ADOBE® BRIDGE 2020

JAVASCRIPT REFERENCE

Adobe

Adobe® Creative Cloud: Adobe Bridge JavaScript Reference for Windows® and Macintosh®.

Contents

Welcome

Welcome to the *Adobe Bridge 2020 JavaScript Reference*. This book describes the JavaScript scripting API that allows you to manipulate and extend Adobe® Bridge.

About this book

This book provides complete reference information for the JavaScript objects, properties, and functions defined by Adobe Bridge. For conceptual information and examples, see the companion document, the *Adobe Bridge JavaScript Guide*.

Who should read this book

This book is for developers who want to call Adobe Bridge functionality from scripts, extend the capabilities of Adobe Bridge using JavaScript, and use scripts to communicate between Adobe Bridge and other applications. It assumes a general familiarity with the following:

- JavaScript
- Adobe Bridge
- Any other Adobe Bridge applications you are using, such as Adobe Illustrator, Adobe Photoshop®, or Adobe InDesign®. The scripting API details for each application are included with the scripting documentation for that product.

What is in this book

This book provides detailed reference information about the JavaScript objects that Adobe Bridge provides.

This book contains the following chapters:

- Chapter 1, "Adobe Bridge DOM Object Reference," provides a complete API reference for the objects, properties, and functions defined in the Adobe Bridge document object model, which any script can use to program Adobe Bridge functionality and interactions with users.
- Chapter 2, "Node-Handling Extension Object Reference," provides a complete API reference for the objects, properties, and functions that allow a product or plug-in developer to extend Adobe Bridge functionality, in order to provide compatibility with another product or plug-in.
- Chapter 3, "External Communication Tools," provides a complete API reference for the Web Access library, which supplies tools for communicating with other computers or the Internet using standard protocols FTP and HTTP.

The Adobe Bridge scripter also has access to a various utilities and tools that are part of ExtendScript, the Adobe extended implementation of JavaScript. These are described separately, in the *JavaScript Tools Guide*.

Document conventions

Typographical conventions

`Monospaced font`	Literal values and code, such as JavaScript code, HTML code, filenames, and pathnames.
Italics	Variables or placeholders in code. For example, in `name="myName"`, the text *`myName`* represents a value you are expected to supply, such as `name="Fred"`. Also indicates the first occurrence of a new term.
Blue underlined text	A hyperlink you can click to go to a related section in this book or to a URL in your web browser.
Sans-serif bold font	The names of Adobe Bridge UI elements (menus, menu items, and buttons). The **>** symbol is used as shorthand notation for navigating to menu items. For example, **Edit > Cut** refers to the **Cut** item in the **Edit** menu.

NOTE: Notes highlight important points that deserve extra attention.

JavaScript conventions

This reference does not list properties and methods provided by the JavaScript language itself. For example, it is common for JavaScript objects to provide a `toString()` method, and many of the objects the SDK supplies implement this method. However, this book does not describe such methods unless they differ from the standard JavaScript implementation.

Similarly, because most objects provided by the SDK have a `name` property, the reference does not list `name` properties explicitly.

Where to go for more information

The Adobe Bridge Software Developer's Kit (SDK) contains the JavaScript documentation and code samples. The SDK is available for download from Adobe Developer Center, http://www.adobe.com/devnet/. You can install the SDK in a folder with a name and location of your choice, referred to here as *`sdkInstall`*. The SDK contains:

`sdkInstall``/docs/`	Adobe Bridge JavaScript documents in PDF format, including: This manual The *Adobe Bridge JavaScript Guide*, which provides an overview of conceptual information and details of scripting techniques.
`sdkInstall``/sdksamples/`	A set of code samples that illustrate Adobe Bridge scripting concepts and techniques. The sections in this manual that discuss particular concepts list the code samples that demonstrate the related techniques.

This book does not describe the JavaScript language. For documentation of the JavaScript language or descriptions of how to use it, see publicly available web resources or any of numerous works on this subject, including the following:

- The public JavaScript standards organization web site: www.ecma-international.org
- *JavaScript: The Definitive Guide, 4th Edition*; Flanagan, D.; O'Reilly 2001; ISBN 0-596-00048-0
- *JavaScript Programmer's Reference*; Wootton, C.; Wrox 2001; ISBN 1-861004-59-1
- *JavaScript Bible. 5th Edition*; Goodman, D. and Morrison, M.; John Wiley and Sons1998; ISBN 0-7645-57432

1 Adobe Bridge DOM Object Reference

This document provides a complete reference for the objects of the Adobe Bridge document object model (DOM). An overview of the objects and usage details are provided in the *Adobe Bridge JavaScript Guide*.

This chapter presents the primary objects. Additional objects are available to advanced developers who wish to extend the node-handling capability of Adobe Bridge by defined their own node types; these objects are described in Chapter 2, "Node-Handling Extension Object Reference."

In addition to these Adobe Bridge-specific objects, an API common to most JavaScript-enabled Adobe applications is supplied by ExtendScript, the Adobe extended implementation of JavaScript. These tools are described in detail in the *JavaScript Tools Guide*. The tools include:

- *ScriptUI* objects that provide user-interface capability.
- The ExtendScript `File` and `Folder` objects that provide portable access to the file system.
- An interapplication messaging framework that provides the ability to communicate among scriptable applications using JavaScript.
- ExtendScript utilities, including tools for debugging, for localization, and for specifying and working with measurement values.

Object summary

The objects are presented alphabetically. For each object, complete syntax details are provided for the constructor, properties, and functions.

AdobePortfolio Object	Creates a new portfolio project based on the project name and the list of files provided.
AdobeStock Object	Provides capability to upload selected images in Bridge to Adobe Stock directly.
App Object	Global information about the Adobe Bridge application.
BitmapData Object	Represents an image as a matrix of pixels.
Color Object	An RGB color description.
Document Object	An Adobe Bridge browser window.
Event Object	A user-interaction event.
Favorites Object	Two arrays of the thumbnails shown in the Standard and User sections of the Favorites palette.
IconListPanelette Object	A member sub-panel in an Inspector panel that displays two or three columns.
InspectorPanel Object	An object-inspection panel in the tabbed palettes.
MenuElement Object	Access to the Adobe Bridge menus and commands.
Metadata Object	Access to file metadata through a thumbnail.

NavBar Object	A configurable navigation bar that can display user-interface controls.
Panelette Base Class	A base class for sub-panels in an Inspector panel.
PDFOutput Object	Provides capability to generate a PDF using the Bridge's native PDF Output module.
Preferences Object	Access to application preferences.
PreferencesDialog Object	Access to the Preferences dialog.
QuickSearch Object	Provides capability to use bridge quick search feature
TabbedPalette Object	A tabbed panel in the Adobe Bridge browser window.
TextPanelette Object	A member sub-panel in an Inspector panel that displays text.
Thumbnail Object	A navigable node representing a file, folder, or web page.
ThumbnailPanelette Object	A member sub-panel in an Inspector panel that displays thumbnails.

AdobePortfolio Object

AdobePortfolio Object provides capability to create a new portfolio project based on the project name and the list of files provided. Adobe Portfolio is a service to quickly and simply build a website to showcase creative work. After project creation, you can go to the Portfolio website to publish the project.

AdobePortfolio functions

createPortfolioProject() `adobePortfolioObj.createPortfolioProject(projectTitle, coverPath, requestType, adultContent)`	Upload provided images on Adobe Stock. Returns true if the snippet ran as expected, false if no files are selected in Bridge
`projectTitle`	Title of the Portfolio project.
`coverPath`	Path of the thumbnail used for the cover picture of the project.
`requestType`	Can be either 0 or 1. 0 denotes publish only to Adobe Portfolio;1 denotes publish to both Adobe Portfolio and Behance.
`adultContent`	Can be true or false.
addFilesForPortfolioProject() `adobePortfolioObj.addFilesForPortfolioProject(thumbnailPath)`	Adds the provided files to the Adobe Portfolio object.
`thumbnailPath`	Thumbnail Path of the images to be added to the project.

AdobeStock Object

AdobeStock Object provides capability to upload selected images in Bridge to Adobe Stock directly via Bridge scripting. Adobe Stock is a service that provides designers and businesses with access to millions of high-quality creative assets.

AdobeStock functions

`uploadToAdobeStock()` `adobeStockObj.uploadToAdobeStock(ImageList)` *`ImageList`*	Upload provided images on Adobe Stock. Return True if the snippet ran as expected, false if no files are selected in Bridge Array of thumbnail objects for the images to be uploaded.

App Object

The App object represents the Adobe Bridge application. A singleton instance is created on startup; access it using the app global variable.

There is only one App object; multiple browser windows are represented by instances of Document, and can be accessed with the app.document or app.documents properties.

App properties

defaultFilterCriteria	Array of Filter Description	A collection of the default FilterDescription Objects used to populate the Filter palette. Read only.

defaultSortCriteria	Array of SortCriterion	A collection of the default SortCriterion Objects used to sort the contents of container nodes. Read only.
		
		Default list is:
		
		`Filename`
		`Document type`
		`Date created`
		`Date file modified`
		`File size`
		`Dimensions`
		`Resolution`
		`Color profile`
		`Label`
		`Rating`
		`Keywords`

displayDialogs	String	The policy for the display of modal dialogs. Read/write. One of: **all** (default)-- Modal dialogs should always be displayed. **none**—Modal dialog should never be displayed. **error**—Only dialogs that report an error to the user should be displayed.
document	Document	The active (top-most) Document Object, representing the active browser window. Read/write. During an open or create event, this value still contains the previous Document object, while the new Document object is passed to the event handler.
documents	Array of Document	A collection of Document Objects representing the set of all open browser windows. Read/write.

eventHandlers	Array of EventHandler	A collection of event handlers installed by scripts. Add an event handler to this array to register it with Adobe Bridge. Registered handler functions are called when any user-interaction event is triggered. Read/write. Each event handler is specified by a JavaScript object with one property, the handler function name: `{ handler: fnName}` The handler function takes one argument, an Event Object, and returns a result object `{handled: boolean}`. When `true`, the event has been completely handled and Adobe Bridge does not look for more handlers or call the default handler. When `false` (or when the handler returns `undefined`), Adobe Bridge continues to call registered handlers, or if there are no more, calls the default handler.
extensions	Array of ExtensionHandler	A collection of ExtensionHandler Objects representing registered node-handling extensions. Read only. Use registerExtension() and unregisterExtension() to modify the list.
favorites	Favorites	The top-level object for the navigation hierarchy displayed in the Favorites palette. This Favorites Object contains two arrays of Thumbnail Objects, for the nodes shown in the two sections. Read only.
folderRoots	Array of String	The list of Bridge URI strings for the root nodes of the Folders pane. Read only. Extension developers can modify the list with addCustomRoot().
inspectorPanels	Array of InspectorPanel	The collection of script-defined InspectorPanel Objects that define Inspector panels for new Document Objects (browser windows). The list is in display order. Read only. Use registerInspectorPanel() and unregisterInspectorPanel() to modify the list.
language	String	The display name of the language for the current locale, as configured by the operating system. This is the name as it appears in the Preferences dialog. Read only.

lastSender	String	The application specifier for the application that has most recently sent an interapplication message to Adobe Bridge.
locale	String	The Adobe locale code for the current locale, as configured by the operating system. Read only. An Adobe locale code consists of a 2-letter ISO-639 language code and an optional 2-letter ISO 3166 country code separated by an underscore. Case is significant. For example, `en_US`, `en_UK`, `ja_JP`, `de_DE`, `fr_FR`.
name	String	The application specifier for this application, `"bridge"`. Read only.
pendingJobs	Number	The number of background tasks that Adobe Bridge has left to process. Background tasks are started for asynchronous operations, such as metadata extraction from thumbnails, or exporting the cache with `app.buildFolderCache()`. When all tasks have been started, this value is 0. Read only. **NOTE:** The 0 value does not mean that all jobs have completed. The application might still be building the cache after no more jobs are pending. To determine if the cache is complete, monitor the cache size to see when it stops growing or simply check if `isProcessingJob()` returns `false`.
preferences	Preferences	The Preferences Object, which provides access to the user preferences shown in the Preferences dialog (invoked from the **Edit > Preferences** command). Read only.
synchronousMode	Boolean	When true, Adobe Bridge attempts to ensure that all `Thumbnail` properties are valid before returning their values. This is particularly important when accessing or setting metadata. Scripts (other than node handlers) typically need to set synchronous mode to `true`. Default is `false`, for performance reasons. The value of `false` is automatically restored when a script has completed.
standardFavorites	Favorites	This Favorites object contains an array of `Thumbnail` objects for the nodes shown in the Standard section of the Favorites palette.
userFavorites	Favorites	This Favorites object contains an array of `Thumbnail` objects for the nodes shown in the User section of the Favorites palette.

version	String	The version number of the Adobe Bridge application. Read only.
watchDirInterval	Number	The duration in seconds between checks for folder consistency (checking whether files have been added or removed).
workspaces	Array of Object	The list of all available workspaces, both default and user- or script-defined. Each workspace is a JavaScript object with two properties, `id` and `name`, specifying the unique identifier and the localized display name; see `Document.workspace`. Read only.

App functions

acquirePhysicalFiles() `app.acquirePhysicalFiles (thumbnails)`	For each specified thumbnail, if it refers to a resource that does not have a local copy (such as the files referenced by AdobeDrive nodes), downloads the resource. **NOTE:** For efficiency, make one call for all files to be processed, rather than calling repeatedly. Returns `true` on success.
thumbnails	An array of Thumbnail Objects.
addCollectionMember() `app.addCollection (collection, member)`	Adds a member thumbnail, or set of member thumbnails, to a collection. Returns `undefined`.
collection	The Thumbnail Object for the collection node, as returned by `app.createCollection()`.
member	A Thumbnail Object or array of Thumbnail Objects to be added to the collection.
addCredits() `app.addCredits (title, content)`	Appends a text string to the Credits area of the Adobe Bridge About box. Returns `undefined`.
title	The unique identifying name of this addition.
content	The localizable text string.
addCustomRoot() `app.addCustomRoot (uri)`	Appends a custom URI to the list of root nodes in `app.folderRoots`, which appear in the Folders pane. Used by script-defined node handlers. Returns `true` on success.
uri	The Bridge URI string.

addLegalNotice() `app.addLegalNotice (title, content)`	Appends a text string to the Legal Notice area of the Adobe Bridge About box. Returns `undefined`.
title	The unique identifying name of this addition.
content	The localizable text string.
beep() `app.beep ()`	Calls on the operating system to emit a short audio tone. Returns `undefined`.
bringToFront() `app.bringToFront ()`	Gives Adobe Bridge the operating system application focus, and brings the current browser window to the front in the windowing system. Returns `undefined`.
buildFolderCache() `app.buildFolderCache (path[, recurse, quality, buildFullSize]`	Forces Adobe Bridge to create thumbnail images for the specified folder. These are stored in a cache file in the folder to which they apply. Returns `undefined`.
path	The folder. An ExtendScript `Folder` object, Thumbnail Object for a folder, or Bridge URI path string. If this specifies a file, the cache is built for the containing folder.
recurse	Optional in Adobe Bridge 1.0, not used in Adobe Bridge 2.0. Boolean. Cache building is always recursive; pass `true`.
quality	Optional. String. Whether to create low or high quality thumbnail images. One of the strings `"quick"` (the default) or `"highQuality"`.
buildFullSize	Optional. Boolean. Whether to export full-size images to cache folder. This matches what happens when users select "Generate 100% previews" in the Bridge window.
cancelTask() `app.cancelTask (taskId)`	Cancels a task that has been scheduled using scheduleTask(). Returns `undefined`.
taskId	The task ID number, as returned from `app.scheduleTask()`.
createCollection() `app.createCollection (name)`	Creates a new, named collection node. Returns the Thumbnail Object for the new node. Use this to access the collection programmatically.
name	The name of the new collection. If a collection with this name already exists, a unique name is generated using this string.
createSmartCollection() `app.createSmartCollection (name, scope, searchSpec)`	Creates a new, named smart collection node. Returns the Thumbnail Object for the new node. Use this to access the collection programmatically

name	The name of the new smart collection. If a collection with this name already exists, a unique name is generated using this string.
scope	A Thumbnail Object for the target container node.
searchSpec	A SearchSpecification Object used to generate the search result.

deleteCollection() app.deleteCollection (*collection*)	Deletes a collection node. Returns `true` on success.
collection	The Thumbnail Object for the collection node, as returned by app.createCollection().
deleteSmartCollection() app.deleteSmartCollection (*collection*)	Deletes a smart collection node. Returns `true` on success.
collection	The Thumbnail Object for the collection node, as returned by app.createSmartCollection().
enqueueOperation() app.enqueueOperation(*operator*)	Enqueues a long-running node-handling operation for execution at an appropriate time. Returns `undefined`.
operator	The Operator Class instance that encapsulates the operation, returned by an ExtensionModel Object method.
exportKeywordsToFile() app.exportKeywordsToFile (*keywordsFile*)	Exports the keywords in the Keywords palette to a file. This is the same as choosing Export from the flyout menu in the Keywords palette. Returns `undefined`.
keywordsFile	The file, specified as a path string or ExtendScript File object.
getCollectionMembers() app.getCollectionMembers (*collection*)	Retrieves the collection members for a collection node. Returns an array of Thumbnail Objects for the collection members.
collection	The Thumbnail Object for the collection node, as returned by app.createCollection().
getCollections() app.getCollections ()	Retrieves all collection nodes, as created with createCollection(). Returns an array of Thumbnail Objects for the collection nodes.
getSmartCollections() app.getSmartCollections ()	Retrieves all smart collection nodes, as created with createSmartCollection(). Returns an array of Thumbnail Objects for the collection nodes.

hide()
`app.hide ()`

Hides or minimizes all Adobe Bridge browser windows.

- In Mac OS, performs the platform-specific hide gesture.
- In Windows, does the equivalent of `app.document.minimize()`.

Returns `undefined`.

importKeywordsFromFile()
`app.exportKeywordsFromFile (keywordsFile[, importType])`

Imports the keywords from a file into the Keywords palette. This is the same as choosing Import from the flyout menu in the Keywords palette.

Returns `undefined`.

keywordsFile	The file, specified as a path string or ExtendScript File object.
importType	Optional. How to handle existing keywords in the palette. One of these strings: **clearExistingKeywords**—Replace existing keywords in the palette. **mergeWithExistingKeywords** (default)—Merge the imported keywords with any existing keywords in the palette.

isCollectionMember()
`app.isCollectionMember (collection, member)`

Reports whether a given thumbnail is a member of a given collection.

Returns `true` if the thumbnail is a member.

collection	The Thumbnail Object for the collection node, as returned by `app.`createCollection().
member	The Thumbnail Object for the node to be tested.

isProcessingJob()
`app.isProcessingJob ()`

Whether Adobe Bridge is processing any task.

Returns `false` if all the jobs are finished.

makeSearch()
`app.makeSearch (scope, searchSpec)`

Adobe Bridge passes the search specification to execute a search and returns a Thumbnail Object which represents a search-result container node. Users can access the `children` property of the Thumbnail Object to get all matched nodes.

scope	A Thumbnail Object for the target container node.
searchSpec	A SearchSpecification Object used to generate this search result.

openUrl()
`app.openUrl (url)`

Opens a page in the platform's default web browser.

Returns `undefined`.

url	The URL for the page to open.

operationChanged() `app.operationChanged (operator)`	Notifies Adobe Bridge of an update to the processing status or progress of a long-running background operation implemented by a ProgressOperator Object. Adobe Bridge queries the object to find the current status and update the UI as appropriate. Returns `undefined`.
operator	The Operator Class instance that encapsulates the operation, returned by an ExtensionModel Object method.
purgeAllCaches() `app.purgeAllCaches ()`	Purges the thumbnail caches for all folders. See also buildFolderCache() and purgeFolderCache(). Returns `undefined`.
purgeFolderCache() `app.purgeFolderCache ([path])`	Purges the thumbnail caches for the specified folder. See also buildFolderCache() and purgeAllCaches(). Returns `undefined`.
path	Optional. The folder to purge. An ExtendScript `Folder` object, Thumbnail Object for a folder, or Bridge URI path string. If this specifies a file, the cache is purged for the containing folder. If not supplied, purges all folder caches.
quit() `app.quit ()`	Shuts down the Adobe Bridge application. All browser windows are closed. Returns `undefined`.
registerExtension() `app.registerExtension (handler)`	Adds a script-defined node-handling extension to the application's list of available handlers. Returns `true` on success, `false` if there is an existing extension with the same name.
handler	The ExtensionHandler Object.
registerInfoset() `app.registerInfoset (handler, infoset)`	Declares a new node-data information set, associating it with a node-handling extension. Sets can be associated with multiple handlers. All handlers must support the core sets. Registering a set makes the defined properties available to node display code. Returns `true` on success.
handler	The ExtensionHandler Object.
infoset	The Infoset Object.

registerInspectorPanel() app.registerInspectorPanel (*panel*)	Registers a script-defined Inspector panel, adding it to app.inspectorPanels. This panel appears in the Inspector palette unless the selected thumbnail explicitly disallows it. Returns undefined.
panel	The InspectorPanel Object.
registerPrefix() app.registerPrefix (*prefix, handler*)	Associates a URI prefix string with a node-handling extension. The prefix identifies a node type managed by the handler. Handlers can register multiple prefixes. Returns true on success.
prefix	The prefix string, which must contain only ASCII characters.
handler	The ExtensionHandler Object.
removeCollectionMember() app.removeCollectionMember (*collection, member*)	Removes one or more member thumbnails from a collection. Returns undefined.
collection	The Thumbnail Object for the collection node, as returned by app.createCollection().
member	The Thumbnail Object or array of Thumbnail Objects for the member or members to be removed.
removeCredits() app.removeCredits (*title*)	Removes a text string from the Credits area of the Adobe Bridge About box. The string must have been previously added with addCredits(). Returns undefined.
title	The unique identifying name of the addition to remove.
removeLegalNotice() app.removeLegalNotice (*title*)	Removes a text string from the Legal Notice area of the Adobe Bridge About box. The string must have been previously added with addLegalNotice(). Returns undefined.
title	The unique identifying name of the addition to remove.
renameCollection() app.renameCollection (*collection, name*)	Renames a collection. Returns true on success.
collection	The Thumbnail Object for the collection node, as returned by app.createCollection().
name	The new name of the collection. If a collection with this name already exists, a unique name is generated using this string.
renameSmartCollection() app.renameSmartCollection (*collection, name*)	Renames a smart collection. Returns true on success.

collection	The Thumbnail Object for the collection node, as returned by app.createSmartCollection().
name	The new name of the collection. If a collection with this name already exists, a unique name is generated using this string.
runSlideshow() app.runSlideshow (*sources*)	Loads a set of files or thumbnails as a slideshow, using the Preference options currently set for Adobe Bridge. Returns undefined.
sources	An array of Thumbnail Objects or file path strings.
scheduleTask() app.scheduleTask (*script, delay*[, *repeat*])	Executes a script after a specified delay. The script can be executed repeatedly, stopping when it returns undefined, or when you cancel the task using cancelTask(). Returns the task ID number, which can be used to cancel the scheduled task. See the *Adobe Bridge JavaScript Guide* and Adobe Bridge SDK for examples.
script	A string containing the script to be run. **NOTE:** If this script needs to load another script, do not use the JavaScript eval() function; instead use the ExtendScript $.evalFile() function. See the *JavaScript Tools Guide*.
delay	A number of milliseconds to wait before executing the script. If 0, waits the default number of milliseconds, which is 10.
repeat	Optional. When true, execute the script repeatedly after each elapsed delay. Stops when a script execution returns undefined, or when this task is cancelled by calling app.cancelTask(). Default is false, which means execute the script only once.
system() app.system (*commandLine*)	Issues the argument to the operating system, as if it were entered on the command line in a shell. Control does not return to Adobe Bridge until this function returns. Returns undefined.
commandLine	The command to pass to the operating system.
unregisterExtension() app.unregisterExtension (*handler*)	Removes a node-handling extension, previously registered with registerExtension(), from the application's global list. Returns true on success.
handler	The ExtensionHandler Object.
unregisterInfoset() app.unregisterInfoset (*handler, infoset*)	Removes the association between an extension and an information set, previously established with registerInfoset(). Returns true on success.

handler	The ExtensionHandler Object.
infoset	The Infoset Object.
unregisterInspectorPanel() app.unregisterInspectorPanel (*panel*)	Removes a script-defined Inspector panel from the global list in app.inspectorPanels. Returns undefined.
panel	The InspectorPanel Object.
unregisterPrefix() app.registerInfoset (*prefix*, *handler*)	Removes a node URI prefix from the list of prefixes that the associated node-handling extension manages. Returns true on success.
prefix	The prefix string.
handler	The ExtensionHandler Object.

BitmapData Object

Represents an image as a matrix of pixels, described by four channels: red, green, blue, and an "alpha" channel that represents the opacity of the pixel. Each channel stores a number between 0 and 255. For the color channels, 0 means an absence of that color and 255 means the maximum amount of that color. For the alpha channel, 0 means the pixel is completely transparent and 255 means it is completely opaque.

This object allows direct manipulation of the pixels in memory. They are assumed to be stored in row-major order, with consecutive bytes for red, green, blue, and alpha channel. Each row may have some padding at the end, and the total width of a row, in bytes, is represented by `rowBytes`.

The maximum width and maximum height of a `BitmapData` object is 8192 pixels.

BitmapData object constructors

There are three forms for the constructor:

```
new BitmapData (width, height, transparent*, fillColor*)
new BitmapData (file[, preserveColorProfile])
new BitmapData (width, height, transparent, rowBytes, data)
```

Arguments for the first form:

`width`	Number	Image width in pixels.
`height`	Number	Image height in pixels.
`transparent`	Boolean	Optional, true to support per-pixel transparency. Default is true.
`fillColor`	Color	Optional, the fill color. Can be the object, or any of the valid constructors of a Color object; for example, the string "#FF4450" or the integer 0xFF4450. Default is Black.

Argument for the second form:

`file`	String or `File`	The path or `File` object for an image file.
`preserveColorProfile`	Boolean	Optional. True to preserve the embedded color profile, if any. If none is present, or if not supplied, embeds the default `sRGB` profile. **NOTE:** ACR cannot be used with a preserved embedded profile; it returns all images with an `sRGB` profile which would conflict with the desired color profile behavior. If you choose to preserve the embedded profile, the standard JPEG or TIFF libraries are used, even if the thumbnail preference "Use ACR for JPEG and TIFF" is set.

Arguments for the third form:

width	Number	Image width in pixels.
height	Number	Image height in pixels.
transparent	Boolean	True to support per-pixel transparency.
rowBytes	Number	The length in bytes of a row of pixels in the supplied data.
data	Number or Array of Number	The memory address of an ARGB pixel buffer, a 32-bit integer or an array of two 32-bit integers.

BitmapData properties

checksum	Number	A 32-bit Adler checksum of the image data: use to compare two object to see if they represent the same image. Read only.
height	Number	Image height in pixels. Read only.
pointer	Number	A pointer to the buffer storing the matrix of pixels. Read only.
rectangle	Array of Number	The rectangle that defines the size of the bitmap image, in the format [0, 0, w, h]. Origin is top left. Read only.
rowBytes	Number	The length in bytes of a row of pixels. This provides the offset from a given pixel to the pixel immediately below it, allowing for padding at the end of each line. Because a pixel is typically represented by 4 bytes, the value is usually around 4 times bigger than the width in pixels. Typically, rows are padded to multiples of 4, sometimes 16. For example, if a bitmap is 3 pixels wide, width is 3, and rowBytes could be 12 or 16. Read only.
transparent	Boolean	True if the bitmap image supports per-pixel transparency. Read only.
width	Number	Image width in pixels. Read only.

BitmapData functions

clone()
bitmapDataObj.clone ()

Duplicates this object, creating a new object with an exact copy of the contained bitmap.

Returns the BitmapData object.

dispose()
bitmapDataObj.dispose ()

Explicitly frees the memory used to store pixel data for this object. If not called, the JavaScript garbage collector eventually frees the memory when there are no references remaining.

Returns undefined.

exportTo()
bitmapDataObj.exportTo (*path*[, *jpegQuality*])

Writes the image data to a file in JPEG format.

Returns undefined.

NOTE: If you create a JPG file with this method in a folder that is already displayed in the browser window, for certain file systems the browser may not update the display. In this case, the script should call verifyExternalChanges() for any currently displayed thumbnail, to ensure that the browser updates to display the generated file.

path	An ExtendScript File object for the target file. Creates the file if it does not exist, or overwrites an existing file. It is recommended that the file name have an extension of ".jpg".
jpegQuality	Optional. The quality of the image. A number in the range [0..100] where 100 is the highest quality image and largest file size, and lower values indicate more compression, lossier image, and smaller file size. Default is 60 (equivalent to Photoshop quality 7).

getPixel()
bitmapDataObj.getPixel (*x*, *y*)

Retrieves the color data for a specific pixel from the image. If the transparent property for this object is true, the returned color number is pre-multiplied.

Returns an integer that represents the ARGB pixel value. This can be used to create a Color Object.

x, *y*	The pixel coordinates relative to this bitmap's origin, the top left.

getPixel32()
bitmapDataObj.getPixel32 (*x*, *y*)

Retrieves the color data for a specific pixel from the image, including its alpha channel.

Returns an integer that represents the ARGB pixel value. This can be used to create a Color Object.

x, *y*	The pixel coordinates relative to this bitmap's origin, the top left.

loadFromJpegStream() *bitmapDataObj*.loadFromJpegStream (*data*, *dataSize*)	Loads the JPEG stream at a memory address into this object, replacing the previous content. The object is resized, if necessary. Returns `undefined`.
data	The address of the data stream, a 32-bit value or an array of two elements containing the low word and high word of a 64-bit address.
dataSize	The length of the data buffer in bytes.
loadFromPngStream() *bitmapDataObj*.loadFromPngStream (*data*, *dataSize*)	Loads the PNG stream at a memory address into this object, replacing the previous content. The object is resized, if necessary. Returns `undefined`.
data	The address of the data stream, a 32-bit value or an array of two elements containing the low word and high word of a 64-bit address.
dataSize	The length of the buffer in bytes.
resize() *bitmapDataObj*.resize (*dimension*[, *quality*])	Resizes the bitmap to the specified dimensions. The target dimensions must be smaller than the largest of the current bitmap dimensions. Returns a new `BitmapData` object whose sides are no greater than the specified dimensions, or `undefined` if the object already satisfies this condition.
dimension	The desired edge size, in pixels, of the resized image. The resized image is obtained by scaling down the source image to fit into a square with sides that are this number of pixels.
quality	Optional. The algorithm to use in scaling the image. One of these strings: **bilinear** (default)—Lower quality, but faster scaling. **bicubic**—Higher quality, but slower scaling. **bicubicSharper**—Slowest, but best quality.
rotate() *bitmapDataObj*.rotate (*angle*)	Rotates the bitmap by the specified multiple of 90 degrees. Returns a new `BitmapData` object containing the rotated image.
angle	The rotation angle in degrees. Positive values rotate clockwise, negative values rotate counterclockwise. Allowed values are -90, 0, 90, 180, 270.
setPixel() *bitmapDataObj*.setPixel (*x*, *y*, *color*)	Sets the color data for a specific pixel from the image. The alpha channel is set to 255 (fully opaque). Returns `undefined`.
x, *y*	The pixel coordinates relative to this bitmap's origin, the top left.
color	A Color Object, or an integer that represents the RGB pixel value, or a predefined color name string.

`setPixel32()` *`bitmapDataObj`*`.setPixel32` `(`*`x, y, color`*`)`	Sets the color data for a specific pixel from the image, including its alpha channel. Returns `undefined`.
`x, y`	The pixel coordinates relative to this bitmap's origin, the top left.
`color`	A Color Object, or an integer that represents the RGB pixel value, or a predefined color name string.

Color Object

This object represents a pixel in the `sRGB` color space, with an optional alpha channel for opacity.

Color object constructor

To create a new object, use the `new` operator:

```
new Color (red, green, blue[, alpha]);
```

Parameters set corresponding properties.

```
new Color (colorValue);
```

colorValue	The color expressed as a 32-bit ARGB value.

```
new Color (name);
```

name	A W3C CSS standard color name string, one of:
	`aqua, black, blue, fuchsia, gray,`
	`green, lime, maroon, navy, olive,`
	`purple, red, silver, teal, white, yellow`
	Alpha channel value is set to 255, fully opaque.
	If the string is not a recognized color name, returns an object for opaque black.

Color properties

alpha	Number [0..255]	Optional. Degree of opacity when the color is composited. An integer in the range [0..255]. If not specified, default is 255, fully opaque.
blue	Number [0..255]	Blue component value.
green	Number [0..255]	Green component value.
number	Number	The color expressed as a 32-bit ARGB value.
red	Number [0..255]	Red component value.

Color functions

toString() *obj*.toString ( )	Retrieves the hexadecimal value of this color, including the alpha channel. Returns a text string, such as "#FF00FF00" for fully-opaque green.

Document Object

Represents an Adobe Bridge browser window. The user can create multiple browser windows by selecting the **File > New Window** command. For each browser window, there is one `Document` instance.

Access the object for the active browser window using `app.document`.

Access an array of objects for all open browser windows in `app.documents`.

For a discussion of how the parts of the browser window map to JavaScript objects, see the *Adobe Bridge JavaScript Guide*.

Document object constructor

To create a new object, use the `new` operator. The argument specifies the file or page to be selected and displayed in the new window.

```
new Document (osPath | thumb | file | folder)
```

osPath	String	The path or URL for the file or page.
thumb	Thumbnail	The Thumbnail Object for the file or page.
file	File	The `File` object for the selected file.
folder	Folder	The `Folder` object for the selected folder.

Document properties

additionalMetadata	Array of String	Identifies up to four lines of additional metadata to display for thumbnails in the Content pane. Overrides the values set in the **Additional Thumbnail Metatdata** drop-down lists and checkboxes in the Thumbnails page of the Preferences dialog, and any value set in `Preferences.extraMetadata`, but does not change the preference values.
		
		Read/write. The first value in the array sets the first line of additional metadata, the second value sets the second line, and so on.
		
		Allowed values are:
		
		`author`
		`bit-depth`
		`color-mode`
		`color-profile`
		`copyright`
		`date-created`
		`date-modified`
		`description`
		`dimensions`

		`document-creator`
		`document-kind`
		`exposure`
		`file-size`
		`focal-length`
		`keywords`
		`label`
		`opening-application`
		
		An array value of `undefined` turns off the display of metadata for that line.

`allowDrags`	Boolean	When `true` (the default), drag-and-drop of thumbnails is allowed in this browser window. When `false`, thumbnails cannot be dragged within or from this browser window.
`browserMode`	String	The browser window mode, corresponding to the UI button on the upper right, "Switch to compact mode". Value can be `full` or `compact`. Ultra-compact mode has no scripting equivalent. Read/write.
`context`	Thumbnail	The Thumbnail Object a user has right-clicked to invoke a context menu. Otherwise `undefined`. Read only.
`displayInspectorView`	Boolean	When `true`, this browser window displays the Inspector palette, showing the panels listed in `app.inspectorPanels`. When `false`, the Inspector is not shown. Read/write.
`groupedSelections`	Array of Array of Thumbnail	A list of selections in the current Content page, where each member is an array containing a single selected Thumbnail Object, or an array of Thumbnail Objects that make up a selected stack. See `Document.stacks`. Read only.
`height`	Number	The height of the browser window in pixels. Legal values are positive integers. Read/write. The window is resized only within the limits of the minimum and maximum size allowed by the screen resolution.
`hwnd`	Number	In Windows only, a platform-specific handle to the window for this browser.
`id`	Number	A unique identifier for the browser window, valid for the life of the window. It is possible for more than one `Document` object to reference the same window. Read only.

jsFuncs	Object	DEPRECATED. Do not use.
		
		A JavaScript object containing the function definitions for one or more callbacks, in the form:
		
		`{ fnName1: function( [args] )`
		`  { fn1_definition },`
		`  fnName2: function( [args] )`
		`  { fn2_definition }`
		`  ...`
		`}`
		
		These functions are available to the code in an HTML page displayed in the Content pane, which can invoke them using the `call` function. They run in Adobe Bridge's ExtendScript engine, and can use Adobe Bridge DOM objects. Read/write.

maximized	Boolean	When `true`, this browser window is in the zoomed or maximized state. Read only.
minimized	Boolean	When `true`, this browser window is in the collapsed or minimized state. Read only. **NOTE:** In Mac OS, a window can be in the zoomed state, and still be minimized. If both `maximized` and `minimized` are `true`, call the document's restore() method to un-zoom the window.
navbars	NavBar	Contains the predefined NavBar Objects for the configurable navigation bars. • To access the navigation bars that can be shown when the Content pane displays files and folders, use `navbars.filesystem.top` and `navbars.filesystem.bottom`. Both of the two bars can be configured to display ScriptUI controls, and are hidden by default.
noItems	String	Text to be displayed in the Content pane when the selected thumbnail is for an empty folder. The default is "No Items to Display". Read/write.
owner	String	The Adobe Bridge-enabled application that created or first activated this browser window, if it was not Adobe Bridge. An application specifier, such as `illustrator` or `photoshop`. For details of application specifier format, see the *JavaScript Tools Guide*.
palettes	Array of TabbedPalette	A collection of TabbedPalette Object for all default and script-defined display palettes available to this browser, regardless of their visibility status. Read only.

position	Object	The position of this browser window on the screen. An object with two properties, `x` and `y`, whose value is the point of the screen coordinates, the screen coordinates are relative to the upper-left corner of the main display. Read/write.
presentationMode	String	The presentation mode of the Content pane. The value is always "browser" in CS5. In support of this, setPresentationMode() and presentationPath now support the "browser" mode only.
presentationPath	String	The path to the content displayed in the Content pane. A Bridge URI, which is a valid filesystem path that Adobe Bridge can interpret. This property no longer supports URL. To display an HTML page, use a TabbedPalette Object.
selectionLength **selectionsLength**	Number	The number of currently selected thumbnails in the Content pane.

selections	Array of Thumbnail	The Thumbnail Objects for all currently selected files in the Content pane of this document. Read only. Change the selections using the Document Object's select(), selectAll(), deselect() and deselectAll() methods. A script should wait until the loaded event has occurred before making calls to document selection methods.
		
		Use getSelection() to limit the request to visible thumbnails, or those for files of a given type. Use groupedSelections to include thumbnails that are in selected stacks.
		
		TIP: Accessing this value is a time-intensive operation. To improve performance, access it outside loops:
		
		`var sel = document.selections;`
		`for (var i = 0; i < sel.length; i++)`
		`process(sel[i]);`
		Also, use selectionLength when possible, rather than checking the length of this array:
		
		`if (document.selectionLength > 0)`

showThumbnailName	Boolean	When `true`, thumbnail names are displayed in the Content pane. This overrides the ShowName preference value. Read/write.

sorts	Array of Objects	How the thumbnails in the Content pane are sorted. References a SortCriterion Object using an array containing one JavaScript object with three properties:
		`{ type, name, reverse }`
		• The `type` value corresponds to the type property of a SortCriterion Object. It is read-only, and is ignored when setting this value.
		Allowed values are:
		`string`
		`date`
		`number`
		`dimensions`
		`resolution`
		`colorProfile`
		`user`
		• The `name` value corresponds to the name property of a SortCriterion Object, and identifies the object that defines the sorting criterion.
		Allowed values are:
		`user`
		`name`
		`date-created`
		`date-modified`
		`label`
		`rating`
		`file-size`
		`document-kind`
		`keywords`
		`dimensions`
		`resolution`
		`color-profile`
		• The `reverse` value is `true` if the thumbnails are sorted in reverse order in the given category.
		To set the value, create an array that contains an object with `name` and `reverse` properties. For example, to sort in reverse by creation date:
		`var mySortObj = {};`
		`mySortObj.name = "date-created";`
		`mySortObj.reverse = true;`
		`var mySortsArray = [];`
		`mySortsArray.push(mySortObj);`
		`app.document.sorts = mySortsArray;`

stacks	Array of Array of Thumbnail	A list of current thumbnail stacks in the Content pane of this document. Each stack is an object with the following properties and functions: • thumbnails—An array in which each item is an array of Thumbnail Objects. • properties—Read/write. An array in which a user can add and retrieve customized properties for the stack. Those properties can stored into the cache file by calling `flushStackProperties()`. • isValid ()—Function. If the stack already exists and is valid, the function returns `true`, otherwise it returns `false`. • isExpanded—Returns `true` if the stack object is expanded and `false` if it is collapsed • collapse()—Collapses the stack object • expand() — Expands the stack object See examples in Bridge SDK samples `SnpManageStacks.jsx`
status	String	The text displayed in the document's status line at the bottom of the Content pane. Read/write.
thumbnail	Thumbnail	The Thumbnail Object for the node currently selected in the Folders or Favorites palette. Read/write. Setting this value navigates to and selects the corresponding node in the Folders pane. The contents of this node are displayed in the Content pane. **NOTE:** The `document.thumbnail.children` array is not populated until the loaded event has occurred for the document.

thumbnailViewMode	String	The view mode of the Content pane, as selected by the **View** menu. Read/write. One of:
		`thumbnails`
		`details`
		`list`

visible	Boolean	When `true`, the browser window is expanded, as opposed to being minimized or collapsed. Read/write. Setting this to `true` is the same as calling `Document.normalize()`. Setting it to `false` collapses the window.

visibleThumbnails	Array of Thumbnail	Read only. An array of Thumbnail Objects that are currently shown in the Content pane. The array is ordered according to the current sort order, and contains only thumbnails whose `visible` property is `true`.
		
		TIP: Accessing this value is a time-intensive operation. To improve performance, access it outside loops:
		
		`var sel = document.visibleThumbnails;`
		`for (var i = 0; i < sel.length; i++)`
		`process(sel[i]);`
		Also, use selectionLength when possible, rather than checking the length of this array:
		
		`if (document.visibleThumbnailsLength > 0)`

visibleThumbnailsLength	Number	Read only. The number of thumbnails in the visibleThumbnails array.
visitUrl	Function	DEPRECATED. Do not use. A callback function that is called when the Content pane is about to open a URL. Allows the script to approve or redirect the browser. The function takes the URL as an argument, and should return an object with these properties: **result**—When `false`, Adobe Bridge does not open the new URL. When `true`, it opens the passed URL or a different URL as provided in this object.

		url—When present, a URL string that replaces the passed URL.
		toHistory—When `false`, the passed or provided URL is not added to the browser's history list. Default is `true`.
		
		For example, this confirms a link with the user:
		
		`var myFn = function(url) {`
		`  if( Window.confirm( "Proceed to " +`
		`  url + " ?" ) )`
		`  return {result:true};`
		`  else`
		`  return {result:false};`
		`}`
		`app.document.visitUrl = myFn;`
		This example replaces a link to an unwanted page with an application-specific help page:
		
		`var helpPageFn= function(url) {`
		`  if( url == "unwanted_page" )`
		`  return {result:true,`
		`  url:"my_help_page",`
		`  toHistory:false};`
		`  else`
		`  return {result:true};`
		`}`
		`app.document.visitUrl = helpPageFn;`
		Within the context of this function, the implicit `this` variable references this Document Object. For example:
		
		`var myFilter = function(url) {`
		`Window.alert(this.thumbnail.displayPath);`
		`return {result:true, url:url};`
		`}`
		NOTE: This function is also called when the Content pane switches from a web page view to a filesystem view. In this case, the URL passed to the function is `"about:blank"`.

width	Number	The width of the browser window in pixels. Legal values are positive integers. Read/write. The window is resized only within the limits of the minimum and maximum size allowed by the screen resolution.

workspace	Object	Retrieves the most recently set workspace, a JavaScript object with two properties, `id` and `name`, whose string values are the unique identifier and display name of the workspace. A user-defined workspace may have been renamed by the user since being set by a script. Read-only. Set with `Document.setWorkspace()`. The current workspace can also be set by user action. Value is `undefined` for a new document before any workspace has been explicitly set. Bridge CS5 uses the XML encoding method instead of the URL encoding method. This changed in CS5. If users copy an old workspace file from CS4 to CS5, and the workspace name contains non-ASCII characters, Bridge CS5 will not translate it. This will cause an issue where the workspace name is not the name which the user saved before. To resolve this problem, users can simply save a new workspace name in Bridge CS5.

Document functions

bringToFront() *docObj*.bringToFront ()	Makes this browser window the topmost active window in the windowing system. Makes Bridge exit stealth mode if it is in that mode. Returns `undefined`.
center() *docObj*.center ()	Centers this browser window on the screen. If there is more than one monitor, centers the window on the monitor where most of the window resides. Returns `undefined`.
chooseMenuItem() *docObj*.chooseMenuItem(*menuId*)	Executes an Adobe Bridge-defined or script-defined menu command programatically. This is the equivalent to the user selecting the command interactively. Returns `undefined`.
menuID	The unique identifier for the command to execute; see MenuElement Object. Predefined identifiers for Adobe Bridge commands are listed in 'Adobe Bridge menu and command identifiers' on page 59. If the ID is for a menu or submenu, the function does nothing.
close() *docObj*.close ()	Closes this browser window. Returns `undefined`.

deselect() *docObj*.deselect(*thumbnail*)	If the specified thumbnail is a child of this document and is selected, removes it from the selections array and deselects it in the browser window. Returns `true` if the thumbnail was deselected. A script should wait until the loaded event has occurred before making calls to document selection methods.
thumbnail	The Thumbnail Object for the node to deselect.
deselectAll() *docObj*.deselectAll ()	Removes all members from the selections array and deselects all thumbnails in the browser window. Returns `undefined`. A script should wait until the loaded event has occurred before making calls to document selection methods.
execJS() *docObj*.execJS (*script*)	DEPRECATED. Do not use. Executes a JavaScript function that is defined within the HTML page displayed in the Content pane when a thumbnail with `displayMode=web` is selected. If the page that defines the function is not currently displayed, causes a run-time error. **NOTE:** Do not call this method from a jsFuncs callback function. This attempts to re-enter the JavaScript engine, which is not allowed, and causes Adobe Bridge to hang. A callback can, instead, schedule a task using `app.scheduleTask()`, and call `execJS` from the function associated with the task. Returns the result of the executed JavaScript function, which must be a Boolean, Number, or String, or `null`.
script	A string containing a script to execute. This typically contains the name and arguments of the JavaScript function to execute, but can have multiple statements, including variable declarations, assignments and so on.
flushStackProperties() *docObj*.flushStackProperties ()	Stores stack properties into the cache file. Returns `undefined`.
getSelection() *docObj*.getSelection([*filter*])	Collects selected thumbnails for files of a given type, if any are selected. If no matching thumbnails are selected, collects matching thumbnails that are currently visible in the Content pane. See examples in Bridge SDK samples `SnpSaveAsJPEG.jsx` and `SnpRotateImage.jsx`. Returns an Array of Thumbnail Object.
filter	Optional. A String containing a comma-delimited list of file extensions to match, or the wildcard character `"*"` to match all file extensions. `"*"` is the default.

getStackforThumbnail () *docObj*.getStackforThumbnail (thumbnail)	Returns the stack object for the passed Thumbnail Object. See examples in Bridge SDK samples `SnpManageStacks.jsx`
maximize() *docObj*.maximize ()	Maximizes or zooms this browser window. Returns `undefined`.
minimize() *docObj*.minimize ()	Minimizes or docks this browser window. Returns `undefined`.
normalize() *docObj*.normalize ()	Centers this browser window on the screen, and sets the height and width to 80% of the screen height and width. Returns `undefined`.
print() *docObj*.print ()	DEPRECATED. Do not use. Prints the page shown in the Content pane, if it shows a web page. Returns `true` on success.
refresh() *docObj*.refresh ()	Refreshes the display of this browser window. Returns `undefined`.
resetToDefaultWorkspace() *docObj*.resetToDefaultWorkspace ()	Restores the default configuration of the tabbed palettes in this browser window. The equivalent of choosing **Window** > **Workspace** > **Reset**. This works only when browserMode is `full`. If browserMode is `compact`, it does nothing. Returns `undefined`.
restore() *docObj*.restore ()	Restores this browser window after it has been minimized. In Windows, makes it user-sizeable. In Mac OS, returns it to the user-configured size. Returns `undefined`.
reveal() *docObj*.reveal(*thumbnail*)	Causes the Content pane (not the Folders or Favorites palette) to show the specified thumbnail, scrolling the display if necessary to make it visible. Does not select the Thumbnail. Returns `undefined`.
thumbnail	The Thumbnail Object for the node to scroll to.

select() *docObj*.select(*thumbnail*)	If the specified thumbnail is a child of this document and is not selected, adds it to the selections array and selects it in the Content pane. This is the same as selecting the icon in the Content pane with CONTROL-click. Returns `true` if the thumbnail was selected. A script should wait until the loaded event has occurred before making calls to document selection methods.
thumbnail	The Thumbnail Object for the node to select.
selectAll() *docObj*.selectAll ()	Adds all child Thumbnail Objects of the current thumbnail (`document.thumbnail`) to the selections array, and selects them in the Content pane. This is the same as typing CONTROL-a (in Windows) or CMD-a (in Mac OS) in the Content pane. Returns `undefined`. A script should wait until the loaded event has occurred before making calls to document selection methods.
setPresentationMode() *docObj*.setPresentationMode (*mode*, [*path*])	Sets the presentation mode of the Content pane, and optionally the path to the current content to display. The mode determines how the presentationPath value is interpreted. CAUTION : In CS5, only the `browser` mode is supported. Setting the presentation mode to anything else will cause the `presentationPath` property to not work properly. To display an HTML page, use the TabbedPalette Object. Returns undefined.
mode	String. The new display mode. In CS5, must be `browser`, otherwise the `presentationPath` property will not work correctly.
path	Optional. The path string, a Bridge URI.

setWorkspace()
docObj.setWorkspace(*workspaceId*)

Sets the browser configuration to a predefined or user-defined workspace.

The current workspace can also be set by user action. A user-defined workspace may have been renamed by the user since being set by a script. If an invalid ID is assigned, the workspace is not changed.

If a script-defined tabbed palette is visible when the user or a script creates a workspace, the workspace references that palette by its unique identifier. If a workspace references a script-defined tabbed palette, the palette must be created before the workspace is applied. Otherwise, the palette does not appear.

NOTE: This works only when browserMode is `full`. If browserMode is `compact`, it does nothing.

Returns `true` on success.

workspaceId — The unique, identifying name string for the new workspace. If it is the same as the name of the current workspace, the function does nothing.

Identifiers of predefined workspaces are:

```
default
lightTable
navigator
metadata
horizontalFilmstrip
verticalFilmstrip
```

Event Object

Represents a user-interaction event, such as clicking a thumbnail. Adobe Bridge creates an `event` object whenever one of the triggering events occurs, and passes it to any event handlers that you have registered with the App Object's eventHandlers property. The only way to access an `event` object is as the argument to such an event-handling function. See the *Adobe Bridge JavaScript Guide* for details of how to define and register these functions.

The object with which the user interacted to generate the event is called the *target object* of that event. Different target object types are associated with different types of events, as listed in Event object types.

`Event` defines no functions.

Event properties

`appPath`	String	When the `type` is openWith, the platform-specific path to the selected opening application. Otherwise `undefined`. Read only.
`document`	Document	When the target object is a Thumbnail Object, the Document Object for the browser window in which the event occurred. Otherwise `undefined`. Read only.
`favorites`	Favorites	When `location` is `favorites`, the Favorites Object for the pane in which the event occurred. Otherwise `undefined`. Read only.
`isContext`	Boolean	When the target object is a Thumbnail Object, and the `type` is selectionsChanged or selectionsChanging, this value is `true` if the event was generated by a right-click (the gesture that normally brings up a context menu). Otherwise `false`.
`location`	String	The location at which the event occurred. This value helps to distinguish events of the same type than can be triggered in different ways. One of: **`app`**—The target object is the App Object and the event was generated for an interaction with the operating system. **`document`**—The target object is a Thumbnail Object and the event was generated for an interaction in the Folders pane, or the target object is a Document Object and the event was generated for an interaction with the windowing environment. **`favorites`**—The target object is a Thumbnail Object and the event was generated for an interaction in the Favorites palette. **`prefs`**—The target object is the PreferencesDialog Object and the event was generated in the Preferences dialog. **`web`**—The target object is a Document Object and the event was generated for an interaction with the Internet. In this case, `event.url` contains the URL of the page. Read only.

object	Thumbnail, Document, App, PreferencesDialog	The target object of the event; that is, the object that generated the event. Read/write.
section	String	When `location` is `favorites`, one of: **standard**—The target object is a predefined member of the `favorites` array. **user**—The target object is a user-added member of the `favorites` array. Otherwise `undefined`. Read only.
type	String	The type of action that triggered the event. Different types of events that are supported for each type of target object; see Event object types. Read only.
uri	String	The Bridge URI of a node that was affected by the event.
url	String	When `location` is `web`, the URL of the web page. Read only.

Event object types

Events of different types are triggered for different target objects. All types are described here according to the target object.

App events

When an application event occurs, the `event` object has the following property values:

- The target, *`eventObj`*`.object`, is the App Object.
- The location, *`eventObj`*`.location`, is the string `app`.
- The type, *`eventObj`*`.type`, is one of these event types:

close	Generated when the Adobe Bridge application has received a request to terminate, but has not yet started the process. If the handler returns a `handled` value of `true` in the result object, the termination is cancelled. To query the user, you can set this with the return value of `Window.confirm`. For example: `return { handled: Window.confirm("Really quit?") };`
destroy	Generated when the Adobe Bridge application terminates. Occurs when the user exits from Adobe Bridge by selecting the **File > Exit** command, when the user closes the final open document, or when a script calls the App Object's quit() function. The handler cannot override the default shutdown behavior, but it can take additional actions before the shutdown completes.

Document events

You cannot define event handlers that override the default behavior of Document events. You can, however, write an event handler to take additional actions prior to the event. Such a handler could return a `handled` value of `true` in the result object, to prevent the default behavior, but this is not recommended.

When a document event occurs, the `event` object has the following property values:

- The target, *`eventObj`*`.object`, is a Document Object.
- The location, *`eventObj`*`.location`, can be `app`, `web`, or `document`, depending on the type.
- The type, *`eventObj`*`.type`, is one of these event types:

`create`	Location is `app`. Generated when a new document is created. Occurs when the user selects the **File > New Window** command, or when a script creates a new document with a constructor call. The new Document Object is passed to the event handler, but `app.document` still contains the previous `Document` object.
`deselect`	Location is `document`. Generated when the OS window focus is removed from the browser window.
`destroy`	Location is `app`. Generated when a browser window is closed. Occurs when the user selects the **File** > **Close Window** command in the UI, when a script closes a browser window using the Document Object's `close ()` method, or when Adobe Bridge closes a browser window because the application is terminated.
`loaded`	Location is `app`. Generated when the Content pane has finished its first iteration through the files to be displayed. The `Document.thumbnail.children` array is not populated until some time after this event has occurred for the document. A script should delay making calls to document selection methods such as select() and deselect().
`open`	Location is `document`. Generated when the browser gets the input focus. The new Document Object is passed to the event handler, but `app.document` still contains the previous `Document` object.
`select`	Location is `document`. Generated when the OS window focuses on the browser window.
`selectionsChanged`	Location is `document`. Generated just after the selection is changed in the UI, as a result of script or user action. The document selections list reflects the post-selection state.
`selectionsChanging`	Location is `document`. Generated just before the selection is changed in the UI, as a result of script or user action. The document selections list reflects the pre-selection state.

workspacesPreLoad	Location is `document`. Generated just before workspaces are loaded from disk into a new Document Object. If your script-created TabbedPalette Object is intended to be part of a workspace, you should create it in handling this event.
workspacesPostLoad	Location is `document`. Generated just after workspaces have been loaded from disk into a new Document Object.

Thumbnail events

When a thumbnail event occurs, the `event` object has the following property values:

- The target, *`eventObj`*`.object`, is a Thumbnail Object.
- The location, *`eventObj`*`.location`, is `document` for an interaction with the Folders or Content pane, or `favorites` for an interaction with the Favorites palette.
    - If `location` is `favorites`, the `favorites` property contains the Favorites Object and the `section` property reflects whether the target thumbnail is a predefined or user-defined member of the favorites array.
- The *`eventObj`*`.document` property contains the Document Object for the browser window in which the event occurred.
- The type, *`eventObj`*`.type`, is one of these event types:

add	Location is `favorites`. Generated when the user adds a new node to the Favorites palette.
hover	Location is `document`. Generated when the cursor hovers over a thumbnail. Your handler can override the text displayed in the tooltip box. Return the text to be displayed in the result object property `tipText`.
modify	Location is `favorites`. Generated when the user modifies new node to the Favorites palette by adding a subnode to it.
move	Location is `favorites`. Generated when the user changes the position of a node in the Favorites palette.

open	Location is `document`. Generated when a file thumbnail in the Content pane is opened with an application other than Adobe Bridge. Occurs when the user successfully opens a thumbnail with the **File > Open** command, or by double-clicking, or when a script calls the Thumbnail Object's `open` method. Also generated when a folder thumbnail is opened in the Content pane, which opens that folder in an Adobe Bridge browser window. If this opens a new browser window, `app.document` contains the `Document` object for the browser in which the thumbnail was clicked, and the new browser that will display the contents is passed to the event handler. By default, Bridge determines which application is used to open a thumbnail based on the file type and Preferences settings. If you want to change this behavior, it is best to try to affect as few file types as possible while still accomplishing the goal of your script. If you do want to override this behavior for all file types, it is better to provide a context menu item rather than overriding the double-click behavior. In the latter case, users will lose the ability to use the Preferences settings through your script.
openWith	Location is `document`. Generated when a user makes a selection of thumbnails in the Content pane, then selects an application from the **Open With** submenu of the **File** or context menu. The object provides a platform-specific path string to the selected application.
preview	Location is `document`. Generated when an image thumbnail in the Content pane is selected. The handler can return an object in which the `result` value is an array containing text captions to display under the image in the Preview pane. For example: `{ handled: true, result: ["my image", "new preview caption"] }` The preview caption can be modified this way for images displayed in filmstrip view as well.
remove	Location is `favorites`. Generated when the user removes a node from the Favorites palette.

PreferencesDialog events

You cannot override the default behavior of a Preferences dialog event. You can, however, write an event handler to take additional actions prior to the default action, such as adding a panel that reflects your own preferences, and interpreting the results from that panel.

When an Preferences dialog event occurs, the `event` object has the following property values:

- The target, *`eventObj`*`.object`, is the PreferencesDialog Object
- The location, *`eventObj`*`.location`, is the string `prefs`.
- The type, *`eventObj`*`.type`, is one of these event types:

cancel	Generated when the user clicks **Cancel** in the Preferences dialog.
create	Generated when the user invokes the Preferences dialog.

destroy	Generated when the user closes the Preferences dialog using the window frame's close button.
disabled	Generated when the user disables a startup script using the Startup Scripts page of the Preferences dialog. The event handler receives an additional argument, the script name, and can remove any Favorites nodes added by a node-handling extension associated with the script. Also generated when the user disables a node in the Standard section of the Favorites palette. In this case, the `event` object's uri property contains the URI of the node.
enabled	Generated when the user enables a startup script using the Startup Scripts page of the Preferences dialog. The event handler receives an additional argument, the script name, and can add any Favorites nodes needed by a node-handling extension associated with the script. Also generated when the user enables a node in the Standard section of the Favorites palette. In this case, the `event` object's uri property contains the URI of the node.
ok	Generated when the user clicks **OK** in the Preferences dialog.

Favorites Object

Represents the navigation nodes that appear in the Favorites palette in the Adobe Bridge browser. The `Favorites` object contains two arrays of Thumbnail Objects; one for the top, or *standard* section, which contains a predefined set of nodes, and one for the bottom, or *user* section, where the user can choose which nodes to display.

While the Folders palette shows the full navigation hierarchy, with all folders and subfolders that Adobe Bridge can access, the Favorites palette shows only certain top-level folders and one level of subfolders. Subfolders in the Favorites palette can be, but are not necessarily, `children` of the `Thumbnail` for the parent node.

Access the `Favorites` object through the App Object's `favorites`, `standardFavorites`, or `userFavorites` properties:

```
currentFavorites = app.favorites
currentStandardFavorites = app.standardFavorites
currentUserFavorites = app.userFavorites
```

Favorites properties

`length`	Number	The number of Thumbnail Objects in the current section of the Favorites palette.
`section`	String	Sets the section of the Favorites palette for the next node operations in the immediate scope. The value does not persist. One of: **`standard`**—The top section of the Favorites palette containing predefined nodes. **`user`** (default)—The bottom section of the Favorites palette containing user-selected nodes.

Favorites functions

`add()` *`favoritesObj`*`.add (`*`thumbnail`*`)`	Appends a new node into the current section of the favorites array, and updates the Favorites palette to show the new node at the root level. Returns `true` on success. If the referenced node is already in the array, returns `false` and does not change the array.
`thumbnail`	The Thumbnail Object for the node to add.
`addChild()` *`favoritesObj`*`.addChild (`*`parentNode, subNode`*`)`	Inserts a new subnode into the current section of the favorites array, and updates the Favorites palette to show the new node below its parent when the parent is selected. Returns `true` on success. If the specified parent node is not in favorites array, returns `false` and does not add the subnode.
`parentNode`	The Thumbnail Object for the parent node. Must be a root node in the favorites array.

subNode	The Thumbnail Object for the subnode. This node can be, but does not need not to be a child of the parent `Thumbnail`. It is not added to the parent's `children` array.
`associateWorkspace()` *`favoritesObj`*`.associateWorkspace(`*`thumbnail, workspace`*`)`	Associates a named workspace with a thumbnail in the standard section of the Favorites palette. When the user clicks this thumbnail, this workspace becomes current. Returns `true` on success. If the thumbnail is in the user section, or is not in the Favorites palette, returns `false` and does nothing.
thumbnail	The Thumbnail Object.
workspace	The workspace name. See `Document.workspace`.
`clearAll()` *`favoritesObj`*`.clearAll ()`	Deletes all the nodes from the current section of the favorites array and updates the Favorites palette. Returns `undefined`.
`contains()` *`favoritesObj`*`.contains(`*`uri`*`)`	Reports whether the list of favorites currently contains a specific node, either in the standard or user sections. Returns `true` if the node is in the current favorites list, `false` otherwise.
uri	The Bridge URI string for the node.
`disable()` *`favoritesObj`*`.disable(`*`uri`*`)`	Disables a node from the standard section, removing it from display in the browser, but leaving it as an unchecked option in the General page of the Preferences dialog. Returns `undefined`.
uri	The Bridge URI string for the node.
`enable()` *`favoritesObj`*`.enable(`*`uri`*`)`	Enables a node from the standard section, displaying it in the browser, and checking the corresponding option in the General page of the Preferences dialog. Returns `undefined`.
uri	The Bridge URI string for the node.
`getChildren()` *`favoritesObj`*`.getChildren (`*`uri`*`)`	Retrieves the subnodes of a node in the Standard section of the Favorites palette. The node can be in either the enabled or disabled state. In this context, *children* means subnodes added with `Favorites.addChild()`, rather then `Thumbnail` children. Returns an array of URI strings for the child nodes, or `undefined` if the node is not in the Standard section or not in the Favorites palette.
uri	The Bridge URI string for the parent node.

insert() *favoritesObj*.insert (*thumbnail* [, *index*])	Inserts a new node into the current section of the favorites array, and updates the Favorites palette to show the new node at the root level. Returns `true` on success. If the referenced node is already in the array, returns `false` and does not change the array.
thumbnail	The Thumbnail Object for the node to insert.
index	Optional. A 0-based index into the existing node array at which to insert the new node, or an object reference for a node in the existing node array. The node is inserted before this existing node. If the value is beyond the end, is not in the existing node array, or is not supplied, the new node is appended to the end of the array.
remove() *favoritesObj*.remove(*thumbnail*)	Removes the specified script-defined node from the favorites array and updates the Favorites palette. Scripts cannot access predefined nodes. Returns `true` on success.
thumbnail	The Thumbnail Object for the node to remove.

IconListPanelette Object

An instantiable subclass of the Panelette Base Class, representing a member sub-panel of a InspectorPanel Object that displays two columns. The left column contains an icon, and the right column contains text. The text can be static, or can be obtained dynamically from the associated thumbnail at display time. See Panelette markup elements.

IconListPanelette constructor

To create a new object, use the `new` operator. The *`name`* and *`titleMarkup`* parameters set the corresponding properties inherited from the Panelette Base Class.

This version can be used when all display data is known in advance:

```
new IconListPanelette(name, titleMarkup, rows);
```

`rows`	A collection of two-member arrays describing the rows to display in the panelette. Sets the rows property.

This version provides the ability to obtain data dynamically at display time.

```
new IconListPanelette(name, titleMarkup, rows, columnText);
```

`rows`	The icons shown in the first column. An array of Thumbnail Objects or a string containing panelette markup that obtains a set of thumbnails at display time.
`columnText`	A corresponding array of strings, where each member is a line of text with which to label the corresponding thumbnail. The text strings can contain Panelette markup elements to access dynamic data.

IconListPanelette properties

`rows`	Array of Array of 2-member Array	A collection of two-member arrays describing the rows to display in the panelette. Each member of the member arrays corresponds to a column. • The first member of each member array specifies the icon displayed in the first column, as a Thumbnail Object or a 16x16 pixel JPG, PNG, or system icon. • The second member, displayed in the second column, is a string that can contain markup elements to access dynamic data. See Panelette markup elements. Read only.

InspectorPanel Object

Represents an object-inspection panel, a special type of tabbed palette that displays contextual information for a selected thumbnail.

Your script defines what kind of related information to display, and how to display it. The panel serves as a frame and parent for subpanels that actually display the information. Subpanels are represented by members of the Panelette Base Class contained in this parent panel.

- Register a inspection panel that you create to make it available to Adobe Bridge, using the `app` method registerInspectorPanel().
- To turn the display of registered inspection panels on or off in a particular browser window, set displayInspectorView in the Document Object.

InspectorPanel constructor

To create a new object, use the `new` operator:

```
new InspectorPanel(title, displayTitle*, visible*, wide*, sortPosition*);
```

Parameters set the corresponding properties.

InspectorPanel properties

displayInInspector	Boolean	When `true`, this panel is displayed when `Document`.displayInspectorView is `true`, if the hidePanelForThumbnail callback returns `true` or is not implemented. When `false`, this panel is never displayed, and is also hidden in the Inspector page of the Preferences dialog.
displayTitle	String	Optional. The localized title string to display in the panel's tab header. The string can include values derived dynamically at display time, using Panelette markup elements. Read only, supplied on creation. Supplying this value allows you to use the same panel object with different titles for different node types. If this value is not supplied, the `title` value is displayed.
flyoutMenuId	String	The unique menu identifier of a script-defined flyout menu for this panel. See MenuElement Object. Read only.
hidePanelForThumbnail	Function	Optional. A script-defined function that takes a Thumbnail Object as a parameter and returns `true` if the thumbnail allows this panel to be displayed. Called on the focus thumbnail when the Inspector is displayed. Read/write.

minimized	Boolean	When `true`, the panel is minimized or iconified.
panelettes	Array of Panelette	A collection of panelettes contained in this panel, in display order. Read only; use registerPanelette() and unregisterPanelette() to manage the list. Contains instances of the type-specific panelette subclasses: IconListPanelette Object TextPanelette Object ThumbnailPanelette Object
sortPosition	Number	The preferred default position of this tabbed panel in the Inspector, relative to other panels. In the range [1..100]. Panels with lower values are above and to the left. Read/write.
title	String	A unique identifying name for this panel. If no displayTitle is specified, this is shown in the top header bar. Read/write.
visible	Boolean	When `true`, the majority of this panel is visible on the screen. When `false`, the panel is minimized or iconified, or most of it is positioned off the screen. Read only.
wide	Boolean	When `true`, this panel occupies the entire available horizontal space. When `false`, the default, the panel occupies one third of available space. Read/write.

InspectorPanel functions

registerPanelette() *panelObj*.registerPanelette(*panelette*)	Registers a script-defined panelette as a member of this panel, adding it to panelettes list. Returns `true` on success, `false` if the panelette is already registered or the operation fails.
panelette	An instance of one of the type-specific panelette subclasses: IconListPanelette Object TextPanelette Object ThumbnailPanelette Object
unregisterPanelette() *panelObj*.unregisterPanelette(*panelette*)	Removes a member panelette from this panel. Returns `undefined`.
panelette	The panelette instance.

MenuElement Object

The `MenuElement` object is used to represent the application menu bar, menus and submenus, and individual items or commands. Adobe Bridge creates `MenuElement` instances for each of the existing menu elements, and you can create additional instances to extend the existing menus.

A script can execute a menu command using `app.document.` chooseMenuItem().

Existing menu elements that can be extended have predefined identifiers, listed in 'Adobe Bridge menu and command identifiers' on page 59. Not all existing menu elements can be extended. You can only add a new menu or command before or after an existing menu or command, which you must specify using the predefined unique identifier.

Use the `Menu.` create() static function to create new menu items, rather than the `new` operator. This function behaves correctly if a menu item with the same name already exists.

MenuElement class functions

The `MenuElement` class defines these static functions that you can use to extend and work with existing menu elements.

`create()` `MenuElement.create (type, text, location[, id]);`	Adds a new menu to the menu bar, a new submenu to an existing menu, or a new command to an existing menu or submenu. Returns the new `MenuElement` object.
`type`	The type of menu element, one of: **`menu`**—a menu or submenu **`command`**—a menu item
`text`	The localizable string that is displayed as the label text. Script-created menu and menu commands cannot have keyboard shortcuts or icons.
`location`	A string describing the location of the new menu element, with respect to existing menu elements. If the relative element is not found, the new element is appended to the Tools menu. The location specifier can take one of the following forms: **`before`** *`identifier`*—Create the new element before the given menu element. **`after`** *`identifier`*—Create the new element before the given menu element. **`at the end of`** *`identifier`*—Append the new element to the given menu. The identifier must be for a menu, not a command item. **`at the beginning of`** *`identifier`*—Create the new element as the first item in the given menu. The identifier must be for a menu, not a command item. To insert a separator before or after the new element, specify a dash (-) at the beginning or end of the location string. For example, this value draws separators before and after the new element, which is added after the Find submenu in the Edit menu: `-after /bridge/edit/find-` A string that does not conform to these rules causes a run-time error.

id	The unique identifier for this element. Optional. • If the ID of an existing menu or submenu is supplied, the call returns that `MenuElement` object. • If the ID of an existing command is supplied, the call causes a JavaScript error. • If not supplied, the call generates a numeric value, which can be found in the `id` property of the returned menu object.

`find()` `MenuElement.find (id)`	Retrieves a menu element using its unique identifier. Returns the `MenuElement` object for the specified menu or menu item, or `null` if no such element is found.

id String. The unique identifier for the menu element to find.

Example

This example checks to see whether a specific menu item already exists to avoid an error if the script is executed a second time.

```
var menu = MenuElement.find ('myMenuId');
if (menu = null) //element does not yet exist
// add menu element
```

`remove()` `MenuElement.remove (id)`	Removes a script-defined menu or menu item. Returns `undefined`.

id String. The unique identifier for the menu element to remove.

MenuElement properties

`altDown`	Boolean	When `true`, the ALT modifier key was pressed when the item was selected. Read only.
`canBeChecked`	Boolean	When `true`, the menu item is a bi-state item that can be checked. Otherwise, the menu item cannot be checked. Read only.
`checked`	Boolean	When `true`, the command is selected. A check mark appears next to the label. When `false`, the item is not selected, and no check mark is shown. Read/write.
`cmdDown`	Boolean	When `true`, the COMMAND modifier key was pressed when the item was selected. Read only.
`ctrlDown`	Boolean	When `true`, the CONTROL modifier key was pressed when the item was selected. Read only.
`enabled`	Boolean	When `true`, the menu or command is selectable. When `false`, it is grayed out and cannot be selected. Read/write.
`id`	String	A unique identifier. Read only. Identifiers take the form: */app/menu/submenu/command* They are not localized, and are case sensitive.

onDisplay	Function	The callback function that is called when the application is about to display this menu or menu item. The function takes no arguments, and returns nothing. It can change the `enabled` and `checked` properties according to the state of the application. **TIP:** This is called frequently and affects performance. Avoid time-intensive processing, such as checking metadata, or iteration over an entire, large selection. Use `Document.selectionLength` to check the size of the selection before accessing it. If an item is enabled incorrectly, you can handle the incorrect cases in the onSelect function, which is called far less often.
optionDown	Boolean	When true, the OPTION modifier key was pressed when the item was selected. Read only.

onSelect	Function	The callback function that is called when the user selects the menu or menu item. The function takes no arguments, and returns nothing. It implements the behavior of a menu item.
		
		The callback can check this object's properties to respond to the following modifier keys:
		
		`if (this.ShiftDown)`
		`  // Shift key pressed`
		`if (this.altDown)`
		`  // Alt key pressed`
		`if (this.ctrlDown)`
		`  // Control key pressed`
		`if (this.cmdDown)`
		`  // Command key pressed`
		`if (this.optionDown)`
		`  // Option key pressed`

shiftDown	Boolean	When true, the SHIFT modifier key was pressed when the item was selected. Read only.
text	String	The displayed label text, a localizable string. Read only.
type	String	The type of menu element, one of: **menu**—A menu or submenu **command**—A menu item Read only.

Adobe Bridge menu and command identifiers

These unique identifiers are predefined for Adobe Bridge menus that can be extended.

NOTE: Some menus and commands are dynamically created, and cannot be located with `MenuElement.find()` unless they are visible. You can, however, use the menu and command IDs at any time to extend the menus.

The menu/command-identifier mapping is organized as follows:

- Top-level menu identifiers: Top-level menus in the menu bar, tool bar, context menus and flyout menus.
- Menu bar submenu and command identifiers: Items that appear in menu-bar menus.
- Toolbar menus and commands: Items that appear in menus that drop down from toolbar icons.
- Context and flyout submenus and commands: Items that appear in context and flyout menus.

Top-level menu identifiers

These tables list unique identifiers for the top-level menus in the menu bar, tool bar, context menus and flyout menus.

Menubar menus	**Menu ID**
Bridge (Mac OS only)	(not available)
File	`File`
Edit	`Edit`
View	`View`
Stacks	`submenu/Stack`
Label	`Labels`
Tools	`Tools`
Window	`Window`
Help	`Help`
Toolbar menus	**Menu ID**
Refine	`iddmenu/RefineTask`
Output	`iddmenu/Menu/OutputTask`
(Other toolbar menus not available to scripts)	
Context menus	**Menu ID**
Content pane thumbnail/background context	`Thumbnail`
Favorites thumbnail context	`Bridge/ContextMenu/Favorites`

Preview thumbnail context	`PreviewContextMenu`
Palette context (commands not available to scripts)	`Bridge/BrowserTabDocMenu/BrowserTabDock`
Collections context	`Bridge/ContextMenu/Collection`
(Other context menus not available to scripts)	
Flyout menus	**Menu ID**
Main window in compact mode	`Bridge/CompactFlyoutMenu`
(Flyout menus and commands generally not available to scripts)	

Menu bar submenu and command identifiers

These tables list unique identifiers for submenus and commands in the top-level menus that appear in the menu bar.

When a command opens a submenu, there is a command identifier for the item itself, which can be used to position commands in the parent menu, and a menu identifier for the submenu, as well as identifiers for the individual commands in the submenu.

Bridge menu items (Mac OS only)

Bridge commands	Menu ID
About Bridge	`mondo/command/about`
Preferences	`Prefs`
Quit Bridge	`mondo/command/quit`

File menu submenus and commands

File submenus	Menu ID
Open With >	`submenu/OpenWith`
Open Recent >	`submenu/OpenRecent`
Move to >	`submenu/MoveTo`
Copy to >	`submenu/CopyTo`
Place >	`submenu/Place`
File commands	**Menu ID**
New Window	`mondo/command/new`
New Folder	`NewFolder`
Open	`Open`
Open With >	`OpenWith`

Open With > [*installed application*]	(not available)
Open Recent >	`item/OpenRecent`
Open Recent > *[recent files}*	(not available)
Open Recent > Clear Menu	`ClearOpenRecentList`
Open in Camera Raw...	`OpenInCameraRaw`
Close Window	`mondo/command/close`
Delete/Move to Trash	`MoveToTrash`
Eject	`Eject`
Return to...	`ReturnToApplication`
Reveal in Explorer/Finder	`Reveal`
Reveal in Bridge	`RevealInBridge`
Get Photos From Camera...	(not available)
Move to >	`MoveTo`
Move to > *[recent folders}*	(not available)
Move to > Choose Folder	`MoveToChooseFolder`
Copy to >	`CopyTo`
Copy to > *[recent folders}*	(not available)
Copy to > Choose Folder	`CopyToChooseFolder`
Place >	`item/Place`
Add To Favorites	`AddToFavorites`
File Info...	`FileInfo`
Hide	`HideBridge`
Exit	`mondo/command/quit`

Edit menu submenus and commands

Edit submenus	**Menu ID**
Develop Settings >	`submenu/CameraRaw`
Edit commands	**Menu ID**
Undo Redo	`mondo/command/undoRedo`
Cut	`mondo/command/cut`
Copy	`mondo/command/copy`

Paste	mondo/command/paste
Duplicate	Duplicate
Select All	mondo/command/selectAll
Deselect All	mondo/command/selectNone
Invert Selection	InvertSelection
Find...	Search
Develop Settings >	ApplyCameraRaw
Develop Settings > Camera Raw Defaults	CRDefault
Develop Settings > Previous Conversion	CRPrevious
Develop Settings > Copy Settings	CRCopy
Develop Settings > Paste Settings	CRPaste
Develop Settings > Clear Settings	CRClear
Rotate 180°	Rotate180
Rotate 90° Clockwise	Rotate90CW
Rotate 90° Counterclockwise	Rotate90CCW
Creative Suite Color Settings...	SharedSettings
Camera Raw Preferences...	CrPreferences
Preferences...	Prefs

View menu commands

View submenus	**Menu ID**
Sort >	submenu/Sort
View commands	**Menu ID**
Full Screen Preview	FullScreenPreview
Slideshow	SlideShow
Slideshow Options...	SlideShowOptions
Review Mode	ReviewMode
Compact Mode	ToggleCompactMode
As Thumbnails	View/Thumbnail
As Details	View/Details
As List	(not available)
Show Thumbnail Only	ShowThumbnailOnly

Grid Lock	`GridLock`
Show Reject Files	`ShowReject`
Show Hidden Files	`ShowHidden`
Show Folders	`ShowFolders`
Show Items from Subfolder	`FlatView`
Sort >	`Sort`
Sort > Ascending Order	`Ascending`
Sort > [*items*]	`submenu/Sort`
Sort > Manually	`SortManually`
Refresh	`Refresh`

Stacks menu commands

Stacks submenus	**Menu ID**
Frame Rate >	`submenu/StackFrameRate`
Stacks commands	**Menu ID**
Group as Stack	`StackGroup`
Ungroup from Stack	`StackUngroup`
Open/Close Stack	`ToggleStackStateOpen` `ToggleStackStateClose`
Promote to Top of Stack	`PromoteToTopOfStack`
Expand All Stacks	`ExpandAllStacks`
Collapse All Stacks	`CollapseAllStacks`
Frame Rate >	(not available as command)
Frame Rate > [*rates*]	(not available)

Label menu commands

Label commands	**Menu ID**
Rating	(not available)
No Rating	`NoDot`
Reject	`Reject`
*	`OneDot`
**	`TwoDots`

***	ThreeDots
****	FourDots
*****	FiveDots
Decrease Rating	RemoveDot
Increase Rating	AddDot
Label	(not available)
No Label	NoLabel
Select	Red
Second	Yellow
Approved	Green
Review	Blue
ToDo	Purple

Tools menu commands

Tools submenus	**Menu ID**
Edit Metadata Template >	submenu/EditMetadata
Append Metadata >	submenu/AppendMetadata
Replace Metadata >	submenu/ReplaceMetadata
Cache >	submenu/Cache
Tools commands	**Menu ID**
Batch Rename...	BatchRename
Create Metadata Template...	CreateMetaTemplate
Edit Metadata Template >	item/EditMetadata
[*templates*]	(not available)
Append Metadata >	item/AppendMetadataTemplate
[*templates*]	(not available)
Replace Metadata >	item/ReplaceMetadataTemplate
[*templates*]	(not available)
Cache >	item/Cache
Cache > Build and Export Cache...	BuildSubCaches
Cache > Purge Cache for Folder *[this folder]*	PurgeCache

Window menu commands

Window submenus	Menu ID
Workspace >	`submenu/Workspace`
Window commands	**Menu ID**
New Synchronized Window	`NewSynchronizedWindow`
Workspace >	(not available as command)
Workspace > Reset Workspace	`ResetCurrentWorkspace`
Workspace > New Workspace	`SaveWorkspace`
Workspace > Delete Workspace	`DeleteWorkspace`
Workspace >Reset Standard Workspace	`ResetWorkspace`
Workspace > [*predefined workspaces*]	(not available)
[*individual panels*]	(not available)
Path Bar	`PathBar`
Minimize	`Minimize`
[*current folder*]	(not available)

Help menu commands

Help commands	Menu ID
Adobe Bridge Help...	`mondo/command/help`
Adobe Bridge Support Center...	`SupportCenter`
Manage Extensions...	`ManageExtensions`
Updates...	`Updates`
Adobe Product Improvement Plan...	`AdobePIP`
About Bridge...	`mondo/command/about`

Toolbar menus and commands

These tables list unique identifiers for submenus and commands that appear in the menus that drop down from toolbar icons.

Reveal recent files commands

Reveal-recent command	Menu ID

Clear Recent Files	`ClearOpenRecentList`
Clear Recent Folders	`ClearRecentFolders`

Refine commands

Refine command	Menu ID
Review Mode	`ReviewMode`
Batch Rename	(not available)
File Info	(not available)

Output commands

Submenus/command	Menu ID
Output to Web or PDF	(not available)

Workspace commands

Submenus/command	Menu ID
Reset Workspace	(not available)
New Workspace	(not available)
Delete Workspace	(not available)
Reset Standard Workspaces	(not available)

Search commands

Submenus/command	Menu ID
Bridge Search: Current Folder	(not available)
Clear Recent Search Menu	(not available)

Thumbnail and preview options commands

Submenus/command	Menu ID
Prefer Embedded (Faster)	(not available)
High Quality On Demand	(not available)
Always High Quality	(not available)
Generate 100% Previews	(not available)

Filter by ratings commands

Submenus/command	Menu ID
Clear Filter	(not available
Show Rejected Items Only	`RejectStars`
Show Unrated Items Only	`NoStars`
Show *n* Stars	(not available)
Show Labeled Items Only	`ShowLabeled`
Show Unlabeled Items Only	`ShowUnlabeled`

Sort commands

Submenus/command	Menu ID
By [items]	(not available)
Manually	(not available)

Open recent file commands

Submenus/command	Menu ID
Clear Menu	`ClearOpenRecentList`

Context and flyout submenus and commands

These tables list unique identifiers for submenus and commands that appear in context menus and flyout menus.

Thumbnail context menu submenus

Thumbnail context in Folders submenus	Menu ID
Sort >	`submenu/Sort`
Thumbnail context in Content pane: submenus	**Menu ID**
Open With >	`submenu/OpenWith`
Move to >	`submenu/MoveTo`
Copy to >	`submenu/CopyTo`
Label >	`submenu/Label`

Sort >	`submenu/Sort`
Stack >	`Stacks`

Thumbnail context menu commands

Thumbnail context in Favorites commands	**Menu ID**
Remove from Favorites	`Bridge/ContextMenu/Keyword/Delete`
Reveal in Explorer/Finder	`Bridge/ContextMenu/Keyword/Reveal`
Thumbnail context in Folders commands	**Menu ID**
Open	`Thumbnail/Open`
Open in New Window	`Bridge/ContextMenu/Folders/NewWindow`
Cut	`Bridge/ContextMenu/Folders/Cut`
Copy	`Bridge/ContextMenu/Folders/Copy`
New Folder	`Bridge/ContextMenu/Folders/NewFolder`
Sort >	(not available as command)
Sort > [commands]	(not available)
Reveal in Explorer/Finder	`Bridge/ContextMenu/Folders/Reveal`
Add to Favorites	`Bridge/ContextMenu/Folders/AddToFavorites`
Remove from Favorites	`Bridge/ContextMenu/Folders/RemoveFromFavorites`
Delete	`Bridge/ContextMenu/Folders/Delete`
Rename	`Bridge/ContextMenu/Folders/Rename`

Thumbnail context in Content pane: commands	**Menu ID**
Open	`Thumbnail/Open`
(*files*) Open With >	(not available as command)
Open With > [*installed application*]	(not available)
(*folders*) Open in New Window	`Thumbnail/NewWindow`
Purge Cache for Selection	`PurgeCacheForSelected`
Cut	`Thumbnail/Cut`
Copy	`Thumbnail/Copy`
Paste	`Thumbnail/Paste`

Delete	`Thumbnail/Delete`
Move to >	(not available as command)
Move to > *[recent folders}*	(not available)
Move to > Choose Folder	`MoveToChooseFolder`
Copy to >	(not available as command)
Copy to > *[recent folders}*	(not available)
Copy to > Choose Folder	`CopyToChooseFolder`
(*JavaScript files*) Install	`Thumbnail/InstallScript`
Reveal in Explorer/Finder	`Thumbnail/RevealLocation`
Reveal in Bridge	`Thumbnail/RevealInBridge`
Add to Favorites Remove from Favorites	`Thumbnail/ToggleAsFavorite`
Rename	`Thumbnail/Rename`
(*files*) Batch Rename...	`Thumbnail/BatchRename`
(*image files*) Generate High Quality Thumbnail	`MakeHighQualityThumbnail`
(*image files*) Generate Quick Thumbnail	`MakeQuickThumbnail`
(*files*) Lock Item	`Thumbnail/LockFile`
(*files*) Unlock Item	`Thumbnail/UnlockFile`
(*image files*) File Info...	`Thumbnail/FileInfo`
Label >	`submenu/Label`
Label > No Label	`NoLabel`
Label > Select	`Red`
Label > Second	`Yellow`
Label > Approved	`Green`
Label > Review	`Blue`
Label > ToDo	`Purple`
New Folder	`Thumbnail/NewFolder`
Sort >	(not available as command)
Sort > Ascending Order	`Ascending`
Sort > By Filename	`SortFileName`
Sort > By Document Type	`SortFileType`
Sort > By Date created	`SortDateCreated`

Sort > By Date file modified	`SortDateModified`
Sort > By File size	`SortFileSize`
Sort > By Dimensions	`SortDimensions`
Sort > By Resolution	`SortResolution`
Sort > By Color profile	`SortColorProfile`
Sort > By Copyright	`SortCopyright`
Sort > By Keywords	`SortKeywords`
Sort > By Label	`SortByLabel`
Sort > By Rating	`SortRating`
Sort > Manually	`SortManually`
(*multi-select*) Stack >	`Stacks`
Stack > Frame Rate >	`submenu/StackFrameRate`
Stack > Frame Rate > [rates]	(not available)
Stack > Enable Onion Skin	`StackEnableOnionSkin`
Stack > Disable Onion Skin	`StackDisableOnionSkin`
Stack > Ungroup from Stack	`Stacks/Ungroup`
Stack > Group as Stack	`Stacks/Group`
Remove From Collection	`Thumbnail/RemoveFromArbitraryCollection`

Thumbnail context menu in Preview pane: additional commands for image files	**Menu ID**
No Rating	`NoDot`
Reject	`Reject`
*	`OneDot`
**	`TwoDots`
***	`ThreeDots`
****	`FourDots`
*****	`FiveDots`
No Label	`NoLabel`
Select	`Red`
Second	`Yellow`
Approved	`Green`

Review	`Blue`
ToDo	`Purple`
Rotate 90° Clockwise	`Rotate90CW`
Rotate 90° Counterclockwise	`Rotate90CCW`
Open	`Open`

Collections context menu commands

Collections context menu commands	Menu ID
New Smart Collection	`Bridge/ContextMenu/Collection/NewSmartKey`
New Collection	`Bridge/ContextMenu/Collection/NewArbitraryKey`
Rename	`Bridge/ContextMenu/Collection/Rename`
Delete	`Bridge/ContextMenu/Collection/Delete`
Add to Favorites	`Bridge/ContextMenu/Collection/Edit`

Keywords context and flyout menu commands

Keywords context menu commands	Menu ID
New Keyword	`Bridge/ContextMenu/Keyword/NewKey`
New Sub Keyword	`Bridge/ContextMenu/Keyword/NewSubKey`
Rename	`Bridge/ContextMenu/Keyword/Rename`
Delete	`Bridge/ContextMenu/Keyword/DeleteNode`
Exclude	`Bridge/ContextMenu/Keyword/Exclude`
Include	`Bridge/ContextMenu/Keyword/Include`
Expand All	`Bridge/ContextMenu/Keyword/ExpandNode`
Collapse All	`Bridge/ContextMenu/Keyword/CollapseNode`
Find...	`Bridge/ContextMenu/Keyword/Search`
Keywords flyout menu: additional commands	**Menu ID**
Remove Keywords	`Bridge/ContextMenu/Keyword/Delete`
Import... Clear and Import... Export...	(not available)

Keywords search menu commands	Menu ID
Contains Equals Starts With	(not available)

Compact-mode flyout menu commands

Submenus	Menu ID
Label >	`submenu/Label`
View >	(not available)

Commands	Menu ID
New Window	`mondo/command/new`
New Folder	`NewFolder`
Open	`Open`
Open With	(not available)
Open in Camera Raw	`OpenInCameraRaw`
Reveal in Explorer/Finder	`Reveal`
Label >	`submenu/Label`
Label > No Label	`NoLabel`
Label > Select	`Red`
Label > Second	`Yellow`
Label > Approved	`Green`
Label > Review	`Blue`
Label > ToDo	`Purple`
Compact Window Always On Top	(not available)
View >	(not available)
View > Full Mode	(not available)
View > Sort >	(not available)
View > Sort > Ascending Order	`Ascending`
View > Sort > By Filename	`SortFileName`
View > Sort > By Document Type	`SortFileType`
View > Sort > By Date created	`SortDateCreated`

View > Sort > By Date file modified	`SortDateModified`
View > Sort > By File size	`SortFileSize`
View > Sort > By Dimensions	`SortDimensions`
View > Sort > By Resolution	`SortResolution`
View > Sort > By Color profile	`SortColorProfile`
View > Sort > By Copyright	`SortCopyright`
View > Sort > By Keywords	`SortKeywords`
View > Sort > By Label	`SortByLabel`
View > Sort > By Rating	`SortRating`
View > Sort > Manually	`SortManually`
View > Refresh	`Refresh`
View > Show Thumbnails Only	`ShowThumbnailsOnly`
View > Grid Lock	`GridLock`
View > Show Hidden Files	`ShowHidden`
Path Bar	`PathBar`
Exit Bridge	`mondo/command/quit`

Metadata Object

Allows you to access the Extensible Metadata Platform (XMP) metadata associated with the file node of a Thumbnail Object. This is embedded metadata associated with the file, such a copyright owner, author, or camera settings.

Metadata is organized into schemas that group related types of metadata; for example the XMP Rights Management Schema groups metadata associated with ownership and rights, such as copyright and owner. The metadata properties found in a specific schema are accessed via the *namespace* of the schema and the *property name* of the metadata item. For example, the namespace of the XMP Rights Management Schema is `http://www.adobe.com/xap/1.0/rights`, and the copyright property name is `Copyright`.

For more information about XMP metadata, see the XMP Specification at Adobe Developer Center, http://www.adobe.com/devnet/.

Access the `Metadata` object for a file-type thumbnail through the Thumbnail Object's metadata property:

```
var t = new Thumbnail (File ("/C/mydir/myfile"));
var mdata = t.metadata
```

When a script needs to access the metadata through the `Thumbnail` object, it is important to make sure that the returned object contains the most current data. To ensure this, your script should set `app.synchronousMode` to `true` before attempting to retrieve or set values through `Thumbnail.metadata`, or else use `Thumbnail.synchronousMetadata`. Keep in mind, however, that metadata access is a time-intensive operation. Do not do it unnecessarily, or as part of operations that occur very frequently, such as a `MenuItem.onDisplay` callback function.

The `Metadata` object does not support multi-valued properties.

Example code

The sample code distributed with the Adobe Bridge SDK includes these code examples that specifically demonstrate the use of this object:

Thumbnail metadata access	
`SnpInspectMetadata.jsx`	Shows how to acquire metadata.
`SnpModifyMetadata.jsx`	Shows how to alter metadata on a selected file.

Metadata properties

namespace	String	The current XMP namespace, used to search for XMP properties. Default is the root namespace. Read/write. Assigning a new namespace creates that namespace in the XMP metadata. To access values in a specific schema, the namespace for that schema must be set before referencing the properties in the schema.

xmpPropertyName	String	Get or set a simple XMP property value for a thumbnail by specifying it as a property of that thumbnail's `metadata` object. Properties are accessed in the current namespace. Read/write.
		
		New simple metadata properties are created and added to the current namespace when a script references a new property name. You can add properties only to currently defined namespaces, not to the root namespace. Property names are case sensitive.
		
		If no metadata is defined for a thumbnail, and you attempt to access a property through the `Thumbnail.metadata` property, the value `undefined` is returned. Note that this differs from the behavior in Adobe Bridge CS2, where an exception was thrown in this case.
		
		NOTE: For metadata properties that are known date formats, the corresponding `Metadata` object property contains an ISO-8601 date string. These include:
		
		`xmp/DateCreated`
		`xmp/DateModified`
		`xmp/MetadataDate`
		`photoshop/OriginDateCreated`
		`tiff/DateTime`
		`exif/DateTimeOriginal`
		`exif/ExifDateTimeDigitized`
		`exif/GPS_TimeStamp`
		`exif/GPSDateStamp`

Metadata functions

applyMetadataTemplate()
metadataObj.applyMetadataTemplate (*templateName, modType*)

Adds metadata properties to this object that were saved to an XMP template from the FileInfo dialog.

Returns `undefined`.

templateName — String, The name of the XMP template. Templates are stored for each user in:

- (Windows) `%APPDATA%/Adobe/XMP/Metadata Templates/`
- (Mac OS) `/Users/`*username*`/Library/Application Support/Adobe/XMP/Metadata Templates/`

modType — The modification type, one of:

append—Adds to the metadata any property that is in the template but not in the source. If a property in the template already exists in the source, its value is not changed, unless it is an array. For an array, adds members that are in the template but not in the source. If an array member already exists in the source, the value is not changed.

replace—Adds to the metadata all properties and values that are in the template. If a property in the template already exists in the source, its value is changed to the template value.

read()
metadataObj.read (*namespace, property*)

Retrieves and returns the string value of a metadata property in the specified namespace.

Returns the string value, or an empty string if the specified property does not exist.

namespace — String. The XMP namespace.

property — String. The property name. To access a multivalue (complex) property, use an XPath to the individual value. For example:

```
var text = md.read("http://purl.org/dc/elements/1.1/",
"rights/*[1]");
```

serialize()
metadataObj.serialize ()

Serializes the XMP packet into a string.

Returns the string containing the serialized metadata.

NavBar Object

Represents a configurable navigation bar, one of which can be displayed at the top of the browser window (below the application navigation bar), and one at the bottom (above the status bar). You do not create new `NavBar` objects. Instead, you access the existing objects through the Document Object's properties:

```
topbarF = app.document.navbars.filesystem.top
btmbarF = app.document.navbars.filesystem.bottom
```

The bars in `navbars.filesystem` can be shown when the Content pane displays files and folders.

The navigation bars are hidden by default. You can show and hide them by setting the `NavBar` object's `visible` property.

Your script can configure a navigation bar to contain user-interface controls such as push buttons, radio buttons, edit fields, list boxes, and so on. The `NavBar` objects are initially empty. You can add ScriptUI controls.

NavBar properties

`height`	Number	The height of the navigation bar in pixels. Default is 40. Read/write.
`id`	String	A unique identifier that can be used by a node-handling extension to identify a bar that it manages during configuration operations. The extension is responsible for supplying and interpreting this value. See ExtensionHandler Object. Default values (used by the default node-handler) are `topFilesystemNavbar`, `bottomFilesystemNavbar`, `topWebNavbar`, and `bottomWebNavbar`.

`jsFuncs`	Object	DEPRECATED. Do not use.
		
		A JavaScript object that defines a set of callback functions that access the Adobe Bridge DOM, but can be called from within an HTML page displayed in this navigation bar. Used only when `type` is `html`. Read/write.
		
		Each property in the object is a callback function name, and the value is the function declaration:
		
		`{`
		`fnName1: function( [args] ) { fn1_definition },`
		`fnName2: function( [args] ) { fn2_definition }`
		`}`
		
		The HTML page displayed by this bar can access the Adobe Bridge DOM by invoking one of these callbacks, using the JavaScript function `call`. For example, suppose `jsFuncs` has the value:
		
		`{myFn: function(x) { return x > app.document.topNavBar.height }}`
		
		A script on the displayed HTML page can invoke this function as follows:
		
		`var toobig = call("myFn", 55);`

onResize	Function	For a bar that displays ScriptUI, you can provide this callback function to resize the component elements automatically when the bar is resized by the user. For details, see the *JavaScript Tools Guide*.
type	String	The type of user-interface controls displayed in the navigation bar. Read/write. One of: **scriptui**: Display the ScriptUI controls added with this object's add() method.
visible	Boolean	Controls whether the bar is shown. If `true`, the navigation bar is visible. Default is `false`. Read/write.

NavBar functions

add()

```
navBarObj.add (type
[, bounds, text,
 { creation_props> } ]);
```

Creates and returns a new ScriptUI control or container object and adds it to the children of this navigation bar.

Returns the new object, or `null` if unable to create the object.

type	The control type. See the *JavaScript Tools Guide*.
bounds	Optional. A bounds specification that describes the size and position of the new control or container, relative to its parent. See `Bounds` object for specification formats. If supplied, the method creates a new `Bounds` object which is assigned to the new control object's `bounds` property.
text	Optional. A string containing the initial text to be displayed in the control as the title, label, or contents, depending on the control type. If supplied, this value is assigned to the new object's `text` property.
creation_props	Optional. The properties of this JavaScript object specify creation parameters, which are specific to each object type. See the *JavaScript Tools Guide*.

execJS() *navBarObj*.execJS (*script*)	DEPRECATED. Do not use. Executes a JavaScript function that is defined within the HTML page displayed in the navigation bar when `type` is `html`. If the page that defines the function is not currently displayed, causes a run-time error. **NOTE:** Do not call this method from a `NavBar` callback function defined in `jsFuncs`. This attempts to re-enter the JavaScript engine, which is not allowed, and causes Adobe Bridge to hang. A callback can, instead, schedule a task using `app.scheduleTask()`, and call `execJS` from the function associated with the task. Returns the result of the executed JavaScript function, which must be a Boolean, Number, or String, or `null`.
script	A string containing a script to execute. This typically contains the name and arguments of the JavaScript function to execute, but can have multiple statements, including variable declarations, assignments and so on.
print() *navBarObj*.print ()	DEPRECATED. Do not use. Prints the HTML page displayed in the navigation bar when `type` is `html`. Does nothing if the HTML is not yet loaded when the call is made, or if `type` is `scriptui`. Returns `true` on success.

Panelette Base Class

A base class for the subpanel types that can be contained in the panelettes property of a InspectorPanel Object:

- IconListPanelette Object
- TextPanelette Object
- ThumbnailPanelette Object

The base class is not instantiable. Use the `new` operator with the subclasses to create subpanel objects.

Panelette class properties

name	String	The unique, identifying name of this subpanel. Read/write.
titleMarkup	String	Optional. Localizable text shown in the subpanel header bar. Can include Panelette markup elements. If not supplied, the `name` string appears as the header. Read/write.

Panelette markup elements

You can specify dynamic or calculated string content to be displayed in the subpanels, or in the title string of the panel (`InspectorPanel.displayTitle`) or subpanel tabs (`Panelette.titleMarkup`). To specify these special string values, you use *panelette markup elements*.

Markup elements are enclosed by double brackets. They can indicate:

- **Dynamic text**: Dynamic text values are retrieved from the thumbnail's associated node data. To insert a dynamic value retrieved from node data, use a markup element that identifies the `ExtensionHandler`, `Infoset`, and member element:

```
[[extensionName.infosetName.elementName]]
```

- **JavaScript**: Values can be retrieved or calculated at display time using JavaScript. To specify a dynamically calculated value, embed JavaScript within the content string, using this tag:

```
[[javascript:code]]
```

A function in this context is not allowed to block; if it takes more than 10 milliseconds, the display string is converted to an error string.

Within the context of the markup tag, you can refer to the currently selected `Thumbnail` object using a special variable `inspectorThumbnail`. This is useful for accessing embedded file metadata. For example:

```
[[javascript:"Name: " + inspectorThumbnail.name]]
[[javascript:"Author: " + inspectorThumbnail.metadata.author]]
```

For additional examples, see the *Adobe Bridge JavaScript Guide* and SDK code-snippet examples.

PDFOutput Object

PDFOutput object provides capability to generate a PDF using the Bridge's native PDF Output Module. PDFOutput object takes a template name and a list of files to be added in the PDF. The PDF file generated with the provided template and assets can be exported at a specified location.

PDFOutput Functions

resetDocument() `pdfOutPutObj.resetDocument()`	Resets the Output preview. This can be used to clear the list of files added for PDF generation. Returns undefined.

setPDFTemplate()
`pdfOutPutObj.setPDFTemplate(templateName)`
 	
`templateName`
	Sets template for the pdf file.
	
	Returns undefined.
	
	Name of the PDF template.

addToDocument() `pdfOutPutObj.addToDocument()`	Adds the provided files to be generated as the PDF file. Returns undefined.

exportToPDF()
`pdfOutPutObj.exportToPDF(filePath, fileName)`
 	
`filePath`
 	
`fileName`
	Exports the pdf file at provided path.
	
	Returns undefined.
	
	Path of the folder for exporting of the PDF file.
	
	Name of the exported PDF file with extension.

Preferences Object

Allows access to the Adobe Bridge application preferences, as viewed in and controlled by Preferences dialog (invoked by the **Edit > Preferences** command).

- Some existing preferences can be set or read by setting or retrieving the associated property value. Not all existing preferences are available in the scripting environment. Those that are available are listed below. Preference values do not take effect until the Adobe Bridge application is restarted.
- You can set certain preference values for the current session only. That is, the changes take effect immediately, but do not persist across sessions. The next time the Adobe Bridge application is restarted, the global preference value is used.
- A script can create a new preference by simply referencing a new property name in this object. New preferences must be of the type String, Number, or Boolean. Composite types (such as Rect and Point) are retrieved as String objects.

Access the `Preferences` object through the App Object's preferences property:

```
var prefs = app.preferences;
```

Preferences properties

The following current-view properties allow you to set these styles for a specific Content pane view. They do not change the related global preference, and the changes do not persist beyond the current view:

`extraMetadata`	Array of String	An array of up to four values, where each value identifies a metadata property to be displayed beneath a thumbnail icon. Read/write.
		
		Setting this property is the same as setting the preferences associated with the **Additional Thumbnail Metatdata** drop-down lists and checkboxes in the Thumbnail page of the Preferences dialog, except that the setting does not persist beyond the current view.
		
		The first value in the array sets the first line of additional metadata, the second value sets the second line, and so on. Allowed values are:
		
		`author`
		`bit-depth`
		`color-mode`
		`color-profile`
		`copyright`
		`date-created`
		`date-modified`
		`description`
		`dimensions`
		`document-creator`
		`document-kind`
		`exposure`
		`file-size`
		`focal-length`
		`keywords`
		`label`
		`opening-application`

		An array value of `undefined` turns off the display of metadata for that line.
showName	Boolean	When `true`, the names of thumbnails are displayed beneath the icon in this view. When `false`, they are not. Read/write. (This is overridden by the document's showThumbnailName value.)

The following properties allow access to existing application preferences. Preference values do not take effect until the Adobe Bridge application is restarted:

AutoExportCaches	Boolean	In the Cache page of the Preferences dialog, the preference associated with **Cache** choices, `true` when **Automatically Export Caches to Folders When Possible** is selected. Default `false`. Read/write.
CacheDirectory	String or Folder	In the Cache page of the Preferences dialog, the preference associated with the **Cache Location**. The location of the centralized cache. A folder path, specified as a string or ExtendScript `Folder` object. Read/write.
ColorTheme	Number	In the Interface page of the **Preferences** dialog, the preference associated with the **ColorTheme** selection. Read/write. One of: 1 2 3 4
Favorites	Array of String	In the General page of the Preferences dialog, the preference associated with **Favorite Items** choices. A collection of Bridge URI strings for checked nodes, which are displayed in the Favorites palette.
FavoritesDisplayNames	Array of String	A collection of localized display names for the nodes displayed in the Favorites palette, where each member corresponds to URI member of the Favorites array.
FileSize	Number	In the Thumbnails page of the Preferences dialog, the preference associated with **Do not process files larger than: *nnn* MB**. Default `1000`. Read/write.
HideEmptyFields	Boolean	In the Metadata page of the Preferences dialog, the preference associated with the **Hide Empty Fields** checkbox, `true` when checked. Default `true`. Read/write.
HideUnknownOpeners	Boolean	In the File Type Associations page of the Preferences dialog, the preference associated with the **Hide Undefined File Associations** checkbox, `true` when checked. Default `false`. Read/write.

`ImageBackdrop`	Number	In the General page of the Preferences dialog, the preference associated with the **Image Backdrop** slide bar. Read/write. Sets background of the Content pane.The background color is set in the range of 1 - 10, where 1 is black, and 10 is white. Default value is 2.
`Keyboard`	String	In the Advanced page of the Preferences dialog, the preference associated with **Keyboard**. Read/write, takes effect on restart.

`Label1`
`Label2`
`Label3`
`Label4`
`Label5`
	String	In the Labels page of the Preferences dialog, the preferences associated with the label colors and their keyboard shortcuts. These preferences control the choices that appear in the **Label** menu in the menu bar and in the right-click context menu for image thumbnails. Read/write.
		
		The preference value is any string. For example, if you associate the red flag with the string `Urgent`, the string `Urgent` appears in **Label** menu (in place of the default string `Red`), in the tooltip for the labeled thumbnail, and in a labeled thumbnail's label value. The thumbnail is displayed with a red highlight frame.
		
		The labeling feature is only available for those thumbnails associated with image files.
		
		By default, no labels are set. Labels can be set interactively by choosing from the **Label** menu or programmatically by setting the `Thumbnail.label` value to any string. If that string is not one of the preferences, it is associated with a white highlight frame.

`LabelCtrlKey`	Boolean	In the Labels page of the Preferences dialog, the preference associated with the **Require the Control Key to Apply Labels and Ratings** checkbox, `true` when checked. Default `true`. Read/write, takes effect on restart.
`Language`	String	In the Advanced page of the Preferences dialog, the preference associated with **Language**. Read/write.
`MRUCount`	Number	In the General page of the Preferences dialog, the preference associated with **Number of Recent Items to Display**. Read/write.
`MRUFolders`	Array of String	The set of absolute path strings for recently-visited folders, displayed when the MRUCount is greater than 0. Read/write.
`PermittedStartupScripts`	Array of String	In the Startup Scripts the Preferences dialog, the script names associated with selected script checkboxes. This is the set of scripts that load automatically on startup. Read/write.

`PreferencePanel`	Number	The panel to be displayed when the Preferences dialog is invoked. A zero-based index of the panel, in the order in which they appear in the dialog. Read/write.
`ShowCameraRawInterface`	Boolean	In the General page of the Preferences dialog, the preference associated with the **Double-Click Edits Camera Raw Settings in Bridge** checkbox, `true` when checked. Default `false`. Read/write.
`ShowName`	Boolean	When `true`, the names of thumbnails are displayed beneath the icon. When `false`, they are not. Read/write.
`ShowPlacard`	Boolean	In the Metadata page of the Preferences dialog, the preference associated with the **Show Metadata Placard** checkbox, `true` when checked. Default is `true`. Read/write.
`ShowTooltips`	Boolean	In the Thumbnails page of the Preferences dialog, the preference associated with **Show Tooltips**, `true` when checked. Default is `false`. Read/write.
`StackFrameRate`	Number	In the Playback page of the Preferences dialog, the preference associated with **Stack Playback Framerate**. Read/write. One of: `2 4 6 10 12 15 24 25 30 50 60`
`StartupScriptsShouldLoad`	Boolean	In the Startup Scripts the Preferences dialog, setting to `true` is the equivalent of clicking **Enable All**, setting to `false` is the equivalent of clicking **Disable All**. Read/write.
`ThumbnailQuality`	String	Options for thumbnail quality and preview generation. Read/write. One of: `draft`–Prefer Embedded (Faster) `proof`—Always High Quality `drafttoproof`—High Quality On Demand

UseSoftwareRendering	Boolean	In the Advanced page of the Preferences dialog, the preference associated with the **Use Software Rendering** checkbox, `true` when checked. Read/write, takes effect on restart. When `true`, hardware acceleration is disabled for the Preview panel and slideshows. Default is `false`.

anyPropertyName	Number, String, or Boolean	A script-defined preference. Read/write.
		
		This example creates a new preference named `mypref` by assigning a value to a property of that name, then accesses the value by reading the property.
		
		`app.preferences.mypref = "sample value";`
		`Window.alert("New preference mypref = " +`
		`app.preferences.mypref);`
		To add your script-defined preference to the Preferences dialog, use the PreferencesDialog Object's addPanel() function.
		
		NOTE: The script must implement default values and initialization of any private setting stored in the Adobe Bridge preferences.

Preferences functions

clear() *prefObj*.clear ([*name*[, *name2*...]])	Removes script-created keys and values from the Adobe Bridge preferences, or resets preferences. • If one or more preference names is passed, each is removed. If you try to access the property for a preference that has been removed, the property returns `undefined`. • If no preference names are passed, removes all script-defined preferences, and resets all Adobe Bridge application preferences to their default values. Returns `undefined`.
name	Optional. One or more names of preferences to remove.
resetFileAssociations() *prefObj*.resetFileAssociations ()	Resets file type associations to their default values. Corresponds to the **Reset to Default Associations** button in the File Type Associations page of the Preferences dialog.
resetWarningDialogs() *prefObj*.resetWarningDialogs ()	Resets "Do not show again" settings to `false` for all warning dialogs. Corresponds to the **Reset** button in the General page of the Preferences dialog.

PreferencesDialog Object

Provides access to the Adobe Bridge Preferences dialog, allowing you to add a panel to the dialog with your own ScriptUI controls that access and set any application preferences that you have defined by adding properties to the Preferences Object.

You can only access this object as the target of an event. The object is returned in the object property of an Event Object that results from an event in a Preferences dialog. See PreferencesDialog events.

The Preferences dialog is modal, which means that no other Adobe Bridge events can occur until the user dismisses it with the **OK** or **Cancel** button, or closes it with the window-frame icon.

- For the **OK** button, the dialog generates an `ok` event. Your handler can collect the values from the controls in your panel, and modify the property values in the `Preferences` object accordingly.
- For the **Cancel** button, the dialog generates a `cancel` event, and for the window-close gesture, it generates a `destroy` event. Your handler can, for example, clean up structures you created for the window.

The class defines no properties.

PreferencesDialog functions

`addPanel()` *`prefObj`*`.addPanel (`*`name`*`)`	Creates and returns a ScriptUI `Window` object to be used as a new page in the Preferences dialog. You can add ScriptUI controls to the window to allow users to access and set preferences that you provide. Returns the new `Window` object.

`name` The name of the new page, used as the title of the new `Window` object.

Example

This example adds a page to the Preferences dialog that contains a single checkbox, which controls the boolean preference named `myPref`.

```
function doPrefs(dialog) {
    var panel = dialog.addPanel("My Preferences");
    var aBox = panel.add( 'checkbox', [50, 50, 200, 100],  "My Pref",
        { alignment:['center','top'] } );
    aBox.onClick = function() { app.preferences.myPref = aBox.value; };
}

var myHandler = function(event) {
    if (event.type == "create" && event.location == "prefs") {
        doPrefs(event.object);
    }
    return { handled: false };
};

app.eventHandlers.push( { handler: myHandler } );
```

close() *prefObj*.close (*isOK*)	Closes the Preferences dialog. Returns `undefined`.
isOK	Pass `true` to simulate the user clicking **OK** to close the dialog, `false` for **Cancel**.

QuickSearch Object

QuickSearch object provides capability to the quick search feature in Bridge through Bridge SDK. Using QuickSearch, assets can be searched quickly from different locations such as current folder, system, or web (Adobe Stock search). Search strings can be retrieved or cleared. Search Results appear in the Content panel, or the default web browser if the selected search option is Adobe Stock search.

There are four options available for search as provided in the quick search bar on the upper-right corner of the Bridge window.

Quick Search Option	String Equivalent
Bridge Search: Current folder	"bridge"
Search Adobe Stock	"Stock_Search"
Spotlight (Mac)/ Desktop Search (Windows): Current folder	"os_folder"
Spotlight (Mac)/ Desktop Search (Windows): Computer	"os_computer"

QuickSearch Functions

`clearSearchString()` `quickSearchObj.clearSearchString()`	Clears the current string from Quick Search bar. Returns undefined.
`getSearchString ()` `quickSearchObj.getSearchString ()`	Gets the current string from Quick Search bar. Returns searched term as string.

`searchString()`
`quickSearchObj.searchString(serachString)`
 	
`searchString`
	Searches the provided string using the selected Quick search option. The search results are displayed in the content panel/browser.
	
	Returns undefined.
	
	The string to be searched.

`getSearchMethod()` `quickSearchObj.getSearchMethod()`	Gets currently selected QuickSearch option. Returns string equivalent of quick search option selected. Refer to previous table for values.

`setSearchMethod()`
`quickSearchObj.setSearchMethod(quickSearchOption)`
 	
`quickSearchOption`
	Sets quick Search Option. Allowed values are:
	`bridge`
	`Stock_Search`
	`os_folder`
	`os_computer`
	Returns undefined.
	
	Search option string for setting the scope of search.

TabbedPalette Object

Allows a script to define and add a tabbed palette to a browser window. A script-defined palette is displayed in addition to the default palettes such as Favorites, Folders, Preview, Filter, Keywords, and Metadata. A script-defined palette can display a user interface defined in ScriptUI, or it can display HTML.

You can add a palette to an existing browser window at any time (as long as the identifier is unique), and you can use the create document event to add your palette to new browser windows on creation.

The name of a script-defined palette is automatically added to all relevant menus. You can specify where the palette goes, or move it programmatically. When it is shown, however, it can be dragged and dropped like the default palettes, and scripts cannot query the current position.

You can show and hide individual palettes using this object's properties. A list of all defined palettes for a browser, including default palettes, is available in `app.document.palettes`.

If a script-defined tabbed palette is visible when the user or a script creates a workspace, the workspace references that palette by its unique identifier. If a workspace references a script-defined tabbed palette, the palette must be created before the workspace is applied. Otherwise, the palette does not appear.

TabbedPalette constructor

To create a new object, use the `new` operator:

```
new TabbedPalette (document, title, id, type, *paletteColumn, *paletteRow)

new TabbedPalette (document, title, id, type, *url, *paletteColumn,
                   *paletteRow)
```

document	The browser window to which to add the palette.
title	The localizable title string that appears in the tab.
id	The unique identifying string for the palette.
type	The type of palette. One of: **`script`**—A ScriptUI window **`web`**—A browser view
url	Optional. When *type* is `web`, the web page URL to display. Default is the empty string, in which case the displayed palette is blank until the url property is set.
paletteColumn	Optional. The horizontal location of the palette in the browser. A string or number, one of: **`left`**, 0—The leftmost column (the default) **`center`**, 1—The middle column **`right`**, 2—The rightmost column

paletteRow	Optional. The vertical location of the palette in the browser. Can be a number, or one of these strings: **top**—The topmost row (index 0, the default) **middle**—The middle row (or close to the middle, if there are an even number) **bottom**—The bottommost row If a number, it is the 0-based index of the row, where 0 is the topmost row. If the index is out of range, the palette is placed in the closest existing row. The number of rows can vary according to the current workspace configuration. This function cannot create new rows.

Example:

```
#target bridge
// create browser palette
var webPalette = new TabbedPalette(app.document, "myWebPalette",
       "myWebID", "web", "http://www.adobe.com");

// create ScriptUI palette
var scriptPalette = new TabbedPalette(app.document, "myScriptPalette",
       "myScripID", "script");
scriptPalette.content.add('statictext', [15,15,105,35],
    'Display this text in my tab.');
```

TabbedPalette properties

content	Object	When `type` is `script`, the ScriptUI `Group` object to display. Use this object's `add()` method to add UI elements to the palette. You can provide an `onResize` callback method for the `Group` object, which will be used to resize the contained elements when the user resizes the palette. For details of these methods and ScriptUI usage, see the *JavaScript Tools Guide*.

id	String	A non-localized unique identifier for the palette. The identifiers for the built-in palettes are:
		`favoritesTab`
		`foldersTab`
		`filterTab`
		`metadataTab`
		`keywordsTab`
		`contentTab`
		`cinemaPreviewTab`
		`inspectorTab`

title	String	The localized title string to display in the palette's tab header. The string can include values derived dynamically at display time, using Panelette markup elements.

type	String	The type of palette. One of: **script**—A ScriptUI window **web**—A browser view
url	String	When `type` is `web`, the path to the page to display.
visible	Boolean	When `true`, this palette is visible, when `false` it is hidden. Read/write. **Note**: **Setting the visible parameter to false will destroy the panel created in UI.**

TabbedPalette object methods

setLocation() *tabObj*.setLocation (*paletteColumn*[, *paletteRow*])	Moves this palette to a specific docking location in the browser. Returns `undefined`.
paletteColumn	The horizontal location of the palette in the browser. A string, one of: **left**—The leftmost column **center**—The middle column **right**—The rightmost column
paletteRow	Optional. The vertical location of the palette in the browser. The number of rows can vary according to the current workspace configuration. This function cannot create new rows. A string or number, one of: **top**—The topmost row (the default) **middle**—The middle row (or close to the middle, if there are an even number) **bottom**—The bottommost row Can be a number, the 0-based index of the row, where 0 is the topmost row. If the index is out of range, the palette is placed in the closest existing row.
remove() *tabObj*.remove ( )	Removes this palette from the list of available palettes and destroys it. Returns `undefined`.

TextPanelette Object

An instantiable subclass of the Panelette Base Class, representing a member subpanel of an InspectorPanel Object that displays textual information about a set of thumbnails. It differs from the ThumbnailPanelette Object in that it does not display the thumbnail icon, only the related text.

The text can be static, or can be obtained dynamically from the associated thumbnail at display time. See Panelette markup elements.

TextPanelette constructor

To create a new object, use the `new` operator:

```
new TextPanelette(name, titleMarkup, thumbnails, keyValuePairs);
```

Parameters set the corresponding properties. The `name` and `titleMarkup` properties are inherited from the Panelette Base Class.

TextPanelette properties

`keyValuePairs`	Array of 2-element Array	A set of two-element arrays in the format `[key, value]`. The array corresponds to the `thumbnails` array, each pair describing the text for the corresponding thumbnail. The key is shown on the left of each field in bold, and the value on the right in plain text. The fields contains string literals combined with Panelette markup elements, which specify the text to be displayed.
`thumbnails`	Array of Thumbnail or String	An array of Thumbnail Objects or node URI strings for which to display descriptive text; or the special markup `[[this]]` to indicate the currently selected thumbnail in the Content pane.

Thumbnail Object

Represents a reference to a node in the browser navigation hierarchy. Thumbnail objects can represent:

- Files and folders in the local file system.
- URLs
- Navigation nodes of types defined by an ExtensionHandler Object.

A thumbnail's applicable node handler determines how nodes are displayed when that thumbnail is selected. The Content pane can show thumbnail icons or a local or remote web page.

CAUTION: When a script accesses the properties of a `Thumbnail` object, some properties of the object may not be immediately available. To ensure the object contains current data, set `app.synchronousMode` to `true` before accessing properties.

Thumbnail object constructor

Adobe Bridge automatically creates `Thumbnail` objects for files and folders in the local file system and for the default and interactively added contents of the Favorites palette.

To create a `Thumbnail` object with a script for use in the Favorites palette, use the `new` operator:

```
new Thumbnail (node[, name]);
```

node	The node specifier. One of the following: • An ExtendScript `File` or `Folder` object for a file or folder that exists on the local file system. If the referenced file or folder does not exist, causes a run-time error. This object becomes the value of the new object's spec property. • A `Thumbnail` object. This creates a new `Thumbnail` object that references the same node. See Multiple references to the same node. • A string containing a fully qualified Bridge URI (uniform resource identifier). To be a fully qualified Bridge URI, the path should include a prefix that identifies the node type and its associated the node handler; the default is `bridge:` for the default node handler. A path to a local or remote file, folder, or page, which becomes the value of the new object's path property.
name	Optional. A localizable string to use as the display name for the thumbnail icon in the browser window. For script-defined node types, the node-handling extension must be registered before the thumbnail is created for the name to take effect. If not supplied, the display name defaults to the path or spec value. **CAUTION:** For a `Thumbnail` object associated with an ExtendScript `File` or `Folder` object, using the `name` argument renames the folder or file on disk.

Examples of thumbnail creation

```
// references a folder
```

```
var myLocation = new Thumbnail(Folder("/C/myFolder"));
// a second reference to the same node
var newLocation = new Thumbnail(myLocation);
// references a file, and renames the file on disk
var myFile = new Thumbnail(File("/C/myFolder/file.txt"), "myfile.txt");
// references a URL
var myURL = new Thumbnail ("http://www.adobe.com");
```

Multiple references to the same node

Multiple `Thumbnail` objects can refer to the same node. In JavaScript terminology, two such objects are equal, but not identical. That is, if you declare two `Thumbnail` objects that point to the same file, the JavaScript equality operator "`==`" returns `true`, but the identity operator "`===`" returns `false`. Any values that are assigned (not predefined) in one of the objects are not reflected in the other.

This example creates two `Thumbnail` objects that reference the same node, and shows that an arbitrary property defined on one cannot be referenced on the other.

```
var t1 = new Thumbnail(File("/C/Temp/afile.txt");

var t2 = new Thumbnail(File("/C/Temp/afile.txt");


t1 == t2;  // returns true

t1 === t2; // returns false


t1.newNote = "a note for the thumbnail";

alert(t2.newNote); // t2.newNote is undefined
```

For a thumbnail that references a file, however, you can assign arbitrary data to the `Thumbnail.metadata` object, which can be referenced from either object.

```
var t1 = new Thumbnail(File("/C/myFolder/myfile.txt"));

var t2 = new Thumbnail(File("/C/myFolder/myfile.txt"));

t1.newProperty = "arbitrary value";

var val = t2.newProperty; // result is undefined.

//properties created directly in thumbnail are not shared

var md = t1.metadata;

md.namespace = "http://ns.adobe.com/photoshop/1.0/";

md.SpecialNotes = "Special notes for this file.";

// You can access SpecialNotes from either Thumbnail object

t2.metadata.namespace = "http://ns.adobe.com/photoshop/1.0/";

alert("Special Notes: ", t2.metadata.SpecialNotes);
```

The `spec` values of the two thumbnail objects reference different `File` objects, and so are not equal. However, the two `File` objects reference the same file, as shown by inspecting the string value:

```
t1.spec == t2.spec; //returns false
```

```
t1.spec.toString() == t2.spec.toString(); // returns true
```

Thumbnail properties

extensionName	ExtensionModel	A model object for the node-handling extension that applies to this thumbnail is accessible through a property with the same name as the ExtensionHandler Object name. Adobe Bridge instantiates the ExtensionModel Object when it creates the `Thumbnail` object in order to display the node.

aliasType	String	If the value of type is `alias`, the kind of target this thumbnail represents, one of:
		
		`file`
		`folder`
		
		Otherwise `undefined`.

children	Array of Thumbnail	An array of `Thumbnail` objects for the children of this container node. When this object references a folder, the children are the thumbnails that reference the contents of the folder. By default, when the thumbnail is selected in a navigation pane, its children are shown in the Content pane. Read only. **NOTE:** This array is not populated until the loaded event has occurred for the document. The list of children is cached on the first reference so that subsequent references do not result in further disk access. To ensure that the list is up to date (for example after you have performed operations that may have resulted in children being deleted, added, or renamed) call the refresh() method to make sure the list is updated on the next access. You do not need to refresh if you changed the content or properties of a child thumbnail.
container	Boolean	When `true`, the node is a container; that is, it can have child nodes (regardless of whether it currently has any children). Only container nodes can appear in the Folders and Favorites palettes. Folder and web-browser thumbnails are containers; a node-handling extension can define other container node types. Read only.

core	Object	Provides access to the core node-data sets defined by the default node handler. Contains a set of Infoset Objects. Refer to core node attributes through the name of the core Infoset Object and InfosetMemberDescription Object. For example, `myThumbSize = myThumb.core.immediate.size`
creationDate	Date	Date the referenced file or folder was created, if it can be determined. Read only.
exists	Boolean	When `true`, the resource for this file or folder node exists on the local disk. Node-handling extensions can define other criteria for whether a node exists.
extensions	Array of ExtensionHandler	All of the ExtensionHandler Objects that could handle this node; the last one in the list is the one that does handle it.
hasMetadata	Boolean	When `true`, this thumbnail is associated with a file that contains embedded metadata. Otherwise `false`.
hidden	Boolean	When `true`, this thumbnail is hidden. When `false` (the default), it is shown. Read only.
iconPath	String	The path to the operating-system icon image file for this node, when it represents a web page.
label	String	The label string for this thumbnail. Can be one of the predefined label strings (as seen in the Label menu) to apply one of the standard color tags. Any string that does not match a predefined label is displayed with the default white color tag. Removing the label string removes the color tag as well. Read/write.
lastModifiedDate	Date	Date the referenced file or folder was last modified, if it can be determined. Read only.

location	String	Whether the thumbnail is associated with a local file-system object or an Adobe Drive® node (which can have both a local and remote replica). One of:
		
		`local`
		`unknown`
		`AdobeDriveExtension`

locked	Boolean	When `true`, this node represents a read-only file in Windows, or a file that has been locked in the Finder in Mac OS.

metadata	Metadata	The Metadata Object associated with this thumbnail, if any. Otherwise `undefined`. Read only. Some properties of this `Metadata` object may not be immediately available. To ensure the object contains current data, set `app.synchronousMode` to `true`, or use `Thumbnail.synchronousMetadata`. If no metadata is defined for a thumbnail, and you attempt to access a metadata property through this property, the value `undefined` is returned. Note that this differs from the behavior in Adobe Bridge CS2, where an exception was thrown in this case.
mimeType	String	The referenced file's MIME type, if it can be determined; otherwise, the empty string. Read only.
model	ExtensionModel	The ExtensionModel Object associated with this node. Read-only.
name	String	The label displayed for the thumbnail. Read/write. Default is the `path` value.
parent	Thumbnail	The `Thumbnail` object for the parent node of this thumbnail. The value is `undefined` for thumbnails added to the root level of `app.favorites`. This object is in the `children` array of its parent. Read-only.
path	String	A Bridge URI containing the path or URL for the referenced node. Set when the object is created, using the first argument to the Thumbnail object constructor. Read only.
rating	Number	The rating value for this thumbnail, in the range [-1..5]. A negative value signifies a rejection. If set to a value that is out of range, the rating is set to 0. Read/write. Applies to all thumbnails regardless of whether they support embedded metadata.
rotation	Number	This thumbnail's rotation, one of: 0: No rotation 90: Rotated 90 degrees clockwise -90: Rotated 90 degrees counterclockwise 180: Rotated 180 degrees All other values are ignored. Read/write.

spec	File, Folder	An ExtendScript `File` or `Folder` object for this thumbnail's referenced node. Set when the object is created, using the first argument to the Thumbnail object constructor. If the thumbnail does not reference a file or folder, the value is `undefined`. Read only.
synchronousMetadata	Metadata	Waits for confirmation of a valid value to return the Metadata Object associated with this thumbnail, if any. Otherwise `undefined`. Read only. If `app.synchronousMode` is `true`, this is the same as `Thumbnail.metadata`.

type	String	The type of node this thumbnail references. One of:
		
		`file`
		`folder`
		`alias`
		`package`
		`application` (an executable file)
		`other`

uri	String	The full Bridge URI (unique resource identifier) for this thumbnail. This is the path string preceded by a registered node-type identifying prefix such as "vc:". Read only.

Thumbnail functions

Additional functions can be defined for the `Thumbnail` object by a node-handling extension; see `ExtensionHandler.methods`.

copyTo() *thumbnailObj*`.copyTo (`*path*`)`	Creates a new Thumbnail Object that references the same node as this one, and adds it to the target thumbnail's `children` list. Each call to this function is added to the Undo stack. Returns `true` on success.
path	The parent node of the new copy. A File or Folder object, a Thumbnail Object, or a Bridge URI string.

moveTo()
thumbnailObj.moveTo (*path*)

Removes this thumbnail from its current parent, and adds it to the target thumbnail's `children` list. Each call to this function is added to the Undo stack.

Returns `true` on success.

NOTE: If the thumbnail refers to an existing file or folder, this moves the referenced file or folder on disk.

```
var thumbnail =
  new Thumbnail(File.openDialog("Source?"));
var target = new
Thumbnail(Folder.selectDialog("Target?"));
if (thumbnail.moveTo(target)) {
  Window.alert("move succeeded");
}
else Window.alert("move failed");
```

path The new parent node. A File or Folder object, a Thumbnail Object, or a Bridge URI string.

open()
thumbnailObj.open ()

Launches the file referenced by this thumbnail in the appropriate application (such as Photoshop for JPEG files). This is the same as choosing **Open** from the **File** or context menu, or double-clicking the thumbnail icon in the Content pane.

If this thumbnail references a JSX file, runs the script in its target application, or, if no target is specified, in the ExtendScript Toolkit. See the *JavaScript Tools Guide*.

If this thumbnail references a folder, navigates to that folder in the Folders pane—that is, sets `document.thumbnail` to this thumbnail.

Returns `true` on success.

openWith()
thumbnailObj.openWith (*appPath*)

Launches the file referenced by this thumbnail in the specified application.

Returns `true` on success.

appPath A platform-specific path string to the application, as returned in `appPath` property of the openWith event object when a user makes a selection of thumbnails in the Content pane, then selects an application from the **Open With** submenu of the **File** or context menu.

refresh()
thumbnailObj.refresh ([*infosetName*])

Refreshes an associated information set or sets to reflect the current state of this node's referenced file or folder.

For container thumbnails, marks the `Thumbnail` object so that the next access to the `children` property causes a disk access to update the cached list of children.

- For non-container thumbnails, returns `true` if the node has changed since the last access.
- For container thumbnails, returns `true` if the node has been renamed since the last access.

infosetName — Optional. An array of Infoset Object names, or the string `all` (the default), which refreshes all information sets associated with this thumbnail.

registerInterest()
thumbnailObj.registerInterest (*callback*)

Registers a callback function that is executed whenever a node-data value in this thumbnail changes.

Returns `undefined`.

callback — A developer-defined function that conforms to the following prototype:

```
function interestCallback (thumb, message)
```

thumb—This `Thumbnail` object.
message—A string, the name of the Infoset Object whose update triggered the call.

remove()
thumbnailObj.remove ()

Deletes this `Thumbnail` object, and also deletes the file or folder associated with the thumbnail from the disk.

Returns `undefined`.

resolve()
thumbnailObj.resolve ()

If the value of type is `alias`, retrieves a `Thumbnail` object for the target of the alias.

- If the alias can be resolved, returns the `Thumbnail` object for the target.
- If the alias cannot be resolved, returns `undefined`.
- If the `type` is not `alias`, returns this `Thumbnail` object.

NOTE: Adobe Bridge does not support symbolic links (that is, links created in Mac OS or Unix with `-s`).

revealInSystemBrowser()
thumbnailObj.revealInSystemBrowser ()

Opens the platform-specific native file browser, displaying and selecting the file or folder for this thumbnail.

Returns `undefined`.

unregisterInterest() *thumbnailObj*.registerInterest(*callback*)	Removes a callback function from the list of callbacks registered for this thumbnail. Returns `undefined`.
callback	A developer-defined function, previously registered with registerInterest().
verifyExternalChanges() *thumbnailObj*.verifyExternalChanges()	Re-enumerates the children of a container node. Has no effect if the node is not a container. Returns `undefined`.

ThumbnailPanelette Object

An instantiable subclass of the Panelette Base Class, representing a member subpanel of a InspectorPanel Object that displays resizeable thumbnail icons, with corresponding text labels for each thumbnail. The text can be specified with literal strings, derived from data in various ways, or calculated using JavaScript; see Panelette markup elements.

The displayed thumbnails are mouse-sensitive. A single click makes a thumbnail the inspection focus for the Inspector, and reveals or navigates to that thumbnail in the Content pane.

ThumbnailPanelette constructor

To create a new object, use the `new` operator:

```
new ThumbnailPanelette(name, titleMarkup, thumbnails, keyValuePairs,
                       textPosition*);
```

Parameters set the corresponding properties. The `name` and `titleMarkup` properties are inherited from the Panelette Base Class.

ThumbnailPanelette properties

keyValuePairs	Array of Array of 2-element Array	A collection corresponding to the `thumbnails` array, where each member contains a set of two-element arrays, each of which specifies a text field for the corresponding thumbnail. Field arrays are in the format [*key*, *value*]. The key is shown on the left of each field in bold, and the value on the right in plain text. The key and value fields containing string literals combined with Panelette markup elements, which specify the text to be displayed with the thumbnail icons. Read/write.
textPosition	String	Optional. The placement of the displayed text in the horizontal presentation mode. Read/write. One of: **below**—(Default) Displays text below the thumbnail icon. **right**—Displays text to the right of the thumbnail icon.
thumbnails	String or Array of Thumbnail	An array of Thumbnail Objects to be displayed in this subpanel, or a string containing panelette markup that obtains a set of thumbnails at display time. Read only.

2 Node-Handling Extension Object Reference

This chapter presents objects that are available to product or plug-in developers who wish to extend the node-handling capability of Adobe Bridge. This object model allows advanced developers to integrate a product or plug-in with Adobe Bridge by defining new node types.

Object summary

The objects are presented alphabetically. For each object, complete syntax details are provided for the constructor, properties, and functions.

Badge Object	Represents a status icon associated with a node in the Content pane.
CacheData Object	Tracks the current cache status of node data in an Infoset Object.
CacheElement Object	Encapsulates all node-handling data and the node handler for a Thumbnail Object.
ExtensionHandler Object	Defines an extension to the Adobe Bridge node-handling model.
ExtensionModel Object	Provides a framework for developer-implemented node-handling methods for a specific node.
FilterDescription Object	Encapsulates a filtering criterion for handled nodes.
Infoset Object	Encapsulates private node data associated with a node-handling extension, as defined by an ExtensionHandler Object. Adobe Bridge defines Core infosets, which script-defined handlers must support.
InfosetMemberDescription Object	Describes a data member of an Infoset Object. Corresponds to a developer-defined property of the Thumbnail Object for a handled node.
ModalOperator Object	An independent node-handling operation with its own user interface.
Operand Object	Utility class for searches in handled nodes.
Operator Class	A base class for lengthy or complex node-handling operations.
ProgressOperator Object	A lengthy node-handling operation that can report its progress and be interrupted or canceled.
Rank Object	Utility class for searches in handled nodes.
Scope Object	Utility class for searches in handled nodes.
SearchCondition Object	Defines a specific condition that must be met for a handled node to match a search. Returned from a selection in the Find dialog.
SearchCriteria Object	Defines one possible search criterion for a search among handled nodes. Passed to Adobe Bridge to populate the Find dialog.
SearchDefinition Object	Defines a set of search criteria for a search among handled nodes. Passed to Adobe Bridge to populate the Find dialog.

SearchDetails Object	Utility class for searches in handled nodes.
SearchSpecification Object	Defines a specific search among handled nodes. Returned from a selection in the Find dialog.
SortCriterion Object	Defines a sorting criterion property for handled node.

Badge Object

Represents a status icon that can be displayed with a node in the Content pane. A node can be associated with up to four badge icons, specified in the `badges` member of the badges core node-data set. See 'Core infosets' on page 132.

Badge properties

`badge`	BitmapData	The BitmapData Object that defines the icon image.
`toolTip`	String	A string that is shown when the mouse hovers over the badge icon (in the `details` view).

CacheData Object

This object associates a cache status with each Infoset Object in a CacheElement Object. The status determines whether the data needs to be refreshed.

Your ExtensionModel Object method for refreshInfoset() should update the cache status for each data set it updates, including core data sets:

```
myModel.refreshInfoset = function(infosetName) {
// retrieve the cache
    thisCache = this.privateData.cacheElement;
// update the cache status for core data
    if(infosetName == "immediate") {
        thisCache.immediate.cacheData.status = "good";
    }
...
```

Adobe Bridge does not check any data value until the cache status is set.

CacheData properties

cookie	String	Opaque storage to aid extensions in discovering the cache status. The string contains data in an extension-defined format. Read/write.
status	String	The cache status for a member of the associated information set, or of the set itself. Read/write. One of: `good` (known valid data) `bad` (was good at one point, but not now) `unknown` `inProgress` (status is `inProgress` after a refresh has been requested but before the data is confirmed as `good`) `invalid` (status is `invalid` if the ExtensionModel Object no longer exists)

CacheElement Object

This object associates a Thumbnail Object with the ExtensionModel Object that handles the node and that defines additional node data. The cache collects all currently defined node data.

This object actually contains the ExtensionModel Object that is created for the thumbnail, as well as the associated Infoset Objects. Each Infoset Object in the cache is associated with a CacheData Object object that contains its cache status.

When Adobe Bridge needs to display a handled node, it instantiates this object. It creates the ExtensionModel Object using the handler's `makeModel()` method, and stores it in the `CacheElement`. It then passes the `CacheElement` object to the node handler's model method `registerInterest()`.

Your implementation of the `registerInterest()` method must store the cache object (typically in the model object's `privateData` property) so that the model's `refreshInfoset()` method can use it to update the data. For example, to store the reference to the containing `CacheElement` in the model (and remove the reference when the node is no longer displayed):

```
// associate this node with the node data cache
myModel.registerInterest = function(cacheElement) {
   this.privateData.cacheElement = cacheElement;
 }
// dissociate this node from the node data cache
myModel.unregisterInterest = function() {
   this.privateData.cacheElement = undefined;
 }
```

Your model methods can access the cache element, and through it all Adobe Bridge-defined and script-defined thumbnail properties:

```
thisCache = this.privateData.cacheElement;
myProp = thisCache.myInfoset.myInfosetMember;
```

CacheElement properties

infosetNames	Infoset	Every node data set associated with this cache is accessible through a property with the same name as the Infoset Object name. Read only.
path	String	The path of the asset associated with this object. Read only.

CacheElement functions

doAuthentication() *obj*.doAuthentication ()	Calls the `authenticate()` method defined in the ExtensionHandler Object associated with this element. Returns `undefined`.

ExtensionHandler Object

This object defines the properties and methods needed to extend the Adobe Bridge node model. It does not implement any of the methods; you must implement them to define you own node type and handler. Your `ExtensionHandler` must implement all of the methods that are applicable to its node model.

Register a script-defined extension handler with `app.registerExtension()`. You can access the global list of all registered extensions through `app.extensions`.

Your node-handling extension defines a node type. Your node types are identified by a Bridge URI prefix. You must associates your handler with at least one prefix, using `app.registerPrefix()`.

When it needs to display a node of a type that is managed by this handler, Adobe Bridge uses the handler's `makeModel()` method to create an instance of ExtensionModel Object, and associates it with the Thumbnail Object that it creates for the node, through a CacheElement Object.

Your model implementation allows you to create and update a set of script-defined properties in the Thumbnail Objects for your nodes. The Thumbnail Object has a property with the same name as the `ExtensionHandler` that manages it, which allows scripts to access the node data defined by the handler. Data managed by each model is kept in Infoset Objects. Each Infoset Object member corresponds to one `Thumbnail` property. To access a script-defined property value in a Thumbnail Object, use this format:

```
Thumbnail.handlerName.infosetName.memberName
```

Extensions must support Core infosets defined by Adobe Bridge, but can also add new properties. In order to define you own thumbnail properties for nodes of the type you define, define and register an Infoset Object using `app.registerInfoset()`.

NOTE: This object is designed to extend the node-handling behavior of Adobe Bridge itself, not the scripting functionality. The full range of methods are not available to scripts from the user-level Thumbnail Object.

Code examples

The sample code distributed with the Adobe Bridge SDK includes these code examples that demonstrate how to define node-handling extensions:

Node-handling extension examples in *sdkInstall*`/sdksamples/`	
`BasicExtensionHandler.jsx`	Shows how to create a basic node-handler, defining a minimal set of handler and model methods.

ExtensionHandler object constructor

To create a new object, use the `new` operator:

```
new ExtensionHandler(name)
```

name	String	The name of this extension. Must be a valid JavaScript identifier (containing no colons or special characters, and beginning with a lowercase letter).

ExtensionHandler properties

infosets	Array of Infoset	A collection of Infoset Objects defining node data managed by this handler, reflected in handler-defined Thumbnail Object properties. Read only. Modify with `app.registerInfoset()`and `app.unregisterInfoset()`.
methods	Object	New methods that are defined on Thumbnail Objects that are managed by this handler. Each object property is a function definition; for example: `ext.methods.fnName = function(arg1, arg2){body}` Each method can be accessed at run time through `Thumbnail.fnName()`.
name	String	The unique identifying name of this node-handling extension. Must be a valid JavaScript identifier (containing no colons or special characters, and beginning with a lowercase letter). Read only.
prefixes	Array of String	A collection of lexical prefixes for Bridge URIs, which identify node types for which this handler supplies a model. Read only. Modify with `app.registerPrefix()`and `app.unregisterPrefix()`.

ExtensionHandler methods

Your `ExtensionHandler` instance must implement all of the methods described here. Handler methods can be immediate or long-running:

Immediate handler operations simply perform an operation and return when it is done. These functions must not take a significant amount of time; if they are slow, they will negatively affect Adobe Bridge browsing performance.

Long-running handler operations create and return `Operator` objects to perform time-intensive file-system operations that block the main thread. Adobe Bridge view code or your display code passes the object to `app.enqueueOperation()` to initiate the action when appropriate.

Immediate handler operations

getBridgeURIForPath() *obj*.getBridgeURIForPath (*path*)	Convert a path string to a canonical Bridge URI, that is, one that includes the node-type identifying prefix. If the path is already in the form of a canonical Bridge URI, the method should simply return it. If the path cannot be parsed into a Bridge URI, the method should return `undefined`. Return the Bridge URI string for the path, or `undefined` if the path cannot be parsed.
path	A string containing a node path.
getBridgeURIForSearch() *obj*.getBridgeURIForSearch (*scope*, *specification*)	Execute an extension-defined search among Adobe Bridge nodes of an extension-defined node type. The Find dialog calls this method in response to a click on **Find**, if the dialog has been invoked for a node handled by this extension, or for a container that contains a handled node type. Your method can store the parameters such that they can be retrieved by the `ExtensionModel`.getSearchDetails() method for the returned container node, or that method can recreate the specification and target by some other means. Return the search result, a Bridge URI for a container node that contains the matching nodes.
scope	A Thumbnail Object for the target node, which was selected when the user invoked the Find dialog. Your search can be extended or limited by the handler-defined scope given in the search specification.
specification	The SearchSpecification Object that defines how to perform the search. The Find dialog creates this object from the user's choices, which can include choices added by your handler's SearchDefinition Object.

getSidecars()
obj.getSidecars (*masters*, *possibleExtensions*, *result*)
	Retrieve existing sidecar files for a set of nodes. A sidecar file is a file used to store information related to another file, called the master file. A sidecar has the same base file name as its master file, but a different extension. It can contain metadata (typically XMP), a rendition of the master file (such as a thumbnail version), or represent some status information of the master file (such as whether it is in use or locked).
	
	The handler should spawn a thread to perform the operation and return immediately. The thread should search for matching sidecar files in the same container as each master file, and set `result.masterAndSidecars` to an Array of JavaScript objects in the format
	
	`{ master : Thumbnail,`
	`  sidecars : Array of Thumbnail }`
	
	This array must correspond to the original `masters` array, setting the sidecars element to `undefined` or an empty Array, `[]`, if no sidecar files are found for a master file.

masters	Array of Thumbnail Object. The set of nodes, children of a single parent node handled by this extension, for which to find sidecar files.
possibleExtensions	Array of String. A list of file extensions that are considered sidecars.
result	A JavaScript object containing the result, set by the spawned thread.
makeModel() *obj*.MakeModel (*path*)	Create a model instance that implements node handling. Adobe Bridge calls this each time it needs to display a handled node. Return the new ExtensionModel Object.
path	A string containing the path for the node to be displayed.

Long-running handler operations

Implement these functions to create instances of the Operator Class which can perform the desired operation, and if needed, provide Adobe Bridge with information about the status and progress.

Each function creates and returns a ModalOperator Object or ProgressOperator Object which can perform the operation in a separate thread, and, if needed, provide Adobe Bridge with information about the status of the background operation. Adobe Bridge calls `app.enqueueOperation()` to initiate the action when appropriate. This in turn calls the start() method defined for the returned `Operator` object.

acquirePhysicalFiles()
obj.acquirePhysicalFiles (*sources*, *timeoutInMs*, *showUi*, *message*, *recursionOption*)

Create and return an operator that acquires actual file data for a set of placeholder nodes.

Return a ModalOperator Object or ProgressOperator Object.

sources	An Array of Thumbnail Object for the set of nodes to operate on.
timeoutInMs	Optional. A number of microseconds after which to abort the operation. Default is 0, meaning no timeout.
showUi	Optional. Whether to show a user interface during the operation, one of `allowUi` (the default) or `suppressUi`.
message	Optional. A display string that describes this operation.
recursionOption	Optional. Whether to perform the operation recursively in children of the source nodes, one of `doNotRecurse` (the default) or `recurse`.

duplicate()
obj.duplicate (*sources*, *timeoutInMs*, *showUi*, *message*)

Create and return an operator that duplicates a set of nodes that are handled by this handler.

Return a ModalOperator Object or ProgressOperator Object.

sources	An Array of Thumbnail Object for the set of nodes to operate on.
timeoutInMs	Optional. A number of microseconds after which to abort the operation. Default is 0, meaning no timeout.
showUi	Optional. Whether to show a user interface during the operation, one of `allowUi` (the default) or `suppressUi`.
message	Optional. A display string that describes this operation.

moveToTrash()
obj.moveToTrash (*sources*, *timeoutInMs*, *showUi*, *message*)

Create and return an operator that deletes a set of nodes, marking the associated files for deletion on disc by moving them to the system trash or recycle bin.

Return a ModalOperator Object or ProgressOperator Object.

sources	An Array of Thumbnail Object for the set of nodes to operate on.
timeoutInMs	Optional. A number of microseconds after which to abort the operation. Default is 0, meaning no timeout.
showUi	Optional. Whether to show a user interface during the operation, one of `allowUi` (the default) or `suppressUi`.
message	Optional. A display string that describes this operation.

rotate()
obj.rotate (*targets*, *rotation*, *timeoutInMs*, *showUi*, *message*)

Create and return an operator that sets the rotation setting in metadata for a set of thumbnails to the same value for all. This does not rotate the image bits.

Return a ModalOperator Object or ProgressOperator Object.

targets	An Array of Thumbnail Object, the set of target thumbnails.

rotation	A Number, the rotation angle in degrees. Positive values rotate clockwise, negative values rotate counterclockwise. Allowed values are -90, 0, 90, 180, 270.
timeoutInMs	Optional. A number of microseconds after which to abort the operation. Default is 0, meaning no timeout.
showUi	Optional. Whether to show a user interface during the operation, one of `allowUi` (the default) or `suppressUi`.
message	Optional. A display string that describes this operation.

`setLabels()` *obj*`.setLabels (`*targets, labels, timeoutInMs, showUi, message*`)`	Create and return an operator that sets the labels for a set of thumbnails. Return a ModalOperator Object or ProgressOperator Object.
targets	An Array of Thumbnail Object, the set of target thumbnails.
labels	An Array of Strings, the set of label values corresponding to the target thumbnails. See `Thumbnail.label`.
timeoutInMs	Optional. A number of microseconds after which to abort the operation. Default is 0, meaning no timeout.
showUi	Optional. Whether to show a user interface during the operation, one of `allowUi` (the default) or `suppressUi`.
message	Optional. A display string that describes this operation.

`setRatings()` *obj*`.setRatings (`*targets, ratings, timeoutInMs, showUi, message*`)`	Create and return an operator that sets the ratings for a set of thumbnails. Return a ModalOperator Object or ProgressOperator Object.
targets	An Array of Thumbnail Object, the set of target thumbnails.
ratings	An Array of Numbers, the set of rating values corresponding to the target thumbnails.
timeoutInMs	Optional. A number of microseconds after which to abort the operation. Default is 0, meaning no timeout.
showUi	Optional. Whether to show a user interface during the operation, one of `allowUi` (the default) or `suppressUi`.
message	Optional. A display string that describes this operation.

`setXmp()` *obj*`.setXmp (`*targets, xmpPackets, timeoutInMs, showUi, message*`)`	Create and return an operator that embeds XMP file metadata packets in a set of files. Return a ModalOperator Object or ProgressOperator Object.
targets	An Array of Strings, the set of file paths.
xmpPackets	An Array of Strings, the set of XMP packets corresponding to the target files.
timeoutInMs	Optional. A number of microseconds after which to abort the operation. Default is 0, meaning no timeout.

showUi	Optional. Whether to show a user interface during the operation, one of `allowUi` (the default) or `suppressUi`.
message	Optional. A display string that describes this operation.

ExtensionModel Object

Supports the basic framework for Adobe Bridge node-handling extensions by tracking the connection between your display model and the file or page sources. To implement an extension, you must define the methods that handle nodes, as described here.

When Adobe Bridge needs to display a handled node, it uses the makeModel() method defined in the ExtensionHandler Object to instantiate this object. It then creates a CacheElement Object to contain the model object and associate it with the new Thumbnail Object that it creates for the node.

The `ExtensionModel` that your handler creates implements the actual node-handling methods that perform operations on a selected thumbnail for your node types. For details of how to implement a node-handling model, see the *Adobe Bridge JavaScript Guide* and the code example in the SDK, *`sdkInstall`*`/sdksamples/BasicExtensionHandler.jsx`.

The model can define private data needed for node handling, accessible through additional properties for the Thumbnail Object. Data managed by a model is kept in Infoset Objects. Each data member corresponds to one script-defined `Thumbnail` property. To access a script-defined property value in a Thumbnail Object, use this format:

```
Thumbnail.handlerName.infosetName.memberName
```

ExtensionModel constructor

Your makeModel() method uses the `new` operator to create an object:

```
new ExtensionModel(path)
```

`path`	String	The absolute path or fully-qualified URL for the source file or page to be displayed. Adobe Bridge passes the path to makeModel() when it is creating a new Thumbnail Object for a handled node.

ExtensionModel properties

`privateData`	String	Stores private data associated with your node-handling model. Typically, you use it to store the parent CacheElement Object, which is passed to your model's registerInterest() method. This in turn provides access to each Infoset Object that contains data managed by this model. This is the only way to store private data in this object. If you assign a value such as `model.myProp=7`, it will *not* be available in the context of model functions. Within a model function, `this.myProp` will be undefined. You can, however, assign a value to `model.privateData.myProp` and access it through the `this` object.

ExtensionModel methods

Your `ExtensionModel` instance must implement the methods marked as required. Model methods can be immediate or long-running:

Immediate model operations simply perform an operation and return when it is done. These functions must not take a significant amount of time; if they are slow, they will negatively affect Adobe Bridge browsing performance.

Long-running model operations create and return `Operator` objects to perform time-intensive file-system operations that block the main thread. Adobe Bridge view code or your display code passes the object to `app.enqueueOperation()` to initiate the action when appropriate.

Immediate model operations

`addToDrag()` *`obj.`* `addToDrag` `(`*`pointerToOsDragObject`*`)`	Add this model object to the platform-specific drag object. Return `true` on success.
`pointerToOsDragObject`	A pointer to the platform-specific drag object.
`authenticate()` *`obj.`*`authenticate ()`	Required. Handle any required authentication for this node. Return `undefined`.
`cancelRefresh()` *`obj.`* `cancelRefresh (`*`infosetName`*`)`	Cancel a background refresh task started by a call to refreshInfoset(). Return `undefined`.
`infosetName`	The name of the Infoset Object.
`createNewContainer()` *`obj.`*`createNewContainer (`*`name`*`)`	Create a new container node in this container node. If this node is not a container, do nothing. Return the URI string for the new folder, or the Thumbnail Object for the new container node.
`name`	Optional. The name string of the new container node. Default is "New Folder" in Windows, "untitled folder" in Mac OS.
`doLosslessRotate()` *`obj.`*`doLosslessRotate (`*`oriantation`*`)`	Rotate this node without changing image data. Return `true` if the operation can be performed on this node, `false` if it cannot.
`orientation`	The rotation angle in degrees. Positive values rotate clockwise, negative values rotate counterclockwise. Allowed values are -90, 0, 90, 180, 270.
`exists()` *`obj.`*`exists ()`	Required. Report whether this node is valid according to this model. Return `true` if this node is valid, `false` otherwise.
`getCacheStatus()` *`obj.`* `getCacheStatus (`*`infoset, cookie`*`)`	Required. Report the cache status of a node-data set for this node. See CacheElement Object and CacheData Object. Return the cache status string.

infosetName	The name of the Infoset Object.

cookie	A string buffer in which to return the cache status, one of:
	`bad`
	`inProgress`
	`good`
	`unknown`

getDisplayName() *obj*.getDisplayName ()	Retrieve a localized display name for this node. Return the display string.
getFilterCriteria() *obj*. getFilterCriteria ()	Create the full set of filter criteria that can be applied to this container node. These filters appear in the Filter palette when Adobe Bridge displays the contents of this container. Return an array of FilterDescription Objects for the complete set of filters with which to populate the Filter palette.
getParent() *obj*.getParent ()	Retrieve the parent node of this node. Return the Bridge URI string for the parent node.
getPhysicalFileName() *obj*.getPhysicalFileName ()	Retrieve the full file name for this node, including extensions. Return the file name string.
getSearchDefinition() *obj*.getSearchDefinition ()	Create a search definition with which to populate the Find dialog when it is invoked for this container node. Return the SearchDefinition Object.
getSearchDetails() *obj*.getSearchDetails ()	Retrieve or recreate the search specification and target node that were used to create this search-result container node, when it was created by the `ExtensionHandler.` getBridgeURIForSearch() method. Return a SearchDetails Object.
getSortCriteria() *obj*.getSortCriteria ()	Create the full set of sorting criteria for member nodes of this container node. Can construct an entirely new list of criteria, or retrieve the default set from `app.` defaultSortCriteria and modify or append criteria, or return the set unchanged. Return an array of SortCriterion Objects.
getUserSortOrder() *obj*.getSortCriteria ()	Retrieve the opaque XML code representing a user-defined sorting order for container nodes managed by this model, as previously saved by the setUserSortOrder() method. The browser uses the returned value to sort the displayed nodes of this container node (if it returns true for supportsUserSortOrder()). Return a string containing the XML code.

initialize() *obj*.initialize ()	Required, a constructor for this model instance. Initialize the model for this node. Create any necessary support data structures and store them in this object. Adobe Bridge calls this after creating the object with the handler's makeModel() method, whenever it needs to display a handled node. Return `undefined`.
lock() *obj*.lock ()	Make the file associated with this node read-only. • In Windows, modify the read-only file attribute. • In Mac OS, modify the Finder "lock" attribute. Return `false`.
needAuthentication() *obj*.needAuthentication ()	Report whether this node requires authentication. Return `true` if the node requires authentication, `false` otherwise.
refreshInfoset() *obj*. refreshInfoset (*infoset, priority, cost, pageNumber*)	Required. Start a background task with the specified priority and processing cost, to update the data in a node-data set for this node. Adobe Bridge calls a model's refresh method when it needs data from a particular `Infoset` for a particular view or operation. Within this method, access each data element in the stored data cache, using this format (assuming you have stored the cache reference in the `privateData` property): `this.privateData.cacheElement.`*`setName.memberName`* • The operation should set appropriate core data set values, such as `item` and `itemContent` capabilities, to reflect which optional model methods are supported by this handler. See Core infosets. • If the node is a container, the operation must add its child nodes to the core `children` data set, using `Infoset`.addChild(). • The operation must set the cache status of the updated node-data set. See CacheData Object. Return `undefined`.

infosetName The name of the Infoset Object.

priority Optional. The priority to assign this background task, one of `low` (first-in, first-out queue), `normal` (last-in, first-out stack), or `high` (first-in, first-out). High priority is used for currently selected nodes.

cost Optional. The desired processing cost for this background task, one of:

```
lowCostEvenIfFail
guaranteedLow
lowCostEvenIfLowQuality
unlimited
```

pageNumber Optional. The current page number for nodes that represent multi-page documents; for other node types, it is ignored. Default is 0.

registerInterest() *obj*. registerInterest (*cacheElement*)	Required. Notify this model object of the cache that contains the model itself and all its associated data. Your implementation must store the cache object, and use it to access the node data. Typically, you store it in this model's privateData property. Adobe Bridge instantiates the CacheElement Object and passes it to this method whenever it displays a handled node. Return `undefined`.
cacheElement	The name of the CacheElement Object.
registerStructuralInterest() *obj*. registerStructuralInterest (*cacheElement*)	Notifies Adobe Bridge that the cache should be updated when changes occur in children of the displayed node. Return `undefined`.
cacheElement	The name of the CacheElement Object.
setName() *obj*. setName (*newName*)	Set the file name of this node. Change the base name and extension, but not the path name. Return the new URI string for the node.
newName	The new name string.
setUserSortOrder() *obj*. setName (*inXML*)	When the user sorts the children of this container node, the browser passes an opaque string of XML code to this function that represents the user sort order (if this container returns true for supportsLosslessRotate()). The model is responsible for saving it such that it can be retrieved by getUserSortOrder(). Return true on success.
inXML	A string containing the XML code.
supportsLosslessRotate() *obj*.supportsLosslessRotate ()	Report whether this model supports rotation of an image node without changing pixel data. Return true if the model supplies the doLosslessRotate() method.
supportsUserSortOrder() *obj*.supportsUserSortOrder ()	Report whether this model supports user sorting of displayed child nodes. Return true if this is a container node, and the model supplies getUserSortOrder() and setUserSortOrder() methods.

terminate()
obj.terminate ()

Required, a destructor for the model instance.

- A complex node-handling extension can use this to clean up private data created by the initialization and entirely managed by the extension.
- A purely script-based node-handling extension should simply return without attempting to clean up JavaScript data, which is normally handled by the JavaScript garbage collector.

Return `undefined`.

unlock()
obj.unlock ()

Make the file associated with this node read-write.

- In Windows, modify the read-only file attribute.
- In Mac OS, modify the Finder "lock" attribute.

Return `false`.

unregisterInterest()
obj. unregisterInterest (*cacheElement*)

Required. Remove the association between this model and the cache element that contains it. Your implementation must remove the stored reference to the cache object, typically in the model's `privateData` property.

Return `undefined`.

cacheElement The name of the CacheElement Object.

unregisterStructuralInterest()
obj. unregisterStructuralInterest()

Removes the instruction to update the associated cache when changes occur in children of a displayed node.

Return `undefined`.

verifyExternalChanges()
obj.verifyExternalChanges ()

Called when the user attempts to view data in this model's `children` core `Infoset`, and its cache status is `good`. Typically occurs when an Adobe Bridge view regains focus after a period of inactivity. The model can decide whether to force a refresh or not.

Return `undefined`.

wouldAcceptDrop()
obj. wouldAcceptDrop (*type, sources, osDragRef*)

Report whether this node can accept a drop of a specific set of nodes in a drag-and-drop operation of a particular type.

Return `false` if the drop would not be accepted by this node, or one of the action type strings ("copy" or "move") if the drop of all of the sources would be accepted.

type A string specifying the type of drop requested, one of:

```
copy
move
```

sources An array of path strings for the nodes being dragged.

osDragRef A pointer to a platform-specific drag structure containing the source nodes.

Long-running model operations

Implement these functions to create instances of the Operator Class which can perform the desired operation, and if needed, provide Adobe Bridge with information about the status and progress.

Each function creates and returns a ModalOperator Object or ProgressOperator Object which can perform the operation in a separate thread, and, if needed, provide Adobe Bridge with information about the status of the background operation. Adobe Bridge calls `app.enqueueOperation()` to initiate the action when appropriate. This in turn calls the start() method defined for the returned `Operator` object.

`copyFrom()` *`obj`*`.copyFrom` `(`*`sources, timeoutInMs, showUi, message, newNames`*`)`	Create and return an operator that copies a set of nodes into this container, allowing rename. If this node is not a container, the operator should do nothing. Return a ModalOperator Object or ProgressOperator Object.
`sources`	An Array of Thumbnail Object for the set of nodes to operate on.
`timeoutInMs`	Optional. A number of microseconds after which to abort the operation. Default is 0, meaning no timeout.
`showUi`	Optional. Whether to show a user interface during the operation, one of `showUi` (the default) or `suppressUi`.
`message`	Optional. A display string that describes this operation.
`newNames`	Optional. An array of strings the same size as the `sources` array with new names to assign to the copies.
`eject()` *`obj`*`.eject` `(`*`path, timeoutInMs, showUi, message`*`)`	Create and return an operator that unmounts a path. Return a ModalOperator Object or ProgressOperator Object.
`path`	The path string.
`timeoutInMs`	Optional. A number of microseconds after which to abort the operation. Default is 0, meaning no timeout.
`showUi`	Optional. Whether to show a user interface during the operation, one of `showUi` (the default) or `suppressUi`.
`message`	Optional. A display string that describes this operation.
`moveFrom()` *`obj`*`.moveFrom` `(`*`sources, timeoutInMs, showUi, message, newNames`*`)`	Create and return an operator that moves a set of nodes into this container, allowing rename. If this node is not a container, the operator should do nothing. Return a ModalOperator Object or ProgressOperator Object.
`sources`	An Array of Thumbnail Object for the set of nodes to operate on.
`timeoutInMs`	Optional. A number of microseconds after which to abort the operation. Default is 0, meaning no timeout.

showUi	Optional. Whether to show a user interface during the operation, one of `showUi` (the default) or `suppressUi`.
message	Optional. A display string that describes this operation.
newNames	Optional. An array of strings the same size as the `sources` array with new names to assign to the moved nodes.

`resolveLink()` *obj*`.resolveLink (`*sources*`,` *timeoutInMs*`,` *showUi*`,` *message*`)`	Create and return an operator that resolves the link path for a set of nodes, associating each node directly with its link target. **NOTE:** Adobe Bridge CS5 does not support symbolic links (symlinks) in Mac OS. Return a ModalOperator Object or ProgressOperator Object.
sources	An Array of Thumbnail Object for the set of nodes to operate on.
timeoutInMs	Optional. A number of microseconds after which to abort the operation. Default is 0, meaning no timeout.
showUi	Optional. Whether to show a user interface during the operation, one of `showUi` (the default) or `suppressUi`.
message	Optional. A display string that describes this operation.

FilterDescription Object

This object provides programmatic control and customization of the Filter palette, which allows the user to organize and filter the display of thumbnails in the Content pane.

Filters are applied to children of a container node when Adobe Bridge needs to display that container's contents in the Content pane, or display a list of children in a menu. A filter description identifies a metadata property (from embedded XMP metadata) or a node property (from a node-handler-defined Infoset Object) to display in the Filter palette.

The Filter palette displays each filter property, with a line under each property for each value it finds for that property in any child node. The filter description can provide a narrower list of allowed values to display for an XMP property, if the property has a closed value list.

When the user selects a filter, a child node is displayed only if it contains the selected filter property and value.

The list of filter objects that Adobe Bridge uses by default to populate the Filter palette is kept in `app.defaultFilterCriteria`. When displaying a handled container node, Adobe Bridge builds the list of filters by calling the developer-defined getFilterCriteria() method of the node's ExtensionModel Object. Your implementation of this method can create these filter objects, and use them to replace, modify, or add to the default list.

FilterDescription constructor

To create a new object, use the `new` operator:

```
new FilterDescription (name, displayName, filterType,
    xmpNamespace, xmpProperty, closedValueList*);

new FilterDescription (name, displayName, filterType,
    infosetMember, closedValueList*);
```

Parameters set the corresponding properties.

FilterDescription properties

`closedValueList`	Array of String	The set of allowed values for the XMP property, if it has a closed value list. In this case, the Filter palette does not count nodes that have no value for the property. You can cause it to do so by adding the empty string to this list. Empty for properties with open value types. In this case, the Filter palette displays all values found in nodes in the current scope.
`displayName`	String	Optional. A localized name for this filter, shown in the heading line for this filter in the Filter palette. If not supplied, the `name` value is used.

filterType	String	The data type of filter-property value, used in sorting the list of values. String comparisons are case-insensitive. One of:
		
		`date`
		`dimensions`
		`label`
		`number`
		`orientation`
		`rating`
		`string`
		`stringList`

infosetMember	String	The name of the node property to use as a filter, as defined in the InfosetMemberDescription Object. **NOTE:** The filter property must be *either* an XMP metadata property or an Infoset Object node-data property; if both are defined, the XMP property takes precedence and the node-data property is ignored.
isExclusive	Boolean	When `true`, only one of the filter values can be set at a time. Selecting one value in the Filter palette automatically deselects other values.
name	String	The unique identifying name of the filter. If there is no `displayName`, this is shown in the heading line for this filter in the Filter palette.
xmpNamespace	String	The namespace of the XMP property used as a filter.
xmpProperty	String	The key name of the XMP property used as a filter.

Infoset Object

This object represents application-defined or script-defined data for Adobe Bridge nodes.

- For a script-defined node-handling extension, you can register an `Infoset` object that defines a related set of script-defined Thumbnail Object properties for handled nodes.
- Adobe Bridge-defined `Infoset` objects and their members are listed in 'Core infosets' on page 132.

To declare the properties, create the `Infoset` object and associate it with your ExtensionHandler Object using `app.registerInfoset()`. The `Infoset` object is added to the list in the `ExtensionHandler` infosets property.

An `Infoset` is a named set of data elements. Each member element has a name and type, defined by a InfosetMemberDescription Object. Each member name becomes a property of the containing `Infoset`, and you can access the data value, of the corresponding type, through that property.

To access a script-defined property value in a Thumbnail Object, use this format:

```
ThumbnailObject.handlerName.infosetName.memberName
```

For example, to access a `color` value in `myInfo` in thumbnail `t1`, where the `myInfo` set is managed by `myExtension`, use:

```
t1.myExtension.myInfo.color.
```

Infoset object constructor

Create the object with the `new` operator:

```
new Infoset (name, description)
```

Parameters set the corresponding properties (*`name`* sets `infosetName`).

Infoset properties

`cacheData`	CacheData	The CacheData Object containing cache status for this set in the CacheElement Object that collects all node data for this node. The status reflects whether any associated values have changed such that the set needs to be refreshed. Read/write.
`description`	Array of InfosetMemberDescription	The InfosetMemberDescription Objects containing the member names and data types of data values contained in this set. Read/write.
`extension`	String	The name of the ExtensionHandler Object that manages this data. Available after this set has been registered with `app.registerInfoset()`. Read only.

memberValueName	*memberValueType*	The `InfosetMemberDescription.name` of each member is a property of the set. The property provides access to the data value, of the type specified by the corresponding InfosetMemberDescription Object. Read only.
infosetName	String	The name of this set. Must be a valid JavaScript identifier. This becomes a property of the ExtensionModel Object for the managing extension. Read/write.

Infoset functions

addChild()
obj.addChild
(*path, model, containerHint*)

Adds a child node to the core data set `children`. (See Core infosets.) Use this in the model's refreshInfoset() method to add any children of a handled container node. For example:

```
myModel.refreshInfoset = function( infosetName ) {
  if( infosetName == "children"  ) {
   this.privateData.cacheElement.children.addChild (
      "bridge:myNode:myChildNodeSource.ext" );
  }
}
```

path	The Bridge URI (path and file name) of the child node.
model	Optional. An ExtensionModel Object that manages the new child. Can be `undefined` (the default).
containerHint	Optional. Whether the new child is a container, one of "`container`" or "`notContainer`" (the default). Ignored if *`model`* is provided; otherwise, controls which icon is used for the child.

initializeMembersToDefaultValues()
obj.initializeMembersToDefaultValues ()

Sets all members of this set to the default value for the data type:

String: "" (empty string)
Boolean: `false`
Number: 0
SizeInBytes: 0

Core infosets

Adobe Bridge defines a set of core node-data sets, represented by `Infoset` objects. The core node data must be updated as appropriate by all script-defined node-handling extensions, in order to support the default node-handling behavior of Adobe Bridge. The following table shows the names of the core data sets and their members.

Infoset name	Member names	
`immediate`	Contains mandatory node information, supplied at node creation.	
	`creationDate`	The creation date of the file or folder node as determined by the operating system.
	`displayPath`	The user-readable platform specific display path of the file or folder node.
	`fileUrl`	The URL for the file or folder node.
	`isApplication`	True if the node is an executable file.
	`isContainer`	True if the node is a container.
	`isDeleted`	True if the node has been marked for deletion (moved to the trash or recycle bin).
	`isHidden`	True if the file or folder for the node is hidden.
	`isLink`	True if the node is a shortcut or alias for a file or folder.
	`isPackage`	True if the node is a package in Mac OS.
	`modificationDate`	The modification date of the file or folder for the node as determined by the operating system.
	`name`	The name of the file or folder for the node.
	`size`	The node's file size.
	`sortIndex`	A string used to sort the node by name.

Infoset name	Member names	
item	Node information that can be determined without opening and inspecting the contents of the referenced file.	
item *capabilities*	canBeDragSource	True if the node can be the source of a drag-and-drop action.
	canBeDropTarget	True if the node can be the target of a drag-and-drop action.
	canCreateNewContainer	True if the node supports creation of child container nodes.
	canCreateNewLink	True if a link or alias can be created from this node.
	canDelete	True if the node can be deleted (moved to the trash or recycle bin).
	canDuplicate	True if the node can be duplicated.
	canEject	True if the node represents removable media, such as a CD or network drive.
	canGetFileUrl	True if the node can be accessed by the operating system through a file URL.
	canLock	If the node is writable and canLock is true, the node can be locked/unlocked. In addition, the "Lock Item"/"Unlock Item" context menu is enabled. If users implement their own ExtensionModel Object, they should set this property to true in the `refreshInfoset()` method if they want the current node to support lock functionality.
	canOpen	True if the node can be opened and the "Open" menu is enabled. If users want the current node to support open functionality, this property should be set to true in the `refreshInfoset()` function of a user-defined ExtensionModel Object.
	canSearch	True if the node supports search operations.
	canSetName	True if the node can be renamed.

Infoset name	Member names	
item *descriptors*	isExternalEditInProgress	True if the file for this node is open in another application.
	isLinkToContainer	True if the node is a shortcut or alias that links to a container node.
	isLockedByUser	True if the file or folder for this node is set as read-only.
	isNeverWritable	True if this node is a volume that is never writable, such as a CD or disk image.
	isPhysicalFile	True if this node is for a physical file or folder on disk.
	noWritePermission	True if the current user does not have write permission for this node, regardless of whether it is generally writable.
itemContent	Node information that must be determined by opening the referenced file.	
itemContent *capabilities*	canDoCameraRaw	True if the image file is in a camera-raw format.
	canGetFullSize	True if the node supports full-size previews.
	canGetPreview	True if the file can be previewed.
	canGetQuickPreview	True if the camera-raw image file contains a quick-preview image.
	canGetThumbnail	True if the file contains a thumbnail image.
	canGetXmp	True if the file contains embedded metadata.
	canLabelOrRate	True if the node supports labeling and rating.
	canRotate	True if the image file can be rotated.
	canSetXmp	True if the file's embedded metadata is writable.

Infoset name	Member names	
itemContent *descriptors*	dynamicMediaType	The file's dynamic media type. One of: 0 (invalid) 1 (not a dynamic media file) 2 (QuickTime) 3 (DirectShow) 4 (Animated GIF) 5 (Flash®) If an extension sets this to undefined and the cache status to good, Adobe Bridge determines the proper dynamic media type.
	fileFormat	For a file node, the file format string, such as "jpg".
	hasSubContainers	True if this container node can have child nodes that are also containers.
	mimeType	The MIME type for the node, if applicable, such as "image/jpeg".
	pageCount	The number of pages in the file, if applicable.
	tooltip	The node's tooltip string.
quickMetadata	This is the authoritative source of displayed values, although the same properties are also kept in various other places.	
	bitDepth	For image files, the number of bits per pixel.
	colorProfile	For image files, the name of the color profile.

Infoset name	Member names	
quickMetadata *(cont'd)*	colorMode	For image files, the color mode used. One of: -1 (invalid) 0 (monochrome bitmap) 1 (gray scale) 2 (indexed) 3 (RGB) 4 (CMYK) 5 (HSL) 6 (HSB) 7 (multi-channel) 8 (duotone) 9 (LAB) 10 (XYZ)
	hasCrop	For camera-raw images, true if the image is cropped.
	hasRawSettings	For camera-raw images, true if the file has saved settings.
	height	For image files, the image's height in pixels.
	label	The label string assigned to the file or folder, if any.
	rating	The rating number assigned to the file or folder, if any.
	rotation	For image files, the rotation value. One of 0, 90, 180, 270.
	stockPhotoStatus	The Stock Photos status. One of: 0 (none) 1 (thumbnail search image) 2 (comp image) 3 (purchased image)
	xResolution	For image files, the horizontal resolution in pixels per inch (PPI).
	yResolution	For image files, the vertical resolution in pixels per inch (PPI).
	width	For image files, the image's width in pixels.
badges	badges	An array of Badge Objects representing the node's status icons. A node in the Content pane can have up to four badge icons.

Infoset name	Member names	
cameraRaw	rawSupportType	Identifies the extent to which this file can be handled by the Camera Raw plug-in. One of: 0 (the file is of a type that is not handled by the plug-in) 1 (the file is in a camera-raw format) 2 (the file is in JPEG or TIFF format)
children	children	An Array of Thumbnail Objects representing the child nodes of a container node. Container nodes must update their child node lists.
fullsize	fullsize	A BitmapData Object representing the pixels for the file's full-size preview thumbnail.
icon	bitmap	A BitmapData Object representing the pixels for the node's icon.
linktarget	linkTarget	A string containing the full path to the target, if this node is a link.
metadata	metadata	The metadata blob for the file, if applicable.
preview	hasHighQualityPreview	True if the file contains a high-quality preview image.
	preview	A BitmapData Object representing the pixels for the file's preview thumbnail image.
thumbnail	hasHighQualityThumbnail	True if the file contains a high-quality thumbnail image.
	thumbnail	A BitmapData Object representing the pixels for the file's thumbnail image.

InfosetMemberDescription Object

Associates a data type with a single node-data value for Adobe Bridge nodes. Each node-data value is a member of an Infoset Object associated with an ExtensionHandler Object.

The `name` becomes a property of the parent Infoset Object, which provides access to a data value of this `type`. For example, to access a `color` value in `myInfo` in thumbnail `t1`, where the `myInfo` set is managed by `myExtension`, use:

```
t1.myExtension.myInfo.color
```

InfosetMemberDescription constructor

Create the object with the `new` operator:

```
new InfosetMemberDescription ( name, type )
```

The parameters set the corresponding properties.

InfosetMemberDescription properties

`name`	String	The name of this value, which becomes a property of the parent Infoset Object. Must be a valid JavaScript identifier containing no colons or special symbols. Read/write.

`type`	String	The data type for values accessed through the `name` property of the parent Infoset Object. Read/write. One of:
		`Boolean`
		`String`
		`Integer`
		`Icon` (32x32)
		`BitmapData` (a BitmapData Object)
		`SizeInBytes`
		`Date`
		`Array` (an array of *type* for any of these types: nested arrays are not allowed)

ModalOperator Object

Encapsulates a synchronous operation. Performs a task that blocks the main thread, and provides its own user interface.

See Operator Class for basic properties and methods. For this object, the start() method yields the main thread to the extension. The getType() method should return `modal`.

ModalOperator constructor

To create a new object, use the `new` operator:

```
new ModalOperator (sources, target)
```

type	The operator type, "`modal`".
sources	An array of Thumbnail Objects that the operation acts upon.
target	A target Thumbnail Object for the operation.

Operand Object

For use in node searches. An array of these objects is kept in the operands property of a SearchCriteria Object. They are used to populate the right-side field in the line that corresponds to the criterion in the Find dialog (values to be compared against). If there is more than one, the field displays a drop-down list.

Operand object constructor

Create the object using the `new` operator:

```
new Operand(valueName, displayName);
```

Parameters set corresponding properties.

Operand properties

valueName	String	The operand value; that is, a possible value of the searchField property of the SearchCriteria Object. Read-write.
displayName	String	Optional. The localized display name for the corresponding field in the Find dialog. If not supplied, the `valueName` is used. Read-write.

Operator Class

When implementing a node-handling extension, you can use `Operator` objects to implement and monitor long-running operations, such as file-system interactions, or operations that require a user interface. An operation can be modal, blocking the main thread until it is complete, or it can spawn a background task that provides feedback and allows interaction through a Progress bar and other dialogs that Adobe Bridge provides.

You define certain methods for a node-handler's ExtensionHandler Object and ExtensionModel Object to create and return an `Operator` object, which actually implements the operation. The model method returns immediately. See Long-running handler operations and Long-running model operations.

The `Operator` class is a base class for two types of operator:

- ModalOperator Object: Blocks the main thread and must provide any desired user interface.
- ProgressOperator Object: Spawns a background task that can perform operations incrementally, while occasionally notifying Adobe Bridge of changes that require an update to the Adobe Bridge-supplied UI.

To start the operation, your node handler (or Adobe Bridge) passes the returned `Operator` object to `app.enqueueOperation()`. This in turn calls the start() method defined in the object.

- For a `ModalOperator`, the `start()` method returns when the operation is complete.
- For a `ProgressOperator`, your `start()` method should begin the background task and return. Adobe Bridge displays the Progress bar and resumes activity on the main thread. When the background task notifies Adobe Bridge of a change by calling `app.operationChanged()`, Adobe Bridge queries the `Operator` object and updates the Adobe Bridge-supplied user interface.

You can use `app.scheduleTask()` to schedule the execution of the operation, and make periodic progress updates. Note that Adobe Bridge does not update the UI for a `ProgressOperator` unless and until you call `app.operationChanged()`.

The `Operator` class is a template; it does not implement any state or behavior. When creating an operator object, you must implement the interface described here, in order to perform the desired operation, and to provide Adobe Bridge with expected information about the progress and result of the operation.

Operator common properties

`cancelRequested`	Boolean	When `true`, the user has requested that the operation be canceled.

`conflictType`	String	The type of the current file-system conflict encountered during the operation. One of:
		`none`
		`userConfirmationRequired`
		`fatal`

conflictMessage	String	A string describing the current file-system conflict that prevents the operation from being performed. Can identify one of the preset Adobe Bridge error messages, or can be an arbitrary descriptive string.
		
		Preset messages are identified by the following string values:
		
		`none`
		`deleteFile`
		`deleteMultipleFiles`
		`deleteReadOnlyFile`
		`moveReadOnlyFile`
		`readOnlyFileExists`
		`fileExists`
		`fileIsBusy`
		`targetFolderExists`
		`fatalErrorSameFile`
		`fatalErrorSameFolder`
		`fatalErrorMoveToChild`
		`fatalErrorSourceNotAvailable`
		`fatalErrorStorageFull`
		`fatalErrorSourceAccessDenied`
		`fatalErrorTargetAccessDenied`
		`fatalErrorUnknown`
		`noXMPSupport`
		`undoDelete`
		`messageCustom`

description	String	A description of the operation, suitable for display.
errorTarget	Thumbnail	When operationStatus is `inError`, the problematic Thumbnail Object.
newNames	Array of String	When sources has a value, an array of the same length containing the new names to be assigned to the source Thumbnail Objects after they are transferred to the target.

operationStatus	String	The status of the operation with respect to the immediate action. Read/write. One of:
		
		`incomplete`
		`inCancellation`
		`inConflict`
		`inError`
		`succeeded`
		`cancelled`
		`failed`

percentageComplete	Number	How much of the operation has currently been completed, in a float value with the range [0, 1]. Read/write. Also returned by getPercentageComplete().

processingStatus	String	The current overall status of the operation with respect to Adobe Bridge; that is, whether the operation has begun, is still going on, has been paused by the user, or has finished. Read/write. Also returned by getProcessingStatus(). One of:
		
		`notStarted`
		`inProgress`
		`awaitingResume`
		`completed`

progressMessage	String	A description of the current state of the operation, suitable for display. Read/write. Also returned by getProgressMessage().

resolvePolicy	String	How to apply the conflict-resolution method (resolveMethod). This is for the developer's information in a `ModalOperator`; Adobe Bridge does not check it. One of:
		
		`applyForOneConflictOnly`
		`applyToAllConflicts`

resolveMethod	String	How to resolve file-system conflicts. This is for the developer's information in a `ModalOperator`; Adobe Bridge does not check it. One of:
		
		`abort`
		`noOverride`
		`override`
		`overrideConditionally`

result	Object	An optional result for an operation, such as the path that results from a `createNewContainer()` operation.
sources	Array of Thumbnail	A set of Thumbnail Objects that the operation acts upon.
target	Thumbnail	A target Thumbnail Object for the operation.
timeout	Number	A number of milliseconds to wait before aborting the operation. Default is 0, meaning that the operation does not time out.

Operator functions

getConflictInfo()
obj.getConflictInfo ()
	Implement a method that returns a description of a file-system conflict that prevents the operation from being performed on the current thumbnail.
	
	The returned string can identify one of the preset Adobe Bridge error messages, or can be an arbitrary descriptive string suitable for display in a conflict-resolution dialog. Preset messages are identified by the following string values:
	
	`none`
	`moveReadOnlyFile`
	`targetFolderExists`
	`readOnlyFileExists`
	`fileExists`
	`fatalErrorSameFile`
	`fatalErrorSameFolder`
	`fatalErrorMoveToChild`
	`fatalErrorSourceFileNotAvailable`
	`fatalErrorStorageFull`
	`fatalErrorSourceAccessDenied`
	`fatalErrorTargetAccessDenied`
	`fatalErrorUnknown`
	`deleteFile`
	`deleteMultipleFiles`
	`noXMPSupport`
	`fileIsBusy`
	`undoDelete`
	`messageCustom`

getPercentageComplete() *obj*.getPercentageComplete ()	Implement a method that returns the percentage of the operation that has currently been completed, for use in displaying the Progress dialog. Adobe Bridge invokes this when it needs to update the Progress bar. Return a number in the range [0...1].
getProcessedNodeCount() *obj*.getProcessedNodeCount ()	Implement a method that returns the number of source nodes that have been processed so far. Return a number.

getProcessingStatus()
obj.getProcessingStatus ()
	Implement a method that returns the current overall status of the operation with respect to Adobe Bridge; that is, whether the operation has begun, is still going on, has been paused by the user, or has finished.
	
	Return one of the following string values:
	
	`notStarted`
	`inProgress`
	`awaitingResume`
	`completed`

getProgressMessage() *obj*.getProgressMessage ()	Implement a method that returns a message suitable for display in the Progress dialog. Return a string.
getTotalBytesTransferred() *obj*.getTotalBytesTransferred ()	Implement a method that returns the current number of bytes that have been transferred to the target in the course of this operation. Return a number.
getTotalNodeCount() *obj*.getTotalNodeCount ()	Implement a method that returns the total number of source nodes to be operated on. Return a number.

getType()
obj.getType ()
	Implement a method that returns the subclass type of this operator.
	
	Return a string, one of:
	
	`modal`
	`progress`
	`background`
	`progressBackground`

resolveConflict() *obj*.resolveConflict (*method*, *policy*)	Implement a method that resolves a file-system conflict, as identified by the conflictType and conflictMessage values. Adobe Bridge invokes this after the user makes selections in a conflict-resolution dialog, passing in the user's choices. Return `undefined`.

method How to resolve the conflict. One of:

`noOverride`—Do not perform the current action, but continue with the operation. Corresponds to **Skip** in the conflict-resolution dialog.
`override`—Make another attempt to perform the current action. Corresponds to **Replace** in the conflict-resolution dialog.
`overrideConditionally`—Use an extension-defined default style of resolving the conflict. Corresponds to **Auto-resolve** in the conflict-resolution dialog.
`abort`—Does not perform the action for the current thumbnail, and terminates the operation. Corresponds to **Cancel** in the conflict-resolution dialog.

policy How to apply the conflict resolution method. Corresponds to the checked state of **Apply to all** in the conflict-resolution dialog. One of:

`applyForOneConflictOnly`—Resolve as specified for the current action, but request user input again if the same type of conflict occurs again.
`applyToAllConflicts`—Resolve as specified for the current action, then resolve with this method again if the same type of conflict occurs again.

resume() *obj*.resume ()	Implement a method that restarts the operation after it has been stopped by user interaction. Return `true` if the operation has been successfully restarted.
start() *obj*.start ()	Implement a method that initiates the operation. Adobe Bridge invokes this after the operator has been enqueued. • For a modal operation, the method should return when the operation is complete. • For a non-modal operation, the method should begin the background task and return. Return `undefined`.
stop() *obj*.stop ()	Implement a method that terminates the operation. Adobe Bridge invokes this after the operation has been stopped by user interaction. Return `undefined`.

ProgressOperator Object

Encapsulates an operation that performs a background task, while Adobe Bridge displays a Progress bar. It can do so incrementally, periodically notifying Adobe Bridge of the current status. See Operator Class for the inherited properties and methods.

For this object, the getType() method should return `progress`. The start() method should spawn a thread to perform the operation and return immediately. Adobe Bridge displays a Progress bar, and resumes activity on the main thread.

When the background thread updates the status in any way that affects the display, it must pass this object to `app`.operationChanged(). Adobe Bridge queries this object in order to update the Progress dialog or display the Adobe Bridge-supplied error handling or resolution conflict dialogs.

ProgressOperator constructor

To create a new object, use the `new` operator:

```
new ProgressOperator (type, sources, target)
```

type	The operator type, "`progress`".
sources	An array of Thumbnail Objects that the operation acts upon.
target	A target Thumbnail Object for the operation.

Rank Object

For use in node searches. A SearchDefinition Object can limit the number of results to return, and, if results are limited, it can define a set of possible properties to use in ranking results. Adobe Bridge sorts result nodes by the value of the chosen rank property, and returns no more than the maximum number of result nodes with the highest rank values. When the result is displayed, the view sorts the nodes again using its sorting criteria.

The attribute name and display name of a property used for ranking are encapsulated in a `Rank` object.

An array of these objects kept in the ranks property of a SearchDefinition Object. They are used to populate the Rank field that corresponds to the definition in the Find dialog. If there is more than one, the field displays a drop-down list. The rank that the user selects becomes the rankField value in the SearchSpecification Object.

Rank object constructor

Create the object using the `new` operator:

```
new Rank(valueName, displayName);
```

Parameters set corresponding properties.

Rank properties

valueName	String	The property name for the ranking property. Read-write.
displayName	String	Optional. The localized display name for the corresponding field in the Find dialog. If not supplied, the `valueName` is used. Read-write.

Scope Object

Identifies a scope modifier to use in node searches among handled nodes. The modifier can expand or limit the scope of the search from the original target node. The scope value and usage is defined entirely by the getSearchDefinition() method of the node's ExtensionModel Object.

An array of these objects, kept in the scopeSpecifiers property of a SearchDefinition Object, is used to populate the Find dialog. The box displays a check box for each possible scope extension or limitation. When the user selects a scope, its name becomes a value of scopeSpecifiers in the resulting SearchSpecification Object object.

Scope object constructor

Create the object using the `new` operator:

```
new Scope(valueName, displayName);
```

Parameters set corresponding properties.

Scope properties

valueName	String	The unique identifying name for the scope modifier. Read-write.
displayName	String	Optional. The localized display name for the corresponding field in the Find dialog. If not supplied, the `valueName` is used. Read-write.

SearchCondition Object

Defines a specific condition that must be met for a node to match a search. The Find dialog returns a SearchSpecification Object for a specific search, which contains a list of these objects in the conditionList property; each object corresponds to the user's selection in one line of the Criteria box in the dialog. The `SearchSpecification.conjunction` controls whether all or any of the conditions must be met.

Your node handler can define possible search criteria for your nodes by creating SearchCriteria Objects and passing them in the SearchDefinition Object created by the model's getSearchDefinition() method.

Each condition specifies a property associated with a node (the searchField), whose value is compared to a selected operand value, using a selected operator such as "equals." Operators are predefined. Some operators, such as "exists," do not require an operand.

SearchCondition object constructor

Create the object with the `new` operator:

```
new SearchCondition(searchField, operatorType, operand);
```

Parameters set corresponding properties.

SearchCondition properties

searchField	String	The name of some property associated with the search node, typically a metadata property or a member of an Infoset Object associated with handled nodes. This corresponds to the left side of a line in the Criteria box of the Find dialog. Read/write.

operatorType	String	The comparison operator for the search. This corresponds to the middle field of a line in the Criteria box of the Find dialog. Read/write. One of:
		
		`exists`
		`doesNotExist`
		`equals`
		`notEquals`
		`less`
		`lessThanOrEqual`
		`greater`
		`greaterThanOrEqual`
		`contains`
		`doesNotContain`
		`startsWith`
		`endsWith`

operand	String	The value to compare against the value of the search field in each node. This corresponds to the right side of a line in the Criteria box of the Find dialog. Read/write.

SearchCriteria Object

Encapsulates one search criterion for a search among handled nodes. Your node handler can define possible search criteria for your nodes by creating these objects and passing them to Adobe Bridge in a SearchDefinition Object, during the call to the getSearchDefinition() method of the node's ExtensionModel Object.

Each object corresponds to one line in the Criteria box of the Find dialog.

- The left side is a property associated with possible matching nodes, called the *search field*.
- The middle value is the comparison *operator*.
- The right side is the comparison value, or *operand* (some operators, such as "exists," do not require an operand).

For each node in the scope, a search that uses a selected criterion matches the selected search-field value against the operand using the selected comparison operator. This object specifies the left and right sides. By default, all of the predefined operators are displayed for user selection. You can use this object to limit which of these operators are available for selection.

The user's choices in the dialog are returned to Adobe Bridge in a set of a SearchCondition Objects contained in a SearchSpecification Object.

SearchCriteria object constructor

Create the object with the `new` operator:

```
new SearchCriteria (searchField, operandType,
                    *searchFieldDisplay, *operands);
```

Parameters set corresponding properties.

SearchCriteria properties

operands	Array of Operand	Optional. One or more Operand Objects used to populate the drop-down list for the right-side field of this line in the Find dialog. This allows you to specify a closed list of possible values to match against in the search field.

operandType	String	The data type of the operand values. Determines the description that appears in the operand field in the absence of a closed list of `operands`. The description is the expected format for `Date` values, otherwise generally "Enter text".
		
		One of:
		
		`String`
		`Number`
		`Float`
		`Date`
		`Boolean`

operatorTypesToDisable	Array of String	Optional. A set of predefined operator strings that are *not* displayed for selection.
		
		Predefined operators are:
		
		`exists`
		`doesNotExist`
		`equals`
		`doesNotEqual`
		`less`
		`lessThanOrEqual`
		`greater`
		`greaterThanOrEqual`
		`contains`
		`doesNotContain`
		`startsWith`
		`endsWith`

searchField	String	A search field, the name of some property associated with the search node, typically a metadata property or a member of an Infoset Object associated with handled nodes. The value of the selected search field is compared to the selected operand, using the selected comparison operator.
searchFieldDisplay	String	Optional. A localized display name for the search field, displayed in the Find dialog. Default is the `searchField` value.
searchFieldSort	Boolean	Optional. When `true`, search field display names are sorted alphabetically in the Find dialog. Default is `false`.

SearchDefinition Object

Provides a way for Adobe Bridge extensions to specify how the Find dialog should be populated for a search among handled nodes. It specifies possible search criteria, as well as result scope and ranking criteria.

If the user invokes the Find dialog for a handled node, the dialog calls the model's getSearchDefinition() method. Your node-handling extension must define this method to return a `SearchDefinition` object that can be used to populate Find dialog.

When a user clicks Find in the Find dialog, Adobe Bridge uses the dialog selections to create a SearchSpecification Object, which, together with a target node, specifies a search.

SearchDefinition object constructor

Create the object with the `new` operator:

```
new SearchDefinition (criteriaList, defaultResultsLimit,
                      *ranks, *scopeSpecifiers);
```

Parameters set corresponding properties.

SearchDefinition properties

criteriaList	Array of SearchCriteria	A collection of possible SearchCriteria Objects to use for this search, used to populate the Criteria box in the Find dialog.
defaultResultsLimit	Number	If non-zero, the Find dialog offers choices to limit the result set to certain sizes, and the choice defaults to this value.
ranks	Array of Rank	Optional, a set of Rank Objects to use only if the search can limit results. Read-write. Used to populate the Rank list in the Results section of the Find dialog.
scopeSpecifiers	Array of Scope	Optional, one or more Scope Objects. Your search can use these to extend or limit the scope of the search. Read-write. The Results section of the Find dialog displays a check box for each scope modifier.

SearchDetails Object

An object that encapsulates information about how a search result node was generated by a node-handler's getBridgeURIForSearch() method. Returned by an `ExtensionModel`.getSearchDetails() method for a search-result node.

SearchDetails object constructor

Create the object with the `new` operator:

```
new SearchDetails (searchSpecification, searchTargetUri);
```

Parameters set corresponding properties.

SearchDetails properties

`searchCriteria`	SearchSpecification	A SearchSpecification Object that was used to generate this search result. Read/write.
`searchTargetUri`	String	The Bridge URI for the search target node that was used to generate this search result. Read/write.

SearchSpecification Object

Encapsulates a specific search among member nodes of a target container node. The object contains a set of conditions to be met in order for a node to match, and instructions for how to return matching nodes.

Adobe Bridge creates this object from user selections in the Find dialog. For a search that involves handled nodes, Adobe Bridge passes the search specification to the handler's getBridgeURIForSearch() method. Your handler can either save that object, or recreate one to return from the getSearchDetails() model method of the search-result container node.

SearchSpecification object constructor

Create the object with the `new` operator:

```
new SearchSpecification (conditionList, conjunction, maximumResults,
                         rankOrdering, rankField, scopeSpecifiers);
```

Parameters set corresponding properties.

SearchSpecification properties

conditionList	Array of SearchCondition	A collection of SearchCondition Objects to use for this search. Each object specifies a *search field*, which identifies a property associated with a node, a comparison operator, such as "exists" or "equals", and an *operand*, the value to compare with the search field value (if the operator requires a comparison value).
conjunction	String	The search conjunction, `and` or `or`, as selected in the Find dialog. When it is `and`, all conditions must succeed for a node to match. When it is `or`, the success of any condition results in a match.
maximumResults	Number	The maximum number of result nodes to return from the search. The search halts after this number of matches are found.
rankOrdering	String	The ordering style, one of `ascending` (the default) or `descending`.
rankField	String	The name of a Rank Object, as specified for a SearchDefinition Object. If the number of results are limited, results are sorted on the named attribute value, and the maximum number of result nodes with the highest rank values are returned. The returned results are again sorted by the view's sorting criteria upon display.

`scopeSpecifiers`	Array of String	One or more Scope Object names, as specified for a SearchDefinition Object. Each scope modifier can expand or limit the original scope defined by the target node. The scope value and usage is defined entirely by your getSearchDefinition() model method implementation.

SortCriterion Object

Provides a way for Adobe Bridge extensions to specify how handled nodes can be sorted. Sorting compares the values of a property associated with the displayed nodes. This object identifies that property, which can be in handler-defined node data (that is, defined in an Infoset Object), or defined in embedded XMP metadata.

When Adobe Bridge enters a container node, it calls the getSortCriteria() method of the node's ExtensionModel Object, which returns a list of these objects. The method can supply a completely new list, or can get the default list from `app`.defaultSortCriteria and modify it, append to it, or return it unchanged.

You can apply a sorting criterion to currently displayed nodes by referencing a `SortCriterion` object from the `Document`.sorts property.

SortCriterion object constructor

Create the object with the `new` operator:

```
new SortCriterion(name, type, xmpNamespace, xmpUri, *displayName)

new SortCriterion(name, type, infosetMember, *displayName)
```

Parameters set corresponding properties.

SortCriterion properties

displayName	String	Optional. A localized display name for this sorting criterion. Used as a label for the Sort menu and Filter palette flyout menu. If not assigned, `name` is displayed. Read-write.
infosetMember	String	The name of an Infoset Object and InfosetMemberDescription Object by which to sort. Read/write. For example, "mySet.color".

name	String	The unique identifying name of this sort criterion. The name can be:
		`user`
		`name`
		`date-created`
		`date-modified`
		`label`
		`rating`
		`file-size`
		`document-kind`
		`keywords`
		`dimensions`
		`resolution`
		`color-profile`

type	String	The data type of the criterion property. Read only. One of:
		`string`
		`date`
		`number`
		`dimensions`
		`resolution`
		`colorProfile`
		`user`

xmpNamespace	String	The namespace portion of an XMP property by which to sort. Read/write.
xmpUri	String	The URI key portion of an XMP property by which to sort. Read/write.

3 External Communication Tools

Adobe Bridge offers the Web Access library, which supplies tools for communicating with other computers or the Internet using standard protocols. The Web Access library defines:

- The FtpConnection object, which supports FTP and SFTP communication protocols.
- The HttpConnection object, which supports HTTP and HTTPS communication protocols.

Your script must load the platform-compiled Web Access library as an `ExternalObject` in order to use these objects. See 'Loading the Web Access library' on page 159.

Loading the Web Access library

To use the FtpConnection object or HttpConnection object, you must dynamically load the Web Access library into Adobe Bridge as an `ExternalObject`. This library is compiled as a shared library; a DLL in Windows, a bundle or framework in Mac OS.

For example, use the following JavaScript code:

```
if ( !ExternalObject.webaccesslib ) {
    ExternalObject.webaccesslib = new ExternalObject('lib:webaccesslib');
}
```

The location of the compiled library files is determined by the operating system.

- In Windows, the DLLs reside in the executable directory.
- In Mac OS, bundles and frameworks are loaded from the `@executable/../Frameworks/` directory. Use the layout of bundles and Frameworks from the `shellframework` sample application as a template.

For more information on loading compiled libraries into JavaScript, see the *JavaScript Tools Guide*.

FtpConnection object

This class supports the FTP and SFTP protocols for file transfer. The object allows you to send data to or receive data from an FTP server, synchronously or asynchronously.

To use the `FtpConnection` object, you must load the Web Access library (`webaccesslib`) into JavaScript as an `ExternalObject`. See "Loading the Web Access library" on page 159.

Using File objects with the FtpConnection object

Typically, you create a `File` object for use with your `FtpConnection` object. The get() and put() operations automatically open the file for read and write, respectively, if you have not done so explicitly. The default transfer mode is binary.

- To transfer binary files to the server, use code such as the following:

```
var file = new File('/c/Photo.jpg') ;
var ftp = new FtpConnection('ftp://server') ;
ftp.put(file,'Photo.jpg') ;
ftp.close() ;
file.close() ;
```

- Similarly, to transfer binary files from the server:

```
var file = new File('/c/Photo.jpg') ;
var ftp = new FtpConnection('ftp://server') ;
ftp.get('Photo.jpg',file) ;
ftp.close() ;
file.close() ;
```

The operations do not automatically close the file. This allows you, for example, to use get() to copy many files to a single file on your local file system. For example:

```
var file = new File("/c/archive.bin") ;
ftp.get("a.txt",file) ;
ftp.get("c.txt",file) ;
file.close() ;
```

Open files are eventually closed by the JavaScript garbage collector when there are no remaining JavaScript references.

ExtendScript supports many file filters; see the *JavaScript Tools Guide* for details.

Synchronous and asynchronous operation

Two properties of the `FtpConnection`, sync and async, control whether get() and put() operations are performed synchronously or asynchronously. The property values are tied together, and are mutually exclusive. You can set either one, and the other is automatically toggled to the opposite value.

When the property sync is set to true (the default), the connection operation blocks the main thread. All operations must be completed before your script continues.

Example: synchronous operation (blocking)

```
var ftp = new FtpConnection("ftp://localhost") ;
var file = new File("here.text") ;
// synchronous mode is the default
ftp.get("remote.txt",file) ;
// the operation simply returns when complete
file.close() ;
ftp.close() ;
```

When the property sync is set to false (or async set to true), the connection operation occurs in a background thread while your script continues to do other work. The background thread sets the property isComplete to true when the current operation has finished. If the operation times out, isComplete is set to true and error is set to `FtpConnection.errorTimeout`.

Only a single connection to the FTP server is allowed; you cannot start two operations on the server at the same time. If you do attempt to do so, error is set to `FtpConnection.errorCommandActive` to indicate that the connection is waiting to complete a previous operation.

You can define a callback function in the onCallback property, that checks the completion status of an asynchronous call, and closes the file and connection when it is done. Use the pump() function to call that function periodically from the main thread. Typically, a callback function displays and updates a dialog that shows the progress, and allows the user to cancel an asynchronous operation before its completion; your callback can accomplish this using cancel().

Example: asynchronous operation (non-blocking)

```
var file = new File("here.text") ;
ftp.sync = false ; // set asynchronous mode
// define callback to check status and close when complete
ftp.onCallback = function(reason,p_log,total) {
    if ( this.isComplete ) {
        file.close();
    }
}
// the operation spawns a new thread and returns
ftp.get("remote.txt",file) ;
// at some time and occasionally
// update progress by calling ftp.onCallback()
ftp.pump() ;
```

FtpConnection object reference

This section provides details of the `FtpConnection` object's properties and functions.

FtpConnection object constructor

```
[new] FtpConnection ( [url] );
```

url	Optional. The URL to which to connect. The URL specifies the protocol; for example:
	`ftp://localhost`
	`sftp://localhost`
	If not provided, you must set the object's `url` property.

FtpConnection object properties

active	Boolean	When true, the connection is active, not passive. Sets `passive` to false. See the FTP standard (RFC 959) for details. Read-write.
ascii	Boolean	When true, the encoding used to transmit data is ASCII. Default is false. When set to true, sets `binary` to false. Read-write.

async	Boolean	When true, the connection is asynchronous. Operations spawn a thread and return immediately to the main thread. The background thread sets `isComplete` to true when the current operation has finished. If the operation times out, `isComplete` is set to true and `error` is set to `errorTimeout`. Default is false. When set to true, sets `sync` to false. Read-write.
binary	Boolean	When true, the encoding used to transmit data is binary. Default is true. When set to true, sets `ascii` to false. Read-write.
cd	String	Sets the current directory when the connection is open. Default is `undefined`. Read-write. Setting to a directory that does not exist causes a JavaScript error, and sets the `error` and `errorString` properties of the object.
dates	Array of Date	The dates of the files in the current directory. Set by the ls() call. An array corresponding to the files array, where each member is a JavaScript `Date` object (as returned by date() for an individual file). Default is `undefined`. Read only.

error	Number	The most recent error encountered in the course of connecting or executing the operation. All functions set this value before returning. A constant value, one of:
		
		`FtpConnection.errorNoError`
		`FtpConnection.errorCommandActive`
		`FtpConnection.errorUnknownException`
		`FtpConnection.errorUnknown`
		`FtpConnection.errorOutOfMemory`
		`FtpConnection.errorCancelled`
		`FtpConnection.errorUnknownHost`
		`FtpConnection.errorConnectFailed`
		`FtpConnection.errorTimedOut`
		`FtpConnection.errorLoginFailed`
		`FtpConnection.errorProtocolError`
		`FtpConnection.errorUnknownProtocol`
		`FtpConnection.errorChannelOpen`
		`FtpConnection.errorChannelClosed`
		`FtpConnection.errorOperationPending`
		`FtpConnection.errorBadParameters`
		`FtpConnection.errorResourceExists`
		`FtpConnection.errorResourceDoesntExist`
		`FtpConnection.errorResourceInUse`
		`FtpConnection.errorAccessDenied`
		`FtpConnection.errorOutOfDisk`
		`FtpConnection.errorLocalIoError`
		`FtpConnection.errorRemoteIoError`
		`FtpConnection.errorNotEmpty`
		`FtpConnection.errorNotDirectory`
		`FtpConnection.errorNotFile`
		`FtpConnection.errorBadPathname`
		`FtpConnection.errorNotImplemented`
		`FtpConnection.errorNotLocked`
		`FtpConnection.errorLocked`
		`FtpConnection.errorMethodNotAllowed`
		`FtpConnection.errorResourceRedirected`
		
		Default is `errorNoError`. Read only.

errorString	String	A description of the most recent error encountered in the course of connecting or executing the operation. Default is “OK.” Read only.
files	Array of String	The files in the current directory. Set by the ls() call. Default is `undefined`. Read only.

flags	Array of Number	The access permissions and types for the files in the current directory. Set by the ls() call. An array corresponding to the files array, where each member is a logical OR of these constant values:
		
		`FtpConnection.flagOtherExecute`
		`FtpConnection.flagOtherWrite`
		`FtpConnection.flagOtherRead`
		`FtpConnection.flagGroupExecute`
		`FtpConnection.flagGroupWrite`
		`FtpConnection.flagGroupRead`
		`FtpConnection.flagOwnerExecute`
		`FtpConnection.flagOwnerWrite`
		`FtpConnection.flagOwnerRead`
		`FtpConnection.flagDirectory`
		`FtpConnection.flagSymLink`
		
		Default is `undefined`. Read only.

isComplete	Boolean	When true, the operation is completed. See "Synchronous and asynchronous operation" on page 160. Default is true. Read only.
isOpen	Boolean	When true, the connection to the FTP server is open. Default is false. Read only.
onCallback	Function	Optional. A callback function to the connection thread for asynchronous mode. The object stores progress messages from operation thread; to check on the progress, call pump() on the main thread. The `pump()` method invokes this function on each stored message, passing the operation status at that point. Within the call, you can use `this.`cancel() to halt the asynchronous operation. Read-write. The function must return `undefined`, and take these arguments: `function(`*`reason,p_log,total`*`) { }` *`reason`*: The type of progress message. One of: `FtpConnection.reasonStart`: The transfer started. `FtpConnection.reasonComplete`: The transfer is complete. `FtpConnection.reasonFailed`: The transfer failed. `FtpConnection.reasonProgress`: The transfer is in progress. `FtpConnection.reasonLog`: The operation generated a log message. *`p_log`*: Depends on the reason for the message: For a log message, the message string. For a progress message, the current number of bytes transferred. Otherwise, `undefined`. *`total`*: Depends on the reason for the message: For a progress message, the total number of bytes to be transferred. Otherwise `undefined`.

passive	Boolean	When true, the connection is passive, not active. See the FTP standard (RFC 959) for details. When set to true, sets `active` to false. Default is false. Read-write.
password	String	The connection password for the FTP server. Set this to override the password given in the URL. Default is `undefined`. Read-write.
proxy	String	Not used.
renamestyle	String	The rename() function takes a source and destination path and file name, so that it can both move and rename the source object. You can normally specify the source and destination without a path or with a relative path (such as `../myfile.htm`). The function interprets the path as relative to the current working directory. This typical case is handled by the default value for this property, `"style1"`. If you connect to an FTP server that cannot parse the "../.." notation, change this value to `"style2"`, and specify both source and destination with absolute paths.
sizes	Array of Number	The sizes of the files in the current directory. Set by the ls() call. An array corresponding to the files array, where each member is a number of bytes. Default is `undefined`. Read only.
sync	Boolean	When true, the connection is synchronous. Operations block the main thread and return when complete. Default is true. When set to true, sets `async` to false. Read-write.
timeout	Number	An integer, the number of seconds to continue attempting the operation before completing with the error message `errorTimeout`. Default is 5. Read-write.
url	String	The URL of the FTP server, and optionally the port, to which to connect. This includes the protocol (FTP or SFTP), and can include a login user name and password in this format: `[s]ftp://[[username:]password@]server[:port]` This string must use escape sequences for special characters, such as %20 for space and %40 for @. Default is `undefined`. Read-write.
username	String	The connection user name for the FTP server. Set this to override the user name given in the URL. Default is `undefined`, for anonymous FTP. Read-write.

FtpConnection object functions

All functions set the `error` property to indicate the status of the operation when completed (`errorNoError` on success).

`cancel()`
`ftpObj``.cancel ();`

Cancels the current operation, if it is being performed asynchronously. See "Synchronous and asynchronous operation" on page 160.

Returns true on success.

`close()`
`ftpObj``.close ();`

Terminates the open connection. Deleting the object also closes the connection, but not until JavaScript garbage-collects the object. The connection might stay open longer than you wish if you do not close it explicitly. There are a limited number of open connections available; failing to close connections can make you unable to open a new one.

Returns true if the connection was closed, false on I/O errors.

`chmod()`
`ftpObj``.chmod (`*`remote`*`[, `*`flags`*`]);`

`remote` String. The name of the remote file-system object.

`flags` Optional. The new permissions. A logical OR of the flags constants.

Changes the permissions and/or type of a file-system object on the FTP server.

Returns true on success.

`cwd()`
`ftpObj``.cwd (`*`remote`*`);`

`remote` String. The name of the remote directory.

Changes the current directory on the FTP server.

Returns true on success.

`date()`
`ftpObj``.date (`*`remote`*`);`

`remote` String. The name of the remote file.

Retrieves the date information for a file-system object on the FTP server.

Returns an array of three JavaScript `Date` objects, for the creation, modification, and most recent access dates of the given file. If a date is unavailable, the corresponding array member is `undefined`. See also dates.

Returns false if all dates are unavailable; as when the file-system object does not exist, is a directory, or is a link that cannot be resolved.

del()
```
ftpObj.del (remote);
```

remote String. The name of the remote file-system object.

Deletes a file-system object on the FTP server.

Returns true on success.

exists()
```
ftpObj.exists (remote);
```

remote String. The name of the remote file-system object.

Reports whether a file-system object exists on the FTP server.

Returns true if the object exists on the server, false if it does not exist or is a link that cannot be resolved.

get()
```
ftpObj.get (remote, file);
```

remote String. The name of the remote file containing data to transfer.

file A `File` object, the local file in which to receive the data.

Transfers date from a file on the FTP server to a local file.

Returns true on success.

isDir()
```
ftpObj.isDir (remote);
```

remote String. The name of the remote file-system object.

Reports whether a file-system object on the FTP server is a directory.

Returns true if the file is a directory on the server, false otherwise.

ls()
```
ftpObj.ls ();
```

Retrieves information about the current directory, and returns it in the files, dates, sizes, and flags properties of this object.

Returns true on success, false on I/O errors.

mkdir()
```
ftpObj.mkdir (remote);
```

remote String. The name of the new remote directory.

Creates a directory on the FTP server.

Returns true on success.

open()
```
ftpObj.open ();
```

Opens the FTP connection explicitly. This in not typically needed; calling a function to perform an operation opens the connection if necessary.

Returns true if the connection was successfully opened, false on I/O errors.

pump()
```
ftpObj.pump ();
```

Executes the callback procedure defined in onCallback on all progress messages that have been received since the last call to this function.

Use this function in the main thread to invoke the callback, in order to check on the progress of an asynchronous operation. It is not required, however; the asynchronous operation continues to progress on the spawned thread, whether or not you make this call.

Returns true on success, false on I/O errors.

put()
```
ftpObj.get (file, remote[, putMode]);
```

file A `File` object, the local file containing data to transfer.

remote String. The name of the remote file in which to receive the data.

putMode Optional. The style of transfer, one of these constants:

`FtpConnection.putModeTruncateOrCreate` (default) — Allows creation of the target file, and truncates an existing file to the size of data written. Does not lock the target file.

`FtpConnection.putModeExclusive` — Locks the target file during the write operation.

Transfers data from a local file to a file on the FTP server. Overwrites the target file, if it already exists.

Returns true on success.

rename()
```
ftpObj.rename (from, to);
```

from String. The path and file name of the source object in the remote file system.

to String. The path and file name of the destination object in the remote file system.

Moves and changes the name of a file-system object on the FTP server. The path can be absolute, or (in most cases) relative to the current working directory; see renamestyle.

Returns true on success.

rmdir()
```
ftpObj.rmdir (remote);
```

remote String. The name of the remote directory.

Deletes a directory on the FTP server.

Returns true on success.

size()

```
ftpObj.size (remote);
```

remote String. The name of the remote file-system object.

Retrieves the size of a file-system object on the FTP server.

Returns the number of bytes in the file, or -1 if there is no such file, or if the object is a directory or a link that cannot be resolved.

HttpConnection object

Supports the HTTP and HTTPS protocols for Internet communication. The object allows your script to open a connection to a remote computer that acts as an HTTP server, send an HTTP request, and receive the response.

To use the `HttpConnection` object, you must load the Web Access library (`webaccesslib`) into JavaScript as an `ExternalObject`. See "Loading the Web Access library" on page 159.

The `HttpConnection` object can open only one connection to the internet. If you call execute() before the current operation is complete (status is `HttpConnection.statusCompleted`), the current operation is terminated.

Requests and responses

The method property of the `HttpConnection` object determines the type of operation: GET, PUT, POST, HEAD, or DELETE. The GET operation is the default.

The request and response properties can contain `File` objects or strings.

- Request and response files

  The default encoding for both request and response files is BINARY; you can specify another encoding in the `File` object; see the *JavaScript Tools Guide* for information on `File`- and `Folder`-supported encoding names. (The `HttpConnection` properties requestencoding and responseencoding affect only string values, not files.)

  If the file is not open, it will be opened for reading (for a request) or for writing (for a response). Request and response files are not closed automatically; when there are no remaining JavaScript references to a file, it is eventually closed by the garbage collector.

- Request and response strings

  When the request is a string, it is converted to binary as specified by the requestencoding value. The default encoding is UTF-8.

  When the server response is anything other than a file, it is converted to a string using the responseencoding value; the default is ASCII.

Getting a file

```
var http = new HttpConnection("http://www.clanmills.com/robin.shtml") ;

http.response = new File("/c/temp/robin.shtml") ;

// Get is the default method

http.execute() ;

http.response.close() ;
```

Posting a string

```
var http = new HttpConnection("http://localhost/perlasp/wform.asp" ) ;

http.request = "Yourname=Fred Smith" ;

http.method = "POST"
```

```
http.execute() ;
```

Adding request headers and printing response headers

```
var http = new HttpConnection("http://localhost/perlasp/httpvar.asp") ;
http.requestheaders = ["MyHeader" , "MyValue"] ;
http.execute() ;
http.response = new File("/c/temp/dumpvars.txt") ;
var a = http.responseheaders ;
for ( i = 0 ; i < a.length/2 ; i++ ) {
    alert(a[i*2] + " => " + a[i*2+1]) ;
}
```

Asynchronous operations

The `HttpConnection` object can operate asynchronously; when you set async to true (or sync to false) the operation is performed in the background, while your script continues to do other work. However, the asynchronous behavior is not automatic. You must execute the pump() method periodically to increment the progress of the operation, and periodically test the status and lastread properties. After the status is `HttpConnection.statusCompleted`, you must continue to call pump() to transfer all bytes from the server to your object, until lastread is negative.

Blocking (synchronous use)

```
var http = new HttpConnection("http://someserver/index.html") ;

http.response = new File("/c/index.html") ;

http.execute() ;
```

Not blocking (asynchronous call)

```
var http = new HttpConnection("http://some.website/file.html") ;
http.async = true ; // or http.sync = false ;
http.onCallback = function() {
    with ( this ) {
        if ( status == HttpConnection.statusComplete && http.lastread < 0 ) {
            alert("done") ;
            this.close() ;
            }
        }
    return HttpConnection.actionContinue ;
}
http.execute() ; // returns immediately
//
// . . . Somewhere and occasionally
if ( http.status <= HttpConnection.statusComplete && http.lastread >= 0 )
    http.pump() ;
```

Authentication

You can specify a user login name and password in the URL using the standard syntax:

`http://[`*username*`:][`*password*`@]`*server*`[:`*port*`]/`*path*`?`*querystring*

Use an escape sequence for special characters, such as `%20` for space and `%40` for @.

You can override the user name and password specified in the URL by setting the username and password on the `HttpConnection` object.

If the connection is challenged by the server and authentication is required, the operation invokes your onAuthentication callback function. You can use this set the username and password properties; you cannot use it to change the URL.

Authentication callback

```
var http = new HttpConnection("http://www.website.com") ;
http.onAuthentication= function (host,realm,isProxy,retries,
    currentUser,currentPassword) {
        alert ("onHttpAuthentication CALLED" + \n +
            "host = " + host + \n +
            "realm = " + realm + \n +
            "isProxy = " + isProxy + \n +
            "retries = " + retrie + \n +
            "currentUser = " + currentUser + \n +
            "currentPassword = " + currentPassword ) ;
        this.username = "therealusername" ;
        this.password = "thepassword" ;
        return HttpConnection.actionContinue ;
        }
http.execute() ;
```

HttpConnection object reference

This section provides details of the `HttpConnection` object's properties and functions.

HttpConnection object constructor

```
[new] HttpConnection ( [url] );
```

url	Optional. The URL to which to connect. The URL specifies the protocol; for example:
	`http://localhost`
	`https://localhost`
	If not provided, you must set the object's `url` property.

HttpConnection object properties

async	Boolean	When true, the connection is asynchronous. Operations spawn a thread and return immediately to the main thread. The background thread sets `isComplete` to true when the current operation has finished. If the operation times out, `isComplete` is set to true and `error` is set to `errorTimeout`. Default is false. When set to true, sets `sync` to false. Read-write.
chunked	Boolean	When true, send the response using chunked encoding. Default is true. Read-write.

bytesReceived	Number	The number of bytes received from the HTTP server. -1 when there is no connection.
bytesSent	Number	The number of bytes transmitted to the HTTP server. -1 when there is no connection.

fault	Number	The error status of the connection. Read only. A constant value, one of:
		
		`HttpConnection.faultNone`
		`HttpConnection.faultUserCancelled`
		`HttpConnection.faultNoConnection`
		`HttpConnection.faultHostNotFound`
		`HttpConnection.faultNetTimeout`
		`HttpConnection.faultClientTimeout`
		`HttpConnection.faultMalformedUrl`
		`HttpConnection.faultInvalidResponse`
		`HttpConnection.faultUnauthorized`
		`HttpConnection.faultRelocated`

isOpen	Boolean	When true, the connection to the FTP server is open. Default is false. Read only.
lastread	Number	The number of bytes read from the HTTP server during the last call to pump(). Negative when execution is completely finished. Default is 0. Read only.

method	String	The HTTP method. Read-write. One of:
		
		`GET` (default)
		`PUT`
		`HEAD`
		`POST`
		`DELETE`

mime	String	The MIME type of the request. Default is `text/html`. Read-write.

network	Number	The network status of the connection. Read only. A constant value, one of:
		
		`HttpConnection.networkIdle`
		`HttpConnection.networkConnecting`
		`HttpConnection.networkSendingRequestHeaders`
		`HttpConnection.networkSendingRequestBody`
		`HttpConnection.networkAwaitingResponse`
		`HttpConnection.networkReceiveingResponseHeaders`
		`HttpConnection.networkReceiveingResponseBody`
		`HttpConnection.networkResponseComplete`
		`HttpConnection.networkProxyIdle`
		`HttpConnection.networkProxyConnecting`
		`HttpConnection.networkProxyConnected`

onAuthentication	Function	Optional. A callback function invoked by the server if authentication fails using the username and password passed with the original URL. Use this method to override the username and password by setting `this.username` and `this.password`. The callback function takes these arguments: *host*: the server name string. *realm*: a string provided by the server. *isProxy*: true if the server is a proxy. *retries*: always 1 *currentUser*: the user name string already presented to the server. *currentPassword*: the password string already presented to the server. The function should return `HttpConnection.actionContinue`.
onCallback	Function	Optional. A callback function for the operation being executed. It is automatically invoked periodically during synchronous operations. For an asynchronous operation, each call to pump() invokes this function. Read-write. You can use this function to monitor the progress and check the completion status in this object (the value of `this` in the function), in order to provide progress feedback in the user interface and allow cancellation of long operations. Use `this.close()` in this function to halt the operation. The function takes no arguments. It should return `HttpConnection.actionContinue` or `HttpConnection.actionCancel`.
password	String	The connection password for the HTTP server. Set this to override the password given in the URL. Default is `undefined`, for an unsecured or anonymous connection. Read-write.
proxy	String	The HTTP proxy server. A string containing an IP address and port, or the empty string to use the operating-system default, or `undefined` (the default) for no proxy server. Read-write.
redirect	Number	The maximum number of redirection tries for the request. If the server redirects the request to another server (returning a response status of 301 or 302), this connection resends the request to that server. If it redirects this number of times without success, it returns an error. Default is 5. Read-write.
response	String or File	The response to the request, received from the HTTP server. Read only.
responseencoding	String	The encoding to use in converting the request to a string. Default is `ascii`. Read-write.

responseheaders	Array of String	The response headers, an array of key-value pairs. Read only.
responseStatus	Number	The response status, an HTTP Response code (such as 200 for OK, or 404 for "file not found") or -1 if no status has been received. Read only.
request	String or File	The request to execute on the HTTP server. Read-write.
requestencoding	String	The encoding to use in converting the request string to binary. Default is `utf8`. Read-write.
requestheaders	Array of String	The request headers, an array of key-value pairs. Read-write.
snooze	Number	A number of milliseconds to wait before checking the completion status of synchronous operations. Default is 10. Read-write.

status	Number	The execution status of the request. Read only. A constant value, one of:
		
		`HttpConnection.statusIdle`
		`HttpConnection.statusRunning`
		`HttpConnection.statusCompleted`
		`HttpConnection.statusSuspended`
		`HttpConnection.statusFailed`

sync	Boolean	When true, the connection is synchronous. Operations block the main thread and return when complete. Default is true. When set to true, sets `async` to false. Read-write.
timeout	Number	An integer, the number of seconds to continue attempting to make the connection before completing with the message `faultNetTimeout`. Default is 5. Read-write.
url	String	The URL of the HTTP server, and optionally the port, to which to connect. This includes the protocol (HTTP or HTTPS), and can include a login user name and password in this format: `http[s]://[[`*`username`*`:]`*`password`*`@]`*`server`*`[:`*`port`*`]` Default is `undefined`. Read-write.
username	String	The connection user name for the HTTP server. Set this to override the user name given in the URL. Default is `undefined`, for an anonymous connection. Read-write.

HttpConnection object functions

close()
httpObj.close ();

Terminates the open connection. Deleting the object also closes the connection, but not until JavaScript garbage-collects the object. The connection might stay open longer than you wish if you do not close it explicitly. There are a limited number of open connections available; failing to close connections can make you unable to open a new one.

Returns true if the connection was closed, false on I/O errors.

execute()
httpObj.execute ();

Opens a connection if necessary, executes the request on the HTTP server, and receives the response.

Returns true on success, false on errors. Check fault for the error c ode.

pump()
httpObj.pump ();

Increments the progress of an asynchronous connection. You must call this function periodically to advance the progress of an asynchronous operation.

Executes the callback procedure defined in onCallback, passing no arguments.

Returns true on success, false on I/O errors.

Index

I

J

K

L

M

N

O

T

U

V

W

X