getAbsolutePath());
```

### getFileSize

Number **getFileSize** (path)

Returns the size of the specified file.

**Parameters**

{Object} path

**Returns**

Number

**Sample**

```
var f = plugins.file.convertToJSFile('story.txt');
application.output('file size: ' + plugins.file.getFileSize(f));
```

### getFolderContents

JSFile[] **getFolderContents** (targetFolder)

Returns an array of JSFile instances corresponding to content of the specified folder. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{JSFile} targetFolder - JSFile object.

**Returns**

JSFile[]

**Sample**

```
var files = plugins.file.getFolderContents('stories', '.txt');
for (var i=0; i<files.length; i++)
        application.output(files[i].getAbsolutePath());
```

### getFolderContents

JSFile[] **getFolderContents** (targetFolder, fileFilter)

Returns an array of JSFile instances corresponding to content of the specified folder. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{JSFile} targetFolder - JSFile object.
{Object} fileFilter - Filter or array of filters for files in folder.

**Returns**

JSFile[]

**Sample**

```
var files = plugins.file.getFolderContents('stories', '.txt');
for (var i=0; i<files.length; i++)
        application.output(files[i].getAbsolutePath());
```

### getFolderContents

JSFile[] **getFolderContents** (targetFolder, fileFilter, fileOption)

Returns an array of JSFile instances corresponding to content of the specified folder. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{JSFile} targetFolder - JSFile object.
{Object} fileFilter - Filter or array of filters for files in folder.
{Number} fileOption - 1=files, 2=dirs

**Returns**

JSFile[]

**Sample**

```
var files = plugins.file.getFolderContents('stories', '.txt');
for (var i=0; i<files.length; i++)
        application.output(files[i].getAbsolutePath());
```

### getFolderContents

JSFile[] **getFolderContents** (targetFolder, fileFilter, fileOption, visibleOption)

Returns an array of JSFile instances corresponding to content of the specified folder. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{JSFile} targetFolder - JSFile object.
{Object} fileFilter - Filter or array of filters for files in folder.
{Number} fileOption - 1=files, 2=dirs
{Number} visibleOption - 1=visible, 2=nonvisible

**Returns**

JSFile[]

**Sample**

```
var files = plugins.file.getFolderContents('stories', '.txt');
for (var i=0; i<files.length; i++)
        application.output(files[i].getAbsolutePath());
```

### getFolderContents

JSFile[] **getFolderContents** (targetFolder, fileFilter, fileOption, visibleOption, lockedOption)

Returns an array of JSFile instances corresponding to content of the specified folder. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{JSFile} targetFolder - JSFile object.
{Object} fileFilter - Filter or array of filters for files in folder.
{Number} fileOption - 1=files, 2=dirs
{Number} visibleOption - 1=visible, 2=nonvisible
{Number} lockedOption - 1=locked, 2=nonlocked

**Returns**

JSFile[]

**Sample**

```
var files = plugins.file.getFolderContents('stories', '.txt');
for (var i=0; i<files.length; i++)
        application.output(files[i].getAbsolutePath());
```

### getFolderContents

JSFile[] **getFolderContents** (targetFolder)

Returns an array of JSFile instances corresponding to content of the specified folder. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{String} targetFolder - File path.

**Returns**

JSFile[]

**Sample**

```
var files = plugins.file.getFolderContents('stories', '.txt');
for (var i=0; i<files.length; i++)
        application.output(files[i].getAbsolutePath());
```

### getFolderContents

JSFile[] **getFolderContents** (targetFolder, fileFilter)

Returns an array of JSFile instances corresponding to content of the specified folder. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{String} targetFolder - File path.
{Object} fileFilter - Filter or array of filters for files in folder.

**Returns**

JSFile[]

**Sample**

```
var files = plugins.file.getFolderContents('stories', '.txt');
for (var i=0; i<files.length; i++)
        application.output(files[i].getAbsolutePath());
```

### getFolderContents

JSFile[] **getFolderContents** (targetFolder, fileFilter, fileOption)

Returns an array of JSFile instances corresponding to content of the specified folder. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{String} targetFolder - File path.
{Object} fileFilter - Filter or array of filters for files in folder.
{Number} fileOption - 1=files, 2=dirs

**Returns**

JSFile[]

**Sample**

```
var files = plugins.file.getFolderContents('stories', '.txt');
for (var i=0; i<files.length; i++)
        application.output(files[i].getAbsolutePath());
```

### getFolderContents

JSFile[] **getFolderContents** (targetFolder, fileFilter, fileOption, visibleOption)

Returns an array of JSFile instances corresponding to content of the specified folder. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{String} targetFolder - File path.
{Object} fileFilter - Filter or array of filters for files in folder.
{Number} fileOption - 1=files, 2=dirs
{Number} visibleOption - 1=visible, 2=nonvisible

**Returns**

JSFile[]

**Sample**

```
var files = plugins.file.getFolderContents('stories', '.txt');
for (var i=0; i<files.length; i++)
        application.output(files[i].getAbsolutePath());
```

### getFolderContents

JSFile[] **getFolderContents** (targetFolder, fileFilter, fileOption, visibleOption, lockedOption)

Returns an array of JSFile instances corresponding to content of the specified folder. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{String} targetFolder - File path.
{Object} fileFilter - Filter or array of filters for files in folder.
{Number} fileOption - 1=files, 2=dirs
{Number} visibleOption - 1=visible, 2=nonvisible
{Number} lockedOption - 1=locked, 2=nonlocked

**Returns**

JSFile[]

**Sample**

```
var files = plugins.file.getFolderContents('stories', '.txt');
for (var i=0; i<files.length; i++)
        application.output(files[i].getAbsolutePath());
```

### getHomeFolder

JSFile **getHomeFolder** ()

Returns a JSFile instance corresponding to the home folder of the logged in used.

**Returns**

JSFile

**Sample**

```
var d = plugins.file.getHomeFolder();
application.output('home folder: ' + d.getAbsolutePath());
```

### getModificationDate

Date **getModificationDate** (path)

Returns the modification date of a file.

**Parameters**

{Object} path

**Returns**

Date

**Sample**

```
var f = plugins.file.convertToJSFile('story.txt');
application.output('last changed: ' + plugins.file.getModificationDate(f));
```

### getRemoteFolderContents

JSFile[] **getRemoteFolderContents** (targetFolder)

Returns an array of JSFile instances corresponding to content of the specified folder on the server side. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Since**

Servoy 5.2.1

**Parameters**

{JSFile} targetFolder

**Returns**

JSFile[] - the array of file names

**Sample**

```
// retrieves an array of files located on the server side inside the default upload folder:
var files = plugins.file.getRemoteFolderContents('/', '.txt');
```

### getRemoteFolderContents

JSFile[] **getRemoteFolderContents** (targetFolder, fileFilter)

Returns an array of JSFile instances corresponding to content of the specified folder on the server side. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{JSFile} targetFolder - Folder as JSFile object.
{Object} fileFilter - Filter or array of filters for files in folder.

**Returns**

JSFile[] - the array of file names

**Sample**

```
// retrieves an array of files located on the server side inside the default upload folder:
var files = plugins.file.getRemoteFolderContents('/', '.txt');
```

### getRemoteFolderContents

JSFile[] **getRemoteFolderContents** (targetFolder, fileFilter, fileOption)

Returns an array of JSFile instances corresponding to content of the specified folder on the server side. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{JSFile} targetFolder - Folder as JSFile object.
{Object} fileFilter - Filter or array of filters for files in folder.
{Number} fileOption - 1=files, 2=dirs

**Returns**

JSFile[] - the array of file names

**Sample**

```
// retrieves an array of files located on the server side inside the default upload folder:
var files = plugins.file.getRemoteFolderContents('/', '.txt');
```

### getRemoteFolderContents

JSFile[] **getRemoteFolderContents** (targetFolder, fileFilter, fileOption, visibleOption)

Returns an array of JSFile instances corresponding to content of the specified folder on the server side. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{JSFile} targetFolder - Folder as JSFile object.
{Object} fileFilter - Filter or array of filters for files in folder.
{Number} fileOption - 1=files, 2=dirs
{Number} visibleOption - 1=visible, 2=nonvisible

**Returns**

JSFile[] - the array of file names

**Sample**

```
// retrieves an array of files located on the server side inside the default upload folder:
var files = plugins.file.getRemoteFolderContents('/', '.txt');
```

### getRemoteFolderContents

JSFile[] **getRemoteFolderContents** (targetFolder, fileFilter, fileOption, visibleOption, lockedOption)

Returns an array of JSFile instances corresponding to content of the specified folder on the server side. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{JSFile} targetFolder - Folder as JSFile object.
{Object} fileFilter - Filter or array of filters for files in folder.
{Number} fileOption - 1=files, 2=dirs
{Number} visibleOption - 1=visible, 2=nonvisible
{Number} lockedOption - 1=locked, 2=nonlocked

**Returns**

JSFile[] - the array of file names

**Sample**

```
// retrieves an array of files located on the server side inside the default upload folder:
var files = plugins.file.getRemoteFolderContents('/', '.txt');
```

### getRemoteFolderContents

JSFile[] **getRemoteFolderContents** (targetFolder)

Returns an array of JSFile instances corresponding to content of the specified folder on the server side. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{String} targetFolder - Folder path.

**Returns**

JSFile[] - the array of file names

**Sample**

```
// retrieves an array of files located on the server side inside the default upload folder:
var files = plugins.file.getRemoteFolderContents('/', '.txt');
```

### getRemoteFolderContents

JSFile[] **getRemoteFolderContents** (targetFolder, fileFilter)

Returns an array of JSFile instances corresponding to content of the specified folder on the server side. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{String} targetFolder - Folder path.
{Object} fileFilter - Filter or array of filters for files in folder.

**Returns**

JSFile[] - the array of file names

**Sample**

```
// retrieves an array of files located on the server side inside the default upload folder:
var files = plugins.file.getRemoteFolderContents('/', '.txt');
```

### getRemoteFolderContents

JSFile[] **getRemoteFolderContents** (targetFolder, fileFilter, fileOption)

Returns an array of JSFile instances corresponding to content of the specified folder on the server side. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{String} targetFolder - Folder path.
{Object} fileFilter - Filter or array of filters for files in folder.
{Number} fileOption - 1=files, 2=dirs

**Returns**

JSFile[] - the array of file names

**Sample**

```
// retrieves an array of files located on the server side inside the default upload folder:
var files = plugins.file.getRemoteFolderContents('/', '.txt');
```

### getRemoteFolderContents

JSFile[] **getRemoteFolderContents** (targetFolder, fileFilter, fileOption, visibleOption)

Returns an array of JSFile instances corresponding to content of the specified folder on the server side. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{String} targetFolder - Folder path.
{Object} fileFilter - Filter or array of filters for files in folder.
{Number} fileOption - 1=files, 2=dirs
{Number} visibleOption - 1=visible, 2=nonvisible

**Returns**

JSFile[] - the array of file names

**Sample**

```
// retrieves an array of files located on the server side inside the default upload folder:
var files = plugins.file.getRemoteFolderContents('/', '.txt');
```

### getRemoteFolderContents

JSFile[] **getRemoteFolderContents** (targetFolder, fileFilter, fileOption, visibleOption, lockedOption)

Returns an array of JSFile instances corresponding to content of the specified folder on the server side. The content can be filtered by optional name filter(s), by type, by visibility and by lock status.

**Parameters**

{String} targetFolder - Folder path.
{Object} fileFilter - Filter or array of filters for files in folder.
{Number} fileOption - 1=files, 2=dirs
{Number} visibleOption - 1=visible, 2=nonvisible
{Number} lockedOption - 1=locked, 2=nonlocked

**Returns**

JSFile[] - the array of file names

**Sample**

```
// retrieves an array of files located on the server side inside the default upload folder:
var files = plugins.file.getRemoteFolderContents('/', '.txt');
```

### moveFile

Boolean **moveFile** (source, destination)

Moves the file from the source to the destination place. Returns true on success, false otherwise.

**Parameters**

{Object} source
{Object} destination

**Returns**

Boolean

**Sample**

```
// Move file based on names.
if (!plugins.file.moveFile('story.txt','story.txt.new'))
        application.output('File move failed.');
// Move file based on JSFile instances.
var f = plugins.file.convertToJSFile('story.txt.new');
var fmoved = plugins.file.convertToJSFile('story.txt');
if (!plugins.file.moveFile(f, fmoved))
        application.output('File move back failed.');
```

### openFile

Boolean **openFile** (file)

Launches the associated application to open the file.

**Parameters**

{JSFile} file

**Returns**

Boolean - success status of the open operation

**Sample**

```
var myPDF = plugins.file.createFile('my.pdf');
 myPDF.setBytes(data, true)
        plugins.file.openFile(myPDF);
```

### readFile

byte[] **readFile** ()

Reads all or part of the content from a binary file. If a file name is not specified, then a file selection dialog pops up for selecting a file. (Web Enabled only for a JSFile argument)

**Returns**

byte[]

**Sample**

```
// Read all content from the file.
var bytes = plugins.file.readFile('big.jpg');
application.output('file size: ' + bytes.length);
// Read only the first 1KB from the file.
var bytesPartial = plugins.file.readFile('big.jpg', 1024);
application.output('partial file size: ' + bytesPartial.length);
// Read all content from a file selected from the file open dialog.
var bytesUnknownFile = plugins.file.readFile();
application.output('unknown file size: ' + bytesUnknownFile.length);
```

### readFile

byte[] **readFile** (file)

Reads all or part of the content from a binary file. If a file name is not specified, then a file selection dialog pops up for selecting a file. (Web Enabled only for a JSFile argument)

**Parameters**

{JSFile} file - JSFile.

**Returns**

byte[]

**Sample**

```
// Read all content from the file.
var bytes = plugins.file.readFile('big.jpg');
application.output('file size: ' + bytes.length);
// Read only the first 1KB from the file.
var bytesPartial = plugins.file.readFile('big.jpg', 1024);
application.output('partial file size: ' + bytesPartial.length);
// Read all content from a file selected from the file open dialog.
var bytesUnknownFile = plugins.file.readFile();
application.output('unknown file size: ' + bytesUnknownFile.length);
```

### readFile

byte[] **readFile** (file, size)

Reads all or part of the content from a binary file. If a file name is not specified, then a file selection dialog pops up for selecting a file. (Web Enabled only for a JSFile argument)

**Parameters**

{JSFile} file - JSFile.
{Number} size - Number of bytes to read.

**Returns**

byte[]

**Sample**

```
// Read all content from the file.
var bytes = plugins.file.readFile('big.jpg');
application.output('file size: ' + bytes.length);
// Read only the first 1KB from the file.
var bytesPartial = plugins.file.readFile('big.jpg', 1024);
application.output('partial file size: ' + bytesPartial.length);
// Read all content from a file selected from the file open dialog.
var bytesUnknownFile = plugins.file.readFile();
application.output('unknown file size: ' + bytesUnknownFile.length);
```

### readFile

byte[] **readFile** (file)

Reads all or part of the content from a binary file. If a file name is not specified, then a file selection dialog pops up for selecting a file. (Web Enabled only for a JSFile argument)

**Parameters**

{String} file - the file path.

**Returns**

byte[]

**Sample**

```
// Read all content from the file.
var bytes = plugins.file.readFile('big.jpg');
application.output('file size: ' + bytes.length);
// Read only the first 1KB from the file.
var bytesPartial = plugins.file.readFile('big.jpg', 1024);
application.output('partial file size: ' + bytesPartial.length);
// Read all content from a file selected from the file open dialog.
var bytesUnknownFile = plugins.file.readFile();
application.output('unknown file size: ' + bytesUnknownFile.length);
```

### readFile

byte[] **readFile** (file, size)

Reads all or part of the content from a binary file. If a file name is not specified, then a file selection dialog pops up for selecting a file. (Web Enabled only for a JSFile argument)

**Parameters**

{String} file - the file path.
{Number} size - Number of bytes to read.

**Returns**

byte[]

**Sample**

```
// Read all content from the file.
var bytes = plugins.file.readFile('big.jpg');
application.output('file size: ' + bytes.length);
// Read only the first 1KB from the file.
var bytesPartial = plugins.file.readFile('big.jpg', 1024);
application.output('partial file size: ' + bytesPartial.length);
// Read all content from a file selected from the file open dialog.
var bytesUnknownFile = plugins.file.readFile();
application.output('unknown file size: ' + bytesUnknownFile.length);
```

### readTXTFile

String **readTXTFile** ()

Read all content from a text file. If a file name is not specified, then a file selection dialog pops up for selecting a file. The encoding can be also specified. (Web Enabled only for a JSFile argument)

**Returns**

String

**Sample**

```
// Read content from a known text file.
var txt = plugins.file.readTXTFile('story.txt');
application.output(txt);
// Read content from a text file selected from the file open dialog.
var txtUnknown = plugins.file.readTXTFile();
application.output(txtUnknown);
```

### readTXTFile

String **readTXTFile** (file)

Read all content from a text file. If a file name is not specified, then a file selection dialog pops up for selecting a file. The encoding can be also specified. (Web Enabled only for a JSFile argument)

**Parameters**

{JSFile} file - JSFile.

**Returns**

String

**Sample**

```
// Read content from a known text file.
var txt = plugins.file.readTXTFile('story.txt');
application.output(txt);
// Read content from a text file selected from the file open dialog.
var txtUnknown = plugins.file.readTXTFile();
application.output(txtUnknown);
```

### readTXTFile

String **readTXTFile** (file, charsetname)

Read all content from a text file. If a file name is not specified, then a file selection dialog pops up for selecting a file. The encoding can be also specified. (Web Enabled only for a JSFile argument)

**Parameters**

{JSFile} file - JSFile.
{String} charsetname - Charset name.

**Returns**

String

**Sample**

```
// Read content from a known text file.
var txt = plugins.file.readTXTFile('story.txt');
application.output(txt);
// Read content from a text file selected from the file open dialog.
var txtUnknown = plugins.file.readTXTFile();
application.output(txtUnknown);
```

### readTXTFile

String **readTXTFile** (file)

Read all content from a text file. If a file name is not specified, then a file selection dialog pops up for selecting a file. The encoding can be also specified. (Web Enabled only for a JSFile argument)

**Parameters**

{String} file - the file path.

**Returns**

String

**Sample**

```
// Read content from a known text file.
var txt = plugins.file.readTXTFile('story.txt');
application.output(txt);
// Read content from a text file selected from the file open dialog.
var txtUnknown = plugins.file.readTXTFile();
application.output(txtUnknown);
```

### readTXTFile

String **readTXTFile** (file, charsetname)

Read all content from a text file. If a file name is not specified, then a file selection dialog pops up for selecting a file. The encoding can be also specified. (Web Enabled only for a JSFile argument)

**Parameters**

{String} file - the file path.
{String} charsetname - Charset name.

**Returns**

String

**Sample**

```
// Read content from a known text file.
var txt = plugins.file.readTXTFile('story.txt');
application.output(txt);
// Read content from a text file selected from the file open dialog.
var txtUnknown = plugins.file.readTXTFile();
application.output(txtUnknown);
```

### showDirectorySelectDialog

JSFile **showDirectorySelectDialog** ()

Shows a directory selector dialog.

**Returns**

JSFile

**Sample**

```
var dir = plugins.file.showDirectorySelectDialog();
application.output("you've selected folder: " + dir.getAbsolutePath());
```

### showDirectorySelectDialog

JSFile **showDirectorySelectDialog** (directory)

Shows a directory selector dialog.

**Parameters**

{JSFile} directory - Default directory as JSFile.

**Returns**

JSFile

**Sample**

```
var dir = plugins.file.showDirectorySelectDialog();
application.output("you've selected folder: " + dir.getAbsolutePath());
```

### showDirectorySelectDialog

JSFile **showDirectorySelectDialog** (directory, title)

Shows a directory selector dialog.

**Parameters**

{JSFile} directory - Default directory as JSFile.
{String} title - Dialog title.

**Returns**

JSFile

**Sample**

```
var dir = plugins.file.showDirectorySelectDialog();
application.output("you've selected folder: " + dir.getAbsolutePath());
```

### showDirectorySelectDialog

JSFile **showDirectorySelectDialog** (directory)

Shows a directory selector dialog.

**Parameters**

{String} directory - Default directory as file path.

**Returns**

JSFile

**Sample**

```
var dir = plugins.file.showDirectorySelectDialog();
application.output("you've selected folder: " + dir.getAbsolutePath());
```

### showDirectorySelectDialog

JSFile **showDirectorySelectDialog** (directory, title)

Shows a directory selector dialog.

**Parameters**

{String} directory - Default directory as file path.
{String} title - Dialog title.

**Returns**

JSFile

**Sample**

```
var dir = plugins.file.showDirectorySelectDialog();
application.output("you've selected folder: " + dir.getAbsolutePath());
```

### showFileOpenDialog

Object **showFileOpenDialog** ()

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileOpenDialog

Object **showFileOpenDialog** (selectionMode)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Number} selectionMode - 0=both,1=Files,2=Dirs

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileOpenDialog

Object **showFileOpenDialog** (selectionMode, startDirectory)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Number} selectionMode - 0=both,1=Files,2=Dirs
{JSFile} startDirectory - JSFile instance of default folder; null=default/previous

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileOpenDialog

Object **showFileOpenDialog** (selectionMode, startDirectory, multiselect)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Number} selectionMode - 0=both,1=Files,2=Dirs
{JSFile} startDirectory - JSFile instance of default folder, null=default/previous
{Boolean} multiselect - true/false

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileOpenDialog

Object **showFileOpenDialog** (selectionMode, startDirectory, multiselect, filter)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Number} selectionMode - 0=both,1=Files,2=Dirs
{JSFile} startDirectory - JSFile instance of default folder,null=default/previous
{Boolean} multiselect - true/false
{Object} filter - A filter or array of filters on the folder files.

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileOpenDialog

Object **showFileOpenDialog** (selectionMode, startDirectory, multiselect, filter, callbackfunction)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Number} selectionMode - 0=both,1=Files,2=Dirs
{JSFile} startDirectory - JSFile instance of default folder,null=default/previous
{Boolean} multiselect - true/false
{Object} filter - A filter or array of filters on the folder files.
{Function} callbackfunction - A function that takes the (JSFile) array of the selected files as first argument

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

## showFileOpenDialog

Object **showFileOpenDialog** (selectionMode, startDirectory, multiselect, filter, callbackfunction, title)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Number} selectionMode - 0=both,1=Files,2=Dirs
{JSFile} startDirectory - JSFile instance of default folder, null=default/previous
{Boolean} multiselect - true/false
{Object} filter - A filter or array of filters on the folder files.
{Function} callbackfunction - A function that takes the (JSFile) array of the selected files as first argument
{String} title - The tile of the dialog

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileOpenDialog

Object **showFileOpenDialog** (selectionMode, startDirectory, multiselect, callbackfunction)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Number} selectionMode - 0=both,1=Files,2=Dirs
{JSFile} startDirectory - JSFile instance of default folder,null=default/previous
{Boolean} multiselect - true/false
{Function} callbackfunction - A function that takes the (JSFile) array of the selected files as first argument

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileOpenDialog

Object **showFileOpenDialog** (selectionMode, startDirectory, callbackfunction)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Number} selectionMode - 0=both,1=Files,2=Dirs
{JSFile} startDirectory - JSFile instance of default folder,null=default/previous
{Function} callbackfunction - A function that takes the (JSFile) array of the selected files as first argument

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileOpenDialog

Object **showFileOpenDialog** (selectionMode, startDirectory)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Number} selectionMode - 0=both,1=Files,2=Dirs
{String} startDirectory - Path to default folder; null=default/previous

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileOpenDialog

Object **showFileOpenDialog** (selectionMode, startDirectory, multiselect)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Number} selectionMode - 0=both,1=Files,2=Dirs
{String} startDirectory - Path to default folder, null=default/previous
{Boolean} multiselect - true/false

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileOpenDialog

Object **showFileOpenDialog** (selectionMode, startDirectory, multiselect, filter)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Number} selectionMode - 0=both,1=Files,2=Dirs
{String} startDirectory - Path to default folder,null=default/previous
{Boolean} multiselect - true/false
{Object} filter - A filter or array of filters on the folder files.

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileOpenDialog

Object **showFileOpenDialog** (selectionMode, startDirectory, multiselect, filter, callbackfunction)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Number} selectionMode - 0=both,1=Files,2=Dirs
{String} startDirectory - Path to default folder,null=default/previous
{Boolean} multiselect - true/false
{Object} filter - A filter or array of filters on the folder files.
{Function} callbackfunction - A function that takes the (JSFile) array of the selected files as first argument

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileOpenDialog

Object **showFileOpenDialog** (selectionMode, startDirectory, multiselect, filter, callbackfunction, title)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Number} selectionMode - 0=both,1=Files,2=Dirs
{String} startDirectory - Path to default folder, null=default/previous
{Boolean} multiselect - true/false
{Object} filter - A filter or array of filters on the folder files.
{Function} callbackfunction - A function that takes the (JSFile) array of the selected files as first argument
{String} title - The tile of the dialog

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileOpenDialog

Object **showFileOpenDialog** (selectionMode, startDirectory, multiselect, callbackfunction)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Number} selectionMode - 0=both,1=Files,2=Dirs
{String} startDirectory - Path to default folder,null=default/previous
{Boolean} multiselect - true/false
{Function} callbackfunction - A function that takes the (JSFile) array of the selected files as first argument

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileOpenDialog

Object **showFileOpenDialog** (selectionMode, startDirectory, callbackfunction)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Number} selectionMode - 0=both,1=Files,2=Dirs
{String} startDirectory - Path to default folder,null=default/previous
{Function} callbackfunction - A function that takes the (JSFile) array of the selected files as first argument

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileOpenDialog

Object **showFileOpenDialog** (selectionMode, callbackfunction)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Number} selectionMode - 0=both,1=Files,2=Dirs
{Function} callbackfunction - A function that takes the (JSFile) array of the selected files as first argument

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileOpenDialog

Object **showFileOpenDialog** (callbackfunction)

Shows a file open dialog. Filters can be applied on what type of files can be selected. (Web Enabled, you must set the callback method for this to work)

**Parameters**

{Function} callbackfunction - A function that takes the (JSFile) array of the selected files as first argument

**Returns**

Object

**Sample**

```
// This selects only files ('1'), previous dir must be used ('null'), no multiselect ('false') and
// the filter "JPG and GIF" should be used: ('new Array("JPG and GIF","jpg","gif")').
/** @type {JSFile} */
var f = plugins.file.showFileOpenDialog(1, null, false, new Array("JPG and GIF", "jpg", "gif"));
application.output('File: ' + f.getName());
application.output('is dir: ' + f.isDirectory());
application.output('is file: ' + f.isFile());
application.output('path: ' + f.getAbsolutePath());

// This allows mutliple selection of files, using previous dir and the same filter as above. This also casts
the result to the JSFile type using JSDoc.
// if filters are specified, "all file" filter will not show up unless "*" filter is present
/** @type {JSFile[]} */
var files = plugins.file.showFileOpenDialog(1, null, true, new Array("JPG and GIF", "jpg", "gif", "*"));
for (var i = 0; i < files.length; i++)
{
        application.output('File: ' + files[i].getName());
        application.output('content type: ' + files[i].getContentType());
        application.output('last modified: ' + files[i].lastModified());
        application.output('size: ' + files[i].size());
}
//for the web you have to give a callback function that has a JSFile array as its first argument (also works
in smart), only multi select and the title are used in the webclient, others are ignored
plugins.file.showFileOpenDialog(null,null,false,new Array("JPG and GIF", "jpg", "gif"),
mycallbackfunction,'Select some nice files')
```

### showFileSaveDialog

JSFile **showFileSaveDialog** ()

Shows a file save dialog.

**Returns**

JSFile

**Sample**

```
var file = plugins.file.showFileSaveDialog();
application.output("you've selected file: " + file.getAbsolutePath());
```

### showFileSaveDialog

JSFile **showFileSaveDialog** (fileNameDir)

Shows a file save dialog.

**Parameters**

{JSFile} fileNameDir - JSFile to save.

**Returns**

JSFile

**Sample**

```
var file = plugins.file.showFileSaveDialog();
application.output("you've selected file: " + file.getAbsolutePath());
```

### showFileSaveDialog

JSFile **showFileSaveDialog** (fileNameDir, title)

Shows a file save dialog.

**Parameters**

{JSFile} fileNameDir - JSFile to save
{String} title - Dialog title.

**Returns**

JSFile

**Sample**

```
var file = plugins.file.showFileSaveDialog();
application.output("you've selected file: " + file.getAbsolutePath());
```

### showFileSaveDialog

JSFile **showFileSaveDialog** (fileNameDir)

Shows a file save dialog.

**Parameters**

{String} fileNameDir - File (give as file path) to save.

**Returns**

JSFile

**Sample**

```
var file = plugins.file.showFileSaveDialog();
application.output("you've selected file: " + file.getAbsolutePath());
```

### showFileSaveDialog

JSFile **showFileSaveDialog** (fileNameDir, title)

Shows a file save dialog.

**Parameters**

{String} fileNameDir - File to save (specified as file path)
{String} title - Dialog title.

**Returns**

JSFile

**Sample**

```
var file = plugins.file.showFileSaveDialog();
application.output("you've selected file: " + file.getAbsolutePath());
```

### streamFilesFromServer

JSProgressMonitor **streamFilesFromServer** (files, serverFiles)

Stream 1 or more files from the server to the client.

**Since**

Servoy 5.2

**Parameters**

{Object} files - file(s) to be streamed into (can be a String path a JSFile) or an Array of these
{Object} serverFiles - the files on the server that will be transfered to the client, can be a String or a String[]

**Returns**

JSProgressMonitor - a JSProgressMonitor object to allow client to subscribe to progress notifications

**Sample**

```
// transfer all the files of a chosen server folder to a directory on the client
var dir = plugins.file.showDirectorySelectDialog();
if (dir) {
	var list = plugins.file.getRemoteFolderContents('/images/user1/', null, 1);
	if (list) {
		var monitor = plugins.file.streamFilesFromServer(dir, list, callbackFunction);
	}
}

// transfer one file on the client
var monitor = plugins.file.streamFilesFromServer('/path/to/file', 'path/to/serverFile', callbackFunction);

// transfer an array of serverFiles to an array of files on the client
var files = new Array();
files[0] = '/path/to/file1';
files[1] = '/path/to/file2';
var serverFiles = new Array();
serverFiles[0] = '/path/to/serverFile1';
serverFiles[1] = '/path/to/serverFile2';
var monitor = plugins.file.streamFilesFromServer(files, serverFiles, callbackFunction);
```

## streamFilesFromServer

JSProgressMonitor **streamFilesFromServer** (files, serverFiles, callback)

Stream 1 or more files from the server to the client, the callback method is invoked after every file, with as argument the filename that was transfered. An extra second exception parameter can be given if an exception did occur.

**Since**

Servoy 5.2

**Parameters**

{Object} files - file(s) to be streamed into (can be a String path or a JSFile) or an Array of these
{Object} serverFiles - the files on the server that will be transfered to the client, can be a JSFile or JSFile[], a String or String[]
{Function} callback - the Function to be called back at the end of the process (for every file); the callback function is invoked with argument the filename that was transfered; an extra second exception parameter can be given if an exception occured

**Returns**

JSProgressMonitor - a JSProgressMonitor object to allow client to subscribe to progress notifications

**Sample**

```
// transfer all the files of a chosen server folder to a directory on the client
var dir = plugins.file.showDirectorySelectDialog();
if (dir) {
	var list = plugins.file.getRemoteFolderContents('/images/user1/', null, 1);
	if (list) {
		var monitor = plugins.file.streamFilesFromServer(dir, list, callbackFunction);
	}
}

// transfer one file on the client
var monitor = plugins.file.streamFilesFromServer('/path/to/file', 'path/to/serverFile', callbackFunction);

// transfer an array of serverFiles to an array of files on the client
var files = new Array();
files[0] = '/path/to/file1';
files[1] = '/path/to/file2';
var serverFiles = new Array();
serverFiles[0] = '/path/to/serverFile1';
serverFiles[1] = '/path/to/serverFile2';
var monitor = plugins.file.streamFilesFromServer(files, serverFiles, callbackFunction);
```

## streamFilesToServer

JSProgressMonitor **streamFilesToServer** (files)

Overloaded method, only defines file(s) to be streamed

**Since**

Servoy 5.2

**Parameters**

{Object} files - file(s) to be streamed (can be a String path or a JSFile) or an Array of these

**Returns**

JSProgressMonitor - a JSProgressMonitor object to allow client to subscribe to progress notifications

**Sample**

```
// send one file:
var file = plugins.file.showFileOpenDialog( 1, null, false, null, null, 'Choose a file to transfer' );
if (file) {
        plugins.file.streamFilesToServer( file, callbackFunction );
}
//plugins.file.streamFilesToServer( 'servoy.txt', callbackFunction );

// send an array of files:
var folder = plugins.file.showDirectorySelectDialog();
if (folder) {
        var files = plugins.file.getFolderContents(folder);
        if (files) {
                var monitor = plugins.file.streamFilesToServer( files, callbackFunction );
        }
}
// var files = new Array()
// files[0] = 'file1.txt';
// files[1] = 'file2.txt';
// var monitor = plugins.file.streamFilesToServer( files, callbackFunction );
```

## streamFilesToServer

JSProgressMonitor **streamFilesToServer** (files, serverFiles)

Overloaded method, defines file(s) to be streamed and a callback function

**Since**

Servoy 5.2

**Parameters**

{Object} files - file(s) to be streamed (can be a String path or a JSFile) or an Array of these
{Object} serverFiles - can be a JSFile or JSFile[], a String or String[], representing the file name(s) to use on the server

**Returns**

JSProgressMonitor - a JSProgressMonitor object to allow client to subscribe to progress notifications

**Sample**

```
// send one file:
var file = plugins.file.showFileOpenDialog( 1, null, false, null, null, 'Choose a file to transfer' );
if (file) {
        plugins.file.streamFilesToServer( file, callbackFunction );
}
//plugins.file.streamFilesToServer( 'servoy.txt', callbackFunction );

// send an array of files:
var folder = plugins.file.showDirectorySelectDialog();
if (folder) {
        var files = plugins.file.getFolderContents(folder);
        if (files) {
                var monitor = plugins.file.streamFilesToServer( files, callbackFunction );
        }
}
// var files = new Array()
// files[0] = 'file1.txt';
// files[1] = 'file2.txt';
// var monitor = plugins.file.streamFilesToServer( files, callbackFunction );
```

## streamFilesToServer

JSProgressMonitor **streamFilesToServer** (files, serverFiles, callback)

Overloaded method, defines file(s) to be streamed, a callback function and file name(s) to use on the server

**Since**

Servoy 5.2

**Parameters**

{Object} files - file(s) to be streamed (can be a String path or a JSFile) or an Array of these)
{Object} serverFiles - can be a JSFile or JSFile[], a String or String[], representing the file name(s) to use on the server
{Function} callback - the Function to be called back at the end of the process (for every file); the callback function is invoked with argument the filename that was transfered; an extra second exception parameter can be given if an exception occured

**Returns**

JSProgressMonitor - a JSProgressMonitor object to allow client to subscribe to progress notifications

**Sample**

```
// send one file:
var file = plugins.file.showFileOpenDialog( 1, null, false, null, null, 'Choose a file to transfer' );
if (file) {
        plugins.file.streamFilesToServer( file, callbackFunction );
}
//plugins.file.streamFilesToServer( 'servoy.txt', callbackFunction );

// send an array of files:
var folder = plugins.file.showDirectorySelectDialog();
if (folder) {
        var files = plugins.file.getFolderContents(folder);
        if (files) {
                var monitor = plugins.file.streamFilesToServer( files, callbackFunction );
        }
}
// var files = new Array()
// files[0] = 'file1.txt';
// files[1] = 'file2.txt';
// var monitor = plugins.file.streamFilesToServer( files, callbackFunction );
```

## streamFilesToServer

JSProgressMonitor **streamFilesToServer** (files, callback)

Overloaded method, defines file(s) to be streamed and a callback function

**Since**

Servoy 5.2

**Parameters**

{Object} files - file(s) to be streamed (can be a String path or a JSFile) or an Array of these
{Function} callback - the Function to be called back at the end of the process (for every file); the callback function is invoked with argument the filename that was transfered; an extra second exception parameter can be given if an exception occured

**Returns**

JSProgressMonitor - a JSProgressMonitor object to allow client to subscribe to progress notifications

**Sample**

```
// send one file:
var file = plugins.file.showFileOpenDialog( 1, null, false, null, null, 'Choose a file to transfer' );
if (file) {
        plugins.file.streamFilesToServer( file, callbackFunction );
}
//plugins.file.streamFilesToServer( 'servoy.txt', callbackFunction );

// send an array of files:
var folder = plugins.file.showDirectorySelectDialog();
if (folder) {
        var files = plugins.file.getFolderContents(folder);
        if (files) {
                var monitor = plugins.file.streamFilesToServer( files, callbackFunction );
        }
}
// var files = new Array()
// files[0] = 'file1.txt';
// files[1] = 'file2.txt';
// var monitor = plugins.file.streamFilesToServer( files, callbackFunction );
```

## writeFile

Boolean **writeFile** (file, data)

Writes data into a binary file. (Web Enabled: file parameter can be a string 'mypdffile.pdf' to hint the browser what it is, if it is a JSFile instance it will be saved on the server)

**Parameters**

{JSFile} file - a local JSFile
{byte[]} data - the data to be written

**Returns**

Boolean

**Sample**

```
/**@type {Array<byte>}*/
var bytes = new Array();
for (var i=0; i<1024; i++)
	bytes[i] = i % 100;
var f = plugins.file.convertToJSFile('bin.dat');
if (!plugins.file.writeFile(f, bytes))
	application.output('Failed to write the file.');
// mimeType variable can be left null, and is used for webclient only. Specify one of any valid mime types
as referenced here: http://www.w3schools.com/media/media_mimeref.asp'
var mimeType = 'application/vnd.ms-excel'
if (!plugins.file.writeFile(f, bytes, mimeType))
	application.output('Failed to write the file.');
```

### writeFile

Boolean **writeFile** (file, data, mimeType)

Writes data into a binary file. (Web Enabled: file parameter can be a string 'mypdffile.pdf' to hint the browser what it is, if it is a JSFile instance it will be saved on the server)

**Parameters**

{JSFile} file - a local JSFile
{byte[]} data - the data to be written
{String} mimeType - the mime type

**Returns**

Boolean

**Sample**

```
/**@type {Array<byte>}*/
var bytes = new Array();
for (var i=0; i<1024; i++)
	bytes[i] = i % 100;
var f = plugins.file.convertToJSFile('bin.dat');
if (!plugins.file.writeFile(f, bytes))
	application.output('Failed to write the file.');
// mimeType variable can be left null, and is used for webclient only. Specify one of any valid mime types
as referenced here: http://www.w3schools.com/media/media_mimeref.asp'
var mimeType = 'application/vnd.ms-excel'
if (!plugins.file.writeFile(f, bytes, mimeType))
	application.output('Failed to write the file.');
```

### writeFile

Boolean **writeFile** (file, data)

**Parameters**

{String} file - the file path as a String
{byte[]} data - the data to be written

**Returns**

Boolean

**Sample**

```
/**@type {Array<byte>}*/
var bytes = new Array();
for (var i=0; i<1024; i++)
        bytes[i] = i % 100;
var f = plugins.file.convertToJSFile('bin.dat');
if (!plugins.file.writeFile(f, bytes))
        application.output('Failed to write the file.');
// mimeType variable can be left null, and is used for webclient only. Specify one of any valid mime types
as referenced here: http://www.w3schools.com/media/media_mimeref.asp'
var mimeType = 'application/vnd.ms-excel'
if (!plugins.file.writeFile(f, bytes, mimeType))
        application.output('Failed to write the file.');
```

### writeFile

Boolean **writeFile** (file, data, mimeType)

Writes data into a binary file. (Web Enabled: file parameter can be a string 'mypdffile.pdf' to hint the browser what it is, if it is a JSFile instance it will be saved on the server)

**Parameters**

{String} file - the file path as a String
{byte[]} data - the data to be written
{String} mimeType - the mime type

**Returns**

Boolean

**Sample**

```
/**@type {Array<byte>}*/
var bytes = new Array();
for (var i=0; i<1024; i++)
        bytes[i] = i % 100;
var f = plugins.file.convertToJSFile('bin.dat');
if (!plugins.file.writeFile(f, bytes))
        application.output('Failed to write the file.');
// mimeType variable can be left null, and is used for webclient only. Specify one of any valid mime types
as referenced here: http://www.w3schools.com/media/media_mimeref.asp'
var mimeType = 'application/vnd.ms-excel'
if (!plugins.file.writeFile(f, bytes, mimeType))
        application.output('Failed to write the file.');
```

### writeTXTFile

Boolean **writeTXTFile** (file, text_data)

Writes data into a text file. (Web Enabled: file parameter can be a string 'mytextfile.txt' to hint the browser what it is, if it is a JSFile instance it will be saved on the server)

**Parameters**

{JSFile} file - JSFile
{String} text_data - Text to be written.

**Returns**

Boolean - Success boolean.

**Sample**

```
var fileNameSuggestion = 'myspecialexport.tab'
var textData = 'load of data...'
var success = plugins.file.writeTXTFile(fileNameSuggestion, textData);
if (!success) application.output('Could not write file.');
// For file-encoding parameter options (default OS encoding is used), http://download.oracle.com/javase/1.4.2
/docs/guide/intl/encoding.doc.html
// mimeType variable can be left null, and is used for webclient only. Specify one of any valid mime types
as referenced here: http://www.w3schools.com/media/media_mimeref.asp'
```

### writeTXTFile

Boolean **writeTXTFile** (file, text_data, charsetname)

Writes data into a text file. (Web Enabled: file parameter can be a string 'mytextfile.txt' to hint the browser what it is, if it is a JSFile instance it will be saved on the server)

**Parameters**

{JSFile} file - JSFile
{String} text_data - Text to be written.
{String} charsetname - Charset name.

**Returns**

Boolean - Success boolean.

**Sample**

```
var fileNameSuggestion = 'myspecialexport.tab'
var textData = 'load of data...'
var success = plugins.file.writeTXTFile(fileNameSuggestion, textData);
if (!success) application.output('Could not write file.');
// For file-encoding parameter options (default OS encoding is used), http://download.oracle.com/javase/1.4.2
/docs/guide/intl/encoding.doc.html
// mimeType variable can be left null, and is used for webclient only. Specify one of any valid mime types
as referenced here: http://www.w3schools.com/media/media_mimeref.asp'
```

### writeTXTFile

Boolean **writeTXTFile** (file, text_data, charsetname, mimeType)

Writes data into a text file. (Web Enabled: file parameter can be a string 'mytextfile.txt' to hint the browser what it is, if it is a JSFile instance it will be saved on the server)

**Parameters**

{JSFile} file - JSFile
{String} text_data - Text to be written.
{String} charsetname - Charset name.
{String} mimeType - Content type (used only on web).

**Returns**

Boolean - Success boolean.

**Sample**

```
var fileNameSuggestion = 'myspecialexport.tab'
var textData = 'load of data...'
var success = plugins.file.writeTXTFile(fileNameSuggestion, textData);
if (!success) application.output('Could not write file.');
// For file-encoding parameter options (default OS encoding is used), http://download.oracle.com/javase/1.4.2
/docs/guide/intl/encoding.doc.html
// mimeType variable can be left null, and is used for webclient only. Specify one of any valid mime types
as referenced here: http://www.w3schools.com/media/media_mimeref.asp'
```

### writeTXTFile

Boolean **writeTXTFile** (file, text_data)

Writes data into a text file. (Web Enabled: file parameter can be a string 'mytextfile.txt' to hint the browser what it is, if it is a JSFile instance it will be saved on the server)

**Parameters**

{String} file - The file path.
{String} text_data - Text to be written.

**Returns**

Boolean

**Sample**

```
var fileNameSuggestion = 'myspecialexport.tab'
var textData = 'load of data...'
var success = plugins.file.writeTXTFile(fileNameSuggestion, textData);
if (!success) application.output('Could not write file.');
// For file-encoding parameter options (default OS encoding is used), http://download.oracle.com/javase/1.4.2
/docs/guide/intl/encoding.doc.html
// mimeType variable can be left null, and is used for webclient only. Specify one of any valid mime types
as referenced here: http://www.w3schools.com/media/media_mimeref.asp'
```

### writeTXTFile

Boolean **writeTXTFile** (file, text_data, charsetname)

Writes data into a text file. (Web Enabled: file parameter can be a string 'mytextfile.txt' to hint the browser what it is, if it is a JSFile instance it will be saved on the server)

**Parameters**

{String} file - The file path.
{String} text_data - Text to be written.
{String} charsetname - Charset name.

**Returns**

Boolean

**Sample**

```
var fileNameSuggestion = 'myspecialexport.tab'
var textData = 'load of data...'
var success = plugins.file.writeTXTFile(fileNameSuggestion, textData);
if (!success) application.output('Could not write file.');
// For file-encoding parameter options (default OS encoding is used), http://download.oracle.com/javase/1.4.2
/docs/guide/intl/encoding.doc.html
// mimeType variable can be left null, and is used for webclient only. Specify one of any valid mime types
as referenced here: http://www.w3schools.com/media/media_mimeref.asp'
```

### writeTXTFile

Boolean **writeTXTFile** (file, text_data, charsetname, mimeType)

Writes data into a text file. (Web Enabled: file parameter can be a string 'mytextfile.txt' to hint the browser what it is, if it is a JSFile instance it will be saved on the server)

**Parameters**

{String} file - The file path.
{String} text_data - Text to be written.
{String} charsetname - Charset name.
{String} mimeType - Content type (used only on web).

**Returns**

Boolean

**Sample**

```
var fileNameSuggestion = 'myspecialexport.tab'
var textData = 'load of data...'
var success = plugins.file.writeTXTFile(fileNameSuggestion, textData);
if (!success) application.output('Could not write file.');
// For file-encoding parameter options (default OS encoding is used), http://download.oracle.com/javase/1.4.2
/docs/guide/intl/encoding.doc.html
// mimeType variable can be left null, and is used for webclient only. Specify one of any valid mime types
as referenced here: http://www.w3schools.com/media/media_mimeref.asp'
```

### writeXMLFile

Boolean **writeXMLFile** (file, xml_data)

Writes data into an XML file. The file is saved with the encoding specified by the XML itself. (Web Enabled: file parameter can be a string 'myxmlfile.xml' to hint the browser what it is, if it is a JSFile instance it will be saved on the server)

**Parameters**

{JSFile} file - a local JSFile
{String} xml_data - the xml data to write

**Returns**

Boolean

**Sample**

```
var fileName = 'form.xml'
var xml = controller.printXML()
var success = plugins.file.writeXMLFile(fileName, xml);
if (!success) application.output('Could not write file.');
```

### writeXMLFile

Boolean **writeXMLFile** (file, xml_data, encoding)

Writes data into an XML file. The file is saved with the encoding specified by the XML itself. (Web Enabled: file parameter can be a string 'myxmlfile.xml' to hint the browser what it is, if it is a JSFile instance it will be saved on the server)

**Parameters**

{JSFile} file - a local JSFile
{String} xml_data - the xml data to write
{String} encoding - the specified encoding

**Returns**

Boolean

**Sample**

```
var fileName = 'form.xml'
var xml = controller.printXML()
var success = plugins.file.writeXMLFile(fileName, xml);
if (!success) application.output('Could not write file.');
```

### writeXMLFile

Boolean **writeXMLFile** (file, xml_data)

Writes data into an XML file. The file is saved with the encoding specified by the XML itself. (Web Enabled: file parameter can be a string 'myxmlfile.xml' to hint the browser what it is, if it is a JSFile instance it will be saved on the server)

**Parameters**

{String} file - the file path as a String
{String} xml_data - the xml data to write

**Returns**

Boolean

**Sample**

```
var fileName = 'form.xml'
var xml = controller.printXML()
var success = plugins.file.writeXMLFile(fileName, xml);
if (!success) application.output('Could not write file.');
```

### writeXMLFile

Boolean **writeXMLFile** (file, xml_data, encoding)

Writes data into an XML file. The file is saved with the encoding specified by the XML itself. (Web Enabled: file parameter can be a string 'myxmlfile.xml' to hint the browser what it is, if it is a JSFile instance it will be saved on the server)

**Parameters**

{String} file - the file path as a String
{String} xml_data - the xml data to write
{String} encoding - the specified encoding

**Returns**

Boolean

**Sample**

```
var fileName = 'form.xml'
var xml = controller.printXML()
var success = plugins.file.writeXMLFile(fileName, xml);
if (!success) application.output('Could not write file.');
```