



XHQ

Reference Guide

About This Guide

Expressions

1

Operators

2

Constants

3

Methods

4

Functions

5

Overflow Underflow and Type Conversion Policies

A

XHQ Data Mapping

B




Units of Measure Conversion

C

Legal information

Warning notice system

This manual contains notices you have to observe in order to ensure your personal safety, as well as to prevent damage to property. The notices referring to your personal safety are highlighted in the manual by a safety alert symbol, notices referring only to property damage have no safety alert symbol. These notices shown below are graded according to the degree of danger.

 DANGER
Indicates that death or severe personal injury will result if proper precautions are not taken.
 WARNING
Indicates that death or severe personal injury may result if proper precautions are not taken.
 CAUTION
Indicates that minor personal injury can result if proper precautions are not taken.
NOTICE
Indicates that property damage can result if proper precautions are not taken.


If more than one degree of danger is present, the warning notice representing the highest degree of danger will be used. A notice warning of injury to persons with a safety alert symbol may also include a warning relating to property damage. See the topic, [Visual Cues for Online Viewing](#), for additional XHQ-specific notices.

Qualified Personnel

The product/system described in this documentation may be operated only by personnel qualified for the specific task in accordance with the relevant documentation, in particular its warning notices and safety instructions. Qualified personnel are those who, based on their training and experience, are capable of identifying risks and avoiding potential hazards when working with these products/systems.

Proper use of Siemens products

Note the following:

 WARNING
Siemens products may only be used for the applications described in the catalog and in the relevant technical documentation. If products and components from other manufacturers are used, these must be recommended or approved by Siemens. Proper transport, storage, installation, assembly, commissioning, operation and maintenance are required to ensure that the products operate safely and without any problems. The permissible ambient conditions must be complied with. The information in the relevant documentation must be observed.

Trademarks

All names identified by ® are registered trademarks of Siemens AG. The remaining trademarks in this publication may be trademarks whose use by third parties for their own purposes could violate the rights of the owner. For a complete list, see the [Copyright](#) topic.

Disclaimer of Liability

We have reviewed the contents of this publication to ensure consistency with the hardware and software described. Since variance cannot be precluded entirely, we cannot guarantee full consistency. However, the information in this publication is reviewed regularly and any necessary corrections are included in subsequent editions.

Copyright © 1998-2019 Siemens AG. All rights reserved. Protected by U.S. Patents Nos. 6,700,590, 7,069,514, 7,478,128, 7,689,579, 7,698,292, 7,814,123, 7,840,607, 8,001,332, 8,078,598, 8,260,783, 8,442,938, 8,566,781, 8,700,671 and 8,700,559; Patents Pending.

Siemens Product Lifecycle Management Software, Inc.
6 Journey, Suite 200
Aliso Viejo, CA 92656-5318, USA
siemens.com/xhq

XHQ® is a registered trademark of Siemens AG in the United States. This License does not grant LICENSEE any rights to trademarks or service marks of Siemens AG.

All other company, product and service names and logos may be trademarks or service marks of their respective companies. Any rights not expressly granted herein are reserved. LICENSEE may not remove or alter any trademark, logo, copyright or other proprietary notices, legends, symbols or labels from the Licensed Software or the Documentation.

This software is proprietary and confidential. Siemens AG or its suppliers own the title, copyright, and other intellectual property rights in the Software. The Software is licensed, not sold.

Adobe, the Adobe logo, Acrobat, the Adobe PDF logo, PostScript, and the PostScript logo, Distiller, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Microsoft, Active Directory, ActiveX, Authenticode, Developer Studio, DirectX, Microsoft, MS-DOS, Outlook, Excel, PowerPoint, Visual Basic, Visual C++, Visual C#, Visual J#, Visual SourceSafe, Visual Studio, Win32, Windows, Windows Server, WinFX, Windows 7, Windows 10, Windows Server 2008, Windows Server 2012, Windows Server 2016, and the Windows logo are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries, or both.

HTML, XML, XHTML and W3C are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Massachusetts Institute of Technology.

IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Oracle, Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates. Oracle, or its licensor, shall at all times retain all rights, title, interest, including intellectual property rights, in Oracle Programs and media.

SAP, SAP R/3, R/3, R/3 software, mySAP, mySAP.com, xApps, xApp, ABAP, BAPI, and SAP NetWeaver are trademarks or registered trademarks of SAP AG in Germany and in several other countries.

Documentum, OpenText Documentum, OpenText and the Corporate Logo are trademarks or registered trademarks of OpenText in the United States and throughout the world.

IBM, the IBM logo, DB2, and ibm.com are trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

InstallShield® is a registered trademark and service mark of Macrovision Corporation and/or Macrovision Europe Ltd. in the United States and/or other countries. DemoShield, InstallFromTheWeb and PackageForTheWeb are service marks and registered trademarks of Macrovision Corporation and/or Macrovision Europe Ltd. in the United States and/or other countries. InstallShield Express, InstallShield for Windows Installer, InstallShield for Windows CE, Express Wizard, InstallShield Objects, WebUpdate, FastReg and NetInstall are trademarks and/or service marks of Macrovision Corporation and/or Macrovision Europe Ltd. InstallShield Software Corporation. InstallShield is a member of Macrovision Corporation.

Siemens provides products and solutions with industrial security functions that support the secure operation of plants, systems, machines and networks.

In order to protect plants, systems, machines and networks against cyber threats, it is necessary to implement – and continuously maintain – a holistic, state-of-the-art industrial security concept. Siemens' products and solutions only form one element of such a concept.

Customer is responsible to prevent unauthorized access to its plants, systems, machines and networks. Systems, machines and components should only be connected to the enterprise network or the internet if and to the extent necessary and with appropriate security measures (e.g. use of firewalls and network segmentation) in place.

Additionally, Siemens' guidance on appropriate security measures should be taken into account. For more information about industrial security, please visit <https://www.siemens.com/industrialsecurity>.

Siemens' products and solutions undergo continuous development to make them more secure. Siemens strongly recommends to apply product updates as soon as available and to always use the latest product versions. Use of product versions that are no longer supported, and failure to apply latest updates may increase customer's exposure to cyber threats.

For the Siemens Security Advisory, visit <https://www.siemens.com/industrialsecurity>.

To stay informed about product updates, subscribe to the Siemens Industrial Security RSS Feed under <https://www.siemens.com/industrialsecurity>.

While every effort is made to ensure the accuracy of content, the XHQ product documentation set (which includes online help) could contain inaccuracies or out-dated material (which includes product screenshots and images) due to the large number of product enhancements being added. As such, the documentation set is subject to change at any time without notice. Refer to the README for documentation corrections and addendum. Please note, updates to the documentation set are reflected in the next general availability major release of XHQ.

Table of Contents

Table of Contents	5
About This Guide	12
Conventions Used in This Guide	12
Visual Cues for Online Viewing	13
Related XHQ Product Documentation	14
Contacting Customer Support	16
General Feedback and Comments	17
1 Expressions	18
Server-side versus Client-side	19
Self-referencing Server-side Expressions	19
Client-side Expression Functions	19
Concepts	20
Naming Rules and Identifier Types	20
Identifiers for Server-side Expressions	20
Identifiers for Client-side Expressions	21
Qualified Values	22
Data Types	22
About Constant Postfix	23
Type Conversions	23
Numeric Constants	24
Time Types	24
Expressions and Collection Members	25
Late-binding Expressions	26
About the Evaluation Schedule	27
Using Formatting Functions in Expressions	28
Using Expressions in the XHQ Interactive Trender	29
Expressions and Aliases	29
Expressions Containing More than One Reference	29
2 Operators	30
Arithmetic Operators	30
Relational Operators	31
Bitwise Operators	32
Logical Operators	33

Comma Operator	34
Operator Precedence	35
3 Constants	36
4 Methods	38
Collection Aggregate Methods	38
Collection Methods	38
Collection Member Methods	39
Historical Access and Aggregation Methods	43
About Historical Access Methods	43
List of Historical Access Methods	43
.at(date)	43
.at(interval_string, intervalnumber)	44
.actual(date)	44
.actual(interval_string, intervalnumber)	44
About Client-side Historical Access Methods	45
About Server-side Historical Access Methods	45
Historical Aggregate Methods	45
About the .avg and .integral methods	46
Historical Aggregate Calculations	47
About Client-side Historical Aggregate Methods	48
About Server-side Historical Aggregate Methods	49
About Connection Support	50
About Data Types Support	51
5 Functions	52
Math Functions	53
abs(a)	55
acos(a)	56
asin(a)	56
atan(a)	56
atan2(a, b)	57
ceil(a)	57
cos(a)	57
exp(a)	57
floor(a)	58
log(a)	58
log10(a)	58
max(a, b)	59

min(a, b)	59
pow(a, b)	60
random()	60
random_Double()	60
random_Real()	60
round(a)	61
sin(a)	61
sqrt(a)	61
tan(a)	61
toDegrees(angrad)	62
toRadians(angdeg)	62
trunc(a)	62
Trinary Operator Function	63
iff	63
Quality and iff Expressions	63
String Functions	64
Interval Formatting	66
compareTo	66
compareToIgnoreCase	66
indexOf	67
Format Functions	67
format(dateTime dt)	67
format(dateTime dt, string s)	67
format(decimal m, string fmt)	67
format(double d, string fmt)	68
format(real r, string fmt)	68
format(string s)	68
format(timeInterval ti)	68
format(timeInterval ti, string type)	68
format(timeInterval ti, string fmt, string char)	68
indexOf	68
lastIndexOf	69
length	69
middle	69
replace	70
right	70
substring	70
toLowerCase	71

toUpperCase	71
trim	72
System Function	73
getSystemProperty	73
Client System Functions	75
closePopupView	80
closeWindow	80
displayedUnits	80
getClientVariable	80
getCookie	81
isInRole	82
navigate	83
setClientVariable	84
setCookie	84
setViewVariableValue	86
showView	86
startHistoryNavigationSlideShow	88
subscribeToClientVariable	88
VTQ Functions	90
quality	91
setQuality	92
time	92
Metadata Function	93
getProperty	93
Type Conversion Functions	95
Coercion Function Errors	98
Supported Hash Functions	98
Conversion Functions with Radix Parameter	98
Parse Functions	98
Date/Time Functions	99
Time Manipulation	102
Date and Time Data Types	102
Considering Time in Different Locations	102
Time Zones and XHQ	102
DST and XHQ	103
Working with Date, Time and Time Intervals	104
Using AM and PM	104
Default Date/Time Format	105

Date Format Rules	105
Time Format Rules	105
Expressing Time Intervals	106
Time Interval Format Rules	107
Using the Date/Time Functions	107
Defining Parameters	108
Using the format_string parameter	111
Using the interval_string parameter	114
date	116
dateAdd with String	116
dateAdd with TimeInterval	117
dateDiff with DateTime	118
dateDiff with String	119
dateEnd	121
dateFormat with DateTime	122
dateFormat with String	122
dateInterval	123
dateOffset with DateTime	124
dateOffset with String	124
datePart	125
dateRound	126
dateSerial	127
dateStart	127
dateSub with String	128
dateSub with TimeInterval	129
dateValue	130
intervalPart	131
now	132
timeSerial	132
Units of Measure Functions	134
baseUnits	134
convertUnits	134
unitsCategory	135
Late-Binding Reference Functions	136
About ReferenceTo	137
Late-Binding Reference Aliases	137
Late-Binding for constructed names to aliases or solution members	138
Using Reference Methods with Historical Aggregate Methods	138

Late-Binding Subscription Functions	139
About SubscribeTo	139
Late-binding for constructed names to aliases or solution members	141
Abbreviated Naming	142
Miscellaneous Functions	143
Historical Mode Function	144
isInHistoricalMode	144
Log Functions	144
logMessage	144
logMessageTS	144
Cache Collections and NVARCHAR2	144
Action Link Functions	145
Java Applet Scripting Functions	146
createIdxApplet	147
getClientTimeAsDate	148
getClientTimeAsLong	148
getClientTimeAsString	148
getCollectionData	148
getHistoricalData() and getHistoricalDataEx()	149
getMemberInfo	150
getQualityUtil	150
getViewInfo()	150
onAfterShowView	151
onBeforeShowView	151
onHistMode	152
onShowView	152
setHistMode	153
setTrendPaths	153
setTrendScooter	154
setTrendTime	154
showView	155
JavaScript Call Function	156
Collection Data Functions	156
Historical Data Functions	165
MemberInfo Functions	170
QualityUtil Functions	171
ViewInfo Functions	175
Appendices	177

A - Overflow, Underflow, and Type Conversion Policies	178
About the Up-casting Mode, Overflows, and Underflows	178
Binary Arithmetic Operations	179
Unary Arithmetic Operations and Mathematical Functions	180
Handling of STRING and TEXT types	180
B - XHQ Data Mapping	181
Database to XHQ	181
For JDBC-based Connectors	182
For the OPC Connector	182
For the PHD Connector	183
For the PI Connector	183
For the IP.21 Connector	184
For the XHQ Connector	184
For the Documentum Connector	185
For the SIMATIC IT Historian Connector	185
For the SAP Connector	186
For the XML Connector	186
C - Units of Measure Conversion	187
How units are converted	188
Conversion Chart	189

About This Guide

Conventions Used in This Guide

The following formatting cues are designed to allow you to quickly locate and understand the information provided in this guide.

Formatting Conventions

Convention	Example
Acronyms are spelled out the first time they appear.	Alert Notification System (ANS)
Bold is used for menu names, command options, and dialog box names in primary task procedures.	From the XHQ Workbench , go to the Add menu and click New Component .
<i>Italic</i> is used for glossary terms.	The first step in building this model is to develop reusable software building blocks, called <i>components</i> .
A monospaced font is used for program and code examples.	The subdirectory <code>\log</code> is automatically created below the location you choose. All log files are written to this subdirectory. <code>C:\XHQ</code>
Key combinations appear in uppercase, bold. If joined with a plus sign (+), press and hold the first key while you press the remaining keys.	CTRL+B
The .x (in italics) is used to indicate release numbers of a product.	Enable (by checking) the Use Java x.x.x_xx for <applet> option.
In See Also notices, sub-chapter headings are in italics, chapter headings are in quotes, and guide titles are in bold.	For more information, go to the <i>About install.properties</i> topic located in the "Working with PROPERTIES Files" chapter of the XHQ Administrator's Guide .

Visual Cues for Online Viewing

This document uses the following styled paragraphs.

Notes are used to offer information that supplement important points of the main text. Tips suggest certain techniques and procedures that may help you achieve your task quickly.



Depending on your network configuration, include domain information only if the domains are different.

See Also notices provide you with additional references to similar topics and/or concepts within the documentation set. Sub-chapter headings are in italics, chapter headings are in quotes, and guide titles are in bold.



For more information, go to the About the Options Menu topic located in the "Working with PROPERTIES Files" chapter of the **XHQ Administrator's Guide**.

Web References point you to external web sites that give additional information on the given topic.



Refer to Microsoft support information with regards to the various server settings for application performance and network utilization.
<http://support.microsoft.com>

Tips provide additional hints to help you use the product more efficiently.



Use the `NavbarWestVerticalOffset` property to make fine adjustments in pixels. The upper, left-hand corner is the origin. The positive horizontal direction moves to the right and the positive vertical direction moves down.

Important notices provide information that are required to completing a given task.



XHQ must run as a domain user.

Warnings tell you that failure to take or avoid a certain action could result in loss of data or application malfunction.



WARNING

Do not modify the `shutdown.dat` template file.

Related XHQ Product Documentation

The XHQ documentation set includes the following titles.

XHQ Documentation Set

Title	Target Audience
XHQ Administrator's Guide Provides the steps required to begin administering XHQ. It also covers security and access, property settings, redundancy, and localization.	Administrators
XHQ ANS User's Guide Learn how to use and administer the XHQ Alert Notification System (XHQ ANS).	ANS Users, Administrators
XHQ Backup and Recovery Guide Learn how to properly backup XHQ.	Administrators
XHQ Connection Guide Provides information on injecting an XHQ-supported connector type and configuring the connection.	Connector Developers
XHQ Developer's Guide Introduces the XHQ Development Client (Workbench and Solution Builder) user interface and provides information on how to set-up XHQ, develop reusable components, create views, and build a solution hierarchy.	Content and Solution Developers
XHQ Getting Started Gives you step-by-step instruction on how to set up your model and solution.	Content, Connector, and Solution Developers
XHQ Installation Guide Provides the system requirements, installation instructions, and upgrade information for the current release of the XHQ System.	Administrators
XHQ Integrated Data Gateway Guide Includes information on the ADO.NET and the XHQ OPC UA Server.	Application Engineers, Integrators
XHQ Performance Analytics Guide Learn how to use the Engineering Environment to enable the generation of the processes necessary to extract and transform data for source systems, and populate the XHQ Data Store and Data Mart.	Solution Developers/Users, Analysts
XHQ Performance Management Guide Learn how to use Target Management to monitor performance indicators and eLogs to create shift reports.	Administrators, End Users
XHQ Reference Guide Lists the functions and methods used in XHQ, and provides examples,	Content and Solution Developers

Title	Target Audience
usage notes, and parameter descriptions.	
XHQ Reporting Services Guide	Application Engineers, End Users
Introduces the XHQ Reporting Services and provides instruction on how to connect to an XHQ data source.	
XHQ SDK Reference Guide	Application Engineers, Integrators
Provides a set of development tools that allows you to create applications that extend XHQ. Includes information on the Client API and Web Services.	
XHQ Solution Design and Architecture	Solution Architects
Provides best-practice examples for XHQ solution design. Includes information on tag synchronization.	
XHQ Solution Viewer User's Guide	All End Users
Gives you step-by-step instruction on how to access your solution through a browser client and set browser preferences.	
XHQ System Guide	Administrators, Application Engineers, Integrators
Contains information regarding secure handling of an XHQ implementation.	
XHQ Trend Viewer User's Guide	All End Users
Learn how to use the XHQ Trend Viewer to view both real-time and historical data.	
XHQ Visual Composer Guide	Content Developers
Provides end-user information for the XHQ Visual Composer and associated programs, which are used in the development of presentation content.	

Contacting Customer Support

For general XHQ product support or related questions, pre-registered customer or partner support staff with a valid XHQ customer support agreement may contact the XHQ Customer Support Team using any of the following means:

- **Web Portal**

The support portal leverages a system called GTAC (Global Technical Access Center). GTAC provides one common support entry point for many Siemens products. It is available via this URL:

<https://www.siemens.com/gtac>

Customers must be pre-registered to be able to use the web portal. A log-in can be requested at any time by self-registering in the GTAC portal. Note, the end user "sold to" identifier is needed in order to register.

Use of the support portal is the preferred means to report incidents to the XHQ Customer Support Team unless immediate interactive telephone assistance is required. The support portal is available twenty four hours per day/seven days per week ("24/7").

- **E-mail**

support.xhq@siemens.com

- **Phone Support and Hours of Coverage**

International: +1 (949) 448-7463

U.S. only: +1 (877) 700-4639

The following paid support levels are available:

Bronze Support: 9/5

9 x 5 hours support. 9 hours per day, 5 days per week. Monday to Friday. Daylight Saving Time is honored.

Choice of one coverage zone out of the following options (default: Americas):

- Americas (15-1 GMT)
- Europe (8-17 GMT)
- Asia (1-10 GMT)

Excludes national holidays as defined by the following countries for the related coverage zone:

- USA (Americas)
- Germany (Europe)
- Singapore (Asia)

Example Americas: *Implies coverage from 7:00 AM to 5:00 PM, Pacific Time, Monday to Friday, excluding US national holidays.*

Silver Support: 24/5

24 x 5 hours support. 24 hours per day, 5 days per week. Monday to Friday. Daylight Saving Time is honored.

Choice of one coverage zone out of the following options (default: Americas):

- Americas
- Europe
- Asia

The weekly start/end times of coverage follow the local times of the following countries in each coverage zone:

- California/USA (Americas)
- Germany (Europe)
- Singapore (Asia)

Example Americas: *Implies coverage from midnight on Sunday until midnight on Friday, Pacific Time, Monday to Friday.*

Gold Support: 24/7

24 x 7 hours support. 24 hours per day, 7 days per week.

- **Postal Mail**

Siemens Product Lifecycle Management Software, Inc.

XHQ Operations Intelligence

Attn: XHQ Customer Support Department

6 Journey, Suite 200

Aliso Viejo, CA 92656, USA

General Feedback and Comments

Please send an e-mail to:

info.xhq@siemens.com

Siemens Product Lifecycle Management Software, Inc. and affiliated Siemens Industry Software companies (collectively referred to as "SISW") are committed to working with our customers. Your comments, suggestions, and ideas for improvements are very important to us. Thank you for taking the time to send us your feedback.

1 | Expressions

An *expression* is a way to configure and extend a component using mathematical formulas and Boolean logic. For example, expressions are similar to the entries you can make in the formula bar in Microsoft Excel. It can be as simple as:

```
item1 + item2
```

or as complex as:

```
(item1 + 3 * sin(item2)) / (item3 % 5)
```

The expression language is a functional subset of the expression capability found in most programming languages.

Normally, you enter the appropriate XHQ variable names when configuring a component in the XHQ Solution Builder (*server-side expression*) or "pointing to" a component in the XHQ Workbench (*client-side expression*). You can also enter an expression in place of such variables.

In every expression, there are one or more *operands*, which are the elements to be calculated, which may be separated by calculation *operators*. XHQ calculates the expression from left to right, according to a specific order for each operator in the formula. The evaluation of an expression performs one or more operations, yielding a result.

The simplest form of an expression consists of a single literal constant or variable. In that case, the operand is without an operator.

More commonly, an expression performs a calculation or operation, such as simple addition and subtraction, or more complex operations, such as relational, bitwise, logical, and complex math operations. You can also use expressions in place of any single variable when creating links to URLs and views in the XHQ Workbench.

Server-side versus Client-side

As stated in the previous overview, expressions that are configured from the XHQ Solution Builder are executed within the XHQ Solution Server and are therefore labeled **server-side expressions**. For example, connection expressions are generated when an item is selected in the XHQ Solution Builder and the Connect tab is configured. In this case, a connection could be an expression instead of an item in a Connection Group.

Client-side expressions are configured from the XHQ Workbench and are executed within a view. For example, you can input an expression into the Animation tab of the XHQ Workbench.



For additional information on client-side expressions, refer to the topic, [Working with Client-side Expressions](#), located in the XHQ Developer's Guide.

Self-referencing Server-side Expressions

In cases when values for self-referencing, sever-side expressions never display after the XHQ Server starts, you can configure another point with a valid expression and reference that point in the original point's expression. The values for both appear shortly after the XHQ Server starts up.

Consider the following example.

EXAMPLE

Given: An expression that uses only `getProperty` calls, and is therefore self-referencing.

```
::Top.string1 -- Expression: substring(getProperty("path",string1),0,lastIndexOf(getProperty("path",string1),"."))
```

The value may not appear after start-up. But you can add another point (for example, "real1") and set the expression (for example, to "pi").

```
::Top.real1 -- Expression: "pi"
```

Then reference `real1` in the original point, `string1`:

```
::Top.string1 -- Expression: substring(getProperty("path",real1),0,lastIndexOf(getProperty("path",real1),"."))
```

The value for `string1` appears after the XHQ Server is started.

Client-side Expression Functions

The following expression functions are valid only for client-side (XHQ Workbench):

- [getCookie](#)
- [setCookie](#)
- [navigate](#)
- [showView](#)
- [closeWindow](#)
- [isInHistoricalMode](#)
- [getClientVariable](#)
- [setClientVariable](#)
- [subscribeToClientVariable](#)
- [setViewVariableValue](#)
- [startHistoryNavigationSlideShow](#)

Concepts

Naming Rules and Identifier Types

Identifier names for the operands used in an expression are case-insensitive. Identifiers that are reserved words must be quoted. Identifiers that use "non-identifier" characters (initial character followed by characters, numbers, periods, or underscores) must be quoted with **single quotes**.

The rules for the naming of aliases (global tag names for full global names) are less restrictive than those used for components and members. Because identifiers in systems already in the field don't have to honor reserved words and because aliases can include characters (such as dashes) that can create ambiguities for the parser, **single quotes** are used to quote, or delimit, identifiers. Quoting is only required if an ambiguity exists.

Identifiers for Server-side Expressions

Server-side expressions may include identifiers that are:

- Members
- Global references (full paths prefaced by two colons)
- Members of members (to any depth)
- Aliases (quoted, or prefaced, by cast)



For a server-side expression, in reference to a global alias and not to members, you can use any of the following formats:

`r:aliasName` or `r:'alias_Name'` or `r:`alias_Name``

The quotes are needed if the alias name has characters that are not alpha-numeric or an underscore.



When you enter a zero (0) before a number in a server-side expression, XHQ reads it as an octal (base-8).

If you enter ...

010

012

XHQ reads it as ...

8

10

More about Aliases and Casts

In general, aliases have the following format:

- `cast:alias_name`
- `cast:backquoted_alias`
- `backquoted_alias`
- `cast:quoted_alias`
- `quoted_alias`

Where:

- `alias_name` is a simple name that has only numbers, digits, or underscore characters **and does not** start with a letter or an underscore;
- `quoted_alias` begins with an apostrophe and is terminated by the next apostrophe;
- `backquoted_alias` begins with a grave accent and is terminated by the next grave accent.



The alias reference must be cast unless it is quoted. If no cast is used, it may lead to compile-/import-time errors or incorrect resolution at runtime if the alias changes between the time it is saved and when it is evaluated.

The data type of the alias must be specified with a type cast before the ":" (colon). The following casts are allowed (either upper or lower case):

Alias Cast	Data Type	Alias Cast	Data Type
b:	Boolean	m:	Decimal
i:	Integer	t:	Date/Time
l:	Long	v:	Interval
r:	Real	s:	String
d:	Double	x:	Text

If the specified type for the alias is not the same as the cast value, the value is cast at runtime and precision loss, or BAD quality values, occur if types are incompatible.

Things to note about Alias Names

- Alias names that have a single quote must be quoted by using the back-quote character.
- Alias names that have a back-quote must be quoted by using the single quote.
- Alias names containing both a single quote and a back quote are not supported.



Casted aliases are resolved at expression evaluation time not at compile time.

A quoted alias name (single or back quote) is considered to be an alias of unspecified type. Since the data type of the reference must be known to compile the expression, the alias is resolved at compile time *to discover the type*. This leads to unexpected dependencies if an expression refers to an alias that is not yet defined in the system.

Identifiers for Client-side Expressions

Client-side expressions may include identifiers that are:

- Members
- Members of members (to any depth)
- Local variables (that begin with a dollar sign, \$)

Qualified Values

The output of an expression is a *qualified value*. In addition to its value, it also has a data quality and a timestamp. References used by expressions also have a data quality and a timestamp. Intermediate values within an expression also have a data quality and a timestamp. Constants and literals always have a GOOD data quality and a zero timestamp. Every operation or function takes one or more arguments and **combines** them into a single result. The data qualities and timestamps of the arguments are combined into the result. As a rule, when combining, the newest timestamp is used. A zero timestamp (from a constant or literal) is considered the oldest of all. For data qualities, the worst quality is used when combining.

The data quality hierarchy is as follows:

- UNCERTAIN is worse than GOOD;
- BAD is worse than UNCERTAIN.

Data Types

The *data types* used in expressions are the:

- **Primitive types**
Used in XHQ.
- **Component types**
Used to reference a component rather than the contents of the component.
- **Data quality types**
Used only inside an expression to permit data quality comparisons.

XHQ Data Type Groups

Group	Members
Simple Numeric Types	Integer, Long, Real and Double Types
Numeric Types	Simple Numeric Types plus the Decimal type
Non-Floating Types	Integer and Long Types
Simple Floating-Point Types	Real and Double Types
Floating-Point Types	Simple Floating-Point Types plus the Decimal Type
32-bit Numeric Types	Integer and Float Types
64-bit Numeric Types	Long and Double Types
Time Types	Date/Time and Interval Types
Ordered Types	Numeric Types and the Date/Time Type
Textual Types	String and Text Types
Actual Value Types	Numeric Types, Time Types and Textual Types (but excluding the Boolean Type and the Quality pseudo-type)
All Value Types	Actual Value Types plus the Quality pseudo-type

Things to Note about Data Types

- In expressions, constant, non-floating point numbers < Integer.MAX_VALUE are of type **Integer** by default.
- Likewise, constant, floating-point numbers < Float.MAX_VALUE are of type **Real** by default.

About Constant Postfix

A constant postfix is appended to constant strings that represent numbers in smaller data type to force them to represent a larger type. The supported data types are Long, Double, and Decimal.

For example, the integer "12" can be forced to a long by adding the letter "L" (either capitalized or lowercase) to the end.

Examples: The integer 12
 The long 12L

Data Type Constant Postfix

Long	L or l
Double	D or d
Decimal	M or m

Things to Note about Constant Postfixing

- When you postfix a constant, non-floating point number < Integer.MAX_VALUE with 'l' or 'L', that makes it a **Long**.
- When you postfix a constant, floating point number < Float.MAX_VALUE with 'd' or 'D', that makes it a **Double**.
- If you postfix either type of constant with an 'm' or 'M', that makes it a **Decimal**.

Type Conversions

The following table lists the automatic type conversions within an expression given operands of different types:

Type Conversions

	Integer	Long	Real	Double	Decimal
Integer	Integer	Long	Real	Double	Decimal
Long	Long	Long	Double	Double	Decimal
Real	Real	Double	Real	Double	Decimal
Double	Double	Double	Double	Double	Decimal
Decimal	Decimal	Decimal	Decimal	Decimal	Decimal

Numeric Constants

When creating expressions you can differentiate between various numeric types as well as indicate a specific type for a numeric constant (cast).

The lexical analysis of a numeric constant input allows the expression author to differentiate between various numeric types as well as to be able to specify a specific type for numeric constant (cast). The following table shows the various numeric types and allowed input and the effect of various casting specifiers.

Numeric Types

Type	Decimal Range	Octal Range	Hexadecimal Range
INT	0 - 2147483647	00 - 017777777777	0x0 - 0x7FFFFFFF
LONG	2147483648 - 9223372036854775807	020000000000 - 07777777777777777777	0x80000000 - 0x7FFFFFFFFFFFFFFF
REAL	$2^{-149} - (2 \cdot 2^{-23}) \cdot 2^{127}$	<Not Supported>	<Not Supported>
DOUBLE	$2^{-1074} - (2 \cdot 2^{-52}) \cdot 2^{1023}$ (unless REAL)	<Not Supported>	<Not Supported>
DECIMAL	> 9223372036854775807, (other floating)	> 07777777777777777777	> 0x7FFFFFFFFFFFFFFF

Time Types

Date/Time and Interval types are now full-fledged XHQ types. Here are a couple of things to be aware of with regards to how time inputs/outputs are now handled:

- **Time Inputs**

In earlier XHQ releases, where a Time type was used as an input to an expression, it was necessary to provide an expression with a return value of that Time type as the input. In this current release, it is possible to also provide a reference to a Time type data member as well.

- **Time Outputs**

In earlier XHQ releases, where an expression had a Time type output, it was not possible to set the value of a data member directly from that output (for example, in a server-side expression). This function is now supported.

Expressions and Collection Members

For expressions, it is helpful to remember what the sibling members are of the member for which you are writing the expression. This reduces the confusion when considering which member names are valid in which contexts.

For example, as can be seen from the Inventory Items panel of the XHQ Solution Builder, a collection member is a sibling of the other members listed. So, when you are writing an expression for the given collection member, your expression can only refer to sibling members, which are listed alongside it in Inventory Items panel .

Basically, there are two sets of members:

1. The global collection members that you can use in the constraint (SQL).
2. The object's members (from the Inventory Items panel) that can be used in the expression that is embedded in the constraint.

Consider the following example.

Given:

A component has the following members: `colRecords` (collection link), `sPlantFilter` (string), `iCount` (integer), and `iMinDays` (integer).

A global collection named `PlantEquipment` has the following members: `sPlant` (string), `sEqName` (string), `sEqType` (string), and `tEqInstall` (Date/Time).

For the `colRecords` collection link, the Link tab of the XHQ Solution Builder can be configured using either of the following example constraints.

Example 1:

```
WHERE sPlant LIKE %sPlantFilter%
```

This first example is the "classic" style that allows you to reference a sibling member of the collection link member.

Example 2:

```
WHERE sPlant LIKE {xhq-expr sPlantFilter}
```

This second example is that of a "full" expression syntax that allows you to compose more complex expressions.

If, for example, you want to know which pieces of equipment have been in service for more than a certain number of days, but only if the `iMinDays` member has a configured value; otherwise, a default value is used. Then, the expression can be *expanded* to:

```
WHERE sPlant LIKE %sPlantFilter%
AND tEqInstall > {xhq-expr DateAdd( "day", iff(quality(iMinDays) == GOOD, -iMinDays, -180)) }
```



Give special attention to the timing of these operations to make sure that errors do not occur. For instance, it may seem possible to use a sibling member to get the count of rows in the collection, and then use that in the link constraint. This, however, causes a circular reference and would lead to ambiguous results, and is therefore not supported.

Late-binding Expressions

Both the client-side and the server-side expression subsystems support the late-binding of names to solution members or aliases through the use of the **SubscribeTo** or the **ReferenceTo** function.



For more information on these functions, refer to the topics, [Late-Binding Subscription Functions](#) and [Late-Binding Reference Functions](#).

When writing expressions, use late-binding functions to dynamically construct reference names at runtime.

For example, you have a member that has tank, and you want to display the tank level. Given the tank number is "iProdTankNo", you could use the following expression to late-bind the correct tank level tank based on the tank number:

Example 1: `SubscribeToReal("TK-" + iProdTankNo + ".LEVEL")`

This expression constructs a string based on the chaining value of "iProdTankNo" member. If this value is 101, then the resulting string is "TK-101.LEVEL". This string is passed as the argument to the **SubscribeToReal** method, which then subscribes to the TK-101.LEVEL tag and produces the value of the tag as it changes.

In addition, these late-binding reference methods can be chained with historical aggregate methods, allowing aggregate data to be obtained for members (tags) that are determined only when the expression is evaluated.

For example, the following expression is placed on a member of a collection to get the hourly average of a tank level that is determined by other members on the same row of data.

Example 2: `rRef(sTankName + ".LEVEL").avg(DateAdd("hour", -1), Now())`



This example uses the [abbreviated naming](#) convention. **rRef** is equivalent to **ReferenceToReal**.



The late-binding expressions mentioned in this guide should only be used in circumstances when an alias or full path to a primitive member cannot be known upfront. In other words, only use late-binding expressions in scenarios where part or the entirety of the alias name or the full path depends on values that can only be evaluated at runtime (such as values retrieve from a backend, client-side expressions, and other forms of data only available in runtime).

If the entire alias or full primitive member path can be determined upfront - and not just at runtime - then use other syntax, similar to the following examples, since these have significant performance advantages over the late-binding technique.

Example for Real data type:

`r:'alias' Or r:'fullPrimitiveMemberPath'`

Example for String data type:

`s:'alias' Or s:'fullPrimitiveMemberPath'`

For server-side expressions, you can disable/enable support for late-binding references by setting the solution property, **ExprItem.AllowLateBinding**. Note, if a late-binding reference is encountered in an expression at runtime where it is **not allowed** (disabled), then the result of the expression is BAD and no usable value is returned; for a primitive, the resulting quality is BAD, Access Denied



For more information, go to the topic, [Using solutionsettings.properties](#), located in the XHQ Administrator's Guide.

About the Evaluation Schedule

The evaluation schedule for these items is context dependent.

The **client-side** (the XHQ Solution Viewer) evaluates late-bound references and returns the current value of the referent item based on the input (string parameter) and the data update events of the late-binding member/tag. In other words, if the input string is an expression of other values that are changing, this may cause the resulting member name to change. If this is the case, the newly determined name is subscribed and a value results.

However, even if the input to the name does not change, the referent member may post data updates, causing the expression to be evaluated.



Typically, aggregates are handled with the five minute throttle on the client-side.

On the **server-side**, there are different evaluation schedules depending on the context of the expression. Expressions on members of global collections are evaluated only when the collection is updated. There is also the possibility, however, that some of the referenced members were never subscribed before the evaluation occurred. In this case, the collection processing and expression evaluation uses a two-pass mechanism to subscribe all referenced members in a block of inserted rows so that coherent results may be obtained.



Once the collection processing is finished with the updates to the cache, no more evaluations are performed on the expressions. In other words, when the results are stored in the cache, it is final.

Normal primitive members (not on collections) can also use late-bound expressions. Current-value methods are done on an update basis much like the client-side. However, aggregate methods must be configured with a schedule.



For more information, go to the topic, [*The Server-side Expression Scheduler*](#), located in the XHQ Developer's Guide.

Using Formatting Functions in Expressions

Consider the following example:

```
"Tag Value: " + RealToString(rCurrentValue)
```

It is important to note that functions such as **RealToString** and **DoubleToString**, for example, return **raw data** in string format. These functions are useful for passing values from the applet to JavaScript. They are, however, not useful in a Tooltip (for example).

In the case of a Tooltip expression, the problem with these functions is that raw data values can potentially have several digits trailing a decimal point. So, a raw data value of "21.2767888999999923" displayed in a Tooltip at runtime may be excessive and unnecessary for the end user.

To address this issue, use the **format()** function, as seen in the following examples:

- `String format(float number, string formattingPattern)`
- `String format(double number, string formattingPattern)`
- `String format(decimal number, string formattingPattern)`

Where:

Parameter	Description
number	Is the number to be formatted.
formattingPattern	Is the formatting pattern that supports the standard number formatting characters such as the pound (#) sign. Web Reference: For a list of supported standard number formatting characters, go to: http://docs.oracle.com/javase/7/docs/api/java/text/DecimalFormat.html
Returns:	A formatted number that is rounded up.
Example:	<code>format(21.2767888999999923, "0.###")</code> Returns: "21.277"



These number formatting functions are supported in both client- and server-side expressions.

So, referring back to the original example at the beginning of this topic, `"Tag Value: " + RealToString(rCurrentValue)` becomes `"Tag Value: " + format(rCurrentValue, "0.###")`.

Using Expressions in the XHQ Interactive Trender

With the support for expressions in the XHQ Interactive Trender, you can reference multiple members in these expressions, using their full path name within the XHQ model, or a suitable alias.

EXAMPLES: VALID PEN EXPRESSIONS FOR THE INTERACTIVE TREND VIEWER

```
::MySolution.Component1.IntVal + 'intAliasName'
::MySolution.Component1.RealVal
'realAliasName' + 'realAliasName'
2*::MySolution.Component1.IntVal
sin (realAliasName)
```



Use the fully, qualified global path for the object. The path must begin with two colon characters (::) and each member in the path must be separated by a period (.).

Also, tag names in tag calculations need to be enclosed in single quotes (' '). Consider the following examples:

'tag1'/1000

'tag1'*1000

'tag1' + 'tag2'

'tag1' - 'tag2'

'tag1'*2 - 'tag2'

The expression operators and functions you can use include:

+	%	atan2	floor	pow	sqrt
-	abs	ceil	log	random	tan
*	acos	cos	max	round	todegrees
/	atan	exp	min	sin	toradians



For more information on the Interactive Trend Viewer, go to the topic, [Using the XHQ Trend Viewer](#), located in the XHQ Trend Viewer User's Guide.

Expressions and Aliases

Currently, only simple, formatted alias names are supported in expressions with more than one reference. Complex aliases can only be used if the expression consists of exactly one alias reference – the complex alias itself. Otherwise, complex aliases must be enclosed by single quotes.

Expressions Containing More than One Reference

For a pen expression, the history for all reference members is retrieved using the Interpolated mode with the same time span for all members. Pen expressions with a single reference support all aggregate types. However, in any time-sliced mode, multi-reference expressions will only display historical data, not real-time data.



If all reference histories of an expression are in step-wise interpolation mode, then the resulting expression is also in step-wise interpolation mode.

2 | Operators

Arithmetic Operators

Arithmetic operators perform simple arithmetic functions, such as addition, subtraction, multiplication, and division. The following table lists the arithmetic operators supported by XHQ.

Arithmetic operators are placed between two operands to perform an arithmetic function.

Arithmetic Operators

Operator	Description	Applicable Types	Example
+	Placed before a variable in an expression to indicate a positive value.	Numeric Types	+Item1
-	Placed before a variable in an expression to indicate a negative value.	Numeric Types	-Item2
+	Returns the sum of the two variables or values.	Numeric Types*	Item1+Item2
-	Subtracts the second value from the first and returns the result.	Numeric Types	Item1-Item2
*	Multiplies the two values and returns the product.	Numeric Types	Item1*Item2
/	Divides the first value by the second and returns the result.	Numeric Types	Item1/Item2
%	Computes the remainder of division between two values. It can be applied only to operands of the integer type. The left operand of the modulus operator is the dividend. The right operand is the divisor.	Numeric Types	Item%Item2

*In addition to numeric types, the '+' operator can also support textual types, in which case it becomes a concatenation operator for string and text values.

Relational Operators

Relational operators perform comparative analysis on two values and evaluate it as either true or false.

Relational Operators

Operator	Description	Applicable Types	Example
==	Returns TRUE if the first value is equal to the second value.	All Value Types	x==y (Is x equal to y?)
!=	Returns TRUE if the first value is not equal to the second value.	All Value Types	x!=y (Is x not equal to y?)
<	Returns TRUE if the first value is less than the second value.	Ordered Types	x<y (Is x less than y?)
>	Placed between two variables to determine if the first value is greater than the second value.	Ordered Types	x>y (Is x greater than y?)
<=	Returns TRUE if the first value is less than or equal to the second value.	Ordered Types	x<=y (Is x less than or equal to y?)
>=	Returns TRUE if the first value is greater than or equal to the second value.	Ordered Types	x>=y (Is x greater than or equal to y?)

Bitwise Operators

Bitwise operators interpret operands as an ordered collection of bits. A bitwise operator allows the user to test and set individual bits or bit subsets. Bitwise operators either shift bits to the left or right or perform arithmetic manipulations.

Bitwise Operators

Operator	Description	Applicable Types
~	Placed to the left of a value, this operator "flips" the bits of its operand. Each 1 bit is set to 0; each 0 is set to 1.	Non-Floating Types
>>	Shifts the bits of the left operand some number of positions to the right.	Non-Floating Types
<<	Shifts the bits of the left operand some number of positions to the left.	Non-Floating Types
>>>	Shifts the bits of the left operand some number of spaces to the right if they are unsigned.	Non-Floating Types
&	The bitwise AND operator ("&") takes two integer operands. For each bit position, the result is a 1-bit if both operands contain 1-bits; otherwise, the result is a 0-bit.	Non-Floating Types
	The bitwise OR (inclusive) operator (" ") takes two integer operands. For each bit position, the result is a 1-bit if either or both operands contain a 1-bit; otherwise, the result is a 0-bit.	Non-Floating Types
^	The bitwise XOR (exclusive) operator ("^") takes two integer operands. For each bit position, the result is a 1 bit if either but not both operands contain a 1-bit; otherwise, the result is a 0-bit.	Integer and Long Types

Logical Operators

Logical operators evaluate Boolean values (values that are either TRUE or FALSE) to determine if an expression is TRUE or FALSE. An expression using Logical operators returns TRUE if all arguments are true and FALSE if one or more arguments is false.

For example, the expression `pump.status&&logical12` equals TRUE if both values are TRUE, and FALSE if one or both values are FALSE.

The logical AND ("&&") operator evaluates to TRUE only if both of its operands evaluate to TRUE. The logical OR ("||") operator evaluates to TRUE if either of its operands evaluate to TRUE. The operands are evaluated from left to right. Evaluation stops as soon as the truth or falsity of the expression is determined.

Given the forms:

`expr1 && expr2`

`expr1 || expr2`

Expr2 is not evaluated if either of the following is true:

- In the logical AND expression, `expr1` evaluates to FALSE; or,
- In the logical OR expression, `expr1` evaluates to TRUE.

Logical Operators

Operator	Description	Applicable Types
&&	Logical AND Returns TRUE if all arguments are true and FALSE if one or more arguments is false.	Boolean
	Logical OR Returns TRUE if any argument is true and FALSE if all arguments are false.	Boolean
^	Logical XOR Returns TRUE if one and only one argument is TRUE.	Boolean
!	Logical NOT Returns FALSE if the operand or expression is true; otherwise, returns TRUE.	Boolean

Comma Operator

Commas can be used to separate a "chain" of multiple expressions.

```
<expression1>, <expression2>, <expression3>, <expression4>, . . .
```

The comma operator can be used for cases where an application calls for more than one action or on a button press (for example, saving separate cookies for three different controls). A comma is a binary operator (using two operands), but can be "chained" together with operands of any data type.

Example: `setCookie("editBox_1"), setCookie("comboBox_2"), setCookie
 ("editBox_2")`

However, the value of the first operand is always returned. Therefore, the above example will not work for applications where it is important to indicate the success or failure of *all* the SetCookie functions, because only the *first* SetCookie call is returned. So, using the logical AND operator (&&) to separate the expressions would be more accurate.

Example: `setCookie("editBox_1") && setCookie("comboBox_2")
 && setCookie("editBox_2")`

Operator Precedence

Operator precedence – the order in which operators are evaluated in a compound expression – is important to understand so that you can avoid common sources of program error. Priorities are in descending order. Therefore, the greater the priority number, the higher the precedence. For example, multiplication (priority #12) has a higher precedence than addition (priority #11). This means that multiplication is evaluated first.

Operators that have the same precedence are evaluated from left to right.

As shown in the following table, parentheses have the highest priority. Parentheses mark off subexpressions. In the evaluation of a compound expression, the first action is to evaluate all parenthetical subexpressions. Each subexpression is replaced by its result; evaluation then continues. Innermost parentheses are evaluated before outer pairs.

Operator Precedence

Priority	Operator
15	()
14	! ~
13	RESERVED
12	* / %
11	+ -
10	<< >> >>>
9	< > <= >=
8	== !=
7	&
6	^
5	
4	&&
3	
2	, (comma)
1	RESERVED

3 | Constants



As with all XHQ expression syntax, identifiers are case-preserving and case-insensitive.

XHQ Constants

Constant	Type	Value	Meaning
@bad	Quality	0x00	A bad quality.
@double_e	Double	2.718281828459045	The double value closer than any other to e, the base of the natural logarithms.
@double_pi	Double	3.141592653589793	The double value closer than any other to pi, the ratio of the circumference of a circle to its diameter.
@e	Real	2.718281	The real value closer than any other to e, the base of the natural logarithms.
@epoch	DateTime	1 JAN 1970 00:00:00 UTC	The “zero” time (epoch) that marks the start of the calendar upon which all XHQ date/time calculations are performed.
@epoch_dos	DateTime	1 JAN 1980 00:00:00 UTC	Start of the DOS epoch used for expressing date/times relative to MS-DOS systems.
@epoch_iso8601	DateTime	1 JAN 1 AD 00:00:00 UTC	Starting date defined by ISO-1601 standard. The Calendar used to set the date and year of Jan 1, 1 AD is the Proleptic Gregorian Calendar (which has no date discontinuities at the Julian-Gregorian changeover).
@epoch_ntfs	DateTime	1 JAN 1601 00:00:00 UTC	FILETIME/NTFS epoch date
@epoch_ntp	DateTime	1 JAN 1901 00:00:00 UTC	Network Time Protocol (NTP) epoch date
@epoch_posix	DateTime	1 JAN 1970 00:00:00 UTC	<i>See @epoch.</i>
@false	Boolean	false	Equivalent to a false condition.
@good	Quality	0xC0	A good quality.
@pi	Real	3.141592	The real value closer than any other to pi, the ratio of the circumference of a circle to its diameter.
@real_e	Real	2.718281	<i>See @e.</i>
@real_pi	Real	3.141592	<i>See @pi.</i>

Constant	Type	Value	Meaning
@true	Boolean	true	Equivalent to a true condition.
@uncertain	Quality	0x40	An uncertain quality.

The following **legacy constant** names (without the prefix @ symbol) are still accepted.

XHQ Legacy Constants

Constant	Type	Meaning
bad	Quality	A bad quality.
double_e	Double	The double value closer than any other to e , the base of the natural logarithms.
double_pi	Double	The double value closer than any other to π , the ratio of the circumference of a circle to its diameter.
e	Real	The real value closer than any other to e , the base of the natural logarithms.
false	Boolean	Equivalent to a false condition.
good	Quality	A good quality.
pi	Real	The real value closer than any other to π , the ratio of the circumference of a circle to its diameter.
real_e	Real	Same as e .
real_pi	Real	Same as π .
true	Boolean	Equivalent to a true condition.
uncertain	Quality	An uncertain quality.

4 | Methods

Methods, like functions, are used to "act" on XHQ components. The difference is that methods are directly associated with the component. Currently, XHQ supports aggregates for collection and historical data.

Collection Aggregate Methods

XHQ allows two types of aggregate methods: **Collection Methods** and **Collection Member Methods**.



About Rollups

Rollups are calculations across an entire collection that "rollup" information about the collection. The support calculations are in the form of aggregate methods that include the total, average, minimum and maximum values for a particular numeric (primitive) member of the collection or the number of rows in the collection. You use aggregate methods wherever expressions can be entered, either from the server-side within the XHQ Solution Builder or from the client-side within the XHQ Workbench

Collection Methods

Aggregate Collection methods return information about the entire collection or results on calculations against the collection itself. The collection method returns the results as they pertain to the entire result set (objects satisfying the collection query) on the server, not the subset of the result set currently displayed as views in the client.

The Collection Method **syntax** is as follows:

```
[<object-member>.*<collection-member>.<method-name>()]
```

XHQ supports the **count ()** collection method.

Collection Method

Method	Return Value	Description
count ()	Integer	Returns the total number of "records" within the collection's result set.

Syntax: [<object-member>.*<collection-member>.count()]

Example: ColalInventoryLots.count()

Collection Member Methods

Aggregate Collection Member methods return information about a member within the collection or results on calculations against the member within the collection. The collection member method returns the results as they pertain to the entire result set (objects satisfying the collection query) on the server, not the subset of the result set currently displayed as views in the client.

Collection member method **syntax**:

```
[<object-member>.*<collection-member>.<data-member>.<method-name>()]
```

XHQ supports the **avg()**, **max()**, **min()**, **sum()**, and **count()** collection member methods.

Collection Member Methods

Method	Member Type	Return Value	Description
Avg ()	32-bit Numeric	Real	Returns the numerical average of the specified data member for all the non-NULL "records" within the collection's result set.
	64-bit Numeric	Double	
	Decimal	Decimal	
	Boolean	Real (duty cycle)	
	DateTime	Not supported	
	Interval	Interval	
	String	Not supported	
	Text	Not supported	

Syntax: [<object-member>.*<collection-member>.<data-member>.avg()]

Example: ColalInventoryLots.numberOfCases.avg()

Max ()	Integer	Integer	Returns the largest value found for the specified data member for all the non-NULL "records" within the collection's result set
	Real	Real	
	Long	Long	
	Double	Double	
	Decimal	Decimal	
	Boolean	Not supported	
	DateTime	DateTime	
	Interval	Interval	
	String	Not supported	
	Text	Not supported	

Syntax: [<object-member>.*<collection-member>.<data-member>.max()]

Example: ColalInventoryLots.numberOfCases.max()

Min ()	Integer	Integer	Returns the smallest value found for the specified data member for all the non-NULL "records" within the collection's result set.
	Real	Real	

Method	Member Type	Return Value	Description
	Long	Long	
	Double	Double	
	Decimal	Decimal	
	Boolean	Not supported	
	DateTime	DateTime	
	Interval	Interval	
	String	Not supported	
	Text	Not supported	

Syntax: [<object-member>.*<collection-member>.<data-member>].min()

Example: ColalInventoryLots.numberOfCases.min()

Sum ()	32-bit Numeric	Real	Returns the total value of the specified data member for all the non-NULL "records" within the collection's result set.
	64-bit Numeric	Double	
	Decimal	Decimal	Using the Sum () function for a Real or an Integer column may return unexpected results, as both are treated as Float. For this case, it is recommended to use a Double or Long member type for the global collection where the numbers are expected to exceed $2^{23} = 8388608$.
	Interval	Interval	

[For more information on 32- or 64-bit Numerics, see the topic, Data Types.](#)

Syntax: [<object-member>.*<collection-member>.<data-member>].sum()

Example: ColalInventoryLots.numberOfCases.sum()

Count ()	All types	Integer	Returns the count of non-NULL values of the specified data member for all the "records" within the collection's result set.
-----------------	-----------	---------	--

Syntax: [<object-member>.*<collection-member>.<data-member>].count()

Example: ColalInventoryLots.numberOfCases.count()

XHQ also supports two forms of **Standard Deviation** and **Variance**: population and sample.

Standard Deviation and Variance Methods

Method	Member Type	Return Value	Description
stddev_pop ()	Boolean	Boolean	Computes the <u>population</u> standard deviation.
	Integer	Real/Double	Returns the square root of the population variance of the specified data member for all the non-NULL "records" within the collection's result set.
	Real	Real	
	Long	Long	

Method	Member Type	Return Value	Description
	Double	Double	
	Decimal	Decimal	

Syntax: [<object-member>.*<collection-member>.<data-member>].stddev_pop()

Example: ColaInventoryLots.numberOfCases.stddev_pop()

stddev_samp()	Boolean	Boolean	Computes the <u>cumulative</u> standard deviation.
	Integer	Real/Double	Returns the square root of the sample variance of the specified data member for all the non-NULL "records" within the collection's result set.
	Real	Real	
	Long	Long	
	Double	Double	
	Decimal	Decimal	

Syntax: [<object-member>.*<collection-member>.<data-member>].stddev_samp()

Example: ColaInventoryLots.numberOfCases.stddev_samp()

var_pop()	Boolean	Boolean	Returns the population variance of a set of numbers of the specified data member for all the non-NULL "records" within the collection's result set.
	Integer	Real/Double	
	Real	Real	
	Long	Long	
	Double	Double	
	Decimal	Decimal	

Syntax: [<object-member>.*<collection-member>.<data-member>].var_pop()

Example: ColaInventoryLots.numberOfCases.var_pop()

var_samp()	Boolean	Boolean	Returns the sample variance of a set of numbers of the specified data member for all the non-NULL "records" within the collection's result set.
	Integer	Real/Double	
	Real	Real	
	Long	Long	
	Double	Double	
	Decimal	Decimal	

Syntax: [<object-member>.*<collection-member>.<data-member>].var_samp()

Example: ColaInventoryLots.numberOfCases.var_samp()

SIDEBAR: HANDLING BAD AND UNCERTAIN QUALITIES

Collection member methods set the quality based on whether there are "bad" data (for example, NULLs) in the records. There are, however, cases when "bad" values are normal.

Consider the following example.

A `sum()` method is placed on a view and one of the records in the collection has a "bad" value. The `sum()` value appears as an "uncertain" quality (displaying in the "uncertain" color). In this case, however, "bad" values are considered normal. Therefore, you force the "uncertain" value to display in the same color as a "good" value.

There are a couple of ways to do this.

- **Set the same color**

Change the color of "uncertain" values to the same color as "good" values. The downside of this approach is if you have a lot of different colors for the value (that is, if all are not black), it can be tedious to set both the "good" and "uncertain" colors each and every time.

- **Use the `setQuality` function**

This overrides the quality returned by the collection member method.

Example: Instead of using: `Orders.TotalValue.sum()`

Use: `setQuality(Orders.TotalValue.sum(), GOOD)`

- **Alter the query**

Change the query to ensure that any null values are changed to valid values (for example, using the `NVL` function in Oracle to change a null string to " ").

Historical Access and Aggregation Methods

About Historical Access Methods

The purpose of Historical Access methods is to retrieve historical data from any XHQ-supported historian. These methods work in conjunction with date/time expressions, forming requests that allow you to access historical values of any member (given that history is defined for that particular member).

Currently, XHQ supports the following methods:

- **at** (.at)
This method uses interpolation to derive a value at a given time.
- **actual** (.actual)
This method returns the actual value for the time specified or the first actual prior to that time.



For the list of XHQ-supported historians, refer to the topic, [Supported Back-end Data Sources and Historians](#), located in the XHQ Installation Guide.

List of Historical Access Methods

Historical Access methods use the **.methodname ()** syntax used by collection aggregates.

Quick Reference – Historical Access Methods

Method	Description
.at (date)	The interpolated value for the specified date.
.at (interval_string, intervalnumber)	The interpolated value for the specified interval before the current time.
.actual (date)	The actual value for the specified date or the first actual prior to that date.
.actual (interval_string, intervalnumber)	The actual value for the specified interval before the current time.

.at(date)

The interpolated value for the specified date.

Parameter	Description
date	The date must be the result of another time function. See Also: For more information on time functions, refer to the topic, Date Time Functions .

Example(s): `aMember.at (DateValue ("07-24-00 10:30"))`

Where "aMember" is the name of a particular member with history defined.
This expression returns an interpolated value for July 24, 2000 at 10:30 am.

`.at(interval_string, intervalnumber)`

The interpolated value for the specified interval before the current time.

Parameter	Description
interval_string	A unit of time. Common parameters include: "year"; "month"; "week"; "hour"; "y" (which is the "day of the year" and not the year).
intervalnumber	Do <u>either</u> of the following: <ul style="list-style-type: none"> Enter a negative time interval that is preceded with a minus sign (-) to get the historical value. This is the negative offset. OR <ul style="list-style-type: none"> Enter a positive time interval to get future data.
Example(s)	<pre>aMember.at("hours",-3)</pre> <p>Where "aMember" is the name of a particular member with history defined. This expression returns the interpolated value at three hours before the current time.</p> <pre>aMember.at("h",-5)</pre>

`.actual(date)`

The actual value for the specified date or the first actual prior to that date.

Parameter	Description
date	<p>The date must be the result of another time function.</p> <p>See Also: For more information on time functions, refer to the topic, Date Time Functions.</p>
Example(s):	<pre>aMember.actual(DateValue("07-24-00 10:30"))</pre> <p>Where "aMember" is the name of a particular member with history defined.</p>

`.actual(interval_string, intervalnumber)`

The actual value for the specified interval before the current time.

Parameter	Description
interval_string	A unit of time. Common parameters include: "year"; "month"; "week"; "hour"; "y" (which is the "day of the year" and not the year).
intervalnumber	Do <u>either</u> of the following: <ul style="list-style-type: none"> Enter a negative time interval that is preceded with a minus sign (-) to get the historical value. This is the negative offset. OR <ul style="list-style-type: none"> Enter a positive time interval to get future data.

About Client-side Historical Access Methods

The syntax for invoking a client-side Historical Access method is:

```
<primitive_data_member>.<method_name>(<argument(s)>)
```



The primitive data member must be referenced by a relative XHQ path.

For client-side Historical Access methods, a *throttling* technique is used in order to prevent excessive execution of access methods and protect the Solution Server's performance. The value for the throttle timer can be specified in the **globalsettings.properties** file of the XHQ repository. The property name is **AggregatesThrottleTimer** and its value is specified in seconds. The default value for the throttle timer is 300 seconds (or 5 minutes).



For more information on the `AggregatesThrottleTimer` property, go to the topic, [Additional Global Settings](#), located in the XHQ Administrator's Guide.

About Server-side Historical Access Methods

Server-side Historical Access methods are invoked with the **same syntax** as client-side aggregates, except that the subject of the method invocation can also be referenced by an absolute path or an alias.

The syntax for invoking a server-side Historical Access method is:

```
<primitive_data_member>.<method_name>(<argument(s)>)
```



Unlike the client-side, server-side Historical Access methods do not use the throttling technique. Instead, they use the **expression scheduler**.

For more information, go to the topic, [The Server-side Expression Scheduler](#), located in the XHQ Developer's Guide.

Historical Aggregate Methods

For any historized member, Historical Aggregate methods take two date/time parameters and aggregate the historical data within the time period defined by the parameters. A **single value** is returned, not a trend of aggregated value. So, for example, these methods will not return the hourly averages for an entire week. (Rather, such a feature is provided by the XHQ Trend Viewer and XHQ Trend Table applications.)

XHQ supports the following historical aggregates.



The parameters, t1 and t2, are the start and end times (respectively) of the request interval.

Examples:

```
format(dateAdd("hour",-2)) time is 2 hours before Now()
```

```
format(dateAdd("hour",2)) time is 2 hours after Now()
```

Quick Reference – Historical Aggregate Methods

Method	Description	Example
<code>.min(t1, t2)</code>	A minimum value.	<code>value.min(DateAdd("hour", -24), Now())</code> This returns minimum value for the previous 24 hours.
<code>.max(t1, t2)</code>	A maximum value.	<code>value.max(DateValue("01-01-2002"), Now())</code> This returns maximum value from the first day of 2002 through the current time.
<code>.avg(t1, t2)</code>	A time-weighted average value.	<code>currentValue.avg(DateAdd("hour", -12), Now())</code> This returns the average for the previous 12 hours.
<code>.integral(t1, t2)</code>	An integral value.	<code>pressure.integral(DateAdd("minute", -15), Now())</code> This returns the integral for the previous 15 minutes.
<code>.count(t1, t2)</code>	A total value.	<code>value.count(DateValue("05-01-2002"), DateValue("05-07-2002"))</code> This returns the count for the first week of May 2002.
<code>.sum(t1, t2)</code>	A sum value.	<code>value.sum(DateAdd("day", -7), Now())</code> This returns the sum of the values for the previous week.
<code>.first(t1, t2)</code>	A first value.	<code>value.first(DateAdd("day", -30), Now())</code> This returns the first value in the previous 30 day period.
<code>.last(t1, t2)</code>	A last value.	<code>value.last(DateAdd("day", -14), DateAdd("day", -7))</code> This returns the last value for the end of the previous week.
<code>.delta(t1, t2)</code>	A delta value.	<code>value.delta(DateAdd("day", -7), Now())</code> This returns the difference between the first and last values for the previous week.

About the .avg and .integral methods

The **time-weighted average** factors in how long data has been at a given value and calculates the following equation:

$$\frac{\sum_{i=1}^n (x_i * d_i)}{\sum_{i=1}^n d_i}$$

The **integral** method returns the area under the trend line for a particular period. This method is useful in calculating the total flow from a flow rate.



What happens when bad values occur?

Bad values will cause the quality to UNCERTAIN or BAD, depending on the duration of the bad intervals.

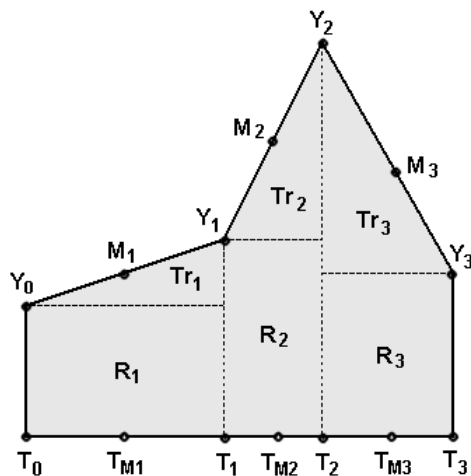
For **stepwise interpolation**, the quality of interval between points is BAD when the quality of the first point is BAD.

For **linear interpolation**, the quality of interval between points is BAD when the quality of the first point is BAD and/or the quality of the second point is BAD.

For example, the quality of the first point is GOOD and the quality of the second point is BAD. In this case, because the quality of the first point is GOOD, for **stepwise** interpolation, the quality of interval between points is GOOD. However, because the quality of the second point is BAD, the interval quality for **linear** interpolation is BAD.

Historical Aggregate Calculations

Consider the following illustration.



Graph Example – Historical Aggregate

The points T_0 through T_3 represent the times associated with values Y_0 through Y_3 . The calculations will use actual stored values in the historian, except for cases where the endpoints of the time range do not have actual values. Therefore, in this case, the values will be derived based on linear interpolation.

The **average** is the sum of the values Y_0 through Y_3 , divided by the number of values. In this case, it is four (4).

Equation:
$$\text{average} = (Y_0 + Y_1 + Y_2 + Y_3) / 4$$

The **time-weighted average** (avg) is calculated by multiplying the midpoints (M_1 through M_3) of each segment by the time of each segment (T_{M1} through T_{M3}) and dividing the total by the total time (T_3).

Equation:
$$\text{avg} = (M_1 T_{M1} + M_2 T_{M2} + M_3 T_{M3}) / T_3$$

The **integral** calculates the area under the polyline drawn between the values. In this case, the integral is the combined area of rectangles **R₁** through **R₃**, and triangles **Tr₁** through **Tr₃**.

Equation:

$$\text{integral} = \text{Area } R_1 + \text{Area } R_2 + \text{Area } R_3 + \text{Area } Tr_1 + \text{Area } Tr_2 + \text{Area } Tr_3$$

The integral assumes the value is in units per **minute**. For example, if a member represents liters per minute, calculating the integral over the last hour will correctly calculate the total flow for the hour. If, however, the measurement is in liters per *hour*, the integral result should be divided by 60 (minutes/hour) to get the correct flow for the hour.

About Client-side Historical Aggregate Methods

The syntax for invoking a client-side Historical Aggregation method is the same as the Historical Access method syntax:

```
<primitive_data_member>.<method_name>(<argument(s)>)
```



The arguments to the Historical Aggregation methods must always be the bounding times of the aggregation interval.

Like the client-side Historical Access methods, a throttling technique is used in order to prevent excessive execution of aggregation methods and protect the XHQ Solution Server's performance. The value for the throttle timer can be specified in the **globalsettings.properties** file of the XHQ repository. The property name is **AggregatesThrottleTimer** and its value is specified in seconds. The default value for the throttle timer is 300 seconds (or 5 minutes).



For more information on the [AggregatesThrottleTimer](#) property, go to the topic, [Additional Global Settings](#), located in the XHQ Administrator's Guide.

About Server-side Historical Aggregate Methods

Server-side Historical Aggregate methods are invoked with the **same syntax** as client-side aggregates, except that the subject of the method invocation can also be referenced by an absolute path or an alias.

The syntax for invoking a server-side Historical Aggregate method is:

```
<primitive_data_member>.<method_name>(<argument(s)>)
```



Unlike the client-side, server-side Historical Aggregate methods do not use the throttling technique. Instead, they use the **expression scheduler**.

For more information, go to the topic, [The Server-side Expression Scheduler](#), located in the XHQ Developer's Guide.

EXAMPLE: USING SERVER-SIDE HISTORICAL AGGREGATE METHOD IN THE SOLUTION BUILDER

In this example, the following server-side Historical Aggregate call is entered into the **Expression tab** of the XHQ Solution Builder:

The screenshot shows the XHQ Solution Builder interface with the 'Expression' tab selected. The 'Expression' field contains the text: `HighLimit.avg(DateAdd("hour", -12), Now())`. Below this, the 'Use Scheduler' checkbox is checked. The 'Schedule base date/time' field shows '06/05/2007 13:00:00' with an example '(ex: 12/25/2007 23:00:00)'. The 'Schedule period (ms)' field shows '3600001'. At the bottom, there is a table titled 'SIMPoint Inventory Items'.

Name	Type	Description	Value
CurrentValue	Real	Current value of measurement	122.05119
Description	String	Description of measurement	Tide Level5
DoubleValue	Double		43.918704986572266
HelpURL	String	URL pointing to connector h...	http://grinds.corp.indxhq.co...

XHQ Solution Builder: Expression Tab with Server-side Historical Aggregate Example

Example: `HighLimit.avg(DateAdd("hour", -12), Now())`

The left argument of a server-side Historical Aggregate call is always a primitive member.

Aliases are not supported. However, aliases can be used in conjunction with Historical Aggregates in server-side expressions like this:

```
i:alias + HighLimit.avg(DateAdd("hour", -12), Now())
```

About Connection Support

The methods discussed in this section can be used with **any** connector that supports time-series requests. Native connector aggregates are returned when they are available.

Most commercial historians support both Historical Access and Aggregation methods. However, relational databases - when used as simple historians - do not natively support any access (or historical aggregation) method. In this case, you can use a connector, like the JDBC connector, to retrieve a small (in effect RAW) trend and from which XHQ collect **at** and **actual** values.

In the case of a PHD connector, for example, it supports **first**, **last**, **min**, **max**, **avg**, and **delta** methods. Therefore, all of the supported aggregate requests will be supplied by PHD. All non-native aggregates, however, will be computed by XHQ from the RAW time-series data.

In general, the answering of history access requests is delegated, when possible, to the supported historians. This means that the results of these requests might not always be consistent if multiple historians are involved. If one historian calculates interpolated values differently from another, it is possible that the two will give different results for an **at ()** request, even given the same raw data.

Also, if a connector does not support the retrieval of data for a given type, then it does not support any of the aggregation methods for that type. Using PHD as an example again, it is noted that PHD does not support either date/time or time interval as native data types. Therefore, no historical access or aggregation of data of either of those types is supported by the PHD connector.

The following table shows the details of which methods are directly supported by specific connectors and which are computed by XHQ based on raw historical data. A check mark indicates that the back-end data source will compute the aggregate.

Connector Support

	JDBC*	PI	PHD	OPC	IP.21	SITH
at			✓		✓	
actual			✓		✓	
min		✓	✓	**	✓	✓
max		✓	✓	**	✓	✓
avg		✓	✓	**	✓	✓
integral				**		
count				**	✓	✓
sum				**	✓	✓
first				**	✓	
last				**	✓	
delta				**		

*The JDBC column represents the Oracle, ODBC, and SQL Server connectors and only when used in Time-series/VTQ queries.

**The aggregate will be computed by XHQ unless the backend data source that OPC is configured against supports that aggregate.

About Data Types Support

For some data types, some of the Historical Access and Aggregation methods are not supported. The following table gives a full breakdown of what is supported. The first column header indicates the access or aggregation method; the first row, the data type. The cells where they intersect gives the type of the return values of the method invocations. If a cell is empty, then the access or aggregation method for the given data type is not supported.

Supported Data Types

	Boolean	Integer	Long	Real	Double	Decimal	DateTime	Interval	String	Text
at	*	Integer	Long	Real	Double	Decimal	*	*	*	*
actual	Boolean	Integer	Long	Real	Double	Decimal	DateTime	Interval	String	Text
min		Integer	Long	Real	Double	Decimal	DateTime	Interval		
max		Integer	Long	Real	Double	Decimal	DateTime	Interval		
avg	Real **	Real	Double	Real	Double	Decimal		Interval		
integral		Real	Double	Real	Double	Decimal		Interval		
count	Integer	Integer	Integer	Integer	Integer	Integer	Integer	Integer	Integer	Integer
sum		Real	Double	Real	Double	Decimal		Interval		
first	Boolean	Integer	Long	Real	Double	Decimal	DateTime	Interval	String	Text
last	Boolean	Integer	Long	Real	Double	Decimal	DateTime	Interval	String	Text
delta		Integer	Long	Real	Double	Decimal	Interval	Interval		

*Should function like actual for these types.

**Should return a real value between 0.0 and 1.0.

5 | Functions

Section Contents

- [Math Functions](#)
- [Trinary Operator Function](#)
- [String Functions](#)
- [System Function](#)
- [Client System Functions](#)
- [VTQ Functions](#)
- [Metadata Functions](#)
- [Type Conversion Functions](#)
- [Date Time Functions](#)
- [Units of Measure Functions](#)
- [Late-Binding Reference Functions](#)
- [Late-Binding Subscription Functions](#)

Miscellaneous Functions:

- [Historical Mode Function](#)
- [Log Functions](#)
- [Cache Collections and NVARCHAR2](#)
- [Action Link Functions](#)
- [Java Applet Scripting Functions](#)

Math Functions

XHQ supports the following math functions for use within expressions. Math function arguments, which are signified by **(a)** or **(a,b)**, are of some numeric type.

Quick Reference - Math Functions

Function	Description	This Argument Type:	Returns This Value Type:
<code>abs(a)</code>	Returns the absolute value of a value.	Integer	Integer
		Real	Real
		Long	Long
		Double	Double
		Decimal	Decimal
<code>acos(a)</code>	Returns the arc cosine of an angle, in the range of 0.0 through π (π).	32-bit Numeric	Real
		All other Numeric types	Double
<code>asin(a)</code>	Returns the arc sine of an angle, in the range of $-\pi/2$ through $\pi/2$.	32-bit Numeric	Real
		All other Numeric types	Double
<code>atan(a)</code>	Returns the arc tangent of an angle, in the range of $-\pi/2$ through $\pi/2$.	32-bit Numeric	Real
		All other Numeric types	Double
<code>atan2(a,b)</code>	Converts rectangular coordinates (b, a) to polar (r, theta). Theta is returned.	a is 32-bit Numeric and b is 32-bit Numeric	Real
		All other Numeric types	Double
<code>ceil(a)</code>	Returns the smallest (closest to negative infinity) value that is not less than the argument and is equal to a mathematical integer.	32-bit Numeric	Real
		All other Numeric types	Double
<code>cos(a)</code>	Returns the trigonometric cosine of an angle in radians.	32-bit Numeric	Real
		All other Numeric types	Double
<code>exp(a)</code>	Returns the exponential number e (i.e., 2.718...) raised to the power of the value.	32-bit Numeric	Real
		All other Numeric types	Double
<code>floor(a)</code>	Returns the largest (closest to positive infinity) value that is not greater than the argument and is equal to a mathematical integer.	32-bit Numeric	Real
		All other Numeric types	Double
<code>log(a)</code>	Returns the natural logarithm (base e) of a value.	32-bit Numeric	Real
		All other Numeric types	Double
<code>log10(a)</code>	Returns the base-10 logarithm (base e) of a value.	32-bit Numeric	Real
		All other Numeric types	Double

Function	Description	This Argument Type:	Returns This Value Type:
<code>max(a,b)</code>	Returns the greater of two values.	a is Decimal and b is Numeric	Decimal
		a is Numeric and b is Decimal	Decimal
		a is Double and b is Simple Numeric	Double
		a is Simple Numeric and b is Double	Double
		a is Real and b is Non-floating, Real	Real
		a is Non-floating, Real and b is Real	Real
		a is Long and b is Non-floating	Long
		a is Non-floating and b is Long	Long
		a is Integer and b is Integer	Integer
<code>min(a,b)</code>	Returns the smaller of two values.	a is Decimal and b is Numeric	Decimal
		a is Numeric and b is Decimal	Decimal
		a is Double and b is Simple Numeric	Double
		a is Simple Numeric and b is Double	Double
		a is Real and b is Non-floating, Real	Real
		a is Non-floating, Real and b is Real	Real
		a is Long and b is Non-floating	Long
		a is Non-floating and b is Long	Long
		a is Integer and b is Integer	Integer

Function	Description	This Argument Type:	Returns This Value Type:
<code>pow(a,b)</code>	Returns the value of the first argument raised to the power of the second argument.	a is 32-bit Numeric and b is 32-bit Numeric	Real
		a is 64-bit Numeric and b is 64-bit Numeric	Double
		a is Decimal and b is Decimal	Decimal
<code>random()</code>	Returns a random Real greater than or equal to 0.0 and less than 1.0. Note: Same as <code>random_Real()</code> .	<no argument>	Real
<code>random_Real()</code>	Returns a random Real greater than or equal to 0.0 and less than 1.0. Note: Same as <code>random()</code> .	<no argument>	Real
<code>random_Double()</code>	Returns a random Double greater than or equal to 0.0 and less than 1.0.	<no argument>	Double
<code>round(a)</code>	Returns the closest integer to the argument.	32-bit Numeric	Integer
		All other Numeric types	Long
<code>sin(a)</code>	Returns the trigonometric sine of an angle in radians.	32-bit Numeric	Real
		All other Numeric types	Double
<code>sqrt(a)</code>	Returns the square root of a value.	32-bit Numeric	Real
		All other Numeric types	Double
<code>tan(a)</code>	Returns the trigonometric tangent of an angle in radians.	32-bit Numeric	Real
		All other Numeric types	Double
<code>trunc(a)</code>	Converts a Real to an Integer through truncation.	32-bit Numeric	Integer
		All other Numeric types	Long
<code>toDegrees(angrad)</code>	Converts an angle measured in radians to the equivalent angle measured in degrees.	32-bit Numeric	Real
		All other Numeric types	Double
<code>toRadians(angdeg)</code>	Converts an angle measured in degrees to the equivalent angle measured in radians.	32-bit Numeric	Real
		All other Numeric types	Double

abs(a)

Returns the absolute value of a value.

Argument Type(s)	Return Value
Integer	Integer
Real	Real
Long	Long
Double	Double
Decimal	Decimal

Example(s): `abs(14)` returns 14
 `abs(-7.5)` returns 7.5
 `abs(0)` returns 0

acos(a)

Returns the arc cosine of an angle, in the range of 0.0 through π (π).

Argument Type(s)	Return Value
32-bit Numeric	Real
All other types	Double

Example(s): `acos(1)` returns 0
 `acos(-1)` returns 3.141592653589793

asin(a)

Returns the arc sine of an angle, in the range of $-\pi/2$ through $\pi/2$.

Argument Type(s)	Return Value
32-bit Numeric	Real
All other types	Double

Example(s): `asin(1)` returns 90
 `asin(-1)` returns -1.5707963267948966

atan(a)

Returns the arc tangent of an angle, in the range of $-\pi/2$ through $\pi/2$.

Argument Type(s)	Return Value
32-bit Numeric	Real
All other types	Double

Example(s): `atan(1)` returns 45
 `atan(0)` returns 0.0

atan2(a, b)

Converts rectangular coordinates (b, a) to polar (r, theta). Theta is returned.

Argument Type(s)		Return Value
32-bit Numeric,	32-bit Numeric	Real
All other types		Double

Remark(s): A **positive** result represents a **counterclockwise** angle from the x-axis.

Example(s): A **negative** result represents a **clockwise** angle from the x-axis.
 atan2(1, 1) returns 0.7853981633974483
 atan2(1, -1) returns -0.7853981633974483
 atan2(0, 20) returns 1.5707963267948966

ceil(a)

Returns the smallest (closest to negative infinity) values that is not less than the argument and is equal to a mathematical integer.

Argument Type(s)	Return Value
32-bit Numeric	Real
All other types	Double

Example(s): ceil(-23.5676) returns -23.0
 ceil(23.5676) returns 24.0
 ceil(23.0676) returns 24.0

cos(a)

Returns the trigonometric cosine of an angle in radians.

Argument Type(s)	Return Value
32-bit Numeric	Real
All other types	Double

Example(s): cos(0) returns 1.0
 cos(ToRadians(180)) returns -1.0

exp(a)

Returns the exponential number e (such as 2.178 . . .) raised to the power of the value.

Argument Type(s)	Return Value
32-bit Numeric	Real
All other types	Double
<i>Example(s):</i>	exp(0) returns 1.0 exp(1) returns 2.7182818284590455 exp(-1) returns 0.36787944117144233

floor(a)

Returns the largest (closest to positive infinity) value that is not greater than the argument and is equal to a mathematical integer.

Argument Type(s)	Return Value
32-bit Numeric	Real
All other types	Double
<i>Example(s):</i>	floor(23.0676) returns 23.0 floor(-23.5676) returns -24.0 floor(23.5676) returns 23.0

log(a)

Returns the natural logarithm (**base e**) of a value. Also known as the **natural** logarithm.

Argument Type(s)	Return Value
32-bit Numeric	Real
All other types	Double
<i>Example(s):</i>	log(0.36787944) returns -0.999999975128387 (which is approximately -1.0)

log10(a)

Returns the base-10 logarithm of a value. Also known as the **common** logarithm.

Argument Type(s)	Return Value
32-bit Numeric	Real
All other types	Double
<i>Example(s):</i>	log10(10000) returns 4

max(a, b)

Returns the **greater** of two values.

Argument Type(s)		Return Value
Decimal,	Numeric	Decimal
Numeric,	Decimal	Decimal
Double,	Simple Numeric	Double
Simple Numeric,	Double	Double
Real,	Non-Floating, Real	Real
Non-Floating Real,	Real	Real
Long,	Non-Floating	Long
Non-Floating,	Long	Long
Integer,	Integer	Integer

Example(s): max(2.0, -4.67) returns 2.0

min(a, b)

Returns the **smaller** of two values.

Argument Type(s)		Return Value
Decimal,	Numeric	Decimal
Numeric,	Decimal	Decimal
Double,	Simple Numeric	Double
Simple Numeric,	Double	Double
Real,	Non-Floating, Real	Real
Non-Floating Real,	Real	Real
Long,	Non-Floating	Long
Non-Floating,	Long	Long
Integer,	Integer	Integer

Example(s): min(2.0, -4.67) returns -4.67

pow(a, b)

Returns the value of the first argument raised to the power of the second argument.

Argument Type(s)	Return Value
32-bit Numeric	Real
64-bit Numeric	Double
Decimal	Decimal

Example(s):

pow(2, 5) returns 32.0
 pow(-2, 5) returns -32.0
 pow(-2, 6) returns 64.0

random()

Returns a random Real greater than or equal to 0.0 and less than 1.0.



This function is the same as `random_Real()`.

Argument Type(s)	Return Value
<no argument>	Real

Example(s):

random() returns some number x, where $0.0 \leq x < 1.0$

random_Double()

Returns a random Double greater than or equal to 0.0 and less than 1.0.

Argument Type(s)	Return Value
<no argument>	Double

Example(s):

random_Real()

Returns a random Real greater than or equal to 0.0 and less than 1.0.



This function is the same as `random()`.

Argument Type(s)	Return Value
<no argument>	Real

Example(s):

round(a)

Returns the closest integer to the argument.

Argument Type(s)	Return Value
32-bit Numeric	Integer
All other types	Long

Example(s):

- round(-23.5676) returns -24
- round(23.5676) returns 24
- round(23.0676) returns 23
- round(23.0) returns 23

sin(a)

Returns the trigonometric sine of an angle in radians.

Argument Type(s)	Return Value
32-bit Numeric	Real
All other types	Double

Example(s):

- sin(1.5707) returns 0.9999999953653317 (which is approximately 1.0)
- sin(toRadians(90)) returns 1.0

sqrt(a)

Returns the square root of a value.

Argument Type(s)	Return Value
32-bit Numeric	Real
All other types	Double

Example(s):

- sqrt(25) returns 5.0
- sqrt(-25) returns a bad value

tan(a)

Returns the trigonometric tangent of an angle in radians.

Argument Type(s)	Return Value
real	Any real number.

Example(s):

- tan(0) returns 0.0

toDegrees(angrad)

Converts an angle measured in radians to the equivalent angle measured in degrees.

Argument Type(s)	Return Value
32-bit Numeric	Real
All other types	Double

Example(s):

```

toDegrees(1) returns 57.29577951308232
toDegrees(-3.14) returns -179.90875368164495
(which is approximately -180.0)
toDegrees(Acos(-1)) returns 180.0
toDegrees(Asin(-1)) returns -90.0
toDegrees(Atan(-1)) returns -45.0
toDegrees(Atan2(0, 20)) returns 90.0
  
```

toRadians(angdeg)

Converts an angle measured in degrees to the equivalent angle measured in radians.

Argument Type(s)	Return Value
32-bit Numeric	Real
All other types	Double

Example(s):

```

toRadians(180) returns 3.141592653589793
toRadians(-180) returns -3.141592653589793
toRadians(Sin(1.5707)) returns 0.017453292439053074
  
```

trunc(a)

Converts a Real to an Integer through truncation.

Argument Type(s)	Return Value
32-bit Numeric	Integer
All other types	Long

Example(s):

Trinary Operator Function

iff

Both **exp1** and **exp2** are always evaluated, regardless of the result of the evaluation of **bool**. In addition, the order of evaluation is not guaranteed.

Syntax: `iff(bool, exp1, exp2)`

Consider the following example.

Example: `iff(FALSE, setProperty("prop", "A"), setProperty("prop", "B"))`

In this expression, both `setProperty` calls are evaluated in no certain order. Therefore, the value for "prop" may be either A or B. The `iff()` function returns TRUE if the second `setProperty` call succeeds.

The type of the return value matches the type of whichever expression is returned.

Argument Type(s)			Return Value
bool is Boolean;	exp1 is any type;	exp2 is any type.	Returns exp1 if bool is true; returns exp2 if bool is false. The type of the return value matches the type of whichever expression is returned.

EXAMPLE: USING A NESTED IFF STATEMENT IN THE EXPRESSION TAB

Here is an example of the Nested `iff` statement. Given the following variable declarations:

- x changing number of the curve
- c1 calculation executed if x=1, for example `c1=2*1`
- c2 calculation executed if x=2, for example `c2=2*2`
- c3 calculation executed if x=3, for example `c3=2*3`
- c4 calculation executed if x=4, for example `c4=2*4`

Then, in the Expression Tab of another variable called test,

```
iff(x==1, c1, iff(x==2, c2, iff(x==3, c3, iff(x==4, c4, 0))))
```

Therefore, if the value of x=1, then it is true that it will perform the calculation in c1. Else, it will test to see if x=2, and so on. If it continues through the whole statement and it still does not find a value, it is given a value of 0.

Quality and iff Expressions

The quality of the Boolean argument is always taken into consideration by `iff()`. Consider this example:

```
iff(setQuality(TRUE, UNCERTAIN), exp1, exp2)
```

The result of this expression will evaluate to the value of `exp1` with UNCERTAIN quality.

String Functions

XHQ supports the following string functions for use within expressions. A string function returns either a **String** or an **Integer** value.



The '+' sign becomes a concatenation operator for string and text values.

Quick Reference - String Functions

Function	Description	Returns This Value Type:
<code>compareTo</code> (String a, String b)	Compares two strings.	Integer
<code>compareToIgnoreCase</code> (String a, String b)	Compares two strings, ignoring the case.	Integer
<code>indexOf</code> (String a, String b, Integer nth)	Returns index of the nth occurrence of pattern b in string a.	Integer
<code>indexOf</code> (String a, String b, Integer nth, Integer startIndex)	Returns index of the nth occurrence of pattern b in string a with search starting at startIndex.	Integer
<code>format</code> (DateTime dt)	Formats the specified date/time (dt) as a string using the default date and time formatting.	String
<code>format</code> (DateTime dt, String s)	Formats the specified date/time (dt) as a string using the specified formatting, s.	String
<code>format</code> (Decimal m, String fmt)	Formats a decimal value with the specified formatting.	String
<code>format</code> (Double d, String fmt)	Formats a double-precision value with the specified formatting.	String
<code>format</code> (Real r, String fmt)	Formats a real value with the specified formatting.	String
<code>format</code> (String s)	Formats the current date/time as a string using the specified formatting, s.	String
<code>format</code> (TimeInterval ti)	Formats the specified time interval (ti) as a string using default time interval formatting.	String
<code>format</code> (TimeInterval ti, String type)	Formats the specified time interval (ti) as a string using the specified format symbol name: "xhq" or "iso".	String
<code>format</code> (TimeInterval ti, String fmt, String char)	Formats the specified time interval (ti) as a string using the specified formatting pattern, fmt, and the character, char, as the token delimiter.	String
<code>indexOf</code> (String a, String b,	Returns the index within a, of the first	Integer

Function	Description	Returns This Value Type:
<code>Integer startIndex)</code>	occurrence of b, with search starting at startIndex.	
<code>lastIndexOf(String a, String b)</code>	Returns the index within a, of the last occurrence of b.	Integer
<code>lastIndexOf(String a, String b, Integer startIndex)</code>	Returns the index within a, of the last occurrence of b, with search starting at startIndex.	Integer
<code>length(String str)</code>	Returns the count of characters in str.	Integer
<code>middle(String str, Integer beginIndex, Integer endIndex)</code>	Returns a new string that is a substring of str, starting at beginIndex and ending at endIndex-1.	String
<code>replace(String s1, String s2, String s3)</code>	Returns a string that replaces occurrences of s2 found in s1 with s3.	String
<code>right(String str, Integer beginIndex)</code>	Returns a new string that is a substring of str, starting at beginIndex and ending at the end of the string.	String
<code>substring(String str, Integer beginIndex)</code>	Returns a new string that is a substring of str, starting at beginIndex and ending at the end of the string.	String
<code>substring(String str, Integer beginIndex, Integer endIndex)</code>	Returns a new string that is a substring of str, starting at beginIndex and ending at endIndex-1.	String
<code>toLowerCase(String str)</code>	Converts all the characters in str to lower case.	String
<code>toUpperCase(String str)</code>	Converts all the characters in str to upper case.	String
<code>trim(String str)</code>	Removes the white spaces before and after str.	String

Interval Formatting

The *default* formatting for intervals is the standard XHQ format:

```
[ - ] [ D ] H:M:S [ . S ]
```

The *second* form of interval formatting accepts a format-type string:

- **xhq**
The default formatting style shown above.
- **iso**
The ISO 8601 interval duration.

The *third* form takes a format string and a token character. In this form, the format string is parsed for tokens identified as starting with the token character specified in the third parameter. Tokens include those defined for use by the standard "interval string", as well as the format-type specifiers defined above.

compareTo

Compares two strings.

Syntax: `compareTo(String a, String b)`

Argument Type(s)	Return Value
String a and String b	Integer
<i>Remark(s):</i>	<p>If < 0, then String a comes before String b.</p> <p>If > 0, then String a comes after String b.</p> <p>If = 0, then String a equals String b.</p>
<i>Example(s):</i>	<p><code>compareTo("ABCD", "ABCDE")</code> returns -1</p> <p><code>compareTo("ABCD", "abcd")</code> returns -32</p> <p><code>compareTo("WXYZ", "ABCD")</code> returns 22</p> <p><code>compareTo("ABCD", "ABCD")</code> returns 0</p>

compareToIgnoreCase

Compares two strings, ignoring the case.

Syntax: `compareToIgnoreCase(String a, String b)`

Argument Type(s)	Return Value
String a and String b	Integer
<i>Example(s):</i>	<p><code>compareToIgnoreCase("ABCD", "abcd")</code> returns 0</p> <p><code>compareToIgnoreCase("ABCD", "ABCD")</code> returns 0</p> <p><code>compareToIgnoreCase("WXYZ", "ABCD")</code> returns 22</p>

findIndexOf

Returns index of the nth occurrence of pattern b in string a [with search starting at startIndex].

Syntax: `findIndexOf(String a, String b, Integer nth[, Integer startIndex])`

Property	Description
a	String in which to search for the pattern.
b	The pattern to find.
Integer nth	The number of the occurrence to find (positive means from the start, negative means from the end, and 0 is invalid).
Integer startIndex	The starting index in the string to begin search.



The comparison is case-sensitive. If the sought-for string is not found, -1 is returned.

Example(s):

```
findIndexOf( "The rain in Spain falls mainly on the plain", "ain", 2 ) returns
14
findIndexOf( "The rain in Spain falls mainly on the plain", "ain", -1 ) returns
40
findIndexOf( "The rain in Spain falls mainly on the plain", "dude!", 1 )
returns -1
findIndexOf( "The rain in Spain falls mainly on the plain", "ain", 0 ) returns -
1 [0th occurrence is undefined]
findIndexOf( "The rain in Spain falls mainly on the plain", "ain", 1, 30 )
returns 40 [the first occurrence after index 30 is the final "ain"]
findIndexOf( "The rain in Spain falls mainly on the plain", "ain", -2, 30 )
returns 14 [the second occurrence from end - defined at index 30]
```

Format Functions

format(dateTime dt)

Formats the specified date/time (dt) as a string using the default date and time formatting.

Returns: String

format(dateTime dt, string s)

Formats the specified date/time (dt) as a string using the specified formatting, s.

Returns: String

format(decimal m, string fmt)

Formats a decimal value with the specified formatting, fmt.

Returns: String

format(double d, string fmt)

Formats a double-precision value with the specified formatting, *fmt*.

Returns: String

format(real r, string fmt)

Formats a real value with the specified formatting, *fmt*.

Returns: String

format(string s)

Formats the current date/time as a string using the specified formatting, *s*.

Returns: String

format(timeInterval ti)

Formats the specified time interval (*ti*) as a string using the default time interval formatting.

Returns: String

format(timeInterval ti, string type)

Formats the specified time interval (*ti*) as a string using the specified format symbol name: "xhq" or "iso".

Returns: String

format(timeInterval ti, string fmt, string char)

Formats the specified time interval (*ti*) as a string using the specified formatting pattern, *fmt*, and the character, *char*, as the token delimiter.

Returns: String

indexOf

Returns the index within *a*, of the first occurrence of *b*, with search starting at *startIndex*.

Syntax: indexOf(String a, String b, Integer startIndex)

Property	Description
<i>a</i>	String in which to search for the pattern.

Property	Description
b	The pattern to find.
startIndex	The starting index in the string to begin search.



The return value is "zero-based", that is, if the string sought for is found at the beginning of the string, zero is returned. Also, the comparison is case-sensitive.

If the sought-for string is not found, -1 is returned.

Example(s): `indexOf("The quick brown fox jumped", "e")` returns the integer 2
 `indexOf("The quick brown fox jumped", "E")` returns the integer -1
 `indexOf("The quick brown fox jumped", "Z")` returns the integer -1

lastIndexOf

Returns the index within a, of the last occurrence of b [with search starting at startIndex].

Syntax: `lastIndexOf(String a, String b[, Integer startIndex])`

Property	Description
a	String in which to search for the pattern.
b	The pattern to find.
startIndex	The starting index in the string to begin search.



The return value is "zero-based", that is, if the string sought for is found at the beginning of the string, zero is returned. Also, the comparison is case-sensitive.

If the sought-for string is not found, -1 is returned.

length

Returns the count of characters in str.

Syntax: `length(String str)`

Argument Type(s)	Return Value
String str	Integer

Example(s): `length("ABC")` returns 3

middle

Returns a new string that is a substring of str, starting at beginIndex and ending at endIndex-1.

Syntax: `middle(String str, Integer beginIndex, Integer endIndex)`

Property	Description
str	String in which to search for the pattern.
beginIndex	The starting index in the string to begin search.
endIndex	The ending index.

replace

Replaces occurrences of s2 found in s1 with s3.

Syntax: `replace(String s1, String s2, String s3)`



String values are case-sensitive.

Argument Type(s)	Return Value
String s1, String s2, String s3	String

Example(s):

```
replace("The quick brown fox jumped","E","E")
Returns the string: The quick brown fox jumped
replace("The quick brown fox jumped","e","E")
Returns the string: ThE quick brown fox jumpEd
replace("The quick brown fox jumped"," ","")
Returns the string: Thequickbrownfoxjumped
```

right

Returns a new string that is a substring of str, starting at beginIndex and ending at the end of the string.

Syntax: `right(String str, Integer beginIndex)`

Property	Description
str	String in which to search for the pattern.
beginIndex	The starting index in the string to begin search.

substring

Returns a new string that is a substring of str, starting at beginIndex [and ending at endIndex-1].

Syntax: `substring(String str, Integer beginIndex [, Integer endIndex])`



Brackets [] indicate that the argument is optional. If the argument **Integer endIndex** is not defined, then the last character is the **endIndex** default value.

Property	Description
<code>str</code>	Any string value.
<code>beginIndex</code>	A zero-based integer index for the first character of the substring to be returned. If returning the first character, this value should be zero.
<code>endIndex</code>	A zero-based integer index for the last character of the substring to be returned.
<i>Remarks(s):</i>	If <code>endIndex</code> is less than <code>beginIndex</code> or greater than its length, then the entire <code>str</code> is returned.
<i>Example(s):</i>	<pre>substring("ABCDEFGH", 2, 4)</pre> <p>Returns the string: "CD"</p> <pre>substring("The quick brown fox jumped",IndexOf("The quick brown fox jumped","fox"),Length("The quick brown fox jumped"))</pre> <p>Returns the string: "fox jumped"</p> <pre>subString("The quick brown fox jumped",IndexOf("The quick brown fox jumped","fox"),0)</pre> <p>Returns the string: "The quick brown fox jumped"</p> <pre>subString("The quick brown fox jumped",0,IndexOf("The quick brown fox jumped","fox"))</pre> <p>Returns the string: "The quick brown"</p>

toLowerCase

Converts all the characters in `str` to lower case.

Syntax: `toLowerCase(String str)`

Argument Type(s)	Return Value
String <code>str</code>	String
<i>Example(s):</i>	<pre>toLowerCase("AbCdEfG")</pre> <p>Returns the string: "abcdefg"</p>

toUpperCase

Converts all the characters in `str` to upper case.

Syntax: `toUpperCase(String str)`

Argument Type(s)	Return Value
String <code>str</code>	String
<i>Example(s):</i>	<pre>toUpperCase("AbCdEfG")</pre> <p>Returns the string: "ABCDEFGH"</p>

trim

Removes the white spaces before and after str.

Syntax: `trim(String str)`

Argument Type(s)	Return Value
String str	String

Remark(s): Only leading and trailing spaces are removed; spaces embedded within the string are preserved.

Example(s): `trim(" AB C D ")`
Returns the string: "AB C D"

System Function

XHQ supports the following system function for use within expressions.

Quick Reference - System Function

Function	Description	Returns This Value Type:
getSystemProperty (String property)	<p>Returns the system property specified.</p> <p>Property tag values (which must be in double quotes) are as follows:</p> <ul style="list-style-type: none"> • date • description (of a component) • hostName (web server) • name (of a component instance) • solutionName • time • userName • viewName • EvalCount 	String

getSystemProperty

Returns the system property specified.

Syntax: `getSystemProperty(String property)`



Property tag values must be in double quotes.

Property	Description
date	Returns the current date.
description	<p>For the root node in the solution, the solution description is returned, as specified in the "Solutions" application.</p> <p>For all other nodes, the name of the component containing the view (for client-side) or member (for server-side expressions).</p>
hostname	<p>When viewed through XHQ Solution Builder at runtime:</p> <p>The <i>client-side</i> expression appears blank.</p> <p>The <i>server-side</i> expression displays the hostname.</p> <p>When viewed through the browser client:</p> <p>Both the <i>client-side</i> and the <i>server-side</i> expressions display the hostname of the web server.</p>

Property	Description
name	<p>For the root node in the solution, this is the solution name, as specified in the "Solutions" application.</p> <p>For all other nodes, this is the name of the component instance containing the view (for client-side) or member (for server-side expressions).</p>
solutionName	<p>This is the solution name, as specified in the "Solutions" application, and as it appears in the root of the navigation tree.</p> <p>Note: For the root node, the "name" and "solutionName" properties are the same.</p>
time	Returns the client's current system time.
userName	<p>When viewed through XHQ Solution Builder at runtime:</p> <p>The <i>client-side</i> expression displays the current user name.</p> <p>The <i>server-side</i> expression appears blank.</p> <p>When viewed through the browser client:</p> <p>The <i>client-side</i> expression displays the current user name.</p> <p>The <i>server-side</i> expression appears blank.</p>
viewName	<p>When viewed through the XHQ Solution Builder at runtime:</p> <p>The <i>client-side</i> expression displays the current view name.</p> <p>The <i>server-side</i> expression appears blank.</p> <p>When viewed through the browser client:</p> <p>The <i>client-side</i> expression displays the current view name.</p> <p>The <i>server-side</i> expression appears blank.</p>

Example(s):

```
getSystemProperty("date")
Returns the string: Wed Jul 4 2001
getSystemProperty("time")
Returns the string: 15:37:02 PDT
getSystemProperty("description")
Returns the string: The solution description.
getSystemProperty("hostname")
Returns the string: localhost
getSystemProperty("name")
Returns the string: Enterprise
getSystemProperty("solutionName")
Returns the string: Enterprise
getSystemProperty("userName")
Returns the string: JohnSmith
getSystemProperty("viewName")
Returns the string: RefineryTopView
```

Client System Functions

XHQ supports the following Client system functions for use within expressions.

Quick Reference - Client System Functions

Function	Description	Returns This Value Type:
<u>closePopUpView</u> (String cn, String vn)	Closes the pop-up view for the component member name, cn, and the view name, vn. Returns TRUE if a pop-up window is closed.	Boolean
<u>closeWindow</u> (Integer mode)	Closes the current window according to the given mode. Returns TRUE if window is closed. Modes: <ul style="list-style-type: none"> 1 - Closes the active view window only if it is a pop-up view. This is the default setting when the mode is not specified or is invalid. 2 - Closes the active window regardless of window style (for example, pop-up, top-level). 	Boolean
<u>displayedUnits</u> (String item)	Returns the units displayed for the indicated item which can be a tag name or path to a primitive member.	String
<u>getClientVariable</u> (String s)	Returns the value of the given client variable, s. Returns null if no such named client variable exists.	String
<u>getCookie</u> (String s)	Returns the value of the given cookie, s. Returns null if no such named client variable exists.	String
<u>IsInRole</u> (String roles)	Returns TRUE if the current user belongs to any XHQ role in the comma-delimited list, roles.	Boolean
<u>IsInRole</u> (String roles, String delim)	Returns TRUE if the current user belongs to any XHQ role in the list, roles, delimited by delim.	Boolean
<u>navigate</u> (Integer mode)	Navigates the solution viewer in the given mode. Modes: <ul style="list-style-type: none"> 1 = home 2 = back (to previous view) 3 = up (component level) 	
<u>setClientVariable</u> (String property, String value)	Sets property to specified value. Returns TRUE for success and FALSE for failure. This client system function has the ability to	Boolean

Function	Description	Returns This Value Type:
	change an application (client) variable, using the variable name. If the specified client variable does not exist, it is created and the value is assigned.	
setCookie (String cn, String cv)	Sets the cookie, cn, to the value, cv. Returns true upon success.	Boolean
setCookie (String cn, String cv, Real exp)	Sets the cookie, cn, to the value, cv, with expiration in days, exp. Returns true upon success.	Boolean
setCookie (String cn, String cv, Real exp, String url)	Sets the cookie, cn, to the value, cv, with expiration in days, exp, and the URL path scope, url. Returns true upon success.	Boolean
setCookie (String cn, String cv, Real exp, String url, String dmn)	Sets the cookie, cn, to the value, cv, with expiration in days, exp, URL path scope, url, and TCP domain scope, dmn. Returns true upon success.	Boolean
setViewVariableValue (string, constant/expression)	<p><i>This function is valid for combo boxes only.</i></p> <p>Returns TRUE if the function executes successfully.</p> <p>This client system function allows an expression to change the value of a view variable using the view variable's name.</p>	Boolean
showView (String path, String vname)	Shows the view, vname, for the object, path. Returns TRUE if successful.	Boolean
showView (String path, String vname, Integer mode)	<p>Shows the view, vname, for the object, path, using the given mode. Returns TRUE if successful.</p> <p>Modes:</p> <ul style="list-style-type: none"> • 0 – use view properties This mode sets the options selected from the View Properties dialog box. This is the default mode. • 1 – replace This mode causes the existing view in the view panel to be replaced. • 2 – popup This mode causes a new pop-up window to display the view. • 3 – use view properties (support for legacy expressions) This mode sets the options selected from the View Properties dialog box. • 4 – do not add to view stack Replace view mode but does not add to the view history 	Boolean

Function	Description	Returns This Value Type:
	<p>stack.</p> <ul style="list-style-type: none"> • 5 – replace current view with this view in the view stack Replace view mode used to control peer-level view history behavior. 	
showView (String path, String vname, Integer mode, Integer autoClose)	<p>Shows the view, vname, for the object, path, using the given mode. The autoClose parameter determines how the window is handled when the launching window is closed. Returns TRUE if successful.</p> <p>Modes:</p> <ul style="list-style-type: none"> • 0 – use view properties This mode sets the options selected from the View Properties dialog box. This is the default mode. • 1 – replace This mode causes the existing view in the view panel to be replaced. • 2 – popup This mode causes a new pop-up window to display the view. • 3 – use view properties (support for legacy expressions) This mode sets the options selected from the View Properties dialog box. • 4 – do not add to view stack Replace view mode but does not add to the view history stack. • 5 – replace current view with this view in the view stack Replace view mode used to control peer-level view history behavior. <p>Auto Close modes:</p> <ul style="list-style-type: none"> • 0 - No auto close • 1 - Auto close 	Boolean
showView (String path, String vname, Integer mode, Integer autoClose, Integer x, Integer y)	<p>Shows the view, vname, for the object, path, at position x and y, using the given mode. The autoClose parameter determines how the window is handled when the launching window is closed. Returns TRUE if successful.</p> <p>Modes:</p> <ul style="list-style-type: none"> • 0 – use view properties This mode sets the options selected from the View Properties dialog box. This is the default mode. • 1 – replace This mode causes the existing view in the view panel to be replaced. 	Boolean

Function	Description	Returns This Value Type:
	<ul style="list-style-type: none"> • 2 – popup This mode causes a new pop-up window to display the view. • 3 – use view properties (support for legacy expressions) This mode sets the options selected from the View Properties dialog box. • 4 – do not add to view stack Replace view mode but does not add to the view history stack. • 5 – replace current view with this view in the view stack Replace view mode used to control peer-level view history behavior. <p>Auto Close modes:</p> <ul style="list-style-type: none"> • 0 - No auto close • 1 - Auto close 	
startHistoryNavigationSlideShow (DateTime startTime, DateTime endTime, String jogIncrement, String transitionSpeed)	Launches and plays the Historical Navigation movie (or slide show) from a view. The functions returns TRUE is it executes successfully; otherwise, it is returns FALSE. <ul style="list-style-type: none"> • startTime - The time in which the Historical Viewing starts playing. • endTime - The time in which the Historical Viewing stops playing. • jogIncrement - The jog increment in s (seconds), m (minutes), h (hours), d (days), or w (weeks). • transistionSpeed - The transition speed in s (seconds). 	Boolean
startHistoryNavigationSlideShow (DateTime startTime, DateTime endTime, String jogIncrement, String transitionSpeed, String aggregateType, String aggregateInterval)	Launches and plays the Historical Navigation movie (or slide show) from a view. The functions returns TRUE is it executes successfully; otherwise, it is returns FALSE. <ul style="list-style-type: none"> • startTime - The time in which the Historical Viewing starts playing. • endTime - The time in which the Historical Viewing stops playing. • jogIncrement - The jog increment in s (seconds), m (minutes), h (hours), d (days), or w (weeks). • transistionSpeed - The transition speed in s (seconds). 	Boolean

Function	Description	Returns This Value Type:
	<ul style="list-style-type: none">• aggregateType - Displays the aggregation type. The default type is Interpolate.• aggregateInterval - Displays the aggregation interval. The default interval is 30 seconds (30s).	
subscribeToClientVariable (String)	This system function allows an expression to subscribe to an application (client) variable. Whenever the referenced client variable is changed, the expression is notified and the new value is returned. Since client variables have only string values, this function returns a string.	String

closePopupView

Closes the pop-up view for the component name, *cn*, and the view name, *vn*. Returns TRUE if a pop-up window is closed.

Syntax: `closePopupView(String cn, String vn)`

Examples: `closePopupView("Enterprise", "Overview")`

or

```
eval("var applet = getApplet(); applet.closePopupView('Enterprise',
'Overview');")
```

closeWindow

Closes the current window according to the given mode. Returns TRUE if window closed. .

Syntax: `closeWindow (Integer mode)`

Mode	Description
1	Closes the active view window only if it is a pop-up view. This is the default setting when the mode is not specified or is invalid.
2	Closes the active window regardless of window style (for example, pop-up, top-level).

From an action link configuration, this function is added to the end of the list of functions.

Example: `eval("showView('::Top','circle',2,null,0,100)", closeWindow(1)`

displayedUnits

Returns the units displayed for the indicated item which can be a tag name or path to a primitive member.

Syntax: `displayedUnits (String item)`

getClientVariable

Returns the value of the given client variable, *s*. Returns null if no such named client variable exists.

Syntax: `getClientVariable (String s)`



This system function is not available in server-side expressions.
Property tag values must be in double quotes.



Set a [global property to enable/disable client variable quality](#).

EXAMPLE: USING THE GETCLIENTVARIABLE FUNCTION

The following is an example of how to display the value of the number of times the **eval ()** function has been invoked during the current user session. The count is being tracked using the client variable **EvalCount**.



The client variable `EvalCount` is only meaningful if it is incremented when `eval ()` is invoked. The value expression, `EvalCount`, is **case-sensitive**.

1. From the XHQ Workbench, insert a **Value item**.
2. Click the **Animation property**.
3. With the **Animation Type** set to **Value**, click **Edit**.
The "Edit Expression" dialog box appears.
4. Enter `integerValue (getClientvariable ("EvalCount"))`.
5. Click **OK**.

getCookie

Returns the value of the given cookie, `s`. Returns null if no such named client variable exists.



If the result is assigned to a control that has a non-string type, explicitly cast the string. For example, use `IntegerValue (getCookie ("AccountBalance"))`.

Also, a `getCookie` request against a non-existent cookie returns a BAD value.

Syntax: `getCookie (String s)`

Parameter	Description
<code>s</code>	The name of the cookie.

Example(s): *Given:* The cookie "color" is set to "Red" by using the function `SetCookie ("color","Red")`.
`GetCookie("color")`
 Returns the string: Red

isInRole

Returns a Boolean TRUE/FALSE value that indicates whether the current use has certain security privileges. When running on the client, `IsInRole` returns TRUE or FALSE based on whether the current user belongs to at least one role in a specified list.



This function is only available in client-side (view) contexts.

`IsInRole` returns a data quality of GOOD in all cases except if the delimiter argument is incorrect.

Syntax: `IsInRole(String roleList[, String delimiter])`

Argument	Description
<code>roleList</code>	The delimited list of roles. This argument is required.
<code>delimiter</code>	The delimiter used for the <code>roleList</code> string. If this argument is omitted, then the default comma (",") is used. If defined then the delimiter string must be one character long.

Things to Note

- Again, this function is only available in client-side contexts. It is not supported in server-side expressions and results in BAD quality if encountered there.
- `IsInRole` parses the `roleList` string based on the delimiter character, and then removes leading and trailing whitespace characters around each role.
- In order for the `IsInRole` function to return TRUE, the user must belong to at least one role specified in the `roleList`.
- An empty role (string with no content) in `roleList` is ignored and is handled as if the user does not belong to that role (a user does not belong to an empty role).
- Role name comparison is not case-sensitive.
- This function is called on the view load only. It doesn't subscribe to role updates.
- Data quality is GOOD if the `delimiter` string is: 1) omitted (default is used) or 2) defined and is one character long. Data quality will be BAD_CONFIG_ERROR if the `delimiter` string is specified and the string length is not one character long.
- Unicode is supported for both `roleList` and `delimiter`.
- There are no roles defined when running with security turned off for the repository. In this case, the function returns a value of FALSE with GOOD data quality.

Examples The current user belongs to roles "ABC" and "DEF". On the client, the following will be returned:

This function...	Returns this...
<code>IsInRole("ABC")</code>	TRUE
<code>IsInRole("DEF")</code>	TRUE
<code>IsInRole("XYZ")</code>	FALSE

This function...	Returns this...
IsInRole("abc")	TRUE
IsInRole("")	FALSE
IsInRole("ABC,DEF")	TRUE
IsInRole("ABC,")	TRUE
IsInRole("ABC,,,>>")	TRUE
IsInRole(" , , ")	FALSE
IsInRole(" ABC , XYZ ")	TRUE
IsInRole("XYZ ; DEF", ";")	TRUE
IsInRole("XYZ ; DEF", ":")	FALSE
IsInRole("XYZ,DEF", "")	data quality of BAD_CONFIG_ERROR
IsInRole("XYZ,DEF", " , ")	data quality of BAD_CONFIG_ERROR

navigate

Navigates the solution viewer to the given mode.

Syntax: `navigate(Integer mode)`

Parameter	Description
mode	<p>An integer value.</p> <ul style="list-style-type: none"> 1 – home Displays the home view configured for the solution. The home view is determined by finding a valid configuration for one of the following settings (search is in the order listed): <ul style="list-style-type: none"> The command-line value "net.indx.homeView". The HomeView value in the clientsettings.properties file. The default home view configured for the system. 2 – back Displays the last view visited. 3 – up Displays the default view for the current selected node of the parent's node. <p>Important: This mode can only be supported when a navigation tree has been created, as the feature relies on the tree for its implementation.</p>



See also, [Action Link Functions](#).

setClientVariable

This system function has the ability to change an application (client) variable, using the variable name. It sets property to specified value. Returns TRUE for success and FALSE for failure. If the specified client variable does not exist, it is created and the value is assigned.

Syntax: `setClientVariable(String property, String value)`



This system function is not available in server-side expressions.
Property tag values must be in double quotes.

EXAMPLE: USING THE SETCLIENTVARIABLE FUNCTION

The following is an example of how to increment the client variable **EvalCount** when the **eval()** function is invoked from the action expression of a button control.



The value expression, `EvalCount`, is **case-sensitive**.

1. From the XHQ Workbench, insert a **Button control**.
2. Click the **Button property**.
3. In the **Label** box, enter **Reset**.
4. In the **Expression** box, enter the following:

```
eval("reset()"), setClientVariable("EvalCount", IntegerToString(integerValue  
(getClientVariable("EvalCount")) + 1))
```

In runtime, when the button is pressed, the JavaScript function **reset()** is called and the client variable **EvalCount** is incremented by one.

setCookie

Saves a user's last input for the controls. For example, if you place this function in a button control, the function is executed each time the button control is pressed. This is especially useful for buttons used to trigger a filter criteria to update.



Brackets [] indicate that the parameter is optional.

Cookies used by the XHQ Solution Builder are not shared with the browser client. Therefore, they can have different values.

Syntax: `setCookie(String cn, String cv[, Real exp[, String url[, String
dmn]]])`

Parameter	Description
cn	The name of the cookie.
cv	The value of the cookie.
exp	<i>OPTIONAL</i> The expiration in days.
url	<i>OPTIONAL</i>

Parameter	Description
	The URL path scope of the cookie.
dmn	<i>OPTIONAL</i>
	The TCP domain name scope of the cookie.

Important Things To Consider

A cookie is a text file kept on a client machine that is used for storing information. Cookies are often used to persist information between client sessions, but they may also be used to store information within a session. Cookies have expiration dates. If you set the expiration date to now, the state information in the cookie will only be good for the current session. Otherwise, the information is good until the cookie expires.

The XHQ Solution Viewer has three ways of setting a cookie on the client machine:

1. There is a **method** for setting a cookie that can be called against the applet object. This method can be invoked against the applet object in custom JavaScript code.
Example: `XHQ_Solution_Viewer.setCookie("foo", "bar")`
This custom code may even be the argument of an "eval" expression in an XHQ view.
2. There is also an **XHQ expression** for setting a cookie. This expression may be embedded anywhere in a view and be invoked at given times, for example when a button is pushed or an action link is exercised.
3. There is also a **function** implemented in the XHQ Solution Viewer's supporting JavaScript that can set a cookie. As with the method on the applet object, this function may be invoked in custom JavaScript code (or in an "eval" expression).

All of these operations go by the name "setCookie" when you invoke them. Operations 1 and 2 use the same code to set the cookie. The code is implemented in Java and included in the applet. For this reason, operations 1 and 2 are referred to as the "Java setCookie method". On the other hand, operation 3 is referred to as the "JavaScript setCookie function".

All three of the setCookie operations take at least a name argument and a value argument. They may also take an optional argument to specify an expiration. But the two types (method versus function) differ slightly in how you specify expiration information.

- The **Java setCookie method** takes an argument that specifies the *lifespan* of the cookie in days from the moment the method is invoked. That is to say, within the setCookie method, the lifespan is added to the current time and the result is set as the cookie's expiration time.

The default lifespan can be configured to any number of days. As usual, the configuration is made in the `repos\conf\web\appletsettings.js` file (which is also known as the `/indx/custom/appletsettings.js` file). You configure it by setting the variable named `defaultCookieExpDays` to a non-negative number. If a non-numerical or negative value is specified (or if no value at all is specified), the old default of five days is used.

If you want to set a cookie that will only be valid for the current session, invoke `setCookie(name, value, 0)`.

- The **JavaScript setCookie function** takes an argument that specifies a JavaScript Date object that directly represents the cookie's expiration time.

The JavaScript function now behaves more like the Java setCookie method. The optional expiration argument is still expected to be a JavaScript date. But if no date is specified, a date object representing the current time plus the default lifespan is used. The default lifespan is also specified by the `defaultCookieExpDays` variable.

If you want to set a cookie that will only be valid for the current session, invoke `setCookie(name, value, new Date())`.

setViewVariableValue

This system function allows an expression to change the value of a view variable using the view variable's name. It returns TRUE if the function executes successfully, FALSE otherwise.

Syntax: `setViewVariableValue(string, constant/expression)`



Currently, only combo boxes and edit boxes are allowed to have their values changed.

Argument	Description
string	Specifies the name of the view variable without using the \$ (dollar sign) prefix.
constant or expression	If the type of the constant or the expression does not match that of the view variable, then an attempt will be made at runtime to convert the value to the appropriate type. If the type conversion fails, the function will fail and will return a FALSE.

EXAMPLE: USING THE SETVIEWVARIABLEVALUE FUNCTION

In the following example, three edit boxes are used to create an ad hoc filter on a collection. A button is then placed in the view that allows the user to reset all three edit boxes at once.

The ad hoc filter has three sets of criteria, with each set dependent on one of the edit boxes.

The button then resets all three combo boxes. Select the button control and click the Button tab.

Enter these values:

Label: ResetFilter

Expression: `setViewVariableValue("EditBox1","%"),setViewVariableValue("EditBox2",0),
setViewVariableValue("EditBox3",0.0)`

showView

Shows the view, vname, for the object, path, at position x and y, using the given mode. The autoClose parameter determines how the window is handled when the launching window is closed. Returns TRUE if successful.



Brackets [] indicate that the parameter is optional. Each parameter is enclosed by double quotation marks (as opposed to the JavaScript function showView parameters, which are enclosed by single quotes).

Syntax: `showView(String path, String vname[, Integer mode[, Integer autoClose[, Integer x, Integer y]])`

Valid:

```
showView("::MySolution.comp1", "defaultView")
showView("::MySolution.comp2", "testView", 2, 0)
showView("::MySolution.comp2", "testView", 2, 1, 300, 300)
showView(stringVar1, stringVar2)
```

Invalid: `showView("::MySolution.comp2", "testView", 2, 0, 0)`

Parameter	Description
path	A string value. The fully qualified path of the object to view. The path must begin with two colon characters (::) and each member in the path must be separated by a period (.).

Parameter	Description
	<i>Example:</i> ShowView("::Acme.WesternDivision.Sales","Forecasts")
vname	<p>A string value.</p> <p>The name of the view of the object.</p> <p><i>Example:</i> ShowView("::Acme.WesternDivision.Sales","Forecasts")</p>
mode	<p>An integer value.</p> <ul style="list-style-type: none"> • 0 – use view properties This mode sets the options selected from the View Properties dialog box. This is the default mode. • 1 – replace This mode causes the existing view in the view panel to be replaced. • 2 – popup This mode causes a new pop-up window to display the view. • 3 – use view properties (support for legacy expressions) This mode sets the options selected from the View Properties dialog box. • 4 – do not add to view stack Replace view mode but does not add to the view history stack. • 5 – replace current view with this view in the view stack Replace view mode used to control peer-level view history behavior. <p>Note: If a mode is not specified or if a mode is invalid, then the default mode of 0 is used.</p> <p><i>Example:</i> ShowView("::Acme.WesternDivision.Sales","Forecasts",2)</p>
autoClose	<p>An integer value. If the parent window (the window where this view will be launched from) is a pop-up, then this optional parameter indicates whether the parent pop-up window should automatically close once the action link is activated.</p> <ul style="list-style-type: none"> • 0 - No auto close • 1 - Auto close <p><i>Example:</i> ShowView("::Acme.WesternDivision.Sales","Forecasts",2,1)</p>
x,y	<p>The values you set (in pixels) define the x- and y-coordinates of the top, left corner of the pop-up window.</p> <p>Important: To use the XY parameters, you must set the mode to a value of 2-"popup" and specify an autoClose value.</p> <p>Note: If the specified mode is 0-"use view properties" and the view has a pop-up defined in the view properties, then the default pop-up position (which is at the center of the screen) is applied. These are optional parameters associated with the mode 2-"popup", and are ignored in the 1-"replace" or 0-"use view properties" modes.</p> <p>See Also: For more information on setting the x- and y-coordinates of the pop-up view window, go to the topic, <i>More About Pop-up Views</i>, located in the XHQ Developer's Guide.</p> <p><i>Example:</i> ShowView("::Acme.WesternDivision.Sales","Forecasts",2,1,300,300)</p>



See also, [Action Link Functions](#).

startHistoryNavigationSlideShow

Launches and plays the Historical Navigation movie (or slide show) from a view. The function returns TRUE if it executes successfully; otherwise, it returns FALSE.

Syntax: `startHistoryNavigationSlideShow(DateTime startTime, DateTime endTime, String jogIncrement, String transitionSpeed[, String aggregateType, String aggregateInterval])`



Property tag values must be in double quotes.

Property	Description
startTime	The time in which the Historical Viewing starts playing.
endTime	The time in which the Historical Viewing stops playing.
jogIncrement	The jog increment in s (seconds), m (minutes), h (hours), d (days), or w (weeks).
transitionSpeed	The transition speed in s (seconds).
aggregateType	Displays the aggregation type. The default type is Interpolate.
aggregateInterval	Displays the aggregation interval. The default interval is 30 seconds (30s).

Example: `startHistoryNavigationSlideShow(Now(), DateAdd("hour", -2, Now()), "10m", "3s")`

The Historical Viewing starts playing at the start time, which is set to the current time, with an end time two hours before the current time. The jog increments is set to 10 minutes, and the transition speed is 3 seconds.



For more information, go to the topic, [Historical Viewing and Navigation](#), located in the XHQ Developer's Guide.

subscribeToClientVariable

This system function allows an expression to subscribe to an application (client) variable. Whenever the referenced client variable is changed, the expression is notified and the new value is returned. Since client variables have only string values, this function returns a string.

Syntax: `subscribeToClientVariable(string)`

EXAMPLE: USING THE SUBSCRIBETOCLIENTVARIABLE FUNCTION

In the following example, the view consists of three view elements:

- A Button control
- An Edit Box
- A Value item

The **Button** sets the client variable. Select the button control and click the **Button tab**. Enter these values:

Label: Set

Expression: `setClientVariable("Foo",$EditBox1)`

The **Edit Box** gets a value for the client variable.

The **Value** item then uses `subscribeToClientVariable()` to set its value animation. Select the value item, click the **Animation tab**, and click **Edit**. The "Edit Expression" dialog box appears.

From the "Edit Expression" dialog box, enter `subscribeToClientVariable("Foo")` and click **OK**.

VTQ Functions

XHQ supports the following VTQ (Value, Time, Quality) functions for use within expressions.

Quick Reference - VTQ Functions

Function	Description	Returns This Value Type:
<code>description(AnyType a)</code>	Returns the description associated with the value. If none is configured, then it returns an empty string/blank. This shows value metadata and is used with primitive members/tags.	String
<code>expiration(AnyType a)</code>	Returns the expiration timestamp associated with the value. See the <code>timestamp()</code> function below for more information. Note: The expiration function is only supported in server-side expressions.	DateTime
<code>isConfigured(AnyType a)</code>	Returns FALSE if the data quality of the value indicates it is NOT configured.	Boolean
<code>isInHistoricalMode(AnyType a)</code>	Returns TRUE if the data quality of the value is in historical mode.	Boolean
<code>quality(AnyType a)</code>	Returns the quality of the value.	Quality
<code>setQuality(AnyType a, Quality q)</code>	Returns the value with the quality q applied to it.	AnyType
<code>source(AnyType a)</code>	Returns the source of the value. If the source is unknown, then it returns an empty string/blank. This shows value metadata and is used with primitive members/tags. <i>Format:</i> Source type(Specifics) <i>Examples:</i> <code>EXPRESSION(Server)</code> <code>USER(joesmith)</code> <i>Source types:</i> (Note, these are not localized, English only) <ul style="list-style-type: none"> • CONNECTOR The source of value is a data connector. The name of connection group will appear in parentheses. • EXPRESSION The source of value is an expression. Specific source description will be Server or 	String

Function	Description	Returns This Value Type:
	<p>Client.</p> <ul style="list-style-type: none"> INTERNAL This is an internally set value (system value). USER The User input value. The user name appears in parentheses. CALCULATION The value from a calculation, currently reserved for future use. EXTERNAL The value from an external program write. NONE No source is associated with the value. 	
<code>time(AnyType a)</code>	Returns a string representing the "time stamp" of the value.	String
<code>timestamp(AnyType a)</code>	<p>Returns the timestamp associated with the value.</p> <p>Note: The timestamp function is only supported in server-side expressions.</p>	DateTime
<code>units(AnyType a)</code>	<p>Returns the Unit of Measure (uom) associated with the value. If none is configured, then it returns an empty string/blank.</p> <p>This shows value metadata and is used with primitive members/tags.</p>	String

quality

Returns the quality of the value.

Syntax: `quality(AnyType a)`

Argument Type(s)	Return Value
AnyType a	<p>Quality</p> <p>Possible qualities are:</p> <ul style="list-style-type: none"> GOOD BAD UNCERTAIN

Example(s):

Given some real member "aRealMember" with an alias of "RM001"

Quality(aRealMember) returns GOOD

Quality(SetQuality(57.3, BAD)) returns BAD

setQuality

Returns the value with the quality *q* applied to it.

Syntax: `setQuality(AnyType a, Quality q)`

Example: `setQuality(57.3, BAD)` returns 57.3 with a BAD quality

time

Returns a string representing the *timestamp* of the value.

Syntax: `time(AnyType a)`

Examples: `Time(aRealMember)` returns Wed Jul 4 15:37:02 PDT 2001
 `DateValue(Time(aRealMember), "EEE MMM d HH:mm:ss z yyyy")`
 returns 4-Jul-01 15:37:02 PDT

Metadata Function

XHQ supports the following metadata function for use within expressions. A metadata function returns a **String** value.

Quick Reference – Metadata Function

Function	Description
<code>getProperty(String property, Reference r)</code>	<p>Returns the property of this reference. Property tag values (must be in double quotes):</p> <ul style="list-style-type: none"> • alias • aliasOrPath (alias returned if it exists, otherwise the path is returned) • description • name • path • shortPath

getProperty

Returns a string for the given property of this reference.

Syntax: `getProperty(String property, Reference r)`



Property tag values must be in **double quotes**.

For the **String property** parameter use the following properties:

Property	Description
Alias	If it exists, the alias is returned.
AliasOrPath	<p>If it exists, the alias is returned; otherwise, the path is returned.</p> <p>This property is for primitive members only.</p> <p>If the reference is a collection or object member, an empty string is returned.</p>
Path	<p>The full path is returned.</p> <p>If the reference is a nested collection or a member of a nested collection, an empty string is returned.</p>
ShortPath	An abbreviated path is returned.
Name	The name of the member at this instance is returned.
Description	The description of the member is returned.

Example(s): Given some real member "aRealMember" with an alias of "RM001"

```
getProperty("alias", aRealMember)
```

Returns the string: RM01

getProperty("aliasOrPath", aRealMember)

Returns the string: RM01

getProperty("path", aRealMember)

Returns the string:

::Enterprise.Baytown.Hydrocracker.Pump.aRealMember

getProperty("name", aRealMember)

Returns the string: aRealMember

getProperty("shortPath", aRealMember)

Returns the string: Hydrocracker.Pump.aRealMember

getProperty("description", aRealMember)

Returns the string: This is the member description.

Type Conversion Functions

XHQ supports the following type conversion functions for use within expressions. A type conversion function converts either a string value to an integer or an integer to a string.

Quick Reference - Type Conversion Functions

Function	Argument Type	Description	Returns Type
cryptographicHash(s, f)	String, String	Returns the byte encoding as a formatted string for the given input string, s, and the named cryptographic algorithm, f.	String
cryptographicHash(t, f)	Text, String	Returns the byte encoding as a formatted string for the given input text, t, and the named cryptographic algorithm, f.	String
decimalToString(m[, r])	Decimal[, Integer]	Converts the decimal to a string. An optional radix range of 2..36 may be supplied.	String
decimalValue(a[, r])	Numeric String[, Integer]	Coerces a Numeric value to a decimal. Returns the decimal value of a string. An optional radix range of 2..36 may be supplied.	Decimal
doubleToString(d)	Double	Converts the double to a string.	String
doubleValue(s)	Numeric String	Coerces a Numeric value to a double. Returns the double value of a string.	Double
hash (s)	String	Returns the hash value of the supplied string as an integer. The value is calculated with the Java string hash function.	Integer
hash (t)	Text	Returns the hash value of the supplied text as an integer. The value is calculated with the Java string hash function.	Integer
hash (s, f)	String, String	Returns a hash value of the supplied string according to the hash function name supplied. The returned value is a natural number (integer; or a large integer form of decimal, larger than a 64-bit integer). The size of the number is 32-bits or greater and is defined by the function used.	Decimal
hash (t, f)	Text, String	Returns a hash value of the supplied text according to the hash function name supplied. The returned value is a natural number (integer). The size of the number is 32-bits or greater and is defined by the function used.	Decimal
integerToString(i[, r])	Integer[, Integer]	Converts the integer to a string. An optional radix range of 2..36 may be supplied.	String
integerValue(s[, r])	Numeric	Coerces a Numeric value to an integer. An	Integer

Function	Argument Type	Description	Returns Type
		optional radix range of 2..36 may be supplied.	
	String[, Integer]	Parses the given string for a properly formatted integer value. An optional radix may be supplied	
intervalValue(m)	Long	Returns a time interval value based on the number of milliseconds given	TimeInterval
intervalValue(s)	String	Accepts ISO 8601 format. Similar to dateInterval (interval_string). When using the XHQ API (for example, <code>XhqWI</code> , <code>XhqWse</code>) to retrieve, the value is "365:5:49:12:0" (365 days, 5 hours, 49 minutes and 12 seconds). In XHQ Solution Builder, however, the value is displayed as "P1Y" due to the nondeterministic year. This also applies to months.	TimeInterval
longToString(l[, r])	Long[, Integer]	Converts the long to a string. An optional radix range of 2..36 may be supplied.	String
longValue(s[, r])	Numeric	Coerces a Numeric value to a long. An optional radix range of 2..36 may be supplied.	Long
	String[, Integer]	Parses the given string for a properly formatted integer value. An optional radix may be supplied.	
milliseconds(ti)	Long	Returns the value of the milliseconds field in the given time interval.	Decimal
parseDecimal(s)	String	Parses the given string for a properly formatted decimal value and returns the value if found.	Decimal
parseDouble(s)	String	Parses the given string for a properly formatted double-precision real value and returns the value if found.	Double
parseInteger(s)	String	Parses the given string for a properly formatted integer value and returns the value if found.	Integer
parseLong(s)	String	Parses the given string for a properly formatted long integer value and returns the value if found.	Long
parseReal(s)	String	Parses the given string for a properly formatted double-precision real value and returns the value if found.	Real
qualityToString(q)	Quality	Converts the quality to a string.	String
realToString(r)	Real	Converts the real to a string.	String
realValue(a)	Numeric	Coerces a Numeric value to a real.	Real
	String	Returns the real value of a string.	
stringToText(s)	String	Converts a string type to a text type.	Text

Function	Argument Type	Description	Returns Type
stringValue(n)	Numeric	Converts the numeric to a string.	String
textToString(t)	Text	Converts a string type to a text type.	String



If conversion is not possible, or results in a numeric overflow, the quality of the return value is set to BAD.

Coercion Function Errors

For numeric conversions, the resulting value might not be presentable in the defined return type. When this type or error occurs, the quality of the result is set to BAD with a sub-quality that indicates the type of error (for example, overflow, underflow).

Supported Hash Functions

Cryptographic and Non-cryptographic Hash Functions Support

Function Name	Hash Type	Bit Size	Supported?
Java	Non-Cryptographic	32 bits	Yes
MD5	Cryptographic	128 bits	In future release
MurmurHash3-32	Non-Cryptographic	32 bits	Yes
MurmurHash3-128	Non-Cryptographic	128 bits	Yes
SHA-1	Cryptographic	160 bits	In future release
SHA-224	Cryptographic	224 bits	In future release
SHA-256	Cryptographic	256 bits	In future release
SHA-384	Cryptographic	384 bits	In future release



The `cryptographicHash` is currently not supported but is planned for in a future release.

Conversion Functions with Radix Parameter

The functions, `decimalValue`, `integerValue`, and `longValue`, for example, may take an optional **radix** that is used to interpret the characters in the given string when converting to a number. This base must be in the range of 2..36 and the string must only contain characters supported by the given base. Upper and lower case versions of alphabetic characters corresponding to bases 11..36 are accepted.

Parse Functions

The parsing functions use the formatting of the string to determine the radix (8, 10, or 16). In addition, the following rules apply:

- If the string starts with 0x, the base is assumed to be 16 (hexadecimal).
- If the string starts with a 0 and is followed by another number, the base is assumed to be 8 (octal).
- Strings not starting with 0 are assumed to be base 10 (decimal).

Date/Time Functions

With date/time functions you can define server- and client-side expressions that manipulate date and time data. These expressions also enable you to calculate delta times, using various time intervals.

Quick Reference - Date Time Functions

Function	Description	Return Type
<code>date()</code>	Returns the current date.	DateTime
<code>dateAdd(interval_string, number)</code>	Returns a date/time with added number of time units specified by the <code>interval_string</code> from current date.	DateTime
<code>dateAdd(interval_string, number, date_time)</code>	Returns a date/time with added number of time units specified by the <code>interval_string</code> from the specified <code>date_time</code> .	DateTime
<code>dateAdd(time_interval)</code>	Returns a date/time with the added time interval from current date.	DateTime
<code>dateAdd(time_interval, date_time)</code>	Returns a date/time with the added time interval from the specified <code>date_time</code> .	DateTime
<code>dateDiff(interval_string, date_time)</code>	Returns the difference between current date and specified <code>date_time</code> in units, <code>interval_string</code> .	Integer
<code>dateDiff(interval_string, date_time1, date_time2)</code>	Returns the difference between the dates, <code>date_time1</code> and <code>date_time2</code> , in units, <code>interval_string</code> .	Integer
<code>dateDiff(date_time)</code>	Returns the difference between current date and specified <code>date_time</code> as a time interval.	TimeInterval
<code>dateDiff(date_time1, date_time2)</code>	Returns the difference between the dates, <code>date_time1</code> and <code>date_time2</code> , as a time interval.	TimeInterval
<code>dateEnd(interval_string)</code>	Returns the date/time for the end of the specified period, <code>interval_string</code> .	DateTime
<code>dateEnd(interval_string, date_time)</code>	Returns the date/time for the end of the specified period, <code>interval_string</code> , beginning at specified <code>date_time</code> .	DateTime
<code>dateFormat(date_time)</code>	Aliases for analogous <code>format()</code> variations. See String Functions .	String
<code>dateFormat(date_time, format_string)</code>	Aliases for analogous <code>format()</code> variations. See String Functions .	String
<code>dateFormat(format_string)</code>	Aliases for analogous <code>format()</code> variations. See String Functions .	String
<code>dateInterval(interval_string)</code>	Returns a time interval for the ISO2801 encoded string.	TimeInterval

Function	Description	Return Type
<code>dateInterval(interval_string, real)</code>	Returns a time interval for the number of time units, <code>real</code> , of the specified interval type specified by <code>interval_string</code> (for example, "day", "hours") from current date/time.	TimeInterval
<code>dateInterval(interval_string, real, date_time)</code>	Returns a time interval for the number of time units, <code>real</code> , of the specified interval type specified by <code>interval_string</code> (for example, "day", "hours") from the specified date/time, <code>date_time</code> .	TimeInterval
<code>dateOffset()</code>	Returns the time interval value of the difference between the local time zone and the server time zone. Always zero for server-side expressions.	TimeInterval
<code>dateOffset(string)</code>	Returns the time interval value of the difference between the specified time zone, <code>string</code> parameter, and the server time zone.	TimeInterval
<code>dateOffset(date_time)</code>	Returns the time interval value of the difference between given date, <code>date_time</code> , and the server time zone.	TimeInterval
<code>dateOffset(date_time, timezone)</code>	Returns the time interval value of the difference between given date, <code>date_time</code> , in the specified time zone, <code>timezone</code> , and the server time zone.	TimeInterval
<code>datePart(interval_string)</code>	Returns the integer value of the specified part of the date, <code>interval_string</code> , of the current date.	Integer
<code>datePart(interval_string, date_time)</code>	Returns the integer value of the specified part of the date, <code>interval_string</code> , of the specified date, <code>date_time</code> .	Integer
<code>datePart(interval_string, time_interval)</code>	Returns the integer value of the specified part of the date, <code>interval_string</code> , of the current date with the added <code>time_interval</code> .	Integer
<code>datePart(interval_string, time_interval, date_time)</code>	Returns the integer value of the specified part of the date, <code>interval_string</code> , of the specified <code>date_time</code> , with the added <code>time_interval</code> .	Integer
<code>dateRound(interval_string)</code>	Returns the current date rounded the nearest <code>interval_string</code> .	DateTime
<code>dateRound(interval_string, date_time)</code>	Returns the specified <code>date_time</code> , rounded to the nearest <code>interval_string</code> .	DateTime
<code>dateSerial(yyyy, month, day)</code>	Returns a date/time for the given four-digit	DateTime

Function	Description	Return Type
	year, month, and day in the time zone of executing context.	
<code>dateStart(interval_string)</code>	Returns the date/time for the start of the specified period, <code>interval_string</code> .	DateTime
<code>dateStart(interval_string, date_time)</code>	Returns the date/time for the start of the specified period, <code>interval_string</code> , beginning at <code>date_time</code> .	DateTime
<code>dateSub(interval_string, number)</code>	Returns the resulting date of subtracting the given number, of units, <code>interval_string</code> , from the current date.	DateTime
<code>dateSub(interval_string, number, date_time)</code>	Returns the resulting date of subtracting the given number, of units, <code>interval_string</code> , from the specified <code>date_time</code> .	DateTime
<code>dateSub(time_interval)</code>	Returns the resulting date of subtracting the given time interval, from the current date.	DateTime
<code>dateSub(time_interval, date_time)</code>	Returns the resulting date of subtracting the given time interval, from the specified <code>date_time</code> .	DateTime
<code>dateValue(date_string)</code>	Converts the given string, <code>date_string</code> , to a date/time.	DateTime
<code>dateValue(date_string, format_mask)</code>	Converts the given string, <code>date_string</code> , to a date/time using the provided date format, <code>format_mask</code> .	DateTime
<code>dateValue(date_string, format_mask, country)</code>	Converts the given string, <code>date_string</code> , to a date/time using the provided date format, <code>format_mask</code> , and default local rules for the country.	DateTime
<code>dateValue(date_string, format_mask, country, language)</code>	Converts the given string, <code>date_string</code> , to a date/time using the provided date format, <code>format_mask</code> , and default local rules for the country and language.	DateTime
<code>intervalPart(string, time_interval)</code>	Returns the integer value of the specified part of the interval, <code>string</code> , for the given time interval.	Integer
<code>now()</code>	Returns the current date/time.	DateTime
<code>timeSerial(hours, minutes, seconds)</code>	Returns a time interval from the given hours, minutes, and seconds.	TimeInterval

Time Manipulation

XHQ allows you to define server- and client-side expressions that manipulate date and time. The results of these expressions are then displayed in a view or are stored in existing numeric or string members of a component. These expressions can be used to access historical values of any member with history defined.

This section helps you learn the concepts behind the use of these expressions and covers important things to consider when creating them.

Date and Time Data Types

A data type identifies the type of information that is processed by the application. For example, in an XHQ solution, component primitive members hold values of variables that are of the following data types:

- Boolean
- Integer
- Long
- Real
- Double
- Decimal
- Date/Time
- Interval
- String
- Text

The date/time specific data types (such as Date/Time and Interval) allow you to convert timestamps to and from existing string data types. Functions within expressions can use dates, but any server-side expressions that set the value in the solution must be converted to a string or an integer value first.

For example, the string member `dateAdd("hour", 1)` is not a valid server-side expression since it returns a date/time value. However, `dateFormat(dateAdd("hour", 1))` is valid because the function `dateFormat` returns a string.

Considering Time in Different Locations

When using date/time expressions, differing time zones and time changes can play an important factor in your calculations, impacting your return values.

This section will help you determine whether or not you need to consider these issues.



For more information on Daylight Saving Time, refer to the topic, [DST and XHQ](#).

Time Zones and XHQ

XHQ supports the three-character abbreviation used to represent the time zone (for example, "PST" for Pacific Standard Time).



If you change the time zone on the XHQ Server, you will need to restart XHQ.

Since each time zone differs from the preceding and the following zone by one hour, they may effect your client-side time expressions if the server and clients are located in different time zones.

Consider the following example.

The XHQ Server is located in the Mountain Standard Time (MST) time zone. You, the client, are located in the Pacific Standard Time (PST) time zone, which is one hour behind MST.

You use the expression `dateFormat(dateEnd("day"), "d-MMM-yyyy hh:mm:ss z")` to return the string representation of date/time for the end of the day. The result is "6-Nov-00 23:59:59 PST". This, however, does not reflect the 1-hour difference between the MST and the PST time zones.

To do this, you will need to add (using the `dateAdd` function) the 1-hour UT/GMT offset (using the `dateOffset` function) between you (the client) and the XHQ server.

The expression to execute is `dateFormat(dateAdd(dateOffset(), dateEnd("day")), "d-MMM-yyyy hh:mm:ss z")`. The result in this case is "6-Nov-00 22:59:59 PST". Compare this result to the previous result and notice the time difference.



The United States and its territories observe eight time zones which include both Standard Time and Daylight Time.

SIDEBAR: ABOUT STANDARD TIME AND TIME ZONES

Standard Time is the specified time in any of the 24 international time zones around the world and is designated as a number of hours earlier or later than Universal Time (UT).

Universal Time is the precise measure of time used as the basis for all civil time keeping. It is the mean solar time determined by the meridian that runs through Greenwich, England. For the most part, UT is equivalent to Greenwich Mean Time (GMT).

Standard Time within each time zone is a given number of hours offset from the UT.



To determine your time zone, refer to this web site:
<http://www.worldtimezone.com/time-world.htm>



For the XHQ HTML5 Solution Viewer

At times, there's an issue with JavaScript determining the user's current timezone abbreviation. For example, instead of displaying the correct MSK timezone, the TRT timezone is incorrectly given (which are both UTC+03).

DST and XHQ

During Daylight Saving Time (DST), clocks are adjusted one hour ahead during spring, and adjusted back one hour to Standard Time every autumn.



There are many locales that do not observe DST (also referred to as Summer Time). Especially countries in the equatorial region where daylight hours are similar during every season, thus posing no advantage to moving clocks ahead.

At the transition from Standard Time to DST, an hour is skipped and therefore that day has only 23 hours instead of 24. And the day when the DST period ends and Standard Time begins has 25 hours. Therefore, the DST change may effect expressions that are hour related. This can be an important consideration in some calculations.

For example, you want to calculate the elapsed portion of a day and you use the expression:

```
dateDiff("minute", DateStart("day")) / 60.0 / 24
```

This function returns a real value between "0" and "1". Though this expression will be correct most of the time, it unfortunately does not account for DST changes.

The following expression is more reliable.

```
1.0 * dateDiff("minute", dateStart("day")) / dateDiff("minute", dateStart("day"), dateEnd("day"))
```

This accurately calculates the elapsed portion of the day by providing for DST.



Since both `dateDiff()` return integer values, the result is multiplied by a factor of 1.0, converting it to a real value.
There is a difference between the integer "1" and the real value "1.0."

**WARNING**

The XHQ system does not support clearing (deselecting) the "Automatically adjust clock for daylight saving changes" checkbox due to an external issue with Java 1.1.x. To make sure that this checkbox is selected, open the **Control Panel**, click **Date and Time**. The "Date and Time Properties" dialog box appears. Click the **Time Zone** tab.

SIDEBAR: ABOUT DAYLIGHT SAVING TIME

Daylight Saving Time (DST) is a system in which clocks are set ahead a certain amount of time (which ranges from 20 minutes to 2 hours, depending on your country) for a specific period.

There are approximately 70 countries that use DST in at least a portion of the country. There are also many countries that do not observe DST, especially countries in the equatorial region where daylight hours are similar during every season, thus posing no advantage to moving clocks ahead.

Daylight Saving Time is also referred to as Summer Time in many countries since it usually occurs during summertime. Countries on the Northern Hemisphere usually observe DST during the months between March/April through September/October. In contrast, summer comes in December to countries located in the Southern Hemisphere. Therefore, DST is usually observed between October and March.



This site lists countries that observe DST and gives their start and end dates.

<http://webexhibits.org/daylightsaving/g.html>

This site provides a listing of major international cities that are currently in DST.

<http://www.timeanddate.com/worldclock/full.html>

Working with Date, Time and Time Intervals

Time manipulation involves three elements:

- The **Date**

Example: "20-Jul-2000"

- The **Time**

Example: "10:26:37.763 am"

- The **Time Interval**

Example: "6" weeks

In this section, you will learn how to use these elements in a format supported by XHQ.

Using AM and PM

If the format contains an AM or PM, the hour is based on the 12-hour clock. "AM" indicates times from midnight until noon and "PM" indicates times from noon until midnight. Else, the hour is based on the 24-hour clock.

Default Date/Time Format

There are a couple of factors that determine how the **default** Date/Time is displayed. When parsing a date, the default date/time format is based on the operating system's locale settings.

Example: 07/04/2020 2:28 PM

However, when outputting a date/time to a view (conversion from a date/time data type to a string), the default date format is fixed.

Example: 4-Jul-20 13:00:00

Date Format Rules

The **date** element must comply with the following format rules.

Date Format Rules

Format Description	Example
The day, month, and the year can be in any order. If converting from a string, the system allows the user to specify the order. There is, however, a default order that is based on the operating system's locale settings.	"7-4-76" or "4-7-76" or "76-7-4"
You can use a three-letter month name, a month number or a full month name.	"Jul," "7," or "July"
You can use a space, period (.), dash (-), comma (,), comma and space (,), or forward slash (/) as a separator between individual parts of the day.	"July 4, 1776" "July/4/1776"
Years can be displayed in 2- or 4-digits. If converting from a string with a two-digit year, the century is assumed to be the year closest to the current year.	If "64" is the year in the string and the current year is "2000", then the year will be interpreted as "1964." If "64" is the year in the string and the current year is "2020", then the year will be interpreted as "2064."

Time Format Rules

The **time** element must comply with the following format rules.

Time Format Rules

Format Description	Example
Use a 24-hour clock with either an uppercase or lowercase "a.m./p.m."	"22:23" or "10:23 PM"
Use exactly two digits for the minutes and seconds and	"07:02:09"

Format Description	Example
optionally for the hour.	"7:02:09"
Use either a colon (:) or a period (.) as the separator character between the parts of time.	"02:15"
Use the letter "h" as the separator between the hours and minutes.	"2h15"
<i>Optional</i> Include seconds.	"10:23" or "10:23:35"
<i>Optional</i> Include milliseconds only if the seconds are included.	"10:23:35" or "10:23:35.763"
<i>Optional</i> Include the offset from GMT. The GMT offset must always have a sign followed by a two-digit hour, a colon, and a two-digit minute. For conversion from a string, the locale at the place of execution (client or server) will be used if no GMT offset is included.	"10:23" or "10:23-08:00"
<i>Optional</i> Include a three-letter time zone (daylight saving designator).	"PDT" "CST"

Expressing Time Intervals

In the XHQ System, time intervals can be expressed as an integer in the following ways.

- An integer number representing the number of weeks, days, hours, or seconds. Real numbers can be derived by using a smaller interval.
- An integer number of months or years.
- An integer number of months, weeks or days that adjusts for daylight saving time changes when converted to hours, minutes, or seconds. These intervals require a specific reference date.

Time Interval Format Rules

The **time** element must comply with the following format rules.

Time Interval Format Rules

With time intervals, you can:	Example
Use the ISO 8601 extended format and is expressed as "PnYnMnDTnHnMnS", where "nY" represents the number of years, "nM" the number of months, "nD" the number of days, "T" is the date/time separator, "nH" the number of hours, "nM" the number of minutes, and "nS" the number of seconds.	"P1Y2M3DT10H30M" indicates a duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes.
Display the number of seconds with decimal digits separated by a period (.) followed by milliseconds.	"P1Y2M3DT10H30M35.743S" indicates a duration of 1 year, 2 months, 3 days, 10 hours, 30 minutes, 35 seconds, and 743 milliseconds.
<i>Optional</i> Precede the time interval with a minus sign (-) to indicate a negative duration. If no sign is displayed, a positive duration is assumed.	"-P1Y2M3DT10H30M"
Use a time-like format, indicating the number of hours, minutes, and seconds, delimited by either colons (:) or periods (.).	"49:27:16"
Display hours in two digits or one.	"1:27:16" or "01:27:16"
<i>Optional</i> Precede the hours with the number days followed by a space.	"2 1:27:16"

Using the Date/Time Functions

The table below provides some examples of what you can do with date/time functions.

Uses for Date/Time Functions

Task	Example Expressions
Calculate an absolute time from another absolute time and a delta time, using various time intervals.	
number of months between two dates	<code>dateDiff("month",dateValue("10-01-86"),dateValue("10-01-01"))</code>
the date two weeks after another date	<code>dateAdd(dateInterval("14 0:00:00"),dateValue("10-01-00"))</code>
45 seconds after the current time	<code>dateAdd(dateInterval("0:00:45"),now())</code>
Extract parts of a timestamp.	
month	<code>datePart("month")</code>
year	<code>datePart("year")</code>

Task	Example Expressions
Convert timestamps to a string.	
8-Apr-2000 15:38:17.456 PDT	<code>dateFormat("d-MMM-yyyy HH:mm:ss.SSSS z")</code>
4/8/00 15:38	<code>dateFormat("M/d/y HH:mm")</code>
April 8, 2000 3:38	<code>dateFormat("MMM d, yyyy h:mm")</code>
8/4/00	<code>dateFormat("M/d/y")</code>
Support special absolute and relative times.	
"beginning of this month" (midnight on the 1 st)	<code>dateStart("month")</code>
"end of last month" (one millisecond before midnight on the last day of the previous month)	Given: The current month is November. <code>dateEnd("month", dateValue("10-01-00"))</code>
"beginning of day" (midnight today)	<code>dateStart("day")</code>
"end of day" (one millisecond before midnight yesterday)	<code>dateEnd("day")</code>
"beginning of yesterday" (a 24 hour period beginning at midnight yesterday)	<code>dateAdd("day", -, dateStart("day"))</code>
"last week" (a seven day period beginning at midnight on Sunday of last week)	<code>dateAdd("week", -1)</code>
"tomorrow"	<code>dateAdd("day", 1)</code>
"today"	<code>date()</code>
"now"	<code>now()</code>



Currently, time-dependent, *server-side* expressions do not update unless triggered. To trigger, you simply connect the primitive inventory item to the Simulator. Conversely, time-dependent, *client-side* expressions update whenever anything on the view changes.

Defining Parameters

Each date/time function, with the exception of the `now()` and `date()` functions, use parameters.

The table below lists and describes all the supported parameters.

List of Parameters

Parameter	Description
<code>country</code>	This is the two-letter string corresponding to the ISO code for country. Functions that use this parameter: <code>dateValue</code>
<code>date_time</code>	The <code>date_time</code> parameter must be the result of another time function.

Parameter	Description
	<p><u>Valid Example:</u> dateValue("10-27-01")</p> <p><u>Invalid Example:</u> "10-27-01"</p> <p>Functions that use this parameter: dateAdd, dateDiff, dateEnd, dateFormat, dateInterval, datePart, dateRound, dateStart, format</p>
date_string	<p>Omitted parts of the time will default to zero (relative to local time) and omitted parts of the date defaults to the current period (for example, if you omit the year then the default is the current year).</p> <p>Functions that use this parameter: dateValue</p>
day	<p>This parameter may be of type real, but will be truncated to an integer.</p> <p>Functions that use this parameter: dateSerial</p>
format_mask	<p>Similar to the "format_string" parameter.</p> <p>Common "format_mask" parameters include:</p> <p>"yyyy" for year (for example, 1963); "MMMM" for month (for example, July); "hh" for hour in am/pm (for example, 12:00); "zzzz" for time zone (for example, Pacific Standard Time)</p> <p>For a complete list of supported <code>format_string</code> parameters, refer to the Using the format_string parameter.</p> <p>Functions that use this parameter: dateValue</p>
format_string	<p>Common "format_string" parameters include:</p> <p>"yyyy" for year (for example, 1963); "MMMM" for month (for example, July); "hh" for hour in am/pm (for example, 12:00); "zzzz" for time zone (for example, Pacific Standard Time)</p> <p>For a complete list of supported <code>format_string</code> parameters, refer to the Using the format_string parameter.</p> <p>Functions that use this parameter: dateFormat, format</p>
YYYY	<p>The year, using four digits only. This parameter may be of type real, but will be truncated to an integer.</p> <p>Functions that use this parameter: dateSerial</p>

Parameter	Description
hours	<p>This parameter may be of type real, but will be truncated to an integer.</p> <p>Functions that use this parameter:</p> <p>timeSerial</p>
interval_string	<p>A unit of time. Common parameters include: "year"; "month"; "week"; "hour"; "y" (which is the "day of the year" and not the year)</p> <p>For a complete list of supported <code>interval_string</code> parameters, refer to the Using the interval_string parameter topic.</p> <p>Functions that use this parameter:</p> <p>dateAdd, dateDiff, dateEnd, dateInterval, datePart, dateRound, dateStart</p>
language	<p>This is the three-letter string corresponding to the ISO code for language.</p> <p>Functions that use this parameter:</p> <p>dateValue</p>
minutes	<p>This parameter may be of type real, but will be truncated to an integer.</p> <p>Functions that use this parameter:</p> <p>timeSerial</p>
month	<p>This parameter may be of type real, but will be truncated to an integer.</p> <p>Functions that use this parameter:</p> <p>dateSerial</p>
number	<p>The <code>number</code> parameter may be of type real, but will be truncated to an integer.</p> <p>Functions that use this parameter:</p> <p>dateAdd, dateInterval</p>
seconds	<p>This parameter may be of type real, but will be truncated to an integer.</p> <p>Functions that use this parameter:</p> <p>timeSerial</p>
string	<p>For the "hours, minutes, seconds" format, the "hours" must be specified and any missing parts will default to zero (for example, "3:12" implies zero days and zero seconds).</p> <p>Functions that use this parameter:</p> <p>dateInterval, intervalPart</p>
time zone	<p>The time zone is represented by a three character abbreviation (for example, "PDT" for "Pacific Daylight Time").</p> <p>For more information on time zones, refer to the topic on Time Zones and XHQ.</p> <p>Functions that use this parameter:</p> <p>dateOffset</p>
time_interval	<p>The time interval must be the result of another time function.</p>

Parameter	Description
	<u>Valid Example:</u> dateInterval("4 3:05:01")
	<u>Invalid Example:</u> "4 3:05:01"
	Functions that use this parameter: dateAdd, intervalPart

Using the `format_string` parameter

The date and time formats use a pattern string. In this pattern, the following ASCII letters are reserved as pattern letters.

ASCII Letters

Letter	Meaning	Output	Example(s)
G	Era designator	Text	AD
y	Year	Year	2016 or 16
Y	Week year	Year	2009 or 09
M	Month in year	Month	July or Jul or 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2 (as in the 2 nd Wednesday in July)
E	Day name in week	Text	Tuesday or Tue
u	Day number of week (1=Monday, ..., 7=Sunday)	Number	7
a	AM or PM marker	Text	PM
H	Hour in day (0 to 23)	Number	0
k	Hour in day (1 to 24)	Number	24
K	Hour in AM/PM (0 to 11)	Number	0
h	Hour in AM/PM (1 to 12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone (general)	Time zone	Pacific Standard Time or PST

Letter	Meaning	Output	Example(s)
			or GMT-08:00
Z	Time zone (RFC 822)	Time zone	-0800
X	Time zone (ISO 8601)	Time zone	-08 (for X), or -0800 (for XX), or -08:00 (for XXX)

The number of letters you use determines the format. Here are some rules to follow.

- For **Text** outputs

If you designate 4 or more pattern letters, then the "full form" is outputted. If less than 4 letters, then the short or abbreviated form (if one exists) is outputted.

Example: z outputs PST

zzzz outputs Pacific Standard Time

- For **Number** outputs

The number of letters you use represents the minimum number of digits to output. Shorter numbers are zero-padded to this amount.

Example: ss outputs 2 digits of the seconds in the minute

- For **Year** outputs (for the Gregorian calendar)

Years are handled specially; if the count of 'y' is 2, then the year will be truncated to the last 2 digits.

Example: yyyy outputs 2016

yy outputs 16

- For **Month** outputs

If you designate 3 or more pattern letters, then the text version is used. Else, a number is outputted.

Example: MMMM outputs July

MMM outputs Jul

MM outputs 07

M outputs 7

- For **Time Zone, general** outputs

Time zones are interpreted as text if they have names. Note, format is locale independent. For a GMT offset value, the following format is used:

Format GMT<sign><hours>:<minutes>

Where:

- <sign> is either + or -
- <hours> is between 0 and 23
- <minutes> is between 00 and 59

Example: PST

GMT+3:59

GMT-08:00



For the XHQ HTML5 Solution Viewer

At times, there's an issue with JavaScript determining the user's current timezone

abbreviation. For example, instead of displaying the correct MSK timezone, the TRT timezone is incorrectly given (which are both UTC+03).

- For **Time Zone, RFC 822** outputs

The following 4-digit format is used:

Format <sign><twoDigitHours><minutes>

Where:

- <sign> is either + or -
- <twoDigitHours> must be between 00 and 23
- <minutes> must be between 00 and 59

Example: -0800

- For **Time Zone, ISO 8601** outputs

The number of pattern letters determines the format.

Format <sign><twoDigitHours> when X is used

or

<sign><twoDigitHours><minutes> when XX is used

or

<sign><twoDigitHours>:<minutes> when XXX is used

Where:

- <sign> is either + or -
- <twoDigitHours> must be between 00 and 23
- <minutes> must be between 00 and 59

Example: a single X returns -08

or

XX returns -0800

or

XXX returns -08:00

Symbols

Symbol	Meaning	Output
'	Escape for text	Delimiter
"	Single quote	Literal

Examples Using the US Locale

To display...	Use this pattern...
2001.07.04 AD at 12:08:56 PST	"yyyy.MM.dd G 'at' HH:mm:ss z"
Wed, Jul 4, '01	"EEE, MMM d, 'yy"
12:08 PM	"h:mm a"
12 o'clock PM, Pacific Standard Time	"hh 'o' clock' a, zzzz"
0:08 PM, PST	"K:mm a, z"

To display...	Use this pattern...
02001.July.04 AD 12:08 PM	"yyyyy.MMMMM.dd GGG hh:mm aaa"
Wed, 4 Jul 2001 12:08:56 -0700	"EEE, d MMM yyyy HH:mm:ss Z"
010704120856-0700	"yyMMddHHmmssZ"
2001-07-04T12:08:56.235-0700	"yyyy-MM-dd'T'HH:mm:ss.SSSZ"
2001-07-04T12:08:56.235-07:00	"yyyy-MM-dd'T'HH:mm:ss.SSSXXX"
2001-W27-3	"YYYY-'W'ww-u"

Using the *interval_string* parameter

The functions `dateStart`, `dateEnd`, `dateRound`, `dateAdd`, `dateDiff`, and `datePart` all have an `interval_string` parameter that expresses the following units.

interval_string Units

Unit	Description
ms	Millisecond
second	Second
minute	Minute
hour	Hour
day	Day of the Month
week	Week
weekday	Weekday
month	Month
year	Year
yy	Year modulo 100
julian	Day of Year
yyyy	Year
q	Quarter
m	Month
y	Day of Year
d	Day of the Month
w	Weekday
ww	Week
h	Hour

Unit	Description
n	Minute
s	Second

date

Returns the current date.



Because the XHQ system does not display dates unless it is a [string](#), this function requires the use of the `dateFormat` function in order to [display](#) a value. See the correct syntax in the examples below.

Syntax: `date()`
 `dateFormat(date())`

Parameter	Description/Format
N/A	N/A

Comments: When converted to a string or displayed, it is formatted using the default date format.

This function depends upon the current date and time.

Examples: *Given:* The current date and time is "November 1, 2003"

Expression: `date()`

Returns a **date**: "1-Nov-03 00:00:00"

Expression: `dateFormat(date())`

Returns a **string**: "1-Nov-03 00:00:00"

dateAdd with String

Returns a date/time with an added number of time units specified by an interval string, from the current date [or from the specified date].



Because the XHQ system does not display dates unless it is a [string](#), this function requires the use of the `dateFormat` function in order to [display](#) a value. See the correct syntax in the examples below.

Brackets around the `date` parameter indicate that it is optional.

Syntax: `dateAdd(interval_string, number[, date])`
 `dateFormat(interval_string, number[, date])`

Parameter	Description/Format
interval_string	Common "interval_string" parameters include: "year"; "month"; "week"; "hour"; "y" (which is the "day of the year" and not the year) For a complete list of supported "interval_string" parameter formats, refer to the topic, Using the interval_string parameter.
number	The "number" parameter may be of type real, but will be truncated to an integer. For example, adding a "1" , or a "1.0", or a "1.25", or a "1.8" all result in the same number (they are all equivalent). To subtract, you can use a negative number parameter or use the DateSub function .

Parameter	Description/Format
date	<p><i>Optional</i></p> <p>If specified, the date must be the result of another time function.</p> <p><u>Valid Example:</u> dateAdd("day",3,DateValue("10-27-01"))</p> <p><u>Invalid Example:</u> dateAdd("day",3,"10-27-01")</p> <p>If the date is <u>not</u> specified, then the integer number is added to the current date.</p>

Example(s):

Given: now()=3-Nov-03 14:20:15 PST

Expression: dateAdd("day",5)

Returns a **date**: "8-Nov-03 14:20:15"

Adds 5 days to the current date.

Expression: dateAdd("day",5,DateValue("10-01-03"))

Returns a **date**: "6-Oct-03 00:00:00"

Adds 5 days to the date specified.

Expression: dateFormat(dateAdd("week",-2))

Returns a **string**: "20-Oct-03 14:20:15"

Subtracts 2 weeks from the current date.

Expression: dateAdd("year",5.932)

Returns a **date**: "3-Nov-08 14:20:15"

The real number "5.932" is truncated to "5". Therefore, it adds 5 years to the current date.

Expression: dateFormat(dateAdd("month",-6.7))

Returns a **string**: "3-May-03 14:20:15"

The real number "-6.7" is truncated to "-6". Therefore, it subtracts 6 months from the current date.

dateAdd with TimeInterval

Adds the time interval to the date/time and returns the resulting **date**.



Because the XHQ system does not display dates unless it is a string, this function requires the use of the `dateFormat` function in order to display a value. See the correct syntax in the examples below.

Brackets around the `date_time` parameter indicate that it is optional.

Syntax:

```
dateAdd(time_interval[, date_time])
dateFormat(dateAdd(time_interval[, date_time]))
```

Parameter	Description/Format
time_interval	<p>The time interval must be the result of another time function.</p> <p><u>Valid Example:</u> dateAdd(dateInterval("4 3:05:01"))</p> <p><u>Invalid Example:</u> dateAdd("4 3:05:01")</p>

Parameter	Description/Format
date_time	<p><i>Optional</i></p> <p>If specified, the date must be the result of another time function.</p> <p><u>Valid Example:</u> <code>dateAdd(dateInterval("4 3:05:01"), dateValue("10-01-00"))</code></p> <p><u>Invalid Example:</u> <code>dateAdd(dateInterval("4 3:05:01"), "10-01-00")</code></p> <p>If the "date_time" parameter is <u>not</u> specified, then time interval is added to the <u>current</u> date.</p>

Example(s): *Given:* `now()`=3-Nov-00 15:30:15 PST
Expression: `dateAdd(dateInterval("4 3:15:15"))`
Returns a **date**: "7-Nov-00 18:45:30"
Adds 4 days to the current date, since the "date" parameter was not specified, and 3 hours, 15 minutes, and 15 seconds to the current time.

Expression: `dateFormat(dateAdd(dateInterval("4 3:00:00"),
dateValue("10-01-00")))`
Returns a **string**: "5-Oct-00 03:00:00"
Adds 4 days to the date, indicated by the "DateValue" function, and 3 hours to the midnight hour.

dateDiff with DateTime

Finds the difference between two date/times and returns a **time interval**.



Because the XHQ system does not display dates unless it is a string, this function requires the use of the `dateFormat` function in order to display a value. See the correct syntax in the examples below.

Brackets around the `date_time2` parameter indicate that it is optional.

Syntax: `dateDiff(date_time1[, date_time2])`
`dateFormat(dateDiff(date_time1[, date_time2]))`

Parameter	Description/Format
date_time1	<p>The date must be the result of another time function.</p> <p><code>dateDiff(dt1)</code> is equivalent to <code>dateDiff(dt1,now())</code>.</p> <p><u>Valid Example:</u> <code>dateDiff(dateValue("10-01-00"))</code></p> <p><u>Invalid Example:</u> <code>dateDiff("10-01-00")</code></p>
date_time2	<p><i>Optional</i></p> <p>If specified, the date must be the result of another time function.</p>

Parameter	Description/Format
	<p><u>Valid Example:</u> <code>dateDiff(dateValue("10-01-00"), dateValue("10-9-00"))</code></p> <p><u>Invalid Example:</u> <code>dateDiff(DateValue("10-01-00"), "10-9-00")</code></p> <p>If <code>datetime2</code> is <u>not</u> specified, then the <u>current</u> date is used to find the difference.</p>
<i>Comments:</i>	You may need to take Daylight Saving Time (DST) into account. For more information, refer to the topic, DST and XHQ .
<i>Example(s):</i>	<p><i>Given:</i> <code>now()</code>=3-Nov-00 15:55:15 PST</p> <p>Expression: <code>dateDiff (dateValue ("10-01-00"))</code> Returns a time interval: "33 15:55:15.487" The number of days difference between the given date at midnight and the current time.</p> <p>Expression: <code>dateFormat (dateDiff (dateValue ("10-01-00"), dateValue ("10-9-00"))))</code> Returns a string: "8 0:00:00" The number of days difference between the two dates (midnight to midnight).</p> <p>Expression: <code>dateDiff (dateValue ("10-27-01"), dateValue ("10-29-01"))</code> Returns a time interval: "2 1:00:00" In the United States, DST occurs on 10-28-01. Therefore, the extra hour is indicated.</p> <p>Expression: <code>dateFormat (dateDiff (now(), dateAdd (dateInterval ("59:05:01"))))</code> Returns a string: "2 11:05:01"</p> <p>Expression: <code>dateDiff (dateAdd (dateInterval ("59:05:01")), now())</code> Returns a time interval: "-2 11:05:01"</p> <p>Expression: <code>dateFormat (dateDiff (dateAdd (dateInterval ("59:05:01"))))</code> Returns a string: "-2 11:05:01" Results are similar to the previous expression. Therefore, this indicates that the "now()" function is implied.</p>

dateDiff with String

Finds the difference between two dates in the specified units (days, hours, and so forth) and returns an **integer**.



Brackets around the `date_time2` parameter indicate that it is optional.

Syntax: `dateDiff(interval_string, date_time1[, date_time2])`

Parameter	Description/Format
interval_string	<p>Common "interval_string" parameters include:</p> <p>"year"; "month"; "week"; "hour"; "y" (which is the "day of the year" and not the year)</p> <p>For a complete list of supported "interval_string" parameter formats, refer to the topic, Using the interval_string parameter.</p>

Parameter	Description/Format
date_time1	<p>The "date_time1" parameters must be the result of another time function.</p> <p>dateDiff(s, dt1) is equivalent to dateDiff(s,dt1,now()).</p> <p><u>Valid Example:</u> dateDiff("day",dateValue("10-01-00"))</p> <p><u>Invalid Example:</u> dateDiff("day","10-01-00")</p>
date_time2	<p><i>Optional</i></p> <p>If specified, the date must be the result of another time function.</p> <p><u>Valid Example:</u> dateDiff("day",dateValue("10-01-00"), dateValue("10-9-00"))</p> <p><u>Invalid Example:</u> dateDiff("day",dateValue("10-01-00"),"10-9-00")</p> <p>If "date_time2" is <u>not</u> specified, then the <u>current</u> date is used to find the difference.</p>
<i>Comments:</i>	<p>If the "interval_string" unit is less than a day (for example, hour, minute, second, millisecond), then Daylight Saving Time (DST) impact must be considered.</p> <p>If date_time1 is AFTER date_time2, then the result is <u>negative</u>.</p> <p>This function depends upon the current date and time.</p> <p>Since the return is an integer, fractions are truncated.</p>
<i>Example(s):</i>	<p><i>Given:</i> now()=6-Nov-00 9:30:15 PST</p> <p>Expression: dateDiff ("day", dateAdd (dateInterval ("24:05:01")))</p> <p>Returns an integer: "-1"</p> <p>Takes the difference between the given date interval and the current date and time.</p> <p>Expression: dateDiff ("hour", DateValue ("10-27-01"), dateValue ("10-29-01"))</p> <p>Returns an integer: "49"</p> <p>In the United States, DST occurs on 10-28-01. Therefore, the extra hour is indicated.</p> <p>Expression: dateDiff ("month", DateValue ("8-15-01"), dateValue ("7-1-01"))</p> <p>Returns an integer: "-1"</p> <p>Since date1 is AFTER date2, the result is negative.</p> <p>Expression: dateDiff ("month", DateValue ("8-15-01"), dateValue ("7-30-01"))</p> <p>Returns an integer: "-1"</p> <p>Note that the result is the same as the previous example. Again, fractions are truncated.</p>

THE DATEDIFF FUNCTION AND DST

The dateDiff function uses the String parameter, s, to pass a unit of time. It is important to note that the unit of time you use determines whether or not DST is taken into account.

If interval_string is less than a day (hour, minute, second, millisecond), the dateDiff function factors in the DST change, adjusting the number of hours.

If interval_string is a day or greater (day, week, month, and so forth), dateDiff ignores the DST change.

dateEnd

Returns the **date/time** for the end of the specified period.



Because the XHQ system does not display dates unless it is a [string](#), this function requires the use of the `dateFormat` function in order to [display](#) a value. See the correct syntax in the examples below.

Brackets around the `date_time` parameter indicate that it is optional.

Syntax:

```
dateEnd(interval_string[, date_time])
dateFormat(dateEnd(interval_string[, date_time]))
```

Parameter	Description/Format
interval_string	<p>Common "interval_string" parameters include:</p> <p>"year"; "month"; "week"; "hour"; "y" (which is the "day of the year" and not the year)</p> <p>For a complete list of supported "interval_string" parameter formats, refer to the topic, Using the interval_string parameter.</p>
date_time	<p><i>Optional</i></p> <p>If specified, the date must be the result of another time function.</p> <p><u>Valid Example:</u> <code>dateEnd("day",dateValue("10-01-00"))</code></p> <p><u>Invalid Example:</u> <code>dateEnd("day","10-01-00")</code></p> <p>If the "date_time" parameter is <u>not</u> specified, then the <u>current</u> date is used (implied "now" function).</p>

Example 1: *Given:* now()=6-Nov-00 9:30:15 PST; the day of the week is Monday.

Expression: `dateEnd("hour")`

Returns a **date**: "6-Nov-00 09:59:59"

Expression: `dateFormat(dateEnd("day"))`

Returns a **string**: "6-Nov-00 23:59:59"

Expression: `dateEnd("week")`

Returns a **date**: "11-Nov-00 23:59:59"

Expression: `dateFormat(dateEnd("month"))`

Returns a **string**: "30-Nov-00 23:59:59"

Example 2: *Given:* now()=6-Nov-00 9:30:15

The server is in MST time zone and the dateEnd function is executed by the client in the PST time zone.

Expression: `dateAdd(dateOffset(),dateEnd("day"))`

Returns a **date**: "6-Nov-00 22:59:59"

The 1 hour difference between the two time zones is reflected in the result.

Example 3: *Given:* now()=6-Nov-00 8:30:15

The server is in MST time zone and the dateEnd function is executed by the client in the MST time zone.

Expression: `dateFormat(dateAdd(dateOffset(),dateEnd("day")))`

Returns a **string**: "6-Nov-00 23:59:59"

The 1 hour difference between the two time zones is reflected in the result.

dateFormat with DateTime

Converts a date or time interval to a **string**.



Brackets around the **format_string** parameter indicate that it is optional.

Syntax: `dateFormat(date_time[, format_string])`

Parameter	Description/Format
date_time	The "date_time" must be the result of another time function. <u>Valid Example:</u> <code>dateFormat(dateValue("10-01-00"))</code>
format_string	Common "format_string" parameters include: "yyyy" for year (for example, 1963); "MMMM" for month (for example, July); "hh" for hour in am/pm (for example, 12:00); "zzzz" for time zone (for example, Pacific Standard Time) For a complete list of supported "format_string" parameter formats, refer to the topic, Using the format_string parameter .

Comments: Identical to [format\(date_time\[, format_string\]\)](#) String function.

Example(s): Expression: `dateFormat(dateValue("10-01-00"))`
Returns a **string**: "1-Oct-00 00:00:00"

Expression: `dateFormat(dateValue("10-01-00"), "EEEE, MMMM, d yyyy hh:mm a zzzz")`
Returns a **string**: "Sunday, October, 1 2000 12:00 AM Pacific Daylight Time"

dateFormat with String

Converts the current date/time to a **string**.

Syntax: `dateFormat(format_string)`

Parameter	Description/Format
format_string	Common "format_string" parameters include: "yyyy" for year (for example, 1963); "MMMM" for month (for example, July); "hh" for hour in am/pm (for example, 12:00); "zzzz" for time zone (for example, Pacific Standard Time) For a complete list of supported "format_string" parameter formats, refer to the topic, Using the format_string parameter .

Comments: Identical to [format\(format_string\)](#) String function.

Example(s): *Given:* now()=6-Nov-00 10:15:15 PST
 Expression: `dateFormat ("dd MMMM yy HH'h'mm")`
 Returns a **string**: "06 November 00 10h15"
 Expression: `dateFormat ("M/d/y HH:mm:ss.SSSS")`
 Returns a **string**: "11/6/00 10:15:15.892"

dateInterval

Converts a number (of hours, minutes, days, and so forth) to a **time interval** based on a specific date. Returns a time interval for the number of time units of the specified interval type specified by "interval_string" (for example, "day", "hours") from the specified date/time.



Because the XHQ system does not display dates unless it is a [string](#), this function requires the use of the `dateFormat` function in order to [display](#) a value. See the correct syntax in the examples below.

Brackets around the `date_time` parameter indicate that it is optional.

Syntax: `dateInterval(interval_string[, real[, date_time]])`
`dateFormat(dateInterval(interval_string[, real[, date_time]]))`

Parameter	Description/Format
interval_string	Common "interval_string" parameters include: "year"; "month"; "week"; "hour"; "y" (which is the "day of the year" and not the year) For a complete list of supported "interval_string" parameter formats, refer to the topic, Using the interval_string parameter.
real	The number of time units.
date_time	If specified, the date must be the result of another time function. <u>Valid Example:</u> <code>dateInterval("day",3,dateValue("10-01-00"))</code> <u>Invalid Example:</u> <code>dateInterval("day",3,"10-01-00")</code> If the "date_time" parameter is <u>not</u> specified, then the <u>current</u> date is used (implied "now" function).

Comments: In general, this is identical to omitting the date; exceptions include "1 year" for a leap year and "1 day" for the day of DST change. Most time intervals do not depend on a specific date.

Example(s): *Given:* now()=6-Nov-00 10:15:15 PST
 Expression: `dateInterval ("day", 3)`
 Returns a **time interval**: "3 0:00:00"
 Expression: `dateInterval ("day", 3, dateValue ("10-27-01"))`
 Returns a **time interval**: "3 1:00:00"
 In the United States, DST occurred on 10-28-01. Therefore, the extra hour is indicated.
 Expression: `dateFormat(dateInterval ("month", 3))`

Returns a **string**: "92 0:00:00"

dateOffset with DateTime

Returns the difference in GMT offsets on the particular date between the XHQ Server and the system (client or server) executing the expression. In other words, it returns the time interval value of the difference between given date, "date_time," (and optionally, in the specified time zone, "timezone") and the server time zone.



Because the XHQ system does not display dates unless it is a string, this function requires the use of the `dateFormat` function in order to display a value. See the correct syntax in the examples below.

Brackets around the `string` time zone parameter indicate that it is optional.

Syntax:

```
dateOffset (date_time[, timezone])
dateFormat (dateOffset (date_time[, timezone]))
```

Parameter	Description/Format
date_time	The given date/time.
timezone	The time zone is represented by a three-character abbreviation (for example, "PDT").

Comments: This value will always be zero if run on the server, unless the specified date is on the other side of the DST change.

Example(s): [For more information on time zones, refer to the topic, Time Zones and XHQ.](#)

Given: The server is located in the Pacific Daylight Time (PST) time zone.

Expression: `dateOffset ()`

Returns a **time interval**: "0:00:00"

Since the value is zero, the expression was either run on the server or on a client in the same time zone as the server.

Expression: `dateFormat(dateOffset ("MST"))`

Returns a **string**: "-1:00:00"

This is the result if the expression was run from a client located in the MST time zone and the server is located in the PST time zone.

dateOffset with String

Returns the difference in GMT offsets on the particular date between the XHQ Server and the system (client or server) executing the expression.



Because the XHQ system does not display dates unless it is a string, this function requires the use of the `dateFormat` function in order to display a value. See the correct syntax in the examples below.

Brackets around the `string` time zone parameter indicate that it is optional.

Syntax:

```
dateOffset ([string])
```

```
dateFormat(dateOffset([string]))
```

Parameter	Description/Format
string	The time zone is represented by a three-character abbreviation (for example, "PDT").
<i>Comments:</i>	This value will always be zero if run on the server, unless the specified date is on the other side of the DST change.
<i>Example(s):</i>	<p>For more information on time zones, refer to the topic, Time Zones and XHQ.</p> <p>Given: The server is located in the Pacific Daylight Time (PST) time zone.</p> <p>Expression: <code>dateOffset()</code> Returns a time interval: "0:00:00"</p> <p>Since the value is zero, the expression was either run on the server or on a client in the same time zone as the server.</p> <p>Expression: <code>dateFormat(dateOffset("MST"))</code> Returns a string: "-1:00:00"</p> <p>This is the result if the expression was run from a client located in the MST time zone and the server is located in the PST time zone.</p>

datePart

Returns the part of a date (month, day, hour, and so forth) as an **integer**.



Brackets indicate that it is optional.

Syntax: `datePart(interval_string[, date_time[, time_interval]])`

Parameter	Description/Format
interval_string	<p>Common "interval_string" parameters include:</p> <p>"year"; "month"; "week"; "hour"; "y" (which is the "day of the year" and not the year)</p> <p>For a complete list of supported "interval_string" parameter formats, refer to the topic, Using the interval_string parameter.</p>
date_time	<p>If specified, the date must be the result of another time function.</p> <p><u>Valid Example:</u> <code>datePart("day",dateValue("10-15-00"))</code></p> <p><u>Invalid Example:</u> <code>datePart("day","10-15-00")</code></p> <p>If the "date_time" parameter is <u>not</u> specified, then the <u>current</u> date is used (implied "now" function).</p>
time_interval	The time interval.
<i>Example(s):</i>	<p>Given: <code>now()</code>=6-Nov-00 10:15:15 PST</p> <p>Expression: <code>datePart("day")</code> Returns an integer: "6"</p> <p>Expression: <code>datePart("y")</code></p>

Returns an **integer**: "311"

Note: In this example, "y" represents the day of the year and NOT the year.

Expression: `datePart("year")`

Returns an **integer**: "2000"

Expression: `datePart("day", DateValue("10-31-01"))`

Returns an **integer**: "31"

dateRound

Rounds the specific **date/time** or **time interval** to the nearest interval (month, day, hour, and so forth).



Because the XHQ system does not display dates unless it is a string, this function requires the use of the `dateFormat` function in order to display a value. See the correct syntax in the examples below.

Brackets around the `date_time` parameter indicate that it is optional.

Syntax:

```
dateRound(interval_string[, date_time])
dateFormat(dateRound(interval_string[, date_time]))
```

Parameter	Description/Format
interval_string	<p>Common "interval_string" parameters include:</p> <p>"year"; "month"; "week"; "hour"; "y" (which is the "day of the year" and not the year)</p> <p>For a complete list of supported "interval_string" parameter formats, refer to the topic, Using the interval_string parameter.</p>
date_time	<p><i>Optional</i></p> <p>If specified, the date must be the result of another time function.</p> <p><u>Valid Example:</u></p> <pre>dateRound("day", dateValue("10-15-00"))</pre> <p><u>Invalid Example:</u></p> <pre>dateRound("day", "10-15-00")</pre> <p>If the "date_time" parameter is <u>not</u> specified, then the <u>current</u> date is used (implied "now" function).</p>

Example(s):

Given: `now()`=6-Nov-00 11:05:15 PST

Expression: `dateRound("month", DateValue("10-14-00"))`

Returns a **date**: "1-Oct-00 00:00:00"

Expression: `dateFormat(dateRound("month", DateValue("10-30-00")))`

Returns a **string**: "1-Nov-00 00:00:00"

Expression: `dateRound("week")`

Returns a **date**: "5-Nov-00 00:00:00"

In this case, the week started on Sunday, November 5.

Expression: `dateFormat(dateRound("hour"))`

Returns a **string**: "6-Nov-00 11:00:00"

dateSerial

Returns a **date** value with the time portion set to zero.



Because the XHQ system does not display dates unless it is a [string](#), this function requires the use of the `dateFormat` function in order to [display](#) a value. See the correct syntax in the examples below.

Syntax:

```
dateSerial(yyyy, month, day)
dateFormat(dateSerial(Integer yyyy, Integer m, Integer d))
```

Parameter	Description/Format
yyyy	The year , using four digits only. This parameter may be of type real, but will be truncated to an integer.
month	The month parameter may be of type real, but will be truncated to an integer.
day	The day parameter may be of type real, but will be truncated to an integer.

Example(s):

Expression: `dateSerial(1963, 7, 24)`
Returns a **date**: "24-Jul-63 00:00:00"

Expression:
`dateFormat(dateSerial(2000, 7, 24), "dd-MMM-yyyy")`
Returns a **string**: "24-Jul-2000"

Expression:
`dateFormat(dateSerial(00, 7, 24), "dd-MMM-yyyy")`
Returns a **string**: "24-Jul-0001"
Note the year returned. "0 A.D." is not a valid year; therefore, it incremented to "1 A.D."

dateStart

Returns the **date/time** for the beginning of the specified period.



Because the XHQ system does not display dates unless it is a [string](#), this function requires the use of the `dateFormat` function in order to [display](#) a value. See the correct syntax in the examples below.

Brackets around the `date_time` parameter indicate that it is optional.

Syntax:

```
dateStart(interval_string[, date_time])
dateFormat(dateStart(interval_string[, date_time]))
```

Parameter	Description/Format
interval_string	Common "interval_string" parameters include: "year"; "month"; "week"; "hour"; "y" (which is the "day of the year" and not the year) For a complete list of supported "interval_string" parameter formats, refer to the topic, Using the interval_string parameter.
date_time	<i>Optional</i>

Parameter	Description/Format
	<p>If specified, the date must be the result of another time function.</p> <p><u>Valid Example:</u> <code>dateStart("month",dateValue("10-15-00"))</code></p> <p><u>Invalid Example:</u> <code>dateStart("month","10-15-00")</code></p> <p>If the "date_time" parameter is <u>not</u> specified, then the <u>current</u> date is used (implied "now" function).</p>

Example(s): *Given:* now()=6-Nov-00 11:05:15 PST

The current date is "November 6, 2000" and the current time is "11:05:15 AM".

Expression: `dateStart ("week")`

Returns a **date**: "5-Nov-00 00:00:00"

In this case, the week started on Sunday, November 5.

Expression: `dateFormat(dateStart ("hour"))`

Returns a **string**: "6-Nov-00 11:00:00"

dateSub with String

Subtracts the given integer number (of hours, minutes, days, and so forth) to a given date.



Because the XHQ system does not display dates unless it is a string, this function requires the use of the `dateFormat` function in order to display a value. See the correct syntax in the examples below.

Brackets around the `date_time` parameter indicate that it is optional.

Syntax: `dateSub(interval_string, number[, date_time])`
`dateFormat(dateSub(interval_string, number[, date_time]))`

Parameter	Description/Format
interval_string	<p>Common "interval_string" parameters include:</p> <p>"year"; "month"; "week"; "hour"; "y" (which is the "day of the year" and not the year)</p> <p>For a complete list of supported "interval_string" parameter formats, refer to the topic, Using the interval_string parameter.</p>
number	<p>The "number" parameter may be of type real, but will be truncated to an integer. For example, using a "1" , or a "1.0", or a "1.25", or a "1.8" all result in the same number (they are all equivalent), and that number is "1".</p> <p>To add, you can use a negative number parameter or use the dateAdd function.</p>
date_time	<p><i>Optional</i></p> <p>If specified, the date must be the result of another time function. If the date is <u>not</u> specified, then the integer number is subtracted from the current date.</p> <p><u>Valid Example:</u> <code>dateSub("day",3,dateValue("10-27-16"))</code></p>

Parameter	Description/Format
	<u>Invalid Example:</u> dateSub("day",3,"10-27-16")

Example(s): *Given:* Current date, now()=15-Dec-16 14:20:15 PST
 Expression: dateSub("day",5)
 Returns a **date**: "10-Dec-16 14:20:15"
 Subtracts 5 days from the current date.

Expression: dateSub("day",5,DateValue("07-24-16"))
 Returns a **date**: "19-Jul-16 00:00:00"
 Subtracts 5 days from the date specified.

Expression: dateFormat(dateSub("week",2))
 Returns a **string**: "1-Dec-16 14:20:15"
 Subtracts 2 weeks from the current date.

Expression: dateSub("year",5.932)
 Returns a **date**: "15-Dec-11 14:20:15"
 The real number "5.932" is truncated to "5". Therefore, it subtracts 5 years from the current date.

Expression: dateFormat(dateSub("month",-6.7))
 Returns a **string**: "15-Jun-17 14:20:15"
 The real number "-6.7" is truncated to "-6". Therefore, it adds 6 months from the current date.

dateSub with TimeInterval

Subtracts the time interval from the date/time.



Because the XHQ system does not display dates unless it is a [string](#), this function requires the use of the [dateFormat](#) function in order to [display](#) a value. See the correct syntax in the examples below.

Brackets around the `date_time` parameter indicate that it is optional.

Syntax: dateSub(time_interval[, date_time])
 dateFormat(dateSub(time_interval[, date_time]))

Parameter	Description/Format
time_interval	The time interval must be the result of another time function. <u>Valid Example:</u> dateSub(dateInterval("4 3:05:01")) <u>Invalid Example:</u> dateSub("4 3:05:01")
date_time	<i>Optional</i> If specified, the date must be the result of another time function. If the date is <u>not</u> specified, then time interval is subtracted from the <u>current</u> date. <u>Valid Example:</u> dateSub(dateInterval("4 3:05:01"), dateValue("10-01-16"))

Parameter	Description/Format
	<u>Invalid Example:</u> dateSub(dateInterval("4 3:05:01"), "10-01-16")

Example(s): *Given:* Current date, now()=15-Dec-16 15:30:15 PST
Expression: dateSub(dateInterval("4 3:15:15"))
Returns a **date**: "11-Dec-16 12:15:00"
Subtracts 4 days from the current date, since the `date` parameter was not specified, and 3 hours, 15 minutes, and 15 seconds from the current time.
Expression: dateFormat(dateSub(dateInterval("4 3:00:00"), dateValue("10-01-16")))
Returns a **string**: "26-Sep-16 09:00:00"
Subtracts 4 days from the date indicated by the `dateValue` function, and 3 hours from the midnight hour.

dateValue

Converts a string to a **date/time** value.



Because the XHQ system does not display dates unless it is a string, this function requires the use of the `dateFormat` function in order to display a value. See the correct syntax in the examples below.

Brackets around the `format_mask`, `country`, and `language` parameters indicate that they are optional.

Syntax: `dateValue(date_string[, format_mask[, country[, language]])`
 `dateFormat(dateValue(date_string[, format_mask[, country[, language]])`

Parameter	Description/Format
date_string	Omitted parts of the time will default to zero (relative to local time) and omitted parts of the date defaults to the current period (for example, if you omit the year then the default is the current year).
format_mask	<i>Optional</i> Common "format_mask" parameters include: "yyyy" for year (for example, 1963); "MMMM" for month (for example, July); "hh" for hour in am/pm (for example, 12:00); "zzzz" for time zone (for example, Pacific Standard Time)
country	<i>Optional</i> This is the two-letter string corresponding to the ISO code for country. Note the following country examples.

Parameter	Description/Format	
	Country	ISO Code
	United States	US
	France	FR
	Germany	DE
	Italy	IT
	Spain	ES
language	<i>Optional</i> This is the three-letter string corresponding to the ISO code for language. Note the following country examples.	
	Language	ISO Code
	English	en
	French	fr
	German	de
	Spanish	es

Example(s):

Expression: `dateValue("7-24-63 15:02:31.349")`
 Returns a **date**: "24-Jul-63 15:02:31"

Expression: `dateFormat(dateValue("7-24-63 2:57:31.349 PM"))`
 Returns a **string**: "24-Jul-63 14:57:31"

Expression: `dateValue("24-Jul-63 2:57:31.349 PM", "dd-MMM-yy h:mm:ss.SSS a")`
 Returns a **date**: "24-Jul-63 14:57:31"

Expression: `dateFormat(dateValue("24-7-63", "dd-M-yy"))`
 Returns a **string**: "24-Jul-63 00:00:00"

Expression: `dateValue("24-Enero-63", "dd-MMMM-yy", "ES", "es")`
 Returns a **date**: "24-Jan-63 00:00:00"

Note the country ("ES" for Spain) and the language code ("es" for Spanish). The term "Enero" is Spanish for "January".

intervalPart

Returns the integer value of the specified part of the interval, "string", of the time interval.

Syntax: `intervalPart(string, time_interval)`

Parameter	Description/Format
string	Specified part of the interval.
time_interval	The given time interval.

now

Returns the **current date/time**.



Because the XHQ system does not display dates unless it is a [string](#), this function requires the use of the `dateFormat` function in order to [display](#) a value. See the correct syntax in the examples below.

Syntax: `now()`
 `dateFormat(now())`

Parameter	Description/Format
N/A	N/A

Comments: When converted to a string or displayed, it is formatted using the default date/time format. This function depends on the current date and time.

Example(s): *Given:* The current date is "November 6, 2005" and the current time is "3:05:15 PM".
Expression: `now()`
Returns a **date**: "6-Nov-05 15:05:15"
Expression: `dateFormat(now())`
Returns a **string**: "6-Nov-05 15:05:15"

timeSerial

Returns a **time interval** value with the date portion set to zero.



Because the XHQ system does not display dates unless it is a [string](#), this function requires the use of the `dateFormat` function in order to [display](#) a value. See the correct syntax in the examples below.

Syntax: `timeSerial(hours, minutes, seconds)`
 `dateFormat(timeSerial(hours, minutes, seconds))`

Parameter	Description/Format
hours	Number of hours . This parameter may be of type real, but will be truncated to an integer.
minutes	Number of minutes . This parameter may be of type real, but will be truncated to an integer.
seconds	Number of seconds . This parameter may be of type real, but will be truncated to an integer.

Comments: Adding the results of a `dateSerial` to a `timeSerial` should produce the associated date/time. See example below.

Example(s): Expression: `timeSerial(73,15,27)`
Returns a **time interval**: "3 1:15:27"
Expression: `dateFormat(timeSerial(-73,-15,-27))`

Returns a **string**: "-3 1:15:27"

Expression: `timeSerial(-73,15,27.4)`

Returns a **time interval**: "-3 0:44:33"

The seconds parameter was truncated to an integer.

Expression: `timeSerial(0,90,-5)`

Returns a **time interval**: "1:29:55"

Expression: `dateFormat(dateAdd(timeSerial(14,57,23),dateSerial(1963,7,24)))`

Returns a **string**: "24-Jul-1963 14:57:23"

Adding the results of a "dateSerial" function to a "timeSerial" function produces the associated date/time.

Units of Measure Functions

Use the following functions for Units of Measure (UoM) conversion in **server-side expressions**.

Quick Reference - UoM Functions

Function	Description	Return Type
<code>baseUnits</code> (String unit)	Returns the base unit of measure for the one given. For example, "kg" would return "g".	String
<code>convertUnits</code> (AnyNumeric value, String unit)	Converts the value from its implicit units of measure (if any) to the specified units of measure. Result is BAD if the value has no current units of measure associated with it.	Numeric
<code>convertUnits</code> (AnyNumeric value, String fromUoM, String toUoM)	Converts the value from the specified units of measure to the specified units of measure. The value is treated as a scalar value and implicit units of measure are ignored.	Numeric
<code>unitsCategory</code> (String unit)	Returns the category to which the units of measure belongs.	String

baseUnits

This function returns the base units defined for the given unit of measure.



Undefined units of measure result in a `BAD_NON_EXISTENT` quality.

Syntax: `baseUnits (String unit)`

Return value: String

Parameters	Description
unit	The unit of measure. Returns the base unit of measure for this given unit.

Example: `baseUnits ("g")`

Returns: "kg"

convertUnits

This function returns a converted numeric value based on the **value** and **toUoM** parameters. The **fromUoM** parameter can be used to override the units of the input value (which is the value parameter).



Units that are undefined or are incompatible results in a `BAD_UOM_CONVERSION_FAILURE` quality.

Syntax: `convertUnits(AnyNumeric value[, String unit][, String fromUoM, String toUoM])`

Return value: Numeric

Parameters	Description
value	This is the value to be converted.
unit	This is the unit converted to.
fromUoM	This overrides the units of the given value.
toUoM	The units to which the given value is converted. Important: This must be compatible to the units of the given value or, if specified, the "fromUoM".

Example: `convertUnits(10, "kg", "g")`

Returns: 10000 with a unit set to "g"

unitsCategory

This function returns the category associated with the given unit of measure.



Undefined units of measure result in a `BAD_NON_EXISTENT` quality.

Syntax: `unitsCategory(String unit)`

Return value: String

Parameters	Description
unit	The unit of measure. Returns a string value.

Example: `unitsCategory("g")`

Returns: "Mass"

Late-Binding Reference Functions

These functions allow late-binding references to be used in expressions. A value is “late bound” when the reference is resolved at run-time. The intent is to allow a name or path of a member to be constructed through calculation and the referenced to find a value after the name is finally known.

Quick Reference - Late-Binding Reference Functions

Function	Description	Return Type
referenceToBoolean (String path)	Returns the Boolean value of the member whose name/path is given. Target type must be a Boolean member.	Boolean
referenceToDateTime (String path)	Returns the DateTime value of the member whose name/path is given. Target type must be a DateTime member.	DateTime
referenceToDecimal (String path)	Returns the Decimal value of the member whose name/path is given. Target type must be a Decimal member.	Decimal
referenceToDouble (String path)	Returns the Double value of the member whose name/path is given. Target type must be a Double member.	Double
referenceToInteger (String path)	Returns the Integer value of the member whose name/path is given. Target type must be an Integer member.	Integer
referenceToInterval (String path)	Returns the Interval value of the member whose name/path is given. Target type must be an Interval member.	Interval
referenceToLong(String path)	Returns the Long value of the member whose name/path is given. Target type must be a Long member.	Long
referenceToReal(String path)	Returns the Real value of the member whose name/path is given. Target type must be a Real member.	Real
referenceToString (String path)	Returns the String value of the member whose name/path is given. Target type must be a String member.	String
referenceToText(String path)	Returns the Text value of the member whose name/path is given. Target type must be a Text member.	Text

About ReferenceTo

The following functions enable support for [late-binding](#) tag references in a pen expression:

- Syntax:*
- `ReferenceToBoolean(s)`
 - `ReferenceToInteger(s)`
 - `ReferenceToLong(s)`
 - `ReferenceToReal(s)`
 - `ReferenceToDouble(s)`
 - `ReferenceToDecimal(s)`
 - `ReferenceToDateTime(s)`
 - `ReferenceToInterval(s)`
 - `ReferenceToString(s)`
 - `ReferenceToText(s)`

Argument	Description
s	A STRING-type expression.

At design time, these **ReferenceTo** functions behave like the cast used by a static tag reference to indicate the data type of the reference. At runtime, the string-type expression (the argument) is resolved to an alias (for example, the tag name).

The following is an example of a pen expression with two late-binding tag references.

Example: `ReferenceToReal($EditBox1) + ReferenceToReal($EditBox2)`
 Where \$EditBox1 and \$EditBox2 are string-type control variables.

Late-Binding Reference Aliases

Because expressions can be quite complex in real-world applications, you can use the abbreviated versions (or aliases) of the **SubscribeTo** and **ReferenceTo** group of late-binding functions. For example, for **ReferenceTo**, simply indicate the **cast type** (r=real, s=string, i=integer, and so forth) followed by **Ref**.

Example: `rRef` is equivalent to `ReferenceToReal`

Late-Binding Reference Aliases

Function	Description	Return Type
<code>bRef(String path)</code>	Alias for <code>referenceToBoolean</code> .	Boolean
<code>dRef(String path)</code>	Alias for <code>referenceToDouble</code> .	Double
<code>iRef(String path)</code>	Alias for <code>referenceToInteger</code> .	Integer
<code>lRef(String path)</code>	Alias for <code>referenceToLong</code> .	Long
<code>mRef(String path)</code>	Alias for <code>referenceToDecimal</code> .	Decimal
<code>rRef(String path)</code>	Alias for <code>referenceToReal</code> .	Real
<code>sRef(String path)</code>	Alias for <code>referenceToString</code> .	String

Function	Description	Return Type
tRef(String path)	Alias for referenceToDateTime.	DateTime
vRef(String path)	Alias for referenceToInterval.	Interval
xRef(String path)	Alias for referenceToText.	Text



For a list of supported cast prefix types, go to the topic, [About Global Tag References](#), located in the XHQ Developer's Guide.

See also [Identifiers for Server side Expressions](#).

Late-Binding for constructed names to aliases or solution members

Both the client-side and the server-side expression subsystems support the late binding of names to solution members or aliases through **ReferenceTo** calls.

Examples:

```
ReferenceToReal("plant1." + sAssetName + ".pv")
ReferenceToInteger("area2." + sReactorName + ".id")
ReferenceToString("country1." + sState + ".capitalCity")
ReferenceToLong("unit1." + sTankName + ".tankId")
ReferenceToDouble("productionFacility." + sVesselName + ".temperature")
```



For more information, refer to the topic, [Late binding Expressions](#).

Using Reference Methods with Historical Aggregate Methods

Late-binding reference methods can be chained with Historical Aggregate methods. This obtains aggregate data for members/tags that are determined only when the expression is evaluated.

Example:

You can place the following expression on a member of a collection to get the hourly average of a tank level that is determined by other members on the same row of data.

```
rRef( TankName + ".LEVEL" ).avg( DateAdd("hour", -1), Now() )
```

Late-Binding Subscription Functions

These functions allow late-binding subscriptions to be used in expressions. A value is late bound when the reference is resolved at run-time. The intent is to allow a name or path of a member to be constructed through calculation and the referenced to find a value after the name is finally known.

Late-Binding Subscription Functions

Function	Description	Return Type
subscriptionToBoolean (String path)	Subscribes to and returns the Boolean value of the member whose name/path is given. Target type must be a Boolean member.	Boolean
subscriptionToDateTime (String path)	Subscribes to and returns the DateTime value of the member whose name/path is given. Target type must be a DateTime member.	DateTime
subscriptionToDecimal (String path)	Subscribes to and returns the Decimal value of the member whose name/path is given. Target type must be a Decimal member.	Decimal
subscriptionToDouble (String path)	Subscribes to and returns the Double value of the member whose name/path is given. Target type must be a Double member.	Double
subscriptionToInteger (String path)	Subscribes to and returns the Integer value of the member whose name/path is given. Target type must be an Integer member.	Integer
subscriptionToInterval (String path)	Subscribes to and returns the Interval value of the member whose name/path is given. Target type must be an Interval member.	Interval
subscriptionToLong (String path)	Subscribes to and returns the Long value of the member whose name/path is given. Target type must be a Long member.	Long
subscriptionToReal (String path)	Subscribes to and returns the Real value of the member whose name/path is given. Target type must be a Real member.	Real
subscriptionToString (String path)	Subscribes to and returns the String value of the member whose name/path is given. Target type must be a String member.	String
subscriptionToText (String path)	Subscribes to and returns the Text value of the member whose name/path is given. Target type must be a Text member.	Text

About SubscribeTo

The use of the browser client data retrieval function **getHistoricalData** to obtain real-time data associated with an alias through a historical call to the back end connector has prompted the use of the **SubscribeTo** functions:

- Syntax:*
- `SubscribeToBoolean(s)`
 - `SubscribeToInteger(s)`
 - `SubscribeToReal(s)`
 - `SubscribeToString(s)`

Argument	Description
s	A STRING representing an alias or an absolute path.

These functions can be used in any view context and obtain a value by subscribing to the point whose alias (or absolute path) is passed as an argument. Return values are the result of the subscription to the primitive member that was identified.



If the primitive member type is not compatible with the type (as implied by the name of the function), the function will attempt to coerce the value of the primitive member. If coercion fails, then the quality of the value generated by the function will be BAD.



The *late-binding expressions* mentioned in this guide should only be used in circumstances when an alias or full path to a primitive member cannot be known upfront. In other words, only use late-binding expressions in scenarios where part or the entirety of the alias name or the full path depends on values that can only be evaluated at runtime (such as values retrieve from a backend, client-side expressions, and other forms of data only available in runtime).

If the entire alias or full primitive member path can be determined upfront - and not just at runtime - then use other syntax, similar to the following examples, since these have significant performance advantages over the late-binding technique.

Example for Real data type:

```
r:'alias' or r:'fullPrimitiveMemberPath'
```

Example for String data type:

```
s:'alias' or s:'fullPrimitiveMemberPath'
```



For a list of supported cast prefix types, go to the topic, *About Global Tag References*, located in the XHQ Developer's Guide.

See also *Identifiers for Server side Expressions*.

Function	Description
<code>SubscribeToBoolean("MYALIAS")</code>	When used in an expression, this function will return a real-time subscription value based on the string "MYALIAS". If the alias is not legitimate, no real-time data will be obtained. If the alias is for a member of type component, no real-time data will be obtained, since you cannot subscribe to components. <code>SubscribeToBoolean</code> expects the result returned by the function to be of type BOOLEAN. If the alias type is incompatible with the result type returned by the function, then a default value is provided or the value may be converted/coerced. Using this function, a primitive member can be subscribed to from any view, as long as the alias or absolute path is known, and the subscription will act like any other (that is, the value will be updated whenever the connector is polled).
<code>SubscribeToInteger("MYALIAS")</code>	When used in an expression, this function will return a real-time subscription value based on the string "MYALIAS". If the alias is not legitimate, no real-time data will be obtained. If the alias is for a member of type component, no real-time data will be obtained, since you cannot subscribe to components. <code>SubscribeToInteger</code> expects the result returned by the function to be of type INTEGER. If the alias type is incompatible with the result type returned by the function, then a default value is provided or the value may be converted/coerced. Using this function, a primitive member can be subscribed to from any view, as long as the alias or absolute path is known, and the subscription will act

Function	Description
	like any other (that is, the value will be updated whenever the connector is polled).
<code>SubscribeToReal("MYALIAS")</code>	<p>When used in an expression, this function will return a real-time subscription value based on the string "MYALIAS". If the alias is not legitimate, no real-time data will be obtained. If the alias is for a member of type component, no real-time data will be obtained, since you cannot subscribe to components. <code>SubscribeToReal</code> expects the result returned by the function to be of type REAL. If the alias type is incompatible with the result type returned by the function, then a default value is provided or the value may be converted/coerced. Using this function, a primitive member can be subscribed to from any view, as long as the alias or absolute path is known, and the subscription will act like any other (that is, the value will be updated whenever the connector is polled).</p> <p>This function is especially useful when constructing tag browsers where the users want to see a tag's real-time value alongside its metadata.</p>
<code>SubscribeToString("MYALIAS")</code>	<p>When used in an expression, this function will return a real-time subscription value based on the string "MYALIAS". If the alias is not legitimate, no real-time data will be obtained. If the alias is for a member of type component, no real-time data will be obtained, since you cannot subscribe to components. <code>SubscribeToString</code> expects the result returned by the function to be of type STRING. If the alias type is incompatible with the result type returned by the function, then a default value is provided or the value may be converted/coerced. Using this function, a primitive member can be subscribed to from any view, as long as the alias or absolute path is known, and the subscription will act like any other (that is, the value will be updated whenever the connector is polled).</p>

Late-binding for constructed names to aliases or solution members

Both the client-side and the server-side expression subsystems support the late binding of names to solution members or aliases through **SubscribeTo** calls.

Examples:

```

SubscribeToReal("plant1." + sAssetName + ".pv")
SubscribeToInteger("area2." + sReactorName + ".id")
SubscribeToString("country1." + sState + ".capitalCity")
SubscribeToLong("unit1." + sTankName + ".tankId")
SubscribeToDouble("productionFacility." + sVesselName + ".temperature")

```



For more information, refer to the topic, [Late binding Expressions](#).

Abbreviated Naming

Because expressions can be quite complex in real-world applications, you can use the abbreviated versions of the **SubscribeTo** and **ReferenceTo** group of late-binding functions. For example, for `ReferenceTo`, simply indicate the **cast type** (r=real, s=string, i=integer, and so forth) followed by **Ref**.

Example: rRef is equivalent to `ReferenceToReal`



For a list of supported cast prefix types, go to the topic, [About Global Tag References](#), located in the XHQ Developer's Guide.

See also [Identifiers for Server side Expressions](#).

Miscellaneous Functions

Section Contents

Historical Mode Function	144
Log Functions	144
Cache Collections and NVARCHAR2	144
Action Link Functions	145
Java Applet Scripting Functions	146

Historical Mode Function

The following function can be used within an expression when the user needs to determine the state of historical viewing for the given member.

isInHistoricalMode

Returns TRUE when the system is in historical mode and history is available for the member specified in the argument; else, it returns FALSE.

Syntax: `isInHistoricalMode (memberName)`

Example 1: To set the color of a value item based on the historical mode of a member, you could use the **Text Color** animation and use **isInHistoricalMode (memberName)** as the **Evaluating** expression. Based on the return value, you could set the color using **Value Breakpoint**.

Example 2: You could use a **Show** animation on a rectangle that is filled with a color to indicate historical data. This rectangle can then be used as the background of a value item. If **isInHistoricalMode (memberName)** is TRUE, the rectangle appears; if FALSE, it does not.

Log Functions

The following functions write messages to the log.

logMessage

This function writes the message supplied to the Solution Viewer's log file.

Syntax: `logMessage (message)`

logMessageTS

This function writes the message supplied to the Solution Viewer's log file prefixed with a timestamp (TS) in the following format:

MM/DD/YY HH:MM.SS.SSS message

Syntax: `logMessageTS (message)`

Example: *Given:* `logMessageTS ('Downloading information')`

Results in: 06/03/08 14:03:22:567 Downloading information

Cache Collections and NVARCHAR2

Functions used in cache collections must return NVARCHAR2. For example, this is done by wrapping them in `TO_NCHAR (function)`.

Action Link Functions

In addition to the URL and View link types on the Links tab, the Action type allows you to perform any action specified by a function expression. The function expression is entered in the Function text box.

The function expression(s) you use should have no type associated with it. Separate multiple expressions with a comma.



For more information on the use of commas in multiple expressions, refer to the topic, [Comma Operator](#).

The following are action function examples:

- `showView(objPath, viewName[, mode, autoClose, x, y])`
Which displays a view given the object path and view name, and optionally the mode.
- `navigate(mode)`
Which displays the view specified by the given mode.



For more information on how to set an action link pop-up, go to the topic, [Setting Action Link Pop-ups](#), located in the XHQ Developer's Guide.

Java Applet Scripting Functions

XHQ supports the following functions. These functions allow script invocation from a JavaScript environment.

Quick Reference - General Functions

Function Syntax	Description
<code>createIndxApplet (String s1, String s2, String s3, String s4, String s5)</code>	Dynamically creates the applet tag for the XHQ Solution Viewer (browser client).
<code>GetClientTimeAsDate ()</code>	Returns a date that represents the current time (historical or server) using the proper time zone and formatting.
<code>GetClientTimeAsLong ()</code>	Returns a long that represents the current time (historical or server) using the proper time zone and formatting.
<code>GetClientTimeAsString ()</code>	Returns a string that represents the current time (historical or server) using the proper time zone and formatting.
<code>getCollectionData (objPath, members, criteria, fetchSize)</code>	This function returns the Java object, <code>CollectionData</code> , which contains the result data for this function.
<code>getHistoricalData</code>	Returns the Java object for <code>HistoricalData</code> .
<code>getHistoricalDataEx</code>	Returns the Java object for <code>HistoricalData</code> using the XHQ Trend Table only.
<code>getMemberInfo</code>	Returns the Java object for <code>MemberInfo</code> .
<code>getQualityUtil</code>	Returns the Java object, <code>QualityUtil</code> , which is a shared instance. Use this object to retrieve detailed quality status or to easily decipher the quality returned by the <code>HistoricalData</code> and <code>CollectionData</code> objects.
<code>getViewInfo</code>	Returns the Java object, <code>ViewInfo</code> , containing the meta data for all views associated with the identified component instance.
<code>onAfterShowView (objPath, viewName)</code>	As the currently displayed view changes in the applet, this function will be called with the object path and view name of the newly shown view after it is loaded.
<code>onBeforeShowView (objPath, viewName)</code>	As the currently displayed view changes in the applet, this function will be called with the object path and view name of the newly shown view prior to that view being loaded. This will allow JavaScript variables that the view may depend on to be set.
<code>OnHistMode (histMode, histTime, jogIncrement)</code>	This is a callback function from the XHQ Applet to JavaScript.
<code>onShowView (objPath, viewName, type)</code>	As the currently displayed view changes in the applet, this function will be called with the object path, view name, and view type of the newly shown view.
<code>SetHistMode (histMode, histTime, jogIncrement)</code>	This is a callback function from the XHQ Applet to JavaScript.
<code>setTrendPaths</code>	Sets the scaling for all pens to auto; sets "display" to true for all pens;

Function Syntax	Description
	sets trend to "all scales" mode; does not change the symbol setting.
<code>setTrendScooter</code>	Replaces the current scooter position for the XHQ Trend Viewer.
<code>setTrendTime</code>	Replaces the current time window for the XHQ Trend Viewer.
<code>showView</code> (objPath, viewfinder [, mode, filter, x, y])	This method changes the currently displayed view in the XHQ Applet.

createIndxApplet

Dynamically creates the applet tag for the XHQ Solution Viewer (browser client).



Brackets [] indicate that the argument is optional.

Syntax 1:

If use_router is "no":

```
createIndxApplet(signedApplet, preferences, use_router, sHost,
mHost)
```

Example:

```
createIndxApplet("yes", location.search, "no", "localhost",
"localhost")
```

Syntax 2:

If use_router is "yes" OR " " (empty):

```
createIndxApplet(signedApplet, preferences, use_router, rsHost,
solutionName)
```

Examples:

```
createIndxApplet("no", location.search, " ", " ", " ")
```

In this example, use_router is empty, rsHost defaults to the web server, and solutionName defaults to the first solution for the enterprise server.

```
createIndxApplet("yes", location.search, "yes", "Router1",
"Enterprise")
```

In this example, use_router is "yes", rsHost is "Router1", and solutionName is "Enterprise".



The default "localhost" indicates that the web server, the XHQ Enterprise Server and the XHQ Solution Server are located on the same machine.

Argument	Description
preferences	<p><i>Optional</i> This is used to set the solutionHome, splitterPosition, etc. The default is blank.</p> <p>For more information, refer to the topic, Setting Browser Preferences, located in the XHQ Solution Viewer User's Guide.</p> <p><i>Example:</i></p> <pre>createIndxApplet("yes", "solutionHome=Acme.WestDiv.Sales~Forecasts &splitterPosition=30&showSolutionHome=yes &showTree=yes&showViewScrollBars=yes" "no", "localhost", "localhost")</pre>
use_router	<p>Specifies whether or not to use the router.</p> <p>If "yes" (or if empty: " "), then define rsHost and solutionName.</p> <p>If "no", then define sHost and mHost.</p>

Argument	Description
sHost	The name of the solution server host.
mHost	Name of enterprise server host.
rsHost	<i>Optional</i> This is the hostname for the router. Defaults to the Web Server.
solutionName	<i>Optional</i> This is the name of the solution. Defaults to the first solution for the XHQ Enterprise Server.

getClientTimeAsDate

Returns a date that represents the current time (historical or server) using the proper time zone and formatting.

Syntax: `getClientTimeAsDate()`

getClientTimeAsLong

Returns a long that represents the current time (historical or server) using the proper time zone and formatting.

Syntax: `getClientTimeAsLong()`

getClientTimeAsString

Returns a string that represents the current time (historical or server) using the proper time zone and formatting.

Syntax: `getClientTimeAsString()`

getCollectionData

This function returns the Java object, `CollectionData`, which contains the result data for this function.

Syntax: `getCollectionData(objPath, members, criteria, fetchSize)`

Argument	Description
objPath	The object path for the collection component instance.
members	A list containing column of the collection to retrieve. Members are separated by commas. <i>Example:</i> "member1, member2, member3"
criteria	The criteria to filter collection. Important: Currently not used.
fetchSize	The fetch size for the result set. Note: If the value is < 50, it defaults to 50.

Example:

```
var sourceObject = document.XHQ_Solution_Viewer.getCollectionData
(objPath, null, null, fetchSize);
```



This function works the same in both the Applet and the HTML5 XHQ Solution Viewer.

```
eval ("
  var sourceObject = document.XHQ_Solution_Viewer.getCollectionData (
    '::root.MyCollection', null, null, fetchSize);
  var numRecords = sourceObject.getCount();
  //....code here....
")
```



See also, [Collection Data Functions](#)

getHistoricalData() and getHistoricalDataEx()

The **getHistoricalData** function returns the Java object for HistoricalData. This function does not support expressions. The **getHistoricalDataEx** function returns the Java object for HistoricalData using the XHQ Trend Table only. This function does support expressions.



WARNING

Note the following restriction for using **getHistoricalDataEx()**. An expression in a view is **eval ("jsGetHistory ('xyz')")**, where **jsGetHistory()** is a JavaScript function that calls the applet's **getHistoricalDataEx()** function to retrieve history data for the tag 'xyz'. As a result, the web browser will hang when the view is loaded. Therefore, the function **getHistoricalData()** should be used for this scenario to prevent the applet from hanging.

Syntax:

```
getHistoricalData(objPath, deltaTorT1, t2, historyMode, numPoints)
getHistoricalDataEx(objPath, deltaTorT1, t2, historyMode, numPoints)
```

Argument	Description
objPath	For getHistoricalData The fully qualified path to a primitive member or an alias. For getHistoricalDataEx The fully qualified path to a primitive member, an alias, or a trend expression.
deltaTorT1	A string containing the starting date, or the delta from the ending date, in milliseconds.
t2	A string containing the ending date of the historical data set in milliseconds. Note: If none is specified, then the current time is used.
historyMode	The type of historical data in the set: "fits" or "actuals". Note: The default is "fits".
numPoints	The maximum number of points to return. Note: Defaults to 1000 if 0 or less is specified.

Example:

```
var sourceObject = document.XHQ_Solution_Viewer.getHistoricalData
(objPath, t1.getTime(), t2.getTime(), "actuals", numPoints);
```



See also, [Historical Data Functions](#).

getMemberInfo

This function returns the Java object for MemberInfo.

Syntax: `getMemberInfo (objPath)`

Argument	Description
objPath	The object path of members.

Example: `var sourceObject = document.XHQ_Solution_Viewer.getMemberInfo (objPath) ;`



See also, [MemberInfo Functions](#).

getQualityUtil

This function returns the Java object, QualityUtil, which is a shared instance. Use this object to retrieve detailed quality status or to easily decipher the quality returned by the HistoricalData and CollectionData objects.

Syntax 1: `getQualityUtil()`

In this case, the quality is assumed to be UNCERTAIN.

Syntax 2: `getQualityUtil(quality)`



See also, [QualityUtil Functions](#).

getViewInfo()

This function returns the Java object, ViewInfo, containing the meta data for all views associated with the identified component instance.

Syntax: `getViewInfo (objPath)`

Argument	Description
objPath	The object path for a component instance.

Example: `var sourceObject = document.XHQ_Solution_Viewer.getViewInfo (objPath) ;`



See also, [ViewInfo Functions](#).

onAfterShowView

As the currently displayed view changes in the applet, this function will be called with the object path and view name of the newly shown view after it is loaded.

Syntax: `onAfterShowView(objPath, viewName)`

Example: `onAfterShowView('::Acme.WesternDivision.Sales', 'Summary')`



This is a *callback* function. If your application needs to use this function to create site-specific customizations, edit the function and add the appropriate JavaScript code. The default implementation of this function is located in the current `xhq.html` file (or a site-specific `.html` file).

The displayed view can change either by the navigating through the tree, drilling down into a view, or by calling `showView`.



Each argument is enclosed by **single** quotation marks.

Argument	Description
<code>objPath</code>	The fully qualified path of the object to view. The path must begin with two colon characters (::) and each member in the path must be separated by a period (.).
<code>viewName</code>	The name of the view of the object.

onBeforeShowView

As the currently displayed view changes in the applet, this function will be called with the object path and view name of the newly shown view prior to that view being loaded. This will allow JavaScript variables that the view may depend on to be set.

Syntax: `onBeforeShowView(objPath, viewName)`

Example: `onBeforeShowView('::Acme.WesternDivision.Sales', 'Forecasts')`



This is a *callback* function. If your application needs to use this function to create site-specific customizations, edit the function and add the appropriate JavaScript code. The default implementation of this function is located in the current `xhq.html` file (or a site-specific `.html` file).

The displayed view can change either by the navigating through the tree, drilling down into a view, or by calling `showView`.



Each argument is enclosed by **single** quotation marks.

Argument	Description
<code>objPath</code>	The fully qualified path of the object to view. The path must begin with two colon characters (::) and each member in the path must be separated by a period (.).
<code>viewName</code>	The name of the view of the object.

onHistMode

This is a callback function from the XHQ Applet to JavaScript.

Syntax: `onHistMode(histMode, histTime, jogIncrement)`



This function is listed in the current `xhq.html` file.

Argument	Description
histMode	The current state. <ul style="list-style-type: none"> • 0 – NOT in historical mode • 1 – In historical mode
histTime	The current reference time in milliseconds.
jogIncrement	The current jog increment in seconds.

onShowView

As the currently displayed view changes in the applet, this function will be called with the object path and view name of the newly shown view.

Syntax: `onShowView(objPath, viewName, type)`

Example: `onShowView('::Acme.WesternDivision.Sales', 'Forecasts', '2')`



This is a *callback* function. If your application needs to use this function to create site-specific customizations, edit the function and add the appropriate JavaScript code. The default implementation of this function is located in the current `xhq.html` file (or a site-specific `.html` file).

The displayed view can change either by the navigating through the tree, drilling down into a view, or by calling `showView`.



Each argument is enclosed by **single** quotation marks.

Argument	Description
objPath	The fully qualified path of the object to view. The path must begin with two colon characters (::) and each member in the path must be separated by a period (.).
viewName	The name of the view of the object.
type	1 = Flat view completed (top level, which contains all embedded views as well as containing view)

setHistMode

This is a callback function from the XHQ Applet to JavaScript.

Syntax: `setHistMode(histMode, histTime, jogIncrement)`



This function is listed in the current `xhq.js` file.

Argument	Description
histMode	The current state. <ul style="list-style-type: none"> • 0 – NOT in historical mode • 1 – In historical mode
histTime	The current reference time (JavaScript Date object).
jogIncrement	The current jog increment in seconds.

About Browser Client APIs

There are three APIs for the browser client applet (navigation bar) that can be called from JavaScript. Each takes into account the following factors:

- The global time zone setting;
- The global time format;
- And whether the browser client is running in historical mode when returning the time.

The property for the global time zone setting is **UseClientTimeZone**, which has a default value of TRUE.



Although the **UseClientTimeZone** property is currently not listed on the `globalsettings.properties` file, it has the default value of TRUE. To change the value to FALSE, you will have to edit the `globalsettings.properties` file and set **UseClientTimeZone=false**.

The property for the global time format is **DefaultTimeStampFormat**. The default value for the timestamp format is **MM/dd/yy HH:mm:ss a**.



For more information on the `globalsettings.properties` file, refer to the topic, *Working with PROPERTIES Files*, located in the XHQ Administrator's Guide.

setTrendPaths

Sets the scaling for all pens to auto; sets "display" to true for all pens; sets trend to "all scales" mode; does not change the symbol setting.

Syntax: `setTrendPaths(pathArray)`

Argument	Description
pathArray	A JavaScript array of JavaScript String objects representing the paths of the member elements of the trend.

Example: `Given: pathTypes = new Array(3)`

```

pathTypes[0] = "::Enterprise.pen_one"
pathTypes[1] = "::Enterprise.pen_two"
pathTypes[2] = "::Enterprise.pen_three"
Call: setTrendPaths(pathTypes)

```

setTrendScooter

Replaces the current scooter position for the XHQ Trend Viewer.

Syntax: `setTrendScooter (scooterDate)`

Argument	Description
scooterDate	The scooter position in units of time (JavaScript Date object).

Example: *Given:* `var scooterDate = new Date("April 30, 2002 10:24:00");`
Call: `setTrendScooter(scooterDate)`

setTrendTime

Replaces the current time window for the XHQ Trend Viewer.

Syntax: `setTrendTime (t1Date, t2Date)`

Argument	Description
Argument	Description
t1Date	The starting date (a JavaScript Date object).
t2Date	The ending date (a JavaScript Date object).

Example: *Given:* `var t1Date = new Date("April 30, 2002 10:24:00");`
`var t2Date = new Date("April 30, 2002 15:24:00");`
Call: `setTrendTime(t1Date,t2Date)`

showView

This method changes the currently displayed view in the applet. The description of this function is located in the current `xhq.js` file.

Syntax: `showView(objPath, viewName [,mode,filter,x,y])`

Example: `showView('::Acme.WesternDivision.Sales','Forecasts','2',null,'300','300')`



Brackets [] indicate that the argument is optional. Each argument is enclosed by **single** quotation marks (as opposed to the Action Link function `ShowView` arguments, which are enclosed by double quotes).

Argument	Description
objPath	The fully qualified path of the object to view. The path must begin with two colon characters (::) and each member in the path must be separated by a period (.).
viewName	The name of the view of the object.
mode	<ul style="list-style-type: none"> • 0 – use view properties This mode sets the options selected from the View Properties dialog box. This is the default mode. • 1 – replace This mode causes the existing view in the view panel to be replaced. • 2 – popup This mode causes a new pop-up window to display the view. • 3 – use view properties (support for legacy expressions) This mode sets the options selected from the View Properties dialog box. • 4 – do not add to view stack Replace view mode but does not add to the view history stack. • 5 – replace current view with this view in the view stack Replace view mode used to control peer-level view history behavior. <p>Note: If a mode is not specified or if a mode that is not supported is selected, then the default mode of 0 is used.</p>
filter	<p>If you plan to set the XY-coordinates for the pop-up window, set the filter value to "null".</p> <p>Important: Aside from the "null" value, the filter argument should only be changed at the advice of the XHQ Customer Support Team.</p>
x,y	<p>The values you set (in pixels) define the x- and y-coordinates of the top, left corner of the pop-up window.</p> <p>Important: To use the XY arguments, you must set the mode to a value of 2-"popup" and specify the filter value as "null".</p> <p>Note: If the specified mode is 0-"use view properties" and the view has a pop-up defined in the view properties, then the default pop-up position (which is at the center of the screen) is applied. These are optional arguments associated with the mode 2-"popup", and are ignored in the 1-"replace" or 0-"use view properties" modes.</p> <p>See Also: For more information on setting the x- and y-coordinates of the pop-up view window, go to the topic, More About Pop-up Views, located in the XHQ Developer's Guide.</p>

JavaScript Call Function

XHQ supports the following function in the expression system, which allows you to call JavaScript from within XHQ client expressions.



The expressions will only function in the browser client (not in the XHQ Solution Builder) since they rely on the browser's JavaScript engine.

The function accepts a single string parameter and returns a single string result, which represents the results of the last JavaScript function executed. For example, the expression:

```
eval("x = 5")
```

will set the JavaScript variable x to 5 and will return a result of 5.

The expression:

```
eval("foo("+aNumericMember+", '"+aStringMember+"'")")
```

will invoke a JavaScript function, `foo`, passing it the values of two component members.

The breakdown of this expression is as follows.

If the value of the first component member, `aNumericMember`, is 5 and the value of the second component member, `sStringMember`, is "SampleString", the following expression is executed in the JavaScript file:

```
foo(5, 'SampleString')
```

If `foo` returns a value, it will be returned by `eval()` as a string.



When writing the call to the `eval` function, consider how the `eval` function will parse it.

SIDEBAR: CUSTOM JAVASCRIPT FUNCTIONS AND NON-MODAL POP-UPS

If a custom JavaScript function involves calling, for example, a JavaScript alert or confirmation function, then **a non-modal pop-up must be used**. If a modal pop-up is used, the calling thread appears frozen until the JavaScript modal dialog box is closed. During this period, dragging the dialog box leaves a "trace" and the application appears to hang. However, once the modal dialog is closed, the XHQ view repaints and the dialog box trace disappears.

Collection Data Functions

XHQ supports the following Collection Data functions for use within expressions.

Quick Reference – Collection Data Functions

Function	Description	Return Value
<code>absolute(integer)</code>	Sets the pointer to the specified absolute row in this collection result set object.	Boolean
<code>first()</code>	Moves the pointer to the first row in this collection result set object.	Boolean
<code>getBoolean()</code> (integer)	Gets the Boolean value of the designated column in the current row of the Collection Data object.	Boolean

Function	Description	Return Value
<code>getCount ()</code>	Retrieves the total number of records in the collection.	Integer
<code>getDateTime () (int index)</code>	Gets the Date/Time value of the designated column in the current row of this Collection Data object.	DateTime
<code>getDecimal () (int index)</code>	Gets the decimal value of the designated column in the current row of this Collection Data object.	BigDecimal
<code>getDouble () (int index)</code>	Gets the double value of the designated column in the current row of this Collection Data object.	Double
<code>getInt () (int index)</code>	Gets the integer value of the designated column in the current row of this result set object.	Integer
<code>getInterval () (int index)</code>	Gets the time interval value of the designated column in the current row of this Collection Data object.	TimeInterval
<code>getLong () (int index)</code>	Gets the long value of the designated column in the current row of this Collection Data object.	Long
<code>getMetaData ()</code>	Retrieves the collection member info object so that names, types, and other meta data can be extracted for each column.	MemberInfo
<code>getReal () (int index)</code>	Gets the float value of the designated column in the current row of this result set object.	Float
<code>getString () (int index)</code>	Gets the string value of the designated column in the current row of this result set object.	String
<code>getText () (int index)</code>	Gets the extended string value of the designated column in the current row of this Collection Data object.	String
<code>last ()</code>	Moves the pointer to the last row in this collection result set object.	Boolean
<code>next ()</code>	Moves the pointer to the next row in this collection result set object.	Boolean
<code>prev ()</code>	Moves the pointer to the previous row in this collection result set object.	Boolean

getBoolean

Gets the boolean value of the designated column in the current row of this Collection Data object.

Syntax: `getBoolean(Integer index)`

Return: The value for the given column in the current row.
If the current data set pointer is invalid, then it returns false.

Parameter	Description/Format
index	The integer value specifying the column number of the current data set row.

getDateTime

Gets the Date/Time value of the designated column in the current row of this Collection Data object.

Syntax: `getDateTime(Integer index)`

Return: The value for the given column in the current row.
If the current data set pointer is invalid, then it returns the epoch.

Parameter	Description/Format
index	The integer value specifying the column number of the current data set row.

getDecimal

Gets the decimal value of the designated column in the current row of this Collection Data object.

Syntax: `getDecimal(Integer index)`

Return: The value for the given column in the current row.
If the current data set pointer is invalid, then it returns zero.

Parameter	Description/Format
index	The integer value specifying the column number of the current data set row.

getDouble

Gets the double value of the designated column in the current row of this Collection Data object.

Syntax: `getDouble(Integer index)`

Return: The value for the given column in the current row.
If the current data set pointer is invalid, then it returns zero.

Parameter	Description/Format
index	The integer value specifying the column number of the current data set row.

getInt

Gets the integer value of the designated column in the current row of this Collection Data object.

Syntax: `getInt(Integer index)`

Return: The value for the given column in the current row.
If the current data set pointer is invalid, then it returns zero.

Parameter	Description/Format
index	The integer value specifying the column number of the current data set row.

getInterval

Gets the time interval value of the designated column in the current row of this Collection Data object.

Syntax: `getInterval(Integer index)`

Return: The value for the given column in the current row.
If the current data set pointer is invalid, then it returns a zero duration interval.

Parameter	Description/Format
index	The integer value specifying the column number of the current data set row.

getLong

Gets the long value of the designated column in the current row of this Collection Data object.

Syntax: `getLong(Integer index)`

Return: The value for the given column in the current row.
If the current data set pointer is invalid, then it returns zero.

Parameter	Description/Format
index	The integer value specifying the column number of the current data set row.

getReal

Gets the float value of the designated column in the current row of this Collection Data object.

Syntax: `getReal(Integer index)`

Return: The value for the given column in the current row.
If the current data set pointer is invalid, then it returns zero.

Parameter	Description/Format
index	The integer value specifying the column number of the current data set row.

getString

Gets the string value of the designated column in the current row of this Collection Data object.

Syntax: `getString(Integer index)`

Return: The value for the given column in the current row.
If the current data set pointer is invalid, then it returns null.

Parameter	Description/Format
index	The integer value specifying the column number of the current data set row.

getText

Gets the extended string value of the designated column in the current row of this Collection Data object.

Syntax: `getText(Integer index)`

Return: The value for the given column in the current row.
If the current data set pointer is invalid, then it returns null.

Parameter	Description/Format
index	The integer value specifying the column number of the current data set row.

COLLECTIONDATA JAVA OBJECT

This Java Object is accessible from the `getCollectionData` JavaScript function. It is used to return the collection data for a given collection identified by the object path. The following are examples of some of the API calls from JavaScript.

Collection Forward:

```
var objPath = "::MySolution.collectionTest";
var fetchSize = 20;

var sourceObject = document.XHQ_Solution_Viewer.getCollectionData(objPath, null, null,
fetchSize);

// retrieve # of records
numRecords = sourceObject.getCount();
if (numRecords == 0)
    return;

// retrieve meta data object for this collection result set
metaData = sourceObject.getMetaData();
numMembers = metaData.getCount();           // retrieve # of members

// set result set to first row
status = sourceObject.first();

while (status == "true") {
    for (i=0;i > numMembers;i++) {
```



```
// retrieve a single member
m = metaData.getItem(i);

type = m.getType();

if (type == 1)                // boolean
    value = sourceObject.getBoolean(i) + "";
else if (type == 2)          // integer
    value = sourceObject.getInt(i) + "";
else if (type == 3)          // real
    value = sourceObject.getReal(i) + "";
else if (type == 4)          // string
    value = sourceObject.getString(i) + "";
}

// move to next row in result set
status = sourceObject.next();
}
```

Collection Absolute:

```

var objPath = "::MySolution.collectionTest";
var members = "id, integer_value, string_value";
var fetchSize = 5;

var sourceObject = document.XHQ_Solution_Viewer.getCollectionData(objPath, members,
null, fetchSize);

// retrieve # of records
numRecords = sourceObject.getCount();
if (numRecords == 0)
    return;

// retrieve meta data object for this collection result set
metaData = sourceObject.getMetaData();
numMembers = metaData.getCount();           // retrieve # of members

// set to absolute row 0, which is the first row
status = sourceObject.absolute(0);

count = 0;
while (status == "true") {

    count++;

    msg = "Row " + count + ": ";
    for (i=0; i < numMembers; i++) {

        // retrieve a single member
        m = metaData.getItem(i);

        type = m.getType();

        if (type == 1)           // boolean
            value = sourceObject.getBoolean(i) + "";
        else if (type == 2)     // integer
            value = sourceObject.getInt(i) + "";
        else if (type == 3)     // real
            value = sourceObject.getReal(i) + "";
        else if (type == 4)     // string
            value = sourceObject.getString(i) + "";

        msg = msg + m.getName() + "=" + value + " ";
    }

    // jump to next absolute row
    status = sourceObject.absolute(count);
}

```

METHOD SUMMARY FOR COLLECTIONDATA*CollectionData Methods*

Data Type	Method
boolean	absolute (int index)

Data Type	Method
	Sets the pointer to the specified absolute row in this collection result set object.
boolean	first() Moves the pointer to the first row in this collection result set object.
boolean	getBoolean (int index) Gets the boolean value of the designated column in the current row of this result set object.
int	getCount() Retrieves the total number of records in the collection.
int	getInt (int index) Gets the integer value of the designated column in the current row of this result set object.
MemberInfo	getMetaData() Retrieves the collection member info object so that names, types, and other meta data can be extracted for each column.
float	getReal (int index) Gets the float value of the designated column in the current row of this result set object.
String	getString (int index) Gets the string value of the designated column in the current row of this result set object.
boolean	last() Moves the pointer to the last row in this collection result set object.
boolean	next() Moves the pointer to the next row in this collection result set object.
boolean	prev() Moves the pointer to the previous row in this collection result set object.

METHOD DETAIL FOR COLLECTIONDATA

first

```
public boolean first()
```

Moves the pointer to the first row in this collection result set object.

Returns: True if the pointer is successfully set to the first row and it is a valid row; false if there are no rows in the collection result set.

next

```
public boolean next()
```

Moves the pointer to the next row in this collection result set object.

Returns: True if the pointer is on a valid row; false if there are no rows or if the current row is already the last row in the collection result set.

last

```
public boolean last()
```

Moves the pointer to the last row in this collection result set object.

Returns: True if the pointer is on a valid row and false if there are no rows in the collection result set.

prev

```
public boolean prev()
```

Moves the pointer to the previous row in this collection result set object.

Returns: True if the pointer is on a valid row and false if there are no rows or if the current row is already the first row in the collection result set.

absolute

```
public boolean absolute(int index)
```

Sets the pointer to the specified absolute row in this collection result set object.

Argument	Description
index	The integer value offset into the result set.

Returns: True if the pointer is on a valid row; false if there are no rows in the collection result set.

getInt

```
public int getInt(int index)
```

Gets the integer value of the designated column in the current row of this result set object.

Argument	Description
index	The integer value specifying the column number of the current data set row.

Returns: Integer value for the given column in the current row. If current data set pointer is not valid, return 0.

getReal

```
public float getReal(int index)
```

Gets the float value of the designated column in the current row of this result set object.

Argument	Description
index	The integer value specifying the column number of the current data set row.

Returns: Float value for the given column in the current row. If current data set pointer is not valid, return 0.

getBoolean

```
public boolean getBoolean(int index)
```

Gets the boolean value of the designated column in the current row of this result set object.

Argument	Description
index	The integer value specifying the column number of the current data set row.
<i>Returns:</i>	Boolean value for the given column in the current row. If current data set pointer is not valid, return false.

getString

```
public java.lang.String getString(int index)
```

Gets the string value of the designated column in the current row of this result set object.

Argument	Description
index	The integer value specifying the column number of the current data set row.
<i>Returns:</i>	String value for the given column in the current row. If current data set pointer is not valid, return null.

getMetaData

```
public MemberInfo getMetaData()
```

Retrieves the collection member info object so that names, types, and other meta data can be extracted for each column.

Returns: MemberInfo object. MemberProperties can be retrieve from this object.

getCount

```
public int getCount()
```

Retrieves the total number of records in the collection

Returns: The total number of records in the collection request.

Historical Data Functions

XHQ supports the following Historical Data functions for use within expressions.

Quick Reference – Historical Data Functions

Function	Description	Return Value
first()	Moves the pointer to the first row in this history result set object.	Boolean
getBoolean()	Retrieves the Boolean value of the point in the current row in the data set.	Boolean
getCount()	Retrieve the number of points for the history request.	Integer
getInt()	Retrieves the integer value of the point in the current row in the data set.	Integer

Function	Description	Return Value
<code>getQuality()</code>	Returns the quality of the point in the current row in the data set as an integer value.	Integer
<code>getReal(integer)</code>	Retrieves the real value of the point in the current row in the data set.	Float
<code>getTime(integer)</code>	Retrieves the time of the point in the current row in the data set.	Date
<code>getValueMetaData()</code>	Retrieves the member properties object for the value member.	MemberProperties
<code>last()</code>	Moves the pointer to the last row in this history result set object.	Boolean
<code>next()</code>	Moves the pointer to the next row in this history result set object.	Boolean
<code>prev()</code>	Moves the pointer to the previous row in this history result set object.	Boolean

HISTORICALDATA JAVA OBJECT

This Java Object is accessible from the `getHistoricalData` JavaScript function. It is used to returns the history for a given primitive member identified by the object path. The following are examples of some of the API calls from JavaScript.

History Forward:

```

var objPath    = "::.MySolution.realVal";
var numPoints  = 100;
var t1         = new Date("April 30, 2002 10:24:00");
var t2         = new Date("April 30, 2002 15:24:00");

var sourceObject = document.XHQ_Solution_Viewer.getHistoricalData(objPath, t1.getTime(), t2.getTime(), "actuals", numPoints);

// retrieve # of points and type
numPoints = sourceObject.getCount();
if (numPoints == 0)
    return;

// retrieve the data type
metaData = sourceObject.getValueMetaData();
dataType = metaData.getType();

// set to first row in result set
status = sourceObject.first();

var qualityUtilObject = document.XHQ_Solution_Viewer.getQualityUtil();

while (status == "true") {
    if (dataType == 1) {
                                                // boolean

```

```

        v = sourceObject.getBoolean() + "";
    } else if (dataType == 2) {                // integer
        v = sourceObject.getInt() + "";
    } else if (dataType == 3) {                // real
        v = sourceObject.getReal() + "";
    }

    t = sourceObject.getTimeAsString();
    q = sourceObject.getQuality();

    qualityUtilObject.setQuality(q)
qualityDescription = qualityUtilObject.toString();
qualityIsGood = qualityUtilObject.isGood();
qualityIsBad = qualityUtilObject.isBad();
qualityIsUncertain = qualityUtilObject.isUncertain();

    // move to next row in result set
    status = sourceObject.next();
}

```

History Backward:

```

var objPath    = "::MySolution.realVal";
var numPoints  = 100;
var deltaT     = "14400000";                // 4 hours

var sourceObject = document.XHQ_Solution_Viewer.getHistoricalData(objPath, deltaT,
null, "actuals", numPoints);

// retrieve # of points
numPoints = sourceObject.getCount();
if (numPoints == 0)
    return;

// retrieve the data type
metaData = sourceObject.getValueMetaData();
dataType = metaData.getType();

// set to last row in result set
status = sourceObject.last();

while (status == "true") {

    if (dataType == 1) {                // boolean
        v = sourceObject.getBoolean() + "";
    } else if (dataType == 2) {        // integer
        v = sourceObject.getInt() + "";
    } else if (dataType == 3) {        // real
        v = sourceObject.getReal() + "";
    }

    t = sourceObject.getTimeAsString();
    q = sourceObject.getQuality();

    var qualityUtilObject = document.XHQ_Solution_Viewer.getQualityUtil(q);
    qualityDescription = qualityUtilObject.toString();
    qualityIsGood = qualityUtilObject.isGood();
    qualityIsBad = qualityUtilObject.isBad();
    qualityIsUncertain = qualityUtilObject.isUncertain();
}

```

```
// move to previous row in result set
status = sourceObject.prev();
}
```

METHOD SUMMARY FOR HISTORICALDATA

HistoricalData Methods

Data Type	Method
boolean	first() Moves the pointer to the first row in this history result set object.
boolean	getBoolean() Retrieves the boolean value of the point in the current row in the data set.
int	getCount() Retrieve the number of points for the history request.
int	getInt() Retrieves the integer value of the point in the current row in the data set.
int	getQuality() Returns the quality of the point in the current row in the data set as an integer value.
float	getReal() Retrieves the real value of the point in the current row in the data set.
Date	getTime() Retrieves the time of the point in the current row in the data set.
MemberProperties	getValueMetaData() Retrieves the member properties object for the value member.
boolean	last() Moves the pointer to the last row in this history result set object.
boolean	next() Moves the pointer to the next row in this history result set object.
boolean	prev() Moves the pointer to the previous row in this history result set object.

METHOD DETAIL FOR HISTORICALDATA

first

```
public boolean first()
```

Moves the pointer to the first row in this history result set object.

Returns: True if the current pointer is successfully set to the first row and it is a valid row; false if there are no rows in the history result set.

next

```
public boolean next()
```

Moves the pointer to the next row in this history result set object.

Returns: True if the pointer is on a valid row and false if there are no rows or if the current row is the last row in the history result set.

last

```
public boolean last()
```

Moves the pointer to the last row in this history result set object.

Returns: True if the current pointer is on a valid row and false if there are no rows in the history result set.

prev

```
public boolean prev()
```

Moves the pointer to the previous row in this history result set object.

Returns: True if the current pointer is on a valid row and false if there are no rows or if the current row is 0 in the history result set.

getTime

```
public java.util.Date getTime()
```

Retrieves the time of the point in the current row in the data set.

Returns: A Date object if the current pointer is on a valid row; null if the current row pointer is on an invalid row.

getQuality

```
public int getQuality()
```

Returns the quality of the point in the current row in the data set as an integer value.

Returns: If invalid row, return -1, else return one of the following value:

- QualityUtil.GOOD
- QualityUtil.BAD
- QualityUtil.UNCERTAIN

getInt

```
public int getInt()
```

Retrieves the integer value of the point in the current row in the data set.

Returns: An integer value if the current pointer is on a valid row; -1 if the current row pointer is invalid.

getReal

```
public float getReal()
```

Retrieves the real value of the point in the current row in the data set.

Returns: A float value if the current pointer is on a valid row; -1 if the current row pointer is invalid.

getBoolean

```
public boolean getBoolean()
```

Retrieves the boolean value of the point in the current row in the data set.

Returns: A boolean value if the current pointer is on a valid row; -1 if the current row pointer is invalid.

getValueMetaData

```
public MemberProperties getValueMetaData()
```

Retrieves the member properties object for the value member.

Returns: MemberProperties object. If objPath is invalid, return null.

getCount

```
public int getCount()
```

Retrieve the number of points for the history request.

Returns: The number of history points actually retrieved.

MemberInfo Functions

Quick Reference – MemberInfo Functions

Function	Description	Return Value
getCount()	Retrieves the number of members	Integer
getItem(integer)	Retrieves a single member.	MemberProperties

MEMBERINFO JAVA OBJECT

This class is accessible from JavaScript. It is used to encapsulate the members for a given object path. The following is an example of the `getMemberInfo()` API call from JavaScript.

```
var objPath = "::MySolution.Test2Comp";
var sourceObject = document.XHQ_Solution_Viewer.getMemberInfo(objPath);

// retrieve # of members
numMembers = sourceObject.getCount();

for (i=0;i < numMembers;i++) {

    // retrieve a single member
    m = sourceObject.getItem(i);

    alert("Item " + i + " : name="      + m.getName() +
          " type="      + m.getType() +
          " typeDesc="  + m.getTypeDescription() +
          " description=" + m.getDescription() +
```

```

        " componentName=" + m.getComponentName +
        " isHidden="      + m.isHidden() +
        " isPrimitive="   + m.isPrimitive() +
        " isComponent="   + m.isComponent() +
        " isCollection="  + m.isCollection();
    }

```

METHOD SUMMARY FOR MEMBERINFO

MemberInfo Methods

Data Type	Method
int	getCount() Retrieves the number of members.
net.indx.browserclient.api.MemberProperties	getItem(int index) Retrieves a single member.

METHOD DETAIL FOR MEMBERINFO

getItem

```
public net.indx.browserclient.api.MemberProperties getItem(int index)
```

Retrieves a single member.

Argument	Description
index	The index of a member to retrieve in a vector.

getCount

```
public int getCount()
```

Retrieves the number of members.

Returns: The number of members.

QualityUtil Functions

XHQ supports the following functions for use within expressions.

Quick Reference – QualityUtil Functions

Function	Description	Return Value
getInstance (short)	Gets the shared singleton instance of this class.	QualityUtil
getQuality ()	Accessory that returns the specific quality of this Quality object.	Short
isBad ()	Determines if the quality is bad.	Boolean

Function	Description	Return Value
isGood()	Determines if the quality is good.	Boolean
isUncertain()	Determines if the quality is uncertain.	Boolean
setQuality()	Mutator to set the quality field of this Quality object.	Void
toString()	Returns the string representation of the quality.	String

QUALITYUTIL JAVA OBJECT

This class is used to carry the quality attribute of a value member. The following is an example of the `getQualityUtil()` API call from JavaScript.

```
var qualityUtilObject = document.XHQ_Solution_Viewer.getQualityUtil();

while (status == "true") {

    if (dataType == 1) { // boolean
        v = sourceObject.getBoolean() + "";
    } else if (dataType == 2) { // integer
        v = sourceObject.getInt() + "";
    } else if (dataType == 3) { // real
        v = sourceObject.getReal() + "";
    }

    t = sourceObject.getTimeAsString();
    q = sourceObject.getQuality();

    qualityUtilObject.setQuality(q)
qualityDescription = qualityUtilObject.toString();
qualityIsGood = qualityUtilObject.isGood();
qualityIsBad = qualityUtilObject.isBad();
qualityIsUncertain = qualityUtilObject.isUncertain();
```

FIELD SUMMARY FOR QUALITYUTIL

QualityUtil Field Summary

Data Type	Field Summary
static short	BAD This means that value is not useful.
static short	GOOD This means that value is good.
static short	QUALITY MASK The mask for quality (good, bad or uncertain).
static short	UNCERTAIN

Data Type	Field Summary
	This means that value is uncertain for reasons indicated by the substatus.

METHOD SUMMARY FOR QUALITYUTIL

QualityUtil Methods

Data Type	Method
static QualityUtil	getInstance (short quality) Gets the shared singleton instance of this class.
short	getQuality () Accessory that returns the specific quality of this Quality object.
boolean	isBad () Determines if the quality is bad.
boolean	isGood () Determines if the quality is good.
boolean	isUncertain () Determines if the quality is uncertain.
void	setQuality (short quality) Mutator to set the quality field of this Quality object.
java.lang.String	toString () Returns the string representation of the quality.

FIELD DETAIL FOR QUALITYUTIL

QUALITY_MASK

```
public static final short QUALITY_MASK
```

The mask for quality (good, bad or uncertain).

BAD

```
public static final short BAD
```

This means that value is not useful. The substatus bits must be examined for status.

UNCERTAIN

```
public static final short UNCERTAIN
```

This means that value is uncertain for reasons indicated by the substatus.

GOOD

```
public static final short GOOD
```

This means that value is good.

METHOD DETAIL FOR QUALITYUTIL**getInstance**

```
public static QualityUtil getInstance(short quality)
```

Gets a shared singleton instance of this class.

Returns: A shared instance of this class.

setQuality

```
public void setQuality(short quality)
```

Mutator to set the quality field of this Quality object. Using this method, you first set the quality returned by one of the other data retrieval methods. Then, use the other methods to decipher/test the quality status.

Argument	Description
quality	The quality value.

getQuality

```
public short getQuality()
```

Accessor that returns the specific quality of this Quality object.

Returns: The quality of this object.

isGood

```
public boolean isGood()
```

Determine if the quality is good.

Returns: True if the quality is good, false otherwise.

isBad

```
public boolean isBad()
```

Determine if the quality is bad.

Returns: True if the quality is bad, false otherwise.

isUncertain

```
public boolean isUncertain()
```

Determine if the quality is uncertain.

Returns: True if the quality is uncertain, false otherwise.

toString

```
public java.lang.String toString()
```

Return the string representation of the quality.

Overrides: toString in class java.lang.Object

Returns: One of the following strings:

- "GOOD"
- "BAD"
- "UNCERTAIN"
- "UNKNOWN"

ViewInfo Functions*Quick Reference – ViewInfo Functions*

Function	Description	Return Value
getCount()	Retrieves the number of views	Integer
getItem(integer)	Retrieves a single view.	ViewProperties

VIEWINFO JAVA OBJECT

This Java Object is accessible from the `getViewInfo` JavaScript function. It is used to encapsulate the views for a given object path. The following is an example of the `getViewInfo()` API call from JavaScript.

```
var objPath = "::MySolution.Test2Comp";
var sourceObject = document.XHQ_Solution_Viewer.getViewInfo(objPath);

// retrieve # of views
numViews = sourceObject.getCount();

for (i=0;i < numViews;i++) {

    // retrieve a single view
    v = sourceObject.getItem(i);

    alert("View " + i + " : name="          + v.getName() +
          " isDrillable="                + v.isDrillable() +
          " isRunTimeChoice="            + v.isRuntimeChoice() +
          " isHidden="                   + v.isHidden() +
          " isDefault="                  + v.isDefault() +
          " isFlow="                     + v.isFlow() +
          " viewType="                   + v.getType() +
          " viewTypeDesc="               + v.getTypeDescription());
}
```

METHOD SUMMARY FOR VIEWINFO

ViewInfo Methods

Data Type	Method
int	getCount() Retrieves the number of views
ViewProperties	getItem(int index) Retrieves a single view.

METHOD DETAIL FOR VIEWINFO

getItem

```
public ViewProperties getItem(int index)
```

Retrieves a single view.

Argument	Description
index	The index of the view to retrieve in a vector.

getCount

```
public int getCount()
```

Retrieves the number of views.

Returns: The number of views.

Appendices

Section Contents

- A - [Overflow Underflow and Type Conversion Policies](#)
- B - [XHQ Data Mapping](#)
- C - [Units of Measure Conversion](#)

A - Overflow, Underflow, and Type Conversion Policies

In both the client and the server, the evaluation engine (the evaluator) can operate in one of two modes:

- **Classic arithmetic handling mode**
- **Up-casting mode**

This is the default setting.



The mode is determined both in both the client and server using `globalsettings.properties`. The property `UseClassicArithmeticHandling` can be set to **true**, to use the Classic Arithmetic Handling mode, the original narrowing behavior. It is important to note that, in this mode, `STRING` is not up-cast to `TEXT`.

About the Up-casting Mode, Overflows, and Underflows

When the evaluator is in up-casting mode it handles arithmetic computations differently with respect to *overflow* and *underflow* handling. If a member or variable is set to a value that cannot be represented by its type, the quality of the member or variable is set to BAD. In the case of overflows, the `LOWLIMITED` or `HIGHLIMITED` limit information is also set. The value is set to the extreme of the range violated. An underflow occurs when a floating-point number is too close to zero to be stored in the desired type (`REAL` or `DOUBLE`). In this case, the value is set to the closest minimum value for the type and the quality is set to BAD. Quality and value-limited down casts are performed when the result of the evaluation is used to set client or server variables. In the up-casting mode, the handling of `STRING` and `TEXT` type is also involved. `STRING` is up-casted to `TEXT` when appropriate.

Since some methods require specific parameter types, up-casted values are coerced to the required types. This may, of course, result in BAD quality inputs because of down casts.

The following table gives the up-cast type when the expected type over flows.

Up-cast Types

Expected Type	Overflow Type
INT	LONG
LONG	DECIMAL
REAL	DOUBLE
DOUBLE	DECIMAL

Another type of up-casting occurs when the expected types of one or more parameters of an operation or method do not match. In this case, the parameters are up-cast to the closest common type and the operator is changed to match the adjusted input types to perform the operation.

Consider the following expression:

Example: $(2147483647 + 1) - (2147483646 + 1)$

In this example, the expression is parsed as an integer subtraction (ISUB) of the result of an integer addition (IADD) from the result of another integer addition (IADD). However, during evaluation, the first IADD operation results in a LONG value on the stack. The ISUB expects to pop off two INT values but instead finds a LONG and an INT. The evaluator then uses a type mapping system to find that both must up cast to LONG and the long subtraction operation must be used instead. The final result is of type LONG but is the expected value of 1.



If the up-casting were not performed (as is the case with the classic arithmetic handling mode) the first addition would have rolled over to -1, giving the final result of type INT with a value of 0.

Binary Arithmetic Operations

The following table shows the up-casting matrix for parameters of binary arithmetic operations: addition, subtraction, multiplication, division, and modulo.

Up-casting Matrix

Types	INT	REAL	LONG	DOUBLE	DECIMAL
INT	N/A	REAL	LONG	DOUBLE	DECIMAL
REAL	REAL	N/A	FLOAT	DOUBLE	DECIMAL
LONG	LONG	REAL	N/A	DOUBLE	DECIMAL
DOUBLE	DOUBLE	DOUBLE	DOUBLE	N/A	DECIMAL
DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL	N/A*

* N/A means no cast is applicable and no casting or operator changes are required.

Each cell shows how an up-cast is performed for non-matching operands given the type of each parameter. The parameter that does not match the type specified in the matrix is up-cast to that indicated type. The resulting operation is also changed to handle the corresponding parameter types and the result is therefore changed to result type of the operation. For example, if an IADD (integer addition) operation encounters parameters of type LONG and REAL, the LONG parameter is up-cast to REAL and the operation is changed to RADD (real addition).

Unary Arithmetic Operations and Mathematical Functions

The casting rules are similar to unary arithmetic operations (like negation) and single parameter numeric functions like sine and cosine. To determine the up-cast type, use the Up-casting Matrix table in the previous section and the type specified by the function to determine if casting and operator changes are necessary.

Things to Note

- Sine can only be REAL or DOUBLE so only those up-cast types are possible.
- Unary minus can be applied to any numeric type.

Handling of *STRING* and *TEXT* types

Any *STRING* value on the stack may be up-casted to *TEXT* type if subsequent operations require *TEXT* parameters.

B - XHQ Data Mapping

This section provides information on how data from the following back-end sources and connection protocols are mapped to XHQ data type values.

Database to XHQ

The following table shows conversions from database types (present in SQL) to XHQ data types.

Database to XHQ Mapping

XHQ Data Type	Database Type		
	All Numeric	CHAR, VARCHAR, LONGVARCHAR, CLOB	DATE, TIME, TIMESTAMP
Boolean	Yes	Yes	No
Integer	Yes	Yes	No
Long	Yes	Yes	No
Real	Yes	Yes	No
Double	Yes	Yes	No
Decimal	Yes	Yes	No
Date/Time	No	No	Yes
Interval	Yes	No	No
String	Yes	Yes	Yes
Text	Yes	Yes	Yes



Float or Real

Although synonymous, *Real* is the preferred term in XHQ.

For JDBC-based Connectors



The JDBC-based connectors are the Oracle, ODBC, SQL, and PostgreSQL connectors. The following table applies to all JDBC-based connectors.

XHQ-JDBC-based Data Type Mapping

XHQ	JDBC-based Connector								
	Boolean	Integer	Real	Double	Decimal	Date Time	Interval	String	Text
Boolean	Yes	Yes	Yes	Yes	Yes	No	N/A	Yes	Yes
Integer	Yes	Yes	Yes	Yes	Yes	No	N/A	Yes	Yes
Long	Yes	Yes	Yes	Yes	Yes	No	N/A	Yes	Yes
Real	Yes	Yes	Yes	Yes	Yes	No	N/A	Yes	Yes
Double	Yes	Yes	Yes	Yes	Yes	No	N/A	Yes	Yes
Decimal	Yes	Yes	Yes	Yes	Yes	No	N/A	Yes	Yes
Date/Time	No	No	No	No	No	Yes	N/A	No	No
Interval	No	Yes	Yes	Yes	Yes	No	N/A	No	No
String	Yes	Yes	Yes	Yes	Yes	No	N/A	Yes	Yes
Text	Yes	Yes	Yes	Yes	Yes	No	N/A	Yes	Yes

For the OPC Connector

XHQ-OPC Data Type Mapping

XHQ	OPC											
	Char	Byte	Short	U-Short	Int	U-Int	Float	Double	Currency	Date	String	Bool
Integer	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Real	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Boolean	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes
String	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Long	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Double	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Decimal	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Date/Time	No	No	No	No	Yes	Yes	No	No	No	Yes	No	No
Interval	No	No	No	No	Yes	Yes	No	No	No	No	No	No
Text	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

For the PHD Connector

XHQ-PHD Data Type Mapping

XHQ	PHD			
	Integer	Float	Boolean	String
Integer	Yes	Yes	Yes	Yes
Real	Yes	Yes	Yes	Yes
Boolean	Yes	Yes	Yes	Yes
String	Yes	Yes	Yes	Yes
Long	Yes	Yes	Yes	Yes
Double	Yes	Yes	Yes	Yes
Decimal	Yes	Yes	No	No
Date/Time	No	No	No	No
Interval	No	No	No	No
Text	Yes	Yes	Yes	Yes

For the PI Connector

XHQ-PI Data Type Mapping

XHQ	PI			
	Integer 16/32	Float 16/32/64	String	Digital
Integer	Yes	Yes	No	Yes
Real	Yes	Yes	No	Yes
Boolean	Yes	No	No	Yes
String	Yes	Yes	Yes	Yes
Long	Yes	Yes	No	Yes
Double	Yes	Yes	No	Yes
Decimal	Yes	Yes	No	No
Date/Time	No	No	No	No
Interval	Yes	No	No	No
Text	Yes	Yes	Yes	Yes

For the IP.21 Connector

XHQ-IP.21 Data Type Mapping

XHQ	IP.21			
	Integer 16/32	Float 16/32/64	String	Time
Integer	Yes	Yes	No	N/A
Real	Yes	Yes	No	N/A
Boolean	Yes	No	No	N/A
String	Yes	Yes	Yes	N/A
Long	Yes	Yes	No	N/A
Double	Yes	Yes	No	N/A
Decimal	Yes	Yes	No	N/A
Date/Time	No	No	No	N/A
Interval	Yes	No	No	N/A
Text	Yes	Yes	Yes	N/A

For the XHQ Connector

XHQ-XHQ Data Type Mapping

XHQ	XHQ Connector									
	Boolean	Integer	Long	Real	Double	Decimal	Date Time	Interval	String	Text
Boolean	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Integer	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Long	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Real	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Double	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Decimal	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Date/Time	No	No	No	No	No	No	Yes	No	Yes	Yes
Interval	No	No	No	No	No	No	Yes	No	Yes	Yes
String	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Text	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

For the Documentum Connector

XHQ-Documentum Data Type Mapping

XHQ	Documentum								
	Boolean	Integer	Real	Double	Decimal	Date Time	Interval	String	Text
Boolean	Yes	No	N/A	No	N/A	No	N/A	No	N/A
Integer	No	Yes	N/A	No	N/A	No	N/A	No	N/A
Long	No	Yes	N/A	No	N/A	No	N/A	No	N/A
Real	No	No	N/A	Yes	N/A	No	N/A	No	N/A
Double	No	No	N/A	Yes	N/A	No	N/A	No	N/A
Decimal	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Date/Time	No	No	N/A	No	N/A	Yes	N/A	No	N/A
Interval	No	Yes	N/A	No	N/A	No	N/A	Yes	N/A
String	No	No	N/A	No	N/A	No	N/A	Yes	N/A
Text	No	No	N/A	No	N/A	No	N/A	No	N/A

For the SIMATIC IT Historian Connector

XHQ-SIMATIC IT Historian Data Type Mapping

XHQ	SIMATIC IT Historian											
	Char	Byte	Short	U-Short	Int	U-Int	Float	Double	Currency	Date	String	Bool
Integer	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Real	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Boolean	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes
String	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Long	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Double	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Decimal	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Date/Time	No	No	No	No	Yes	Yes	No	No	No	Yes	No	No
Interval	No	No	No	No	Yes	Yes	No	No	No	No	No	No
Text	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

For the SAP Connector

SAP supports the following **data types**:

JCO.TYPE_CHAR	JCO.TYPEINT1	JCO.TYPE_INT2
JCO.TYPE_NUM	JCO.TYPE_BCD	JCO.TYPE_FLOAT
JCO.TYPE_BYTE	JCO.TYPE_DATE	JCO.TYPE_TIME
JCO.TYPE_STRING	JCO.TYPE_XSTRING	

For these data types, the SAP connector uses the **getString()** method supplied by SAP to get a string version of the data. This string is then used to parse (or convert) the value to the appropriate XHQ data type.

For the XML Connector

All values returned from XML are brought into the connector as string values. Then a **ValueItem** conversion is used to convert where possible from the string representation into the mapped XHQ data type.

C - Units of Measure Conversion

The Units of Measure Conversion Chart consists of the Unit of Measure, the Base Unit, and the given conversion Factors.

Unit of Measure	Definition	Base Unit	FACTOR A	FACTOR B	FACTOR C	FACTOR D
ft2	square foot	m2	0	0.09290304	1	0
ft2/h	square feet/hour	m2/s	0	0.09290304	3600	0
ft2/s	square feet/second	m2/s	0	0.09290304	1	0
ft3	cubic feet	m3	0	0.028316847	1	0
ft3/d	cubic feet/day	m3/s	0	0.02831685	86400	0
ft3/h	cubic feet/hour	m3/s	0	0.02831685	3600	0
ft3/min	cubic feet/minute	m3/s	0	0.02831685	60	0

Units Of Measure Conversion Chart

The conversion algorithm is based on four conversion factors for each unit (shown as the last four columns in the example table above: Factor A, Factor B, Factor C, and Factor D). Units are grouped into categories (for example, area, currency, or time). The base unit is the "common" unit for the given category.

XHQ Categories and their Base Unit

Unit Category	Base Unit	Unit Category	Base Unit
Acceleration	m/s2	Inductance	H
Angle	rad	Kinematic Viscosity	m2/s
Area	m2	Length	m
Capacitance	F	Magnetic Flux	Wb
Currency US	\$	Magnetic Flux Intensity	Wb/m2
Current	A	Mass	kg
Density	kg/m3	Pressure	Pa
Dynamic Velocity	Pa.s	Resistance	ohm
Electric Quantity	C	Temperature	K
Energy	J	Time	s
Flowrate	m3/s	Velocity	m/s
Force	N	Voltage	V
Frequency	Hz	Volume	m3

How units are converted

Units are converted using a normalization/de-normalization method. When converting from one unit to another in the given category, the value is first normalized to the base unit, and then de-normalized to the target unit.

During **normalization**, a value is converted from some given unit to a target base unit. To normalize, the following conversion algorithm is used.

```
X = value in Unit of Measure (UOM) column
Y = value in Base Unit column
if B != 0
Y = (X + A) * B/C
else if D != 0
Y = (A/X - C) / D
else
Y = A/X/C
```

Where A, B, C, and D represent the given conversion factors.

De-normalization converts the base unit value to the compatible target unit value. The following is the conversion algorithm for de-normalization.

```
X = value in Base Unit column
Y = value in Unit of Measure (UOM) column
if B != 0
Y = C * X/B - A
else if D != 0
Y = A/(D*X + C)
else
Y = A/(C * X)
```

Again, A, B, C, and D represent the given conversion factors.



All conversion calculations use IEEE double-precision, floating-point numbers. The results of conversion may include loss of precision or rounding errors that are associated with the IEEE floating-point math.

Conversion Chart

Units of Measure Conversion Chart

Unit of Measure	Definition	Base Unit	Factor A	Factor B	Factor C	Factor D
A	ampere	A	0	1	1	0
a	annum	s	0	31558150	1	0
A.h	Ampere hour	C	0	3600	1	0
acre	acre	m ²	0	4046.873	1	0
atm	atmosphere	Pa	0	101325.3532	1	0
bar	bar	Pa	0	100000	1	0
bbl	barrel	m ³	0	0.158987295	1	0
bbl/d	barrel/day	m ³ /s	0	0.158987295	86400	0
bbl/hr	barrel/hour	m ³ /s	0	0.158987295	3600	0
bbl/min	barrels/minute	m ³ /s	0	0.158987295	60	0
Btu	British Thermal Unit	J	0	1055.055853	1	0
C	Coulomb	C	0	1	1	0
cal	calorie	J	0	4.184	1	0
cm	centimeter	m	0	0.01	1	0
cm/s	centimeter/second	m/s	0	0.01	1	0
cm ²	square centimeter	m ²	0	0.0001	1	0
cm ² /s	centimeters squared/second	m ² /s	0	0.0001	1	0
cm ³	cubic centimeter	m ³	0	1.00E-06	1	0
cP	centipoises	Pa.s	0	0.001	1	0
cSt	centistokes	m ² /s	0	1.00E-06	1	0
d	day	s	0	86400	1	0
dAPI	API gravity	kg/m ³	141500	1000000	141500	0
dega	degree of an angle	rad	0	3.141592654	180	0
degC	degrees Celsius	K	273.15	1	1	0
degF	degree Fahrenheit	K	459.67	5	9	0
degR	degrees Rankin	K	0	5	9	0
dyne	dynes	N	0	1.00E-05	1	0
F	Farad	F	0	1	1	0
ft	foot	m	0	0.3048	1	0

Unit of Measure	Definition	Base Unit	Factor A	Factor B	Factor C	Factor D
ft/d	feet/day	m/s	0	0.3048	86400	0
ft/h	feet/hour	m/s	0	0.3048	3600	0
ft/min	feet/minute	m/s	0	0.3048	60	0
ft/s	feet/second	m/s	0	0.3048	1	0
ft/s ²	feet/second squared	m/s ²	0	0.3048	1	0
ft ²	square foot	m ²	0	0.09290304	1	0
ft ² /h	square feet/hour	m ² /s	0	0.09290304	3600	0
ft ² /s	square feet/second	m ² /s	0	0.09290304	1	0
ft ³	cubic feet	m ³	0	0.028316847	1	0
ft ³ /d	cubic feet/day	m ³ /s	0	0.02831685	86400	0
ft ³ /h	cubic feet/hour	m ³ /s	0	0.02831685	3600	0
ft ³ /min	cubic feet/minute	m ³ /s	0	0.02831685	60	0
ft ³ /s	cubic feet/second	m ³ /s	0	0.02831685	1	0
g	gram	kg	0	0.001	1	0
g/cm ³	grams/cubic centimeter	kg/m ³	0	1000	1	0
g/L	grams/liter	kg/m ³	0	1	1	0
g/m ³	grams/cubic meter	kg/m ³	0	0.001	1	0
Gal	Galileo	m/s ²	0	0.01	1	0
galUK	UK gallon	m ³	0	0.004546092	1	0
galUK/hr	UK gallons/hour	m ³ /s	0	0.004546092	3600	0
galUK/min	UK gallons/minute	m ³ /s	0	0.004546092	60	0
galUS	US gallons	m ³	0	0.003785412	1	0
galUS/hr	US gallons/hour	m ³ /s	0	0.003785412	3600	0
galUS/min	US gallons/minute	m ³ /s	0	0.003785412	60	0
grad	grad	rad	0	3.141592654	200	0
H	Henry	H	0	1	1	0
h	hour	s	0	3600	1	0
hp.hr	horsepower hour	J	0	2684519.54	1	0
in	inch	m	0	0.0254	1	0
in ²	square inches	m ²	0	0.00064516	1	0
in ² /s	square inches/second	m ² /s	0	0.00064516	1	0

Unit of Measure	Definition	Base Unit	Factor A	Factor B	Factor C	Factor D
in3	cubic inches	m3	0	1.64E-05	1	0
J	joule	J	0	1	1	0
J/m	joules/meter	N	0	1	1	0
K	Kelvin	K	0	1	1	0
kbbbl	thousand barrels	m3	0	158.9872949	1	0
kbbbl/d	thousand barrels/day	m3/s	0	158.9872949	86400	0
kC	kilocoulombs	C	0	1000	1	0
kcal	kilocalories	J	0	4184	1	0
kcf	thousand cubic feet	m3	0	28.31685	1	0
kcf/d	thousand cubic feet/day	m3/s	0	0.02831685	86.4	0
kg	kilogram	kg	0	1	1	0
kg/L	kilograms/liter	kg/m3	0	1	0.001	0
kg/m3	kilograms/cubic meter	kg/m3	0	1	1	0
kgf	kilogram force	N	0	9.80665	1	0
kJ	kilojoules	J	0	1000	1	0
klbm	thousand pounds mass	kg	0	453.5924	1	0
km	kilometer	m	0	1000	1	0
km/h	kilometers/hour	m/s	0	1	3.6	0
km2	square kilometers	m2	0	1000000	1	0
k-m3/d	thousand cubic meters/day	m3/s	0	1000	86400	0
kN	kilonewtons	N	0	1000	1	0
knot	knots	m/s	0	1852	3600	0
kPa	kilopascals	Pa	0	1000	1	0
kV	kilovolt	V	0	1000	1	0
kW.h	kilowatt hours	J	0	3600000	1	0
L	liter	m3	0	0.001	1	0
L/min	liters/minute	m3/s	0	1	60000	0
L/s	liters/second	m3/s	0	0.001	1	0
lbf	pounds force	N	0	4.448222	1	0

Unit of Measure	Definition	Base Unit	Factor A	Factor B	Factor C	Factor D
lbf/ft2	pounds force/square foot	Pa	0	47.88026	1	0
lbm	pounds mass	kg	0	0.4535924	1	0
lbm/bbl	pounds mass/barrel	kg/m3	0	2.85301	1	0
lbm/ft.h	pounds mass/foot hour	Pa.s	0	0.000413379	1	0
lbm/ft.s	pounds mass/foot second	Pa.s	0	1.488164	1	0
lbm/ft3	pounds mass/cubic foot	kg/m3	0	16.01846	1	0
lbm/galUK	pounds mass/UK gallon	kg/m3	0	99.77633	1	0
lbm/galUS	pounds mass/US gallon	kg/m3	0	119.8264	1	0
lbm/kbbl	pounds mass/1000 barrels	kg/m3	0	0.00285301	1	0
m	meter	m	0	1	1	0
m/d	meters/day	m/s	0	1	86400	0
m/h	meters/hour	m/s	0	1	3600	0
m/s	meters/second	m/s	0	1	1	0
m/s2	meters/second squared	m/s2	0	1	1	0
m2	square meters	m2	0	1	1	0
m2/h	square meters/hour	m2/s	0	1	3600	0
m2/s	square meters/second	m2/s	0	1	1	0
m3	cubic meters	m3	0	1	1	0
m3/d	cubic meters/day	m3/s	0	1	86400	0
m3/h	cubic meters/hour	m3/s	0	1	3600	0
m3/min	cubic meters/minute	m3/s	0	1	60	0
m3/s	cubic meters/second	m3/s	0	1	1	0
mA	milliamp	A	0	0.001	1	0
Mbbl	million barrels	m3	0	158987.2949	1	0
mC	millicoulomb	C	0	0.001	1	0
Mcf	million cubic feet	m3	0	28316.85	1	0

Unit of Measure	Definition	Base Unit	Factor A	Factor B	Factor C	Factor D
mg	milligram	kg	0	1.00E-06	1	0
mg/L	milligram/liter	kg/m3	0	0.001	1	0
mH	millihenries	H	0	0.001	1	0
mi	mile	m	0	1609.344	1	0
mi/h	miles/hour	m/s	0	1609.344	3600	0
mi2	square miles	m2	0	2589988.11	1	0
min	minutes	s	0	60	1	0
MJ	megajoules	J	0	1000000	1	0
mL	milliliter	m3	0	1.00E-06	1	0
mm	millimeter	m	0	0.001	1	0
mm/s	millimeters/second	m/s	0	0.001	1	0
mm2/s	square millimeters/second	m2/s	0	1.00E-06	1	0
mmHg(0C)	millimeters of Mercury at 0 deg C	Pa	0	133.3224	1	0
mohm	milliohm	ohm	0	0.001	1	0
ms	milliseconds	s	0	0.001	1	0
mV	millivolts	V	0	0.001	1	0
MW.h	megawatt hours	J	0	3600000000	1	0
mWb	milliwebers	Wb	0	0.001	1	0
N	Newton	N	0	1	1	0
N.m	Newton meter	J	0	1	1	0
N.m/m	Newton meters/meter	N	0	1	1	0
N.s/m2	Newton seconds/meter squared	Pa.s	0	1	1	0
N/m2	Newtons/square meter	Pa	0	1	1	0
nm	nanometer	m	0	1.00E-09	1	0
ns	nanoseconds	s	0	1.00E-09	1	0
ohm	ohm	ohm	0	1	1	0
P	poise	Pa.s	0	0.1	1	0
Pa	Pascal	Pa	0	1	1	0

Unit of Measure	Definition	Base Unit	Factor A	Factor B	Factor C	Factor D
Pa(g)	Pascal gauge	Pa	101325.3532	1	1	0
Pa.s	Pascal seconds	Pa.s	0	1	1	0
psf	pounds/square foot	Pa	0	47.88026	1	0
psi	pounds/square inch	Pa	0	6894.757293	1	0
psia	pounds/square inch absolute	Pa	0	6894.757293	1	0
psig	pounds/square inch gauge	Pa	14.7	6894.757293	1	0
rad	radian	rad	0	1	1	0
s	second	s	0	1	1	0
t	tonne	kg	0	1000	1	0
therm	therms	J	0	105505585.3	1	0
tonfUK	UK tons force	N	0	9964.016	1	0
tonfUS	US tons force	N	0	8896.443	1	0
tonUK	UK tons	kg	0	1016.047	1	0
tonUS	US tons	kg	0	907.1847	1	0
uA	microampere	A	0	1.00E-06	1	0
uC	microcoulomb	C	0	1.00E-06	1	0
uF	microfarads	F	0	1.00E-06	1	0
ug	micrograms	kg	0	1.00E-09	1	0
um	micron	m	0	1.00E-06	1	0
uohm	microohm	ohm	0	1.00E-06	1	0
us	microsecond	s	0	1.00E-06	1	0
uV	microvolts	V	0	1.00E-06	1	0
V	volt	V	0	1	1	0
Wb	weber	Wb	0	1	1	0
wk	weeks	s	0	604800	1	0
yd	yard	m	0	0.9144	1	0