enfocus SWITCH 2018

Scripting

enfocus

Contents

1. Copyrights **5**

2. About scripting in Switch **6**

3. Scripting concepts **7**
- 3.1. Scripting overview 7
- 3.2. Putting a script in a flow 9
  - 3.2.1. Script expressions 9
  - 3.2.2. Script package 14
  - 3.2.3. Script declaration 15
  - 3.2.4. Scripted plug-in 16
- 3.3. Scripting languages 16
  - 3.3.1. JavaScript 16
  - 3.3.2. VBScript 17

4. Developing scripts **19**
- 4.1. Using SwitchScripter 19
  - 4.1.1. Launching SwitchScripter 19
  - 4.1.2. SwitchScripter Workspace window 19
  - 4.1.3. SwitchScripter Toolbar 21
  - 4.1.4. SwitchScripter Declaration pane 22
  - 4.1.5. SwitchScripter Fixture pane 23
  - 4.1.6. SwitchScripter Program pane 27
  - 4.1.7. SwitchScripter Properties pane 28
  - 4.1.8. SwitchScripter Message pane 28
- 4.2. Writing and testing a script 29
  - 4.2.1. Writing a script 29
  - 4.2.2. Testing a script 31
- 4.3. Developing a scripted plug-in 33
  - 4.3.1. Obtaining permission 33
  - 4.3.2. Creating a scripted plug-in 34
  - 4.3.3. Configurator guidelines 36
- 4.4. Script declaration 42
  - 4.4.1. Main script properties 43
  - 4.4.2. Execution mode 46
  - 4.4.3. Property definition properties 48
  - 4.4.4. Property editors 49
  - 4.4.5. Validating property values 53
  - 4.4.6. External property editor 56

5. Scripting reference **60**

5.1. Scripting API - Introduction 60
5.2. JavaScript Reference 61
5.2.1. Language concepts 61
5.2.2. Built-in types and objects 64
5.2.3. Built-in functions 83
5.2.4. Built-in operators 85
5.2.5. Declarations 93
5.2.6. Control statements 95
5.3. Utility module 104
5.3.1. Text encoding 104
5.3.2. ByteArray class 107
5.3.3. File class 110
5.3.4. Dir class 117
5.3.5. Process class 122
5.4. XML module 127
5.4.1. Node class 127
5.4.2. Document class 130
5.4.3. Element class 133
5.4.4. Text class 135
5.4.5. Comment class 136
5.4.6. Attr class 136
5.4.7. List classes: NodeList, AttrList 137
5.5. Network module 138
5.5.1. HTTP class 138
5.5.2. SOAP class 151
5.6. Database module 159
5.6.1. Text encoding 159
5.6.2. DataSource class 160
5.6.3. Statement class 161
5.7. Flow element module 165
5.7.1. Entry points 165
5.7.2. Flow element update 172
5.7.3. Environment class 175
5.7.4. Switch class 187
5.7.5. Connection class 195
5.7.6. Job class 197
5.7.7. Occurrence class 210
5.7.8. WebhookRequest class 212
5.7.9. WebhookResponse class 213
5.7.10. List classes: ConnectionList, JobList 214
5.8. Metadata module 214
5.8.1. Map class 214
5.8.2. List classes 215

5.8.3. FileStatistics class 216
5.8.4. ClientConnection class 225
5.8.5. Job class 226
5.8.6. Dataset class 226
5.8.7. XML data model 228
5.8.8. JDF data model 229
5.8.9. XMP data model 231
5.8.10. Opaque data model 237
5.8.11. CP2 data model 237

1. Copyrights

© 2019 Enfocus BVBA all rights reserved. Enfocus is an Esko company.

Certified PDF is a registered trademark of Enfocus BVBA.

Enfocus PitStop Pro, Enfocus PitStop Workgroup Manager, Enfocus PitStop Server, Enfocus Connect YOU, Enfocus Connect ALL, Enfocus Connect SEND, Enfocus StatusCheck, Enfocus CertifiedPDF.net, Enfocus PDF Workflow Suite, Enfocus Switch, Enfocus SwitchClient, Enfocus SwitchScripter and Enfocus Browser are product names of Enfocus BVBA.

Adobe, Acrobat, Distiller, InDesign, Illustrator, Photoshop, FrameMaker, PDFWriter, PageMaker, Adobe PDF Library™, the Adobe logo, the Acrobat logo and PostScript are trademarks of Adobe Systems Incorporated.

Datalogics, the Datalogics logo, PDF2IMG™ and DLE™ are trademarks of Datalogics, Inc.

Apple, Mac, Mac OS, Macintosh, iPad and ColorSync are trademarks of Apple Computer, Inc. registered in the U.S. and other countries.

Windows, Windows 2000, Windows 7, Windows 8, Windows 8.1, Windows 10, Windows 2008 Server, Windows 2008 Server R2, Windows Server 2012, Windows Server 2012 R2 and Windows Server 2016 are registered trademarks of Microsoft Corporation.

PANTONE® Colors displayed here may not match PANTONE-identified standards. Consult current PANTONE Color Publications for accurate color. PANTONE® and other Pantone, Inc. trademarks are the property of Pantone, Inc. ©Pantone, Inc., 2006.

OPI is a trademark of Aldus Corporation.

Monotype is a trademark of Monotype Imaging Inc. registered in the U.S. Patent and Trademark Office and may be registered in certain jurisdictions. Monotype Baseline is a trademark of Monotype Imaging Inc.

Quark, QuarkXPress, QuarkXTensions, XTensions and the XTensions logo among others, are trademarks of Quark, Inc. and all applicable affiliated companies, Reg. U.S. Pat. & Tm. Off. and in many other countries.

This product and use of this product is under license from Markzware under U.S.Patent No. 5,963,641.

Other brand and product names may be trademarks or registered trademarks of their respective holders. All specifications, terms and descriptions of products and services are subject to change without notice or recourse.

2. About scripting in Switch

SwitchScripter is a Switch application component that offers a scripting development environment. This allows you develop your own scripts, for example to extend the capabilities of Switch, or to create your own configurator(s), for interaction with third party applications that are not yet supported by Switch. As of Switch 13 update 1, you can also develop Switch apps with SwitchScripter.

To able to use the SwitchScripter, you need an active license for the Scripting Module.

Note: An active license will give you complete access to the scripting API. Therefore, information in the internal datasets (XMP), external datasets or databases is accessible through the scripts, even though the Switch Metadata module is not licensed. You can also define script expressions for certain properties of some of the flow elements (for example: the Job priority property for the Hold job flow element).

3. Scripting concepts

3.1. Scripting overview

Switch offers a comprehensive scripting environment. With limited scripting experience, a user can substantially extend Switch's capabilities and provide for interaction with third-party systems in new and powerful ways.

The various topics in this chapter describe the Switch scripting environment, including its capabilities for working with metadata.

Note: In addition to Switch's own scripting environment, certain Switch configurators support executing JavaScript scripts inside the Adobe Creative Suite applications. These types of scripts are not discussed in this chapter; instead refer to Javascript for applications in the Switch Reference Guide available on the *Enfocus website*.

Switch supports two types of scripting languages:

- Operating system scripting languages: use the script execution mechanism provided by the operating system to support a platform-specific scripting language.
- Internal scripting language: a cross-platform JavaScript subset implemented and executed inside Switch.

Operating system scripting languages

Operating system scripting is intended primarily for communicating with other applications or with operating systems services from within Switch. Currently, the only supported scripting language is VBScript on Microsoft Windows.

Note: Starting from Switch 2017, it is no longer possible to create new scripts using AppleScript. However, Switch 2017 and Switch 2018 will still be able to execute the scripts created in older Switch versions.

In VBScript, a script object representing the Switch application provides access to key variables and functions, including the metadata context of the job being processed.

By definition, operating system scripts are platform-specific (i.e. not portable).

Internal scripting language

The Switch-internal scripting language JavaScript is intended primarily for working with variables and functions provided by Switch (rather than the operating system). A primary goal of the internal scripting language is to work with the contents of metadata fields associated with the jobs being processed.

Switch supports a substantial JavaScript subset (more precisely, it supports a subset of the ECMAScript 4.0 language), with extensions for:

- Accessing key Switch variables and functions, including the metadata context of the job being processed.
- Reading and writing plain text files.

- Reading and writing XML files and performing XSLT transforms.
- Executing a command line application.
- Database access.
- Webservices interaction.

JavaScript scripts have limited "external access". For example, there is no way to communicate with interactive applications, and there is no access to operating system services.

JavaScript scripts are cross-platform, i.e. they work without change on both Mac OS X and Microsoft Windows.

Script element and script expression

A script element is a flow element that encapsulates a script written in any of the supported scripting languages. The script implements an "entry point" function called by Switch for each job being processed through the flow element.

A script expression is a text string associated with a flow element property. To determine the value of the property, Switch evaluates the contents of the string as a JavaScript expression in the context of the job being processed. Only JavaScript is supported in script expressions.

The Switch scripting API (application programming interface) provides script elements and script expressions with access to information about the job being processed, including job ticket and metadata contents. Script expressions are limited to read-only access, while script elements can update the information.

Script package

A script that is to be associated with a script element consists of the following components:

- Script declaration: an XML file that specifies information about the script, such as the name and data type of any properties (arguments) used by the script.
- Script program: a plain text file that contains the script program itself, written in one of the supported scripting languages.
- Script icon: optionally, an icon that is used to display the script element in the Switch flow designer.

These components are collected in a single file called a "script package" which is presented to the script element as a single entity. This also allows creating "fat" packages containing an implementation of the same functionality on multiple platforms.

Scripted plug-in

A scripted plug-in is a script package that, when saved with the appropriate password and placed in the appropriate folder, behaves just like a tool or configurator built into Switch.

Scripted plug-ins are of no concern to regular Switch users. Enfocus may license selected third-party vendors to develop scripted plug-ins.

Script development

Switch includes a script development environment called SwitchScripter, installed as a separate application with the following major functions:

- Create and edit Switch script packages and its components.
- Emulate the Switch scripting API for testing purposes.
- Set up specific input/output conditions for testing purposes.

3.2. Putting a script in a flow

3.2.1. Script expressions

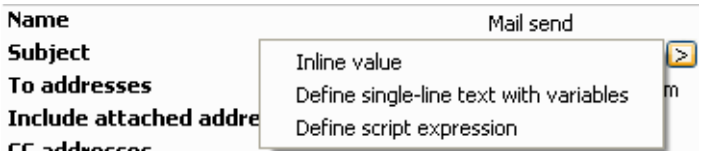
A script expression is a JavaScript program that gets evaluated in the context of a job (file or job folder) to determine the value of a particular property of a flow element or connection. This allows the property value to depend on the job or its associated metadata.

3.2.1.1. Associating a script expression with a property

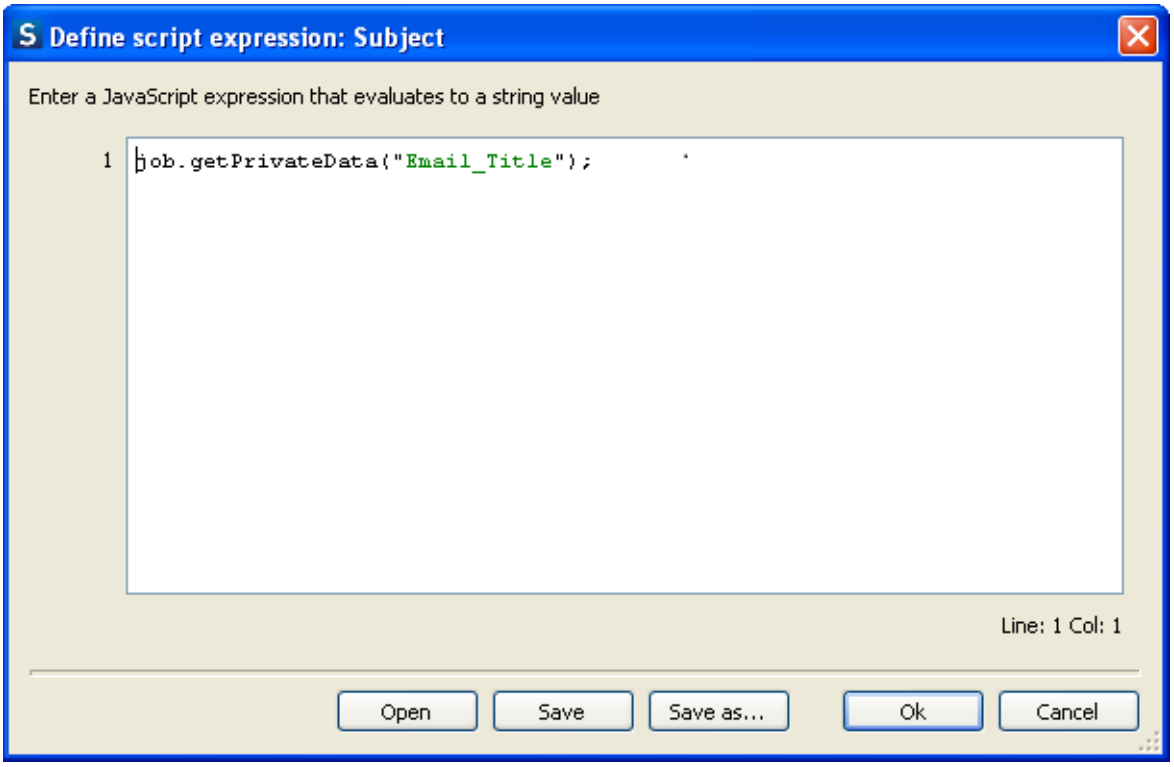
Many flow element and connection properties in Switch offer an extra property editor that allows entering a script expression.

String value example

For example, the "Subject" property of the Mail send flow element allows a script expression in addition to a regular explicit value:



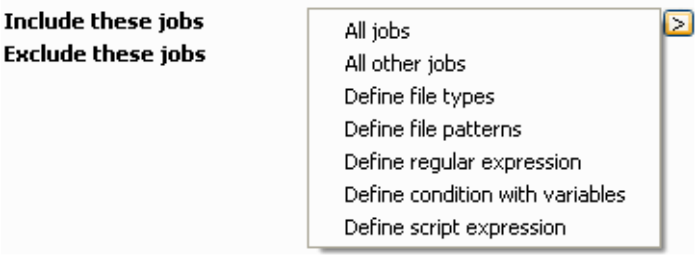
Select "Inline value" (the default) to enter an explicit string value in the text field next to the "Subject" label. Select **Define Script Expression** to enter a script expression in the property editor, which looks as follows:



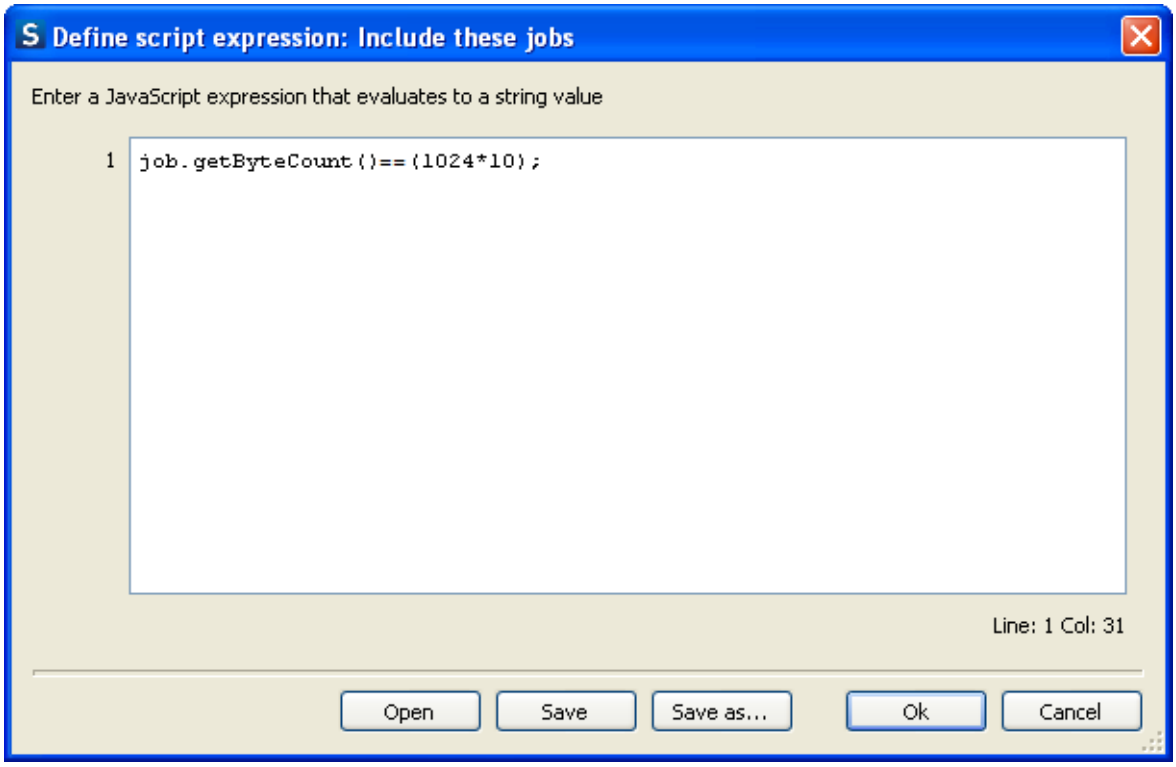
In this case, when it is evaluated in the context of the job being processed, the script expression should return a string value (or a value that can be meaningfully converted to a string under the JavaScript rules). The resulting value is used as the subject line of the email sent out for the job.

Boolean value example

As a second example, the "Include these jobs" property of a connection leaving a folder allows a script expression in addition to other filter types:



Select **Define Script Expression** to enter a filter script expression in the property editor, which looks as follows:



In this case, when it is evaluated in the context of the job being considered as a possible match for the filter, the script expression should return a Boolean value, i.e. true or false (or a value that can be meaningfully converted to a Boolean under the JavaScript rules). If the resulting value is true, the job will be considered to match the filter, otherwise it won't.

Array value example

When the main editor is "Multi-line text with variables" the script element or configurator expects an array in return, but in that case it works a bit differently. Instead of returning the array as is, a joined version of it using \n as a separator should be returned:

```
var mailAddresses = ["123@abc.com","456@def.com"];
//mailAddress;//do not return just this
mailAddresses.join("\n");//but do it like this
```

In the script element or configurator, the return value can then be easily turned into an array again by splitting the string based on \n. Note that not all existing scripts and configurators implement this behavior, so although we recommend this approach it may not always work as described here.

3.2.1.2. Writing a script expression

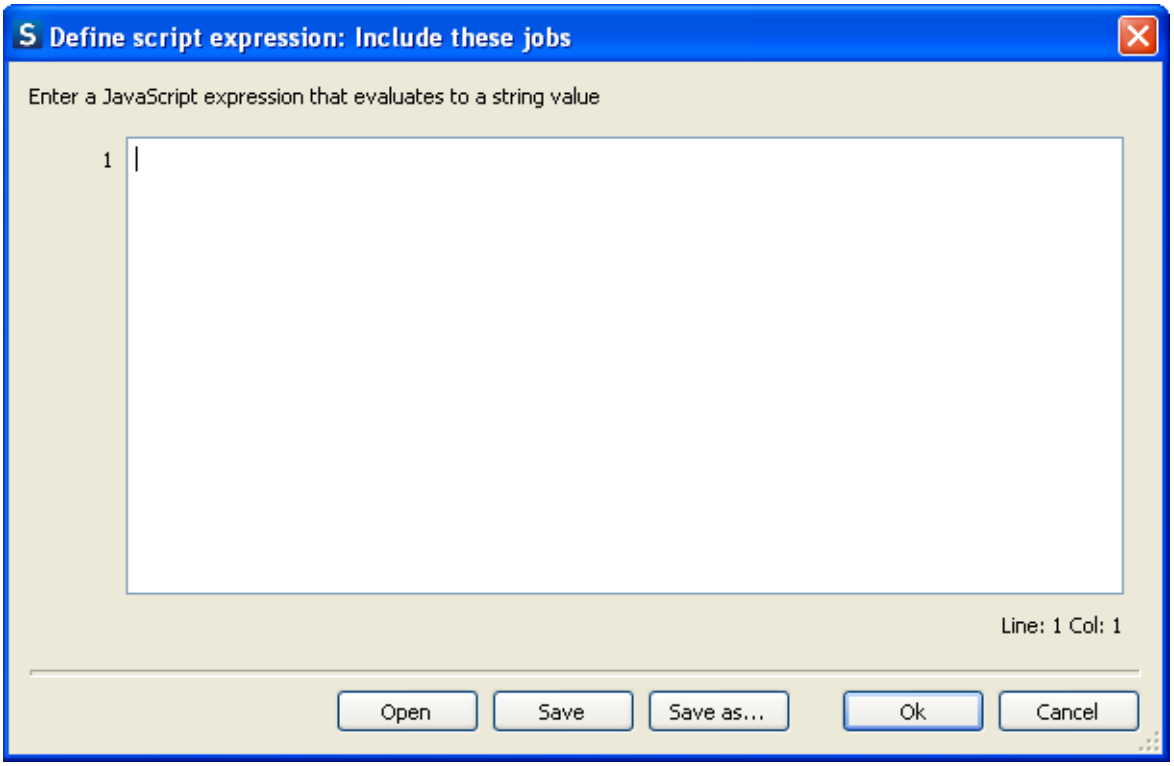
A script expression is written in JavaScript; other scripting languages are not supported.

The value of a script expression's property is set to the result of evaluating the main body of the script as an expression. The main body may invoke functions defined in the script expression itself or provided as part of the Switch scripting API. Also, the script expression can use the

predefined variables "job" and "s" which represent the job and the flow element context at the moment of script evaluation (see predefined variables below).

You have two options for developing and testing a script expression:

- Enter the script expression directly in the appropriate property editor dialog, as demonstrated in the previous sections, and test it by activating the flow and processing some sample jobs.
- Use SwitchScripter to develop and test the script expression, and then load it into the appropriate property editor dialog box.



SwitchScripter offers several advantages for working with complex script expressions. Most importantly, you can setup test fixtures to verify the script expression's results without having to activate a flow and process jobs, and you can save the script expression in a script package for future use (like when using a script expression) or for sharing it with other users.

3.2.1.3. Access to scripting API

A script expression has access to a subset of the Switch scripting API (application programming interface). In general, a script expression should not have any persistent side-effects; for example it cannot modify the current job or write data to disk. There are two notable exceptions to this rule however: a script expression can log messages and it can update global user data. The following subsections describe in more detail which portions of the scripting API can be accessed from a script expression.

Note: Do not attempt to violate these limitations (even if Switch inadvertently allows you to do it); the result is undefined and most likely disastrous.

Predefined variables

A script expression can use the following predefined global variables:

- job: an instance of the Job class which represents the job being processed or considered at the moment of script evaluation.
- s: an instance of the Switch class which represents the flow element in which the script expression is defined (and its environment).

Flow element module

Entry points are not supported (instead the main body of a script expression is evaluated to obtain a value).

The Environment class is included except for the following functions: copy, getApplicationPath, getApplicationLicense, findRegisteredApplication, findApplicationOnDisk, find64BitApplicationOnDisk, getSecondsLeft, sleep.

The Switch class is included except for the following functions: getScriptName, getIn/OutConnections, getJobsForConnection, createNewJob, failProcess, set/getTimerInterval.

The Connection class is excluded.

The Job class is included, but only its functions named "log", "get..." or "is..." are included. This includes functions for logging, getting job file/folder information, getting job ticket info, getting read-only metadata datasets, and evaluating variables. All other functions are excluded.

Metadata module

This module is fully included except for the functions for updating properties.

Specifically, the Map class, the Dataset class and all its subclasses for the supported data models are included, except for the functions for updating properties in the XMP data model.

The CP2 (Certified PDF2) data model is now a part of the Metadata module. It includes the Session class, Certificate class, User class, DataMap class and AltText class.

Utility module

The File class is included but its access mode is limited to read-only. All functions named "remove..." and "write..." are excluded.

The Dir class is included but it is limited to read-only functions. Specifically, all functions named "mkdir...", "rmdir..." and "setCurrent" are excluded.

The Process class is excluded.

XML module

The Document class is included but it is limited to read-only functions. Specifically, the functions named "save" and "transform" are excluded.

All other classes in this module (such as the Element class) are fully included.

Network module

This module allows communication with external applications. It is exposed only to the JavaScript language (not to VBScript).

The classes and functions in the Network module support HTTP/HTTPS and SOAP protocols (a subset of the SOAP 1.1 communication protocol over HTTP).

Switch SOAP class supports two types of attachments:

- DIME
- MIME

Database module

This module allows access to external databases via ODBC and SQL. This module is exposed only to the JavaScript language (not to VBScript).

The classes and functions in the Database module offer access to a small but functional subset of the ODBC API. The DataSource class and Statement class are included.

3.2.2. Script package

A script package contains all of the information needed to execute a particular script program in the context of a script element. See scripting overview for more background information. Use the SwitchScripter development environment to create and edit script packages.

Format

A script package includes the following components:

- Script declaration: an XML file that specifies information about the script, such as the name and data type of any properties (arguments) used by the script.
- Script program: a plain text file that contains the script program itself, written in one of the supported scripting languages.
- Script icon: optionally, an icon that is used to display the script element in the Switch flow designer.

A script package can contain multiple script programs, each program providing an implementation of the same functionality in a different scripting language. Since there is only one script declaration, from a semantic viewpoint a script package contains a single script even if it contains multiple implementations of the same functionality in different scripting languages.

All components of a script package are collected in a single file (stored as a ZIP archive) with the ".sscript" filename extension. Consequently a script package can be exchanged between users as a single entity.

Script declaration

The script declaration is an XML file that specifies information about the script, such as the name and data type of any properties (arguments) used by the script.

Script program

A script program is a plain text file that contains the script program itself, written in one of the supported scripting languages. The syntax and semantics must conform to those of the chosen scripting language. The script program also relies on the Switch scripting API.

A script package may contain multiple script programs, each written in a different scripting language . In that case the Switch execution engine selects a language version in the following order:

- The language version that is native to the operating system on which it is being executed (Visual Basic Scripting for Microsoft Windows).
- The JavaScript version.

If neither is present the script package cannot be executed on that operating system.

Note: Starting from Switch 2017, it is no longer possible to create new scripts using AppleScript. However, Switch 2017 and Switch 2018 will still be able to execute the scripts created in older Switch versions.

Icon

The optional script icon is a PNG image file (not interlaced) with a size of exactly 32x32 pixels in the RGB color space and with support for transparency. When a script package is associated with a script element, and the package contains an icon, the flow designer uses the icon to display the script element in the canvas. Otherwise the flow designer uses the default script element icon.

Password protection

A script package can be saved with a password. This means the script package can no longer be opened in SwitchScripter (for viewing or editing) without providing the same password. However the script package can still be executed by Switch. There is no way to block a script package from being executed.

The password protection allows a script author to distribute script packages without providing other parties with access to the script's source code (with an important exception – see the note below).

Important:

Certain Enfocus employees have access to the mechanism that bypasses the password protection (because even password-protected scripts can be executed and thus opened by Switch). While Enfocus does not intend to open password-protected scripts for viewing, and does not intend to publish the bypass mechanism, the company does not legally guarantee that this will never happen. It is important for script authors to understand the limitations of the protection.

3.2.3. Script declaration

A script declaration is an XML file that specifies information about a script program, such as the name and data type of any properties (arguments) used by the script. The contents of a script declaration can be edited with SwitchScripter as part of a script package. See scripting overview for more background information.

Purpose

When a script package is associated with a script element, the script declaration in the script package is used to:

- Produce the appropriate behavior in the Switch flow designer for the associated script element with regards to supported connections and properties.
- Provide the appropriate information to the runtime environment and the scripting API, for example, to access flow element properties and to allow moving jobs in accordance with the semantics for various outgoing connection types.

Main script properties

A script declaration contains the following information:

- The name of the script.
- The number and type of supported incoming and outgoing connections.
- The script's execution mode (as in concurrent or serialized).
- A list of definitions for properties injected into the script element.
- A list of definitions for properties injected into the outgoing connections leaving the script element.

See main script properties for reference information.

Property definition properties

Each property definition in the script declaration includes:

- A tag used to uniquely identify the property to the script and its human-readable name.
- The type of property editors used to enter a value for the property.
- Specifications for validating the property value.

See property definition properties and property editors for reference information.

3.2.4. Scripted plug-in

A scripted plug-in is a script package which when saved with the appropriate password and placed in the appropriate folder, behaves just like a tool or configurator built into Switch. This allows Enfocus and its licensed partners to create built-in tools and configurators using scripting.

Scripted plug-ins are of no concern to regular Switch users. Enfocus may license selected third-party vendors to develop scripted plug-ins.

If you are interested in developing a scripted plug-in then refer to the following topics:

- Obtaining permission
- Creating a scripted plug-in
- Configurator guidelines

3.3. Scripting languages

3.3.1. JavaScript

The Switch-internal scripting language JavaScript is intended primarily for working with variables and functions provided by Switch (rather than the operating system). A primary goal of

the internal scripting language is to work with the contents of metadata fields associated with the jobs being processed.

Switch supports a substantial JavaScript subset (more precisely, it supports a subset of the ECMAScript 4.0 language), with extensions for:

- Accessing key Switch variables and functions, including the metadata context of the job being processed.
- Reading and writing plain text files.
- Reading and writing XML files and performing XSLT transforms.
- Executing a command line application.
- Database access
- Web services access

JavaScript scripts have limited "external access". For example, there is no way to communicate with interactive applications and there is no access to operating system services.

JavaScript scripts are cross-platform, that is they work without change on both Mac OS X and Microsoft Windows.

Language reference

Refer to the Switch JavaScript language reference included with Switch help.

Extensions reference

Refer to the *Scripting reference* on page 60.

3.3.2. VBScript

VBScript scripting in Switch is intended primarily for communicating with other applications or with operating systems services from within Switch. A script object representing the Switch application provides access to key variables and functions, including the metadata context of the job being processed. VBScript is available on Microsoft Windows only. Refer to the *Scripting overview* on page 7 for more background information.

Note: Switch does not support the JScript language (a scripting language also available on Microsoft Windows).

Language reference

Comprehensive reference and introductory information about VBScript can be found on the Microsoft web site and in widely available literature. Use the search term "VBScript" when looking for information on the web. Don't confuse VBScript with the Visual Basic programming language (a full-featured programming language and development environment offered by Microsoft) which is quite distinct from VBScript.

A good starting page regarding VBScript on the Microsoft web site is the page on Windows Script in the Microsoft MSDN section. This page links to a full VBScript language reference and to a script repository that contains numerous practical example scripts.

Extensions reference

Switch provides a number of Switch-specific classes and functions to support accessing key Switch variables and functions, including the metadata context of the job being processed. Refer to the Switch scripting API for reference documentation about these extensions.

The Switch scripting API reference documentation uses the JavaScript syntax and semantics to describe its various classes and functions. In almost all cases, there is a straightforward correspondence with VBScript syntax and semantics. The following sections highlight some issues to keep in mind while reading the scripting API documentation.

API restrictions

Not all of the classes and functions described in the scripting API are available when using VBScript. Specifically, the classes and functions in the "Utility", "XML", "Database" and "Network" modules are not available. If you need the functionality in these modules, you have to use the native VBScript capabilities.

For example, the "Utility" module provides "File" and "Dir" classes to work with files and folders. In VBScript this functionality is available through the "FileSystemObject".

Functions versus procedures

In JavaScript there is no difference between calling a function (something that returns a value) and calling a procedure (something that does not return a value). In VBScript there is a difference and this is one of the most common pitfalls when translating JavaScript code into VBScript.

For example, here's how to call two of the Job class functions (defined in the Switch scripting API) in JavaScript:

```
job.addEmailAddress("test@test.com");
theResult = job.getPrivateData("key");
```

In VBScript procedure calls do not use parenthesis to enclose the arguments, so the above translates to:

```
job.addEmailAddress "test@test.com"
theResult = job.getPrivateData("key")
```

4. Developing scripts

4.1. Using SwitchScripter

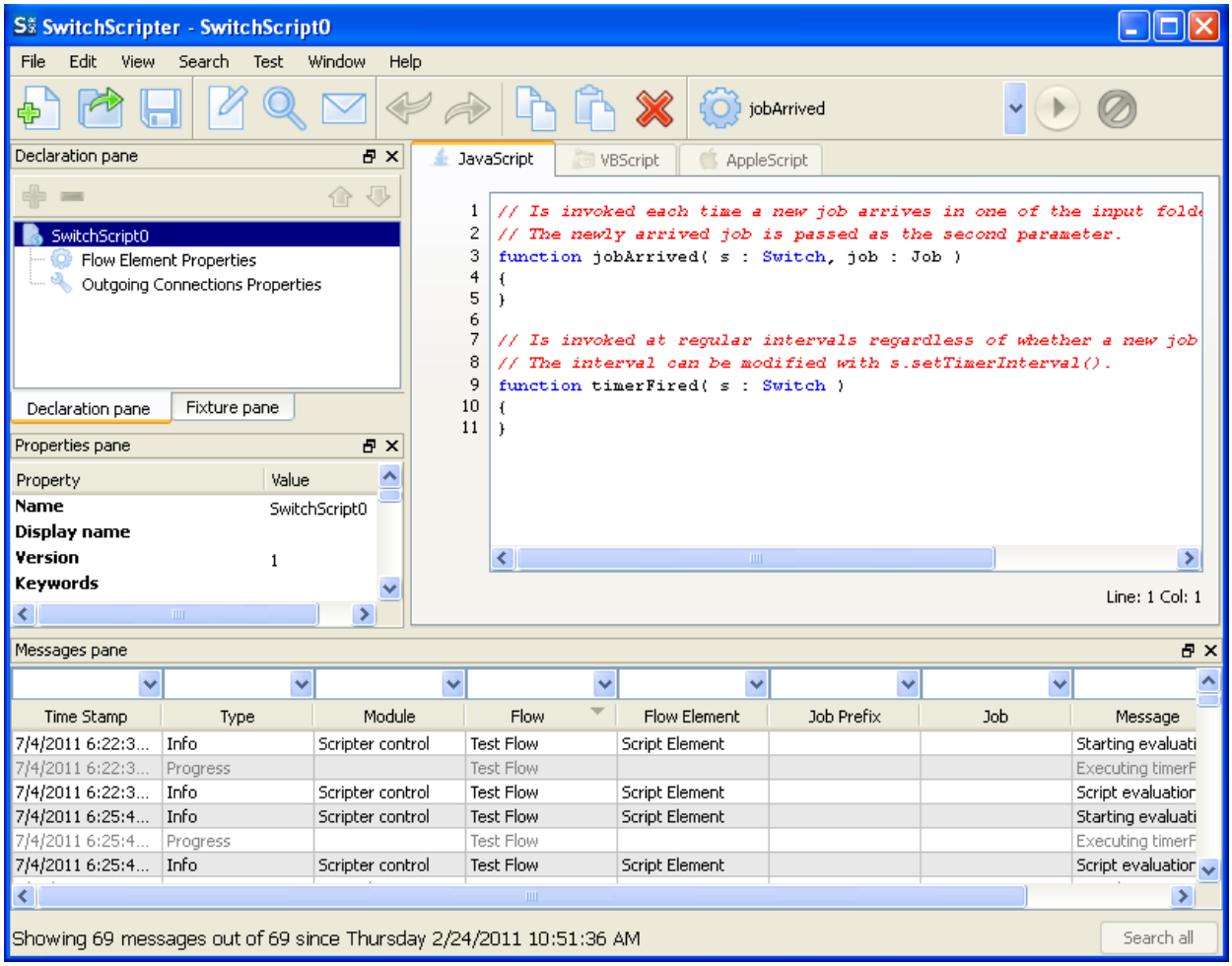
4.1.1. Launching SwitchScripter

To launch SwitchScripter:

- Double-click the SwitchScripter application icon, or
- Double-click a Switch script package (a file with the ".sscript" filename extension), or
- Select a script element in the canvas and choose the "Edit script package" menu item in its context menu.

SwitchScripter offers a workspace window to edit a single script package. SwitchScripter can display multiple instances of the workspace window at the same time.

4.1.2. SwitchScripter Workspace window



The SwitchScripter window offers a workspace with the following important areas (each of which is discussed in a separate topic):

Workspace area	Description
Toolbar	Contains tool buttons for the most frequently used functions
Declaration pane	Allows editing the script declaration, defining support for connections, injected properties and so on
Fixture pane	Allows setting up an emulated test environment so that you can test-run a script without attaching it to a flow
Properties pane	Serves to edit the properties of the item currently selected in the declaration or fixture pane
Program pane	Allows editing your script program in any of the supported scripting languages
Message pane	Displays log messages issued by your script or by the emulated run-time environment during testing

The various panes can be shown or hidden by choosing the corresponding item in the View menu. Panes can be resized by dragging the separators between them, they can rearranged next to one another or overlaid as tabs by dragging their title bar and they can be undocked as a floating pane. All configuration settings are persistent across sessions.

Also see *Writing a script* on page 29 and *Testing a script* on page 31.

4.1.3. SwitchScripter Toolbar

The toolbar offers a number of tool buttons for frequently used operations (all of which can be accessed via menu items as well).

File

Tool	Description
New	Create a new empty script package
Open	Open an existing script package
Save script	Save the script package being edited

View

Tool	Description
Declaration/Fixture pane	Show the Declaration pane or the Fixture pane; toggle between these panes
Properties pane	Show or hide the Properties pane
Messages pane	Show or hide the Messages pane

Edit

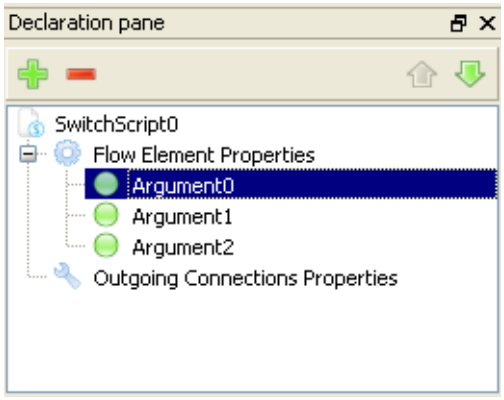
Tool	Description
Undo	Undo the most recent text editing operation in the Program pane (supports multipleundo)
Redo	Redo the most recently undone text editing operation in the Program pane
Copy	Copy the selected text in the Program pane to the clipboard

Tool	Description
Paste	Paste the clipboard's text contents in the Program pane
Delete	Delete the selected text from the Program pane

Test

Tool	Description
Select entry point	Select the entry point that will be invoked when the "Invoke entry point" button is pressed The drop-down menu offers a list of all entry points supported by the Switch scripting API; entry points that are not present in the currently visible script program (Program pane) are preceded by a red "unavailable" icon The menu item "script expression" is special in the sense that it indicates the main script body rather than a specific entry point; it is used for testing script expressions
Invoke entry point	Invoke the selected entry point in the currently visible script program (Program pane), in the context of the text fixture; progress is logged in the Messages pane This button is enabled only if the selected entry point is present in the currently visible script program, if the Fixture pane is visible, and (for evaluating script expressions and for entry points that take a job argument) if a job is selected in the Fixture pane
Abort	Abort script execution

4.1.4. SwitchScripter Declaration pane



The Declaration pane allows viewing and editing the information in the script declaration for the script package. When you select an item in the Declaration pane, the Properties pane displays the properties for that item.

See *Writing a script* on page 29 for background information.

Terminology

In this topic the term "property" is used to mean two different things:

- A property injected into the flow element to which this script package will be attached, or into its outgoing connections.
- A property of the item selected in the SwitchScripter declaration pane (which may be the definition of an injected property!).

This is confusing but hopefully the context of the term's use will resolve the ambiguities.

Main script properties

The properties displayed in the properties pane when you select the main script item in the Declaration pane (that is, "SwitchScript0" in the example above) are described in main script properties.

Property definitions

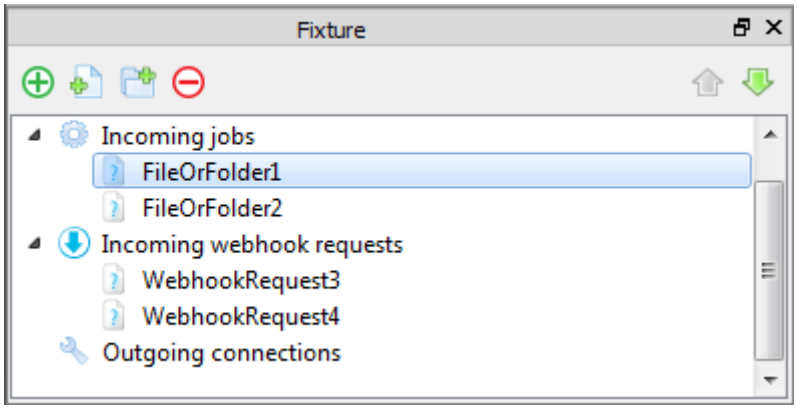
The "Flow element properties" and "Outgoing connections properties" sections of the script declaration define properties to be added to the property of flow element to which this script package will be attached or to the outgoing connections of that flow element. Each section can have any number of property definitions (including zero).

Use the small tool buttons at the top of the Declaration pane to add, remove and reorder property definitions. In the above example, there are two flow element property definitions ("Argument 1" and "Argument 2") and a single outgoing connection property definition ("Injected").

The values of these injected properties are entered in the designer's Properties pane while designing a flow, and can be retrieved by the script at run time.

The properties displayed in the Properties pane when you select one of the property definitions in the Declaration pane (that is, "Argument 1", "Argument 2" and "Injected" in the above example) are described in property definition properties.

4.1.5. SwitchScripter Fixture pane



The information setup through the Fixture pane defines a simulated run-time environment for script testing purposes, including:

- The values of the properties for the flow element to which the script is considered to be attached.
- The number of incoming and outgoing connections and the relevant property values for each.
- A list of input jobs (files or job folders), and the corresponding job ticket and metadata information for each.
- The target location for any output files of the test.

The fixture information is stored in a fixture package alongside the script package (with the same filename but a different extension). The fixture is not part of the definition of a script and may be removed after the script has been tested.

Basic fixture settings

The following table describes the properties displayed in the Properties pane when you select the main item in the Fixture pane (that is, "SwitchScript0" in the example above).

The values of these settings (except for the test folder path) can be accessed in the script at run-time through functions of the Switch class in the Switch scripting API.

Property	Description
Test folder	The absolute path of the folder in which to place files or folders provided to or generated by the script during testing
Global data	The set of global data presented to the script for all scopes
Flow name	The name of the flow to which the script is considered to be attached
Element name	The name of the flow element to which the script is considered to be attached
Injected flow element properties	Additional flow element properties as specified in the Declaration pane

Incoming jobs

The "incoming jobs" section of the fixture defines the jobs that are presented to the script as inputs during testing, including job ticket and metadata info, and the connection on which each job is to be presented.

Use the small tool buttons at the top of the Fixture pane to add, remove and reorder job definitions. In the example above, there are two job definitions ("creaX-cmyk....pdf" and "creaX-spot....pdf").

The following table describes the properties displayed in the Properties pane when you select one of the job definitions in the Fixture pane.

The values of these settings (except for the test file or folder path) can be accessed in the script at run-time through functions of the Job class in the Switch scripting API.

Property	Description
File or folder	The absolute path to the file or folder for this input job
Enabled	Defines whether this input job is indeed included in the simulated test environment (can be used to temporarily exclude a job)
Connection name	The name of the incoming connection on which this job is to be presented
Hierarchy info	The hierarchy path to be stored in the internal job ticket (each segment as a separate string, ordered from top to bottom)
Email addresses	A list of email addresses to be stored in the internal job ticket
Email body text	Email body text to be stored in the internal job ticket
Job state	The job state to be stored in the internal job ticket
Private data	A list of tag/value pairs of private script data to be stored in the internal job ticket
Metadata datasets	A list of metadata dataset references to be stored in the internal job ticket

Incoming webhook requests

The "incoming webhook requests" section of the fixture defines the requests that will be passed to the webhookTriggered entry point. Only the selected request will be passed to the entry point.

Use the small tool buttons at the top of the Fixture pane to add, remove and reorder requests definitions. In the screenshot above, there are two webhook requests definitions ("WebHookRequest3" and " WebHookRequest4").

The following table describes the properties displayed in the Properties pane when you select one of the webhook request definitions in the Fixture pane.

Property	Description
Webhook request file	The absolute path to the json file with the request definition (see example .json file below).
Enabled	Defines whether this webhook request is indeed included in the simulated test environment (can be used to temporarily exclude a webhook request).

Example of a webhook request file:

```
{
    "Body": "{\"key\":\"value\",\"bool\":false}",
    "Headers": {
        "Connection": "close",
        "accept": "application/json",
        "content-length": "28",
        "content-type": "application/json",
        "host": "127.0.0.1:51080"
    },
    "Method": "PUT",
    "Path": "/api/v1/ping",
    "Protocol": "http",
    "Query": "param1=test",
    "RemoteAddress": "127.0.0.1"
}
```

Outgoing connections

The "outgoing connections" section of the fixture defines the outgoing connections, if any, that are presented to the script, including the values of relevant connection properties.

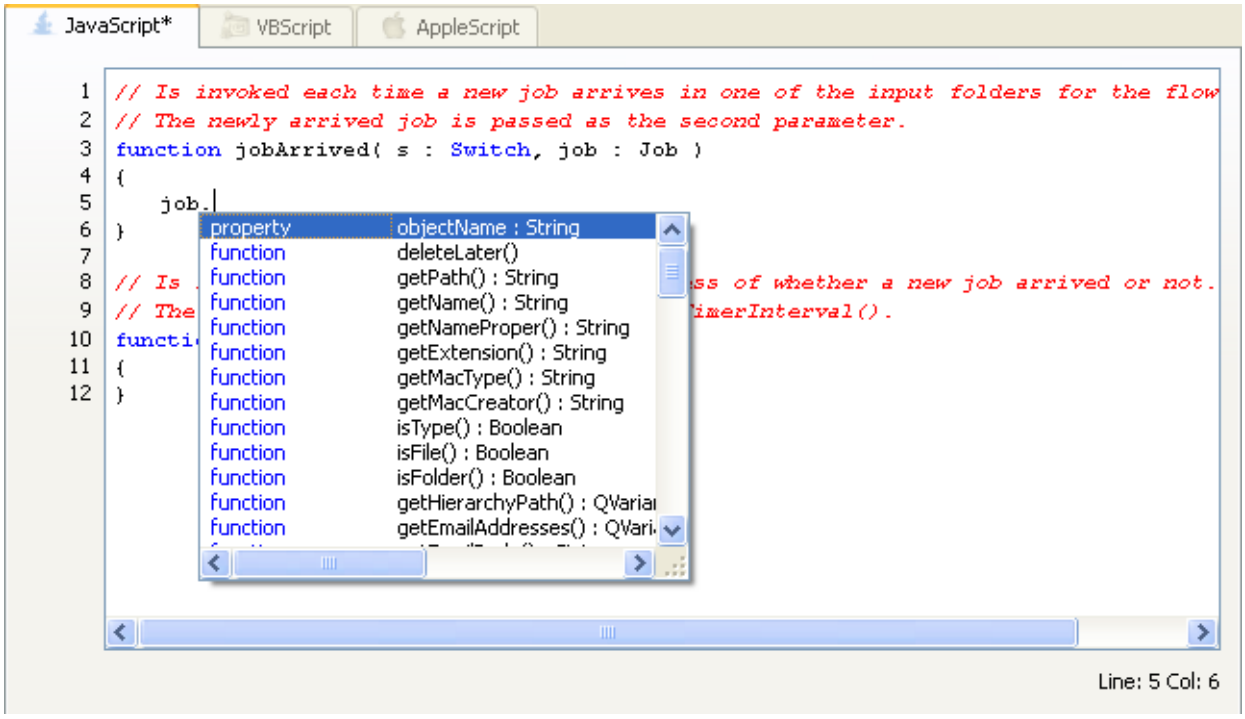
Use the small tool buttons at the top of the Fixture pane to add, remove and reorder connection definitions. In the example above, there are two outgoing connection definitions ("Accepted" and "Rejected").

The following table describes the properties displayed in the Properties pane when you select one of the connection definitions in the Fixture pane.

Property	Description
Connection name	The name of the outgoing connection
Enabled	Defines whether this outgoing connection is indeed included in the simulated test environment (can be used to temporarily exclude a connection)
Hold files	If set to "Yes", the connection is on hold
Include these jobs Exclude these jobs	For filter connections, determines which files flow along this connection
Include these folders Exclude these folders	For folder filter connections, determines which folders flow along this connection

Property	Description
Carry this type of files	For traffic-light connections, determines whether this connection carries data files or log files
Success out Warning out Error out	For traffic-light connections, determines whether a file moves along the connection depending on its success status
Injected connection properties	Additional outgoing connection properties as specified in the Declaration pane

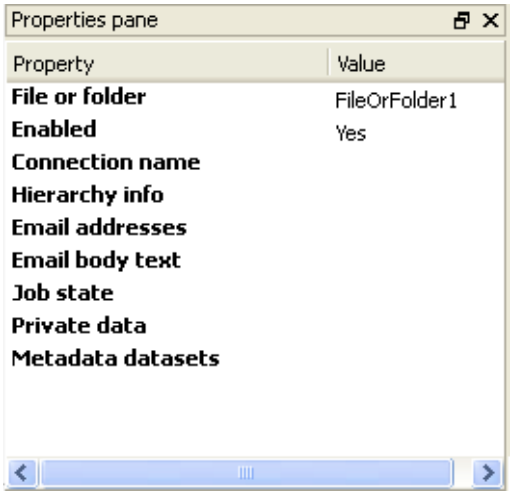
4.1.6. SwitchScripter Program pane



The Program pane offers a separate tab for each of the supported scripting languages (JavaScript, VBScript). Each tab contains a multi-line text editor for entering a script program in the corresponding language. The text editors use syntax coloring appropriate for the language being entered. The JavaScript editor also supports function name & argument completion as shown above.

You can select the scripting languages being included in the script package (and thus enable/disable the corresponding tabs) by setting appropriate values for the "include xxxScript" properties in the Declaration pane. Also see *Writing a script* on page 29.

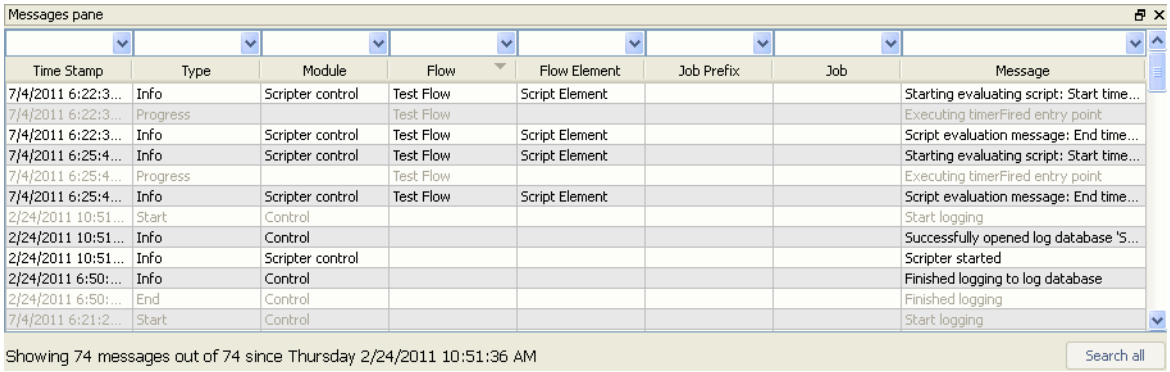
4.1.7. SwitchScripter Properties pane



The Properties pane allows viewing and editing the properties for the currently selected item in the Declaration pane or in the Fixture pane. The properties for each item are described in the Declaration pane and Fixture pane topics.

This pane is very similar to the Properties pane in the Switch designer.

4.1.8. SwitchScripter Message pane



The Messages pane displays a list of log messages generated by test runs of the script (see testing a script). You can filter and sort the messages using the controls at the top of the pane.

This pane is very similar to the Messages pane in the Switch designer.

4.2. Writing and testing a script

4.2.1. Writing a script

SwitchScripter's primary function is to create and edit script packages for use with Switch. A script package is an archive file (with the ".sscript" filename extension) that contains a script declaration, one or more alternate script programs and an optional icon for the script.

4.2.1.1. Creating a new script

To create a new script package for Switch from a blank sheet:

1. Launch SwitchScripter or, if it is already running, click **File** > **New**.
2. In the Declaration pane, select the first element, i.e. the root element (representing the script as a whole).
3. In the Properties pane, select the **Type** of script you want to create:
   - **Script** (if you want to create a script or a configurator)
   - **App** (if you want to create a Switch app for the Switch App Store)
4. Determine the required scripting language(s):
   - **Include JavaScript**: Yes/No (default value: Yes)
   - **Include VBScript**: Yes/No

   See also *Selecting a scripting language* on page 30
5. Configure the other script declaration properties, including supported connections and injected properties.
6. The Program pane has been initialized with a template (empty stubs) for the two most important entry points supported by the scripting API. Enter your custom program text inside the function body of the entry point(s) you need, remove the stubs for entry point(s) you do not need and add any other entry points you may need.
7. Click **File** > **Save script** to save your script package with an appropriate name in a location of your choice.
8. Verify the operation of your script as explained in *Testing a script* on page 31, adjust the script and its declaration as desired and save the final result.

4.2.1.2. Starting from an existing script

In most cases it makes a lot of sense to start developing the script from an example obtained from the Enfocus website or from another Switch user (see *http://forum.enfocus.com*).

To start developing your script package from an existing example:

1. Copy the example script package file under a new name (so that the original is preserved); do this in the system's file browser or by using the **Save script as** menu item in SwitchScripter.
2. Double-click the copy to open it in SwitchScripter.
3. Adjust the configuration in the Declaration pane if needed (if the example is very similar to the script, change the script's name and perhaps its icon).
4. Adjust the program text in the Program pane as desired.
5. Click **File** > **Save script** to save the script package.
6. Verify operation of the script as explained in *Testing a script* on page 31, adjust the script and its declaration as desired, and save the final result.

4.2.1.3. Using a script in Switch

To use a script package in Switch:

1. Create and save the script package in SwitchScripter.
2. Drag a script element onto the canvas in Switch.
3. Set the script element's "Script package" property to the file path of the script package.
4. Add the appropriate connections and configure any relevant properties for the script element and its connections.
5. Activate the flow that contains the script element.

Switch retains just a reference to the script package (that is, it remembers the file's path rather than the file itself). Each time the flow containing the script element is deactivated and activated, the script package is reloaded afresh from the file. This means that the script package can be modified and tried again in a very short cycle.

Note: In contrast, when a flow is exported, the exported ".sflow" file contains a full copy of the script package file (that is, it is distributed with the flow).

4.2.1.4. Script expressions

A script expression is a brief JavaScript program used to calculate the value of a flow element property. Users can enter a script expression in a property editor dialog within Switch, that is they do not need SwitchScripter. However, SwitchScripter provides some functions to help test more complex script expressions. See *Testing a script expression* on page 32.

4.2.1.5. Selecting a scripting language

Switch supports a couple of scripting languages (JavaScript, VBScript) as explained in the *Scripting reference* on page 60.

In general, use JavaScript unless there is a good reason to select an operating system scripting language (like VBScript). JavaScript scripts are cross-platform, that is, they work without

change on both Mac OS X and Microsoft Windows, and most of the Switch scripting examples available on the web or from other users will be in JavaScript.

At the same time, JavaScript scripts have limited "external access". For example, there is no way to communicate with interactive applications and there is no access to operating system services. If these features are necessary, on Microsoft Windows, use VBScript. To use these features on Mac OS X, the recommended way is to write JavaScript that uses the 'osascript' utility. This utility can execute dynamically generated AppleScript scripts to perform platform-specific tasks that cannot be performed in a different way.

If a script package contains more than one script program, Switch selects one of the programs for execution by looking for them in this order:

- The operating system scripting language for the platform on which Switch is running (that is, VBScript).
- JavaScript.

If neither of these are included, the script package cannot be executed.

4.2.2. Testing a script

SwitchScripter provides a simulated run-time environment for script testing purposes, avoiding the need for moving back and forth between the Switch designer and SwitchScripter while developing a script. The set of boundary conditions describing the test environment for a particular script is called a fixture.

4.2.2.1. Fixture package

A fixture package is an archive file (with the ".sfixture" filename extension) that contains the boundary conditions describing the environment for testing a script. Its contents can be configured using the Fixture pane in SwitchScripter.

A fixture package is always stored alongside the script package (with the same filename but a different extension). The fixture is not part of the definition of a script and may be removed after the script has been tested.

A fixture package often contains references to test files. It just remembers the file paths, and never includes the test files themselves. The test files are externally maintained by the script writer/tester.

4.2.2.2. Testing a script

To test a script in SwitchScripter:

1. Create an initial version of your script as described in *Writing a script* on page 29.
2. Save the script package with an appropriate name in an appropriate location.
3. Click **View** > **View declaration/fixture pane** to reveal the Fixture pane.
4. In the Fixture pane, configure the basic fixture settings, the incoming jobs and the outgoing connections to reflect a relevant test scenario. If the script accesses information in the internal job ticket of incoming jobs, setup the corresponding test data as well.

5. To simulate the arrival of a job on one of the script's incoming connections:
   a. Select the desired job in the Fixture pane (make sure it is enabled).
   b. Select the "jobArrived" entry point using the selector in the toolbar and press the "invoke entry point" tool button.

   The selected job is passed to the script's "job" argument and all enabled jobs are returned when the script requests a complete list of incoming jobs. Disabled jobs are hidden from the script.

6. To simulate a timer event, select the "timerFired" entry point using the selector in the toolbar, and click the "Invoke entry point with the test fixture" tool button.
   Again, all enabled jobs are returned when the script requests a complete list of incoming jobs.

7. In the Messages pane, review the messages issued by SwitchScripter and by the script to evaluate the script's operation. Also, use the Environment.log() and Job.log() functions to generate messages for debugging.

8. If desired, adjust the script program and/or the script declaration and start over.

Other entry points

Test any of the supported entry points with a similar procedure. If an entry point that requires an extra argument (such as a property tag, for example) is selected, SwitchScripter displays a dialog for entering the argument value when the "Invoke entry point with test fixture" button is clicked.

Execution mode

SwitchScripter ignores the script's execution mode property. Entry points are always invoked one at a time and data is not persistent between entry point invocations.

For a script, users can select two types of execution modes:

- Serialized
- Concurrent

4.2.2.3. Testing a script expression

A script expression is a brief JavaScript program used to calculate the value of a flow element property. Even though users can enter a script expression in a property editor dialog within Switch, they still need a SwitchScripter to test a complex script expression:

1. Create a new empty script package and save it with an appropriate name in an appropriate location.

2. Enter the script expression in the JavaScript tab of the Program pane (leave the default entry points in place but it is cleaner to remove them).

3. In the Fixture pane, configure the basic fixture settings and at least one incoming job to reflect a relevant test scenario. If the script expression accesses information in the internal job ticket of its job, setup the corresponding test data as well.

4. Select the desired job in the Fixture pane (make sure it is enabled), select "script expression" in the entry point selector in the toolbar, and click the "Invoke entry point with the test fixture" tool button.
   The selected job is passed to the script expression's predefined "job" variable.
5. In the Messages pane, review the message issued by SwitchScripter containing the result of evaluating the script expression.

   Tip: Use the Environment.log() and Job.log() functions to generate messages for debugging.

6. If desired, adjust the script expression and start over.
7. Once the script expression works correctly, copy/paste it into the appropriate property editor dialog within Switch.

4.3. Developing a scripted plug-in

4.3.1. Obtaining permission

Enfocus may license selected third-party vendors to develop scripted plug-ins or configurators for Switch, based on commercial considerations.

In most cases, a third-party vendor has to submit the (unprotected) script package to Enfocus for code review and quality assurance. Upon approval, Enfocus will save the scripted plug-in with the appropriate password.

Certain third-party vendors may be handed a scripted plug-in password (after signing the required licensing agreement), so that they can create a scripted plug-in by themselves.

If you are interested in developing a scripted plug-in or configurator for Switch, please contact Enfocus.

Scripted plug-ins and product flavors

Depending on the password used to save a scripted plug-in, it may be enabled for use with any of the Switch product flavors. In contrast, regular script packages are supported only in Switch.

When loading a scripted plug-in from the "scripted-plugin" or "third-party-plugin" folder, Switch verifies the password with which the plug-in was saved and acts according to the description in the following table:

Table1: Load action

Password	Plug-in type	Load action
Not a scripted-plug password or not valid for the current Switch flavor	Not allowed	Does not load the Plug-in

Password	Plug-in type	Load action
Valid scripted plug-in password but not one of the built-in passwords	Third-party	Loads the plug-in with the following changes: • Always puts it in the custom section • If the plug-in requested a section other than custom or if its name clashes with a built-in tool or configurator, adds an underscore in front of its name
One of the built-in passwords	Built-in	Loads the plug-in in the requested section

4.3.2. Creating a scripted plug-in

Introduction

A scripted plug-in is a regular script package that has been saved with a special password and is placed in the appropriate folder, so that it can be located and loaded by Switch. Once successfully loaded, the scripted plug-in's icon appears in the Elements pane and it can be used just like any other built-in tool or configurator.

The scripting API offers several entry points, properties and functions to support optional features that are specific to scripted plug-ins (for example, centralized management of the path to a third-party application). Since none of these features are required, a regular script package can be turned into a scripted plug-in in a matter of seconds.

Furthermore, all scripted plug-in features can be developed and tested using SwitchScripter. The only thing which cannot be done without an appropriate password is loading the scripted plug-in in the Switch Elements pane.

Password

Enfocus may license selected third-party vendors to develop scripted plug-ins; see *Obtaining permission* on page 33. In most cases, users have to submit their (unprotected) script package to Enfocus for code review and quality assurance. Upon approval, Enfocus saves the scripted plug-in with the appropriate password.

If users have been handed a scripted plug-in password (and signed the required licensing agreement), they can create a scripted plug-in by entering the password in the main script properties in SwitchScripter Declaration pane.

4.3.2.1. Location

A scripted plug-in must be located in the special "scripted-plugins" folder located near the Switch Server executable. Each plug-in consists of a folder (inside the "scripted-plugins" folder) that contains a single script package (with the filename extension .sscript) and any additional resources used by the scripted plug-in.

Scripted plugins created by anyone other than Enfocus must be located in a folder called "third-party-plugins" located next to the regular "scripted-plugins" folder. After a fresh install this folder is empty.

The Switch installer places a number of built-in scripted plug-ins in the appropriate location.

Accessing resources

A scripted plug-in can obtain the absolute path to its plug-in folder through the getSpecialFolderPath("PluginResources") function offered by the Environment class. This allows a scripted plug-in to access resources contained in its plug-in folder (that is, next to the script package).

4.3.2.2. Support for third-party application

Switch offers special support for scripted plug-ins that control a third-party application. The reason is that information about the third-party application's location and licensing should be managed centrally. Rather than offering a property for each instance of the script in a flow, the scripted plug-in's icon in the Elements pane offers a context menu item to set the property value once and for all.

Moreover some scripted plug-ins can automatically discover the third-party application's location and this should happen only once since it may be a time-consuming process.

Special properties

If a script package is loaded as a scripted plug-in, Switch treats the property tags "ApplicationPath" and "ApplicationLicense" in a special way:

- Properties with these tags are not shown in the script element's occurrences in a flow.
- The property values are retrieved from a central memory location managed by the context menu items in the scripted plug-in's icon in the Elements pane.

If the script package is not loaded as a scripted plug-in, these properties are not treated in any special way.

Special entry points

If a script package is loaded as a scripted plug-in, the optional findApplicationPath, checkApplicationPath, licenseApplication, and getApplicationLicensing entry points help manage the special properties discussed above in conjunction with the context menu items.

If the findApplicationPath entry point is present, this means that the scripted plug-in controls an external application. If the entry point returns a non-empty string, this value is used as the application path stored in the central "ApplicationPath" property. Otherwise the application path remains empty, the scripted plug-in's icon in the Elements pane is grayed out and a context menu item is offered to set (or view) the application path manually.

The findApplicationPath entry point can return "/:External resource:/" if there is no application path available, for example if it is on a network server.

The findApplicationPath entry point can use the findRegisteredApplication(), findApplicationOnDisk() and find64BitApplicationOnDisk() functions offered by the Environment class to help discover the third-party application.

The checkApplicationPath entry point offers the possibility to check if the path can be specified as ApplicationPath.

If the licenseApplication entry point is present, this means that the application can be licensed through Switch. In this case the scripted plug-in's icon in the Elements pane offers a context menu item that displays a dialog requesting a license key. When the user enters a license key and closes the dialog box, the licenseApplication entry point is called and the license key is stored in the central "ApplicationLicense" property. This allows the script to perform licensing once (in the entry point, when the key is entered) or pass the license key with every execution (by retrieving the property value).

The getApplicationLicensing entry point serves to provide feedback about the third-party application licensing state through a context menu on the scripted plug-in's icon in the Elements pane.

Testing special properties and entry points

SwitchScripter does not emulate central management of these special properties and entry points. However it does allow testing the special entry points just like any other entry points, and it treats the properties as regular properties (so you can enter a value in the Fixture pane). Thus SwitchScripter fully supports developing and testing these special features; see *Testing a script* on page 31.

4.3.3. Configurator guidelines

This topic introduces the mechanisms used to develop a Switch configurator for a third-party application and it offers guidelines for adjusting third-party applications so that they can be optimally configured and controlled from within Switch.

4.3.3.1. Limited to scripting

Although many of the built-in Switch configurators have historically been developed in the C++ programming language, this topic discusses configurators solely in the context of scripting because:

- Third-party developers have access to the Switch scripting API and not to the C++ API.
- All new Switch configurators (including those developed by Enfocus) use scripting.

4.3.3.2. Definition of a configurator

Note: This topic describes the technical aspects of a configurator. For a a description of the concept and benefits of configurators, refer to the Switch Reference guide, available on the Enfocus website.

A configurator is a scripted plug-in that defines the findApplicationPath entry point and the special property "ApplicationPath". A configurator may (but does not have to) also define the checkApplicationPath (recommended), getApplicationLicensing entry point, the licenseApplication entry point, and/or the special property "ApplicationLicense".

See entry points for *Application discovery* on page 37 and *Application licensing* on page 38, *Creating a scripted plug-in* on page 34, and *Obtaining permission* on page 33.

For the avoidance of doubt:

- A script package assigned to a script element is not a configurator; its application discovery entry points are never invoked and its special properties behave as regular properties.
- A flow element implementation that does not define the findApplicationPath entry point is not a configurator.
- It is an error for a flow element implementation to define the findApplicationPath entry point while not defining the special property "ApplicationPath" (because the flow implementation would have no access to the application path it claims to require).
- It is meaningless (but not an error) for a flow element implementation to define the getApplicationLicensing or licenseApplication entry points without defining the findApplicationPath entry point since in that case the entry points will never be invoked.
- It is meaningless (but not an error) for a configurator to define the "ApplicationLicense" property without defining the licenseApplication entry point since in that case the property will never receive a value (there is no license context menu and the property is hidden).
- Conversely a configurator may define the licenseApplication entry point without defining the "ApplicationLicense" property in case it does not need access to the license string outside of the licenseApplication entry point.

4.3.3.3. Application discovery

Ideally a third-party application can be configured and used from within Switch immediately after it is installed, without the need for separately launching the third-party application (for example, to license it). This is not always feasible and Switch provides several fall-back options, but the objective is to approach this optimal situation.

Application discovery is implemented in the findApplicationPath entry point. The specified path will be checked by the checkApplicationPath entry point after upgrading the configurator or after setting the path manually.

Windows

The installer for the third-party application should make an entry in the system registry specifying the full path of the executable (or a folder that contains it). This registry entry should be clearly documented so that Switch can retrieve the information using the Environment.findRegisteredApplication() function.

Otherwise the third-party application should be installed somewhere inside the standard "Program Files" or "Program Files (x86)" directory, so Switch can search the contents of the "Program Files (x86)" directory for the name of the application using the Environment.findApplicationOnDisk() function (for search in "Program Files" use the Environment.find64BitApplicationOnDisk() function).

Mac OS

If the third-party application is in a regular ".app" bundle with an appropriate property list, Mac OS X will recognize the ".app" and add it to its central registry. Switch can then retrieve the information using the Environment.findRegisteredApplication() function.

Otherwise the third-party application should be installed somewhere inside the standard "Applications" folder, so Switch can search the contents of the Applications folder for the name of the application using the Environment.findApplicationOnDisk() function.

Manual

On both platforms, if Switch does not find the third-party application, the user is offered a "fall-back" option in the form of a context menu item on the configurator's icon in the Elements pane, which allows manually browsing the file system for the application.

4.3.3.4. Application licensing

Independent

In this option the user licenses the third-party application independently from Switch through some user interface provided by or with the third-party application. For example, by entering a license key in a dialog box or on a command line.

Although it violates the objective of allowing the application to be fully controlled from within Switch, this may be the most logical option for regular interactive applications that offer a rich user interface aside from the automation capabilities via Switch.

In Switch

Some third-party applications do not have a graphical user interface and are intended mostly for integration with products such as Switch. Since the user never directly interacts with the third-party application it is more logical to allow licensing from within Switch.

In this option the user enters a license key in a Switch dialog box that is accessed through a context menu item on the configurator's icon in the Elements pane. Switch can pass the license key on to the third-party application at the time the user enters the license key (the most frequently used option) and/or each time an action is executed. The third-party application should document the licensing process for the configurator.

Application licensing is implemented in the licenseApplication entry point.

Licensing state

Assuming that it is possible to automatically detect the licensing state of a third-party application, Switch can display this information through a context menu item on the configurator's icon in the Elements pane. This feedback is meaningful regardless whether the licensing operation itself can be performed from within Switch or not. Thus wherever feasible the third-party application should document a way for the configurator to detect licensing state.

Detecting licensing state is implemented in the getApplicationLicensing entry point.

4.3.3.5. Configuring properties

Flow element properties

Each configurator instance in a Switch flow offers properties to control the behavior of the third-party application for jobs passing through that instance. A property can have various simple data types such as string, number or enumeration, it can reference a named collection of properties (see property sets below), or it can be a value edited by a wide range of property editors. These properties are defined in the script declaration.

Property values for each configurator instance are entered in the Switch designer's Properties pane and they are stored (and exported/imported) with the flow definition.

Preferences and persistent settings

The third-party application may offer preferences or other settings that are persistent between invocations and cannot be controlled from Switch. This is no problem as long as the user has a way to influence these settings independently from Switch, for example through a user interface that is part of or ships with the third-party application.

4.3.3.6. Property sets

A property set is a collection of controls, settings or options used to configure a third-party application. A property set usually combines a large number of options and it vastly simplifies the user interface required in Switch for configuring the application because it can be referred to as a single entity.

Referring to a property set

Depending on how a property set is stored by the third-party application, Switch keeps a different type of reference to the property set – as illustrated in the following table.

Storage model	Reference in Switch	Stored in exported flow
As a regular file anywhere in the file system	Absolute file path	Full copy of the file
In an open repository (folder) controlled by the third-party application but directly accessible to Switch as a regular file	Absolute file path	Full copy of the file
In a private repository (database) controlled by the third-party application and only accessible by Switch by name and through the third-party application	Name	Name

Some applications support multiple storage models for the same property (that is, the property can be specified as a named repository item or as a file path). Switch configurators can easily accommodate this by providing multiple property editors for the property.

Selecting a property set

Switch allows a choice between browsing for a file (the first storage model) or selecting from a list of names (the second and third storage models). Note that there is no way to show a tree structure; only a linear list of names.

The following table describes the implementation mechanism and the cooperation required from the third-party application depending on the storage model.

Storage model	Action in Switch	Implemented through	Cooperation required from application
As a regular file in the file system	Allow the user to browse the file system for a property set	"Choose file" property editor	"Choose file" property editor

Storage model	Action in Switch	Implemented through	Cooperation required from application
In an open repository as a regular file	List all of the appropriate files in the repository folder and show them in a dialog after stripping the filename extension	"Select from library" property editor with getLibraryForProperty entry point that iterates over the repository folder	Document the location of the repository folder
In a private repository, by name	Request the list of names from the third-party application and show them in a dialog box	"Select from library" property editor with getLibraryForProperty entry point that interacts with the third-party application	Provide a run-time mechanism to retrieve the list of names on request

4.3.3.7. Communication requirements

The Switch configurator needs a mechanism to communicate with the third-party application with the following basic requirements:

- Pass on the values of the properties to configure the behavior of an action.
- Start an action on some input file(s).
- Detect termination of an action.
- Locate the output file(s).
- Determine success/warning/error status.

Error handling

The third-party application should clearly document how Switch can discriminate between:

- Successful execution versus failure to execute an action.
- Messages that should be passed on to the Switch execution log versus messages that are irrelevant to the Switch user (or output that can safely be ignored).
- The "level" of messages passed on to the Switch execution log: info/warning/error.

Serialized communication

If required, Switch can serialize all communication with a third-party application, even if many instances of the same configurator are active simultaneously. See *Execution modes* on page 46 for more information.

Switch can also automatically terminate a third-party application if an action does not complete within a certain amount of time (which is configurable as a user preference).

Platform considerations

If a third-party application is available on both Mac and Windows, it is acceptable (but not necessarily preferable) to use a different communication mechanism on each platform. For example, the Adobe Acrobat Distiller configurator uses COM on Windows and AppleScript on

Mac. In other words, a Switch configurator can include platform-specific code, as long as the user's view of the configurator's properties and behavior is the same on both platforms.

Platform-specific behavior is achieved by including an implementation in two different scripting languages (see script package and selecting a scripting language) or through the Environment.isMac() and Environment.isWindows() functions.

4.3.3.8. Communication mechanisms

Communication with the third-party application is implemented mostly (or solely) in the configurator's jobArrived entry point.

Command line

From an implementation standpoint this is often the easiest form of communication. The Switch configurator compiles command line options based on the configurator properties and if necessary Switch provides console input and/or parses console output. Alternative sources of information may be the application's return code and log files written by the application.

Inter-application communication

Applications that offer a graphical user interface (and thus are often launched by the user independently of Switch) can be controlled using inter-application communication mechanisms (usually COM in VBScript on Windows and AppleScript on Mac).

Hotfolder

A hotfolder application could be driven from a Switch configurator on the condition that it is possible to configure the application's behavior using a single watched folder and a control file. There can be several setups including:

- The control file is placed in a watched folder and points to the path of the input file(s).
- The input file and the control file are both placed in the same watched folder (and they are matched through some filename pattern).

A Switch configurator can NOT control an application that requires a different folder watched for each configuration setup.

Preferably the third-party application and Switch should be able to settle on a watched folder path without requiring user setup in the third-party application. The best way to achieve this is to provide a mechanism that allows Switch to set the watched folder path at its discretion. Failing that, the third-party application should document a mechanism for the configurator to retrieve or otherwise determine the watched folder path.

Loadable library

Third-party functionality that is offered as a loadable library (DLL on Windows, Sylib on Mac) can be accessed in a Switch configurator by providing an independent helper application. The helper application calls the library functions and provides a simple interface towards Switch (usually command line and/or console input/output).

4.3.3.9. Text encoding issues

On modern operating systems – including Windows XP and Mac OS X – file and folder names may contain any Unicode code point (except for a few platform-specific separator characters).

This is true even if the default 8-bit code page is not able to represent these characters. For example, on a regular English Windows XP system it is perfectly possible to have Greek or Cyrillic characters in a filename, although these characters cannot be represented in the default Latin code page.

Switch is fully Unicode enabled and for optimal operation it requires a configured third-party application to be Unicode enabled as well. In other words Switch requires that the third-party application:

- Correctly works with filenames that may contain any Unicode code point.
- Performs all text communication with Switch using an appropriate and well-defined character encoding (as explained in more detail below).

Command line on Mac OS X

Mac OS X uses UTF-8 as its default encoding for representing filenames/paths. In our experience, a command line application can simply take a file path from the command line as an 8-bit string and pass it through to a regular UNIX or Mac OS file system call even if the command line application itself is not Unicode aware. Since the functions of the Switch Process class use UTF-8 to invoke command line applications, things will automatically work correctly.

Command line on Windows

On Windows the functions of the Switch Process class use Windows-specific Unicode-enabled function calls to invoke command line applications. The command line application in turn MUST invoke the Windows-specific Unicode-enabled function calls for retrieving the command line AND for opening the files. For example, in C/C++ the command line application must use the Unicode ("wide") version of the Windows-specific "GetCommandLine" function rather than the argv[] argument in the main function (which only supports the current default code page).

Other text input/output

This includes the console input/output streams and any exchanged control files that may contain plain text (as opposed to XML which has built-in Unicode support).

It is strongly recommended to use UTF-8 for all text input/output because this encoding can represent any Unicode code point and it is upwards compatible with 7-bit ASCII in various ways (for example, with regards to line breaks and null-terminators).

If this is not feasible, at the very least the encoding used must be well-defined and documented, and it should follow these guidelines:

- Text that may contain a filename/path must be able to represent any Unicode code point. In essence UTF-8 or UTF-16 are the only options.
- Text that may contain code points outside 7-bit ASCII (for example, localized messages for human readers) must be in a well-defined encoding that is able to represent the characters in the languages used. Preferably this encoding is the same on all platforms; or at least it is well defined. Again UTF-8 is the best option.
- Text that only contains 7-bit ASCII code points (for example, keywords that are not localized and are not intended for human consumption) is not encoding-sensitive. Still it does not hurt to use UTF-8 since it is compatible with 7-bit ASCII.

4.4. Script declaration

4.4.1. Main script properties

The following table describes the main script properties contained in the script declaration, which is part of a script package and is edited using the declaration pane in SwitchScripter.

Property	Description
Name	The name of this script for identification to a user For scripted plug-ins, a tilde should separate the company name from the tool name (for example, "Adobe~Acrobat Distiller"); Switch will derive appropriate long and short versions from this name.
Type	Eement type. Choices are: • App • Configurator • Script (= regular script, for example for use with the Script element in Switch) **Note:** The **Configurator** option is only available if you have a valid Configurator SDK key activated on your computer. If selected, a number of additional properties become available (Sub category in elements pane, Compatibility, Support Info, Application Discovery Details)
Include JavaScript	If set to Yes, the script package includes a JavaScript program and the corresponding tab in the program pane is enabled. This property is dependent on the Type property.
Include VBScript	If set to Yes, the script package includes a VBScript program and the corresponding tab in the program pane is enabled. This property is dependent on the Type property.
Keywords	For scripted plug-ins, defines a list of zero or more keywords, separated by a space, a comma and/or a semicolon. When filtering elements in the Elements pane with the "Search keywords" menu item turned on, this element is displayed if the filter text matches the beginning of at least one of the keywords in this list.
Tooltip	For scripted plug-ins, determines the tooltip shown for the icon in the elements pane
Icon	The icon to be displayed for a flow element associated with this script package; this must be a PNG image file (not interlaced) with a size of exactly 32x32 pixels in the RGB color space (with support for transparency).

Property	Description
Password protected	If set to Yes, the script package is protected by a password; the password is asked when the script package is opened for viewing and editing in SwitchScripter; a script package can always be opened for execution by Switch, even if it is password protected See also password protection
Password Verify password	Defines the password for the script package if it is password protected
Incoming connections	If set to Yes, the script supports incoming connections
Require at least one	If set to Yes, the script requires at least one incoming connection
Outgoing connections	Defines to what extent the script supports outgoing connections: "No", "One", or "Unlimited"
Require at least one	If set to Yes, the script requires at least one outgoing connection (for traffic-light connections: at least one outgoing connection that carries data or data with logs)
Connection type	The type of outgoing connections supported by the script, if there are any; choices are: • Move • Filter • Traffic-light
Include/exclude folder	For outgoing filter connections, if set to Yes, the standard include/exclude folder properties are automatically injected in the outgoing connections
Data success Data warning Data error	For outgoing traffic-light connections, defines which of the standard success, warning and error properties is injected in the outgoing "data" and "data with log" connections
Log success Log warning Log error	For outgoing traffic-light connections, defines which of the standard success, warning and error properties is injected in the outgoing "log" connections
Execution mode	The execution mode for this script; one of these choices: • Serialized: entry points for all script instances in the same execution group are never invoked concurrently • Concurrent: different instances of the script may be executed concurrently

Property	Description
Execution group	The name of the execution group in which instances of this script reside; the precise interpretation depends on the selected execution mode: • Serialized: determines the scope of serialization • Concurrent: not used
Enable performance tuning	If set to Yes, additional performance tuning is enabled for the script package. When using the script in Switch, an extra property "Idle after job (secs)" will be available, which can be used to fine-tune the flow's performance. If script execution mode is also set to "Concurrent", another property, "Number of slots", will become available.
Idle after job (secs)	When performance tuning is enabled, this property contains the default number of seconds the script should remain idle after processing a job. The user can override this value when designing a flow.
Number of slots	The number of slots that can be used concurrently by the specific flow element. Possible values are: "0" (unlimited), "Default" (value from Preferences is used) or any number larger than 0. Note that this property can be overridden by the value specified in Preferences > "Concurrent external processes".
Position in element pane	For scripted plug-ins, determines the position of the icon in the elements pane; the value of the property has the following format: • <section>:<sort-key> Where: • <section> is the name of the elements pane section in which the scripted package should appear: Basics, Tools, Communication, Processing, Metadata, or Custom. If it is not one of these, the section defaults to "Custom". • <sort-key> is a text string that determines the order in which elements are listed within each section; they are sorted alphabetically on the sort key.
Obsolete properties	The list of property tags which were present in the previous versions of the configurator and were removed in the current version. For these tags, Switch does not display the warning messages during flow import or update. See *Flow element update* on page 172 for information on entry points available.
Obsolete connection properties	The list of connection property tags which were present in the previous versions of the configurator and were removed in the current version. For these tags, Switch does not display the warning messages during flow import or update. See *Flow*

Property	Description
	element update on page 172 for information on entry points available.

4.4.2. Execution mode

Switch execution is inherently multi-threaded, so in principle all tasks can run in parallel. To help reduce implementation complexities (both in the Switch framework and in a script), concurrency is limited in a number of ways as explained in this topic.

Switch severely limits the concurrency of VBScript scripts (on Windows): only a single VBScript entry point can be executing at any time. This is due to an implementation limit which may be lifted in a future version. Thus the remainder of this topic applies only to JavaScript scripts.

Script expressions

Script expressions may be evaluated in parallel with each other and with other scripts. Script expressions should not have side effects (other than issuing log messages) so this parallelism should not cause any problems.

Terminology

In the following discussion it is important to differentiate between a script package (that is, the definition of a script) and the script instances created by associating a script package with a flow element (through a script element or a scripted plug-in). There may be several script instances for a particular script package.

4.4.2.1. Execution modes

The script declaration defines the execution mode for all instances of the script package as one of these choices:

- Serialized: entry points for all script instances in the same execution group are never invoked concurrently.
- Concurrent: entry points for the same or different script instances of the script may be executed concurrently.

Concurrent

In concurrent execution mode, the entry points for the same or different script instances of the script may be called concurrently. In other words, two or more script instances of the same type may run in parallel and even two or more entry points for a particular script instance may run in parallel (for example, the jobArrived and timerFired entry points).

For example, a flow containing a single flow element with a concurrent script (in addition to input/output folders) will pick up and process multiple input jobs at the same time – within the overall processing limits configured through user preferences.

Script implementations with concurrent execution mode must take care to synchronize access to resources that are shared between script instances or between different entry points. For example, access to a text file that is updated from both the jobArrived and timerFired entry points should be synchronized using the Environment.lock/unlockGlobalData() functions.

Serialized

In serialized execution mode, entry points for script instances in the same execution group are never called concurrently. In other words, script instances in the same execution group are executed one after another. Still, the entry points in script instances outside of the execution group may be executing in parallel with those in the group.

Script implementations that need to be serialized between each other (perhaps because they use a common external resource) should refer to the same execution group. Usually however a script implementation needs serialization only with itself. In that case, the execution group name should be unique to the script implementation.

4.4.2.2. Execution group

The script declaration defines the execution group for all instances of the script package. The execution group is a string that should be structured as a reverse domain name (similar to Java packages) to avoid collision with other developer's group names; for example: "com.gradual.myProject.myExecutionGroup".

If the value of this property is empty, the execution group name defaults to the script's name (the first property in the scripter's declaration pane). This causes the script package to be in its own execution group unless another script package declares the same script name (that is, there is no strong protection against name collisions).

The precise interpretation of the execution group depends on the selected execution mode.

Concurrent

With concurrent execution mode, the execution group is not used.

Serialized

With serialized execution mode, a name collision between execution groups is mostly harmless in the sense that nothing breaks, but there is a potential performance issue due to the fact that the inadvertently colliding script instances can't run concurrently. Thus, it is strongly recommended that scripts intended for wide deployment provide a unique execution group name structured as described earlier.

4.4.2.3. Selecting the appropriate execution mode

The following table provides an overview of the execution modes and their typical uses.

Execution mode	Instances in same group are serialized	Information can be preserved across invocations	Example usage
Concurrent	No	No	Gather metadata from the internal job ticket of a job and produce an XML file (all done in scripting)

Execution mode	Instances in same group are serialized	Information can be preserved across invocations	Example usage
Serialized	Yes	No	Control a command-line application that is launched and terminated within each invocation of jobArrived

4.4.3. Property definition properties

The following table describes the properties for each property definition contained in the script declaration which is part of a script package and is edited using the declaration pane in SwitchScripter. A property definition may describe a property injected into the flow element to which the script package will be attached or a property injected into its outgoing connections.

Property	Description
Tag	The internal name of the property as it will be referred to from the script
Name	The name of the property as it will be displayed in the designer's Properties pane
Tooltip	A brief description of the property which will be shown in the designer's Properties pane as a tool tip
Inline editor	The inline property editor used to edit this property in the designer's Properties pane, if any; see inline property editors for a list of choices and further information
Editor 1	Extra property editors shown for this property in the designer's Properties pane, if any; see other property editors for a list of choices and further information This allows for a maximum of five property editors plus the inline editor. There must be at least one editor (if all editors are set to "None" a single-line text editor is automatically provided). Note that it is meaningless to specify the same editor twice (the second occurrence is ignored), except the external editor; it is possible to use multiple external editors for one script.
Editor 2	
Editor 3	
Editor 4	
Editor 5	
Default	The default value for the property
Depends on master	If set to yes, this property is displayed only if its master property has a particular value; the master property is the nearest preceding property with "Depends on master" set to "No" Only a single level of dependency is supported (a dependent property can never be a master)

Property	Description
Show if master equals	The string value or values of the master for which this property is displayed; multiple values must be separated by a semicolon
Validation	The validation method for the value of this property, which is one of: • None: perform no validation for this property • Standard: perform standard validation for this property depending on the property editor used to enter the value • Custom: invoke the script's isPropertyValid entry point to validate this property; do not perform standard validation • Standard and custom: first perform standard validation for this property, and if successful then also perform custom validation See validating property values for further information

4.4.4. Property editors

Property values

Regardless of which property editor is used to edit an injected property, the property value is always accessed from the script as a string or a string list (using the getPropertyValue() and getPropertyValueList() functions, respectively). The tables below describe the values for each supported property editor.

Inline property editors

The "Inline editor" property of a property definition offers the choices listed in the following table. These property editors are contained inside the property value field in the designer's properties pane. See working with properties for information on using property editors.

Property editor	Resulting value
Single-line text	The text line entered in the inline editor, as a string
Password	The hidden value entered in the inline editor, as a string (this is the same as single-line text but the value is hidden)
Number	The decimal integer number entered in the inline editor, as a string (this is the same as single-line text but the user can enter only digits)
Hours and minutes	A string formatted as "hh:mm" (including the colon) where h and m stand for a decimal digit; hh and mm include a leading zero if needed (this is the same as single-line text but the user can enter only 4 digits)

Property editor	Resulting value
No-yes list	One of the strings "No" or "Yes"
Dropdown list	The selected item (that is, one of the provided custom strings); see extra properties below

Other property editors

The "Editor 1/2/3/4/5" properties of a property definition offers the choices listed in the following table. Most of these property editors are modal dialogs. See working with properties for information on using property editors.

Property editor	Resulting value
Literal	The fixed string value corresponding to the name of the property editor, which must be selected from a predefined list; see extra properties below
Choose file	The selected absolute file path, as a string
Choose folder	The selected absolute folder path, as a string
Regular expression	The regular expression entered in the dialog, as a string
File type	The file type selected from a dialog with standard file types, as a string with the corresponding Windows filename pattern(s)
Select from library	The string value selected from a list in a "library" dialog; the contents of the dialog is determined by calling the getLibraryForProperty entry point in the script
Multi-line text	The text entered in the dialog as a single string that may include newlines
Script expression	The result of the script expression evaluated in the context of the job, converted to a string under the JavaScript rules
Single-line text with variables	The text line entered in the dialog, as a string, after replacing any variables by their values in the context of the job
Multi-line text with variables	The text entered in the dialog box as a single string that may include newlines after replacing any variables by their values in the context of the job
Condition with variables	The result of the conditional expression after replacing any variables by their values in the context of the job, as one of the strings "true" or "false"
File patterns	The list of pattern strings as they have been entered in the dialog box

Property editor	Resulting value
Folder patterns	The list of pattern strings as they have been entered in the dialog box
File types	The list of file types selected from a dialog with standard file types, each as a string with the corresponding Windows filename pattern(s)
String list	The string values entered in a generic string list dialog; each line is represented is a separate string
Select many from library	The string values selected from a list in a "library" dialog; each selected item is represented as a separate string; the contents of the dialog is determined by calling the getLibraryForProperty entry point in the script
External editor	The file path to a property set edited by a third-party application; see external property editor

4.4.4.1. Extra properties for certain property editors

The following extra properties are shown just after the "Editor" properties only when a particular property editor is chosen.

Dropdown list

Property	Description
Dropdown items	The list of choices to be offered in the dropdown list

Choose file

Property	Description
Include file for export	If set to yes, a copy of the file indicated by this property's value (a file path) is included in the exported flow archive when a flow with this script is exported

Literal

Property	Description
Literal editor name	The name of the literal property editor, that is, one of: Default, None, Automatic, No jobs, All jobs, All Other jobs, No Folders, All Folders

A literal editor can be used to enable a constant value with a specific meaning for a property. For example, if a property requires some file path, it also may allow specifying the 'Default' value by enabling the corresponding literal editor.

When a Switch user specifies 'Default' in this property, it means some default file will be used by the script. Another common use case is the 'None' literal editor, which is the expected way to define a property that may contain an empty value. The 'Standard' validation accepts this value and there is no need to completely disable validation to allow an empty value for the property.

The table below gives an overview of the values returned by the getPropertyValue() for the corresponding literal editors:

Property editor	Returns:
Default	"Default"
None	"" (empty string)
Automatic	"Automatic"
No jobs	"No Files"
All jobs	"All Files"
All other jobs	"All Other Files"
No folders	"No Folders"
All folders	"All Folders"

Before checking the value of the property in the script, it is necessary to check that the editor type is literal using the getPropertyEditor():

```
var theFilePath;
var theEditor = s.getPropertyEditor( "theProperty" );
var theValue = s.getPropertyValue( "theProperty" );
if( theEditor == s.EditorLiteral )
{
 if( theValue == "Automatic" )
 {
  s.log( 2, "Automatic file path" );
  // set some automatic value for theFilePath
 }
 else if ( theValue == "Default" )
 {
  s.log( 2, "Default file path" );
  // set some default value for theFilePath
 }
}
else
{
 theFilePath = theValue;
}
```

Select from library, Select many from library, String list

Property	Description
Message	A custom message displayed on the property editor dialog

External editor

See external property editor.

4.4.5. Validating property values

Introduction

The "Validation" property of a property definition specifies the mechanism for validating the values of this property.

In most cases, properties are validated while the flow is being designed or just before it is activated. If there are any invalid properties, Switch refuses to activate the flow. For performance reasons, Switch does assume that the result of design-time validation remains valid for the duration of flow activation. While this assumption is correct in most production environments, it is not strictly true for all data types; for example a file path becomes invalid when the target file is removed.

Properties that contain a variable or a script expression are validated while the flow is running, since their value cannot be determined at design time. Just before each invocation of the jobArrived entry point, Switch performs run-time validation for all properties of the flow element that contain a variable or a script expression. If a property value is invalid, Switch fails the job with an appropriate error message.

Note that there is NO run-time validation before calling the timerFired entry point (or any of the other entry points). Using non-static property values from those entry points is risky anyway since there is no job context (and referencing job context from a script expression or variable in those circumstances has undefined results).

Validation at design time

Validation of a particular property value is performed at design time if isPropertyValueStatic() returns true for that property.

The following table describes the standard design-time validation for values entered by a particular property editor. Note that all property values are of data type string or string list.

Property editor	Design-time validation requires that the value...
Single-line text	Is non-empty
Password	Is non-empty
Number	Is non-empty and contains only decimal digits
Hours and minutes	Is non-empty and has the format "hh:mm" (including the colon) where h and m stand for a decimal digit, hh and mm include leading zero if needed, hh is in range 00..23 and mm is in range 00..59
No-yes list	Is one of the strings "No" or "Yes" (exact spelling and case)
Dropdown list	Is one of the custom strings provided for the dropdown list
Literal	Is the fixed string value selected from a predefined list

Property editor	Design-time validation requires that the value...
Choose file	Represents a valid absolute path and that a file exists at the path
Choose folder	Represents a valid absolute path and that a folder exists at the path
Regular expression	Is non-empty and has valid regular expression syntax
File type	Is non-empty and has valid filename pattern syntax
Select from library	Is one of the strings returned by the getLibraryForProperty entry point in the script
Multi-line text	Is non-empty
Script expression	N/A
Single-line text with variables	N/A (if the value contains no variables: Is non-empty)
Multi-line text with variables	N/A (if the value contains no variables: Is non-empty)
Condition with variables	N/A
File patterns	Has at least one item and each item has valid filename pattern syntax
Folder patterns	Has at least one item and each item has valid filename pattern syntax
File types	Has at least one item and each item has valid filename pattern syntax
String list	Has at least one item and each item is non-empty
Select many from library	Has at least one item and each item is one of the strings returned by the getLibraryForProperty entry point in the script
External editor	Represents a valid absolute path and that a file exists at the path

Validation at run time

Validation of a particular property value is performed at run time if isPropertyValueStatic() returns false for that property.

In this case the property editor used to enter the source value (example: the script expression or text with variables) is no indication for the appropriate validation scheme. Therefore, as a general principle, standard run-time validation allows the computed property value to conform to the validation scheme of ANY of the property's property editors.

Specifically, the validation algorithm works as follows:

- Compile a list of validation schemes by adding the validation scheme from the following table for each of the property's property editors ("--" means do not add a scheme for this editor).
- The computed property value must be non-empty AND it must comply with at least one of the validation schemes in the list compiled above (unless the list is empty).

Property editor	Run-time validation scheme
Single-line text	--
Password	--
Number	Same as design-time scheme
Hours and minutes	Same as design-time scheme
No-yes list	Same as design-time scheme, or a Boolean value (*)
Dropdown list	Same as design-time scheme
Literal	Same as design-time scheme
Choose file	Same as design-time scheme
Choose folder	Same as design-time scheme
Regular expression	Same as design-time scheme, or a Boolean value (*)
File type	Same as design-time scheme
Select from library	Same as design-time scheme
Multi-line text	--
Script expression	--
Single-line text with variables	--
Multi-line text with variables	--
Condition with variables	--
File patterns	Same as design-time scheme (for one item), or a Boolean value (*)
Folder patterns	Same as design-time scheme (for one item), or a Boolean value (*)
File types	Same as design-time scheme (for one item), or a Boolean value (*)
String list	--

Property editor	Run-time validation scheme
Select many from library	Same as design-time scheme (for one item)
External editor	Same as design-time scheme

(*) a Boolean value is represented by one of the strings "true" or "false"

Validation example

Consider a property that specifies a property set which can be provided as a file or it can be part of an external library. The property is set to standard validation and it has the following property editors:

- Choose file
- Select from library
- Script expression
- Single-line text with variables

For the first two editors, validation is always performed at design time. For the last two editors, validation is performed at run time (except if the single-line text does not contain any variables). In all cases, the property value is guaranteed to contain one of the following:

- A valid file path that points to an existing file.
- One of the library items returned for the property by the getLibraryForProperty entry point.

For most use cases it can be assumed that these values are mutually exclusive, and that the script code can tell the data types apart from looking at the value. If needed however, the script code can check which editor has been used to enter the value.

4.4.6. External property editor

Introduction

Switch supports a property editor called "External editor" that offers integrated editing capabilities for third-party property sets (such as a Preflight Profile) without including third-party code in Switch. The property set must be stored in a file, and an external editing application must be supplied that behaves according to some specific guidelines.

Designing a flow with an external property editor

When a user invokes an external property editor while designing a flow, Switch launches the associated external application and passes the file path to a copy of the property set to be edited. The external application brings up a dialog for editing the contents of the property set and saves any changes back into the file. When the application exits, Switch recognizes the updated property set.

The value of a property edited with the "External editor" property editor is a file path. The actual file is stored in a temporary place allocated by Switch (similar to property sets created while importing a flow).

The "External editor" property editor works well in conjunction with the "Choose file" property editor, since both operate on a file path. Furthermore, Switch always makes a copy of the property set before editing. This means that:

- A property set that was selected with the "Choose file" property editor is never changed.

- Edited property sets are never shared between multiple flow elements.

Specifying an external property editor in SwitchScripter

The "External editor" property editor supports the following extra properties (shown in SwitchScripter and stored in the script declaration):

Property	Description
Application	The path (absolute or relative) to the application (executable) used to edit a property set. If you want to dynamically specify the path to the External editor, you have to leave the value of this property empty and implement the findExternalEditorPath entry point in your script. If a relative path is specified (that is, the path does not start with a drive letter or a forward slash), Switch looks for the external application relative to the system Applications folder. Typically this is the "Program Files" folder (on Windows) or the "Applications" folder (on Mac). If no path is specified, Switch executes the findExternalEditorPath entry point and the returned path is considered the absolute path to the required editor application.
Arguments for new	The argument string passed to the application when there is no pre-existing property set, after substituting all occurrences of %1 and %2 as described below.
Arguments for edit	The argument string passed to the application when there is a pre-existing property set to be edited, after substituting all occurrences of %1 and %2 as described below.
Editor name	The name your external editor user will see when he will use the editor.
Value overlay	The value of this property defines the string that will be shown after the editor is closed. The default value for this property is 'External properties defined'.

File format	There are two values possible:
	• If you want to use your own file format, choose **Custom**.
	• If you want to use a format that Switch can understand, choose **Switch**. In that case, the XML must have the following format:
	`<SwitchExternalEditor>`
	`  <PathsForExport>`
	`  <Path id="000001>c:/MyAppConfig/MyICCProfile.icc</Path>`
	`  <Path id="000002>c:/MyAppConfig/MyKeyFile</Path>`
	`  <Path id="000003>c:/MyAppConfig/MyOwnConfigurationFile.xml</Path>`
	`  <Path id="000004>c:/AnotherFile</Path>`
	`  </PathsForExport>`
	`</SwitchExternalEditor>`

Property	Description
	The XML includes a list of file paths. The files listed in these <Path> tags will be exported within flows. Before returning the paths list to a script, the list is sorted according to the lexical order of the corresponding **id** attribute values. This means the order of <Path> tags does not matter and may be changed by Switch after exporting the flow.

Note: If the file format is set to 'custom', Switch doesn't read the content of the property set and the files listed in the property set will not be exported within flows.

Substitutions in argument strings

The following substitutions are performed in the specified argument strings:

Placeholder	Replaced by
%1	The file path for the property set: • If it points to an existing file, the external application should load this file and replace it by the updated property set (the "edit" argument string is used) • If the path does not point to a file, the external application should place the newly created property set at this path (the "new" argument string is used)
%2	The locale currently in use by Switch specified as a four-letter string consisting of the two-letter ISO 639 language code (lowercase), followed by the two-letter ISO 3166 country code (uppercase); example: "enUS"

External application behavior

The external application must open the property set specified on its command line (or create a new default property set) and display a main window and/or a dialog box. After the user saves or cancels the changes, the application must exit with an appropriate exit code as follows:

Exit code	Meaning	Switch action
Zero	Changes to the property set were successfully saved	Use the newly created or updated property set
Nonzero	The user cancelled the operation or an error occurred	Discard any changes

Note: In case of the *Switch file format*, the external application must save a property set in the format described above (in the last row in the table under *Specifying an external property editor in SwitchScripter*).

Entry point example "findExternalEditorPath"

findExternalEditorPath(s : Switch, tag : String) : String

The function findExternalEditorPath starts when "External Editor" is selected in properties of a script element in the Switch Designer. The function returns a value which is used as a path for the external editor.

```
function findExternalEditorPath( s : Switch, tag : String )
{
 var path = "";
 if ( tag == "Prop1" )
  path = "/mypath/myapp1";
 else if ( tag == "Prop2" )
  path = "/mypath/myapp2";
 else return path;

 if ( s.isMac() == false ) path = "C:" + path + ".exe";

 return path;
}
```

5. Scripting reference

5.1. Scripting API - Introduction

The Switch scripting API (application programming interface) provides script elements and script expressions with access to information about the job (file or job folder) being processed, including job ticket and metadata contents. Script expressions are limited to read-only access, while script elements can update the information.

The scripting API supports two scripting languages: JavaScript and VBScript. The API documentation is provided in JavaScript syntax. Refer to *VBScript* on page 17 for information on how to adjust this syntax for scripting languages.

JavaScript language reference

The JavaScript language reference is provided as part of the scripting API documentation.

Scripting API Modules

The scripting API consists of a number of distinct modules, each of which publishes a set of related classes and functions. Some modules are published only to the JavaScript scripting language since VBScript offers its own built-in functionality in these areas.

Module	Description	Availability
Utility module	Reading and writing plain text files, enumerating files in directories and executing command line applications Text encoding, ByteArray class, File class, Dir class, Process class	Only JavaScript
XML module	Reading and writing XML files, including DOM and XPath access to their contents and performing XSLT transforms Node class, Document class, Element class, Text class, Comment class, Attr class, List classes	Only JavaScript
Network module	Communicating with external applications and web services through SOAP or HTTP SOAP class HTTP class	Only JavaScript
Database module	Accessing external databases with SQL queries via ODBC (Open Database Connectivity) DataSource class, Statement class	Only JavaScript

Module	Description	Availability
Flow element module	Accessing script flow element properties, including incoming and outgoing connections Moving jobs from and to these connections Accessing job ticket information for the current job, including the list of associated metadata datasets Entry points, Environment class, Switch class, Connection class, Job class, Occurrence class, List classes	All scripting languages
Metadata module	Retrieving the contents of metadata fields from a metadata dataset for the three supported data models (XML, JDF, XMP and Certified PDF 2) Map class, Dataset class, XML data model, JDF data model, XMP data model, CP2 data model, Opaque data model, FileStatistics class, Session class, Certificate class, User class, DataMap class, AltText class	All scripting languages

Script declaration

Reference information for the script declaration is presented as part of the scripting API documentation in Main script properties, Execution mode, Property definition properties, Property editors, Validating property values and External property editor.

5.2. JavaScript Reference

(The contents of this topic is adapted from documentation provided by Trolltech.)

This language reference describes the language features provided by Switch's JavaScript, which implements a subset of the ECMAScript 4.0 language. It is divided into the following topics:

- Language concepts
- Built-in types and objects
- Built-in functions
- Built-in operators
- Declarations
- Control statements

Readers are assumed to have a basic understanding of programming.

5.2.1. Language concepts

(The contents of this topic is adapted from documentation provided by Trolltech.)

Identifiers, variables and constants

JavaScript identifiers match the regexp pattern [_A-Za-z][_A-Za-z0-9]*. Identifiers are used for variables, constants, class names, function names and labels. JavaScript reserves some words which are valid identifiers for its own use. See declarations for the complete list.

Variables are declared using the var keyword:

```
var a;
// undefinedvar c = "foliage";
// the string "foliage"x = 1;
// global variable
```

If a variable is assigned without being declared, it is automatically declared as a global variable. Using global variables can make your code difficult to debug and maintain and is not recommended.

Constants are declared using the "const" keyword:

```
const x = "Willow";
const y = 42;
```

Constants must be defined at the point of declaration, because they cannot be changed later. If an attempt is made to assign to a constant, the JavaScript interpreter will issue an error message and stop.

Constants are public globals if they are declared outside of any enclosing braces. When declared within the scope of some braces, (example: within an "if" statement) their scope is local to the enclosing block.

Classes

JavaScript is a fully object oriented language. Classes can be defined using the class keyword as shown in the example below.

```
class Circle
{
  var m_x;
  var m_y;
  var m_r;

  function Circle( posx, posy, radius )
  {
    m_x = posx;
    m_y = posy;
    m_r = radius;
  }

  function setX( posx )
  {
    m_x = posy;
  }

  function setY( posy )
  {
    m_y = posy;
  }

  function setR( radius )
  {
    m_r = radius;
  }

  function x()
```

```
  {
    return m_x;
  }

  function y()
  {
    return m_y;
  }

  function r()
  {
    return m_r;
  }
}

class ColorCircle extends Circle
{
  var m_rgb;

  function ColorCircle( posx, posy, radius, rgbcolor)
  {
    Circle( posx, posy, radius );
    m_rgb = rgbcolor;
  }

  function setRgb( rgbcolor )
  {
    m_rgb = rgbcolor;
  }

  function rgb()
  {
    return m_rgb;
  }
}
```

A class's constructor is the function which has the same (case-sensitive) name as the class itself. The constructor should not contain an explicit return statement; it will return an object of its type automatically. JavaScript does not have a destructor function (a function that is called when the class is destroyed).

The class's member variables are declared with var, and its member functions with function.

The object instance itself is referred to using this operator. Inside a member function of a class, member variables and member functions can be accessed with an explicit "this" (example: this.x = posx;). This is not required, but can sometimes help to increase visibility.

JavaScript supports single inheritance and if a class inherits from another class, the superclass's constructor can be called with super().

Qualified names

When an object of a particular type is declared, the object itself becomes, in effect, a namespace. For example, in JavaScript there is a function called Math.sin(). If you wanted to have a sin() function in your own class that would not be a problem, because objects of your class would call the function using the object.function() syntax. The period is used to distinguish the namespace a particular identifier belongs to.

In most cases JavaScript is intelligent enough to work out the fully qualified name on its own. The only time that you need to qualify your names is when an unqualified name is ambiguous.

Class properties

A property is an undeclared variable that can be written to and accessed if the class supports properties.

```
var obj = new Objectobject.myProperty = 100;
```

The class Object does not define the variable myProperty but since the class supports properties, we can define the variable with that name on the fly and use it later. Properties are associated with the object they are assigned to, so even though the object (obj in the above example) gets the property myProperty, it does not mean that other objects of type Object will have the property "myProperty", unless explicitly stated.

Comments

JavaScript supports the same commenting syntax as C++. One-line comments may appear on a line of their own or after the statements on a line. Multi-line comments may appear anywhere.

```
// A one-line comment.

/*
A multi-line
comment.
*/
```

5.2.2. Built-in types and objects

(The contents of this topic is adapted from documentation provided by Trolltech.)

JavaScript provides a variety of built-in types (or classes) and objects.

- The built-in types include Array, Boolean, Date, Function, Number, Object, Point, Rect, RegExp, Size and String.
- The built-in objects are Math and JSON.

5.2.2.1. Array

An Array is a datatype which contains a named list of items. The items can be any JavaScript object. Multi-dimensional arrays are achieved by setting array items to be arrays themselves.

Arrays can be extended dynamically simply by creating items at non-existent index positions. Items can also be added using push(), unshift() and splice(). Arrays can be concatenated together using concat(). Items can be extracted using pop(), shift() and slice(). Items can be deleted using splice(). Arrays can be turned into strings using join() or toString(). Use reverse() to reverse the items in an array, and sort() to sort the items. The sort() function can be passed a comparison function for customized sort orders.

In general, operations that copy array items perform a deep copy on items that are Number or String objects and a shallow copy on other objects.

Array Construction

Arrays can be constructed from array literals or using the new operator:

```
var mammals = [ "human", "dolphin", "elephant", "monkey" ];
var plants = new Array( "flower", "tree", "shrub", "fungus" );
var things = [];
for ( i = 0; i < mammals.length; i++ ) {
    things[i] = new Array( 2 );
    things[i][0] = mammals[i];
    things[i][1] = plants[i];
```

```
}
```

Arrays can be initialized with a size, but with all items undefined:

```
var a = new Array( 10 ); // 10 items
```

Array Access

Array items are accessed via their names. Names can be either integers or strings.

Example:

```
var m2 = mammals[2];
mammals[2] = "gorilla";
var thing = things[2][1]
```

The first statement retrieves the value of the third item of the mammals array and assigns it to m2, which now contains "monkey". The second statement replaces the third item of the mammals array with the value "gorilla". The third statement retrieves the second item of the third item's array from the things array and assigns it to thing, which now contains "shrub".

As stated above, it is also possible to access the arrays using strings. These act as normal properties, and can be accessed either using the square bracked operator ([]) or by directly dereferencing the array object and specifying the property name (.name). These two accessor types can be mixed freely as seen below with the address and phoneNumber properties.

```
var names = [];
names["first"] = "John";
names["last"] = "Doe";
var firstName = names["first"];
var lastName = names["last"];
names["address"] = "Somewhere street 2";
names.phoneNumber = "+0123456789";
var address = names.address;
var phoneNumber = names["phoneNumber"];
```

Array Properties

length : Number

Holds the number of items in the array. Items with string keys are excluded from the length property.

Array Functions

concat(a1 : Array, a2 : Array, ... aN : Array) : Array

Concatenates the array with one or more other arrays in the order given and returns a single array.

```
    var x = new Array( "a", "b", "c" );
    var y = x.concat( [ "d", "e" ], [ 90, 100 ] );
    // y == [ "a", "b", "c", "d", "e", 90, 100 ]
```

join(optSeparator : String) : String

Joins all the items of an array together, separated by commas or by the specified optSeparator.

```
var x = new Array( "a", "b", "c" );
var y = x.join();          // y == "a,b,c"
var z = x.join( " * " );   // y == "a * b * c"
```

pop( ) : Object

Pops (that is, removes) the top-most (right-most) item off the array and returns it.

```
var x = new Array( "a", "b", "c" );
var y = x.pop(); // y == "c"  x == [ "a", "b" ]
```

push(item1 : Object, optItem2 : Object, ... optItemN : Object)

Pushes (that is, inserts) the given items onto the top (right) end of the array. The function returns the new length of the array.

```
var x = new Array( "a", "b", "c" );
x.push( 121 ); // x == [ "a", "b", "c", 121 ]
```

reverse( )

Reverses the items in the array.

```
var x = new Array( "a", "b", "c", "d" );
x.reverse(); // x == [ "d", "c", "b", "a" ]
```

shift( ) : Object

Shifts (that is, removes) the bottom-most (left-most) item off the array and returns it.

```
var x = new Array( "a", "b", "c" );
var y = x.shift(); // y == "a"  x == [ "b", "c" ]
```

slice(startIndex : Number, optEndIndex : Number) : Array

Copies a slice of the array from the item with the given starting index, startIndex, to the item before the item with the given ending index, optEndIndex. If no ending index is given, all items from the starting index onward are sliced.

```
var x = new Array( "a", "b", "c", "d" );
var y = x.slice( 1, 3 ); // y == [ "b", "c" ]
var z = x.slice( 2 );    // z == [ "c", "d" ]
```

sort(optComparisonFunction : function or Function)

Sorts the items in the array using string comparison. For customized sorting, pass the sort() function a comparison function, optComparisonFunction, that has the following signature and behavior:

```
function comparisonFunction( a, b ) // signature
```

The function must return an integer as follows:

- -1 if a < b

- 0 if a == b
- 1 if a > b

Example 1:

```
var x = new Array( "d", "x", "a", "c" );
x.sort(); // x == [ "a", "c", "d", "x" ]
```

Example 2:

```
function numerically( a, b ) { return a < b ? -1 : a > b ? 1 : 0; }
var x = new Array( 8, 90, 1, 4, 843, 221 );
 x.sort( numerically ); // x == [ 1, 4, 8, 90, 221, 843 ]
```

splice(startIndex : Number, replacementCount : Number, optItem1 : Object, ... optItemN : Object)

Splices items into the array and out of the array. The first argument, startIndex, is the start index. The second argument, replacementCount, is the number of items that are to be replaced. Make the second argument 0 if you are simply inserting items.

The remaining arguments are the items to be inserted. If you are simply deleting items, the second argument must be > 0 (that is, the number of items to delete) and there must be no new items given.

```
var x = new Array( "a", "b", "c", "d" );

// 2nd argument 0, plus new items   ==> insertion
x.splice( 1, 0, "X", "Y" );
// x == [ "a", "X", "Y", "b", "c", "d" ]

// 2nd argument > 0, and no items   ==> deletion
x.splice( 2, 1 );
// x == [ "a", "X", "b", "c", "d" ]

// 2nd argument > 0, plus new items ==> replacement
x.splice( 3, 2, "Z" );
// x == [ "a", "X", "b", "Z" ]
```

toString( ) : String

Joins all the items of an array together, separated by commas. This function is used when the array is used in the context of a string concatenation or is used as a text value, example: for printing. Use join() if you want to use your own separator.

```
var x = new Array( "a", "b", "c" );
var y = x.toString();  // y == "a,b,c"
var z = x.join();      // y == "a,b,c"
```

unshift(item1 : Object, optItem2 : Object, ... optItemN : Object)

Unshifts (that is, inserts) the given items at the bottom (left) end of the array.

```
var x = new Array( "a", "b", "c" );
```

```
x.unshift( 121 ); // x == [ 121, "a", "b", "c" ]
```

5.2.2.2. Boolean

JavaScript provides a Boolean data type. In general, creating objects of this type is not recommended since the behavior will probably not be what you would expect.

Instead, use the boolean constants true and false as required. Any expression can be evaluated in a boolean context, example: in an if statement. If the expression's value is 0, null, false, NaN, undefined (see built-in constants) or the empty string "", the expression is false; otherwise the expression is true.

5.2.2.3. Date

Instances of the Date class are used to store and manipulate dates and times.

A variety of get functions are provided to obtain the date, time or relevant parts, for example, getDay(), getYear(), getHours(), getMilliseconds(), getMinutes(), getMonth(), getSeconds(), getTime().

A complementary set of functions are also provided, including setYear(), setHours(), setMilliseconds(), setMinutes(), setMonth(), setSeconds(), setTime().

The functions operate using local time.

Conversion between Date objects to and from strings are provided by parse() and Date::toString().

Elapsed time (in milliseconds) can be obtained by creating two dates, casting them to Number and subtracting one value from the other.

```
var date1 = new Date();
// time flies..
var date2 = new Date();
var timedifference = date2.getTime() - date1.getTime();
```

Static Date Function

parse(dateString : String) : Number

This is a static function that parses a string, dateString, which represents a particular date and time. It returns the number of milliseconds since midnight on the 1st January 1970. The string must be in the ISO 8601 extended format: YYYY-MM-DD or with time YYYY-MM-DDTHH:MM:SS.

```
var d = new Date( Date.parse( "1976-01-25T22:30:00" ) );
d = Date.parse( "1976-01-25T22:30:00" );
```

Date Construction

Date( )

Date(milliseconds : Number)

Date(year : Number, month : Number, day : Number, optHour : Number, optMinutes : Number, optSeconds : Number, optMilliseconds : Number)

Dates can be constructed with no arguments, in which case the value is the date and time at the moment of construction using local time. A single integer argument is taken as the number of milliseconds since midnight on the 1st January 1970.

```
var today = new Date();
var d = new Date( 1234567 );
var date = new Date( 1994, 4, 21 );
var moment = new Date( 1968, 5, 11, 23, 55, 30 );
```

Date Functions

getDate( ) : Number

Returns the day of the month using local time. The value is always in the range 1..31.

```
var d = new Date( 1975, 12, 25 );
var x = d.getDate();  // x == 25
```

getDay( ) : Number

Returns the day of the week using local time. The value is always in the range 1..7, with the week considered to begin on Monday.

Example 1:

```
var d = new Date( 1975, 12, 25, 22, 30, 15 );
var x = d.getDay();  // x == 4
```

Example 2:

```
var IndexToDay = [ "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun" ];
var d = new Date( 1975, 12, 28 );
 var day = IndexToDay[ d.getDay() - 1 ]; // day == "Sun"
```

getYear( ) : Number

Returns the year using local time.

```
var d = new Date( 1975, 12, 25 );
var x = d.getYear();  // x == 1975
```

getHours( ) : Number

Returns the hours using local time. The value is always in the range 0..23

```
var d = new Date( 1975, 12, 25, 22 );
var x = d.getHours();  // x == 22
```

getMilliseconds( ) : Number

Returns the milliseconds component of the date using local time. The value is always in the range 0..999. In the example, x is 0, because no milliseconds were specified, and the default for unspecified components of the time is 0.

```
var d = new Date( 1975, 12, 25, 22 );
var x = d.getMilliseconds();  // x == 0
```

getMinutes( ) : Number

Returns the minutes component of the date using local time. The value is always in the range 0..59.

```
var d = new Date( 1975, 12, 25, 22, 30 );
var x = d.getMinutes();  // x == 30
```

getMonth( ) : Number

Returns the month component of the date using local time. The value is always in the range 0..12.

Example 1:

```
var d = new Date( 1975, 12, 25, 22, 30 );
var x = d.getMonth();  // x == 12
```

Example 2:

```
var IndexToMonth = [ "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
"Sep", "Oct", "Nov", "Dec" ];
var d = new Date( 1975, 12, 25 );
var month = IndexToMonth[ d.getMonth() - 1]; // month == "Dec"
```

getSeconds( ) : Number

Returns the seconds component of the date using local time. The value is always in the range 0..59. In the example x is 0 because no seconds were specified, and the default for unspecified components of the time is 0.

```
var d = new Date( 1975, 12, 25, 22, 30 );
var x = d.getSeconds();  // x == 0
```

getTime( ) : Number

Returns the number of milliseconds since midnight on the 1st January 1970 using local time.

```
var d = new Date( 1975, 12, 25, 22, 30 );
var x = d.getTime();  // x == 1.91457e+11
```

setDate(dayOfTheMonth : Number)

Sets the day of the month to the specified dayOfTheMonth in local time.

```
var d = new Date( 1975, 12, 25, 22, 30 );
d.setDate( 30 );  // d == 1975-12-30T22:30:00
```

setYear(year : Number)

Sets the year to the specified year in local time. If the month, optMonth is specified, it must be in the range 0..11. If the optDayOfTheMonth is specified it must be in the range 1..31.

```
var d = new Date( 1975, 12, 25, 22, 30 );
 d.setYear( 1980 );          // d == 1980-12-30T22:30:00
 d.setYear( 1980, 11, 2 );  // d == 1980-12-02T22:30:00
```

setHours(hour : Number)

Sets the hour to the specified hour, which must be in the range 0..23, in local time. The minutes, seconds and milliseconds past the hour (optMinutes, optSeconds and optMilliseconds) can also be specified.

```
var d = new Date( 1975, 12, 25, 22, 30 );
 d.setHours( 10 );  // d == 1980-12-30T10:30:00
```

setMilliseconds(milliseconds : Number)

Sets the milliseconds component of the date to the specified value in local time.

```
var d = new Date( 1975, 12, 25, 22, 30 );
d.setMilliseconds( 998 );  // d == 1980-12-30T10:30:00:998
```

setMinutes(minutes : Number)

Sets the minutes to the specified minutes, which must be in the range 0..59, in local time. The seconds and milliseconds past the minute (optSeconds and optMilliseconds) can also be specified.

```
var d = new Date( 1975, 12, 25, 22, 30 );
d.setMinutes( 15 );  // d == 1980-12-30T10:15:00
```

setMonth(month : Number)

Sets the month to the specified month, which must be in the range 0..11, in local time. The day of the month (optDayOfTheMonth) may also be specified.

```
var d = new Date( 1975, 12, 25, 22, 30 );
d.setMonth( 0, 11 );  // d == 1980-01-11T22:30:00
```

setSeconds(seconds : Number, optMilliseconds : Number)

Sets the seconds to the specified seconds, which must be in the range 0..59, in local time. The milliseconds past the seconds (optMilliseconds) can also be specified.

```
var d = new Date( 1975, 12, 25, 22, 30 );
d.setSeconds( 25 );  // d == 1980-12-30T22:30:25
```

setTime(milliseconds : Number)

Sets the date and time to the local date and time given in terms of milliseconds since midnight on the 1st January 1970.

```
var d = new Date( 1975, 12, 25, 22, 30 );
```

```
var duplicate = new Date();
duplicate.setTime( d.getTime() );
```

toString( ) : String

Converts the date into a string on the ISO 8601 extended format: YYYY-MM-DDTHH:MM:SS.

```
var d = new Date( 1975, 12, 25, 22, 30 );
var s = d.toString(); // s == "1975-12-25T22:30:00"
```

5.2.2.4. Function

In some situations it is desirable to create functions at run time.

```
var min = new Function( "x", "y", "return x < y ? x : y;" );
var m = min( 68, 9 ); // m == 9
```

The first arguments are the names of the parameters; the last argument is a string which contains the function's definition. It is also possible to supply the list of argument names as one comma separated string.

```
var min = new Function( "x,y", "return x < y ? x : y;" );
```

Variable numbers of arguments are also supported using the arguments array (see built-in variables). For example:

```
var max = new Function(
     "var max = 0;"
   + "for ( var i = 0; i < arguments.length; i++ ) {"
   + "   if ( arguments[ i ] > max ) max = arguments[ i ];"
   + "}"
   + "return max;"
   );
var x = max( 50, 16, 24, 99, 1, 97 ); // x == 99
```

5.2.2.5. Number

A Number is a datatype that represents a number. In most situations, programmers will use numeric literals like 3.142 directly in code. The Number datatype is useful for obtaining system limits, example: MIN_VALUE and MAX_VALUE, and for performing number to string conversions with toString()

A numeric variable can hold a non-numeric value, in which case isNaN() returns true. The result of an arithmetic expression may exceed the maximum or minimum representable values in which case the value of the expression will be Infinity, and isFinite() will return false. See also built-in contants and built-in functions.

Number Construction

Numbers are not normally constructed, but instead created by simple assignment, example:

```
var x = 3.142;
```

Number Properties

MAX_VALUE : Number
The maximum value for floating point values.

MIN_VALUE : Number
The minimum value for floating point values.

Number Functions

toString( ) : String
Returns the number as a string value.

5.2.2.6. Object

Object is the base type for all JavaScript objects.

Object provides a toString() function which is usually overridden by subclasses.

5.2.2.7. Point

A Point object represents a two-dimensional point.

Point Construction

The Point class provides three different constructors.

```
var point = new Point( 20, 30 );
var duplicate = new Point( point ); // 20, 30
var zero = new Point();             // 0, 0
```

Point Properties

x : Number
The x position of the point.

y : Number
The y position of the point.

5.2.2.8. Rect

A Rect object represents a rectangle.

Rect Construction

The rectangle class provides three constructors.

```
var rect = new Rect( 10, 20, 30, 40 ); // x=10, y=20, width=30, height=40
var duplicate = new Rect( rect );      // x=10, y=20, width=30, height=40
```

```
var empty = new Rect();                // x=0, y=0, width=0, height=0
```

Rect Properties

x : Number

The left position of the rectangle.

y : Number

The right position of the rectangle.

width : Number

The width of the rectangle.

height : Number

The height of the rectangle.

top : Number

The position of the top of the rectangle. This is defined as top = y.

bottom : Number

The position of the bottom of the rectangle. This is defined as bottom = y + height + 1.

left : Number

The position of the rectangle's left side. This is defined as left = x

right : Number

The position of the rectangle's right side. This is defined as right = x + width + 1.

center : Point

The center of the rectangle.

Rect Functions

isNull() : Boolean

Returns true if the rectangle has a width and height of 0.

```
var empty = new Rect();
empty.isNull(); // true;
var square = new Rect( 10, 10, 10, 10 );
square.isNull(); // false;
```

isEmpty() : Boolean

Returns true if the rectangle is empty, meaning that it has width and/or height that is negative.

```
var rect = new Rect( 10, 10, -10, -10 );
rect.isEmpty(); // true
var rect = new Rect( 10, 10, 10, 10 );
rect.isEmpty(); // false
```

contains(point: Point) : Boolean

contains(x : Number, y : Number) : Boolean;

The check of point is inside the rectangle. This can be used as workaround because it is enough to check if lower left and upper right points of the second rectangle are both inside the first rectangle:

```
var rect= new Rect( 0, 0, 100, 100 );
var result = rect.contains( 10, 10 ) && rect.contains( 20, 20 );
if( result )
{
    s.log( 1, "Rect ( 0, 0, 100, 100 ) contains rect ( 10, 10, 20, 20 )" );
}
```

intersection(otherRect : Rect) : Rect

Returns the intersection between this rectangle and another rectangle. The intersection of two rectangles is the part of the rectangles that overlap. If they do not overlap, an empty rectangle is returned.

union(otherRect : Rect) : Rect

Returns the union of two rectangles. The union is a rectangle large enough to encompass both rectangles.

intersects(otherRect : Rect) : Boolean

Returns true if this rectangle intersects the other rectangle.

normalize( )

Normalizes this rectangle. This means changing the prefix of the width and/or height if they are negative. After being normalized, a rectangle will no longer be empty

moveLeft(pos : Number)

Moves the rectangle so that its left property is equal to pos.

moveRight(pos : Number)

Moves the rectangle so that its right property is equal to pos.

moveTop(pos : Number)

Moves the rectangle so that its top property is equal to pos.

moveBottom(pos : Number)

Moves the rectangle so that its bottom property is equal to pos.

moveBy(dx : Number, dy : Number)

Translates the rectangle by dx and dy. dx and dy will be added to x and y. Width and height will be left unchanged.

5.2.2.9. RegExp

A RegExp is a regular expression matcher for strings.

Regular expressions can be created either by using the /expression/ syntax or by using the RegExp constructor as shown below. Note that when using the RegExp constructor, a string is passed, and all backslashes must be escaped using an extra backslash, that is, \\d. Below are two ways to create regular expressions that matches the pattern "QUANTITY=471 # Order quantity" and gets the number 471.

```
var directRegex = /([A-Z]+)=(\d+)/;
var str = "QUANTITY=471 # Order quantity";
str.match(directRegex);
directRegex.capturedTexts[2]; // "471";
var indirectRegex = new RegExp( "([A-Z]+)=(\\d+)" );
```

RegExp Properties

valid : Boolean

True if the regular expression is syntactically valid; otherwise returns false.

empty : Boolean

True if the pattern is empty; otherwise returns false.

matchedLength : Number

The length of the last matched string or -1 if there was no match.

capturedTexts : String[ ]

An array of all the captured texts from the previous match. This can be empty.

global : Boolean

Specifies that the regexp should be matched globally. A global regexp will match every occurrence (that is, as many times as possible), whereas a non-global regexp will match at most once (at the first match it encounters). This is particularly relevant for replace where every occurance of a pattern will be replaced when global is true.

A regular expression can be set to global either by setting the global property on a regexp object or by specifying a trailing g in the pattern.

```
var re = /mypattern/g;     // Global by method #1
var re = /mypattern/;
re.global = true           // Global by method #2
```

ignoreCase : Boolean

Specifies that the regexp ignores case when matching. Case-insensitivity can is enabled by either specifying a trailing i after the pattern or by setting the ignoreCase property.

```
var re = /mypattern/i;     // Case-insensitive by method #1
var re = /mypattern/;
re.ignoreCase = true;      // Case-insensitive by method #2
```

RegExp Functions

toString() : String

Returns the regular expression pattern as a string.

search(text : String) : Number

Searches text for the pattern defined by the regular expression. The function returns the position in the text of the first match or -1 if no match is found.

```
var re = /\d+ cm/; // matches one or more digits followed by space then 'cm'
re.search( "A meter is 100 cm long" ); // returns 11
```

searchRev(text : String) : Number

Same as search(), but searchRev searches from the end of the text.

exactMatch(text : String) : Boolean

Returns true if text exactly matches the pattern in this regular expression; otherwise returns false.

cap(nth : Number) : String

Returns the nth capture of the pattern in the previously matched text. The first captured string (cap(0)) is the part of the string that matches the pattern itself, if there is a match. The following captured strings are the parts of the pattern enclosed by parenthesis. In the above example, we try to capture ([a-zA-Z]+), which captures a sequence of one or more letters and spaces after the name: part of the string.

```
re = /name: ([a-zA-Z ]+)/;
re.search( "name: John Doe, age: 42" );
re.cap(0);  // returns "name: John Doe"
re.cap(1);  // returns "John Doe"
re.cap(2);  // returns undefined, no more captures.
```

pos(nth : Number) : Number

Returns the position of the nth captured text in the search string.

```
re = /name: ([a-zA-Z ]+)/;
re.search( "name: John Doe, age: 42" );
re.pos(0); // returns 0, position of "name: John Doe"
re.pos(1); // returns 6, position of "John Doe"
re.pos(2); // returns -1, no more captures
```

5.2.2.10. Size

A Size object represents a two-dimensional size, using width and height.

Size Construction

The Size class provides three constructors:

```
var size = new Size( 100, 200 ); // width = 100, height = 200
var duplicate = new Size( size ); // width = 100, height = 200
var empty = new Size(); // width = 0, height = 0
```

Size Properties

width : Number

The width of the size.

height : Number

The height of the size.

Size Functions

translate( )

Swaps the width and height of the size.

5.2.2.11. String

A String is a sequence of zero or more Unicode characters.

String Construction

Strings can be created and concatenated as follows.

```
var text = "this is a";
var another = new String( "text" );
var concat = text + " " + another;  // concat == "this is a text"
```

String Properties

length : Number

The length property specifies the length of the string.

Static String Functions

fromCharCode (code1 : Number, code2 : Number, ...)

Returns a string made up of the characters with code code1, code2, etc, according to their Unicode character codes.

```
var s = String.fromCharCode( 65, 66, 67, 68 );
// s == "ABCD"
```

String Functions

charAt(pos : Number) : String

Returns the character in the string at position pos. If the position is out of bounds, undefined is returned.

charCodeAt(pos : Number) : Number

Returns the character code of the character at position pos in the string. If the position is out of bounds, undefined is returned.

indexOf(pattern : String or RegExp, pos : Number) : Number

Returns the index of pattern in the string, starting at position pos. If no position is specified, the function starts at the beginning of the string. If the pattern is not found in the string, -1 is returned.

lastIndexOf(pattern : String or RegExp, pos : Number) : Number

Returns the last index of pattern in the string, starting at position pos and searching backwards from there. If no position is specified, the function starts at the end of the string. If the pattern is not found in the string, -1 is returned.

match(pattern : RegExp) : String or String[ ]

Returns the matched pattern if this string matches the pattern defined by regexp. If the string doesn't match or regexp is not a valid regular expression, undefined is returned. If the regexp has global set and the string matches multiple patterns, an array of strings is returned.

search(pattern : String or RegExp, pos : Number) : Number

Same as find.

searchRev(pattern : String or RegExp, pos : Number) : Number

Same as findRev

replace(pattern : RegExp, newValue : String) : String

Replaces the first occurrence of pattern in the string with newvalue if the pattern is found in the string. A modified copy of string is returned. The original string is not modified.

If pattern is a regular expression with global set, all occurances of pattern in the string will be replaced.

Example:

```
// Sample: remove unwanted whitespaces from a string using backreferences
var functionID = "test( int a ,const int b,int   c , int d, const QString )  ";
functionID = functionID.replace( /(\s)\s*/g, "\\1" );
functionID = functionID.replace( /(\W)\s/g, "\\1" );
functionID = functionID.replace( /\s(\W)/g, "\\1" );
s.log( 1, functionID ); // test(int a,const int b,int c,int d,const QString)
```

split(pattern : String or RegExp) : String[ ]

Returns an array of strings containing this string split on each occurrence of pattern.

substring(startIndex : Number, endIndex : Number) : String

Returns a copy of this string which is the substring starting at startIndex and ending at endIndex.

toLowerCase() : String

Returns a lowercase copy of this string.

lower() : String

Same as toLowerCase().

toUpperCase() : String

Returns an uppercase copy of this string.

upper() : String

Same as toUpperCase().

isEmpty() : Boolean

Returns true if the string is empty, that is, has a length of 0; otherwise returns false.

left(length : Number) : String

Returns a substring containing the length leftmost characters of this string.

right(length : Number) : String

Returns a substring containing the length rightmost characters of this string.

mid(start : Number, length : Number) : String

Returns a copy of this string which is the substring starting at "start" and is "length" characters long. If the optional length parameter is not specified, the new string will be from start until the end of the string.

find(pattern : String or RegExp, pos : Number) : Number

Returns the first position of pattern after pos. If the pattern is not found, -1 is returned. If pos is not specified, position 0 is used.

findRev(pattern : String or RegExp, pos : Number) : Number

Returns the first position of pattern before or at pos, searching backward. If pattern is not found, -1 is returned. If pos is not specified, the search starts at the end of the string.

startsWith(pattern : String) : Boolean

Returns true if the string starts with pattern; otherwise returns false.

endsWith(pattern : String) : Boolean

Returns true if the string ends with pattern; otherwise returns false.

arg(value : String or Number, fieldWidth : Number) : String

This function will return a string that replaces the lowest numbered occurrence of %1, %2, ..., %9 with value.

The fieldWidth parameter specifies the minimum amount of space that value is padded to. A positive fieldWidth will produce right-aligned text, whereas a negative fieldWidth will produce left-aligned text.

argInt(value : Number, fieldWidth : Number, base : Number) : String

This function behaves like arg above, but is specialized for the case where value is an integer.

value is expressed in base base, which is 10 by default and must be between 2 and 36.

argDec(value : Number, fieldWidth : Number, format : String, precision : Number): String

This function behaves like arg() above, but is specialized for the case where value is a decimal value.

Argument value is formatted according to the format specified, which is 'g' by default and can be any of the following:

- e - format as [-]9.9e[+|-]999
- E - format as [-]9.9E[+|-]999
- f - format as [-]9.9
- g - use e or f format, whichever is the most concise
- G - use E or f format, whichever is the most concise

With 'e', 'E', and 'f', precision is the number of digits after the decimal point. With 'g' and 'G', precision is the maximum number of significant digits (trailing zeroes are omitted).

5.2.2.12. Math

The Math object, which always exists in a JavaScript program, supports many common mathematical functions.

```
with ( Math ) {
    var x = PI * 2;
    var angle = 1.3;
    var y = x * sin( angle );
}
```

See also built-in contants and built-in functions.

Math Properties

All the Math properties are read-only constants.

E : Number

Eulers constant. The base for natural logarithms.

LN2 : Number

Natural logarithm of 2.

LN10 : Number

Natural logarithm of 10.

LOG2E : Number

Base 2 logarithm of E.

LOG10E : Number

Base 10 logarithm of E.

PI : Number

Pi.

SQRT1_2 : Number

Square root of 1/2.

SQRT2 : Number

Square root of 2.

Math Functions

abs(number : Number) : Number

Returns the absolute value of the given number. The equivalent of: x = x < 0 ? -x: x;

```
var x = -99;
var y = 99;
with ( Math ) {
    x = abs( x );
    y = abs( y );
}
// x == y
```

acos(number : Number) : Number

Returns the arccosine of the given number in radians between 0 and Math.PI. If the number is out of range, returns NaN.

asin(number : Number) : Number

Returns the arcsine of the given number in radians between -Math.PI/2 and Math.PI/2. If the number is out of range, returns NaN.

atan(number : Number) : Number

Returns the arctangent of the given number in radians between -Math.PI/2 and Math.PI/2. If the number is out of range, returns NaN.

atan2(yCoord : Number, xCoord : Number) : Number

Returns the counter-clockwise angle in radians between the positive x-axis and the point at (xCoord, yCoord). The value returned is always between -Math.PI and Math.PI.

```
function polar( x, y )
{
    return Math.atan2( y, x );
}
```

ceil(number : Number) : Number

If the number is an integer, it returns the number. If the number is a floating point value, it returns the smallest integer greater than the number.

```
var x = 913.41;
   x = Math.ceil( x ); // x == 914
   var y = -33.97;
   y = Math.ceil( y ); // y == -33
```

cos(number : Number) : Number

Returns the cosine of the given number in radians. The value will be in the range -1..1.

exp(number : Number) : Number

Returns Math.E raised to the power of the given number.

floor(number : Number) : Number

If the number is an integer, it returns the number. If the number is a floating point value, it returns the greatest integer less than the number.

log(number : Number) : Number

If the number is > 0, it returns the natural logarithm of the given number. If the number is 0, it returns Infinity. If the number is < 0, it returns NaN.

max(number1 : Number, number2 : Number) : Number

Returns the largest of number1 and number2.

min(number1 : Number, number2 : Number) : Number

Returns the smallest of number1 and number2.

pow(number : Number, power : Number) : Number

Returns the value of the number raised to the power.

random() : Number

Returns a pseudo-random floating point number between 0 and 1. Pseudo random numbers are not truly random but may be adequate for some applications such as simple simulations.

round(number : Number) : Number

Returns the number rounded to the nearest integer. If the fractional part of the number is >= 0.5, the number is rounded up; otherwise it is rounded down.

sin(number : Number) : Number

Returns the sine of the given number in radians. The value will be in the range -1..1.

sqrt(number : Number) : Number

If the number is >= 0, it returns the square root. If the number is < 0, it returns NaN.

tan(number : Number) : Number

Returns the tangent of the given number in radians.

5.2.2.13. JSON

JSON (JavaScript Object Notation) is a data format derived from JavaScript. JSON file names have the extension .json.

Switch supports JSON starting from Switch 2017 update 2.

JSON Functions

parse(text : String) : Object

Parses JSON as a UTF-8 encoded JSON document, and creates an object from it. If unable to parse, an exception is thrown.

stringify(value : Object, space : Boolean) : String

Converts the JavaScript Object specified in the parameter value into a string. If unable to convert, an exception is thrown.

The parameter "space" is optional. The space value defines the format of the JSON byte array produced when converting the object to a string.

If *space = true*, human readable output will be generated:

```
{
    "Array": [
        true,
        999,
        "string"
    ],
    "Key": "Value",
    "null": null
}
```

If *space = false* (default value), a compact string will be generated:

```
{"Array":[true,999,"string"],"Key":"Value","null":null}
```

5.2.3. Built-in functions

JavaScript provides a number of convenient built-in constants, variables and functions.

- The built-in constants include Infinity, NaN and undefined.
- The built-in variables include arguments.
- The built-in functions include eval(), isFinite(), isNaN(), parseFloat(), parseInt().

Built-in constants

Infinity

This is the value of any division by zero, that is,

```
var i = 1/0;
```

In JavaScript, division by zero does not raise an exception; instead it assigns the Infinity value as the result of the expression. Use isFinite() to test whether a value is finite or not.

NaN

Some functions that are supposed to return a numeric result may return NaN instead. Use isNaN() to check for this.

undefined

This is the value of a variable that does not have a defined value, that is, has not been assigned to.

```
var i;
// ...
if ( i == undefined ) {
    i = 77;
}
```

In this example, if execution reaches the if statement and i has not been assigned a value, it will be assigned the value 77.

Built-in variables

arguments : Array

This is an array of the arguments that were passed to the function. It only exists within the context of a function.

```
function sum()
   {
       total = 0;
       for ( i = 0; i < arguments.length; i++ ) {
           total += arguments[ i ];
       }
       return total;
   }
```

Built-in functions

eval(string : String)

This function parses and executes the contents of the string, taking the text to be valid JavaScript.

```
var x = 57;
   var y = eval( "40 + x" ); // y == 97
```

isFinite(expression) : Boolean

Returns true if the expression's value is a number that is within range; otherwise returns false.

isNaN(expression) : Boolean

Returns true if the expression's value is not a number; otherwise returns false.

```
       var x = parseFloat( "3.142" );
       var y = parseFloat( "haystack" );
       var xx = isNaN( x ); // xx == false
```

```
        var yy = isNaN( y ); // yy == true
```

parseFloat(string : String) : Number

Parses the string and returns the floating point number that the string represents or NaN if the parse fails. Leading and trailing whitespace are ignored. If the string contains a number followed by non-numeric characters, the value of the number is returned and the trailing characters ignored.

parseInt(string : String, optBase : Number) : Number

Parses the string and returns the integer that the string represents in the given base optBase or NaN if the parse fails. If the base is not specified, the base is determined as follows:

- base 16 (hexadecimal) if the first non-whitespace characters are "0x" or "0X";
- base 8 (octal) if the first non-whitespace character is "0";
- base 10 otherwise.

Leading and trailing whitespace are ignored. If the string contains a number followed by non-numeric characters, the value of the number is returned and the trailing characters ignored.

```
    var i = parseInt( "24" );       // i == 24
    var h = parseInt( "0xFF" );     // h == 255
    var x = parseInt( " 459xyz " ); // x == 459
```

5.2.4. Built-in operators

Javascript offers built-in operators in the following categories:

- Assignment operators
- Arithmetic operators
- String operators
- Logical operators
- Comparison operators
- Bit-wise operators
- Special operators

Assignment operators

These operators are used to assign the value of expressions to variables.

= operator

```
var variable = expression;
```

The assignment operator is used to assign the value of an expression to the variable.

It is an error to attempt to assign to a constant.

+= operator

```
variable += expression;
```

This operator adds the value of the expression to the variable. It is the same as:

```
 variable = variable + expression;
```

but is shorter to write and less error-prone.

See also += string operator.

-= operator

```
variable -= expression;
```

This operator subtracts the value of the expression from the variable.

***= operator**

```
variable *= expression;
```

This operator multiplies the value of the expression by the value of the variable.

/= operator

```
variable /= expression;
```

This operator divides the value of the variable by the value of the expression.

%= operator

```
variable %= expression;
```

This operator divides the variable by the expression and assigns the remainder of the division (which may be 0), to the variable.

&= operator

```
variable &= expression;
```

This operator performs a bit-wise AND on the value of the expression and the value of the variable and assigns the result to the variable.

^= operator

```
variable ^= expression;
```

This operator performs a bit-wise OR on the value of the expression and the value of the variable and assigns the result to the variable.

|= operator

```
variable |= expression;
```

This operator performs a bit-wise OR on the value of the expression and the value of the variable and assigns the result to the variable.

<<= operator

```
variable <<= expression;
```

This operator performs a bit-wise left shift on the variable by an expression number of bits. Zeros are shifted in from the right.

>>= operator

```
variable >>= expression;
```

This operator performs a bit-wise (sign-preserving) right shift on the variable by an expression number of bits.

>>>= operator

```
variable >>>= expression;
```

This operator performs a bit-wise (zero-padding) right shift on the variable by an expression number of bits.

Arithmetic operators

These operators are used to perform arithmetic computations on their operands.

+ operator

```
operand1 + operand2
```

This operator returns the result of adding the two operands (operand1 and operand2).

See also + string operator.

++ operator

```
++operand; // pre-increment
operand++; // post-increment
```

The pre-increment version of this operator increments the operand and returns the value of the (now incremented) operand.

The post-incremented version of this operator returns the value of the operand and then increments the operand.

- operator

```
var result = operand1 - operand2; // subtraction
operand = -operand;               // unary negation
```

The subtraction version of this operator returns the result of subtracting its second operand (operand2) from its first operand (operand1).

The unary negation version of this operator returns the result of negating (changing the sign) of its operand.

-- operator

```
--operand; // pre-decrement
operand--; // post-decrement
```

The pre-decrement version of this operator decrements the operand and returns the value of the (now decremented) operand.

The post-decremented version of this operator returns the value of the operand and then decrements the operand.

*** operator**

```
operand1 * operand2
```

This operator returns the result of multiplying the two operands (operand1 and operand2).

/ operator

```
operand1 / operand2
```

This operator returns the result of dividing the first operand (operand1) by the second operand (operand2).

Note that division by zero is not an error. The result of division by zero is Infinity.

% operator

```
operand1 % operand2
```

This operator returns the integer remainder (which may be 0) from the division of operand1 by operand2.

String operators

These operators provide string functions using operators. Many other string functions are available, see String.

+ string operator

```
str1 + str2
```

This operator returns a string that is the concatenation of its operands, (str1 and str2).

See also + arithmetic operator.

+= string operator

```
str1 += str2
```

This operator appends its second operand (str2) onto the end of the first operand (str1).

See also += assignment operator.

Logical operators

These operators are used to evaluate whether their operands are true or false. The binary operators use short-circuit logic, that is, they do not evaluate their second operand if the logical value of the expression can be determined by evaluating the first operand alone.

&& operator

```
operand1 && operand2
```

This operator returns an object whose value is true if both its operands are true; otherwise it returns an object whose value is false.

Specifically, if the value of operand1 is false, the operator returns operand1 as its result. If operand1 is true, the operator returns operand2.

|| operator

```
operand1 || operand2
```

This operator returns an object whose value is true if either of its operands are true; otherwise it returns an object whose value is false.

Specifically, if the value of operand1 is true, the operator returns operand1 as its result. If operand1 is false, the operator returns operand2.

! operator

```
! operand
```

If the operand's value is true, this operator returns false; otherwise it returns true.

Comparison operators

These operators are used to compare objects and their values.

== operator

```
operand1 == operand2
```

Returns true if the operands are equal; otherwise returns false.

!= operator

```
 operand1 != operand2
```

Returns true if the operands are not equal; otherwise returns false.

=== operator

```
operand1 === operand2
```

Returns true if the operands are equal and of the same type; otherwise returns false.

!== operator

```
operand1 !== operand2
```

Returns true if the operands are not equal or if the operands are of different types;otherwise returns false.

> operator

```
operand1 > operand2
```

Returns true if operand1 is greater than operand2; otherwise returns false.

>= operator

```
operand1 >= operand2
```

Returns true if operand1 is greater than or equal to operand2; otherwise returns false.

< operator

```
operand1 < operand2
```

Returns true if operand1 is less than operand2; otherwise returns false.

<= operator

```
operand1 <= operand2
```

Returns true if operand1 is less than or equal to operand2; otherwise returns false.

Bit-wise operators

& operator

```
operand1 & operand2
```

Returns the result of a bit-wise AND on the operands (operand1 and operand2).

^ operator

```
operand1 ^ operand2
```

Returns the result of a bit-wise XOR on the operands (operand1 and operand2).

| operator

```
operand1 | operand2
```

Returns the result of a bit-wise OR on the operands (operand1 and operand2).

~ operator

```
~ operand
```

Returns the bit-wise NOT of the operand.

<< operator

```
operand1 << operand2
```

Returns the result of a bit-wise left shift of operand1 by the number of bits specified by operand2. Zeros are shifted in from the right.

>> operator

```
operand1 >> operand2
```

Returns the result of a bit-wise (sign propagating) right shift of operand1 by the number of bits specified by operand2.

>>> operator

```
operand1 >>> operand2
```

Returns the result of a bit-wise (zero filling) right shift of operand1 by the number of bits specified by operand2. Zeros are shifted in from the left.

Special operators

?: operator

```
expression ? resultIfTrue : resultIfFalse
```

This operator evaluates its first operand, the expression. If the expression is true, the value of the second operand (resultIfTrue) is returned; otherwise the value of the third operand (resultIfFalse) is returned.

, operator

```
expression1, expression2
```

This operator evaluates its first and second operand (expression1 and expression2) and returns the value of the second operand (expression2).

The comma operator can be subtle and is best reserved only for use in argument lists.

function operator

```
var variable = function( optArguments ) { Statements }
```

This operator is used to create anonymous functions. Once assigned, the variable is used like any other function name, example variable (1, 2, 3). Specify the argument names (in

optArguments) if named arguments are required. If no optArguments are specified, arguments may still be passed and will be available using the arguments list.

The JavaScript function operator supports closures, for example:

```
function make_counter( initialValue )
{
    var current = initialValue;
    return function( increment ) { current += increment; return current; }
}
// ...
var counterA = make_counter( 3 );  // Start at 3.
var counterB = make_counter( 12 ); // Start at 12.
var x1 = counterA( 2 );  // Adds 2, so x1 == 5
var x2 = counterB( 2 );  // Adds 2, so x2 == 14
var x3 = counterA( 7 );  // Adds 7, so x3 == 12
var x4 = counterB( 30 ); // Adds 30, so x4 ==44
```

Note that for each call to make_counter(), the anonymous function that is returned has its own copy of current (initialized to the initialValue), which is incremented independently of any other anonymous function's current. It is this capturing of context that makes the function that is returned a closure.

See also function declaration and Function type.

in operator

```
property in Object
```

Returns true if the given Object has the given property; otherwise returns false.

instanceof operator

```
object instanceof type
```

Returns true if the given object is an instance of the given type, (or of one of its base classes); otherwise returns false.

new operator

```
var instance = new Type( optArguments );
```

This function calls the constructor for the given Type, passing it the optional arguments (optArguments) if any and returns an instance of the given Type. The Type may be one of the built-in types, one of the library types, or a user-defined type.

Example:

```
var circle = new Circle( x, y );
var file = new File();
```

this operator

```
this.property
```

The "this" operator may only be used within a function that is defined within a class, that is a member function. Within the scope of the function this is a reference to the particular instance (object) of the class's type.

Example:

```
class MinMax {
    var _min;
    var _max;
    function MinMax( min, max ) { this._min = min; this._max = max; }
    function max() { return this._max; }
    function min() { return this._min; }
    function setMax( value ) { this._max = value; }
    function setMin( value ) { this._min = value; }
}
```

typeof operator

```
typeof item
```

This operator returns a type of the object as a string.

Example:

```
var f = new Function("arguments[0]*2"); // "object"
var str = "text"; // "string"
var pi = Math.PI; // "number"
```

Functions and built-in objects have a "typeof" of "function".

5.2.5. Declarations

Classes, functions, variables and constants are declared with class, function, var and const respectively.

Reserved words

JavaScript reserves some words which are valid identifiers for its own use:

- Currently used for declarations: class, function, var and const.
- Reserved for future use: boolean, byte, char, debugger, double, enum, export, float, goto, implements, import, int, interface, long, native, short, synchronized, throws, transient and volatile.

It is unadvisable to use any of these words as identifiers.

class

```
class ClassName {
    static var ClassVariable;
    var MemberVariable;
    static function ClassFunction { Statements; }
    function ClassName() { Statements; } // constructor
    function MemberFunction() { Statements; }
}
```

This keyword is used to define new classes. After the keyword class comes the ClassName, then optionally, the keyword extends followed by a class name from which this class derives, then an opening brace. Class variables are declared with static var (ClassVariable). Only one instance of these variables exists per class. Member variables (MemberVariable) are declared with var; each instance (object) of the class has its own copy of each member variable. Functions declared with the keywords static function are class functions (ClassFunction); these functions are called using the name of the class rather than from an object. In the standard library, the Math functions are all static, for example Math.sin(). Member functions are called by objects and are declared with function. The constructor is the member function with the same name as the class; constructors must not contain an explicit return statement or have an explicit return type, since JavaScript handles these automatically.

A class that only contains static const, var and function definitions does not need a constructor.

Example:

```
class Area {
    static var count = 0;
    var _name;
    function Area( name ) { this._name = name; this.count++ }
    function destroy() { this.count--; }
    static function count() { return this.count; }
    function name() { return this._name; }
    function setName( name ) { this._name = name; }
}
```

In this example we define the "Area" class. When an instance of the class is constructed:

```
var area = new Area( "Berkshire" );
```

the given name is assigned to the object's _name variable and the class's static count is incremented. The destroy() function is not called automatically; it must be called explicitly if there is any clean-up to do. JavaScript does not have destructors.

All the class's variables and functions must be defined within the class definition.

Classes are all derived from Object. It is possible to derive a class from another class using the extends keyword, for example:

```
class City extends Area {
    var _population;
    function City( name, population )
    {
        Area( name );
        _population = population;
    }
    function population() { return _population; }
    function setPopulation( population ) { _population = population; }
}
```

See also function.

const

```
const identifier = Value;
```

This keyword is used to define constant values. The identifier is created as a constant with the given Value. The constant is global unless defined within the scope of a class or function.

Example:

```
const PI2 = Math.PI * 2;
const COPYRIGHT = "Copyright (c) 2006";
```

Attempts to assign to a constant cause the interpreter to issue an error message and stop.

function

```
function functionName( arguments )
{
    Statements;
}
```

Functions may also be declared within the scope of a class definition. In this situation, the functions become member functions of the class that contains them. See class.

var

```
var variableName;
var anotherVariableName = InitialValue;
```

This keyword is used to declare and optionally initialize variables. If just the variableName is given, the variable is created, but it has no value, that is, its value is undefined. If an InitialValue is given, the variable is created and assigned this InitialValue. Variables declared within functions are local to the function in which they are declared. Variables declared outside of functions and classes are global.

Example:

```
var i;
var count = 22;
var str = "string";
```

5.2.6. Control statements

The flow-of-control in JavaScript programs is controlled by control statements, for example: if, switch, for and while. JavaScript also supports exceptions with try and catch.

This topic discusses control statements in alphabetical order: break, case, catch, continue, default, do, else, for, if, finally, label, return, Switch, throw, try, try...catch, try...finally, while, with.

break

```
label:
for ( var i = 0; i < limit; i++ ) {
    if ( condition ) {
        break label;
    }
}
switch ( expression ) {
    case 1:
        Statements1;
```

```
        break;
    default:
        DefaultStatements;
        break;
}
```

This keyword is used in, for loops, do loops, while loops and switch statements. When a break statement is encountered in a loop, control is passed to the statement following the end of the inner-most loop that contains the break statement; unless the break statement is followed by the name of a label, in which case control passes to the statement governed by the label.

A break statement is usually placed at the end of each case in a switch statement to prevent the interpreter "falling through" to the next case. When the interpreter encounters a break statement, it passes control to the statement that follows the inner-most enclosing switch statement. If every case has a corresponding break, then at most one case's statements will be executed.

If the break statement is followed by a label name (see label) then when the break is encountered, control will pass to the statement marked with that label; this is useful, for example, for breaking out of deeply nested loops.

Example:

```
red:
for ( x = 0; x < object.width; x++ ) {
    for ( y = 0; y < object.height; y++ ) {
        if ( color[x][y] == 0xFF0000 ) {
            break red;
        }
    }
}
```

case

```
switch ( expression ) {
    case Value:
        Statements;
        break;
    default:
        DefaultStatements;
        break;
}
```

This keyword is used in switch statements. For each possible value that a switch statement's expression may evaluate to, one case may be written, (but see default.) If a case's literal value (Value) matches the value of the switch statement's expression, then that case's statements (Statements) are executed.

Normally a case's statements are terminated by a break statement which causes execution to pass to the end of the switch statement.

See Switch for an example.

catch

```
try {
    Statements;
}
catch( exception ) {
    ExceptionStatements;
```

```
}
```

This keyword is used in try statements. If an exception occurs, then the ExceptionStatements in a matching catch block are executed.

See try for an example. Also, see throw.

continue

```
for ( var i = 0; i < limit; i++ ) {
    if ( condition ) {
        continue;
    }
    Statements;
}
```

This keyword is used within the context of a for, while or do loop.

If a continue statement is encountered in a for loop, execution immediately passes to the third part of the for loop (normally where a counter is incremented or decremented), and then execution continues normally, that is, the middle part of the for loop (the conditional) is tested and if true, the body of the loop is executed.

If a continue statement is encountered in a while or do loop, execution immediately passes to the conditional, which is retested; the body of the loop will be executed if the conditional is still true.

See do for an example.

default

```
switch ( expression ) {
    case 1 :
        Statements1;
        break;
    default :
        DefaultStatements;
        break;
}
```

This keyword is used in switch statements. It is used instead of case to match anything that the switch statement's expression has evaluated to. If no default is used, and none of the cases match, then the switch statement will not execute anything and control will pass on to the following statement. If default is used, it must be the last case in the switch statement. This is because each case is evaluated in order and since default matches any value, it will always be executed if the interpreter reaches it and any following cases would always be ignored. When the default case is encountered its DefaultStatements are executed. It is customary to end a default statement with a break.

See Switch for an example.

do

```
do {
    Statements;
} while ( condition );
```

This keyword is used in conjunction with while to form a loop which is guaranteed to execute at least once.

The Statements in the braces following the do are executed once. If the while condition evaluates to true, execution passes back to the do and the whole process repeats. If the while loop's conditional ever becomes false, execution continues from the statement following the while statement.

Example:

```
var x = 0;
do {
    x += 5;
    if ( x == 50 )
        continue;
    } while ( x < 100 );
```

The example outputs 5, 10, 15, ..., 45, 55, 60, 65, ..., 95.

See also continue and break.

else

```
if ( condition ) {
    Statements;
}
else {
    ElseStatements;
}
```

The else keyword is used in conjunction with if. See if for details.

for

```
for ( i = 0; i < limit; i++ ) {
    Statements;
}
```

This keyword is used to create a loop that executes a fixed number of times.

The for statement is broken into parts as follows: the keyword for, an opening parentheses, zero or more statements (the first part), a semi-colon, a conditional expression (the second part), a semi-colon, zero or more statements (the third part), a closing parentheses and finally a statement or block that is governed by the for loop.

The first part of the for statement is typically used to initialize (and possibly declare) the variable used in the conditional in the second part of the for loop statement. This part is executed once before the loop begins. This part may be empty.

The second part contains a conditional expression. The expression is evaluated before each iteration of the loop (including before the first iteration). If this expression evaluates to false, the statement or block governed by the for loop is not executed and control passes to the statement that follows. If the condition is never true, the statement or block governed by the for loop will never be executed. If the condition expression is true, the statement or block governed by the for loop is executed and then the third part of the for statement is executed, before control is passed back to the conditional expression with the whole process being repeated. This part should not be empty.

The third part contains statements which must be executed at the end of every iteration of the loop. This part is typically used to increment a variable that was initialized in the first part and whose value is tested in the second part. This part may be empty.

Example:

```
var a = [ "a", "b", "c", "d", "e" ];
for( var i = 0; i < a.length; i++ ) {
    s.log( -1, a[ i ] );
    }
```

See also continue and break.

if

```
if ( expression1 ) {
    // statements1
} else {
    // elsestatements
}
if ( expression1 ) {
    // statements1
} else if ( expression2 ) {
    // statements2
}
// else if ...
} else {
    // elsestatementsN
}
```

An if statement provides a two-way branch. A multi-way branch is achieved using else ifs. (See also switch.)

If the first expression, expression1, is true, then the statements governed by that expression (statements1) will be executed, after which control will pass to the statement following the if block.

If expression1 is false, control passes to the else statement. If the else has no following if, the else statements (elsestatements) are executed, after which control will pass to the statement following the if block. If the else has a following if, then step 1 or step 2 (this step) is repeated for that if statement depending on whether its expression is true or false.

finally

```
try {
    Statements;
}
finally {
    finalStatements;
}
```

A "finally" block is a block of code that is executed after the governing try block. If no exceptions occur within the try block, control will pass to the finally block after the last statement in the try block has been executed. If an exception occurs within the try block, control is passed immediately to the finally block.

See try...finally for an example.

label

```
labelname: Statement;
```

Statements may be labelled; the syntax is labelname followed by a colon, followed by the Statement. The labelname is any valid (unique) identifier.

See break for an example.

return

```
return optExpression;
```

A return statement may occur anywhere within a function, including member functions, but not within constructors. When a return statement is encountered, control leaves the function immediately and execution resumes at the statement following the call that invoked the function. If the return statement is followed by an expression (optExpression) the value of the expression is returned to the caller.

Example:

```
function inRange( v, min, max )
{
    if ( v >= min && v <= max ) {
        return true;
    }
    else {
        return false;
    }
}
```

switch

```
switch( expression ) {
    case Value :
        Statements;
        break;
    default :
        DefaultStatements;
        break;
}
```

A switch statement provides a multi-way branch. The expression is evaluated once, then each case is checked in order to find one with a Value that matches the value of the expression. If a match is made, the statements of the matching case are executed, after which control passes to the statement following the switch block. If no matching case is found and there is a default case, the DefaultStatements are executed, after which control passes to the statement following the switch block. If there is no matching case and no default, no statements within the switch block are executed and control passes to the statement following the switch block.

Note that if a default is used it must be come after all the cases; this is because once default is encountered it is treated as a matching case regardless of what follows.

Every case and the default (if used) should have a break as their last statement. If break is not present, control will "drop through" to the following statements, which is not usually the desired behavior.

The expression may be any arbitrary JavaScript expression that evaluates to an object that can be strictly compared. For example, an expression that evaluates to a Boolean, Date, Number or String value.

Example:

```
switch( expr ) {
    case 1:
        doActionOne();
        break;
    case "two":
        doActionTwo();
        break;
    case 3:
        doActionThree();
    case 4:
        doActionFour();
        break;
    default:
        doDefaultAction();
        break;
}
```

In the example, if expr has the value 1, the doActionOne() function will be called. But if expr has the value 3, both doActionThree() and doActionFour() will be called because case 3 does not have a break statement and execution "falls through" to case 4. If expr is not 1, "two", 3 or 4 then the default case will be matched and doDefaultAction() will be executed.

throw

```
try
{
    // Statements
    throw "an exception";
}
catch( e )
{
    if ( String( e ) == ( "Error: " + "an exception" ) )
    {
    // Exception statements
    }
    else
    {
    // Other exception statements
    }
}
```

The throw keyword is used to raise user-defined exceptions.

Example:

```
function monthToName( i )
{
    var IndexToMonth = [ "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                         "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" ];
    if ( i < 0 || i > 11 ) {
        throw "month number out of range";
    }
    else {
        return IndexToMonth[ i ];
    }
}
```

It is also possible to define a user-defined exception class and throw an object of that type, example:

```
   class AUserDefinedException
    {
    }

    try
    {
        throw new AUserDefinedException();
    }
    catch( e )
    {
        s.log( 2, "Your object-exception: %1", e ); // Your object-exception: Error:
 [class AUserDefinedException]

    }
```

See also try.

try

```
try {
    Statements;
}
catch ( e ) {
}
try {
    Statements;
}
finally {
}
```

The try keyword is used to identify a statement block for which exceptions will be caught. There are two kinds of try block, try...catch and try...finally.

try...catch

If an exception occurs within a try...catch block, control is passed to the first catch block. For each try block you can have only one catch block.

See catch for details of the qualifiers.

try...finally

If an exception occurs within a try...finally block, control is passed to the finally block. This is useful if you want to ensure that something happens at the end of the block, no matter what.

Examples:

```
try {
    file = new File;
    file.open( filename );
    process( file );
}
finally {
    file.close();
}
```

In this example, the file is always closed, even if an exception occurred.

```
try
{
    var a = monthToName( 0 );
    var b = monthToName( 13 );
}
catch ( e )
{
    if ( String( e ) == ( "Error: " + "month number out of range" ) )
    {
        s.log( 2, "Code error: %1", e );
    }
    else
    {
        throw e;
    }
}
```

In this example, the monthToName() function is called to set two variables. If the function fails, it throws an exception rather than returns an error value, so no explicit check for an error value is necessary. If one of the function calls failed s.log() is called; otherwise the exception is re-thrown so that it can be handled at a higher level. (See throw for the definition of monthToName().)

while

```
while ( condition ) {
    Statements;
}
```

This keyword is used to repeat a block of code zero or more times. When the while statement is encountered the condition is evaluated. If the condition is true, the Statements in the while block are executed; otherwise control passes to the statement following the while block. If the condition is true, after the Statements have been executed, the condition is again evaluated and the whole process repeats.

Example:

```
var i = 10;
while ( i > 0 ) {
    i--;
}
```

See also continue and break.

with

```
with ( QualifyingName ) {
    Statements;
}
```

This keyword is used to indicate that any unqualified names in the Statements should be qualified by QualifyingName.

```
Example:
```

```
// Without with
s.log( -1, Math.abs( -4 ) );
```

```
s.log( -1, Math.pow( 2, 3 )  );
s.log( -1, Math.random() );

// With with
with ( Math ) {
    s.log( -1, abs( -4 ) );
    s.log( -1, pow( 2, 3 ) );
    s.log( -1, random() );
}
```

If multiple qualifying names are required, with statements can be nested.

Forcing the interpreter to do the lookup may be slower than using the fully qualified names.

5.3. Utility module

5.3.1. Text encoding

JavaScript strings represent Unicode characters (internally encoded in UTF-16). Care must be taken when converting a Unicode string to or from a sequence of 8-bite bytes, such as a file on disk (File class) or a byte array in memory (ByteArray class).

The Codecs supported by scripting API can be classified as follows:

Codecs	Description	Data conversion
UTF-16, UTF-16BE, UTF-16LE	Map Unicode code points to 16-bit (two byte) representation	String to ByteArray / File & ByteArray / File to String
UTF-8, latin1, Apple Roman, ...	Map Unicode code points to 8-bit (single byte) representation	String to ByteArray / File & ByteArray / File to String
Hex, Base64	Map an 8-bit byte stream into a 7-bit ASCII representation	String to ByteArray / File & ByteArray / File to String & ByteArray to ByteArray

Codec

A codec (short-hand for "code-decode") specifies the text encoding used to represent a Unicode string as a sequence of 8-bit bytes. Various codecs are in use throughout the industry.

The utility module refers to supported codecs through predefined names. To specify a codec in a function argument, use one of the following strings:

- Apple Roman
- Big5
- Big5-HKSCS
- CP949
- EUC-JP
- EUC-KR
- GB18030-0
- IBM 850
- IBM 866
- IBM 874

- ISO 2022-JP
- ISO 8859-1 to 10
- ISO 8859-13 to 16
- Iscii-Bng, Dev, Gjr, Knd, Mlm, Ori, Pnj, Tlg, and Tml
- JIS X 0201
- JIS X 0208
- KOI8-R
- KOI8-U
- MuleLao-1
- ROMAN8
- SHIFT-JIS
- TIS-620
- TSCII
- UTF-8
- UTF-8BOM (see byte order marker below, does NOT work for ByteArrays)
- UTF-16
- UTF-16BE
- UTF-16LE
- UTF-32
- UTF-32BE
- UTF-32LE
- Windows-1250 to 1258
- WINSAMI2

In addition the utility module supports the following special codecs (See Hex and Base64 below for more information):

- Hex (corresponds to xsd:hexBinary)
- Base64 (corresponds to xsd:base64Binary and SOAP-ENC:base64)
- toHex, toBase64
- fromHex, fromBase64

Note: If the codec argument is missing or null, the codec is set to UTF-8 on Mac and to the current ANSI code page on Windows.

Conversion errors

When converting from a byte sequence to a text string, byte sequences that do not match the requirements for the codec are replaced by the Unicode representation of a question mark ("?").

When converting from a text string to a byte sequence, Unicode code points that cannot be represented as a byte sequence using the codec are replaced by the ASCII representation of a question mark ("?").

Byte order marker (BOM): Input (from a byte sequence to a text string)

Regardless of the specified codec, the utility module automatically recognizes a UTF-16 BOM and when found, the codec is automatically adjusted to the appropriate UTF-16 variant. For example, if the codec argument is set to "UTF-8", the utility module will correctly read UTF-8, UTF-16BE and UTF-16LE.

```
var ba = new ByteArray("un garçon", "UTF-16");
var bastr = "";
for(i = 0; i < ba.length; i++){
    bastr += ba.getAt(i) + " ";
}
```

```
job.log(1, bastr); //starts with 255 254 (UTF-16LE) or 254 255 (UTF-16BE)
var str = ba.toString("UTF-16");
job.log(1, str); //"un garçon"
```

Byte order marker (BOM): Output (from a text string to a byte sequence)

When the codec is set to one of the UTF-16 variants, the utility module always outputs an appropriate 2-byte UTF-16 BOM at the beginning of the data.

When the codec is set to UTF-8, the utility module does not output a BOM. This is the standard behavior expected by most applications.

```
var ba = new ByteArray("sample string", "UTF-8");
var bastr = "";
for(i = 0; i < ba.length; i++){
    bastr += ba.getAt(i) + " ";
}
job.log(1, bastr); //starts with 115 (73h:'s') 97 (61h:'a') 109 (6dh:'m')
var str = ba.toString("UTF-8");
job.log(1, str); //"sample string"
```

When the codec is set to UTF-8BOM, the utility module outputs a 3-byte UTF-8 BOM at the beginning of the data (this is just a marker to indicate UTF-8 since there are no byte-order issues in this encoding). Be careful when using this codec, because many applications are not able to correctly interpret a UTF-8 BOM. This encoding is not supported for ByteArrays.

```
//UTF-8BOM use case
//create a sample file
var path = job.createPathWithName("file.txt");
var f = new File(path, "UTF-8BOM");
f.open(File.WriteOnly);
f.writeLine("This is a line.");
f.close();
job.sendToSingle(path);//sends file to single outgoing connection
//result in hex starts with ef bb bf (BOM) 54 68 ...
```

Hex and Base64

Converting between a String and a ByteArray with any of the codecs (except Hex and Base64) is always unambiguous: the String contains the Unicode code points, the ByteArray contains the 16-bit or 8-bit representation.

With a Hex or Base64 codec, the situation is problematic because:

- Both decoded and encoded sides can be represented in 8 bits
- Following the logic that "the encoded side is in the ByteArray" (as with the other codecs) contradicts the frequent use case where "the encoded side is in a String" (for example, part of an XML stream).

To resolve this issue, the 10 release introduces the following directional codecs:

Codec	Description
toHex toBase64	Map an 8-bit byte stream into the specified 7-bit ASCII representation, regardless of the source and target data types If the source is a String, only the 8 lowest bits of each Unicode code point are used (as if latin1 encoding was used)
fromHex fromBase64	Map a 7-bit ASCII representation of the specified type into an 8-bit byte stream, regardless of the source and target data types

Codec	Description
	If the source is a String, only the 8 lowest bits of each Unicode code point are set and higher bits are cleared (as if latin1 encoding was used)

For compatibility reasons, the unidirectional codecs Hex and Base64 can still be used:

- Hex/Base64 will operate as toHex/toBase64 when the target is a ByteArray

```
var ba = new ByteArray("sample string", 'Hex'); //toHex
var bastr = "";
for(i = 0; i < ba.length; i++){
   bastr += ba.getAt(i) + " ";
}
job.log(1, bastr); //starts with 55 (7 ASCII) 51 (3 ASCII): 73 (115dec:'s')
```

- Hex/Base64 will operate as fromHex/fromBase64 when target is a String.

```
//declaration of ba see above
var str = ba.toString('Hex'); //fromHex
job.log(1, str); //"sample string"
```

5.3.2. ByteArray class

An instance of the ByteArray class represents a sequence of 8-bit bytes (as opposed to Unicode characters).

Constructors

ByteArray (length : Number) : ByteArray

Constructs a new byte array with the specified number of bytes and clears all bytes to zero.

An exception is thrown if the specified length is negative or is not a number. All numbers are converted to an integer (e.g. 2.5 becomes 3).

```
var ba;
try{
    ba = new ByteArray(-1);
}catch( exception ){
    job.log(1, "exception thrown");
}//"exception thrown"

try{
    ba = new ByteArray(3);
    job.log(1, "ByteArray constructed, length = " + ba.length);
}catch( exception ){} //"ByteArray constructed, length = 3"

//get first element
job.log(1, ba.getAt(0));//0
```

ByteArray (source : String, codec : String) : ByteArray

Constructs a new byte array by converting a source string with the specified codec; see text encoding.

If no codec is specified, the default one is used.

```
var ba = new ByteArray("Hello","UTF-8");
var bastr = "";
for(var i = 0; i < ba.length; i++){
    bastr += ba.getAt(i) + " ";
```

```
}
job.log(1, bastr); //gives 72 101 108 108 111 (= "Hello" in UTF-8)
```

ByteArray (part1 : ByteArray, part2 : ByteArray) : ByteArray

Constructs a new byte array by concatenating the two specified parts.

```
var ba1 = new ByteArray("Hello", "UTF-8");
var ba2 = new ByteArray("olleH", "UTF-8");
var ba = new ByteArray(ba1, ba2);
var bastr = "";
for(var i = 0; i < ba.length; i++){
    bastr += ba.getAt(i) + " ";
}  job.log(1, bastr); //gives 72 101 108 108 111 111 108 ...
```

Properties

length : Number

size : Number

The number of bytes in the byte array. This value is read-only.

Member functions

getAt(index : Number) : Number

Returns the value of the byte at the specified zero-based index in the byte array. The returned value is an integer in the range [0..255]. An exception is thrown if the index is out of range or if it is not an integer.

putAt(index : Number, value : Number)

Stores a new value for the byte at the specified zero-based index in the byte array. The specified value must be an integer in the range [0..255]. An exception is thrown if the value or the index are out of range, or if they are not integers.

getSubArray (start : Number, length : Number) : ByteArray

Returns the specified portion of the byte array (zero-based start index). An exception is thrown if the requested portion is out of range, or if start and length are not integers.

toString (codec : String) : String

Returns the string obtained by converting the byte array using the specified codec, text encoding.

See *text encoding* for a sample.

convertTo (codec : String) : ByteArray

Returns a new ByteArray which is the re-encoded version of the current one. Supported codecs are Hex and Base64. This function recodes the entire ByteArray even if the data includes null bytes.

Example: ByteArray ('abc', 'UTF-8').convertTo('Hex') returns a ByteArray of length 6 with bytes 54 49 54 50 54 51.

```
//ByteArray use case
//get & put
var ba = new ByteArray(4);
ba.putAt(0, 1);
ba.putAt(1, 5);
ba.putAt(2, 3);
ba.putAt(3, 2);
job.log(1, ba.getAt(0)); //"1"
var subba = ba.getSubArray(1,2);
```

```
job.log(1, subba.getAt(0)); //"5"

//convertTo
var convba = new ByteArray ('abc', 'UTF-8').convertTo('Hex');
var bastr = "";
for(var i = 0; i < convba.length; i++){
    bastr += convba.getAt(i) + " ";
}
job.log(1, bastr); //starts with 54 ("6") 49 ("1"): 61 (=97 dec:"a")
```

Signatures

The ByteArray class offers the following additional functions to generate and verify signatures using standard RSA algorithms (compatible with PKCS#1 v2.0).

generateSignature(privateKeyPath : String) : ByteArray

Returns the digital signature for the data in the receiving ByteArray using the specified private key. This signature allows a receiver to verify that the data was indeed generated by a script that has access to the specified private key (assuming that the receiver has access to the public key corresponding to the private key).

The signature is generated by first calculating SHA-1 digest of the byte array, then encoding the digest using ASN.1 notation and encrypting the resulting data block using the private key.

The privateKeyPath argument specifies the path to a private key file in the standard PKCS #8 format. Since this format is not encrypted or password protected, the user must provide other means to safeguard the private key from inappropriate access.

If the privateKeyPath argument is missing or null, the private key built into Switch is used instead. The resulting signature allows a receiver to verify that the data was indeed generated by a script running in Switch (since the public key corresponding to the private key is part of the public Switch documentation).

Note: This function is deprecated as of Switch 2017 update 2. Please only use generateSignature to maintain compatibility with older Switch versions. In all other cases, please use getSignature instead!

getSignature(privateKeyPath : String, hashFunction : String, privateKeyPassword : String) : ByteArray

Returns the digital signature for the data in the receiving ByteArray using the specified private key. This signature allows a receiver to verify that the data was indeed generated by a script that has access to the specified private key (assuming that the receiver has access to the public key corresponding to the private key).

The signature is generated by first calculating the digest of the byte array using the specified hash function, then encoding the digest using ASN.1 notation and encrypting the resulting data block using the private key.

The privateKeyPath argument specifies the path to a private key file in the PEM format.

The hashFunction argument specifies the name of the hash function to use for calculating the digest of the byte array. The following hash functions are supported:

- md4
- md5
- sha
- sha1
- sha224
- sha256

- sha384
- sha512
- ripemd160

The privateKeyPassword argument is optional; it specifies the passphrase to decrypt the private key. This parameter can be omitted if the private key is not password-protected.

verifySignature(signature : ByteArray, publicKeyPath : String) : Boolean

Verifies whether the specified digital signature fits the data in the receiving ByteArray using the specified public key. If this function returns true, the data was indeed generated by a sender with access to the private key corresponding to the specified public key (with the certainty level offered by PKCS#1 v2.0).

The publicKeyPath argument specifies the path to a public key file in the standard Base64 encoded X.509 format (also called PEM).

If the publicKeyPath argument is missing or null, the public key corresponding to the private key built into Switch is used instead. This avoids including an explicit reference to the public key which is part of the public Switch documentation.

Note: This function can verify signatures generated by both generateSignature (deprecated as of Switch 2017 update 2) and getSignature.

Example: the following code would generate a signature for authenticating Switch with the Alwan CMYK Optimizer CLI:

```
var strData = "2008-10-12T14:16:22+01:00/var/tmp/foo.pdf/var/tmp/bar.pdf";
var binData = new ByteArray(strData, "UTF-8");
var binSignature = binData.generateSignature();
var strSignature = binSignature.toString("toBase64");
```

Public Key in C++ syntax:


```
-----BEGIN CERTIFICATE-----
MIIBgTCCASugAwIBAgIJAN+2brjXZa75MA0GCSqGSIb3DQEBBQUAMA0xCzAJBgNV
BAYTAkJZMB4XDTA5MDUxMjA4MDAwMVoXDTA5MDYxMTA4MDAwMVowDTELMAkGA1UE
BhMCQlkwXDANBgkqhkiG9w0BAQEFAANLADBIAkEAqm5CNFaMI/Jc7++ySmLdVb9W
Y/C+l5Zt1qr8v4qObYZOgpuwucDh9QzC7wgWnCWUINs9xgcU9p6WwrSFcyAbqQID
AQABo24wbDAdBgNVHQ4EFgQUTFkQhg1pkDGONt6MFc2zF+L2vRswPQYDVR0jBDYw
NIAUTFkQhg1pkDGONt6MFc2zF+L2vRuhEaQPMA0xCzAJBgNVBAYTAkJZggkA37Zu
uNdlrvkwDAYDVR0TBAUwAwEB/zANBgkqhkiG9w0BAQUFAANBAKjV5bqRnkgufLyS
ddAIwhyvWa5nvVbobF26cEE+7jkC+fspsSvnJDLIW72zCKFEqtnBtEM4uCweYnDV
ODWBKjg=
-----END CERTIFICATE-----
```


5.3.3. File class

The File class provides functionality for reading and writing binary and text files. A File can be instantiated as an object, giving the script developer complete flexibility when reading and writing files. In addition, the File class provides a set of static convenience functions for reading and writing files in one go.

Enum AccessMode

The enum AccessMode is used in conjunction with File.open() to specify the mode in which the file is opened. The access mode is specified by OR-ing together values from the following list.

Access mode	Description
File.ReadOnly	Opens the file in read-only mode.
File.WriteOnly	Opens the file in write-only mode. If the file already exists, the file is truncated and created otherwise.
File.ReadWrite	Opens the file in read/write mode; If the file does not exist, the file will be created. If the file already exists, the file is not truncated and the write pointer starts at the end of the file while the read pointer starts at the beginning.
File.Append	Opens the file in append mode; If the file does not exist, the file will be created. If the file already exists, the file is not truncated and the write pointer starts at the end of the file. This mode is very useful when you want to write something to a log file.
File.Translate	Enables line break translation (carriage return and linefeed) for text files to platform-specific conventions. File.Translate should only be used in combination with File.ReadOnly.
File.Truncate	If possible, the device is truncated before it is opened. All earlier content is lost.

Static functions

exists(fileName : String) : Boolean

Returns true if fileName exists; otherwise returns false.

remove(fileName : String)

Deletes the file fileName if possible; otherwise throws an exception.

isFile(fileName : String) : Boolean

Returns true if fileName is a file; otherwise returns false.

isDir(fileName : String) : Boolean

Returns true if fileName is a directory; otherwise returns false.

isType(fileName : String, ext : String) : Boolean

Returns true if the job matches the specified file type, specified as a filename extension, and false otherwise.

A file matches if its filename extension and/or its Mac file type (after conversion) match the specified filename extension. A folder matches if any of the files at the topmost level inside the folder match the specified type. These semantics are similar to those of matching cross-platform file types in filter connections.

Checking for arrival

This function checks if a file has "arrived":

- If the file doesn't exist or can't be opened with exclusive read access, return false.
- Note the file's size, and wait for a specified period of time ("stability period").

- If the file's size has changed during the period, return false.
- If the file can't be opened with exclusive read access after the period, return false.
- Otherwise return true.

hasArrived(fileName : String, stabilitySeconds : Number) : Boolean [static]

hasArrived(stabilitySeconds : Number) : Boolean

Returns true if it can be assumed that the file has arrived according to the heuristic describe above; otherwise returns false. The stability period is specified in seconds (as a floating point number).

The length of the required stability period depends on the nature and size of the file, on the process writing the file, and on the performance and workload of the computer system(s) involved. In most cases a stability period of 1 second is both sufficient and acceptable (in terms of delay). When the file is produced by a slow process or transferred across a slow network, a much longer stability period may be required.

A hasArrived() function often returns only after a delay of at least the stability period. However if it can be determined right away that the file has not yet arrived, the function returns immediately. Thus it is never acceptable to invoke a hasArrived() function in a tight loop.

The following table describes the two most important use cases for dealing with an external application through the hasArrived() functions. In fact, the guidance presented in the table does not depend on the employed arrival/ready detection mechanism.

Model	Description	Implementation
Synchronous	If the external process is synchronous, resource-intensive and hosted on the same computer as Switch, the jobArrived entry point should block until the process is finished (so that Switch can correctly manage the number of external processes running in parallel)	jobArrived entry point contains a waiting loop that repeatedly calls hasArrived() followed by Environment.sleep() to introduce additional delay
Asynchronous	If the external process is asynchronous (i.e. you want to feed it the next job even if the previous job hasn't been processed), if it depends on user interaction, or if it potentially takes a long time without using a lot of resources on the local computer, the jobArrived entry point should return immediately and leave the follow-up to the timerFired entry point	timerFired entry point simply calls hasArrived() once, since it is automatically executed repeatedly with a predefined interval

toNativeSeparators(fileName : String) : String [static]

Returns fileName with the '/' separators converted to separators that are appropriate for the underlying operating system.

```
job.log(1, File.toNativeSeparators("this/is/a\\test"));
```

```
//"this/is/a/test" on Macintosh
//"this\is\a\test" on Windows
```

fromNativeSeparators(fileName : String) : String [static]

Returns fileName using '/' as file separator.

```
job.log(1, File.fromNativeSeparators("this/is/a\\test"));
//"this/is/a/test"
```

Static Text I/O functions

read(fileName : String, codec : String) : String

Reads and returns the contents of the file fileName if possible; otherwise throws an exception.

The optional argument codec specifies the text encoding used to read the file; see text encoding.

write(fileName : String, content : String, codec : String)

Writes the string content to the file fileName if possible (completely replacing the original contents if the file already exists); otherwise throws an exception.

The optional argument codec specifies the text encoding used to write the file contents; see text encoding.

```
//copies the content of the jobfile (in UTF-8) to a new path
//read the contents of the jobfile
var str = File.read(job.getPath(), "UTF-8");

//create new path
var destPath = job.createPathWithName("dest." + job.getExtension());

//write the contents to destPath
File.write(destPath, str, 'UTF-8');

//send to single outgoing connection
job.sendToSingle(destPath);
```

Static Binary I/O functions

The functions presented in this section read and write sequences of bytes (rather than text streams). Hence there is no need to specify a codec (see text encoding).

readByteArray(fileName : String) : ByteArray

Reads and returns the content of the file fileName if possible; otherwise throws an exception.

writeByteArray(fileName : String, content : ByteArray)

Writes the content to the file fileName if possible (completely replacing the original contents if the file already exists); otherwise throws an exception.

```
//example
//create a file to write a ByteArray to
var path = job.createPathWithName("ba.txt");
var f = new File(path);

//write some data to the file
f.open(File.WriteOnly);
f.writeLine("this is text.");
f.close();

var ba = new ByteArray("sample", "UTF-8");
File.writeByteArray(path, ba); //note: old contents are removed first
```

```
job.log(1, File.readByteArray(path).toString("UTF-8")); //"sample"
```

Constructor

File(fileName : String, codec : String)

Creates a file object with the fileName. If fileName is missing or is not a String, an exception is thrown.

The optional argument codec specifies the text encoding used to read or write the file contents; see text encoding (if codec is missing or null, the codec is set to UTF-8 on Mac and to the current ANSI code page on Windows).

```
var fpath = job.createPathWithName("file.txt");
var file = new File(fpath, "UTF-8");
```

Properties

The File object's properties are read-only.

name : String

The name of the file including the extension.

e.g.: "SampleFile.txt"

path : String

The path of the file (= folder that contains the file) .

e.g.: "C:/Users/User1/Documents" (with 'Documents' containg the file)

fullName : String

The fullName of the file, including path, name, and extension.

e.g.: "C:/Users/User1/Documents/SampleFile.txt"

baseName : String

The name of the file, excluding its path and extension.

e.g.: "SampleFile"

extension : String

The file name's extension.

e.g.: "txt"

symLink : String

The expansion of the symlink if the file is a symlink; otherwise empty.

e.g.: "C:/Users/User1/Documents/SampleFile.txt" with the file a symlink to 'SampleFile.txt'

exists : Boolean

True if the file exists; otherwise false.

readable : Boolean

True if the file is readable; otherwise false.

writable : Boolean

True if the file is writable; otherwise false.

executable : Boolean

True if the file is executable; otherwise false.

hidden : Boolean

True if the file is hidden; otherwise false.

eof : Boolean

True if reading has reached the end of the file; otherwise false.

created : Date

The creation time of the file.

lastModified : Date

The last modification time of the file.

lastRead : Date

The last time the file was read.

size : Number

The size of the file, in bytes.

codec: String

The codec being used for reading and writing the file's contents; see text encoding.

Member functions

open(accessMode : Number)

Opens the file in the mode accessMode (see enum AccessMode) if possible; otherwise throws an exception.

For example:

```
profile.open(File.ReadOnly);
```

Opens a file for reading only.

```
jobfile.open(File.WriteOnly|File.Translate);
```

Opens a file for writing and enables line break translation.

close()

Closes the file.

remove()

Deletes the file if possible; otherwise throws an exception.

seek(pos : Number) : Boolean

Sets file current position to "pos". Returns true on success, or false if an error occurred.

pos() : Number

Returns the current position in the file. This is the position that data is written to or read from.

Member Text I/O functions

readChar() : Number

Reads one character from the file if possible; otherwise throws an exception. The character is represented as a 16-bit Unicode code point.

read() : String

Returns the entire content of the file as a string if it can be read; otherwise throws an exception.

readLine() : String

Reads one line from file if possible; otherwise throws an exception. Retains any trailing whitespace.

readLines() : String[ ]

Returns the contents of the file as an Array of strings, one for each line. Linebreaks are stripped from the strings. If the file could not be read, an exception is thrown.

writeChar(char : Number)

Writes one character to the file if possible; otherwise throws an exception. The character is represented as a 16-bit Unicode code point.

write(data : String)

Writes the String data to the file if possible; otherwise throws an exception.

writeLine(data : String)

Writes the line data to the file and adds a linebreak. If the file could not be written error is returned.

```
//example
//create file to write character based data to
var path = job.createPathWithName("bytes.txt");
var f = new File(path);

//open file for writing
f.open(File.WriteOnly);
//write a character
f.writeChar(102);
//write data
f.write("this is a string \n");
//write a line
f.writeLine("This is a line");
//close the file
f.close();
//open the file for reading
f.open(File.ReadOnly);
//read a character
job.log(1, f.readChar());
//read lines
var lines = f.readLines();
for(i = 0; i < lines.length; i++){
    job.log(1, lines[i]);
}//"this is a string", "This is a line", ""
f.close();
//reopen the file
f.open(File.ReadOnly);
var str = f.read();
var strparts = str.split(new RegExp("\n\r*"));
for(i = 0; i < strparts.length; i++){
    job.log(1, strparts[i]);
}//"fthis is a string", "This is a line", ""
```

Member Binary I/O functions

The functions presented in this section ignore the file's codec (see text encoding). Never mix these binary I/O function with any of the text-oriented I/O functions on the same file. Violating this rule has unpredictable results.

readByte() : Number

Reads one byte from the file if possible; otherwise throws an exception.

writeByte(byte : Number)

Writes one byte to the file if possible; otherwise throws an exception.

```
//create a new file to write bytes to
var path = job.createPathWithName("bytes.txt");
var f = new File(path);

//open the file in WriteOnly mode
f.open(File.WriteOnly);
f.writeByte(78);
f.writeByte(98);
f.close();
//close the file, reopen it in ReadOnly mode
f.open(File.ReadOnly);
job.log(1, f.readByte()); //78
job.log(1, f.readByte()); //98
f.close(); //close the file
```

Note: A file needs to be opened with the correct AccessMode to use the member read and write functions.

5.3.4. Dir class

The Dir class provides access to file system directory structures and their contents in a platform-independent way. It provides a means of listing directory content, creating filenames with proper path separators, etc.

Enum FilterSpec

This enum describes the filtering options available to Dir for entryList(). The filter value is specified by OR-ing together values from the following list.

FilterSpec	Description
Dir.Dirs	List directories only
Dir.Files	List files only
Dir.Drives	List disk drives only
Dir.NoSymLinks	Do not list symbolic links
Dir.All	List directories, files, drives and symlinks (this does not list broken symlinks unless you specify System)
Dir.TypeMask	A mask for the the Dirs, Files, Drives and NoSymLinks flags

FilterSpec	Description
Dir.Readable	List files for which the application has read access
Dir.Writable	List files for which the application has write access
Dir.Executable	List files for which the application has execute access; executables must be combined with Dirs or Files
Dir.RWEMask	A mask for the Readable, Writable and Executable flags
Dir.Modified	Only list files that have been modified
Dir.Hidden	List hidden files
Dir.System	List system files
Dir.AccessMask	A mask for the Readable, Writable, Executable Modified, Hidden and System flags
Dir.NoDotAndDotDot	Do not list the special entries "." and ".."

Enum SortSpec

This enum describes the sort options available to Dir for entryList(). The sort value is specified by OR-ing together values from the following list.

SortSpec	Description
Dir.Name	Sort by name
Dir.Time	Sort by time (modification time)
Dir.Size	Sort by file size (descending)
Dir.Unsorted	Do not sort
Dir.SortByMask	A mask for Name, Time and Size
Dir.DirsFirst	Put the directories first, then the files
Dir.Reversed	Reverse the sort order
Dir.IgnoreCase	Sort case-insensitively

Static properties

current : String

The current directory.

home : String

The home directory.

root : String

The root directory.

drives : String[ ]

An Array of strings containing the drive names (c:, d:, etc).

Static functions

The Utility module uses "/" as a directory separator throughout (although it understands the conventions of the platform it is used on). If you are working with paths, use "/" within your code, and use convertSeparators() when you want to display a path to the user.

cleanDirPath(filePath : String) : String

Removes all multiple directory separators "/" and resolves any "."s or ".."s found in the path filePath.

```
job.log(1, Dir.cleanDirPath("test/../test3///../test4//test5/../."));
//"test4"
```

convertSeparators(pathName : String) : String

Returns pathName with the "/" separators converted to separators that are appropriate for the underlying operating system.

On Windows, convertSeparators("c:/winnt/system32") returns "c:\winnt\system32".

On Mac the returned string is the same as the argument.

Checking for arrival

This function checks if a folder and all of its files have "arrived". A folder has arrived if the number of files and subfolders in the folder (recursively) has not changed during the stability period, and if all of these items have arrived individually. See also *Checking for arrival* on page 111 in the File class documentation.

hasArrived(filePath : String, stabilitySeconds : Number) : Boolean [static]

hasArrived(stabilitySeconds : Number) : Boolean

Returns true if it can be assumed that the folder and its contents have arrived according to the heuristic describe above; otherwise returns false. The stability period is specified in seconds (as a floating point number).

The length of the required stability period depends on the nature and size of the file, on the process writing the file and on the performance and workload of the computer system(s) involved. In most cases a stability period of 1 second is both sufficient and acceptable (in terms of delay). When the file is produced by a slow process or transferred across a slow network, a much longer stability period may be required.

A hasArrived() function often returns only after a delay of at least the stability period. However if it can be determined right away that the file has not yet arrived, the function returns immediately. Thus it is never acceptable to invoke a hasArrived() function in a tight loop.

The following table describes the two most important use cases for dealing with an external application through the hasArrived() functions. In fact, the guidance presented in the table does not depend on the employed arrival/ready detection mechanism.

Model	Description	Implementation
Synchronous	If the external process is synchronous, resource-intensive and hosted on the same computer as Switch, the jobArrived entry point should block until the process is finished (so that Switch can correctly manage the number of external processes running in parallel)	jobArrived entry point contains a waiting loop that repeatedly calls hasArrived() followed by Environment.sleep() to introduce additional delay
Asynchronous	If the external process is asynchronous (i.e. you want to feed it the next job even if the previous job hasn't been processed), if it depends on user interaction or if it potentially takes a long time without using a lot of resources on the local computer, the jobArrived entry point should return immediately and leave the follow-up to the timerFired entry point	timerFired entry point simply calls hasArrived() once, since it is automatically executed repeatedly with a predefined interval

Constructor

Dir(path : String)

Creates a directory object for path path. If path is empty, the current directory is used.

```
var dirpath = job.createPathWithName("sampledir", true);
var dir = new Dir(dirpath);
```

Properties

name : String

Contains the name of the directory; this is not the same as the path, e.g. a directory with the name "mail", might have the path "c:/spool/mail".

path : String

Contains the path, this may contain symbolic links, but never contains redundant ".", "..", or multiple separators.

absPath : String

Contains the absolute path (a path that starts with "/" or with a drive specification), which may contain symbolic links, but never contains redundant ".", "..", or multiple separators.

canonicalPath : String

Contains the canonical path, i.e. a path without symbolic links or redundant "." or ".." elements.

readable : Boolean

True if the directory is readable; otherwise false.

exists : Boolean

True if the directory exists; otherwise false.

Member functions

filePath(fileName : String) : String

Returns the path name of the file fileName in the directory.

absFilePath(fileName : String) : String

Returns the absolute path name of the file fileName in the directory.

e.g.: "<dirpath>/SampleFile.txt", note that 'SampleFile.txt' doesn't need to exist.

cd(dirName : String)

Changes the Dir's directory to dirName if possible; otherwise throws an exception.

cdUp()

Changes directory by moving one directory up from the Dir's current directory if possible; otherwise throws an exception.

entryList(filter : String, filterSpec : Number, sortSpec : Number) : String[ ]

Returns a list of the names of all the files and directories in the directory, ordered in accordance with sortSpec (see enum SortSpec) and filtered in accordance with filterSpec (see enum FilterSpec).

For example:

```
var profiles = profilesDir.entryList("*.icc", Dir.Files, Dir.Name);
```

Gets a list of ICC profiles in the specified directory, sorted by filename.

```
var writableProfiles = profilesDir.entryList("*.icc", Dir.Files|Dir.Writable,
 Dir.Name);
```

Same as above but the list is restricted to ICC profiles for which the application has write access.

mkdir(dirName : String)

Creates the subdirectory dirName if possible; otherwise throws an exception.

mkdir()

Creates this directory if possible; otherwise throws an exception. Can be used if the directory doesn't already exist.

mkdirs(dirName : String)

Creates the directory tree dirName if possible; otherwise throws an exception.

mkdirs()

Creates this directory tree if possible; otherwise throws an exception. Can be used if the directory tree doesn't already exist.

rmdir(dirName : String)

Deletes the directory dirName if possible; otherwise throws an exception.

rmdir()

Deletes this directory if possible; otherwise throws an exception. Can be used if the directory exists and doesn't contain any files or folders.

rmdirs(dirName : String)

Deletes the directory structure dirName if possible; otherwise throws an exception.

rmdirs()

Deletes this directory structure if possible; otherwise throws an exception. Can be used if the directory structure exists.

fileExists(fileName : String) : Boolean

Returns true if the file fileName exists; otherwise returns false.

setCurrent()

Sets the application's current working directory to this directory if possible; otherwise throws an exception.

```
//Dir use case
//create dirtree
dirPath = job.createPathWithName("root", true);
var dir = new Dir(dirPath);
dir.mkdirs("child1/child11");
dir.mkdirs("child1/child12");
dir.mkdir("child2");
dir.mkdirs("child3/child31");
dir.mkdirs("child3/child32");
dir.mkdir("child4");

job.log(1, "before cd: " + dir.name); //"root"
dir.cd("child1/child11");
job.log(1, "after cd: " + dir.name); //"child11
dir.cdUp();
dir.cdUp(); //same as 'dir.cd(../..)';

//remove child3
dir.rmdirs("child3");

//remove child2
dir.rmdir("child2");

//list folders in root
var list = dir.entryList("*", Dir.Dirs,Dir.Name);
for(i = 0; i < list.length; i++){
    job.log(1, list[i]);
}
//".", "..", "child1", "child4"

//change current directory
dir.setCurrent();
job.log(1, Dir.current);
//".../root"
```

5.3.5. Process class

Note: If you're using a system with '\' file separators, you'll need to use an extra '\' as an escape character for the command names (if there are any separators).

The Process class is used to start external programs and to communicate with them. It can start programs in one of three modes, as described in the following table:

Mode	Function	Comments
Synchronous	Process.execute()	The calling script blocks until the external program has finished; stdout/stderr are collected in the corresponding properties
Asynchronous	new Process().start()	The calling script continues in parallel with the external program and can communicate with its stdin and stdout/stderr through the corresponding write...() and read...() functions
Detached	Process.startDetached()	The external program is started in a separate process; the calling script continues in parallel with the external program but it can NOT communicate with its stdin and stdout/stderr

Static properties

stdout : String

Contains the data sent to stdout during the last call to Process.execute(). This property is read-only.

stderr : String

Contains the data sent to stderr during the last call to Process.execute(). This property is read-only.

Static functions

execute(command : String, stdin : String) : int

Executes command synchronously and passes stdin to its standard input if specified. When the program ends its output is accessible through Process.stdout and Process.stderr. command can contain both the program and command line arguments, separated by spaces. The function returns the executed command's return code on exit.

```
Process.execute("help cd");//executes help command with cd arg
job.log(1, Process.stdout); //help output
```

execute(command : String[ ], stdin : String) : int

Same as above, except that command is an Array of strings, where the first item is the name of the program and the following items are command line arguments. This version is useful if you have arguments that contain spaces or other characters that would need to be quoted if you just passed a single string command line, since it takes care of the quoting for you.

```
var comm = new Array("help", "cd");
Process.execute(comm); //executes help command (with cd arg)
job.log(1, Process.stdout); //help output
```

startDetached (command : String) : bool

Starts command in a new process and detaches from it. command can contain both the program and command line arguments, separated by spaces. The function returns 1 on success; otherwise returns 0. The detached process will continue to live even if the calling process exits.

```
var started = Process.startDetached("help");
job.log(1, started); //"1" if command was started
```

```
var started = Process.startDetached("help");
job.log(1, Boolean(started)); //"true" if command was started
```

Constructors

Process(command : String)

Creates a Process object that will execute command. This does not start the process.

```
var p = new Process("help cd");
```

Process(command : String[ ])

Same as above, except that command is an Array of strings, where the first item is the name of the program and the following items are command line arguments. This version is useful if you have arguments that contain spaces or other characters that would need to be quoted if you just passed a single string command line, since it takes care of the quoting for you.

```
var comm = new Array("help", "cd");
var p = new Process(comm);
```

Examples of common mistakes

Example 1: A python script is used

```
Process.execute( "aws --version" ); <-- WRONG -->
```

In this example, the use of "aws" is incorrect: this is a python script, which cannot be used with the execute commmand. Alternatively, you can try prefixing the aws command with the path to 'python' or find the preferable executable that will start your command, as in the example below:

```
Process.execute( "/usr/bin/python /usr/local/bin/aws --version" ); <-- RIGHT -->
```

There will be no problem if your command starts with a binary executable. For example, the following script should work correctly:

```
Process.execute( "curl --version" ); <-- RIGHT -->
```

Example 2: Quotes are missing

```
var theCommand = "cmd /C dir c:\\folder\\inner folder";
Process.execute( theCommand ); <-- WRONG -->
```

In this example, the path is not enclosed in quotes. This is corrected in the example below.

```
var theCommand = "cmd /C dir \"c:\\folder\\inner folder\"";
```

```
Process.execute( theCommand ); <-- RIGHT -->
```

For more information, refer to the documentation about the Execute command tool in the Switch Reference Guide.

Properties

arguments : String[ ]

Contains an Array of strings where the first item is the program to execute and the following items are the command line arguments.

```
//with p initialized as above
var args = p.arguments;
job.log(1, args[1]); //"cd"
```

workingDirectory : String

Contains the working directory for the process.

running : Boolean

True if the process is currently running; otherwise false. This property is read-only.

exitStatus : Number

Contains the exitStatus of the program when it has finished. This property is read-only.

Member functions

start(env : String[ ])

Tries to run a process for the command and arguments that were specified with the argument property or that were specified in the constructor. The command is searched for in the path for executable programs; you can also use an absolute path in the command itself. If env is not specified, the process is started with the same environment as the starting process. If env is specified, then the values in the Array of strings are interpreted as environment settings of the form VARIABLE=VALUE and the process is started with these environment settings. If the program could not be started, an exception is thrown.

launch(stdin : String, env : String[ ])

Runs the process and writes the data stdin to the process's standard input. If all the data is written to standard input, standard input is closed. The command is searched for in the path for executable programs; you can also use an absolute path in the command itself. If env is unspecified, the process is started with the same environment as the starting process. If env is specified, then the values in the Array of strings are interpreted as environment settings of the form VARIABLE=VALUE and the process is started with these environment settings. If the program could not be started, an exception is thrown.

waitForStarted (seconds : Number)

Blocks until the process has started or until the specified time (in seconds) has passed. Returns true if the process was started successfully; returns false if the operation timed out or if an error occurred. The default value for the argument is 30 seconds. If the value -1 is used, the method will wait infinitely.

waitForFinished (seconds : Number)

Blocks until the process has finished or until the specified time (in seconds) has passed. Returns true if the process finished; returns false if the operation timed out or if an error

occurred. The default value for the argument is 30 seconds. If the value -1 is used the method will wait infinitely.

Note:

It is required to call waitForStarted() before calling waitForFinished().

Example:

```
theProcess.start();
if( !theProcess.waitForStarted( 60 ) ) // 1 minute to start
{
    job.fail( "Failed to start process with the arguments: %1",
 theProcess.arguments );
    return;
}
if( !theProcess.waitForFinished( 600 ) ) // 10 minutes to finish
{
    job.log( 3, "Cannot finish the process: %1", theProcess.arguments );
}
```

readStdout() : String

Reads what is currently available on the process's standard output.

readStderr() : String

Reads what is currently available on the process's standard error.

canReadLineStdout() : Boolean

Returns true if a line can be read from the process's standard output.

canReadLineStderr() : Boolean

Returns true if a line can be read from the process's standard error.

readLineStdout() : String

Reads one line of text from the process's standard output if available; otherwise returns an empty string.

readLineStderr() : String

Reads one line of text from the standard error stream of this process if available; otherwise returns an empty string.

writeToStdin(buffer : String)

Writes the data buffer to the standard input stream of this process. The process may or may not read this data.

closeStdin()

Closes the standard input stream of the process.

tryTerminate()

Asks the process to terminate. Processes can ignore this if they wish. If you want to be certain that the process really terminates, you can use kill() instead.

kill()

Terminates the process. This is not a safe way to end a process since the process will not be able to do any cleanup. tryTerminate() is safer, but processes can ignore tryTerminate().

5.3.5.1. Text encoding

JavaScript strings represent Unicode characters (internally encoded in UTF-16). Care must be taken to preserve Unicode compliance when communicating with external programs - wherever possible.

Command line

The Process class passes the command line to the operating system (and thus to the external program) in full Unicode compliance:

- On Mac OS the command line is UTF-8 encoded, which is the default filename encoding and which turns out to be compatible with virtually all command-line tools.
- On Windows the command line is passed using a "wide" (16-bit-character) system call; external programs should retrieve command line arguments using the "wide" (16-bit-character) version of the Windows-specific "GetCommandLine" function rather than the argv[] argument in the main C function (which supports only characters that can be encoded in the current ANSI code page).

Console text streams

The Process class uses the following scheme for communicating with the stdin and stderr/stdout text streams:

- Text is written to stdin using UTF-8 encoding.
- Text is read from stdout/stderr using UTF-8 encoding, unless the text doesn't conform to UTF-8 syntax, in which case it is interpreted using the current ANSI code page (on Windows) / MacRoman or Latin1 encoding (on Mac OS).

While this behavior is not perfect in all cases, note that:

- UTF-8 is upwards compatible with 7-bit ASCII.
- If a byte stream conforms to UTF-8 syntax, there is a very high probability that it represents a text string in UTF-8 encoding (as opposed to a text string in one of the common 8-bit legacy encodings).

5.4. XML module

5.4.1. Node class

Node is the abstract base class for the Document, Element, Text, Comment and Attribute classes. A Node object represents a node in the XML tree of one of those types.

Note: If it is known which subtype a node represents, the node can be used as an instance of that subtype.

5.4.1.1. Accessing node properties

getNodeType() : String

Returns the node type ("Document", "Element", "Text", "Comment" or "Attr").

```
//...
//doc = Document
var textnode = doc.createText("this is text");
job.log(1, textnode.getNodeType()); //"Text"
```

getOwnerDocument() : Document

Returns the document object in which this node resides.

getParentNode() : Node

Returns this node's parent node or null if this node is a document or if this node is not currently part of a node hierarchy.

5.4.1.2. Evaluating XPath expressions

The functions presented here evaluate an XPath 1.0 expression in the context of the node being queried.

Namespaces

Namespace prefixes in the query expression are resolved into namespace URIs using the prefix map provided to the query functions as a second argument. If the map argument is omitted or null, the default prefix map is used for resolving prefixes.

Result data types

The value of an XPath 1.0 expression can have several data types (a node set with several elements, the text value of an attribute, a number returned by the count() function, the Boolean result of a comparison). The value is converted to the function's result data type using XPath 1.0 semantics – which may very well differ from the conversion semantics in the scripting language. For example the XPath 1.0 conversion from string to Boolean returns true for all non-empty strings (including the string "false").

Functions

evalToNodes(xpath : String, prefix-map : Map) : NodeList

Evaluates an XPath 1.0 expression in the node's context. If the result is a non-empty node-set, returns a list of the nodes in the set. Otherwise returns an empty list.

evalToNode(xpath : String, prefix-map : Map) : Node

Evaluates an XPath 1.0 expression in the node's context. If the result is a non-empty node-set, returns the first node in the set. Otherwise returns null.

evalToString(xpath : String, prefix-map : Map) : String

Evaluates an XPath 1.0 expression in the node's context and converts the result to a string value with the XPath 1.0 string() function.

evalToNumber(xpath : String, prefix-map : Map) : Number

Evaluates an XPath 1.0 expression in the node's context and converts the result to a numeric value with the XPath 1.0 number() function.

evalToBoolean(xpath : String, prefix-map : Map) : Boolean

Evaluates an XPath 1.0 expression in the node's context and converts the result to a Boolean value with the XPath 1.0 boolean() function.

Note: The methods to evaluate XPath expressions will throw the exception in case the node object does not represent a valid XML node. This will happen, for example, in case the Document node was created by opening a not well-formed XML file. It is recommended to use try-catch blocks when using these methods.

For the given document (fragment of *books.xml*)

```
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications
      with XML.</description>
   </book>
   <book id="bk102">
      <author>Ralls, Kim</author>
      <title>Midnight Rain</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-12-16</publish_date>
      <description>A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.</description>
   </book>
   <book id="bk103">
      <author>Corets, Eva</author>
      <title>Maeve Ascendant</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-11-17</publish_date>
      <description>After the collapse of a nanotechnology
      society in England, the young survivors lay the
      foundation for a new society.</description>
   </book>
   <book id="bk104">
      <author>Corets, Eva</author>
      <title>Oberon's Legacy</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2001-03-10</publish_date>
      <description>In post-apocalypse England, the mysterious
      agent known only as Oberon helps to create a new life
      for the inhabitants of London. Sequel to Maeve
      Ascendant.</description>
   </book>
</catalog>
```

we get the following results:

```
var doc = new Document(job.getPath()); //job = sample document
var root = doc.getDocumentElement();
var list = root.evalToNodes("book",null);

var node = root.evalToNode("book");
var exists = Boolean(root.evalToBoolean("book[@id=\"bk1090\"]"));
var price = root.evalToNumber("book[@id=\"bk104\"]/price");
```

```
var fbgenre = root.evalToString("book/genre");
var genre = root.evalToString("book[@id=\"bk103\"]/genre");

//number of 'book' tags = 4
job.log(1,"number of 'book' tags = " + list.length);
//basename: book
job.log(1,"basename: " + node.getBaseName());
//id: bk101
job.log(1,"id: " + node.getAttributeValue("id"));
//parent basename: catalog
job.log(1,"parent basename: " + node.getParentNode().getBaseName());
//bk103 genre: Fantasy
job.log(1,"bk103 genre: " + genre);
//first book genre: Computer
job.log(1,"first book genre: " + fbgenre);
//bk104 price: 5.95
job.log(1,"bk104 price: " + price);
//exists bk1090: false
job.log(1,"exists bk1090: " + exists);
```

5.4.2. Document class

A Document object represents the root node of a well-formed XML document (refer to the XML 1.0 specification). The root node is the "invisible" node that contains the document element and any top-level comments.

Document inherits from the Node class, so all functions defined for Node can be used with Document. Document is the only class in the XML module with a constructor; all other classes are instantiated from within a document.

Note: You cannot attach nodes gotten from one document to other documents. Use Node.cloneNode (described further) to make a copy of the node and attach it to a different document.

Constructing, loading and saving

Document( )

Constructs a new empty XML document (in memory).

```
var doc = new Document(); // creates a new document
```

Document(path : String, discardBlankNodes: Boolean)

Constructs a new XML document (in memory) and parses the contents of the file at the specified absolute path into the document's node tree. If the specified file is not well-formed XML, this function logs an error message and it constructs a partial or empty document.

If discardBlankNodes is true, text nodes that contain only white space (such as line breaks) are discarded while parsing the input stream. If discardBlankNodes is false, null or missing, white-space text nodes are left in place.

```
var xmlPath = ... //path to xml document
var doc = new Document(xmlPath);
```

isWellFormed( ): Boolean

Returns true if the document represents well-formed XML.

save(path : String)

Serializes the XML document into a file at the specified absolute path.

```
var xmlPath = ... //path to save the document to
doc.save(xmlPath); //saves the document to the specified path
```

Performing transformations

Refer to the XSLT 1.0 specification and to widely available literature about XSLT for more information.

createTransform(stylesheet : Document) : Document

Returns a newly created document that contains the transformation of this document according to the rules of the specified stylesheet. The stylesheet argument must be a valid XML document which is assumed to be an XSLT 1.0 stylesheet. The result is assumed to be a valid XML document.

transform(in-path : String, stylesheet-path : String, out-path : String) [static]

Transforms the XML file at "in-path" using the XSLT 1.0 stylesheet at "stylesheet-path" and saves the result in a file at "out-path". The result is not assumed to be an XML document.

Child nodes

A Document can directly contain any number of Comment nodes (including the XML declaration, which if present is always the first node of the document) and a single Element node (which is called the document element). A Document can not contain any Text nodes and it can not contain two or more Element nodes. The functions described in this section log an error if these restrictions are violated.

hasChildNodes( ) : Boolean

Returns true if this element has any child nodes and false if not.

getChildNodes( ) : NodeList

Returns the list of child nodes of this element, in document order, or an empty list if the element has no child nodes.

getFirstChild( ) : Node

Returns the first child node of this element, or undefined if there is none.

getLastChild( ) : Node

Returns the last child node of this element, or undefined if there is none.

appendChild(new-child : Node)

Appends a new child node to the list of children for this element.

insertBefore(new-child : Node, ref-child : Node)

Inserts a new child node before the specified reference node, which must be in the list of children of this element.

removeChild(oldChild : Node) : Node

Removes the specified node from the list of children of this element and returns the removed node.

replaceChild(newChild : Node, oldChild : Node) : Node

Replaces the specified child of this element with another node and returns the removed node.

```
//example, !doesn't work yet!
//create an empty document
var doc = new Document();
//create some comments
var author = doc.createComment("@author John Doe");
var date = doc.createComment("@date 2012-1-1");
var description = doc.createComment("this is a sample doc");
var description2 = doc.createComment("this is a sample comment");
//add comments to doc
//add author
doc.appendChild(author);
//insert date before author
doc.insertBefore(date, author);

//create 1 element, make it the docelement
var root = doc.createElement("root");
doc.setDocumentElement(root);

//add description
doc.appendChild(description);
//replace description by description2
doc.replaceChild(description2, description);

//check if document has child nodes
job.log(1, Boolean(doc.hasChildNodes()));//"true"

//save document
doc.save(docPath);
```

Results in (at docPath):

```
<?xml version="1.0" encoding="UTF-8"?>
<!--@date 2012-1-1-->
<!--@author John Doe-->
<root/>
<!--this is a sample comment-->
```

Directly accessing the document element

getDocumentElement( ) : Element

Returns the document element, i.e. the single element directly under the root node (which is represented by the Document object).

setDocumentElement(doc-element : Element)

Sets the document element, i.e. the single element directly under the root node (which is represented by the Document object). Specifically, if before this function is called the document node contained:

- No children at all, this function adds an XML declaration comment and the specified document element.

  ```
  //create new document
  var doc = new Document();
  var root = doc.createElement("root", null);
  doc.setDocumentElement(root);
  //results in a document with XML declaration and
  //document element root
  ```

- One or more children but none of them is an element, this function appends the specified document element at the end.
- A document element, this function replaces the document element by the specified document element.

Creating new objects

createEmptyMap( ) : Map

Returns a new empty prefix map object.

createDefaultMap( ) : Map

Returns a new prefix map object that already contains all mappings that occur in the document.

createElement(qualified-name : String, prefix-map : Map) : Element

Returns a newly created element node with the specified name, owned by this document. If the element name is qualified its namespace prefix must occur in the map provided in the second argument. If the name is not qualified the second argument may be null.

createText(value : String) : Text

Returns a newly created text node with the specified text content, owned by this document.

createComment(value : String) : Comment

Returns a newly created comment node with the specified content (i.e. all the characters that appear between the '<!--' and '-->'), owned by this document.

cloneNode(node : Node, deep : Boolean) : Node

Returns a clone of the specified node that is owned by this document. If deep is true, all descendant nodes are cloned as well.

Examples: see other examples throughout the whole *XML module*.

5.4.3. Element class

An Element object represents an XML element node. An element may have other Element, Comment and Text nodes as children. Furthermore an element may have attributes (these are not considered to be children).

Element inherits from the Node class, so all functions defined for Node can be used with Element.

Name

getQualifiedName( ) : String

Returns the qualified element name, i.e. including the namespace prefix if there is one.

e.g.: "prfx:node"

getBaseName( ) : String

Returns the element name without the namespace prefix.

e.g.: "node"

getPrefix( ) : String

Returns the namespace prefix or the empty string if there is none.

e.g.: "prfx"

getNamespaceURI( ) : String

Returns the namespace URI corresponding to namespace prefix or the default namespace URI for the document if there is no prefix or the empty string if there is no prefix and no default namespace.

e.g.: "prefix.txt"

Child nodes

hasChildNodes( ) : Boolean

Returns true if this element has any child nodes, and false if not.

getChildNodes( ) : NodeList

Returns the list of child nodes of this element, in document order or an empty list if the element has no child nodes.

getFirstChild( ) : Node

Returns the first child node of this element or undefined if there is none.

getLastChild( ) : Node

Returns the last child node of this element or undefined if there is none.

appendChild(new-child : Node)

Appends a new child node to the list of children for this element.

insertBefore(new-child : Node, ref-child : Node)

Inserts a new child node before the specified reference node, which must be in the list of children of this element.

removeChild(oldChild : Node) : Node

Removes the specified node from the list of children of this element, and returns the removed node.

replaceChild(newChild : Node, oldChild : Node) : Node

Replaces the specified child of this element with another node, and returns the removed node.

```
//example

//create a new empty document
var doc = new Document();
var root = doc.createElement("root");
doc.setDocumentElement(root);

//create a comment
var comment = doc.createComment("This is a comment");
root.appendChild(comment);

//create another element
var content = doc.createElement("content",null);
root.appendChild(content);

//create text
var text = doc.createText("this is text");
content.appendChild(text);
job.log(1, content.hasChildNodes()?"true":"false"); //"true"

//create element
var title = doc.createElement("title",null);
root.insertBefore(title,comment);

//create another textnode
var text2 = doc.createText("this is the content");

//replace the old text in the content node
content.replaceChild(text2, text);

//remove the first child from root
var first = root.getFirstChild();
root.removeChild(first);
```

```
//add attribute (which is also a node)
root.addAttribute("id", null, "123");
//get nodelist
var list = root.getChildNodes();
job.log(1, list.length); //2
doc.save(docPath); //docPath = path to save document to
```

Result (at docPath):

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <!--This is a comment-->
  <content>this is the content</content>
</root>
```

Attributes

getAttributes( ) : AttrList

Returns the list of attributes of this element, in arbitrary order. If there are no attributes, returns an empty list.

getAttributeValue(qualified-name : String, prefix-map : Map) : String

Returns the value for the element's attribute with the specified qualified name or null if there is no such attribute. If the qualified name includes a prefix it is resolved using the map in the second argument, otherwise the second argument may be null.

addAttribute(qualified-name : String, prefix-map : Map, value : String)

Adds an attribute to the element with the specified qualified name and value or replaces an existing attribute with the same name. If the qualified name includes a prefix it is resolved using the map in the second argument, otherwise the second argument may be null.

```
//example

//create an empty document
var doc = new Document();

//create a new element and set this as the document element
var root = doc.createElement("root");
doc.setDocumentElement(root);

//create a new prefix-map
var map = doc.createEmptyMap();
map.put('prfx','prefix');

//add attributes
root.addAttribute("prfx:id",map,"1234");
root.addAttribute("id",null,"597");

//get attribute value
var attrvalue = root.getAttributeValue("prfx:id",map);
job.log(1,attrvalue); //"1234"

//get attributes
var list = root.getAttributes();
job.log(1, list.length); //2
```

5.4.4. Text class

A Text object represents an XML text node. A text node can only appear in a document as a child of an element.

Text inherits from the Node class, so all functions defined for Node can be used with Text.

Text content

getValue( ) : String

Returns the text content of this node.

```
var doc = new Document();
var text = doc.createText("this is text");
job.log(1, text.getValue()); //"this is text"
```

5.4.5. Comment class

A Comment object represents an XML comment node.

Comment inherits from the Node class, so all functions defined for Node can be used with Comment.

Comment content

getValue( ) : String

Returns the content of the comment represented by this node, that is, all the characters that appear between the '<!--' and '-->'.

```
var doc = new Document();
var comment = doc.createComment("this is a comment");
job.log(1, comment.getValue()); //"this is a comment"
```

5.4.6. Attr class

An Attr object represents an attribute of an XML element.

Attr inherits from the Node class, so all functions defined for Node can be used with Attr.

Name

getQualifiedName( ) : String

Returns the qualified attribute name, that is, including the namespace prefix if there is one.

e.g.: "prfx:id"

getBaseName( ) : String

Returns the attribute name without the namespace prefix.

e.g.: "id"

getPrefix( ) : String

Returns the namespace prefix or the empty string if there is none.

e.g.: "prfx"

getNamespaceURI( ) : String

Returns the namespace URI corresponding to namespace prefix or the namespace URI for the element owning this attribute if there is no prefix or the empty string if there is no prefix and no element namespace.

e.g.: "prefix.txt"

For an example, see 'List classes'.

Value

getValue( ) : String

Returns the value of this attribute.

5.4.7. List classes: NodeList, AttrList

Due to implementation limitations Switch is unable to return an array of Node and Attr instances. Instead, it returns a helper object of the corresponding list class, that is, NodeList and AttrList, respectively.

These helper classes offer the following functions to access the items in the list.

getCount( ) : Number

Returns the number of items in the list (may be zero).

getItem(index : Number) : Node or Attr

Returns the item in the list at the specified (zero-based) index. If the index is out of range, the function returns null.

length : Number

This is a read-only property (rather than a function) that contains the value returned by getCount().

at(index : Number) : Node or Attr

Returns the same value as getItem(index).

```
//example

//create document with document element root
var doc = new Document();
var root = doc.createElement("head");
doc.setDocumentElement(root);

//create map
var map = doc.createEmptyMap();
map.put('prfx','prefix.txt');
map.put('o','other.txt');

//add attributes
root.addAttribute("prfx:id",map,"1234");
root.addAttribute("o:title",map,"597");
var attrlist = root.getAttributes();
for( var i = 0; i < attrlist.getCount();i++){//getCount() == length
    var attr = attrlist.at(i); //the same as getItem
    var qname = attr.getQualifiedName();
    var bname = attr.getBaseName();
    var prefix = attr.getPrefix();
    var ns = attr.getNamespaceURI();
    var value = attr.getValue();
    job.log(1,"qname: " + qname);
    job.log(1,"bname: " + bname);
    job.log(1,"prefix: " + prefix);
    job.log(1,"namespace: " + ns);
```

```
    job.log(1,"value: " + value);
}
```

5.5. Network module

5.5.1. HTTP class

Background

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP functions as a request-response protocol in the client-server computing model. A web browser, for example, may be the client and an application running on a computer hosting a web site may be the server. The client submits an HTTP request message to the server. The server, which provides resources such as HTML files and other content, or performs other functions on behalf of the client, returns a response message to the client. The response contains completion status information about the request and may also contain requested content in its message body. HTTP resources are identified and located on the network by Uniform Resource Locators (URLs) using the http or https URI schemes.

For more information, see *http://www.w3.org/Protocols/*.

Overview

The HTTP class implements a standard HTTP client as defined in RFCs 1945 and 2616. It can be used for communication with an HTTP server and it supports the HTTP as well as the HTTPS protocols.

HTTP communication involves the client sending requests to the server and the server sending a response back. The HTTP class allows using different request methods, defining authentication and defining additional parameters when sending requests, and it allows capturing the response into a file.

The current implementation supports the HTTP request methods PUT, HEAD, GET, POST and DELETE. The methods OPTIONS, TRACE, CONNECT and PATCH are not available, but they are also less important.

The HTTP class supports the HTTP Basic authentication scheme through the user and password properties. Other authentication schemes (Digest, NTLM, Negotiate, Proprietary, OAuth) can be implemented by using the authScheme property.

The addParameter() method can be used to add extra parameters to the request and attaching file is done with the setAttachedFile() method.

The server response is written into a file when the localFilePath property is set. This is whatever the server has responded; it can be the HTML source of the requested URL, JSON or XML data returned by a REST API call, or a file or document that was downloaded. If the server responded with an error, the file will not be created or it will be empty. If this property is not specified, the server response can be obtained as byte array using the getServerResponse() method.

The HTTP class also defines a number of constants that are available in the autocomplete list.

Note: By default, the HTTP class uses the proxy settings from the Switch Server Preferences. They can be overridden as required. The HTTP request also uses the proxy settings from the Switch Server Preferences, but these can NOT be overridden.

Constructing

HTTP( )

Constructs a new HTTP object. The connection type (HTTP or HTTPS) is automatically detected based on the URL scheme.

```
var theHTTP = new HTTP();
```

Note: In previous Switch versions this constructor had a connectionType argument that could be “NoSSL” or “SSL” for the HTTP or HTTPS protocol respectively. It was also possible to use the predefined constants "HTTP.NoSSL" and "HTTP.SSL". The connectionType argument is obsolete as of version 17 update 1, however it may still be used in the examples below.

Properties

url : String

Contains the URL to be used for request. The URL must be URI-encoded. The method HTTP.encodeURI() can be used to encode the URL.

```
theHTTP.url = “https://api.dropbox.com/1/fileops/delete”;
```

The URL must specify the protocol to be used (http:// or https://).

enableMime : Boolean

This property specifies if MIME encoding should be used (as specified in RFC1867).

Note: This property is relevant only when POST request is used.

authScheme : String

This property will tell the class which type of authorization to perform when the *user* and *password* properties are set.

- This property should be set to "NoneAuth" (predefined constant HTTP.NoneAuth) when no authentication is to be performed.
- By default, this property is "BasicAuth" (HTTP.BasicAuth), and if the *user* and *password* properties are set, the class will attempt basic authentication.
- If *authScheme* is set to "DigestAuth" (HTTP.DigestAuth), "NTLMAuth" (HTTP.NTLMAuth) or "NegotiateAuth" (HTTP.NegotiateAuth), digest, NTLM or Negotiate authentication will be attempted instead.
- If *authScheme* is set to "ProprietaryAuth" (HTTP.ProprietaryAuth), the authorization token must be supplied through the authorization property.
- If *authScheme* is set to "OauthAuth" (HTTP.OauthAuth), the authorization string must be supplied through the authorization property.

```
theHTTP.authScheme = HTTP.OauthAuth;
```

authorization : String

If the *authorization* property contains a non-empty string, an Authorization HTTP request header is added to the request. This header conveys authorization information to the server. A common use for this property is to specify the OAuth authorization string.

The *authScheme* property defines the authentication scheme used. In the case of HTTP Basic Authentication (default), the *user* and *password* are automatically Base64 encoded by Switch, and the result in the form "Basic [encoded-user-password]" is used for authorization.

```
theHTTP.authorization = "authorization string";
```

user : String

This property contains a user name if authentication is to be used.

- If *authScheme* is set to HTTP Basic Authentication, the *user* and *password* are Base64 encoded, and the result in the form "Basic [encoded-user-password]" is used for authorization.
- If *authScheme* is set to HTTP Digest Authentication, the *user* and *password* properties are used to respond to the HTTP Digest Authentication challenge from the server.
- If *authScheme* is set to NTLM, NTLM authentication will be attempted.

```
theHTTP.user = "User1";
```

password : String

This property contains a password if authentication is to be used.

```
theHTTP.password = "password1";
```

localFilePath : String

The path to a local file that the server response is written to. This can be the file or document being downloaded, JSON or XML data or whatever the server has responded with. If the server responded with an error, the file will not be created or it will be empty.

If this property is not specified, the server response can be obtained as byte array using the getServerResponse() method.

```
theHTTP.localFilePath = job.createPathWithExtension( "pdf", false );
```

Note: This property is not relevant for HEAD requests, because in that case the server response is always empty.

timeOut : Number

If the *timeOut* property is set to "0", all operations will run uninterrupted until successful completion or until an error condition is encountered.

If *timeOut* is set to a positive value, the class will wait for the operation to complete and if *timeOut* expires before the operation is complete, the class fails with an error.

The default value for the *timeOut* property is "60" (seconds).

useProxy : Boolean

This property specifies if a proxy server should be used.

proxyServer : Boolean

If a proxy *proxyServer* is given, the HTTP request is sent to the proxy instead of the server specified in the *url* property.

If the proxyServer property is set to a Domain Name, a DNS request is initiated and upon successful termination of the request, the corresponding address is used for sending the HTTP request. If the search is not successful, an error is returned.

proxyPort : Number

This property contains the TCP port for the proxy server (default: 80).

proxyUser : String

This property contains a user name, if authentication is to be used for the proxy.

- If *proxyAuthScheme* is set to Basic Authentication, the *proxyUser* and *proxyPassword* are Base64 encoded and the proxy authentication token will be generated in the form "Basic [encoded-user-password]".
- If *proxyAuthScheme* is set to Digest Authentication, the proxyUser and proxyPassword properties are used to respond to the Digest Authentication challenge from the server.
- If *proxyAuthScheme* is set to NTLM Authentication, the proxyUser and proxyPassword properties are used to authenticate through NTLM negotiation.

proxyPassword : String

This property contains a password if authentication is to be used for the proxy.

proxyAuthScheme : String

This property is used to tell the class which type of authorization to perform when connecting to the proxy. This is only used when the *proxyUser* and *proxyPassword* properties are set.

- *proxyAuthScheme* should be set to "NoneAuth" (predefined constant HTTP.NoneAuth) when no authentication is expected.
- By default, *proxyAuthScheme* is "BasicAuth" (HTTP.BasicAuth), and if the *proxyUser* and *proxyPassword* properties are set, the component will attempt basic authentication.
- If *proxyAuthScheme* is set to "DigestAuth" (HTTP.DigestAuth), digest authentication will be attempted instead.
- If *proxyAuthScheme* is set to "ProprietaryAuth" (HTTP.ProprietaryAuth), the authorization token will not be generated by the class.
- If *proxyAuthScheme* is set to "NTLMAuth" (HTTP.NTLMAuth), NTLM authentication will be used.

HTTPVersion : String

This property contains the HTTP version from the first line of the last server response. The HTTP protocol specifies the structure of this line as: [HTTP version] [Status Code] [Description].

This property is read-only and becomes available after the HTTP request is finished.

statusCode : Number

This property contains the Status Code from the first line of the last server response. The HTTP protocol specifies the structure of this line as: [HTTP version] [Status Code] [Description]. For the list of HTTP status codes, see *http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html*.

This property is read-only and becomes available after the HTTP request is finished.

statusDescription : String

This property contains the Description from the first line of the last server response. The HTTP protocol specifies the structure of this line as: [HTTP version] [Status Code] [Description].

This property is read-only and becomes available after the HTTP request is finished.

lastError : String

This property contains the last error string.

This property is read-only and becomes available after the HTTP request is finished.

finishedStatus : String

This property contains the request finished status. The possible statuses are:

- "Ok" (predefined constant HTTP.Ok) - the request was executed successfully.
- "Failed" (HTTP.Failed) – the request failed to execute.
- "Interrupted" (HTTP.Interrupted) – the request was interrupted.

Note that when *finishedStatus* returns "Ok" it does not automatically mean that the request yielded the desired result. The URL provided could have been redirected to another URL. In such a case the HTTP exchange with the server was successful, but nothing happened. It is therefore advisable to also read the properties *statusCode* and *statusDescription* to get more details about the completed request. In case of failed status, the *lastError* property should provide more details about the problem.

```
if( theHTTP.finishedStatus == HTTP.Failed )
{
      job.fail( "The request failed: %1", theHTTP.lastError );
      return;
}
```

resumeEnabled : Boolean

Relevant only for GET request. If set to true, Switch will try to resume the interrupted/broken download if possible. The files containing not finished downloads are stored in the folder defined by the tempFolderPath property.

tempFolderPath : String

Relevant only for GET request. This property defines a folder where not finished downloads are stored.

By default this property is empty. In case *resumeEnabled* is set to true, this property must be set as well. It should contain a folder path pointing to an existing writable folder. Switch does not clean up this folder automatically; it removes the files containing not finished downloads after they are completed by some future request.

followRedirects : Boolean

Relevant only for HEAD and GET requests. This property determines what happens when the server issues a redirect. If this property is set to false, the HTTP call returns an error if the server responds with an "Object Moved" message (HTTP status codes in the 300 range). If this property is set to true (default), the new URL for the object is retrieved automatically.

Note: To avoid endless loops, the maximum number of redirects is set to 10. A redirection between protocols (for example HTTP to HTTPS) is not allowed and will generate an error.

Static functions

encodeURI(inURI : String) : String

This function encodes inURI by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character. It assumes that

the URI is a complete URI, so it does not encode reserved characters that have special meaning in the URI. It replaces all characters except the following: alphabetic, decimal digits, - _ . ! ~ * ' ( ) ; , / ? : @ & = + $ #

decodeURI(inURI : String) : String

This function decodes inURI by replacing each escape sequence in the encoded URI with the character that it represents.

encodeURIComponent(inURIComponent : String) : String

This function encodes inURIComponent by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character.

It replaces all characters except the following: alphabetic, decimal digits, - _ . ! ~ * ' ( )

decodeURIComponent(inURIcomponent : String) : String

This function decodes inURIcomponent by replacing each escape sequence in the encoded URI component with the character that it represents.

Member functions

setAttachedFile(filePath : String , fileVariable : String)

Sets the path to the file that is appended to the HTTP request, if the HTTP POST or PUT methods are used (through the post() or put() methods). The second argument is optional. It is taken into account only when POST request is sent and MIME encoding is enabled. It specifies the name of the HTTP form data variable that is used by the receiving HTTP server to identify the correct file part of the uploaded MIME package. If this argument is omitted, the default value "file" will be used.

```
theHTTP.setAttachedFile( job.getPath() );
```

Warning:

In case of a POST request with MIME encoding enabled, this call identifies the MIME part by adding the following MIME part header:

```
Content-Disposition: form-data; filename="name of the file that is being attached";
 name="second parameter"
```

If the file being uploaded is job.getPath(), the filename field will also have the unique ID of the job. To work around this limitation, a temporary copy of the job without the unique ID has to be made and that temporary file should be the attached file:

```
var tmpFile = job.createPathWithName(job.getName());
if(!s.copy(job.getPath(), tmpFile))
{
   job.fail( "Could not copy the job to the temporary location: %1", tmpFile );
   return;
}
theHTTP.setAttachedFile(tmpFile);
```

addParameter(key : String, value: String)

Adds parameters to the HTTP request.

- For the POST and PUT requests, the parameters are included into the HTTP request after the HTTP headers.
- For the HEAD and GET requests, the parameters are appended to the URL after '?'; when calling a REST API it is common to use URLs that look like this: http://www.xyz.com/

updatedatabase?key1=value1&key2=value2. The HTTPS protocol encrypts everything after the '?'.

- For DELETE requests, the parameters are ignored.

The method HTTP.encodeURIComponent() can be used to encode parameters. Note that when adding parameters, the key and value must always be URI-encoded, except when POST requests with MIME encoding are used (the property enableMime is true). In this case the parameters are put in the MIME data and in most cases they should not be URI-encoded by the script.

```
theHTTP.addParameter( ”root”, ”auto” );
theHTTP.addParameter( ”path”, HTTP.encodeURIComponent( job.getName() );
```

resetParameters()

Clears parameters previously added by the *addParameter()* method.

```
theHTTP.resetParameters();
```

post()

This method posts data to the HTTP server using the HTTP POST method. The data includes the file that was specified using the *setAttachedFile()* method and also parameters added using the *addParameter()* method. If the property *localFilePath* is specified, the server response is written to this file; otherwise the response can be obtained as a byte array using the *getServerResponse()* method.

If the *enableMime* property value is **true**, the "Content-Type" header is always present and set to "multipart/form-data; boundary=...". In this case using addHeader("Content-Type",...) does not change it.

If the *enableMime* property value is **false**:

- If there are no *addParameter()* calls in the script, the "Content-Type" is missing. In this case addHeader("Content-Type",...) adds the required header in the request.
- If there is an *addParameter()* call in the script, the "Content-Type" is automatically set to "application/x-www-form-urlencoded". Calling addHeader("Content-Type",...) does not change it.

```
theHTTP.post();
```

Note: This method is asynchronous. It means it does not wait till the request is finished. To wait for the request completion, a script writer has to use the *waitForFinished()* method.

put()

This method sends data to the HTTP server using the HTTP PUT method. The data includes the file that was specified using the *setAttachedFile()* method and also parameters added using the *addParameter()* method. If the property *localFilePath* is specified, the server response is written to this file; otherwise the response can be obtained as a byte array using the *getServerResponse()* method.

If there are no *addParameter()* calls in the script, the "Content-Type" is missing. In this case addHeader("Content-Type",...) adds the required header in the request.

If there is an *addParameter()* call in the script, the "Content-Type" is automatically set to "application/x-www-form-urlencoded". Calling addHeader("Content-Type",...) does not change it.

```
theHTTP.put();
```

Note: This method is asynchronous. It means it does not wait till the request is finished. To wait for the request completion, a script writer has to use the *waitForFinished()* method.

head()

This method requests the headers using the HTTP HEAD method. It can be used to get the headers that would be returned if the specified resource would be requested with HTTP GET method. Such a request can be done before deciding to download a large resource to save bandwidth, for example. The server response is always empty.

```
theHTTP.head();
```

Note: This method is asynchronous, meaning that it does not wait till the request is finished. To wait for the request completion, use the *waitForFinished()* method.

get()

This method fetches the document using the HTTP GET method. If the property *localFilePath* is specified, the server response is written to this file; otherwise the response can be obtained as a byte array using the *getServerResponse()* method.

```
theHTTP.get();
```

Note: This method is asynchronous. It means it does not wait till the request is finished. To wait for the request completion, a script writer has to use the *waitForFinished()* method.

del()

This method is used to delete an object at the URL specified by using the HTTP DELETE method. If the property *localFilePath* is specified, the server response is written to this file; otherwise the response can be obtained as byte array using getServerResponse() method.

```
theHTTP.del();
```

Note: This method is asynchronous. It means it does not wait till the request is finished. To wait for the request completion, a script writer has to use the *waitForFinished()* method.

progress() : Number

Returns the progress (in percent) of the current request. This method should be used in combination with the *waitForFinished()* method.

waitForFinished(delay : Number) : Boolean

This method is used to wait for the request completion. It is used after calling the *post()*, *put()*, *head()*, *get()* or *del()* method.

The delay argument specifies the timeout in seconds that is the maximum time the *waitForFinished()* may block the script execution. Return value 'true' means the request has

finished, otherwise 'false' is returned. Note that the method returns 'true' immediately after request has finished without waiting for the timeout to happen.

This method is convenient in combination with the *progress()* method to inform the user about the request progress.

```
job.log( 4, "Starting request...", 100 ); // max is 100%
while( !theHTTP.waitForFinished( 1 ) )
{
  job.log( 5, "Executing request", theHTTP.progress() );
}
  job.log( 6, "Finished executing request..." );
```

getServerResponse() : ByteArray

Returns the byte array containing the server response. This byte array can contain the content of the file or document being downloaded, JSON or XML data or whatever (he server has responded with.

Note: Avoid using this method when downloading large files because this may cause memory issues. In such cases, the *localFilePath* property should be used instead.

getHeaderNames() : String[]

Returns the array of all HTTP header names present in the server response.

```
var theNames = theHTTP.getHeaderNames();
```

getHeaderValue(headerName : String) : ByteArray

Returns the byte array containing the HTTP header content. If the header is missing, the empty byte array is returned.

```
var theValue = theHTTP.getHeaderValue( HTTP.ContentDisposition );
job.log( 1, "Content-Disposition: %1", theValue.toString("ISO 8859-1") );
theValue = theHTTP.getHeaderValue( "Cache-Control" );
job.log( 1, "Cache-Control: %1", theValue.toString("ISO 8859-1") );
```

addHeader(headerName : String, headerValue : String)

This method can be used to specify additional headers for the HTTP request. This method should be used with caution. If there are invalid headers, HTTP requests may fail.

```
theHTTP.addHeader( "x-metadata", "size=12345; path=/" );
theHTTP.addHeader( "userlogin", "OK" );
```

resetHeaders()

Clears headers previously added by the addHeader() method.

```
theHTTP.resetHeaders();
```

interrupt()

Interrupts the current request to the server.

If this is GET request downloading some file, then such interrupted download may be resumed if the same GET request is executed later. To enable resuming, the resumeEnabled property must be set to true and tempFolderPath property must be defined.

```
theHTTP.interrupt();
```

setPostData(body : ByteArray, contentType : String)

Appends a body to the HTTP Post and Put requests after the HTTP headers.

The first parameter is ByteAray, which contains data, and the second parameter is a string, which represents the Content-Type of the body. For example, if you want to send a JSON as post body, you should set the second parameter to “application/json”. An HTTP 'Content-Length' header is also added to the request. Its value is the length of the ByteAray in setPostData().

Note: Don't use setPostData() and setAttachedFile() together; in that case setAttachedFile() will be ignored.

```
function jobArrived( s : Switch, job : Job )
{
  var theHTTP = new HTTP( HTTP.NoSSL );
  theHTTP.url = "http://httpbin.org/post";

  var postBody = new ByteArray("{\"key\":\"value\"}", "UTF-8");
  theHTTP.setPostData( postBody,"application/json" );

  var theResponseFilePath = job.createPathWithName( job.getNameProper(), false );
  theHTTP.localFilePath = theResponseFilePath;
  theHTTP.post();
  while( !theHTTP.waitForFinished( 3 ) ) { }
  job.log( 1, "Request finish status: %1", theHTTP.finishedStatus );
  job.sendToSingle(theResponseFilePath) ;
}
```

HTTP cookies

An HTTP cookie (web cookie) is a small piece of data that a server sends to a client. The client may store it and send it back with the next request to the same server.

When receiving an HTTP request, a server can send a 'Set-Cookie' header along with the response. The header value is usually stored by the client, and then sent with requests made to the same server inside a 'Cookie' HTTP header.

The example below shows how a cookie header can be used with an HTTP object:

```
 var theHTTP = new HTTP();
 theHTTP.url = "http://127.0.0.1:51088/auth?username=administrator&password=12345";
 theHTTP.post();
 while( !theHTTP.waitForFinished( 1 ) ) { }
 if( theHTTP.finishedStatus != HTTP.Ok )
 {
  job.fail( "Request failed" );
  return;
 }
 var theCookie = theHTTP.getHeaderValue( HTTP.SetCookie ).toString( "UTF-8" );
  if( theCookie.isEmpty() )
 {
  job.fail( "Invalid response" );
  return;
 }

 theHTTP.url = "http://127.0.0.1:51088/api/v1/messages?limit=5";
 theHTTP.addHeader( HTTP.Cookie, theCookie );
 theHTTP.get();
 while( !theHTTP.waitForFinished( 1 ) ) { }
```

Note that it is possible to use the predefined constants "HTTP.SetCookie" and "HTTP.Cookie" to denote the names of the required headers.

Examples

POST request to upload a file to Dropbox

```
function jobArrived( s : Switch, job : Job )
{
  var theHTTP = new HTTP( HTTP.SSL );

  theHTTP.authScheme = HTTP.OauthAuth;
  theHTTP.authorization = "authorization string";
  theHTTP.setAttachedFile( job.getPath() );
  theHTTP.url = "https://api-content.dropbox.com/1/files_put/auto/Test.pdf";

  theHTTP.post();

  job.log( 4, "Upload started", 100);
  while( !theHTTP.waitForFinished( 3 ) )
  {
  job.log( 5, "Uploading...", theHTTP.progress() );
  }
  job.log( 6, "Upload finished" );
  job.log( 1, "Server response: %1",
 theHTTP.getServerResponse().toString( "UTF-8" ) );
  if( theHTTP.finishedStatus == HTTP.Ok && theHTTP.statusCode == 200 )
  {
      job.log( 1, "Upload completed successfully" );
  }
  else
  {
      job.fail( "Upload failed with the status code %1", theHTTP.statusCode );
      return;
  }
  job.sendToSingle( job.getPath() );
}
```

GET request to download a file from Dropbox

```
function jobArrived( s : Switch, job : Job )
{
  var theHTTP = new HTTP( HTTP.SSL );

  theHTTP.authScheme = HTTP.OauthAuth;
  theHTTP.authorization = "authorization string";
  theHTTP.url = "https://api-content.dropbox.com/1/files/auto/Test.pdf";
  theHTTP.localFilePath = job.createPathWithExtension( "pdf", false );

  theHTTP.get();

  job.log( 4, "Download started", 100 );
  while( !theHTTP.waitForFinished( 3 ) )
  {
      job.log( 5, "Downloading...", theHTTP.progress() );
  }
  job.log( 6, "Download finished" );

  if( theHTTP.finishedStatus == HTTP.Ok && theHTTP.statusCode == 200
          && File.exists( theHTTP.localFilePath ) )
  {
      job.log( 1, "Download completed successfully" );
      job.sendToSingle( theHTTP.localFilePath );
  }
  else
  {
      job.fail( "Download failed with the status code %1", theHTTP.statusCode );
      return;
   }
   job.sendToNull( job.getPath() );
}
```

GET request to delete a file from Dropbox

```
function jobArrived( s : Switch, job : Job )
```

```
{
  var theHTTP = new HTTP( HTTP.SSL );

  theHTTP.authScheme = HTTP.OauthAuth;
  theHTTP.authorization = "authorization string";
  theHTTP.url = "https://api.dropbox.com/1/fileops/delete";
  theHTTP.addParameter( "root", "auto" );
  theHTTP.addParameter( "path", "Test.pdf" );

  theHTTP.get();

  while( !theHTTP.waitForFinished( 3 ) )
  {
        job.log( 1, "Delete in progress" );
  }

  job.log( 1, "Server response: %1",
 theHTTP.getServerResponse().toString( "UTF-8" ) );

  if( theHTTP.finishedStatus == HTTP.Ok && theHTTP.statusCode == 200 )
  {
        job.log( 1, "Delete completed successfully" );
  }
  else
  {     job.fail( "Delete failed with the status code %1", theHTTP.statusCode );
        return;
  }
  job.sendToSingle( job.getPath() );
}
```

PUT request to send JSON data from the job file to the HTTP server

```
function jobArrived( s : Switch, job : Job )
{
  var theHTTP = new HTTP( HTTP.NoSSL );
  theHTTP.url = "http://httpbin.org/put";
  theHTTP.authScheme = HTTP.NoneAuth;
  theHTTP.setAttachedFile( job.getPath() );
  var theResponseFilePath = job.createPathWithName( job.getNameProper(), false );
  theHTTP.localFilePath = theResponseFilePath;
  theHTTP.addHeader( "Content-Type", "application/json" );

  theHTTP.put();
  while( !theHTTP.waitForFinished( 3 ) ) { }

  job.log( 1, "Request finish status: %1", theHTTP.finishedStatus );

  job.sendToSingle( theResponseFilePath );
}
```

Using HTTP class to work with Google Drive

If the access token is generated, the script code to use for Google Drive is:

```
theHTTP.authScheme = HTTP.OauthAuth;
theHTTP.authorization = "Bearer " + theAccessToken;
```

Google Drive has a timeout for access token validity that is why you have to re-generate it each time using the refresh token, client ID and client secret provided by Google Drive.

This is the full example of the script code to upload and then delete from Google Drive:

```
function ErrorExists( inMessage : String ) : bool
{
 var theError = inMessage.find( "error" );

 return theError >= 0;
}

function RefreshToken( job : Job ) : String
{
```

```
 var theHTTP = new HTTP( HTTP.SSL );

 theHTTP.addParameter( "refresh_token", "REFRESH_TOKEN" );
 theHTTP.addParameter( "client_id", "CLIENT_ID" );
 theHTTP.addParameter( "client_secret", "CLIENT_SECRET" );
 theHTTP.addParameter( "grant_type", "refresh_token" );

 theHTTP.url = "https://accounts.google.com/o/oauth2/token";

 theHTTP.post();

 while( !theHTTP.waitForFinished( 3 ) )
 {
  job.log( 1, "Refresh token in progress" );
 }

 var theFinishedStatus = theHTTP.finishedStatus; // Statuses: Ok, Failed, Interrupted
 var theServerResponse = theHTTP.getServerResponse().toString();

 if( theFinishedStatus == HTTP.Ok && !ErrorExists( theServerResponse ) )
 {
  var theNewTokenWithQuotes = theServerResponse.split( "," )[0].split( ":" )[1];
  var theNewToken = theNewTokenWithQuotes.substring( 2, theNewTokenWithQuotes.length
 - 1 );

  job.log( 1, "The new token with quotes: %1", theNewTokenWithQuotes );
  job.log( 1, "The new token: %1", theNewToken );
  job.log( 1, "Refresh token is successfully" );

  return theNewToken;
 }
 else if( theFinishedStatus == HTTP.Failed || ErrorExists( theServerResponse ) )
 {
  job.log( 1, "Server response: %1", theServerResponse );
  job.log( 3, "Refresh token failed with the error: %1", theHTTP.lastError );
 }
 else
 {
  job.log( 2, "Unknown error" );
 }
}

function jobArrived( s : Switch, job : Job )
{
 var NewAccessToken = RefreshToken( job );

 var theHTTP = new HTTP( HTTP.SSL );

 theHTTP.authScheme = HTTP.OauthAuth;
 theHTTP.authorization = "Bearer " + NewAccessToken;
 theHTTP.setAttachedFile( job.getPath() );
 theHTTP.url = "https://www.googleapis.com/upload/drive/v2/files";

 // send file to Google Drive
 theHTTP.post();

 while( !theHTTP.waitForFinished( 3 ) )
 {
  job.log( 1, "Upload file to Google Drive in progress" );
 }

 var theFinishedStatus = theHTTP.finishedStatus;
 var theServerResponse = theHTTP.getServerResponse().toString();

 if( theFinishedStatus == HTTP.Ok && !ErrorExists( theServerResponse ) )
 {
  job.log( 1, "Server response: %1", theServerResponse );
  job.log( 1, "Upload file to Google Drive completed successfully" );
 }
 else if( theFinishedStatus == HTTP.Failed || ErrorExists( theServerResponse ) )
 {
  job.log( 1, "Server response: %1", theServerResponse );
  job.log( 3, "Upload file to Google Drive failed with the error: %1",
 theHTTP.lastError );
 }
 else
 {
```

```
 job.log( 2, "Unknown error" );
}

// get file ID
var theFileIdWithQuotes = theServerResponse.split( "," )[1].split( ":" )[1];
var theFileId = theFileIdWithQuotes.substring( 2, theFileIdWithQuotes.length - 1 );

theHTTP.url = "https://www.googleapis.com/drive/v2/files/" + theFileId;

// delete file from Google Drive by fileId
theHTTP.del();

while( !theHTTP.waitForFinished( 3 ) )
{
 job.log( 1, "Delete file from Google Drive in progress" );
}

theFinishedStatus = theHTTP.finishedStatus;
theServerResponse = theHTTP.getServerResponse().toString();

if( theFinishedStatus == HTTP.Ok && !ErrorExists( theServerResponse ) )
{
 job.log( 1, "Delete file from Google Drive is successfully" );

 job.sendToSingle( job.getPath() );
}
else if( theFinishedStatus == HTTP.Failed || ErrorExists( theServerResponse ) )
{
 job.log( 1, "Server response: %1", theServerResponse );
 job.log( 3, "Delete file from Google Drive failed with the error: %1",
theHTTP.lastError );
}
else
{
 job.log( 2, "Unknown error" );
}
}
```

Note: The placeholders REFRESH_TOKEN, CLIENT_ID, CLIENT_SECRET are to be replaced with the actual values.

5.5.2. SOAP class

Background

The SOAP communication protocol and the Web Services framework provide a standard mechanism to communicate between applications and more precisely to request the execution of a particular service in a different application (a "remote procedure call"). The communicating applications may be running on the same computer, on computers in the same local network or on remote computers across the Internet.

For more information, see *http://www.w3.org/TR/SOAP/*.

Overview

The SOAP class supports a subset of the SOAP 1.1 communication protocol over HTTP. Specifically, it allows remote procedure calls to be made to a remote server. The SOAP protocol is used to marshall JavaScript parameters to a remote procedure call and to unmarshall the result as a JavaScript object.

The current implementation has the following restrictions:

- Supports only SOAP 1.1 (and not 1.2).
- Supports only synchronous processing (and not asynchronous).
- Supports DIME and MIME attachments in the SOAP message

- Always uses SOAP encoding style (as defined in the specification section 5).

Note: This class uses the proxy settings from the Switch Server Preferences.

Class instances

The send and request methods - which were already available in older Switch versions – are static functions which allow sending of a simple SOAP message without headers. As they are static functions, there is no need to create instances of the SOAP class.

For more direct control an instance of the SOAP class may be created. This represents an entire SOAP request and/or response message including body, headers and envelope.

Static Functions

request(url : String, request-object : request-object, action : String) : response-object

Initiates a remote procedure call (RPC) or sends an XML message to a SOAP HTTP endpoint, waits for the endpoint to reply (synchronous processing) and returns its response.

An exception is thrown if the request object does not conform, when the SOAP endpoint returns a SOAPFault or when there is a failure from the underlying HTTP transport layer.

send (url : String, request-object : request-object, action : String)

Initiates a remote procedure call (RPC) or sends an XML message to a SOAP HTTP endpoint without waiting for a reply.

An exception is thrown if the request object does not conform or when there is a failure from the underlying HTTP transport layer.

Constructing

SOAP ( )

Constructs a new empty SOAP request.

SOAP (request-object : request-object)

Constructs a new SOAP request and sets the SOAP body

SOAP (in-path : String)

Constructs a new SOAP request and reads the SOAP envelope from the specified XML file.

Property accessing

isResponse ( ) : Boolean

Returns whether the SOAP message is a SOAP response.

getBody ( ) response-object

Returns the SOAP body from the SOAP message.

setBody (request-object : request-object)

Sets the SOAP body to the SOAP request. The body is replaced with the new body and the headers are preserved.

getHeader ( ) : header-object

Returns the SOAP header from the SOAP message.

setHeader (header-object : header-object)

Sets the SOAP header to the SOAP request. The body is associated with the new header and the body itself is preserved.

Sending messages

requestMessage (url : String, action : String) : SOAP

Initiates a remote procedure call (RPC) or sends an XML message to a SOAP HTTP endpoint, waits for the endpoint to reply (synchronous processing) and returns its response.

An exception is thrown if the request object does not conform, when the SOAP endpoint returns a SOAPFault or when there is a failure from the underlying HTTP transport layer.

sendMessage (url : String, action : String)

Initiates a remote procedure call (RPC) or sends an XML message to a SOAP HTTP endpoint without waiting for a reply.

An exception is thrown if the request object does not conform or when there is a failure from the underlying HTTP transport layer.

Logging and debugging

writeToFile (filename : String) : Boolean

Writes the SOAP message envelope (including body and headers) to an XML file

Working with attachments

Data exchange is widely used in Web Services word. SOAP message is convenient to exchange with text data, but not convenient when exchanging large amount of data or when data is in binary format. For such purposes SOAP protocol provides ability to attach additional data to SOAP message. Switch SOAP class supports two types of attachments:

- **DIME** (Direct Internet Message Encapsulation) - in this case SOAP message contains DIME attachments, when sending the message is being sent packet in DIME format and SOAP message header contains type of "`application/dime`". See *http://msdn.microsoft.com/en-us/magazine/cc188797.aspx*
- **MIME** (Multipurpose Internet Mail Extensions) - in this case SOAP message contains MIME attachments, when sending the message is being sent in MIME format and SOAP message header contains type of "`multipart/related`". See article *http://www.w3.org/TR/SOAP-attachments*

addAttachment(file-path : String, id : String, type : String, chunk-size : Number)

add attachment to message specifying the file name, attachment id and type values, chunk size is not used and reserved for future use, should be 0

getAttachmentsCount() : Number

return number of attachments in the SOAP message

getAttachment(index : Number) : ByteArray

getAttachment(id : String) : ByteArray

return contents of attachment specified by index "index" of attachment identifier "id"

Note: These functions are **deprecated** as of Switch 12 Update 1. Please use **getAttachmentFilePath** instead!

getAttachmentFilePath(index : Number) : String
getAttachmentFilePath(id : String) : String

return the path to an attachment specified by index "index" or attachment identifier "id"

getAttachmentID(index : Number) : String

get attachment identifier, specifying attachment by index

getAttachmentType(index : Number) : String

return attachment type string

getAttachmentIndex(id : String) : Number

get index of attachment with the given identifier

saveAttachment(index : Number, file-path : String) : Boolean
saveAttachment(id : String, file-path : String) : Boolean

save attachment to the disk file

setAttachmentSendType(attachment-type : Number)
getAttachmentSendType() : Number

set/get attachment send type: 0 - DIME format; 1 - MIME format

Working with SOAP message body, retrieving information

Since SOAP message body is a text in XML format, it is convenient to gather information from it as from XML. There are some functions to work with SOAP message as with XML document:

parseBody(element-name : String) : String[ ]

returns array of elements' values specified with name equivalent to Document's evalToNodesList, getNodeValue, except that elements specified by name, not by XPath

saveBody(file-path : String, root-element-name : String) : Boolean

saves extract from SOAP body specifying root element into file on disk

changeElement(element-name : String, new-value : String)

set SOAP body element value

5.5.2.1. SOAP class: sample code

Sending attachments to the server

```
theSoapObject.addAttachment( filePath, "e5bb1dbe06c6448f4511f958bc18b50f",
 theFileMimeType, 0 );
// this is DIME attachment
theSoapObject.setAttachmentSendType(0);
```

Remarks:

- When adding an attachment, it is necessary to specify its href. This is usually mentioned somewhere in the SOAP body.
- The SOAP server supports DIME attachments, therefore it is necessary to specify it like this:

```
var Files =
 { "Attachment" :
```

```
        {"Rendition" : "native",
          "Type" : theFileMimeType,
          "Content" : {"soapAttr" : "href=cid:e5bb1dbe06c6448f4511f958bc18b50f",
                       "soapValue" : ""}
         }
  };
```

Getting attachments from the SOAP response

```
var theDocId = parse( soapObject, "//Objects/Object/MetaData/BasicMetaData/ID" );
// get object from server
var theGetObjectsResult = getObject( s, theDocId, lockFile, theRenditionType );
// check returned result
var href = parse(theGetObjectsResult, "//Files/
Attachment[Rendition='"+theRenditionType+"']/Content/@href");
if(href == "")
             {
              s.log( -1, "Server does not return file for document with id %1",
 theDocId );
              return "";
              }

var idx = href.indexOf(":");
if(idx != -1) href = href.mid(idx + 1, href.length - (idx + 1));
// save attachment
var fName = parse(theGetObjectsResult, "//Objects/Object/MetaData/BasicMetaData/
Name");
var fExt = getExtByMimeType( s, gLogonResponse, parse( theGetObjectsResult, "//
Objects/Object/Files/Attachment/Type" ) );
var fileName = s.createPathWithName( fName + fExt, false );
if( theGetObjectsResult.saveAttachment( href, fileName ) )
              {
               s.log( -1, "Attachment saved %1", fileName );
               return fileName;
               }
               else
                   {
                    s.log( 3, "Cannot save attachment" );
                    return "";
                    }
```

Remarks:

- The "parse" function is a wrapper for "parseBody"
- "getObject" performs requests to the SOAP server

5.5.2.2. OASIS support

addOasisSecurity (username : String, password : String, passwordtype : String) : Boolean

This call modifies the SOAP header to include an OASIS security header. It adds a wsse:Security element with a wsse:UsernameToken of the specified password type. Any existing wsse:Security element is replaced by the new element.

The following table describes the parameters and return value.

Parameter	Description
url	The URL for a SOAP HTTP endpoint, e.g. http://serverName:portNumber/URI
request object	An object that specifies the remote procedure name and parameters of the XML message to send; see separate section below

Parameter	Description
action	The SOAPAction, a URN written to an HTTP header used by firewalls and servers to filter SOAP requests; the SOAP service description will usually describe the SOAPAction header required, if any The default is for the action to be an empty string
response object	An object that describes the return value received by the remote procedure call; see separate section below
username	The username used for authentication
password	The password used for authentication
passwordtype	One of "PasswordText" or "PasswordDigest".

Qualified names

In the request object, a qualified name is specified as a string in the <namespace>:<localname> notation, where <namespace> can be a custom namespace or one of the predefined prefixes "xsd" (http://www.w3.org/2001/XMLSchema) or "SOAP-ENC" (http://schemas.xmlsoap.org/soap/encoding/).

For example:

```
"http://soapinterop.org/:param1"
"xsd:int"
```

Request object

The request object is an object literal that specifies the remote procedure name and the parameters to call. The object literal uses the qualified method name of the remote procedure as the key. The value of this key is an object literal in which each key is a parameter of the method and the value of each key is the value of the corresponding parameter of the method.

For example:

```
{ "http://soapinterop.org/:echoString":{inputString: "Echo!"} }
```

When passing parameters to a remote procedure, JavaScript types are bound to SOAP types automatically as listed in the following table.

JavaScript type	SOAP type	Comments
Boolean	xsd:boolean	
Number	xsd:float	
String	xsd:string	
Date	xsd:dateTime	

JavaScript type	SOAP type	Comments
Array	SOAP-ENC:Array	Single-dimension arrays only All items must have the same data type, which must be Boolean, Number, String or Date
ByteArray	SOAP-ENC:base64	Uses the ByteArray "Base64" codec
Other	Not supported	Causes an exception to be thrown

Instead of providing one of these JavaScript data types, a parameter value can also be represented by a literal object with the following properties:

Property	Description
soapType : String	The SOAP type (as a qualified type name) that will be used for the value when generating the SOAP message; this is useful when a datatype is needed other than the automatic bindings described above
soapValue : String	The value that will be used when generating the SOAP message The value is passed unescaped (example: "<" is not converted to "<") which means that it can be a raw XML fragment that will be passed on as is

For example, integers are not supported in JavaScript but an integer parameter to a SOAP method can be constructed as follows:

```
var iPar = { soapType: "xsd:int", soapValue: "1" };
{ "http://soapinterop.org/:echoInteger": { inputInteger: iPar } }
```

Header object

The header-object is an object literal that specifies the soap header to be included. The object literal uses the qualified name of the header XML element(s) as the key. The value of this key is encoded like a request-object.

For example:

```
{ "http://example.org/2001/06/tx:Transaction":5, "http://example.org/2001/06/
tx:PaymentOption":10}
```

will create the following header:

```
<env:Header xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
<t:Transaction xmlns:t="http://example.org/2001/06/tx">
5
</t:Transaction>
<t: PaymentOption xmlns:t="http://example.org/2001/06/tx">
10
```

```
</t: PaymentOption >
</env:Header>
```

Response object

The function returns an object describing the SOAP Body of the returned message in a manner similar to the request object.

The SOAP types in the result are mapped to JavaScript types as listed in the following table.

SOAP type	JavaScript type	Comments
xsd:boolean	Boolean	
xsd:integer	Number	
xsd:float	Number	
xsd:string	String	
xsd:dateTime	Date	
SOAP-ENC:Array	Array	Single-dimension arrays only All items must have the same data type, which must be boolean, integer, float, string or dateTime
xsd:hexBinary	ByteArray	Uses the ByteArray "Hex" codec
xsd:base64Binary	ByteArray	Uses the ByteArray "Base64" codec
SOAP-ENC:base64	ByteArray	Uses the ByteArray "Base64" codec
Other	String	Copies the XML string content without change

Example

```
var url = <get a URL for this service from http://www.whitemesa.com/interop.htm>;
// Call the echoString SOAP method
var testString = "This is my test string";
var response = SOAP.request( url,
  { "http://soapinterop.org/:echoString": { inputString: testString } },
  "http://soapinterop.org/" );
var result = response["http://soapinterop.org/:echoStringResponse"]["return"];

// Call the echoInteger SOAP method
var testInt = { soapType: "xsd:int", soapValue: "10" };
var response = SOAP.request( url,
  { "http://soapinterop.org/:echoInteger": { inputInteger: testInt } },
  "http://soapinterop.org/" );
var result = response["http://soapinterop.org/:echoIntegerResponse"]["return"];
```

5.6. Database module

5.6.1. Text encoding

JavaScript strings in Switch store Unicode text in UTF-16 encoding. A database implementation may use a different text encoding, in which case the appropriate conversion must be performed when exchanging text. The ODBC interface unfortunately does not offer a mechanism to discover the text encoding used by the database, so this information must be provided by the user.

Most modern databases use Unicode-aware encodings such as UTF-16 or UTF-8. Older database implementations may use local 8-bit encodings that are not Unicode-aware. To complicate matters even more, some databases use a mixture of encodings (for example UTF-8 for storing text in data fields and UTF-16 for interpreting queries).

Codec Arguments

Selected functions in the Switch database module offer extra "codec" arguments to support databases that use text encodings other than UTF-16.

The DataSource.connect() function allows specifying two codec arguments:

- query-codec: affects queries sent to and column names retrieved from the database.
- data-codec: affects STRING and BINARY data fields when retrieved as a string.

The codecs established with the connect() function are used for all Statement instances derived from the DataSource during the connection. However the Statement.getString() function allows overriding the data-codec on an individual field basis.

Text items exchanged with the database

Function	Text item	Direction	Encoding
connect()	user name and password	To database	query-codec
execute()	SQL statement	To database	query-codec
tables()	Table names	From database	query-codec
columns()	Table name	To database	query-codec
	Column names	From database	query-codec
getColumnName() getColumnDataType() getColumnSize()	Column name	From database	query-codec

Function	Text item	Direction	Encoding
getNumber() getDate() getString() getBinary()	Column name	From database	query-codec
getString() for STRING and BINARY data types	Data field	From database	data-codec

Default behaviour

The default query-codec and data-codec is UTF-16.

If the data-codec for the connection is UTF-16, the default codec for converting BINARY column data is latin-1 (copy the low byte and clear the high byte of each code point).

If the data-codec for the connection is an 8-bit encoding (i.e. not UTF-16), that encoding is also the default for converting BINARY column data.

The default behavior is summarized in the following table.

Item	Algorithm to determine codec
query-codec	If DataSource.connect() explicitly specifies a query-codec, use that codec. Otherwise use UTF-16
data-codec for STRING data in Statement.getString()	If Statement.getString() explicitly specifies a data-codec, use that code. If DataSource.connect() explicitly specifies a data-codec, use that codec. Otherwise use UTF-16
data-codec for BINARY data in Statement.getString()	If Statement.getString() explicitly specifies a data-codec, use that codec (even if the specified codec is UTF-16). If DataSource.connect() explicitly specifies a data-codec other than UTF-16, use that codec. Otherwise use latin-1

5.6.2. DataSource class

Background

ODBC (Open Database Connectivity) provides a standard mechanism for using databases independent of programming languages, database systems, and operating systems.

The ODBC specification offers a procedural API for using SQL queries to access data. Both Windows and Mac OS X offer a built-in ODBC implementation, and OBDC drivers exist for a large variety of database systems and data sources including spreadsheets, text and XML files.

System requirements

The system administrator of the computer hosting Switch must correctly install and configure the appropriate ODBC drivers and the system's ODBC driver manager.

Overview

The DataSource class encapsulates a connection to a particular ODBC data source. A script can be connected to different data sources at the same time. Within the limited capabilities of the Database module, it is not meaningful to have multiple connections to the same data source.

The Statement class allows executing a SQL statement on a data source, and allows retrieving the results.

Constructing

DataSource( ) : DataSource

Constructs a new unconnected data source.

Connecting

connect(data-source-name : String, user : String, password : String, query-codec : String, data-codec : String) : Boolean

Attempts to connect to the specified data source with the specified user and password. If the user and password are empty or null, no user authentication is performed.

Returns true if the connection succeeds, false otherwise (in which case an error message is logged).

The optional codec arguments specify the text encoding used by the connection to communicate with the database and retrieve data. If an argument is missing or null, UTF-16 is used as the default. For more information see separate section on text encoding issues.

disconnect( )

Disconnects the data source if it was connected; otherwise does nothing. Disconnecting a data source invalidates all associated Statement instances.

A disconnected data source can be reconnected by calling connect().

Switch automatically disconnects a data source when exiting the entry point in which it was constructed .Still, it is good programming style to explicitly disconnect all data sources.

isConnected( ) : Boolean

Returns true if the data source is currently connected, false otherwise.

useConnection(data-source-name : String) : Boolean

Attempts to connect to the database with the parameters specified in "data-source-name" Switch data source (in **ODBC data sources** section of **Preferences**). Data source name, user, password and codecs information will be taken from Switch's data source settings.

Returns true if the connection succeeds, false otherwise (in which case an error message is logged).

5.6.3. Statement class

The Statement class provides a context to execute a series of SQL statements on a data source and retrieve the results. A statement is always associated with a particular data source. By constructing multiple statements on the same data source, you can access the results of multiple queries at the same time.

Constructing

Statement(data-source : DataSource) : Statement

Constructs a statement associated with the specified data source. The data source must be connected; if not the resulting Statement instance is invalid.

Getting state information

isOK( ) : Boolean

Returns true if the previous operation returned "SQL_SUCCESS" or SQL_SUCCESS_WITH_INFO"; false otherwise.

isSuccess( ) : Boolean

Returns true if the previous operation returned "SQL_SUCCESS"; false otherwise.

isSuccessWithInfo( ) : Boolean

Returns true if the previous operation returned "SQL_SUCCESS_WITH_INFO"; false otherwise.

getCode( ) : Number

Returns the native (system- or driver-specific) ODBC error or information code generated by the previous operation, or zero if there was no such code.

getMessage( ) : String

Returns the human-readable ODBC error or information message generated by the previous operation, or the empty string if there was no such message.

The message is formatted to already include the SQL state and the native code.

isValid( ) : Boolean

Returns true if the associated data source is connected, false otherwise.

Executing SQL statements

execute(sql : String) : Boolean

Executes the specified SQL statement on the associated data source, and stores the results.

Returns the value of isOK() after performing the operation.

tables (table-type : String) : String

Returns a list of table names. The list includes all tables of the specified type in the data source. The table type must be specified in all upper-case, and multiple types must be comma-separated (for example: "TABLE, VIEW"). If the argument is missing, or an empty string, information is retrieved about all tables in the data source.

Some databases support the "SHOW TABLES" query to retrieve more details about tables as a regular result set. After passing this query to the database with the execute() function, the result set contains a column with the table names and further columns with addiitonal information on each table.

columns (table-name : String) : String

Returns a list of column names, including all columns in the specified table.

Some databases support the "SHOW COLUMNS FROM <table-name>" query to retrieve more details about columns in a table as a regular result set. After passing this query to the database with the execute() function, the result set contains a column with the column names and further columns with addiitonal information on each column.

5.6.3.1. Getting information about the result set

getRowCount( ) : Number

Returns the number of rows affected by an UPDATE, INSERT, or DELETE statement.

For some data sources, this function returns the number of rows returned by a SELECT statement. However, many data sources cannot provide this information before fetching the rows. If the function is unable to provide the information, it returns -1.

getNumColumns( ) : Number

Returns the number of columns in the result set, or -1 if an error occurred (or if there is no result set)

getColumnName (column : Number) : String

Returns the name of the specified column (as a one-based index), or the empty string if the name can't be determined.

getColumnDataType (column : Number or String) : String

Returns the data type of the specified column (one-based index or column name) as one of these strings:

- INTEGER: integer numeric value or single-bit binary value (Boolean)
- REAL: floating point or fixed point numeric value
- DATETIME: date or time or a combination thereof
- STRING: character string in a known encoding (so that it can be converted to Unicode)
- BINARY: character string in unknown encoding, or binary data, or unsupported data type

getColumnSize (column : Number or String) : Number

Returns the size in bytes needed to store the data in the specified column (one-based index or column name), or -1 if the size can't be determined.

5.6.3.2. Getting the actual data in the result set

isRowAvailable( ) : Boolean

Returns true if a row is available for fetching; false otherwise. This function is used to terminate the iteration over the rows in the result set.

fetchRow( )

Fetches the next available row, making its data values available for the functions described below.

Get functions

The functions described below return the null object (as opposed to an empty string or a zero number) if:

- The specified column can't be found.
- There is no currently fetched row.

- The data type for the specified column can't be converted to the requested data type (see table in the following section).
- The value in the specified column for the currently fetched row does not conform to the format for the requested data type.

getNumber(column : Number or String) : Number

Returns the value in the specified column for the currently fetched row as a number.

getDate(column : Number or String) : Date

Returns the value in the specified column for the currently fetched row as a Date object (date-time value).

getString(column : Number or String , data-codec : String) : String

Returns the value in the specified column for the currently fetched row as a string.

The optional codec argument specifies the text encoding used to convert STRING and BINARY data to a Unicode string. If the argument is missing or null, latin-1, UTF-16 or the encoding specified for the connection is used as the default. For more information see separate section on text encoding issues.

The codec argument is ignored if the specified column has a data type other than "STRING" or "BINARY".

getBinary(column : Number or String) : ByteArray

Returns the value in the specified column for the currently fetched row as a sequence of bytes.

Supported conversions

Column data type	Number	Date	String	Binary
INTEGER	Straightforward (*)	Not supported	Decimal representation of the number	Not supported
REAL	Straightforward (*)	Not supported	Decimal representation of the number	Not supported
DATETIME	Not supported	Straightforward	ISO 8601 representation of the date-time	Not supported
STRING	Decimal number representation (Unicode)	Interpret as ISO date-time representation (Unicode)	Straightforward (Unicode)	Not supported
BINARY	Decimal number representation (ASCII)	Interpret as ISO 8601 date-time representation (ASCII)	Interpret as latin-1 encoding (i.e. clear high bytes)	Straightforward

(*) Values out of the range that can be represented in JavaScript are converted to negative or positive infinity.

5.7. Flow element module

5.7.1. Entry points

Entry points form the mechanism for passing control from Switch to a script in a structured manner. Specifically, an entry point is a function defined in a script (according to the rules specified below) and will be called by the Switch application at the appropriate times. All other functions defined in the scripting API work the other way around: they expect to be called from a script.

Switch invokes the entry points defined in a script at certain appropriate times, as explained below. Also see execution modes.

The first argument for each entry point is an instance of the Switch class or of the Environment class. This object serves as the starting point for accessing the information offered by the flow element and metadata modules in the scripting API.

5.7.1.1. Regular execution

Switch expects at least one of the following three entry points to be present in each script:

jobArrived(s : Switch, job : Job)

This entry point is invoked each time a new job arrives in one of the input folders for the script element. The newly arrived job is passed to the entry point as the second argument.

```
//logs the name of each incoming job
function jobArrived( s : Switch, job : Job )
{
    job.log(1, job.getName());
}
```

timerFired(s : Switch)

This entry point is invoked for the first time immediately after the flow is activated and thereafter it is invoked at regular intervals regardless of whether a new job arrived or not. The default value for the interval is 300 seconds (5 minutes); it can be modified with Switch.setTimerInterval().

```
//logs the username each 10+,20+,10+,20+,... seconds
function timerFired( s : Switch ) {
    if(s.getTimerInterval() == 10){
        s.setTimerInterval(20);
    }
    else{
        s.setTimerInterval(10);
    }
    s.log(1, s.getSystemInfo("UserName"));
}
```

webhookTriggered(s : Switch, r : WebhookRequest)

This entry point allows the script to subscribe to specific requests. Afterwards the entry point is automatically invoked by Switch each time the request with attributes matching the subscription arrives to Switch. For more information, refer to *webhookTriggered* on page 169.

constructWebhookResponse(r : WebhookRequest, response : WebhookResponse, args : Array) : Boolean

This entry point allows the script writer to construct custom responses for specific webhook requests. For more information, refer to *constructWebhookResponse* on page 170.

5.7.1.2. Property editing

These optional entry points support editing and validating properties defined in the script declaration; see *Property definition properties*, *Property editors* and *Validating property values*.

getLibraryForProperty(s : Switch, tag : String) : String[ ]

Invoked when a user chooses the "Select from library" or "Select many from library" property editor in the designer for one of this script's properties. The second argument specifies the tag of the property involved in the operation.

Returns the list of strings to display in the property editor dialog.

If this entry point is absent, it is considered to return an empty list.

```
//returns the same library for each tag and Switch object
function getLibraryForProperty( s : Switch, tag : String ) {
    var opts = [];
    opts[0] = "option1";
    opts[1] = "option2";
    return opts;
}
```

getLibraryForConnectionProperty(s : Switch, c : Connection, tag : String) : String[ ]

Invoked when a user chooses the "Select from library" or "Select many from library" property editor in the designer for a property injected on this script's outgoing connections. The second and third arguments specify the connection and the tag of the property involved in the operation.

Returns the list of strings to display in the property editor dialog.

If this entry point is absent, the getLibraryForProperty entry point is invoked instead, omitting the connection argument. This is meaningful in case the result does not depend on the connection involved in the operation.

```
//returns the same library for each connection, switch and tag
function getLibraryForConnectionProperty( s : Switch,
c : Connection, tag : String ) {
    var opts = new Array("opt1", "opt2", "opt3");
    return opts;
}
```

isPropertyValid(s : Switch, tag : String, value : String) : Boolean

Invoked at appropriate times to validate the value of a property for the script with "custom" validation, example: when the user changes the property value in the designer, when the user attempts to activate the flow in which the script resides, or just before invoking the jobArrived entry point (while the flow is running). See validating property values.

The second argument specifies the tag of the property that needs to be validated.

The third argument specifies the current value of the property. It is provided for convenience only; it is identical to the return value of getPropertyValue(tag).

Returns true if the value of the specified property is deemed valid, false otherwise.

If this entry point is absent, it is considered to return false.

```
function isPropertyValid( s : Switch, tag : String,value : String ){
//condition to test
boolean valid = <condition>;
return valid;
}
```

isConnectionPropertyValid(s : Switch, c : Connection, tag : String, value : String) : Boolean

Invoked at appropriate times to validate the value of a property injected on this script's outgoing connections with "custom" validation, example: when the user changes the property value in the designer, when the user attempts to activate the flow in which the script resides, or just before invoking the jobArrived entry point (while the flow is running). See validating property values.

The second and third arguments specify the connection and the tag of the property involved in the operation. The last argument specifies the current value of the property. It is provided for convenience only; it is identical to the return value of getPropertyValue(tag).

Returns true if the value of the specified property is deemed valid, false otherwise.

If this entry point is absent, the isPropertyValid entry point is invoked instead, omitting the connection argument. This is meaningful in case the result doesn't depend on the connection involved in the operation.

```
function isConnectionPropertyValid( s : Switch, tag : String, value : String ) {
 //condition to test
    boolean valid = <condition>;
    return valid;
}
```

No run-time context

These entry points are often invoked while the flow is inactive, which means there is no valid flow run-time context and no valid job context. As a consequence, within these entry points calling the following Switch class functions has undefined (and possibly destructive) results:

- Working with jobs and processes: getJobs, getJobsForConnections, createNewJob, failProcess
- Accessing the timer interval: setTimerInterval, getTimerInterval

It is safe to call the Switch class functions for accessing the flow definition and accessing injected properties and any of the Environment class functions inherited by the Switch class.

Variables and script expressions

Furthermore, when the flow is inactive, there is no way to determine the value of a property that contains a variable or a script expression. Thus in that case the getPropertyValue() and getPropertyValueList() functions return the source value of a property instead of its computed value (that is, the variable or script expression definition rather than its evaluation result).

The isPropertyValid entry point is invoked only when an actual value is available for the property that needs to be validated (that is, a property that contains a variable or a script expression is never validated when the flow is inactive; see validating property values). So in most cases the validation process does not need to worry about this complexity. However sometimes properties

are interrelated and the validation process requires referring to another property value. In that case, the validation code must verify that an actual value exists for the other property through the isPropertyValueActual() function.

5.7.1.3. Application discovery and licensing

The optional entry points described in this section provide support for scripted plug-ins that serve as a configurator for a third-party application. If the script is not loaded as a scripted plug-in these entry points are never invoked. See also creating a scripted plug-in, configurator guidelines and detecting third-party applications.

findApplicationPath(e : Environment, startup : Boolean) : String

Is invoked for a scripted plug-in to discover the associated third-party application – if any.

Returns the absolute file path for the third-party application's executable (".exe" on Windows, ".app" on Mac). If the entry point returns null or the empty string, this means that the script cannot discover the application and that the user will need to set the application path.

The startup argument is true when this entry point is invoked during Switch startup, and false when invoked as a result of a user action (that is, not during startup). When startup is true, the entry point should not perform time-consuming operations. The Environment.findApplicationOnDisk() function and find64BitApplicationOnDisk are automatically disabled (returning an empty string without searching), so it is safe to call these functions without explicitly checking for startup.

If this entry point is absent, Switch offers no user interface for setting an application path for this scripted plug-in; otherwise it does (even if the script can determine the application path).

This function is not invoked during Switch startup if there is already an ApplicationPath cached (stored in registry). In such a case it is only possible to call it manually or first cleanup the application path in the registry.

checkApplicationPath(e : Environment, path : String) : Boolean

Is invoked for a scripted plug-in to check if specified path is valid for the configurator or not.

This function is automatically called in Switch in the following situations:

- When a configurator is upgraded.
- When a user manually sets the path to the configurator's application (in the Flow Elements pane, via the "Set path to application" option of in the context menu of the configurator).

getApplicationLicensing(e : Environment) : String

Is invoked for a scripted plug-in to discover the licensing state of the associated third-party application. Returns one of the following strings to indicate the licensing state: "Licensed", "Demo", "Trial", "Unlicensed", "Unknown". The Switch designer displays a localized version of these strings.

If this entry point is absent, this means the licensing state is irrelevant or cannot be discovered even in principle. This not the same thing as returning "Unknown", which means there is a mechanism for discovering the licensing state but it failed this time to produce a conclusive result.

licenseApplication(e : Environment, license : String)

Is invoked when the user enters or modifies the license key for the scripted plug-in in the designer. Attempts to license the associated third-party application with the specified string.

If this entry point is absent, Switch offers no user interface for entering a license key for this scripted plug-in.

5.7.1.4. Webhooks

webhookTriggered

The webhookTriggered entry point allows the script to subscribe to specific requests. Afterwards the entry point is automatically invoked by Switch each time the request with attributes matching the subscription arrives to Switch. The attributes include protocol type, request method and URL path.

Note: If a workflow includes multiple flow elements that use the same script (called 'script instances'), then each flow element subscribes separately and independently on other flow elements. So if the script uses a fixed hardcoded path, then all script instances in active flows will receive the same notification. The order of delivering such a notification to the script instances is arbitrary.

If the entry point is present in the script, it will be called the moment the flow is activated; the WebhookRequest argument is null in that case. This is the moment when the script instance can subscribe to the requests it wants to get notifications about. If it doesn't subscribe to any request, then the entry point will never be called again for this script instance (unless the flow is re-activated). A script may subscribe to multiple requests coming to different URL paths. Different scripts may subscribe to the same request.

Subscription of the script instance is automatically cancelled when the flow is deactivated. It is also possible to explicitly unsubscribe from specific requests.

When subscribing to a request, the entry point needs to specify the request method, the URL path and optionally the protocol. If the protocol is not specified, then by default requests arriving via both HTTP and HTTPS will be delivered to the entry point. The subscription may fail (for example, if the corresponding protocol is disabled in the Switch Preferences); in that case, the call will throw an exception.

Note: This entry point is similar to timerFired in all the aspects like concurrency, aborting, debugging in Scripter etc. The difference is that it is not called by a timer event, but by an 'HTTP request arrival' event.

Remarks:

- This entry point is not the request handler - it is invoked AFTER the request has been handled by Switch and the response has been sent to the client. So the entry point is not intended to generate any output to be sent back to the request sender.
- The entry point is *serialized* within each script instance with itself and also with the jobArrived and timerFired entry points, i.e. it can never work concurrently with timerFired, for example. However, in case of the concurrent execution mode the entry point may work concurrenly in different script instances.
- The time interval between the moment a request is sent and the webhookTriggered entry point is executed may vary depending on network speed and the current Switch load. For example, the new entry point cannot be executed concurrently with other entry points like jobArrived or timerFired in the same script instance, so webhookTriggered will be postponed if any of those currently are working.

- Switch doesn't offer any embedded authentication features for webhooks. A script writer, if needed, can implement his own authentication mechanism, based, for example, on some preliminary configured security tokens and signatures passed in custom HTTP headers. See, for example, *https://developer.github.com/webhooks/securing*.

Example:

```
function webhookTriggered(s : Switch, r : WebhookRequest) {
    if (r == null) { // Not a real request, but first initialization call at the
 moment flow is activated -> subscribe
        try {
            s.webhookSubscribe(webhookRequest.POST, "/myscript/notify");
        }
        catch(e) {
            s.log(3, "Failed to subscribe to the request: %1", String( e ));
            return;
        }
        s.log(1, "Subscribed to HTTP and HTTPS requests for the path '/myscript/
notify'");
        return;
    }
    s.log(1, "Received notification from %1", r.remoteAddress);
    var newJob = s.createNewJob();
    newJob.sendToSingle(r.content, "notification.json");
}
```

constructWebhookResponse

The constructWebhookResponse entry point allows the script writer to construct custom responses for specific webhook requests. Using this entry point a script can, for example, integrate with third-party services like Dropbox that require a custom response to a URL verification request. Another use case is when a webhook request from a third-party service requires a specific response that is different from the Switch default response.

function constructWebhookResponse(r : WebhookRequest, response : WebhookResponse, args : Array) : Boolean

The constructWebhookResponse entry point is automatically called when an incoming webhook requestMethod* request comes to path over protocolType in the following conditions:

- The script where this entry point is defined has the webhookTriggered entry point implemented.
- The webhookTriggered entry point has enabled custom responses to webhook requestMethod requests coming to path over protocolType using the Switch::enableCustomWebhookResponse call.

The entry point is enabled the moment enableCustomWebhookResponse is called from webhookTriggered in the same script and disabled the moment disableCustomWebhookResponse is called, or when the flow containing the script is deactivated.

The *WebhookRequest* object is the same as used in the *webhookTriggered* entry point.

The entry point can check the incoming *WebhookRequest* and optionally fill the *WebhookResponse* object with the data that should be the response to send back to the request sender.

If the status code remains 0 after the entry point finishes, the default code 200 will be set automatically. In this case if the response body was empty, the default body '{"status":true}' will be used.

The 'args' argument is the array of simple values (strings, numbers) that optionally can be passed from *webhookTriggered* as the last argument of *enableCustomWebhookResponse*.

The return result indicates if the entry point "consumed" the request:

- if 'true' is returned the request will not be passed to webhookTriggered (for example, the case of "validation" or "ping" requests)
- if 'false' is returned and the 'statusCode' of the WebhookResponse belongs to the range 200-299, the request can be forwarded to the webhookTriggered entry point.

The following implementation of *constructWebhookResponse* reproduces the behavior when this entry point is not implemented at all:

```
function constructWebhookResponse( r : WebhookRequest, response : WebhookResponse,
 args : Array ) : Boolean
{
    response.statusCode = 200;
    response.write('{status:"true"}');
    return false;
}
```

The following example shows how to use the entry point to generate custom responses to URL "validation" requests:

```
function constructWebhookResponse( r : WebhookRequest, response : WebhookResponse,
 args : Array ) : Boolean
{
   // this entry point will only construct responses to 'validation' requests
    if("challenge" in r.query) // expect a parameter named "challenge"
    {
        if(r.query["account"] == args[0]) // verify account name
        {
            response.statusCode = 200;
            response.write(req.query["challenge"]); // the expected response
        }
        else
        {
            response.statusCode = 404;
        }
        return true; // the request was fully consumed - no need to pass it to
 webhookTriggered
    }
    // for other requests use default Switch response
    // other requests should be passed to webhookTriggered
    return false;
}
function webhookTriggered( s : Switch, r : WebhookRequest )
{
    if( r == null )
    {
        try
        {
            var theArguments = [ s.getPropertyValue("theAccount") ];
            // the last argument requires the entry point constructWebhookResponse is
 also defined in the script
            s.enableCustomWebhookResponse(WebhookRequest.GET, "/notification",
 WebhookRequest.HTTP, theArguments); // 'validation' requests are always 'GET'
            s.webhookSubscribe(WebhookRequest.POST, "/notification",
 WebhookRequest.HTTP);
        }
        catch(e)
        {
             s.log(3, "Failed to initialize webhook: %1", e);
        }
        return;
    }
}
```

Note: In case of an internal error that happens before *constructWebhookResponse* can be called, the automatically generated "failed" response will be sent:

```
status code: 500
response body: {status:"false",error:"the error details"}
```

The custom responses are enabled "globally" in Switch. This means that once a specific script enables the custom responses for arriving requests identified by requestMethod, path and protocolType, the responses are always constructed using the script's specific entry point implementation, disregarding how many scripts actually subscribed to these requests using webhookSubscribe. Another script will not be able to enable its own entry point implementation for that combination of requestMethod, path and protocolType arguments.

Enabling and disabling custom responses using the Switch object

Functions*:

- enableCustomWebhookResponse(requestMethod : String, path : String, protocolType : String, args : Array)
- disableCustomWebhookResponse(requestMethod : String, path : String, protocolType : String)
- webhookSubscribe(requestMethod : String, path : String, protocolType : String, enableCustomResponse : Boolean, args : Array)
- webhookUnsubscribe(requestMethod : String, path : String, protocolType : String, disableCustomResponse : Boolean)

Example implementation (verification for DropBox)

```
function constructWebhookResponse( r : WebhookRequest, response : WebhookResponse,
 args : Object ) : Boolean
{
  if("challenge" in r.query)
  {
     response.statusCode = 200;
     response.write(r.query.challenge);
     response.setHeader("Content-Type", "text/plain");
     response.setHeader("X-Content-Type-Options", "nosniff");
     return true;
  }
}
```

* For a full description, refer to the Webhooks section in the topic about the *Switch class* on page 187.

5.7.2. Flow element update

These entry points are invoked during flow import or update. They can be used to provide a new value for a property of an existing flow element, based, for example, on the values of other properties of this element.

The typical use case is when some older script property "Property1" should be replaced by the new property "Property2" and the value of "Property2" for the flow elements in the existing flows should be calculated based on the current value of "Property1". In this case "Property1" replaces "Property2" in the script declaration and the getUpdatedValueForProperty entry point should be implemented that returns the new value for "Property2". Also "Property1" should be

put into the "Obsolete properties" list in the script declaration to make sure Switch does not show a warning about removed property during flow import or update.

Example:

```
function getUpdatedValueForProperty( s : Switch, inTag : String ) : Array
        {
        var theResArray = new Array();
        if( inTag == "Property2" )
        {
        var theNewValue; //calculate it based on the value of
 s.getPropertyValue("Property1");
        theResArray.push(theNewValue);
        theResArray.push(s.EditorSingleLineText); // the editor must match the
 "Property2" definition
        }
        return theResArray;
        }
```

getUpdatedValueForProperty(s : Switch, inTag : String) : Array

getUpdatedValueForConnectionProperty(s : Switch, c : Connection, inTag : String) : Array

Here,

"inTag" is the tag of the property for which new value was requested.

"s" is the Switch object representing the old flow element.

"c" is the Connection object representing the old connection element.

The value returned is an array of two strings where the first item is the new value and the second item is the type of this value (which means the editor which virtually "was used" to set the value).

If the entry point is not implemented or if the array is empty then the corresponding "inTag" property will remain unchanged.

If the array does not contain two items then a warning message will be logged and the property will remain unchanged.

Additionally, if the array contains two items but they cannot be used to set the new property value, a warning message will be logged and the property will remain unchanged.

The reasons why the specified value cannot be used as a new property value are:

- the property does not support such kind of editor
- the value type does not correspond to the property type
- the drop-down value is not in the list of available drop-down values and so on

The list of editors available are:

1. EditorSingleLineText
2. EditorPassword
3. EditorNumber
4. EditorRational
5. EditorHoursAndMinutes
6. EditorDate
7. EditorDateAndTime
8. EditorNoYesList
9. EditorDropdownList
10. EditorLiteral
11. EditorChooseFile

12. EditorChooseFolder
13. EditorRegularExpression
14. EditorFileType
15. EditorSelectFromLibrary
16. EditorMultiLineText
17. EditorScriptExpression
18. EditorSingleLineTextWithVariables
19. EditorMultiLineTextWithVariables
20. EditorConditionWithVariables
21. EditorFilePatterns
22. EditorFolderPatterns
23. EditorFileTypes
24. EditorStringList
25. EditorSelectManyFromLibrary
26. EditorExternalEditor
27. EditorDatabaseQuery

They can be accessed in the script by using the constants like s.EditorName.

For the EditorLiteral, you can use the following literals as the value (first item of the resulting array):

1. LiteralDefault
2. LiteralNone
3. LiteralAutomatic
4. LiteralNoFiles
5. LiteralAllFiles
6. LiteralAllOtherFiles
7. LiteralNoFolders
8. LiteralAllFolders
9. LiteralAllOtherFolder

They can be accessed in the script by using the constants like s.LiteralName.

Example:

```
function getUpdatedValueForProperty( s : Switch, inTag : String ) : Array
{
         var theResArray = new Array();
                  if( inTag == "dataset" )
         {
                theResArray.push( "Xml" );
                theResArray.push( s.EditorSingleLineText );
         }
         else if( inTag == "dataModel" )
         {
                theResArray.push( "XML" );
                theResArray.push( s.EditorDropdownList );
         }         else if( inTag == "locationPath" )
         {
                theResArray.push( s.LiteralNone );
                theResArray.push( s.EditorLiteral );
         }            return theResArray;
}
```

```
function getUpdatedValueForConnectionProperty( s : Switch, c : Connection, inTag :
 String ) : Array
{
         var theResArray = new Array();
```

```
        if( inTag == "Name" )
        {
                if( c.getName().isEmpty() )
                {
                        theResArray.push( "New name" );
                        theResArray.push( s.EditorSingleLineText );
                }
        }
        else if( inTag == "Error" )
        {
                if( !c.allowsSuccess() && !c.allowsError() )
                {
                        theResArray.push( "Yes" );
                        theResArray.push( s.EditorNoYesList );
                }
        }
          return theResArray;
}
```

5.7.3. Environment class

The single instance of the Environment class is passed as an argument to the script entry points that operate out of the context of a particular flow element. By convention, the name for the Environment object is "e" (although the script developer can choose to use another name).

Since the Switch class inherits from the Environment class, any of the functions described in this topic can be called for the Switch class as well. In other words, you can call the Environment class functions with "e.function()" or "s.function()" depending on the argument passed to the entry point.

Logging

log(type : Number, message : String, extra : String or Number)

Logs a message of the specified type (see below) outside of a job context. Where possible use the job.log() function instead, because it includes information about the job in the message.

Log messages are displayed in the Messages pane and in the Progress pane; see also viewing log messages and viewing processes.

Log Type

The "type" argument must have one of the following numeric values:

Numeric type	Display type	Description
1	Info	An informational message that has no other purpose than to inform the user of a certain occurrence
2	Warning	A warning message that informs the user of a recoverable problem or non-fatal error
3	Error	An error message that informs the user of a fatal error
4	Start	A formal message that marks the start of a task that should be displayed in the progress pane

Numeric type	Display type	Description
5	Progress	A formal message that marks the completion of a particular portion of a task for which a start message was previously issued
6	End	A formal message that marks the completion of a task for which a start message was previously issued
-1	Debug	An informational message solely intended for debugging purposes
-2	Assert	A message solely intended for debugging purposes that is issued only when a coding error is discovered

Log Message

The message argument and the optional extra argument specify the message being issued, as a constant and variable part. For example:

```
e.log(2, "Trial period will expire in %1 days", days);
```

The message argument should be a constant string (that is, not constructed programmatically) because it is used as a key in the localization database. The "extra" argument replaces the placeholder "%1" in the localized version of the message.

Note: Placeholder %1 is *always* replaced with the value provided in the "extra" argument; if the "extra" argument is missing, the empty value is used. If for any reason you would like to preserve the %1 placeholder in the log message, you can use the following command:

```
job.log(1, "sample: %1", "%1"); //"sample: %1"
```

If the extra argument is a number, it is rounded to the nearest integer and then converted to the corresponding decimal string representation. This allows picking alternative messages from the localization database depending on the number (example: "1 day" versus "2 days").

```
job.log(1, "sample: %1", 3.5); //"sample: 4"
```

If the extra argument is a string, all leading and trailing white space is removed, and any remaining line breaks (CR, LF, or CRLF) are replaced by a semicolon. This allows passing multi-line text (such as the complete contents of stderr or stdout) to the extra argument without inadvertently causing line breaks in log messages.

```
var str = "      this \n is \n some \n text      ";
job.log(1, "sample: %1", str); //"sample: this ;  is ;  some ;  text"
```

Log: Multiple variable parts

The message argument can use the special placeholders "%11", "%12" through "%19" to access the corresponding segment (indexed 1 through 9) of the extra argument interpreted as a list

of comma-separated values. If the specified segment does not exist the empty string is used. There is no support for quoted strings so individual values in the list can not contain commas.

```
job.log(1, "sample: %11 and %12", "one,two"); //"sample: one and two"
```

Note: Placeholders %2 to %10 are reserved for Switch. Do not use them in scripts.

Log: Showing progress

The messages of type "start", "progress" and "end" cause a task to be displayed in the Progress pane. The extra argument (which must be an integer) is used to indicate the total progress range and the current progress. For example, to indicate a taskconsisting of twelve more or less equal parts one might write:

```
e.log(4, "Task started", 12);
for (var i=0; i < 12; i++)
{
     e.log(5, "Task in progress", i);
     ...
}
e.log(6, "Task ended");
```

Where possible use the job.log() function instead, because it includes information about the job in the Progress pane.

Maintaining global data

The data accessed by these functions is stored in memory as part of the persistent state of the flow execution engine. Access to this data is serialized (to avoid conflicts caused by access in multiple threads) and the content persists across invocations of the flow engine.

It is the script programmer's responsibility to maintain distinct namespaces for their scripts (including those written by other programmers), and to determine the extent to which the data is global. This can be accomplished by providing an appropriate value for the scope argument in the access functions.

Note: These functions cannot be used to define metadata in Submit points and Checkpoints. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

setGlobalData(scope : String, tag : String, value : String, persistent : Boolean)

Sets the value of the global data with the specified scope and tag. The parameter "persistent" indicates whether global data value should be preserved between Switch restarts. When set to false, the value will be lost whenever Switch is restarted. By default this parameter is set to true.

getGlobalData(scope : String, tag : String) : String

Returns the value of a global data variable with the specified scope and tag, or returns an empty string if no global data variable with that scope and tag was set.

lockGlobalData(scope : String)

Acquires a global data lock on the specified scope so that multiple related get/set operations can be performed in the scope without interference from other scripts running in concurrent threads. (It is not necessary to lock global data for a single set or get operation or for multiple unrelated set and get operations).

This function blocks until the requested lock can be obtained.

A script can acquire only a single lock at a time, that is, nested locks are not allowed. While a lock is in effect, the script is allowed to access global data only in the scope on which a lock was acquired. Acquiring a nested lock (even on the same scope) or accessing data outside of the locked scope has undefined (and almost certainly undesirable) results.

unlockGlobalData( )

Releases the current global data lock, if any. Does nothing if there is no current lock.

Switch automatically releases the current lock, if any, after a script returns from the entry point. This is only a safeguard: a script should explicitly release a lock as soon as it is no longer needed.

removeGlobalData(scope: String, tag: String)

Removes the global data for the specified scope and tag. If the tag is not specified, the complete scope is removed. We recommend removing global data as soon as it is not needed anymore.

Note: It is safer to use non-persistent global data if it is not needed to save this data between Switch restarts.

The following table lists some examples for the value of the scope argument. The table assumes that the variable myOrg contains a URI (unique resource identifier) that uniquely identifies the programmer's organization, and that 's' is a variable which represents a Switch object.

Value of scope argument	The data can be accessed by
myOrg + s.getScriptName()	All copies of this script (even if used in different script elements in the same flow or in different flows)
s.getFlowName() + s.getElementName()	The copy of this script for this script element (if the same script is used twice, there is a separate copy of the data for each script element)

```
//setting and getting GlobalData
function jobArrived( s : Switch, job : Job ) {
    var scrname = s.getScriptName();
    //get a lock on the specified scope
    s.lockGlobalData("Enfocus" + scrname);
    //set a value
    s.setGlobalData("Enfocus" + scrname, "org", "Enfocus");
    //because of the lock, the value won't be changed
    s.log(1, s.getGlobalData("Enfocus" + scrname, "org"))//"Enfocus"
    //release lock
    s.unlockGlobalData();
}
```

Copying files

copy(source-path : String, destination-path : String) : Boolean

Copies the file or folder (including its contents, recursively) specified by the source path to the destination path. Both paths are absolute and include a filename; that is, the destination path specifies the filename of the copied file or folder, not the parent folder in which it should be placed.

This function creates all the folders of the destination-path if they do not already exist.

Files are copied with full support for platform-specific information such as Mac file types and resource forks (including emulations on Windows), as if copied by a regular flow operation under the control of Switch.

This function returns true if the file or folder is copied successfully; false otherwise.

Note: This function should be used only to copy files to a temporary location; use the Job.sendTo() functions for moving files to outgoing connections.

```
//example
//copy the jobfile to a temp location
var path =  job.createPathWithName("copy." + job.getExtension());
s.copy(job.getPath(), path);
//send to outgoing connection
job.sendToSingle(path);//only works for single outgoing connection
```

Moving files

move(source-path : String, destination-path : String, safeMove) : Boolean

Moves the file or folder (including its contents, recursively) specified by the source path to the destination path. Both paths are absolute and include a filename; that is, the destination path specifies the filename of the moved file or folder, not the parent folder in which it should be placed.

If safeMove is true, it will first copy the file or folder to a temporary location before deleting it from the original location. By default this parameter is set to true.

Files are moved with full support for platform-specific information such as Mac file types and resource forks (including emulations on Windows), as if moved by a regular flow operation under the control of Switch.

This function returns true if the file or folder is moved successfully; false otherwise.

Note:

- This function is relevant only if the source and destination folders are on different volumes (drives/partitions).
- This function should be used only to move files to a temporary location; use the Job.sendTo() functions for moving files to outgoing connections.

Getting special property values

The following functions allow retrieving the value of the special properties "ApplicationPath" and "ApplicationLicense" without requiring access to a Switch class instance. This is necessary in a scripted plug-in to access these properties from with the getApplicationLicensing entry point. See creating a scripted plug-in for more information on these special properties.

Note: These functions cannot be used to define metadata fields in Checkpoints and Submit points. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

getApplicationPath( ) : String

Returns the centrally managed absolute path to the third-party application associated with this scripted plug-in or null if no such path is available (yet).

getApplicationLicense( ) : String

Returns the centrally managed license key for the third-party application associated with this scripted plug-in or null if no such license key is available (yet).

Getting global user preferences

getServerName( ) : String

Returns the name of the Switch Server hosting this execution, as specified by the user in the communication preferences.

getLanguage( ) : String

Returns the name of the currently active language preference, in English. For example, the currently implemented languages are "English" and "German".

getLanguageEnvironment( ) : String

Returns the name of the currently active language environment preference, in English. For example, the currently implemented language environments are "Western" and "Japanese".

getLicenseSerial( ) : Number

Returns the serial number (max. 9 digits) associated with the license entered in the Switch licensing dialog. If multiple licenses are activated, the license with the longest license time period is used. This is always a positive integer, or zero if Switch has not been properly licensed.

This function cannot be used to define metadata fields in Checkpoints and Submit points. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

isInTrialMode( ) : Boolean

Returns true if Switch currently runs in trial mode; false otherwise.

This function cannot be used to define metadata fields in Checkpoints and Submit points. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

isLicenseFeatureEnabled(feature : String, version : Number) : Boolean

Returns true if Switch currently runs in trial mode or if the specified license feature with at least the specified version is currently enabled (because an appropriate license key has been successfully activated through the Switch licensing dialog); false otherwise.

This function cannot be used to define metadata fields in Checkpoints and Submit points. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

Note: This function is intended solely for internal use by Enfocus or a technology partner employing the Enfocus activation-based license server.

Getting system information

isWindows( ) : Boolean

Returns true if the calling script is running on Microsoft Windows, false otherwise.

isMac( ) : Boolean

Returns true if the calling script is running on Apple Mac OS X, false otherwise.

findRegisteredApplication(key1 : String, key2 : String) : String

Queries the system for an application that has been registered with one of the specified keys, and returns the absolute path of the application if found or null if not. The function verifies that the returned path is valid (that is, a file exists at the location); if not null is returned instead.

There are two key arguments so that a single call can specify the keys required for two different operating systems. The function automatically figures out which key to use on each operating system. The second key argument can be omitted if it is not needed.

On Windows, the function queries the system registry with the specified key, which must be an absolute registry path. Most professional application installers make a registry entry for this purpose but there is no guarantee.

Mac OS X automatically adds an entry to its application database when an application bundle is installed or copied to the system. Thus on Mac OS X the function queries the system application database with the specified key, which must be the filename of the application bundle (that is, the same name one would use to address the application from AppleScript) or the application's bundle identifier as provided in the property list located inside the application bundle (example: "com.adobe.distiller"). In the latter case, when multiple versions of the application are installed, Mac OS X will choose the most recent version.

This function cannot be used to define metadata fields in Checkpoints and Submit points. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

findApplicationOnDisk(name1 : String, name2 : String) : String

Searches the system folder in which applications are usually installed for an application with one of the specified file names and returns the absolute path of the application if found or null if not. There are two name arguments in case the application is named differently on different operating systems. The second name argument can be omitted if it is not needed.

On Windows the function recursively searches the contents of the Program Files (x86) folder. If the specified name has no extension, ".exe" is added.

On Mac OS X the function recursively searches the contents of the Applications folder. If the specified name has no extension, the function looks for the specified name AND for the name with ".app" added. The function does not search inside application bundles.

This function may use substantial resources (and time) to complete since it recursively searches the contents of a potentially large folder. Therefore, when called from the findApplicationPath entry point during Switch startup, this function is automatically disabled and immediately returns an empty string without searching.

This function cannot be used to define metadata fields in Checkpoints and Submit points. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

find64BitApplicationOnDisk(name1 : String, name2 : String) : String

Searches the system folder in which applications are usually installed for an application with one of the specified file names and returns the absolute path of the application if found or null if not. There are two name arguments in case the application is named differently on different operating systems. The second name argument can be omitted if it is not needed.

On Windows the function recursively searches the contents of the Program Files folder. If the specified name has no extension, ".exe" is added.

On Mac OS X the function recursively searches the contents of the Applications folder. If the specified name has no extension, the function looks for the specified name AND for the name with ".app" added. The function does not search inside application bundles.

This function may use substantial resources (and time) to complete since it recursively searches the contents of a potentially large folder. Therefore, when called from the findApplicationPath entry point during Switch startup, this function is automatically disabled and immediately returns an empty string without searching.

This function cannot be used to define metadata fields in Checkpoints and Submit points. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

getSpecialFolderPath(folderID : String) : String

Returns the absolute path to the special folder indicated by the folderID argument, which must be one of the strings listed in the table below. Returns null if folderID is unknown.

folderID	Returns an absolute path to
ApplicationData	The folder that serves as a common repository for application-specific data for the current user (that is, the user who launched Switch Server)
CommonApplicationData	The folder that serves as a common repository for application-specific data that is used by all users
PluginResources	The resource folder associated with the calling script; in the current implementation this is always the folder that contains the calling script package but implementations should not count in this fact The resource folder is intended for use by scripted plug-ins (see creating a scripted plug-in) and should not be used from regular script packages except for testing; there is no reliable way to transport external resources with regular script packages
ScriptData	The folder that serves as a common repository for custom "global" Switch script data; it is shared by all scripts (and all users) so callers must provide unique names for any files stored in this folder This is a subfolder of the Switch application data root, which has the advantage that its contents is relocated with all other information relevant to the operation of Switch, and can be easily located for diagnostics purposes. If such a folder does not exist, the function tries to create it. If for any reason the folder could not be created, the function returns an empty string (works for all interpreters) and throws an exception (works for the Javascript Interpreter).

The PluginResources and ScriptData constants cannot be used to define metadata fields in Checkpoints and Submit points. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

getSystemInfo(infoID : String) : String

Returns the system information indicated by the infoID argument, which must be one of the strings listed in the table below. Returns null if infoID is unknown.

infoID	Returns
UserName	The login name of the current user (i.e. the user who launched Switch Server)

supports64bit( ) : Boolean

Returns true if the calling script is running on a 64-bit capable operating system and processor, false otherwise.

Switch always runs in 64-bit mode. This function is provided solely to inform a script about operating system capabilities. For example, the script may use the information to select the appropriate version of a third-party application.

The Windows operating system comes in two distinct flavors (32-bit and 64-bit), so this function returns true only if it runs on a 64-bit Windows version.

Mac OS X supports a mix of 32-bit and 64-bit applications when running on a 64-bit capable processor, so this function returns true when running on a 64-bit capable processor. Note that all Intel-based Macs are 64-bit capable.

getServerVersion( ) : number

Returns the version of the Switch Server hosting this execution as a decimal number: "major version + update number / 100".

For example, Switch 08 update 6 returns the number 8.06.

This function does not differentiate between pre-release and release builds.

This function was introduced in Switch 08 update 6. If a script may be hosted by an older version of Switch, the script should check for the presence of this function before using it. For example:

```
if ("getServerVersion" in s && s.getServerVersion() >= 8.07) ...
```

Supporting time-out

getSecondsLeft( ) : Number

Returns the number of seconds left before the maximum allotted execution time ("abort processes after" in error handling preferences), is reached for this entry point. This allows a script to fail gracefully when it detects that it is getting close to being aborted, or to choose different algorithms based on the allotted time.

Switch provides no guarantees about the actual CPU time that will be available to the script (since many other tasks may be executing in parallel).

This function cannot be used to define metadata in Submit points and Checkpoints. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

Sleeping

sleep (seconds : Number)

Blocks the calling script for approximately the specified amount of time (in seconds). Other Switch processes are allowed to continue execution in parallel.

Compressing

Note:

- The "compress" and "uncompress" functions cannot be used to define metadata fields in Checkpoints and Submit points. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

- "compress" and "uncompress" are deprecated. We recommend using "archive" and "unarchive", which provide support for Unicode.

---

compress(source-path : String, destination-path : String, password : String, compress : Boolean) : Boolean

Places the source file or folder (including its nested contents) into a ZIP archive at the destination path, using the specified password. If the password is an empty string, the archive is not password-protected.

By default (that is, if the compress argument is true, missing or null), the files in the ZIP archive are compressed. If the compress argument is false, the files are stored in the ZIP archive without compression, dramatically reducing processing time. This can be meaningful when speed is more important than size or when the files will not compress well (example: JPEG images).

Returns true if successful, false if not (in which case an appropriate error message is logged).

This function behaves exactly as the compress tool (except that it does not add a unique name prefix to the output archive's name).

compress(source-path : String, destination-path : String, password : String, compress : Boolean, remove_existing_archive: Boolean) : Boolean

When

```
remove_existing_archive
```

is set to true, the zip file existing in the destination folder is removed and a new zip file is created.

If

```
remove_existing_archive
```

is set to false, the files are appended to the zip file existing in the destination folder.

For example,

1. compress("file1.txt", "result.zip", ... , remove_existing_archive="true");

   compress("file2.txt", "result.zip", ... , remove_existing_archive="true");

   As a result, the result.zip file contains only file2.txt.
2. compress("file1.txt", "result.zip", ... , remove_existing_archive="false");

   compress("file2.txt", "result.zip", ... , remove_existing_archive="false");

   As a result, the result.zip file contains both file1.txt and file2.txt files.

uncompress(source-path : String, destination-path : String, remove-redundant-folders : Boolean, passwords : String[ ]) : Boolean

Extracts all files from the archive at the source path and places them as a file or folder at the destination path. If the third argument is set to yes, all but the deepest subfolder levels are removed while uncompressing. The last (optional) argument provides a list of passwords for opening password-protected archives.

Returns true if successful, false if not (in which case an appropriate error message is logged).

This function only supports the ZIP archive format.

Note: The first correct password is used for all of the password protected files. If there are files with another password, those files will be skipped.

extract(archive-path : String, file-path : String, destination-path : String, passwords : String[ ]) : Boolean

Extracts the single file specified by the file-path argument from the source ZIP archive and places it at the destination path in the file system. The last (optional) argument provides a list of passwords for opening password-protected archives.

Returns true if successful, false if not (in which case an appropriate error message is logged).

Note that this function supports only the ZIP archive format; it is intended to extract a manifest from the archive without having to uncompress the complete archive.

Archiving

archive(source-path : String, destination-path : String, password : String, compress : Boolean) : Boolean

Places the source file or folder (including its nested contents) into a ZIP archive at the destination path, using the specified password. If the password is an empty string, the archive is not password-protected.

By default (that is, if the compress argument is true, missing or null), the files in the ZIP archive are compressed. If the archive argument is false, the files are stored in the ZIP archive without compression, dramatically reducing processing time. This can be meaningful when speed is more important than size or when the files will not compress well (example: JPEG images).

Returns true if successful, false if not (in which case an appropriate error message is logged).

This function behaves exactly as the compress tool (except that it does not add a unique name prefix to the output archive's name) but *adds support for Unicode*.

File names (of the files inside the ZIP archive) are UTF-8 encoded.

Very large files (+4 GB) can be archived as well.

archive(source-path : String, destination-path : String, password : String, compress : Boolean, remove_existing_archive: Boolean) : Boolean

When

```
remove_existing_archive
```

is set to true, the zip file existing in the destination folder is removed and a new zip file is created.

If

```
remove_existing_archive
```

is set to false, the files are appended to the zip file existing in the destination folder.

For example,

1. archive("file1.txt", "result.zip", ... , remove_existing_archive="true");

   archive("file2.txt", "result.zip", ... , remove_existing_archive="true");

   As a result, the result.zip file contains only file2.txt.
2. archive("file1.txt", "result.zip", ... , remove_existing_archive="false");

archive("file2.txt", "result.zip", ... , remove_existing_archive="false");

As a result, the result.zip file contains both file1.txt and file2.txt files.

archive(source-path : String, destination-path : String, password : String, compress : Boolean, remove_existing_archive: Boolean, nameInArchive: String) : Boolean

By default, the **nameInArchive** parameter is not specified, but if set, this parameter allows you to rename the contents of the archive.

Example:

```
archive(/"test.txt","/test.zip", ... , nameInArchive="newName.txt")
```

The file "test.txt" will be renamed to "newName.txt" before it is archived. The resulting archive will be "test.zip".

unarchive(source-path : String, destination-path : String, passwords : String[ ]) : Boolean

Extracts all files from the archive at the source path and places them as a file or folder at the destination path. The last (optional) argument provides a list of passwords for opening password-protected archives.

Returns true if successful, false if not (in which case an appropriate error message is logged).

This function supports the ZIP and RAR archive formats.

Very large archives (+4 GB) can be processed as well.

Note: The first correct password is used for all of the password protected files. If there are files with another password, those files will be skipped.

unarchive(source-path : String, destination-path : String, passwords : String[ ], moveResourceForkFiles : Boolean) : Boolean

Extracts all files from the archive at the source path and places them as a file or folder at the destination path.

If the moveResourceForkFiles parameter is set to true, after unpacking the archive at the destination path, Switch checks if there is a _MACOSX subfolder containing resource fork files. If so, the resource fork files are moved next to the corresponding data fork files and the _MACOSX folder is removed.

If moveResourceForkFiles is set to false, the _MACOSX subfolder (if there is one) is left untouched.

Note: This parameter is only valid on Windows. It's ignored on MAC OSX.

Downloading

download(source-url : String, destination-path : String) : Boolean

Downloads the file specified by the source URL to the destination path. The source URL must specify the 'http', 'https', 'ftp' or 'sftp' protocol, including login and password if applicable. The destination path must include a filename; that is, the it specifies the filename of the copied file, not the parent folder in which it should be placed.

The passive mode is used for ftp protocol. If defined, the FTP proxy preferences are taken into account.

For 'http' and 'https' protocols, the source URL must be URI-encoded.

Returns true if the operation succeeded, false if not. Only one download attempt is made.

This function cannot be used to define metadata fields in Checkpoints and Submit points. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

Note: This function should be used only to place files in a temporary location; use the Job.sendTo() functions for moving files to outgoing connections.

```
//example
//download and save enfocus logo
var dpath = job.createPathWithName("logo.png");
var link = "http://www.enfocus.com/~/media/Enfocus"
+ "/Images/Logos/lgo_enfocus.png";
s.download(link, dpath);
job.sendToSingle(dpath); //only works for single outgoing connection
```

Note: The download method uses server proxy settings for both HTTP and FTP schemes (connections).

Temporary location

createPathWithName(name : String, createFolder : Boolean) : String

This function has the same semantics as the Job.createPathWithName() function. It is provided as a member of the Environment class so that one can obtain a location for storing temporary files or folders without having access to a job (for example, in the timerFired entry point).

Examples: see other samples above.

If for any reason the required path could not be created, the function returns an empty string (works for all interpreters) and throws an exception (works for the Javascript Interpreter).

Client connection

getClientConnection( ) : ClientConnection

Returns the valid ClientConnection object if the script code is executed in the context of a client connection, otherwise null is returned.

Note: The ClientConnection object is available only in the context of a client connection. If there is a client connection which triggered evaluation of a script, then this script returns a valid ClientConnection object after calling this method. In all other cases, the method getClientConnection returns null. This means that the ClientConnection currently can be used only in JavaScript script expressions specified in Metadata properties of Submit point and Checkpoint flow elements.

See also *ClientConnection class* on page 225.

Accessing the file statistics

createFileStatistics (path : String) : FileStatistics

Returns the file creation statistics. For more info, see the *FileStatistics Class*.

5.7.4. Switch class

The single instance of the Switch class is passed as an argument to the script entry points that operate in the context of a particular flow element. By convention, the name for the Switch object is "s" (although the script developer can choose to use another name).

Since the Switch class inherits from the Environment class, any of the functions described for that class can be called for the Switch class as well. In other words, you can call the Environment class functions with "e.function()" or "s.function()" depending on the argument passed to the entry point.

Note: Most functions belonging to the Switch class *cannot* be used to define metadata fields in Submit points and Checkpoints. Exceptions are: getFlowName(), getElementName(), log(), getAvailableCodecs(), getElementID() and getCounter(). For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

Accessing the flow definition

See script element, script declaration and main script properties for background information.

getFlowName( ) : String

Returns the name of the flow, as displayed in the Flows pane, in which this script resides.

getElementName( ) : String

Returns the name of the flow element associated with this script as displayed in the canvas.

getElementID ( ) : String

Returns a string that uniquely identifies the connection (within the limits of the guarantees described below). Callers should not rely on the syntax of the returned string as this may change between Switch versions.

The element ID for a particular flow element offers the following fundamental guarantees:

- It differs from the element ID for any other flow element in the same active flow.
- It remains unchanged as long as the flow in which it resides is not edited, even if the flow is deactivated and reactivated and across Switch sessions.

Note that holding or releasing connections or renaming the flow does not count as editing in this context. However exporting and re-importing (or upgrading) a flow, or renaming a flow element inside the flow, does count as editing.

Note: getElementID () is deprecated. We recommend using getElementUniqueID ( ) instead.

getElementUniqueID ( ) : String

Returns a string that uniquely identifies the flow element associated with this script (within the limits of the guarantees described below). Callers should not rely on the syntax of the returned string as this may change between Switch versions.

The element ID for a particular flow element offers two fundamental guarantees:

- It differs from the element ID for any other flow element in any currently active flow.
- It remains unchanged as long as the flow in which it resides is not edited, even if the flow is deactivated and reactivated and across Switch sessions.

Note that holding or releasing connections or renaming the flow does not count as editing in this context. However, exporting and re-importing (or upgrading) a flow, or renaming a flow element inside the flow, does count as editing.

getScriptName( ) : String

Returns the name of this script as defined in the script declaration.

getInConnections( ) : ConnectionList

Returns a list of Connection instances representing the incoming connections forthe flow element associated with this script. The list is in arbitrary order. If there are no incoming connections the list is empty.

getOutConnections( ) : ConnectionList

Returns a list of Connection instances representing the outgoing connections for the flow element associated with this script. The list is in arbitrary order. If there are no outgoing connections the list is empty.

getOutgoingName( ) : String [ ]

Returns a list of names for the outgoing connections in alphabetical order. If a connection name is empty the target folder name is used as a fall-back. If the folder name is empty as well, the connection is not listed.

getAvailableCodecs( ) : String [ ]

Returns a list of available codecs.

getCounter(Width : Number, ID : String, Update : Boolean) : String

Returns the value of the Switch counter.

This method increments the counter value for the specified ID by one and returns the resulting string with a fixed number of digits specified in the Width argument. The method gives the same result as [Switch.Counter: Width="Width", Id="ID"]. For more information, refer to 'Known variables' in the Switch Reference Guide.

For example, if the counter with ID = "test" has never been used before, then the result of s.getCounter(5, "test", true) will be "00001". Next time, it will be "00002" and so forth. After the counter reached "99999", it circles back to "00000".

The *Width* argument specifies the number of decimal digits in the number; leading zeroes are inserted when needed. The default value for Width is 5; the minimum value is 1 and the maximum value is 15. An invalid value is replaced by the nearest valid value. The internal counter is always 15 digits wide; the Width argument specifies how many of the right-most digits are actually shown.

The *ID* argument is used to refer to a particular physical counter. Switch uses the same physical counter for all counters with the same ID, even if these counters are used in properties of a different flow element or in a different flow (in the same Switch instance). If the ID argument has a non-empty value, this explicitly specifies the ID. Otherwise a default ID is generated that is unique to the flow element in which the counter is used.

Update is an optional argument, by default set to true. It indicates whether or not the counter value should be updated in the internal data storage.

Accessing injected properties

The functions in this section retrieve information about the properties injected into the script element as defined in the script declaration (see *Property definition properties*). The property value is entered by the user in the Properties pane; the property tags are defined in the script declaration.

Property values for which isPropertyValueStatic() returns false can be computed only in the context of a particular job. By default the functions getPropertyValue() and getPropertyValueList() use the current job (that is, the job for which the jobArrived entry point was invoked) to compute property values. Thus in most cases the job argument can be omitted. In situations however where there is no current job (for example: because this function is called

from within the timerFired entry point), it is necessary to specify the job argument when getting non-static property values.

getPropertyValue(tag : String, job : Job) : String

Returns the value of the injected flow element property with the specified tag for the script element associated with this script. If the specified tag is not defined in the script declaration, the function returns null. If the property has a string list value, this function returns the first string in the list.

If this function is invoked while the flow is inactive, it returns the source value of a property that contains a variable or a script expression instead of its computed value (that is, the variable or script expression definition rather than its evaluation result).

The optional job argument specifies the job for which the property value is computed. If the argument is null or missing, the current job is used instead. If the argument is null or missing, and there is no current job, and the property value is non-static, this function returns null.

In case the property was set using the external editor of the Switch file format, the getPropertyValue returns a single path written in Switch format XML. See also *External property editor* on page 56.

getPropertyValueList(tag : String, job : Job) : String[ ]

Returns the string list value of the injected flow element property with the specified tag for the script element associated with this script. If the specified tag is not defined in the script declaration, the function returns null. If the property has a single-string value, this function returns a list with a single string.

When the main editor for the property is "multi-line text with variables" and a script expression is used, the return value is an array with just one element, even when the script expression returns an array! It is therefore recommended to make sure the return value from the script expression editor and the handling of the return value in the script are in sync. The writer of the script expression should return a joined array (see *Associating a script expression with a property* on page 9). The script writer should then handle the returned value in the script like this:

```
var userValues = s.getPropertyValueList("StringList");
if (s.getPropertyEditor("StringList") == "Script expression") {
userValues = userValues[0].split("\n");
}
```

This will of course only work correctly when the writer of the script expression also followed the convention described here and in the topic on associating a script expression with a property.

If this function is invoked while the flow is inactive, it returns the source value of a property that contains a variable or a script expression instead of its computed value (i.e. the variable or script expression definition rather than its evaluation result).

The optional job argument specifies the job for which the property value is computed. If the argument is null or missing, the current job is used instead. If the argument is null or missing, and there is no current job, and the property value is non-static, this function returns null.

In case the property was set using the external editor of the Switch file format, the getPropertyValue returns a list of paths written in Switch format XML. See also *External property editor* on page 56.

getPropertyEditor(tag : String) : String

Returns the name of the property editor that was used to enter the value for the specified property. In case a property offers multiple editors, this information may be required to

correctly interpret the property value. If the specified tag is not defined in the script declaration, the function returns null.

The function returns the property editor's name exactly as it appears in the tables in inline property editors and other property editors. For an overview of the available editors, refer to *Flow element update* on page 172.

Example:

```
var theFilePath;
var theEditor = s.getPropertyEditor( "theProperty" );
var theValue = s.getPropertyValue( "theProperty" );
if( theEditor == s.EditorLiteral )
{
 if( theValue == "Automatic" )
 {
  s.log( 2, "Automatic file path" );
  // set some automatic value for theFilePath
 }
 else if ( theValue == "Default" )
 {
  s.log( 2, "Default file path" );
  // set some default value for theFilePath
 }
}
else
{
 theFilePath = theValue;
}
```

isPropertyValueStatic(tag : String) : Boolean

Returns true if the value for the specified property is guaranteed not to change between entry point invocations for this flow element as long as the containing flow remains active; false otherwise. If the specified tag is not defined in the script declaration, the function returns null.

This function is useful to determine whether an invalid property value should result in calling Job.failProcess() (the value is static) or Job.fail() (the value is dynamic).

Specifically, the function returns false if one of the following conditions holds:

- The property value was entered with the property editor "Script expression" or "Condition with variables".
- The property value was entered with the property editor "Single-line text with variables" or "Multi-line text with variables" and the text does indeed contain a syntactically valid variable.
- The property offers the property editor "Single-line text with variables" or "Multi-line text with variables", and the property value was entered with the inline property editor "Single-line text" instead, and the text does indeed contain a syntactically valid variable.

isPropertyValueActual(tag : String) : Boolean

Returns true if the getPropertyValue() or getPropertyValueList() functions return an actual value for the specified property, in other words if the property value is static or if the flow is active so that the property value can be computed.

Returns false if the getPropertyValue() or getPropertyValueList() functions return the source value for the specified property, in other words if the property value is not static and the flow is inactive.

If the specified tag is not defined in the script declaration, the function returns null.

Working with jobs and processes

getJobs( ) : JobList

Returns a list of Job instances representing all of the jobs currently waiting in the input folders for the flow element. The list is in arbitrary order and includes all jobs that have "arrived" in the jobArrived entry point, and including the job passed to the current invocation of the jobArrived entry point. If there are no such jobs, the list is empty.

getJobsForConnection(c : Connection) : JobList

Returns a list of Job instances representing all of the jobs currently waiting in the input folders for the specified incoming connection. Same semantics as for the getJobs function apply.

Note: From Switch 13 onwards, this call is deprecated. Please use getJobs( ) instead.

createNewJob(origin : String) : Job

Returns a Job instance representing a new job that does not correspond to an incoming job. The job is given a fresh job ticket with default values. The file path associated with the job is the empty string.

A new job is needed only in cases the script generates or imports new jobs into a flow. In all other cases the script should use (one of) the incoming job(s) related to the output so that job ticket information is preserved.

The optional argument provides an indication of the origin of the job before it was injected in the flow, for example its absolute file path or location in a third-party data base. This value is written in the origin attribute of the 'Produced' occurrence in the newly created job ticket (see job occurrence trail). If the origin argument is missing or null, the empty string is used.

```
//creates a new file each timer interval
//and sends it to outgoing connection
function timerFired( s : Switch ) {
    //create new job
    var job = s.createNewJob();
    var f = new File(job.getPath());
    f.open(File.WriteOnly);
    f.writeLine("Sample line.");
    f.close();

    //single outgoing connection only
    job.sendToSingle(job.getPath());
}
```

failProcess(message : String, extra : String or Number)

Logs a fatal error with the specified message and puts the flow element instance in the "problem process" state, without providing the context of a particular job.

This function should be used only when there is no appropriate job context, for example in the timerArrived entry point of a Producer. In all other cases, use the Job.failProcess() function instead so that the job causing the problem is properly retried when the process is retried.

See the description of the Environment.log() function for more information on the message and extra arguments. See viewing problem status for more information on problem processes.

Accessing the timer interval

setTimerInterval(seconds : Number)

Sets the interval, in seconds, between subsequent invocations of the timerFired entry point (if it has been declared in the script). The implementation guarantees only that the time between invocations will not be less than the specified number of seconds. Depending on run-time circumstances the actual interval may be (much) longer. The default value for the timer interval is 300 seconds (5 minutes).

Example see *Regular execution*.

getTimerInterval( ) : Number

Returns the interval, in seconds between subsequent invocations of the timerFired entry point.

Note: If setTimerInterval() is not used, getTimerInterval() is set to "0", which is the default interval of 300 seconds (5 minutes).

Example see *Regular execution*.

isDeactivating ( ) : Boolean

Returns true if the flow in which this script resides is attempting to deactivate at the time of invocation; false otherwise.

When the user deactivates a flow (or quits the server) Switch waits until all executing entry points have run to completion. Therefore, an entry point that potentially executes for a long time, for example because it is waiting for an external event, can invoke this function at regular intervals to determine whether it should abort its operation.

Webhooks

The Switch object offers the following functions:

webhookSubscribe(requestMethod : String, path : String, protocolType : String)

Subscribes to events about *requestMethod* requests coming to *path* over *protocolType*.

If protocolType is missing or null, the default value *WebhookRequest.Any* is used. It can only be called from the webhookTriggered entry point. If subscribing fails for some reason (for example, if HTTPS is requested but it is disabled in the Switch Preferences), it throws an exception.

Allowed values for the requestMethod argument are: "POST", "PUT" and "DELETE".
It is recommended to use the pre-defined constant strings WebhookRequest.POST, WebhookRequest.PUT and WebhookRequest.DELETE. If any other value is used, the function will throw an exception.

Allowed values for the protocolType argument are: "Any" (default), "HTTP" and "HTTPS".
It is recommended to use the pre-defined constant strings WebhookRequest.Any, WebhookRequest.HTTP and WebhookRequest.HTTPS. Using WebhookRequest.Any means "any protocol enabled in the Switch Preferences", so requests arriving via both HTTP and HTTPS will be delivered (assuming both protocols are enabled in the Switch Preferences). If any other value is used, the function will throw an exception.

Example:

```
try {
            s.webhookSubscribe(WebhookRequest.POST, "/myscript/notify");
        }
        catch(e) {
            s.log(3, "Failed to subscribe to the request: %1", String( e ));
            return;
        }
```

webhookUnsubscribe(requestMethod : String, path : String, protocolType : String)

Cancels a subscription.

The arguments must have the same values as used for the corresponding subscribe call. If there is no matching active subscription found, the function will throw an exception. Note that all subscriptions for the script instance are cancelled automatically when the flow is stopped.

enableCustomWebhookResponse(requestMethod : String, path : String, protocolType : String, args : Array)

Enables the custom responses constructed using the *constructWebhookResponse* entry point for incoming webhook *requestMethod* requests coming to *path* over *protocolType*. The optional *args* array will be passed to the entry point each time a matching request arrives. If *protocolType* is missing or null, the default value *WebhookRequest.Any* is used.

This function can only be called from the *webhookTriggered* entry point. If enabling fails for some reason (for example, if HTTPS is requested but it is disabled in the Switch Preferences, or *enableCustomWebhookResponse* is not implemented), it throws an exception.

Allowed values for the *requestMethod* argument are: "GET", "POST", "PUT" and "DELETE". It is recommended to use the pre-defined constant strings *WebhookRequest.GET, WebhookRequest.POST, WebhookRequest.PUT* and *WebhookRequest.DELETE*. If any other value is used, the function will throw an exception.

Note: *WebhookRequest.GET* is allowed only for *enableCustomWebhookResponse*; it cannot be used in *webhookSubscribe*.

Allowed values for the *protocolType* argument are: "Any" (default), "HTTP" and "HTTPS". It is recommended to use the pre-defined constant strings *WebhookRequest.Any, WebhookRequest.HTTP* and *WebhookRequest.HTTPS*. Using *WebhookRequest.Any* means "any protocol enabled in the Switch Preferences", so requests arriving via both HTTP and HTTPS will be handled (assuming both protocols are enabled in the Switch Preferences). If any other value is used, the function will throw an exception.

If the moment *enableCustomWebhookResponse* is called the custom responses for that combination of *requestMethod, path* and *protocolType* arguments are already enabled by this or some other script, the function will throw an exception.

Note: The paths which start from "/reserved/" are reserved by Switch for internal use and cannot be used in the *path*argument.

disableCustomWebhookResponse(requestMethod : String, path : String, protocolType : String)

Disables the custom responses for the specified incoming webhook requests. The arguments must have the same values as used for the corresponding enable call. If no matching enabled custom responses are found, the function will throw an exception. Note that all custom responses enabled by a script instance are cancelled automatically when the flow is stopped.

webhookSubscribe(requestMethod : String, path : String, protocolType : String, enableCustomResponse : Boolean, args : Array)

If the optional argument *enableCustomResponse* is true, the method also enables the custom responses constructed using entry point *constructWebhookResponse* for incoming webhook *requestMethod* requests coming to *path* over *protocolType*. The optional *args* array will be passed to the entry point each time a matching request arrives. If *enableCustomResponse* is missing or null, the default value false is used.

Note:

- The paths which start from "/reserved/" are reserved by Switch for internal use and cannot be used in the *path*argument.
- It is not possible to use GET method in *webhookSubscribe.*

See also the description of the *enableCustomWebhookResponse* call (higher).

webhookUnsubscribe(requestMethod : String, path : String, protocolType : String, disableCustomResponse : Boolean)

If the optional argument *disableCustomResponse* is true, the method also disables the custom responses for the specified incoming webhook requests. If *disableCustomResponse* is missing or null, the default value *true* is used.

See also the description of the *disableCustomWebhookResponse* call (higher).

5.7.5. Connection class

An instance of the Connection class represents an incoming or outgoing connection for the flow element associated with the script (see connection and script element for background information). Connection objects can be obtained through functions of the Switch class.

Note: Functions belonging to the Connection class cannot be used to define metadata in Submit points and Checkpoints. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

Accessing the flow definition

getName( ) : String

Returns the name of the connection as displayed in the canvas; this may be an empty string.

getElementID ( ) : String

Returns a string that uniquely identifies the connection (within the limits of the guarantees described below). Callers should not rely on the syntax of the returned string as this may change between Switch versions.

The element ID for a particular flow element offers the following fundamental guarantees:

- It differs from the element ID for any other flow element in the same active flow.
- It remains unchanged as long as the flow in which it resides is not edited, even if the flow is deactivated and reactivated and across Switch sessions.
- The element ID for a connection is never equal to the element ID for a non-connection flow element.

Note that holding or releasing connections or renaming the flow does not count as editing in this context. However exporting and re-importing (or upgrading) a flow, or renaming a flow element inside the flow, does count as editing.

Note: getElementID () is deprecated. We recommend using getElementUniqueID ( ) instead.

getElementUniqueID ( ) : String

Returns a string that uniquely identifies the flow element associated with this script (within the limits of the guarantees described below). Callers should not rely on the syntax of the returned string as this may change between Switch versions.

The element ID for a particular flow element offers two fundamental guarantees:

- It differs from the element ID for any other flow element in any currently active flow.
- It remains unchanged as long as the flow in which it resides is not edited, even if the flow is deactivated and reactivated and across Switch sessions.

Note that holding or releasing connections or renaming the flow does not count as editing in this context. However, exporting and re-importing (or upgrading) a flow, or renaming a flow element inside the flow, does count as editing.

getConnectionType( ) : String

Returns the type of the connection as one of the following strings: "Move", "Filter", "Traffic-data", Traffic-log", "Traffic-datawithlog".

allowsSuccess ( ) : Boolean

Returns true if this is a traffic-light connection and the Success property is set to yes; otherwise returns false.

allowsWarning ( ) : Boolean

Returns true if this is a traffic-light connection and the Warning property is set to yes; otherwise returns false.

allowsError ( ) : Boolean

Returns true if this is a traffic-light connection and the Error property is set to yes; otherwise returns false.

isOnHold( ) : Boolean

Returns true if the connection is currently on hold, false if it is not.

getFolderName( ) : String

Returns the name of the folder at the other end of this connection (the originating folder for an incoming connection, the target folder for an outgoing connection) as displayed in the canvas; this may be an empty string.

Accessing injected properties

The connection class offers the same functions for accessing injected properties as the Switch class; the function descriptions are not repeated here.

Only outgoing connections have injected properties.

getPropertyLocalizedName(tag : String) : String

Returns the name of the property as visible in the user interface. The name is localized, i.e. it is translated to the language that is currently used by the Switch. This method is mostly intended for getting property name to be included into a log message.

hasProperty(tag : String) : Boolean

Returns true if the flow element has the property with the specified tag, else returns false.

Accessing files at the other end of the connection

getFileCount(nested : Boolean) : Number

Returns the number of files currently residing in the folder at the other end of this connection (the originating folder for an incoming connection, the target folder for an outgoing connection). If nested is false, only items directly inside the folder are counted (i.e. each file and each subfolder is counted as one item). If nested is true, the number of files in subfolders are counted as well, recursively (and subfolders themselves do not contribute to the count).

getByteCount( ) : Number

Returns the size in bytes of the contents currently residing in the folder at the other end of this connection (the originating folder for an incoming connection, the target folder for an outgoing connection). Files in subfolders are included in the count, recursively.

Note: The getFileCount() and getByteCount() functions tally any and all files that are reported by the file system to reside in the connection's folder at the instant of the function's invocation, including files that have not yet fully arrived, and (for outgoing connections) including files that have been placed in the target folder by other processes. As a consequence, the return value of these functions may vary between subsequent invocations as files are added/removed/updated under or outside of the script's control.

Example

```
function jobArrived( s : Switch, job : Job )
{
    var connlist = s.getInConnections();
    for(var i = 0; i < connlist.getCount(); i++){
        var conn = connlist.at(i);
        var name = conn.getName();
        var id = conn.getElementID();
        var fcount = conn.getFileCount(false);
        var bcount = conn.getByteCount();
        var type = conn.getConnectionType();
        var fname = conn.getFolderName();
        job.log(1, name);
        job.log(1, id);
        job.log(1, fcount);
        job.log(1, bcount);
        job.log(1, type);
        job.log(1, fname);
    }
}
```

5.7.6. Job class

An instance of the Job class represents a job (file or job folder) waiting to be processed in one of the input folders for the flow element associated with the script (see working with job folders and internal job tickets for background information). Job objects can be obtained through functions of the Switch class.

The second argument passed to the jobArrived entry point is the newly arrived job that triggered the entry point's invocation. This is commonly called the current job. By convention, the name for the object representing the current job is "job" (although the script developer can choose to use another name).

Processing a job

Processing a job in a script usually consists of the following steps:

- Decide on how to process the job based on file type etc.
- Get the path to the incoming job.
- Get a temporary path for one or more output files or folders.
- Create the output(s) in the temporary location.
- Call one of the sendTo functions for each output.

If the incoming job is passed along without change the Job.sendTo() functions can be called directly on the incoming job path, skipping all intermediate steps.

Based on the above scenario, Switch automatically handles all complexities:

- Moving and copying jobs to the appropriate output connections.
- Providing outgoing jobs with a unique name prefix.
- Creating and updating the related internal job tickets.
- Removing the incoming job and any temporary files created along the way.
- Logging the appropriate messages.

A job remains in the input folder until one of the Job.sendTo() or Job.fail() functions has been called for the job. The jobArrived entry point will be invoked only once for each job, so if the entry point does not call a sendTo() or fail() function for the job, the script should do so at a later time (in a timerFired entry point or in a subsequent invocation of the jobArrived entry point for a different job).

Note: After Switch Server quits and restarts, the jobArrived entry point will be invoked for all jobs in the input folder once again.

Getting job file/folder information

getPath( ) : String

Returns the absolute file or folder path for the job as it resides in the input folder, including unique filename prefix.

getUniqueNamePrefix( ) : String

Returns the unique filename prefix used for the job, without the underscores. For example, for a job called "_0G63D_myjob.txt" this function would return "0G63D".

getName( ) : String

Returns the file or folder name for the job, including filename extension if present, but excluding the unique filename prefix.

getNameProper( ) : String

Returns the file or folder name for the job excluding filename extension and excluding the unique filename prefix.

getExtension( ) : String

Returns the job's filename extension, or an empty string if there is none.

getMacType( ) : String

Returns the job's Mac file type code as a 4-character string if available, otherwise an empty string.

getMacCreator( ) : String

Returns the job's Mac creator code as a 4-character string if available, otherwise an empty string.

isType(ext : String) : Boolean

Returns true if the job matches the specified file type, specified as a filename extension, and false otherwise.

A file matches if its filename extension and/or its Mac file type (after conversion) match the specified filename extension. A folder matches if any of the files at the topmost level inside the folder match the specified type. These semantics are similar to those of matching cross-platform file types in filter connections.

isFile( ) : Boolean

Returns true if the job is a single file, false otherwise.

isFolder( ) : Boolean

Returns true if the job is a folder, false otherwise.

getFileCount( ) : Number

Returns the number of files in the job. If it is a single file, the function returns 1. If it is a job folder, the function returns the number of files in the job folder and any subfolders, recursively (folders and subfolders themselves do not contribute to the count).

getByteCount( ) : Number

Returns the size in bytes of the job. If it is a job folder, all files in subfolders are included in the count, recursively.

Getting temporary output paths

createPathWithName(name : String, createFolder : Boolean) : String

Returns an absolute path to a writable temporary location in the file system with the specified filename (which should include the filename extension if one is required).

If the optional createFolder argument is true, the function creates a new folder at the location indicated by the returned path. The caller can use this folder as the root of an output job folder or just as a location to store temporary files, one of which may become an output file.

If the optional createFolder argument is false or missing, the function does not create a file or folder at the location indicated by the returned path (but the parent folder is guaranteed to exist). The caller can use the path to create an output file or a temporary file.

The returned path is guaranteed to differ between invocations of the function, even for the same job and with identical argument values.

Also, after the entry point returns, Switch deletes any file or folder left at any of the paths created using this function.

If for any reason the required path could not be created, the function returns an empty string (works for all interpreters) and throws an exception (works for the Javascript Interpreter).

createPathWithExtension(ext : String, createFolder : Boolean) : String

Same as above but uses the filename of the job after substituting the specified extension (rather than replacing the complete filename). If the specified extension is an empty string, any trailing dot is removed from the filename; this helps creating a folder path without extension.

If for any reason the required path could not be created, the function returns an empty string (works for all interpreters) and throws an exception (works for the Javascript Interpreter).

```
//create paths
//create temporary folder 'sampleFolder'
var folderPath = job.createPathWithName("sampleFolder", true);

//if job = 'text.txt', wextPath ends with 'texttxt' (filepath)
var wextPath = job.createPathWithExtension("");
var f = new File(wextPath);
//...do something with f

//if job = 'text.txt', wextPath2 ends with 'texttxt' (folderpath)
var wextPath2 = job.createPathWithExtension("", true);
```

Sending jobs to outgoing connections

The semantics for sending files to outgoing connections depend on the connection type. However all outgoing connections of a flow element (and thus a script) have the same type, and this type is defined in the script declaration.

To successfully complete processing of a job, the script must call exactly one sendTo() function for each generated output file/folder (i.e. there may be multiple sendTo() calls for the same job). If there is no output, the script must call the sendToNull() function. It is allowed to defer these calls to a subsequent entry point invocation (for example, when grouping multiple incoming jobs in a job folder).

If a fatal error occurs, the script must call one of the fail() functions as appropriate. Calling a fail() function cancels any and all previous sendTo() function calls for the same job during the same entry point invocation.

Once a fail() function has been called for a job, any further calls to fail() or sendTo() functions for the same job during the same entry point invocation are ignored (such calls just log a warning message rather than performing the requested operation).

Guidance to generating output jobs

Use Case	Guidance on generating output jobs
One to many	To deliver multiple output jobs triggered by or associated with an incoming job, repeatedly call the sendTo() function on the input job rather than using the createNewJob() function; this ensures that all output jobs correctly inherit the job ticket of the originating job This guideline holds even if the output job is a totally different file; for example, a preflight report for a PDF file or a file selected from a database depending on the contents of an incoming xml file
Many to one	Leave jobs in the input folder (by not calling any sendTo() functions) until you have everything to generate a complete output job; then call sendTo() on the "primary" input job and sendToNull() on all other input jobs related to this output job – all in a single entry point invocation The output job inherits only the job ticket of the primary input job; if this is unacceptable you'll have to merge metadata from the other input jobs in a meaningful way (similar to the built-in job assembler)
Many to many	In most situations this is simply a combination of the previous two use cases However if it is not possible to generate all output jobs during the same entry point invocation, you need to call setAutoComplete(false) on the input job so that it is not automatically removed from the input folder at the end of the entry point invocation; this is an extremely rare situation so in most cases you don't need to worry about preventing auto-completion
None to any	You need the createNewJob() function only when there really is no originating job, for example when you eject a new output file based on a timer or an external event that is not associated with a job already in the flow
Any to none	Whenever you're ready with an input job, call sendToNull() on it

SendTo functions

In their "path" argument the sendTo() functions expect to be passed the absolute path of the file or folder that should be sent along. This could be any path:

- The path of the incoming job file/folder as returned by Job.getPath().
- One of the temporary paths returned by Job.createPathWithName/Extension().
- A file inside a folder located at one of the temporary paths, which is useful in case the script can control the location but not the name of the output file to be sent along (because some other process determines it).
- Any other path, which is useful in case the script can't control the location of the output file to be sent along (because some other process determines it).

In their optional "name" argument the sendTo() functions expect to be passed the filename for the output file or job folder (including filename extension; a path portion and/or a unique name prefix are ignored). If the "name" argument is an empty string or missing, the filename in the "path" argument is used instead.

The sendTo() functions automatically insert or replace the unique name prefix as appropriate, and they move or copy files as needed.

After the entry point returns, Switch removes all jobs for which one or more sendTo() functions were called from the input folder, and it deletes any file or folder left at any of the paths passed to any of the sendTo() functions. Thus a script should never call sendTo() on files that must be preserved. For example, a script that injects a file from an asset management system into a flow should copy the file to a temporary location and then call sendTo() on the copy.

Note: SendTo functions *cannot* be used to define metadata in Submit points and Checkpoints. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

sendToNull(path : String)

Marks the job as completed without generating any output. The path is ignored other than for marking the indicated file/folder for deletion.

sendToSingle(path : String, name : String)

Sends a file/folder to the single outgoing move connection. If the flow element has outgoing connection(s) of another type or if it has more than one move connection this function logs an error and does nothing.

sendToData(level: Number, path : String, name : String)

Sends a file/folder to any and all outgoing traffic-light data connections that have the specified level property enabled (success = 1, warning = 2, error = 3). If the flow element has outgoing connection(s) of another type this function logs an error and does nothing. If the flow element has no outgoing data connections of the specified level, fail() is invoked instead with an appropriate error message.

sendToLog(level: Number, path : String, name : String, model : String)

Sends a file/folder to any and all outgoing traffic-light log connections that havethe specified level property enabled (success = 1, warning = 2, error = 3). If the flow element has outgoing connection(s) of another type this function logs an error and does nothing. If the flow element has no outgoing log connections of the specified level, nothing happens (i.e. the log file is discarded).

The model argument is used only in case the log file is sent over a "Data and log" connection as a metadata dataset attached to the job. In that case the value of model determines the data

model of the dataset ("XML", "JDF", "XMP" or "Opaque"). If the argument is nil or missing or if it has an unsupported value, the data model is set to "Opaque".

sendToFilter(path : String, name : String)

Sends a file/folder to all outgoing connections with a file filter that matches the file/folder being sent. If there is no such connection, fail() is invoked instead with an appropriate error message.

The flow element must have outgoing filter connections without folder filter properties, otherwise this function logs an error and does nothing.

sendToFolderFilter(foldernames : String[ ], path : String, name : String)

Similar to sendToFilter() but also honors the folder filter properties on the outgoing connections, using the list of folder names passed in the first argument.

The flow element must have outgoing filter connections with folder filter properties, otherwise this function logs an error and does nothing.

sendTo(c : Connection, path : String, name : String)

Sends a file/folder to the specified outgoing connection, regardless of connection type.

Fail functions

See the description of the Environment.log() function for more information on the message and extra arguments.

A script should invoke failProcess() rather than fail() or failRetry() when it encounters an error condition that does not depend on the job being processed but rather on the process itself (for example, a network resource is not available). In such a case all subsequent jobs would fail as well (because the process is broken) and moving them all to the problem jobs folder makes no sense. It is much more meaningful to hold the jobs in front of the broken process and retry the process from time to time. See viewing problem status for more information on problem jobs and processes.

Note: Fail functions *cannot* be used to define metadata in Submit points and Checkpoints. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

fail(message : String, extra : String or Number)

Logs a fatal error for the job with the specified message and moves the job to the problem jobs folder.

failAndRetry(message : String, extra : String or Number)

Similar to fail() but requests that Switch retries processing the job if this was the first attempt at processing it. This is useful if an external application or resource produced an error that may go away when retrying the job a second time. Switch keeps track of the retry count, so the script doesn't need to worry about it. If the job fails a second time it will be moved to the problem jobs folder.

failProcess(message : String, extra : String or Number)

Logs a fatal error for the job with the specified message and puts the flow element instance in the "problem process" state. The job is not affected (i.e. it stays in the input folder) and a new jobArrived event will be generated for the job when the process is retried.

Logging

Switch automatically issues the appropriate log messages when a script invokes the one of the Job.sendTo() or Job.fail() functions. A script can call the function described below to log additional job-related messages.

log(type : Number, message : String, extra : String or Number)

Logs a message of the specified type for this job, automatically including the appropriate job information.

See the description of the Environment.log() function for more information on the arguments.

```
job.log(1, "This is %1", "an info message");//"This is an info message"
```

Getting job ticket info

See using hierarchy info, using email info, viewing job state statistics, and configuring users for background information.

getHierarchyPath( ) : String[ ]

Returns an Array with the location path segments in the hierarchy info associated with the job, or an empty array if there is no hierarchy info. The topmost path segment is stored at index 0.

getEmailAddresses( ) : String[ ]

Returns an Array with the email addresses in the email info associated with the job, or an empty array if there are none.

getEmailBody( ) : String

Returns the email body text in the email info associated with the job, or an empty string if there is none.

getJobState( ) : String

Returns the job state currently set for the job, or an empty string if the job state was never set.

getUserName() : String

Returns the short user name for the job, or an empty string if no user information has been associated with this job.

getUserFullName( ) : String

Returns the full user name for the job, or an empty string if no user information has been associated with this job.

getUserEmail( ) : String

Returns the user e-mail address for the job, or an empty string if no user information has been associated with this job.

getPrivateData (tag : String) : String

Returns the value of the private data with the specified tag, or an empty string if no private data with that tag was set for the job.

getPrivateDataTags( ) : String[ ]

Returns a list of all tags for which non-empty private data was set for the job.

getPriority( ) : Number

Returns the job priority for this job as a signed integer number; see job priorities.

getArrivalStamp( ) : Number

Returns the arrival stamp for this job as a signed integer number; see arrival stamps.

Updating job ticket info

See using hierarchy info, using email info, viewing job state statistics, and configuring users for background information.

The update functions do not affect jobs that have already been sent (by calling one of the Job.sendTo() functions). They do affect any jobs that will be sent after calling the update function.

Note: The update functions *cannot* be used to define metadata in Submit points and Checkpoints. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

setHierarchyPath(segments : String[ ])

Replaces the location path in the hierarchy info associated with the job to the list of segments in the specified Array. The topmost path segment is stored at index 0.

addBottomHierarchySegment(segment : String)

Adds the specified segment to the location path in the hierarchy info associated with the job, at the end of the list (i.e. at the bottom).

addTopHierarchySegment(segment : String)

Adds the specified segment to the location path in the hierarchy info associated with the job, at the beginning of the list (i.e. at the top).

setEmailAddresses(addresses : String[ ])

Replaces the email addresses in the email info associated with the job by the list of email addresses in the specified Array.

addEmailAddress(address : String)

Adds the specified email address to the email info associated with the job.

setEmailBody(body : String)

Replaces the email body text in the email info associated with the job by the specified string.

appendEmailBody(body : String)

Appends the specified string to the email body text in the email info associated with the job, inserting a line break after the existing body if any.

setJobState(state : String)

Sets the job state for the job to the specified string.

setUserName(username: String)

Sets the short user name associated with the job. If the specified string does not match the name of an existing user in the user database a warning is logged (but the set operation does succeed).

setUserFullName( ) : String

Sets the full user name associated with the job.

setUserEmail( ) : String

Sets the user e-mail address associated with the job.

setPrivateData(tag : String, value : String)

Sets the value of the private data with the specified tag to the specified string. This supports light-weight persistent job information.

setPriority(priority : Number)

Sets the job priority for this job to the specified number, rounded to an integer; see job priorities. Generally speaking, jobs with a higher priority are processed before jobs with a lower priority.

refreshArrivalStamp( )

Refreshes the job's arrival stamp so that it seems that the job just arrived in the flow; see arrival stamps.

Switch uses the arrival stamp to determine the processing order for jobs with equal priority (generally speaking, jobs that arrived in the flow first are handled first). Refreshing the arrival stamp can be meaningful when releasing a job that has been held in some location for a long time, to avoid that the job would be unduly processed before other jobs.

Managing external metadata datasets

The functions described here allow associating external metadata with a job's internal job ticket. The Dataset classes in the metadata module implement each of the supported metadata data models: XML data model, JDF data model, XMP data model, and Opaque data model.

The createDataset() and setDataset() functions must not be used after any of the Job.sendTo() or Job.fail()functions was called for a job.

Note: These functions *cannot* be used to define metadata in Submit points and Checkpoints. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

createDataset (model : String, extension: String) : Dataset

Creates and returns a new external metadata dataset object with the specified data model ("XML", "JDF", "XMP" or "Opaque") and with a backing file path and name appropriate for this job, without creating the actual backing file. Returns null if an unknown data model is requested, or if any of the Job.sendTo() or Job.fail() functions was already called for the job.

The optional “extension” argument defines the backing file extension for the "opaque" data model. Other data models do not support custom extensions.

The caller is responsible for creating a backing file that conforms to the specified data model before attempting to read any data from the dataset. The backing file path can be retrieved from the returned Dataset object.

It is not possible to create a writable or an embedded dataset with this function.

setDataset (tag : String, value : Dataset)

Associates a metadata dataset object with the specified tag, for this job. It is allowed to associate a dataset with a job that has been created for another job.

To prevent datasets owned by multiple jobs, if the input dataset was obtained by the getDataset() method, the setDataset() method creates a copy of the dataset backing file. This also happens if setDataset() is invoked for the same job as for which getDataset() was called.

If the dataset was obtained by the createDataset() method, and when it's *the first time* setDataset() is called for this dataset, no copy is created. However, *for each subsequent call* of setDataset() for the same dataset object, a copy of the dataset backing file will be created.

getDataset (tag : String) : Dataset

Returns the metadata dataset object associated with the specified tag for this job, or null if there is none.

getDatasetTags( ) : String[ ]

Returns a list of all tags for which a metadata dataset object is associated with the job.

removeDataset(tag : String)

Removes the dataset for the job with the specified tag.

Managing embedded metadata datasets

getEmbeddedDataset (writable : Boolean) : Dataset

Returns an embedded metadata dataset object with the XMP data model for the metadata embedded in the job. If there is no supported embedded metadata, the function returns a valid but empty dataset.

The backing file path for the dataset may point to the job itself (if it is an individual file) or to one of the files in the job folder (in some cases). Metadata may be embedded in the file as an XMP packet and/or as binary EXIF or IPTC tags. Metadata fields from multiple sources are synchronized into a unified XMP data model. See supported file formats for more information.

When the getEmbeddedDataset() function is invoked on a job for the first time in a certain entry point, it returns:

- A writable dataset if the "writable" argument is set to true and Switch supports updating metadata for the file format of the backing file (see supported file formats).
- A read-only dataset if the "writable" argument is set to false or if Switch does not support updating metadata for the file format of the backing file.

If the function is called again on the same job in the same entry point, it returns the embedded dataset object that was created in the first call, ignoring the "writable" argument in the repeat calls.

Note:

- A writable dataset keeps its backing file (the job!) open for update. Thus it is necessary to invoke the finishWriting() function on the dataset (which closes the backing file) before attempting to move or process the job in any way.
- Extra option for CP2, see *Job class* in Metadata module.
- This function can be used to define metadata in Submit points and Checkpoints, but NOT with writable datasets and/or CP2 datasets. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

Managing job families

The job family of a job is the set of jobs that have been directly or indirectly generated from the same original job. See Job families.

Note: These functions *cannot* be used to define metadata in Submit points and Checkpoints. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

isSameFamily(job : Job) : Boolean

Returns true if the receiving job and the specified job belong to the same job family; otherwise returns false.

startNewFamily( )

Disconnects the receiving job from its current family and makes it start a fresh family. In other words, the job becomes the equivalent of an original job.

joinFamily(job : Job)

Disconnects the receiving job from its current family and makes it join the family of the specified job.

getJobsInFamily(job : Job) : JobList

Returns a list of Job instances representing the jobs in the job family of the specified job.

The returned Job objects are "read-only" and they support only a limited subset of the functions offered by the Job class. Specifically, these objects support the functions described in the sections Getting job file/folder information and Getting job ticket info, plus the functions isSameFamily() and getJobsInFamily(). Invoking any other function on these objects is not allowed and causes an error.

Furthermore, the information returned by the read-only Job objects is cached in memory at the time the objects are created (i.e. before they are returned). There is no guarantee that the information is still valid, since other processes may be concurrently operating on these jobs.

```
function jobArrived( s : Switch, job : Job ) {
    var logPath = job.createPathWithName("log.txt");
    var logf = new File(logPath);
    logf.open(File.WriteOnly);
    job.startNewFamily();
    var njob = s.createNewJob();
    var njob2 = s.createNewJob();
    njob.joinFamily(job);
    logf.writeLine("isSameFam1: " + Boolean(job.isSameFamily(njob)));
    logf.writeLine("isSameFam2: " + Boolean(job.isSameFamily(njob2)));
    var joblist = job.getJobsInFamily(job);
    logf.writeLine("JOBS IN FAMILY");
    for(var i = 0; i < joblist.length; i++){
        logf.write(i + '. ');
        var j = joblist.at(i);
        logf.writeLine(j.getName());
    }
    logf.close();
    job.sendToSingle(logPath);
}
```

results in: (with single outgoing connection)

```
isSameFam1: true
isSameFam2: false
JOBS IN FAMILY
0. <jobname>
1. dummyFileForNewJob.dat
```

Accessing the occurrence trail

Information about things that happen to a job (occurrences) is written to the internal job ticket as a job moves along the flow; see job occurrence trail. The following functions allow retrieving occurrences for each job. The Occurrence class allows retrieving the attributes for each occurrence.

getOccurrences(type : String) : OccurrenceList

Returns a list of all Occurrence instances of the specified type associated with the job, in chronological order (the most recent occurrence is listed last). If the type argument is missing or if it is an empty string, all occurrences associated with the job are returned.

getMostRecentOccurrence(type : String) : Occurrence

Returns the most recent Occurrence instance of the specified type associated with the job, or null if there is no such instance. If the type argument is missing or if it is an empty string, the most recent occurrence is returned regardless of its type.

Note: For an example, see *Occurrence class*.

Evaluating variables

These functions obtain the value of a single variable in the context of the job (and if needed, in the context of a flow element). The variable must be specified with the usual syntax including the square brackets. Thus by definition the first argument must start with a "[" and end with a "]" and include no white space.

The second argument provides the appropriate Switch class instance for variables that need the flow element context. It can be missing or null for variables that don't need such context.

All of these functions return null if the argument has invalid syntax, if the specified variable is unknown, if an unsupported argument is specified, if the variable needs the flow element context and the Switch argument is null or missing, if an indexed variable is specified without its Index argument, if the variable's text value does not conform to the format for the requested data type, or if the text value is empty.

getVariableAsString(variable : String, s : Switch) : String

Returns the text representation of the variable, formatted appropriately for the variable's data type and taking into account any formatting and indexing options.

```
var name = job.getVariableAsString("[Job.Name]", s);
```

getVariableAsNumber(variable : String, s : Switch) : Number

Same as getVariableAsString() but interprets the string as a decimal number (with or without a decimal point) or as a rational number (two decimal integer numbers separated by a forward slash). For example, "1.25" and "5/4" represent the same number. The function converts the strings "-INF", "-Infinity", "INF", and "Infinity" to negative and positive infinity, respectively. Numbers outside the range of the scripting language are clipped to negative of positive infinity.

```
var nr = job.getVariableAsNumber("[Job.FileCount]", s);
```

getVariableAsBoolean(variable : String, s : Switch) : Boolean

Same as getVariableAsString() but interprets the string as a Boolean value. The preferred strings are "True" and "False". If these do not match, a case insensitive comparison is tried, then simply "t" or "f", and finally non-zero and zero integer representations.

```
var b = job.getVariableAsBoolean("[Job.isFolder]", s);
```

getVariableAsDate(variable : String, s : Switch) : Date

Same as getVariableAsString() but interprets the string as a date-time in the ISO 8601 format. The class of the returned Date object is specific to the scripting environment in use.

```
var date = job.getVariableAsDate("[Job.FileCreationDate]", s);
```

getTextForVariable(variable : String, s : Switch) : String

Returns the text representation of the variable, formatted appropriately for the variable's data type and taking into account any formatting and indexing options.

Returns an empty string if the variable's value happens to be unavailable or if the Index argument for an indexed variable is out of range.

Returns null if the argument's syntax is invalid, if the specified variable is unknown, or if an unsupported argument is specified.

getOperatorForVariable(variable : String, s : Switch) : String

Returns the value of the variable for use as an operand in a script expression, i.e. in the appropriate JavaScript data type for the variable's data type, and after the appropriate conversion.

Returns the appropriate conversion of the empty string if the variable's value happens to be unavailable or if the Index argument for an indexed variable is out of range.

Returns null if the argument's syntax is invalid, if the specified variable is unknown, or if an unsupported argument is specified.

Activating fonts

Note: These functions *cannot* be used to define metadata in Submit points and Checkpoints. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

activateFonts(targetPath : String)

Activates the fonts residing in this job folder by copying them to the specified target location. The target path specifies the folder in which the fonts will be copied, for example an application specific font folder. If this argument is missing, or if it is an empty string, the user's default system font location is used instead.

This function does nothing if the receiving job is an individual file or if the job folder doesn't contain any fonts.

deactivateFonts( )

Undoes the effect of any prior invocations of activateFonts() in this entry point. If necessary this function is called automatically when exiting the entry point, however it is good programming style to call it explicitly.

Automatic Job Completion

The following function is provided to allow generating output jobs for the same input job during multiple subsequent entry point invocations. Normally an input job is removed when exiting the first entry point in which an output job is generated.

Note: This function *cannot* be used to define metadata in Submit points and Checkpoints. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

setAutoComplete(auto-complete : Boolean)

Sets the auto-completion flag for the job to the specified value.

If the auto-completion flag has a value of "false" when exiting the entry point, the job will stay in the input folder, even if any number of sendTo() functions have been called for the job during the entry point invocation, and even if the 'leave originals in place' option has been set to no (on the properties pane in the designer app).

If the auto-completion flag has a value of "true" when exiting the entry point, and one or more sendTo() functions have been called for the job during the entry point invocation, the job will be removed from the input folder. This is the default behavior.

At the start of each entry point invocation, the auto-completion flag is initialized to its default value of "true" for all jobs. Consequently the effect of the setAutoComplete() function is restricted to the current entry point invocation. Furthermore, since the auto-completion flag is checked only when exiting the entry point, the order in which the setAutoComplete() and sendTo() functions are called is of no importance.

The auto-completion flag does not affect the behavior of the fail() functions. After calling a fail function for a job, the job will be removed from the input folder regardless of the value of the auto-completion flag.

5.7.7. Occurrence class

An instance of the Occurrence class represents a single item in a job's occurrence trail, and allows retrieving the occurrence's attributes. See job occurrence trail.

Getting occurrence attributes

getTimeStamp( ) : Date

Returns the moment in time when this occurrence happened. The class of the returned Date object is specific to the scripting environment in use.

getType( ) : String

Returns the occurrence type as a string (one of "Detected", "Produced", "Moved", "Duplicated", "Processed" or "Failed").

getModule( ) : String

Returns the module or type of flow element causing this occurrence (same as the module attribute in a log message).

getFlow( ) : String

Returns the name of the flow causing this occurrence.

getElement( ) : String

Returns the name of the flow element (as displayed in the canvas) causing this occurrence.

getOrigin( ) : String

Returns an indication of the origin of the job before it was injected in the flow by this occurrence. For example, a Submit point writes the absolute path of the original job (on the client computer) in this attribute. This attribute is meaningful only for 'Produced' occurrence types. This function returns the empty string for other occurrence types.

getOutFolder( ) : String

Returns the name of the folder (as displayed in the canvas) that received the job as a result of this occurrence. This attribute is meaningful only for 'Produced', 'Moved', and 'Processed' occurrence types. This function returns the empty string for other occurrence types.

Note: This function returns the outelement attribute.

getConnectionType( ) : String

Returns the type of the connection that carried the job as a result of this occurrence, as one of the following strings: "Noop", "Move", "Filter", "Traffic-data", Traffic-log", "Traffic-datawithlog". This attribute is meaningful only for 'Produced', 'Moved', and 'Processed' occurrence types. This function returns the empty string for other occurrence types.

getSuccessLevel( ) : Number

Returns the success level of the job when carried over a traffic-light connection, as an integer (success = 1, warning = 2, error = 3). This attribute is meaningful only if the job was actually carried over a traffic-light connection. If the getConnectionType() does not return a string that starts with "Traffic", getSuccessLevel() returns zero.

getBasicMessage( ) : String

Returns the basic message associated with this occurrence, without argument replacement or localization. The basic message is not intended for human consumption, but can be used in a script to discriminate between various causes of an error, for example (similar to an error code).

This attribute is meaningful only for 'Failed' occurrence types. This function returns the empty string for other occurrence types.

Note: This function returns the operation attribute without further processing.

getLocalizedMessage( ) : String

Returns the message associated with this occurrence after argument replacement and localization. The localized message is intended for human consumption.

This attribute is meaningful only for 'Failed' occurrence types. This function returns the empty string for other occurrence types.

Note: This function returns the localized message derived from the operation attribute and the relevant message arguments using the algorithm used for log messages.

```
/*
 * logging the info of the most recent occurrence
 */

//create logfile
var logPath = job.createPathWithName("log.txt");
```

```
var f = new File(logPath);
f.open(File.WriteOnly);

//get most recent occurence (no argument)
var occ = job.getMostRecentOccurrence();

//write info to file
f.writeLine("Date: " + occ.getTimeStamp());
f.writeLine("Type: " + occ.getType());
f.writeLine("Module: " + occ.getModule());
f.writeLine("Element: " + occ.getElement());
f.writeLine("Flow: " + occ.getFlow());
f.writeLine("Origin: " + occ.getOrigin());
f.writeLine("OutFolder: " + occ.getOutFolder());
f.writeLine("ConnType: " + occ.getConnectionType());
f.writeLine("Successlvl: " + occ.getSuccessLevel());
f.writeLine("BasicMsg: " + occ.getBasicMessage());
f.writeLine("LocalizedMsg: " + occ.getLocalizedMessage());

//save and send to single outgoing connection
f.close();
job.sendToSingle(logPath);
```

5.7.8. WebhookRequest class

Represents the arrived request.

Properties

method

String - The HTTP request method (POST, PUT or DELETE)

protocol

String - The HTTP protocol type (HTTP or HTTPS)

path

String - The request URL path ("/myscript/notify" for "http://enfocus.com/myscript/notify")

query

Object– An object with the properties representing the query string parameters from the request's URL

queryString

String – The original decoded query string from the URL

headers

Object – An object with the properties representing the HTTP request headers (like 'Content-Type')

remoteAddress

String - The request's origin IP address

content

The path to a temporary file on disk containing the body

Note: The file is automatically removed when the webhookTriggered entry point finishes.

Functions

getBody(): ByteArray

Returns the content of the request, after it has been decoded as needed (e.g. decompressed as specified by a Content-Encoding header)

Example

This is and example script that shows how to read a header ('my_header') and a query parameter ('my_parameter') from an incoming POST request sent to the URL 'http://localhost:51080/myscript/notify?my_parameter=test'

```
function webhookTriggered( s : Switch, r : WebhookRequest )
{
  if (r == null)
  {
    try
    {
      s.webhookSubscribe(WebhookRequest.POST, "/myscript/notify");
    }
    catch(e)
    {
      s.log(3, "Error: %1", String( e ));
    }
    return;
  }
  s.log(1,"Received request using method: %1", r.method);
  s.log(1,"Request data: %1", r.getBody().toString());
  s.log(1,"Request headers: %1", JSON.stringify(r.headers));
  s.log(1,"Request query: %1", JSON.stringify(r.query));
  if ('my_header' in r.headers)
  {
    s.log(1,"Value of 'my_header': %1", r.headers.my_header);
  }
  if ('my_parameter' in r.query)
  {
    s.log(1,"Value of 'my_parameter': %1", r.query.my_parameter);
  }
}
```

5.7.9. WebhookResponse class

Represents the response to the webhook request.

Properties

statusCode: Number

Contains the status code of the response.

headers: Object

Contains the headers of the response.

Functions

setHeader(name : String, value : String)

Sets the value of the specified header.

getHeader(name : String) : String

Gets the value of the specified header.

isEmpty()

Checks that the response is empty i.e. *statusCode* == 0, *content* and *headers* are empty.

clear()

Makes the response empty.

write(data : String, codec : String)

Writes the string content to the file fileName if possible (completely replacing the original contents if the file already exists); otherwise throws an exception. If codec is not specified "UTF-8" is used.

writeByteArray(data : ByteArray)

Writes the content to the file fileName if possible (completely replacing the original contents if the file already exists); otherwise throws an exception.

5.7.10. List classes: ConnectionList, JobList

Due to implementation limitations Switch is unable to return an array of Connection, Job or Occurrence instances. Instead, it returns a helper object of the corresponding list class, that is, ConnectionList, JobList or OccurrenceList respectively.

These helper classes offer the following functions to access the items in the list.

getCount( ) : Number

Returns the number of items in the list (may be zero).

getItem(index : Number) : Connection or Job or Occurrence

Returns the item in the list at the specified (zero-based) index. If the index is out of range, the function returns null.

length : Number

This is a read-only property (rather than a function) that contains the value returned by getCount().

at(index : Number) : Connection or Job or Occurrence

Returns the same value as getItem(index).

For an example, see *Connection class*.

Note: Functions belonging to the JobList and ConnectionList class cannot be used to define metadata in Submit points and Checkpoints. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

5.8. Metadata module

5.8.1. Map class

A Map object (that is, an instance of the Map class) holds a set of mappings from an XML namespace prefix to a full namespace URI. These mappings are used for resolving namespace prefixes in XML element and attribute names. See also XPath expressions, XMP location paths and the Node class in the XML module.

Constructing Map objects

There is no direct constructor for Map objects. Instead the *Dataset class* (in the Metadata module) and the *Document class* (in the XML module) offer functions to obtain a new Map object. Each class offers two distinct functions:

- createEmptyMap( ) returns a new empty prefix map object.
- createDefaultMap( ) returns a new prefix map object that already contains all mappings that occur in the XML document associated with the receiving object

In case the script developer can easily determine the set of prefix mappings used by a script, the preferred mechanism is to add that list of prefix mappings to a newly obtained empty prefix map. In reality however this is not always practical. For example, a query expression (with prefixes) might be provided by a user at run-time and the script developer may wish to avoid exposing the complexity involving namespaces to the user.

With the default prefix map the query expression can use any of the prefixes that occur in the backing file. While this is certainly not according to the book, the conventions for the use of prefixes are often stable enough to allow this sort of convenience trick.

Adding/accessing mappings in a Map object

put(prefix: String, namespaceURI: String)

Adds the specified prefix to URI mapping to the map, replacing the previous mapping for this prefix if any.

get(prefix: String) : String

Returns the URI mapped to the specified prefix or null if there is no mapping for the prefix.

contains(prefix: String) : Boolean

Returns true if the map contains the URI mapped to the specified prefix.

5.8.2. List classes

Due to limitations in implementation, the scripting API is unable to return an array of sessions, users, certificates, or AltText objects. Instead, it returns a helper object of the corresponding list class, that is, SessionList, UserList, CertificateList, or AltTextList respectively.

These helper classes offer functions to access the items in the list, identical to the functions offered by the list classes in the flow element module (*JobList* etc).

```
//dataset is a Dataset object with a CP2 model
var sessions = dataset.getAllSessions();
for(var i = 0; i < sessions.length; i++){
  var session = sessions.at(i);
  //..do something with session
}
```

Implementation note on Exceptions

The scripting API does not throw exceptions. All C++ toolkit exceptions must be caught within each scripting API function. In some cases the scripting API function description may describe what to do if an exception is caught (for example, return a null object). In many other cases, the scripting API states that the result is undefined or there simply is no described (or obvious) behavior. The recommended exception handling is as follows:

- Issue an error message to the Switch message log, describing as clearly as possible what the problem is and what caused it.
- Provide stable behavior, even if the requested function cannot be executed. For example, when writing to a read-only object, do nothing at all other than issuing an error message.
- Avoid performing an operation partially; try to execute every operation completely or not at all.

5.8.3. FileStatistics class

The FileStatistics class allows retrieving certain statistics about file contents for a number of supported file formats. The class does not allow modifying file contents.

Each FileStatistics instance references a particular file, which may be any file on the local file system (whether it is a job or not). The FileStatistics class does not support folders.

Constructing

FileStatistics(file-path : String) : FileStatistics

Constructs a FileStatistics instance associated with a file specified through its absolute file path. If the specified path references a folder rather than file, the constructed instance behaves as if the referenced file doesn't exist.

Getting file system attributes

The functions in this section work independently of the file's file format.

getPath( ) : String

Returns the absolute file path.

getName( ) : String

Returns the filename including filename extension if present.

getNameProper( ) : String

Returns the filename excluding filename extension.

getExtension( ) : String

Returns the filename extension after the last '.', or the empty string if there is none.

getMacType( ) : String

Returns the Mac file type code as a 4-character string if available, otherwise the empty string.

getMacCreator( ) : String

Returns the Mac creator code as a 4-character string if available, otherwise the empty string.

isType(ext : String) : Boolean

Returns true if the file matches the specified file type, specified as a filename extension, and false otherwise. A file matches if its filename extension and/or its Mac file type (after conversion) match the specified filename extension.

This function does not examine the file contents.

isFile( ) : Boolean

Returns true if the file exists, false otherwise.

getByteCount( ) : Number

Returns the size in bytes of the file, or zero if the file doesn't exist.

getCreated( ) : Date

Returns the creation time of the file, or null if the file doesn't exist.

getModified( ) : Date

Returns the last modification time of the file, or null if the file doesn't exist.

Accessing embedded metadata

getEmbeddedDataset( ) : Dataset

Returns a read-only embedded metadata dataset object with the XMP data model for the metadata embedded in the file, or null if the file doesn't exist. If the file exists but it has no supported embedded metadata, the function returns a valid but empty dataset.

The backing file path for the dataset points to the file. Metadata may be embedded in the file as an XMP packet and/or as binary EXIF or IPTC tags. Metadata fields from multiple sources are synchronized into a unified XMP data model. See supported file formats for more information.

This function behaves similarly to the Job.getEmbeddedDataset() function; it supports the same file and metadata formats and performs the same synchronizations. The advantage is that it can be used with any file (for example, to iterate over all files inside a job folder). It does not however allow metadata updates (it always returns a read-only dataset) and it does not support folders (i.e. it doesn't look for an appropriate backing file inside a folder).

Recognizing file format

The functions in this section recognize file format by looking at the file contents. The current implementation supports the following formats:

Filename extension	Description
AI	Adobe Illustrator (internal format is PDF)
AVI	Video clip
EPS	Encapsulated PostScript
INDD	Adobe InDesign
JPEG	JPEG image
MOV	QuickTime movie
MP3	MP3 sound
PDF	Adobe PDF (Portable Document Format)
PNG	PNG image
PS	Adobe PostScript
PSD	Adobe Photoshop

Filename extension	Description
TIFF	TIFF image
WAV	Wave sound

getFileFormat( ) : String

Returns the format of the file contents (as one of the strings in the first column of the table above), or the empty string if the format is not recognized or if the file doesn't exist. This function checks for all supported file formats, so it may be a bit slow.

isFileFormat(format : String) : Boolean

Returns true if the file exists and its contents has the specified format (as one of the strings in the first column of the table above); otherwise it returns false. This function is faster than getFileFormat() since it has to check for only a single file format.

Getting contents statistics

The functions in this section interpret the file contents to obtain certain file-format-specific statistics. The requested statistic is specified through its name (as defined for each format in subsequent sections). Each statistic has a well-defined data type (listed with its description). It is recommended to use the "get" function with the appropriate data type, but reasonable conversions between data types are supported.

The functions return the null object (as opposed to an empty string or a zero number) if:

- The file doesn't exist.
- The requested statistic is unknown or unsupported for the file format.
- The requested statistic is not present in the file or is not applicable to the file.
- The result can't be converted to the requested data type.

getString(query : String) : String

Returns the requested statistic as a string.

getStringList(query : String) : String[ ]

Returns the requested statistic as a list of strings.

getNumber(query : String) : Number

Returns the requested statistic as a number.

getBoolean(query : String) : Boolean

Returns the requested statistic as a Boolean.

getDate(query : String) : Date

Returns the requested statistic as a date-time.

Data type conversions

Statistic data type	String	Number	Boolean	Date
String	Straightforward	Decimal number representation	Interpret as in the	Interpret ISO 8601

Statistic data type	String	Number	Boolean	Date
			XMP data model	date-time representation
Integer	Decimal representation of the number	Straightforward	False if zero; true if nonzero	Not supported
Rational	Decimal floating point representation of the number	Straightforward	False if zero; true if nonzero	Not supported
Boolean	"true" or "false"	1 or 0	Straight forward	Not supported
Date	ISO 8601 representation of the date-time	Not supported	Not supported	Straight forward

A string list is converted to another data type by converting the first string in the list (if the list is empty, the conversion is not supported). Any other data type is converted to a string list by converting it to a string and forming a list of one item.

Supported statistics

Statistic name	Data type	Supported for	Description
NumberOfPages	Integer	All formats	The number of pages in the document (or a value of one if the format doesn't support multiple pages)
SamplesPerPixel	Integer	JPEG, TIFF, PNG	Number of components per pixel
PixelXDimension	Integer	JPEG, TIFF, PNG	Valid image width, in pixels
PixelYDimension	Integer	JPEG, TIFF, PNG	Valid image height, in pixels
ColorMode	Integer	JPEG, TIFF, PNG	The color mode used: 0 = Bitmap; 1 = Gray; 2 = Indexed color; 3 = RGB; 4 = CMYK; 7 = Multichannel; 8 = Duotone; 9 = Lab color
ColorSpace	Integer	JPEG, TIFF, PNG	Color space information: 1 = sRGB; 65535 = uncalibrated
ICCProfile	String	JPEG, TIFF, PNG	The name of ICC color profile used, if any

Statistic name	Data type	Supported for	Description
Colorants	String list	TIFF	A list of the names of all colorants for Duotone or Multichannel images; the list is empty for all other color models
CellWidth	Integer	TIFF	The width of the dithering or halftoning matrix used to create a dithered or halftoned bilevel file
CellLength	Integer	TIFF	The length of the dithering or halftoning matrix used to create a dithered or halftoned bilevel file
TileWidth	Integer	TIFF	The width (number of columns) of each tile
TileLength	Integer	TIFF	The length (number of rows) of each tile
ColorIntent	Integer	PNG	The sRGB rendering intent: 0 = Perceptual, 1 = RelativeColorimetric, 2 = Saturation, 3 = AbsoluteColorimetric 4 = invalid value
Version	String	PDF	The version of the file format (for example "1.6")
PageWidth	Rational	PDF	The width of the first page in the document not taking into account page rotation and scaling, in points (derived from the media box)
PageDefinedWidth	Rational	PDF	The width of the first page in the document not taking into account page rotation and scaling, in points (derived from the media box)
PageEffectiveWidth	Rational	PDF	The width of the first page in the document taking into account page rotation and scaling, in points (derived from the media box)
PageHeight	Rational	PDF	The height of the first page in the document not taking into account page rotation and scaling, in points (derived from the media box)

Statistic name	Data type	Supported for	Description
PageDefinedHeight	Rational	PDF	The height of the first page in the document not taking into account page rotation and scaling, in points (derived from the media box)
PageEffectiveHeight	Rational	PDF	The height of the first page in the document taking into account page rotation and scaling, in points (derived from the media box)
PageLabels	String list	PDF	A list of the page labels for all pages in the document, or null if none of the pages has a page label
Colorants	String list	PDF	A list of the names of all colorants as they appear in Separation and DeviceN color spaces and in page separation info dictionaries in the document
Fonts	String list	PDF	A list of the names of all fonts (without subsetting prefix) as they appear in font dictionaries in the document
SecurityMethod	String	PDF	The method used to protect the document; possible values: "None", "Password", "Certificate", "LiveCycle"

PDF media boxes

The FileStatistics class offers the following additional statistics for the PDF file format:

Statistic name	Data type	Description
PageBoxesEqual	Boolean	True if all pages have identical page boxes; false otherwise (verifies media, crop, bleed, trim and art box)
MediaBoxWidth	Rational	The width of the media box for the first page in the document not taking into account page rotation and scaling, in points This is a synonym for the 'PageWidth' statistic
MediaBoxDefinedWidth	Rational	The width of the media box for the first page in the document not taking into account page rotation and scaling, in points This is a synonym for the 'PageWidth' statistic

Statistic name	Data type	Description
MediaBoxEffectiveWidth	Rational	The width of the media box for the first page in the document taking into account page rotation and scaling, in points This is a synonym for the 'PageWidth' statistic
MediaBoxHeight	Rational	The height of the media box for the first page in the document not taking into account page rotation and scaling, in points This is a synonym for the 'Pageheight' statistic
MediaBoxDefinedHeight	Rational	The height of the media box for the first page in the document not taking into account page rotation and scaling, in points This is a synonym for the 'Pageheight' statistic
MediaBoxEffectiveHeight	Rational	The height of the media box for the first page in the document taking into account page rotation and scaling, in points This is a synonym for the 'Pageheight' statistic
CropBoxWidth	Rational	The width of the crop box for the first page in the document not taking into account page rotation and scaling, in points
CropBoxDefinedWidth	Rational	The width of the crop box for the first page in the document not taking into account page rotation and scaling, in points
CropBoxEffectiveWidth	Rational	The width of the crop box for the first page in the document taking into account page rotation and scaling, in points
CropBoxHeight	Rational	The height of the crop box for the first page in the document not taking into account page rotation and scaling, in points
CropBoxDefinedHeight	Rational	The height of the crop box for the first page in the document not taking into account page rotation and scaling, in points
CropBoxEffectiveHeight	Rational	The height of the crop box for the first page in the document taking into account page rotation and scaling, in points

Statistic name	Data type	Description
BleedBoxWidth	Rational	The width of the bleed box for the first page in the document not taking into account page rotation and scaling, in points
BleedBoxDefinedWidth	Rational	The width of the bleed box for the first page in the document not taking into account page rotation and scaling, in points
BleedBoxEffectiveWidth	Rational	The width of the bleed box for the first page in the document taking into account page rotation and scaling, in points
BleedBoxHeight	Rational	The height of the bleed box for the first page in the document not taking into account page rotation and scaling, in points
BleedBoxDefinedHeight	Rational	The height of the bleed box for the first page in the document not taking into account page rotation and scaling, in points
BleedBoxEffectiveHeight	Rational	The height of the bleed box for the first page in the document taking into account page rotation and scaling, in points
TrimBoxWidth	Rational	The width of the trim box for the first page in the document not taking into account page rotation and scaling, in points
TrimBoxDefinedWidth	Rational	The width of the trim box for the first page in the document not taking into account page rotation and scaling, in points
TrimBoxEffectiveWidth	Rational	The width of the trim box for the first page in the document taking into account page rotation and scaling, in points
TrimBoxHeight	Rational	The height of the trim box for the first page in the document not taking into account page rotation and scaling, in points
TrimBoxDefinedHeight	Rational	The height of the trim box for the first page in the document not taking into account page rotation and scaling, in points
TrimBoxEffectiveHeight	Rational	The height of the trim box for the first page in the document taking into account page rotation and scaling, in points
ArtBoxWidth	Rational	The width of the art box for the first page in the document not taking into account page rotation and scaling, in points

Statistic name	Data type	Description
ArtBoxDefinedWidth	Rational	The width of the art box for the first page in the document not taking into account page rotation and scaling, in points
ArtBoxEffectiveWidth	Rational	The width of the art box for the first page in the document taking into account page rotation and scaling, in points
ArtBoxHeight	Rational	The height of the art box for the first page in the document not taking into account page rotation and scaling, in points
ArtBoxDefinedHeight	Rational	The height of the art box for the first page in the document not taking into account page rotation and scaling, in points
ArtBoxEffectiveHeight	Rational	The height of the art box for the first page in the document taking into account page rotation and scaling, in points

PDF/X version key

The FileStatistics class offers the following additional statistics for the PDF file format:

Statistic name	Data type	Description
PDFXVersionKey	String	The contents of the PDF/X version key in the document, or the empty string if there is no such key; one of the following values may be returned: • PDF/X-1a:2001 • PDF/X-3:2002 • PDF/X-1a:2003 • PDF/X-3:2003 • PDF/X-4 • PDF/X-4p The PDF/X version key indicates a claim that the document conforms to the PDF/X specification, but it does not offer any guarantees

PDF page content

The FileStatistics class offers the following additional statistics for the PDF file format.

Obtainging the information for these statistics requires parsing the complete page contents, so it might be time consuming.

Statistic name	Data type	Description
ColorSpaceFamilies	String list	A list of the names of all color space families actually used in the document's page contents; the following names may be returned: DeviceGray, DeviceRGB, DeviceCMYK, CalGray, CalRGB, Lab, ICCBasedGray, ICCBasedRGB, ICCBasedCMYK, Indexed, Pattern, Separation, DeviceN. For Indexed or Pattern, the underlying color space families are also listed
TransparentColor	Boolean	True if the document's page contents uses transparency for defining colors; false otherwise
FontTypes	String list	A list of the names of all font types actually used in the document's page contents; the following names may be returned: • TrueType • Type1 • Type3 • MultipleMaster • Composite

Example

```
var fileStat = new FileStatistics(job.getPath());

var created = fileStat.getCreated();
s.log(1, "Creation date: " + created);

var numberOfPages = fileStat.getNumber("NumberOfPages");
s.log(1, "Number of pages: " + numberOfPages);
```

5.8.4. ClientConnection class

The ClientConnection object provides information about the current client connection that triggered evaluation of the script. It means that only scripts defined in Checkpoint and Submit point metadata can use the ClientConnection object. See also the getClientConnection() method of *Environment class* on page 175.

Following methods are supported:

QString getEmail() const;

Returns the *email* of the user that established the connection.

QString getUserName() const;

Returns the *name of the user* that established the connection.

QString getFullUserName() const;

Returns the *full name of the user* that established the connection.

QString getIP() const;

Returns the *IP address* from which the connection is established.

5.8.5. Job class

The getEmbeddedDataset() function offers an additional optional argument and additional semantics related to Certified PDF 2.

getEmbeddedDataset(writable: Boolean, cp2 : Boolean) : Dataset

If the cp2 argument is false or missing, the function behaves as before. If the cp2 argument is true, the function returns a CP2 dataset if the backing file is a PDF file, otherwise it returns an XMP dataset. The behavior is summarized in the table below. A returned CP2 dataset adheres to all semantics of an XMP dataset. For example metadata fields from multiple sources are synchronized into a unified XMP data model.

When the getEmbeddedDataset() function is invoked on a job for the first time in a certain entry point, it returns a dataset of the type listed in the table below. If the function is called again on the same job in the same entry point, it returns the embedded dataset object that was created in the first call, ignoring the value of the "writable" and "cp2" arguments in the repeat calls.

CP2	Writable	File format	Type of dataset returned
False	False	Any	Read-only XMP
False	True	Supported for update Not supported for update	Writable XMP Read-only XMP
True	False	PDF Other than PDF	Read-only CP2 Read-only XMP
True	True	PDF Other supported for update Not supported for update	Writable CP2 Writable XMP Read-only XMP

5.8.6. Dataset class

Since external datasets can never have the "CP2" data model, functions applying solely to external datasets are not affected by this specification. Specifically, the model arguments in the Job.createDataset() and Job.sendToLog() functions do not support "CP2".

The Dataset class is a base class that offers a number of functions common to all types of metadata datasets. There is a separate inheriting class for each metadata data model supported by the scripting API: XML data model, JDF data model, XMP data model, and Opaque data model.

Instances of the Dataset class (or rather of its inheriting classes) can be obtained with the Job.createDataset(), Job.getDataset() and Job.getEmbeddedDataset() functions in the flow element module.

See also external metadata and embedded metadata.

Getting dataset attributes

getPath( ) : String

For an external dataset, returns the absolute file path (including filename and extension) for the backing file for this dataset. This is used predominantly for placing the backing file in the correct location while creating a new dataset. However it might conceivably be used to bypass the data model access mechanisms and provide direct access to the backing file.

For an embedded dataset, returns the absolute path to the file that embeds the metadata (rather than the metadata packet as a separate entity).

hasValidData( ) : Boolean

Returns true if the backing file exists, can be read, and conforms to the syntax and semantics of the dataset's underlying data model. Otherwise returns false.

getModel( ) : String

Returns the name of the data model for the dataset ("XML", "JDF", "XMP" or "Opaque"). In effect this determines the actual class of this instance.

Returns "CP2" for a CP2 dataset; otherwise behaves as before.

isWritable( ) : Boolean

Returns true if this data set supports updating its corresponding embedded metadata using the dataset classes (only embedded metadata can be updated). Otherwise returns false.

This function cannot be used to define metadata in Submit points and Checkpoints. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

Note: XML datasets can't be updated by the dataset class. To update them it is necessary to get the path to the backing file via the job and write the changes directly to the backing file.

isEmbedded( ) : Boolean

Returns true if this data set corresponds to embedded metadata, i.e. if it has been returned by the Job.getEmbeddedDataset() function. Otherwise returns false.

Creating prefix maps

createEmptyMap( ) : Map

Returns a new empty prefix map object.

createDefaultMap( ) : Map

Returns a new prefix map object that already contains all mappings that occur in the backing file. For an Opaque dataset this function returns an empty prefix map object.

For XML and JDF datasets, the default namespace (if any) is included in the default map with one or more special prefixes. This allows XPath queries to refer to the default namespace using these special prefixes. (XPath does not support the default namespace concept, so you always have to explicitly provide a namespace prefix).

The following table summarizes the contents of the default map for each data model.

Data model	Default namespace defined in backing file	Namespace prefixes defined in the backing file
XML	Included in default map with special prefixes "default_switch_ns" and "dn" unless these prefixes are otherwise defined in the file	Included in default map using the prefix specified in xmlns:<prefix> attribute
JDF	Included in default map with special prefixes "jdf", "default_switch_ns" and "dn" unless these prefixes are otherwise defined in the file	
XMP	Not applicable	
Opaque	Not applicable	Not applicable

For a CP2 dataset, the "cp2" prefix (with corresponding namespace) is always included in the map, even if the underlying XMP does not define it (because if does not contain Certified PDF 2 metadata or because it uses another prefix). Otherwise the function behaves as before.

5.8.7. XML data model

The XML data model describes a class that inherits all functions described in the Dataset class.

Query language

The XML data model expects a well-formed XML backing file (refer to the XML 1.0 specification), which is parsed with support for namespaces into an XML document object model (DOM). The object model is then queried with an XPath 1.0 expression using the root node as the context node.

Namespaces

Namespace prefixes in the query expression are resolved into namespace URIs using the prefix map provided to the query functions as a second argument. If the map argument is omitted or null, the default prefix map is used for resolving prefixes.

Result data types

The value of an XPath expression can have several data types (a node set with several elements, the text value of an attribute, a number returned by the count() function, the Boolean result of a comparison). The value is converted to one of the data types string, number, or Boolean using XPath 1.0 semantics – which may very well differ from the conversion semantics in the scripting language. For example the XPath 1.0 conversion from string to Boolean returns true for all non-empty strings (including the string "false").

Query functions

evalToString(xpath : String, prefix-map : Map) : String

Evaluates the XPath 1.0 expression against the backing file and returns the result after converting it to a string value with the XPath 1.0 string() function.

evalToNumber(xpath : String, prefix-map: Map) : Number

Evaluates the XPath 1.0 expression against the backing file and returns the result after converting it to a numeric value with the XPath 1.0 number() function.

evalToBool(xpath : String, prefix-map: Map) : Boolean

Evaluates the XPath 1.0 expression against the backing file and returns the result after converting it to a Boolean value with the XPath 1.0 boolean() function.

Note: For more information and examples, see *XML module*.

5.8.8. JDF data model

The JDF data model describes a class that inherits all functions described in the Dataset class.

Query language

The JDF data model expects an XML backing file that conforms to the JDF 1.3 specification. A JDF instance describes a nested structure of JDF nodes and their resources. The result is a nicely structured XML file which can be validated by an XML schema. The JDF specification uses XPath 1.0 notation to refer to items in a JDF instance. Thus it is appropriate to use this notation to locate items in a query.

The current implementation of the JDF data model in fact considers the underlying XML file with some limited JDF-specific functionality layered on top.

Namespaces

Namespace prefixes in the query expression are resolved into namespace URIs using the prefix map provided to the query functions as a second argument. If the map argument is omitted or null, the default prefix map is used for resolving prefixes.

A JDF instance is required to set its default namespace to the JDF namespace, and all XML elements and attributes described in the JDF specification reside in this namespace. XPath 1.0 however does not have the concept of a default namespace. Therefore the default prefix map for the JDF data model includes a definition of the "jdf" prefix (in addition to any other prefixes defined in the backing file), and JDF query expressions should explicitly use this prefix.

A JDF instance may contain elements or attributes in other namespaces, allowing third-party extensions to the specification. The query functions allow providing a prefix map to enable accessing these extensions.

Generic XPath querying

The JDF data model implementation offers two types of query mechanisms. The first mechanism allows evaluating an arbitrary XPath 1.0 expression with the same semantics as those in the XML data model, i.e. the expression's value is converted to one of the data types string, number or Boolean using XPath 1.0 semantics.

This generic mechanism supports complex queries that may involve multiple elements or attributes (for example, counting the number of elements in a particular node set).

evalToString(xpath : String, prefix-map : Map) : String

Evaluates the XPath 1.0 expression against the backing file, and returns the result after converting it to a string value with the XPath 1.0 string() function.

evalToNumber(xpath : String, prefix-map: Map) : Number

Evaluates the XPath 1.0 expression against the backing file, and returns the result after converting it to a numeric value with the XPath 1.0 number() function.

evalToBool(xpath : String, prefix-map: Map) : Boolean

Evaluates the XPath 1.0 expression against the backing file, and returns the result after converting it to a Boolean value with the XPath 1.0 boolean() function.

JDF data type querying

The second query mechanism uses an XPath 1.0 location path (not an arbitrary expression) to indicate a single XML attribute or a single XML element, and thus to indicate a single text value. This text value is then interpreted and converted to a script object according to the JDF data type semantics described in the JDF specification.

The current implementation supports a limited subset of the JDF data types, as described in the following table.

Query function	Supported JDF data types
getString()	string, NMTOKEN, telem, text any other data type (since everything is stored as text)
getNumber()	double, integer, LongInteger
getBoolean()	boolean
getDate()	date, dateTime, gYearMonth

The query functions described below return the null object (as opposed to an empty string or a zero number) if:

- The location path does not evaluate to a single attribute or to a single element; for example it evaluates to an empty node set, to a node set with two elements, to a comments node or to a number.
- The text value of the indicated attribute or element does not conform to the format for the requested data type.

getString(xpath : String, prefix-map : Map) : String

Returns the text value of the item at the specified XPath 1.0 location path. The text content of an element is concatenated into a single string. Leading and trailing whitespace is removed. Whitespace including linebreaks may occur in the body of the string.

getNumber(xpath : String, prefix-map: Map) : Number

Same as getString but interprets the string as a decimal number with support for negative and positive infinity (-INF and INF) as per the JDF specification. Numbers outside the range of the scripting language are clipped to negative of positive infinity.

getBoolean(xpath : String, prefix-map: Map) : Boolean

Same as getString but interprets the string as a boolean ("true" or "false") as per the JDF specification.

getDate(xpath : String, prefix-map : Map) : Date

Same as getString but interprets the string as a date and/or a time according to the JDF specification (based on the ISO 8601 format). The class of the returned Date object is specific to the scripting environment in use.

5.8.9. XMP data model

The XMP data model describes a class that inherits all functions described in the Dataset class.

5.8.9.1. Query language

The XMP data model expects a backing file that conforms to the Adobe XMP specification dated June 2005. The XMP backing file is parsed into an XMP-specific document object model. This XMP object model is then queried with an XMP location path (which is a small subset of XPath 1.0). The standard built-in aliases for XMP property names are automatically resolved.

Note: An XMP packet or file is a subset of RDF serialized to XML. Because a particular RDF construct (and thus an XMP property) can be serialized to XML in various ways, it is virtually impossible to correctly query an XMP file using generic XPath 1.0 expressions.

Namespaces

Namespace prefixes in the XMP location path are resolved into namespace URIs using the prefix map provided to the query functions as a second argument. If the map argument is omitted or null, the default prefix map is used for resolving prefixes.

The query functions do not provide a separate argument to specify the top-level namespace (also called the "schema namespace" in XMP). This means that all property names in the XMP location path (including the top-level property name) must have a namespace prefix.

The default prefix map for a dataset with this model includes all mappings for the standard XMP namespaces, augmented with any extra mappings that occur in the backing file.

5.8.9.2. Querying existence

hasLeaf(xmp-path : String, prefix-map : Map) : Boolean

Returns true if there is a leaf property at the XMP location path.

hasStruct(xmp-path : String, prefix-map : Map) : Boolean

Returns true if there is a structure property at the XMP location path.

hasArray(xmp-path : String, prefix-map : Map) : Boolean

Returns true if there is an array property at the XMP location path.

hasAltTextArray(xmp-path : String, prefix-map : Map) : Boolean

Returns true if there is an Alt-Text array property at the XMP location path (that is, an array property of type "Alt" with leaf properties as children).

getArrayItemCount(xmp-path : String, prefix-map : Map) : Number

Returns the number of items in the array at the XMP location path or 0 if there is no such property or if it is not an array.

5.8.9.3. Querying values

The query functions described below return the null object (as opposed to an empty string or a zero number) if:

- The location path does not point to a leaf property (i.e. the property is not present or it is present but it is an array or a struct rather than a leaf).
- The string value does not conform to the format for the requested data type.

getString(xmp-path : String, prefix-map : Map) : String

Returns the value of the leaf property at the specified XMP location path.

getNumber(xmp-path : String, prefix-map: Map) : Number

Same as getString but interprets the string as a decimal number (with or without a decimal point) or as a rational number (two decimal integer numbers separated by a forward slash). For example, "1.25" and "5/4" represent the same number.

getBoolean(xmp-path : String, prefix-map: Map) : Boolean

Same as getString but interprets the string as a Boolean value. The preferred strings are "True" and "False". If these do not match, a case insensitive comparison is tried, then simply "t" or "f", and finally non-zero and zero integer representations.

getDate(xmp-path : String, prefix-map : Map) : Date

Same as getString but interprets the string as a date-time in the ISO 8601 format. The class of the returned Date object is specific to the scripting environment in use.

Querying localized text

Localized text properties are stored in alt-text arrays. They allow multiple concurrent localizations of a property value, for example a document title or copyright in several languages.

The most important aspect of these functions is that they select an appropriate array item based on one or two RFC 3066 language tags. One of these languages, the "specific" language, is preferred and selected if there is an exact match. For many languages it is also possible to define a "generic" language that may be used if there is no specific language match. The generic language must be a valid RFC 3066 primary subtag or the empty string.

For example, a specific language of "en-US" should be used in the US, and a specific language of "en-UK" should be used in England. It is also appropriate to use "en" as the generic language in each case. If a US document goes to England, the "en-US" title is selected by using the "en" generic language and the "en-UK" specific language.

It is considered poor practice, but allowed, to pass a specific language that is just an RFC 3066 primary tag. For example "en" is not a good specific language, it should only be used as a generic language. Passing "i" or "x" as the generic language is also considered poor practice but allowed.

RFC 3066 language tags must be treated in a case insensitive manner.

The XMP specification defines an artificial language, "x-default", that is used to explicitly denote a default item in an alt-text array. The localized text functions have several special features related to the x-default item.

The selection of the array item is performed as follows:

- Look for an exact match with the specific language.
- If a generic language is given, look for a partial match.
- Look for an x-default item.
- Choose the first item.

A partial match with the generic language is where the start of the item's language matches the generic string and the next character is '-'. An exact match is also recognized as a degenerate case.

It is fine to pass x-default as the specific language. In this case, selection of an x-default item is an exact match by the first rule, not a selection by the 3rd rule. The last 2 rules are fallbacks used when the specific and generic languages fail to produce a match.

getLocalizedText(xmp-path : String, prefix-map : Map, genericLang : String, specificLang: String) : String

Returns the localized string in the specified alt-text array selected according to the rules described above or null if there is no such string (even no default).

When querying a file with following code segment:

```
<mns:subject>
    <rdf:Alt>
        <rdf:li xml:lang="nl-BE">Voorbeeld</rdf:li>
        <rdf:li xml:lang="x-default">Sample</rdf:li>
        <rdf:li xml:lang="en-US">Sample</rdf:li>
        <rdf:li xml:lang="fr-FR">Exemple</rdf:li>
        <rdf:li xml:lang="de-DE">Beispiel</rdf:li>
    </rdf:Alt>
</mns:subject>
```

using following functions (with map already initialized, containing prefix 'mns'):

```
//exact match
job.log(1,  ds.getLocalizedText('mns:subject', map, 'fr', 'fr-FR' ));
//partial match
job.log(1,  ds.getLocalizedText('mns:subject', map, 'en', 'en-UK' ));
//x-default (or first line if no x-default)
job.log(1,  ds.getLocalizedText('mns:subject', map, 'es', 'es-ES' ));
```

the result will be:

```
Exemple
Sample
Sample (or Voorbeeld if x-default left out)
```

5.8.9.4. Updating properties

The functions described in this section are available only for writable datasets, that is, datasets for which the isWritable() function returns true. Invoking any of these functions on a non-writable dataset is a programming error and has unpredictable results.

Finishing

finishWriting( )

Flushes any unsaved metadata changes to the backing file, closes the backing file, and turns the dataset into a read-only dataset. From now on, the isWritable() function returns false and further updates are impossible.

For a CP2 dataset, an appropriate Certified PDF 2 signature is written to the backing file, in addition to updating the XMP and CP2 data structures. These changes are affected through incremental save. Otherwise, the function behaves as before.

A writable dataset keeps its backing file open for updating after the dataset is created. Since the backing file is the current job (or a file in the job folder), it is necessary to close it before attempting to move or process job in any way. In practice it is seldom necessary to explicitly call the finishWriting() function because Switch automatically invokes it on the embedded dataset for a job at the following times:

- When one of the Job.sendTo() or Job.fail() functions is invoked on the job.
- When the entry point in which the dataset was created returns.

Setting leaf values

The set functions described below set the value of a leaf property. They succeed if:

- The specified property exists and it is a leaf property: the property value is updated.
- The specified property does not exist, but its immediate parent does exist and it is a struct: the property is created with the specified name and its value is set.
- The specified property does not exist, but its immediate parent does exist and it is an array: the property is created with the specified index and its value is set; any missing array items (that is, siblings of the property with a lower index) are created as well and set to an empty string value.

setString(xmp-path : String, prefix-map : Map, value : String) : Boolean

Sets the value of the leaf property at the specified XMP location path to the specified string. Returns true if successful, false otherwise.

setNumber(xmp-path : String, prefix-map : Map, value : Number, precision : Number) : Boolean

Same as setString but produces a decimal representation of the number; "precision" indicates the number of digits after the decimal point; if precision is zero the representation has no decimal point (that is, this is suitable for an integer number).

setBoolean(xmp-path : String, prefix-map : Map, value : Boolean) : Boolean

Same as setString but produces the string "True" or "False" depending the Boolean value.

setDate(xmp-path : String, prefix-map : Map, value : Date) : Boolean

Same as setString but produces a date-time representation in the ISO 8601 format. The class of the Date object is specific to the scripting environment in use.

Note:

- To set the value of a child property, first verify that its parent exists, and if not, create the parent with the set functions described in the following section. It may be necessary to apply this process recursively to create the parent's parent.
- To set the value of an item in an alt-text array, use the setLocalizedText() function described later. The above set functions do no allow a language selector in the specified XMP location path.

Setting localized text

setLocalizedText(xmp-path : String, prefix-map : Map, value : String, genericLang : String, specificLang: String) : Boolean

Modifies the value of a selected item in an alt-text array. Creates an appropriate array item if necessary, and handles special cases for the x-default item.

If the selected item is from a match with the specific language, the value of that item is modified. If the existing value of that item matches the existing value of the x-default item, the x-default item is also modified. If the array only has 1 existing item (which is not x-default), an x-default item is added with the given value.

If the selected item is from a match with the generic language and there are no other generic matches, the value of that item is modified. If the existing value of that item matches the existing value of the x-default item, the x-default item is also modified. If the array only has 1 existing item (which is not x-default), an x-default item is added with the given value.

If the selected item is from a partial match with the generic language and there are other partial matches, a new item is created for the specific language. The x-default item is not modified.

If the selected item is from the last 2 rules then a new item is created for the specific language. If the array only had an x-default item, the x-default item is also modified. If the array was empty, items are created for the specific language and x-default.

Returns true if successful, false if the specified property does not exist or is not an Alt-Text array.

Adding structs and arrays

The set functions described below create struct or array properties. They succeed if:

- The specified property exists and it is has the appropriate type: nothing happens.
- The specified property does not exist, but its immediate parent does exist and it is a struct: the property is created with the appropriate type.
- The specified property does not exist, but its immediate parent does exist and it is an array: the property is created with the specified index and the appropriate type; any missing array items (that is, siblings of the property with a lower index) are created as well with the same type.

setStruct(xmp-path : String, prefix-map : Map) : Boolean

Creates a struct property at the specified XMP location path. Returns true if successful, false otherwise.

setUnorderedArray(xmp-path : String, prefix-map : Map) : Boolean

Creates an unordered array property ("Bag") at the specified XMP location path. Returns true if successful, false otherwise.

setOrderedArray(xmp-path : String, prefix-map : Map) : Boolean

Creates an ordered array property ("Seq") at the specified XMP location path. Returns true if successful, false otherwise.

setAlternateArray(xmp-path : String, prefix-map : Map) : Boolean

Creates an alternate array property ("Alt") at the specified XMP location path. Returns true if successful, false otherwise.

Note: To change the type of a property between leaf, array and struct, first remove the property and then create it again with the appropriate set function. If the other property is not removed first, an error can be produced.

At some (already existing) location paths, it's impossible to randomly use setters, some paths are fully specified by *the latest XMP spec*.

Removing properties

removeProperty(xmp-path : String, prefix-map : Map)

Removes the property at the specified XMP location path and the complete sub-tree rooted at the property. If the property does not exist the function does nothing.

Use case

The following example makes use of the functions described in the whole XMP chapter.

```
//get a writable XMP Dataset
var ds = job.getEmbeddedDataset(true);
/*
 * add some XMP metadata to the file
 */

//get the default prefixmap
var map = ds.createDefaultMap();

//add own namespaces
map.put('mns', 'MyNamespace');
map.put('mns2', 'MyNamespace2');

//own struct
ds.setStruct('mns:general', map);

//new author
ds.setString('mns:general/mns:author', map, 'John Doe');

//new keywords
ds.setUnorderedArray('mns:general/mns:keywords', map);
ds.setString('mns:general/mns:keywords/*[1]', map, 'sample');
ds.setString('mns:general/mns:keywords/*[2]', map, 'xmp');
ds.setString('mns:general/mns:keywords/*[3]', map, 'metadata');

//alttextarray  ds.setAlternateArray('mns:general/mns:subject', map);
ds.setLocalizedText('mns:general/mns:subject', map, 'Sample', 'en', 'en-US');
ds.setLocalizedText('mns:general/mns:subject', map, 'Voorbeeld', 'nl', 'nl-BE');
ds.setLocalizedText('mns:general/mns:subject', map, 'Exemple', 'fr', 'fr-FR');
ds.setLocalizedText('mns:general/mns:subject', map, 'Beispiel', 'de', 'de-DE');

//ordered array
ds.setOrderedArray('mns2:ordernrs', map);
ds.setNumber('mns2:ordernrs/*[1]', map, 7456);
ds.setNumber('mns2:ordernrs/*[2]', map, 3236);
ds.setNumber('mns2:ordernrs/*[3]', map, 9946);
ds.setNumber('mns2:ordernrs/*[4]', map, 1147);

//remove a property
ds.removeProperty('mns2:ordernrs/*[2]');

//datelist
ds.setOrderedArray('mns2:changedates');
ds.setDate('mns2:changedates/*[1]', map, new Date(Date.parse("2012-02-23T12:30:09")));
ds.setDate('mns2:changedates/*[2]', map, new Date(Date.parse("2012-02-24T14:33:22")));
ds.setDate('mns2:changedates/*[3]', map, new Date(Date.parse("2012-02-27T13:45:33")));
ds.setDate('mns2:changedates/*[4]', map, new Date(Date.parse("2012-02-29T11:24:19")));

//boolean
ds.setBoolean('mns2:readOnly', map, true);

//finished with writing
ds.finishWriting();

//save file
job.sendToSingle(job.getPath());
```

Results in the following code segment:

```
<rdf:Description rdf:about=""
            xmlns:mns="MyNamespace/">
         <mns:general rdf:parseType="Resource">
            <mns:author>John Doe</mns:author>
            <mns:keywords>
               <rdf:Bag>
                  <rdf:li>sample</rdf:li>
                  <rdf:li>xmp</rdf:li>
                  <rdf:li>metadata</rdf:li>
               </rdf:Bag>
            </mns:keywords>
            <mns:subject>
               <rdf:Alt>
                  <rdf:li xml:lang="x-default">Sample</rdf:li>
                  <rdf:li xml:lang="en-US">Sample</rdf:li>
                  <rdf:li xml:lang="nl-BE">Voorbeeld</rdf:li>
                  <rdf:li xml:lang="fr-FR">Exemple</rdf:li>
                  <rdf:li xml:lang="de-DE">Beispiel</rdf:li>
               </rdf:Alt>
            </mns:subject>
         </mns:general>
      </rdf:Description>
      <rdf:Description rdf:about=""
            xmlns:mns2="MyNamespace2/">
         <mns2:ordernrs>
            <rdf:Seq>
               <rdf:li>7456</rdf:li>
               <rdf:li>9946</rdf:li>
               <rdf:li>1147</rdf:li>
            </rdf:Seq>
         </mns2:ordernrs>
         <mns2:changedates>
            <rdf:Seq>
               <rdf:li>2012-02-23T11:30:09Z</rdf:li>
               <rdf:li>2012-02-24T13:33:22Z</rdf:li>
               <rdf:li>2012-02-27T12:45:33Z</rdf:li>
               <rdf:li>2012-02-29T10:24:19Z</rdf:li>
            </rdf:Seq>
         </mns2:changedates>
         <mns2:readOnly>True</mns2:readOnly>
      </rdf:Description>
```

5.8.10. Opaque data model

The Opaque data model describes a class that inherits all functions described in the Dataset class; it does not add any functions to those of the Dataset class.

The Opaque data model does not provide access to its contents other than the backing file as a whole. This data model can be used to associate a chunk of arbitrary, non-interpreted data with a job.

5.8.11. CP2 data model

The CP2 data model (part of the metadata module) describes a class that inherits all functions of the XMP data model and all functions of the Dataset class. The following subsections describe the functions specific to the CP2 data model.

Note: The functions in this class cannot be used to define metadata fields in Checkpoints and Submit points. For more information, refer to 'Defining metadata fields' in the Switch Reference Guide.

Overview

Since most Certified PDF 2 information is stored as XMP, the "CP2" data model inherits from the "XMP" data model. Thus any XMP data model function can be invoked on a CP2 dataset as well.

A CP2 dataset is always embedded. It is not possible to create an external dataset with this data model.

A CP2 dataset can either be read-only or writable. When a writable CP2 dataset is finalized, an appropriate Certified PDF 2 signature is written to the backing file - in addition to updating the XMP and CP2 data structures.

A CP2 dataset can be obtained for any PDF file, whether it is a valid Certified PDF 2 file or not. This allows converting a regular PDF file into a Certified PDF 2 file.

A writable CP2 dataset can be "cleared", removing all Certified PDF 2 related information. This allows converting a Certified PDF 2 file into a regular PDF file.

File formats other than PDF are not supported, since Certified PDF 2 is tied to PDF.

Effects on the scripting API

Supporting the new CP2 data model requires:

- Adding semantics to a limited number of existing functions.
- Offering a range of new classes (and corresponding functions).

Compatibility

GWG Proof of Preflight

Since a PDF file with a valid GWG Proof of Preflight ticket is – by definition – also a valid Certified PDF 2 file, the scripting API presented in this document fully supports the GWG Proof of Preflight specification.

Certified PDF 1

The scripting API does not support Certified PDF 1 files. In other words, Certified PDF 1 files are treated as regular PDF files, and any Certified PDF 1 data structures are ignored.

Accessing XMP metadata

Certified PDF 2 metadata

After obtaining a CP2 dataset, the Certified PDF 2 metadata stored in the XMP packet must be accessed exclusively through the functions specific for the CP2 data model. Accessing any XMP metadata fields in the CP2 namespace using the regular XMP access functions has undefined results.

Other XMP Metadata

It is perfectly valid however to access any other XMP metadata (i.e. unrelated to Certified PDF 2) using the regular XMP access functions on the CP2 dataset, even intermixed with the use of functions specific for the CP2 data model.

File level operations

Validity

The CP2 data model offers functions to determine whether the underlying file has a valid Certified PDF 2 signature, and whether it contains a valid Certified PDF 2 data structure even if the signature is invalid.

Creating a Certified PDF 2 file

Converting a regular PDF file to a Certified PDF 2 file requires no extra functions; it can be accomplished as follows:

- Obtain a writable CP2 dataset for the PDF file.
- Add any relevant Certified PDF 2 metadata information to the dataset using the CP2 data model functions described later.
- Explicitly finish writing on the dataset or exit the entry point.

Removing Certified PDF 2 information

Converting a Certified PDF 2 file to a regular PDF file can be accomplished as follows:

- Obtain a writable CP2 dataset for the Certified PDF 2 file.
- Call the removeCertifiedPDF() function on the CP2 dataset.
- Explicitly finish writing on the dataset or exit the entry point.

Changing the PDF file

The scripting API does not offer any functions to update portions of a PDF file other than the changes directly related to the Certified PDF 2 data structures (including the XMP packet and the Certified PDF 2 signature).

Also, in an entry point that obtains a writable CP2 dataset on the incoming PDF file, it is not allowed to invoke an external application that changes that PDF file (or keeps it open for update without actually modifying it). This is because:

- A writable CP2 dataset keeps the underlying PDF file open for update.
- The process of writing the changes back to a temporary copy of the file is not under the script programmer's control. Thus cannot be coordinated with the external application.

File level validity

hasValidSignature( ) : Boolean [R]

Returns true if the backing file for which this dataset was created has a valid Certified PDF 2 signature; returns false if the file has no conforming signature or if the signature is no longer valid because the file was modified.

hasPreviousSessions( ) : Boolean [R]

Returns true if the backing file for which this dataset was created contains a valid Certified PDF 2 data structure with at least one session (not counting any sessions automatically added while obtaining the dataset); returns false otherwise.

If hasValidSignature() returns true, hasPreviousSessions() should return true as well (unless some application has created a corrupt Certified PDF 2 file). However if hasValidSignature() returns false, hasPreviousSessions() may still return true.

Removing Certified PDF 2

removeCertifiedPDF2(fullSave : Boolean) [w]

Removes all metadata related to Certified PDF 2 from the dataset and causes the dataset to subsequently behave as a regular XMP dataset. When finishWriting() is called (or the entry point exits), any data structures related to Certified PDF 2 are removed from the underlying file.

If fullSave is false or missing, these changes are affected through incremental save, which means the file will contain the original information and overhead. If fullSave is true, a full save is performed, eliminating any information that is no longer referenced and thus reducing overhead.

After this function was invoked on a dataset, invoking any of the CP2-specific functions on the dataset is a programming error and has unpredictable results.

Sessions

A session represents the work done to a Certified PDF 2 file between saves.

Active session

When obtaining a CP2 dataset, a new session object is automatically created and stored in the dataset. This session is called the "Active" session and it contains any changes made to the CP2 dataset during this entry point. The session is automatically completed when the dataset is finished for writing (explicitly or by exiting the entry point).

Delaying with uncertified files

If a previously valid Certified PDF 2 file has been updated by a nonconforming application, its XMP metadata do not contain session objects for each performed save. When obtaining a writable CP2 dataset for such a file, session objects are automatically created for each uncertified save, before creating the active session.

Updating sessions

Once a session has been finished it can no longer be modified. The active session is the only editable session object. Any session can be removed from the dataset. However when a session is removed, all previous sessions are automatically removed as well. In addition any certificates added during the removed sessions are automatically removed as well. As an alternative to removing a session completely (and as an exception to the rule of not modifying previous sessions), it is possible to strip a session. Stripping a session removes all optional session properties while leaving any certificates intact.

getAllSessions( ) : SessionList [R]

Returns a list of all sessions in the dataset, including the active session, in order of occurrence (that is, the active session is the last session in the list).

getPreviousSessions( ) : SessionList [R]

Returns a list of the sessions that were already in the backing file before the dataset was created, that is, excluding the active session and any other automatically added sessions. The list is in order of occurrence (that is, the most recent session is listed last). The list may be empty.

getActiveSession( ) : Session [R]

Returns the active session; this is the only editable session.

touchZones(zones : String | String[]) : Boolean [W]

Notifies the dataset that the specified editing zone(s) have been touched in the PDF file during the active session. This function performs three actions:

- Add the specified zone(s) to the editing zones of the active session.
- Set the state of any certificate with overlapping zones and residing in the active session to "Unknown".
- Cause any certificate with overlapping zones and residing in previous sessions to become dirty (there is no actual change in the data structure, just in the internal caches for the dataset).

The function returns true if any certificate was affected, false otherwise.

removeSessionsIncluding(index : Number) [W]

Removes the session at the specified zero-based index (in the list returned by getAllSessions) and all previous sessions. All certificates associated with these sessions and any orphaned users are removed as well.

stripSessionAt(index : Number) [W]

Removes all optional information from the session at the specified zero-based index (in the list returned by getAllSessions). If the associated user becomes orphaned, it is removed as well.

Certificates

A CP2 dataset can contain zero or more certificates. Each certificate references the session during which it was created.

Adding certificates

New certificates can be added to a writable dataset. A new certificate is automatically associated with the active session.

Certificate class

The class ID of a new certificate must be specified when the certificate object is constructed which cannot be changed any time later. The scripting API does not support access to class properties in certificates.

Updating certificates

Certificates other than those associated with the active session cannot be updated. It is possible however to completely remove any certificate.

Proof of Preflight certificate

When obtaining a CP2 dataset for a PDF file that contains a GWG Proof of Preflight ticket without a corresponding preflight certificate, a new preflight certificate is automatically created and stored in the dataset. Depending on implementation issues, such auto-generated preflight certificate may be associated with the active session or with a previous session.

getAllCertificates( ) : CertificateList [R]

Returns a list of all certificates in the dataset, including any certificates in the active session, in order of appearance in the dataset. The list may be empty.

addNewCertificate(classID : String) : Certificate [W]

Creates a new certificate of the specified vendor-neutral class, associates it with the current session, adds it to the dataset and returns a reference to the new certificate.

A certificate class associates additional semantics with a particular type of certificate. Even if the scripting API does not support access to the class properties that may be defined for certain certificate classes, it is important to specify the appropriate class. For example, a preflight certificate may cause a GWG Proof of Preflight ticket to be written in the PDF file, while other certificates do not show this correspondence.

The Certified PDF 2 specification describes two "built-in" classes, as described in the following table:

classID (CP2:class_id)	Description
"" (the empty string)	A generic certificate, without additional semantics
"Preflight"	A preflight certificate, which may correspond to a GWG Proof of Preflight ticket in the PDF file

In many cases these built-in classes provide sufficient functionality. Third-party vendors may agree on other values for classID; it is recommended to register such other values with Enfocus to avoid name collisions.

After adding a new certificate with this function, and before finishing the dataset (or exiting the entry point), the script must set the certificate's required properties (see "Required and Automatic Properties" section below) to a non-empty value. Alternatively the script may remove the certificate. Failing to do either of these is a programming error and has unpredictable results.

removeCertificateAt(index : Number) [W]

Removes the certificate at the specified zero-based index (in the list returned by getAllCertificates).

After completion of this function, the indexes for the remaining certificates may have shifted and references to the removed certificate may have become invalid. Using such invalid references is a programming error and has unpredictable results.

Users

A CP2 dataset can contain zero or more user objects. Each session can reference a user; multiple sessions can share the same user.

Active user

There is a function to create a user object for the active session. If the specified user turns out to have the same properties as one of the users already present in the CP2 dataset, the active session is updated to reference that existing user. Otherwise a new user object is automatically added to the dataset.

Updating users

Only active users can be updated. When a user is no longer referenced by any session (because the referencing sessions have been stripped or removed), the "orphaned" user object is automatically removed from the dataset.

getAllUsers( ) : UserList [R]

Returns a list of all users in the dataset, including the active user if any, in arbitrary order. The list may be empty.

addActiveUser( ) : User [W]

If the active session has an associated user, this function returns it. If the active session has no associated user, this function creates a new user object, associates it with the active session and returns a reference to it.

removeActiveUser( ) : User [W]

Removes the active user from the dataset and dissociates the active session from any user.

Required and automatic properties

Session, user and certificate objects have optional and required properties, as stated in the Certified PDF 2 file format specification.

Automatic properties

The scripting API automatically generates appropriate values for most (but not all) required properties and for some optional properties. Some of these automatic properties are not even exposed to the script programmer. Others are available for reading but can't be changed under the script programmer's control. The following table lists the automatic properties and corresponding details.

Object	Property	Exposure
Session	CP2:session_id CP2:start_byte	Not exposed

Object	Property	Exposure
	CP2:user_id_ref	
Session	CP2:start_time CP2:end_time	Read-only
Session	CP2:tool_id CP2:tool_version CP2:tool_desc CP2:appl_id CP2:appl_version CP2:appl_desc	Read-only
User	CP2:user_id	Not exposed
Certificate	CP2:session_id_ref	Not exposed
Certificate	CP2:time	Read-only

Other required properties

For sessions and users all required properties are automatic. For certificates however the required properties listed in the following table must be supplied by the script programmer; most even don't have a default value.

Object	Property	Default value
Certificate	CP2:type_id CP2:type_version CP2:type_desc CP2:impl_id CP2:impl_version CP2:statement_desc	None Value to be supplied by script
Certificate	CP2:state	"Unknown" Value can be overridden by script
Certificate	CP2:zones	"All" Value can be overridden by script

Editing zones

Zones for which a certificate is sensitive

A new certificate is automatically sensitive to all editing zones. If this is not changed, any zone touched in any following session makes the certificate dirty. It is advisable to narrow the certificate's editing zones down to avoid it becoming dirty unnecessarily.

Zones touched during a session

The active session starts without any editing zones set. For each modification applied to the PDF file during that session, appropriate editing zones must be added. Adding an editing zone to a session has the following effect on any certificates that are sensitive to that zone:

- If the certificate resides in a previous session it becomes dirty. This is a consequence of the definitions in the Certified PDF 2 file format specification; there is no need to update the certificate.
- If the certificate resides in the active session, its state is automatically reset to "Unknown". According to the Certified PDF 2 file format specification a certificate can never become dirty due to an editing zone listed for the session during which the certificate was added. Resetting the certificate state is the only way to indicate that the certificate may no longer represent the state of the file.

External datasets

Since external datasets can never have the "CP2" data model, functions applying solely to external datasets are not affected by this specification. Specifically, the model arguments in the Job.createDataset() and Job.sendToLog() functions do not support "CP2".

5.8.11.1. Session class

A Session object represents a session stored in a CP2 dataset.

Getters [R]

getStartTime( ) : Date

Returns the time when this session was started (CP2:start_time), or null if the property is absent.

getEndTime( ) : Date

Returns the time when this session was ended (CP2:end_time), or null if the property is absent.

getZones( ) : String[]

Returns an unordered list of the editing zone strings specifying the areas of the PDF file that were changed during this session (CP2:zones), or an empty list if the property is absent.

getComment( ) : String

Returns the user-supplied comment that describes the changes to the PDF file during this session (CP2:comment). This function returns the English language alternate, or the empty string if the property is absent.

getCommentLocalized( ) : AltText

Returns the user-supplied comment that describes the changes to the PDF file during this session (CP2:comment). This function returns the complete set of language alternates, which is empty if the property is absent. For the active session, the returned AltText object is writable.

getChangeDescriptions( ) : String[]

Returns an ordered list of descriptions of the changes to the PDF file during this session (CP2:change_desc), or an empty list if the property is absent. This function returns the English language alternate for each item.

getChangeDescriptionsLocalized( ) : AltTextList

Returns an ordered list of descriptions of the changes to the PDF file during this session (CP2:change_desc), or an empty list if the property is absent. This function returns the complete set of language alternates for each item. The returned AltText objects are read-only.

getUser( ) : User

Returns the user object referenced by this session (through CP2:user_id_ref), or null if the session doesn't reference a user.

getDataMap( ) : DataMap

Returns a data map referencing the collection of private data for this session (CP2:data), which is empty if the property is absent. For the active session, the returned DataMap object is writable.

getToolID( ) : String

Returns the unique identifier for the toolkit or library that created this session (CP2:tool_id), or the empty string if the property is absent.

getToolVersion( ) : String

Returns the version string for the toolkit or library that created this session (CP2:tool_version), or the empty string if the property is absent.

getToolDescription( ) : String

Returns the human-readable description of the toolkit or library that created this session (CP2:tool_desc). This function returns the English language alternate, or the empty string if the property is absent.

getToolDescriptionLocalized( ) : AltText

Returns the human-readable description of the toolkit or library that created this session (CP2:tool_desc). This function returns the complete set of language alternates, which is empty if the property is absent. The returned AltText object is read-only.

getApplicationID( ) : String

Returns the unique identifier for the application that created this session (CP2:appl_id), or the empty string if the property is absent.

getApplicationVersion( ) : String

Returns the version string for the application that created this session (CP2:appl_version), or the empty string if the property is absent.

getApplicationDescription( ) : String

Returns the human-readable description of the application that created this session (CP2:appl_desc). This function returns the English language alternate, or the empty string if the property is absent.

getApplicationDescriptionLocalized( ) : AltText

Returns the human-readable description of the application that created this session (CP2:appl_desc). This function returns the complete set of language alternates, which is empty if the property is absent. The returned AltText object is read-only.

getCertificates( ) : CertificateList

Returns a list of all certificates associated with this session, in order of appearance in the dataset. The list may be empty.

Setters [W]

These functions may be invoked only on the active session. Invoking them on any other session is a programming error and has unpredictable results.

setComment(value : String)

Sets the user-supplied comment that describes the changes to the PDF file during this session (CP2:comment). This function sets the English language alternate and erases all other alternates.

To set other language alternates, use appropriate setters on an AltText object obtained with getCommentLocalized().

addChangeDescription(value : String)

Appends the specified string to the list of descriptions of the changes to the PDF file during this session (CP2:change_desc). This function sets the English language alternate for the new item, omitting any other alternatives.

addNewChangeDescriptionLocalized( ) : AltText

Appends a newly created item to the list of descriptions of the changes to the PDF file during this session (CP2:change_desc), and returns a reference to the new item. The returned AltText object is empty, and it is writable. Use appropriate setters on the returned AltText object to set any language alternates.

5.8.11.2. Certificate class

A Certificate object represents a certificate stored in a CP2 dataset.

Getters [R]

getClassID( ) : String

Returns the vendor-neutral class identifier of this certificate (CP2:class_id). This may be one of the "built-in" identifiers described in the Certified PDF 2 specification (the empty string for generic certificates, or "Preflight" for preflight certificates) or another value agreed between third-party vendors. It is recommended to register such other values with Enfocus.

getClassVersion( ) : String

Returns the version string for the data type of this certificate's class properties (CP2:class_version), or the empty string if the property is absent. The scripting API doesn't support access to class properties.

getTypeID( ) : String

Returns the unique identifier for the type of this certificate (CP2:type_id), or the empty string if the property is absent. This identifier indirectly specifies the data type of the certificate's private properties, and is used by a conforming application to select a certificate handler implementation that can interpret the contents of these properties.

getTypeVersion( ) : String

Returns the version string for the type of this certificate (CP2:type_version), or the empty string if the property is absent. This version string indirectly specifies the version of the certificate's private properties, and can be used by a certificate handler implementation while interpreting these private properties.

getTypeDescription( ) : String

Returns the human-readable description of this certificate's type and version (CP2:type_desc). This function returns the English language alternate, or the empty string if the property is absent.

getTypeDescriptionLocalized( ) : AltText

Returns the human-readable description of this certificate's type and version (CP2:type_desc). This function returns the complete set of language alternates, which is empty if the property is absent. For a certificate associated with the active session, the returned AltText object is writable.

getImplementationID( ) : String

Returns the unique identifier for the certificate handler implementation that created this certificate (CP2:impl_id), or the empty string if the property is absent.

getImplementationVersion( ) : String

Returns the version string for the certificate handler implementation that created this certificate (CP2:impl_version), or the empty string if the property is absent.

getImplementationDescription( ) : String

Returns the human-readable description of the certificate handler implementation that created this certificate (CP2:impl_desc). This function returns the English language alternate, or the empty string if the property is absent.

getImplementationDescriptionLocalized( ) : AltText

Returns the human-readable description of the certificate handler implementation that created this certificate (CP2:impl_desc). This function returns the complete set of language alternates, which is empty if the property is absent. For a certificate associated with the active session, the returned AltText object is writable.

getSession( ) : Session

Returns the session during which this certificate was created (referenced through CP2:session_id_ref).

getStatementDescription( ) : String

Returns the human-readable description of the statement made by this certificate (CP2:statement_desc). This function returns the English language alternate, or the empty string if the property is absent.

getStatementDescriptionLocalized( ) : AltText

Returns the human-readable description of the statement made by this certificate (CP2:statement_desc). This function returns the complete set of language alternates, which is empty if the property is absent. For a certificate associated with the active session, the returned AltText object is writable.

getState( ) : String

Returns the state of this certificate (CP2:state) as one of the following string values: "Errors", "Warnings", "Info", "Success", or "Unknown". For a new certificate the state is initialized to "Unknown". In most cases this should be replaced by an appropriate value.

getStateDescription( ) : String

Returns the human-readable description of the state of this certificate (CP2:state_desc). This function returns the English language alternate, or the empty string if the property is absent.

getStateDescriptionLocalized( ) : AltText

Returns the human-readable description of the state of this certificate (CP2:state_desc). This function returns the complete set of language alternates, which is empty if the property is absent. For a certificate associated with the active session, the returned AltText object is writable.

getZones( ) : String[]

Returns an unordered list of the editing zone strings specifying the type of changes to which this certificate is sensitive (CP2:zones), or an empty list if the property is absent (which means that the certificate is not sensitive to any changes in the PDF file). For a new certificate this property is initialized to the single editing zone "All". It is advisable to narrow the certificate's editing zones down to avoid it becoming dirty unnecessarily.

getDataMap( ) : DataMap

Returns a data map referencing the collection of private data for this certificate (CP2:data), which is empty if the property is absent. For a certificate associated with the active session, the returned DataMap object is writable.

getDataDescription( ) : String

Returns the human-readable description of this certificate's private properties taken as a whole (CP2:data_desc). This function returns the English language alternate, or the empty string if the property is absent.

getDataDescriptionLocalized( ) : AltText

Returns the human-readable description of this certificate's private properties taken as a whole (CP2:data_desc). This function returns the complete set of language alternates, which is empty if the property is absent. For a certificate associated with the active session, the returned AltText object is writable.

getFingerPrint( ) : String

Returns the fingerprint representing a unique identifier for this certificate's configuration (CP2:fingerprint), or the empty string if the property is absent. The fingerprint allows determining whether or not two configurations are equivalent without accessing the contents of class or private properties.

getTime( ) : Date

Returns the time when this session was created (CP2:time), or null if the property is absent. For a new certificate this property is initialized to the current system time; it can't be updated.

isDirty( ) : Boolean

Returns true if this certificate certificate's editing zones overlap with any of the editing zones touched in a later session (excluding the certificate's own session), false otherwise.

isValid( ) : Boolean

Returns true if this certificate is valid, false otherwise. A certificate in any session other than the active session is valid if and only if all of the following conditions apply:

- The certificate resides in a Certified PDF file with a valid signature.
- The certificate's state is not "Unknown".
- The certificate is not dirty, that is, its editing zones do not overlap with any of the editing zones touched in a later session (excluding the certificate's own session).

A certificate in the active session is valid if and only if its state is not "Unknown". This is because it is assumed that the certificate will be saved in a file with a valid signature and because the editing zones in the active session never make the certificate dirty.

Setters [W]

These functions may be invoked only on certificates residing in the active session. Invoking them on any other certificate is a programming error and has unpredictable results.

setClassVersion(value : String)

Sets the version string for the data type of this certificate's class properties (CP2:class_version). The scripting API does not support access to class properties.

setTypeID(value : String)

Sets the unique identifier for the type of this certificate (CP2:type_id). This identifier indirectly specifies the data type of the certificate's private properties, and is used by a conforming application to select a certificate handler implementation that can interpret the contents of these properties.

setTypeVersion(value : String)

Sets the version string for the type of this certificate (CP2:type_version). This version string indirectly specifies the version of the certificate's private properties, and can be used by a certificate handler implementation while interpreting these private properties.

setTypeDescription(value : String)

Sets the human-readable description of this certificate's type and version (CP2:type_desc). This function sets the English language alternate and erases all other alternates. To set other language alternates, use appropriate setters on an AltText object obtained with getTypeDescriptionLocalized().

setImplementationID(value : String)

Sets the unique identifier for the certificate handler implementation that created this certificate (CP2:impl_id).

setImplementationVersion(value : String)

Sets the version string for the certificate handler implementation that created this certificate (CP2:impl_version).

setImplementationDescription(value : String)

Sets the human-readable description of the certificate handler implementation that created this certificate (CP2:impl_desc). This function sets the English language alternate and erases all other alternates. To set other language alternates, use appropriate setters on an AltText object obtained with getImplementationDescription().

setStatementDescription(value : String)

Sets the human-readable description of the statement made by this certificate (CP2:statement_desc). This function sets the English language alternate and erases all other alternates. To set other language alternates, use appropriate setters on an AltText object obtained with getStatementDescriptionLocalized().

setState(value : String)

Sets the state of this certificate (CP2:state). The state must be specified as one of the following values (or a case-insensitive variation thereof): "Errors", "Warnings", "Info", "Success", or "Unknown". Any other value is interpreted as "Unknown". For a new certificate the state is initialized to "Unknown". In most cases this should be replaced by an appropriate value.

setStateDescription(value : String)

Sets the human-readable description of the state of this certificate (CP2:state_desc). This function sets the English language alternate and erases all other alternates. To set other language alternates, use appropriate setters on an AltText object obtained with getStateDescriptionLocalized().

setZones(value : String[])

Sets an unordered list of editing zone strings specifying the type of changes to which this certificate is sensitive (CP2:zones). An empty list indicates that the certificate is not sensitive to any changes in the PDF file. For a new certificate this property is initialized to the single editing zone "All". It is advisable to narrow the certificate's editing zones down to avoid it becoming dirty unnecessarily.

setDataDescription(value : String)

Sets the human-readable description of this certificate's private properties taken as a whole (CP2:data_desc). This function sets the English language alternate and erases all other alternates. To set other language alternates, use appropriate setters on an AltText object obtained with getDataDescriptionLocalized().

setFingerPrint(value : String)

Sets the fingerprint representing a unique identifier for this certificate's configuration (CP2:fingerprint). The fingerprint allows determining whether or not two configurations are equivalent without accessing the contents of class or private properties.

5.8.11.3. User class

A User object represents a user stored in a CP2 dataset.

Getters [R]

getName( ) : String
getCompany( ) : String
getStreet( ) : String
getPostalCode( ) : String
getCity( ) : String
getState( ) : String
getCountry( ) : String
getEMail( ) : String
getPhone( ) : String
getFax( ) : String

Returns the corresponding property of this user's contact information, or the empty string if the property is absent.

getUserMessage( ) : String

Returns the user-supplied message associated with this user (CP2:message). This function returns the English language alternate, or the empty string if the property is absent.

getUserMessageLocalized( ) : AltText

Returns the user-supplied message associated with this user (CP2:message). This function returns the complete set of language alternates, which is empty if the property is absent. For the active user, the returned AltText object is writable.

Setters [W]

These functions may be invoked only on the active user. Invoking them on any other user is a programming error and has unpredictable results.

setName(value : String)
setCompany(value : String)
setStreet(value : String)
setPostalCode(value : String)
setCity(value : String)
setState(value : String)
setCountry(value : String)
setEMail(value : String)
setPhone(value : String)
setFax(value : String)

Sets the corresponding property of this user's contact information.

setUserMessage(value : String)

Sets the user-supplied message associated with this user (CP2:message). This function sets the English language alternate and erases all other alternates. To set other language alternates, use appropriate setters on an AltText object obtained with getUserMessageLocalized().

5.8.11.4. DataMap class

A DataMap object represents a collection of private data associated with a session or certificate stored in a CP2 dataset. Each data map retains a reference to its underlying data structure, so it can be used for both reading and writing. Private data items can be stored in three different ways. Each storage type has associated semantics as described in the following table.

Storage type	Storage form	Encoding	Recommended for
test	Plain text string	Unicode	brief segments of human-readable text
base64	Base64 encoded stream	Binary	short data streams
ref	PDF object data stream (i.e. outside of the XMP packet)	Binary	longer data streams

Note: The scripting API fully supports the "ref" storage type; it implements the appropriate methods to read and write data from and to PDF objects and registers these methods with the C++ toolkit.

Getters [R]

isEmpty( ) : Boolean

Returns true if the collection is empty, false otherwise. The collection is empty if the underlying property is absent or if no private data items are listed for the property.

getTags( ) : String[]

Returns an unordered list of tags used to identify private data items in this collection. The list is empty if the collection is empty.

hasTag(tag : String) : Boolean

Returns true if the collection contains a private data item with the specified tag, false otherwise.

isString(tag : String) : Boolean

Returns true if the collection contains a private data item with the specified tag, and that item has storage type "text". Otherwise the function returns false.

isBinary(tag : String) : Boolean

Returns true if the collection contains a private data item with the specified tag, and that item has storage type "base64" or "ref". Otherwise the function returns false.

isPDFObject(tag : String) : Boolean

Returns true if the collection contains a private data item with the specified tag, and that item has storage type "ref". Otherwise the function returns false.

getString(tag : String) : String

Returns the contents of the private data item with the specified tag as a string, or an empty string if there is no such item or if the item does not have storage type "text".

getBinary(tag : String) : ByteArray

Returns the contents of the private data item with the specified tag as a byte array, or null if there is no such item or if the item does not have storage type "base64" or "ref". If the private data has storage type "base64" it is decoded to binary form before returning.

Setters [W]

These functions may be invoked only on data maps associated with the active session or with a certificate in the active session. Invoking them on any other data map is a programming error and has unpredictable results.

put(tag : String, contents : String)

Places a private data item of storage type "text" in the collection with the specified tag and contents. If an item with the same tag already existed, it is replaced by this new item.

put(tag : String, contents : ByteArray, asPDFObject : Boolean)

Places a private data item of storage type "base64" or "ref" in the collection with the specified tag and contents. If an item with the same tag already existed, it is replaced by this new item. If asPDFObject is false, the item is of storage type "base64" and the binary data is encoded as Base64 before being stored in the XMP packet. If asPDFObject is true, the item is of storage type "ref" and the binary data is stored as a PDF stream object, using one or more lossless encodings as determined by the implementation.

Note: The intention is to avoid introducing new types of stream encodings to the PDF file which could trigger preflight errors; refer to the implementation of Certified PDF 1 in this respect.

remove(tag : String)

Removes the private data item with the specified tag from the collection. If there is no such item, this function does nothing.

5.8.11.5. AltText class

An AltText object represents a collection of language alternates for a localized property of a session, certificate or user stored in a CP2 dataset. Each AltText object retains a reference to its underlying data structure, so it can be used for both reading and writing.

Getters [R]

isEmpty( ) : Boolean

Returns true if the collection of alternates is empty, false otherwise. The collection is empty if the underlying property is absent or if no alternates are listed for the property.

getLocalizedText(genericLang : String, specificLang: String) : String

Returns one of the collection's alternates according to the rules described for the getLocalizedText() function in the XMP data model.

getText( ) : String

Returns one of the collection's alternates, selected by preference in the following order: the English language string, the default string, or any other language string. The getter returns a non-empty string as long as there is at least one alternate string.

Setters [W]

These functions may be invoked only on AltText objects associated with the active session or with a certificate in the active session. Invoking them on any other AltText object is a programming error and has unpredictable results.

setLocalizedText(value : String, genericLang : String, specificLang: String)

Adds or replaces one of the collection's alternates according to the rules described for the setLocalizedText() function in the XMP data model.

setText(value : String)

Sets the specified string as the English language string and erases all other alternates from the collection (to avoid discrepancies between the meaning of existing alternates and the new string).