



Cisco Unified Contact Center Express Expression Language Reference Guide, Release 11.0(1)

Cisco Unified Contact Center Express Scripting and Development Series:
Volume 3

First Published: August 27, 2015

Americas Headquarters

Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
<http://www.cisco.com>
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883

THE SPECIFICATIONS AND INFORMATION REGARDING THE PRODUCTS IN THIS MANUAL ARE SUBJECT TO CHANGE WITHOUT NOTICE. ALL STATEMENTS, INFORMATION, AND RECOMMENDATIONS IN THIS MANUAL ARE BELIEVED TO BE ACCURATE BUT ARE PRESENTED WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. USERS MUST TAKE FULL RESPONSIBILITY FOR THEIR APPLICATION OF ANY PRODUCTS.

THE SOFTWARE LICENSE AND LIMITED WARRANTY FOR THE ACCOMPANYING PRODUCT ARE SET FORTH IN THE INFORMATION PACKET THAT SHIPPED WITH THE PRODUCT AND ARE INCORPORATED HEREIN BY THIS REFERENCE. IF YOU ARE UNABLE TO LOCATE THE SOFTWARE LICENSE OR LIMITED WARRANTY, CONTACT YOUR CISCO REPRESENTATIVE FOR A COPY.

The Cisco implementation of TCP header compression is an adaptation of a program developed by the University of California, Berkeley (UCB) as part of UCB's public domain version of the UNIX operating system. All rights reserved. Copyright © 1981, Regents of the University of California.

NOTWITHSTANDING ANY OTHER WARRANTY HEREIN, ALL DOCUMENT FILES AND SOFTWARE OF THESE SUPPLIERS ARE PROVIDED "AS IS" WITH ALL FAULTS. CISCO AND THE ABOVE-NAMED SUPPLIERS DISCLAIM ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THOSE OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OR ARISING FROM A COURSE OF DEALING, USAGE, OR TRADE PRACTICE.

IN NO EVENT SHALL CISCO OR ITS SUPPLIERS BE LIABLE FOR ANY INDIRECT, SPECIAL, CONSEQUENTIAL, OR INCIDENTAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOST PROFITS OR LOSS OR DAMAGE TO DATA ARISING OUT OF THE USE OR INABILITY TO USE THIS MANUAL, EVEN IF CISCO OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Cisco and the Cisco logo are trademarks or registered trademarks of Cisco and/or its affiliates in the U.S. and other countries. To view a list of Cisco trademarks, go to this URL: www.cisco.com/go/trademarks. Third-party trademarks mentioned are the property of their respective owners. The use of the word partner does not imply a partnership relationship between Cisco and any other company. (1110R)

Cisco Unified Contact Center Express Expression Language Reference Guide 11.0(1)
Copyright © 2015 Cisco Systems, Inc. All rights reserved.



CONTENTS

Preface i

Purpose i-i

Audience i

Organization ii

Related Documentation ii

Conventions iii

Obtaining Documentation, Obtaining Support, and Security Guidelines iv

Documentation Feedback iv

CHAPTER 1

About the Cisco Unified CCX Expression Language 1-1

The Language Purpose 1-1

How to Access the Language 1-1

The Language Syntax 1-2

The Language Classes and Interfaces 1-3

Language Code Examples 1-3

Expression Language Terminology 1-4

Expression Language Operator Summary 1-7

Operators Used with Prompts and Documents 1-8

 The Prompt Substitution Operator ||| 1-9

 Qualifier Operators 1-9

 Additive Operators 1-9

 Integer/Boolean Conditional-Or Operator || 1-10

 Escalation Operator || 1-11

 Compound Assignment Operators 1-13

Expression Language Keywords 1-14

Expression Language Literals 1-15

 Lexical Literals 1-15

 Class Literals 1-15

 Complex Literals 1-16

Expression Language Data Types 1-16

 Type Variables 1-17

 Type Values 1-17

 Primitive Values 1-18

 Reference Values 1-24

- Where Types Are Used 1-27
- The Language Variables 1-27
 - About Language Variables 1-27
 - Primitive Variables 1-28
 - Reference Variables 1-28
 - Variable Categories 1-28
 - “final” Variables 1-29
 - Initial Values of Variables 1-29
 - Definite Local Variable Assignment 1-30
 - Variable Types, Classes, and Interfaces 1-30
- About Conversions in the Expression Language 1-31
 - Prompt Conversions 1-32
 - Document Conversions 1-32
 - String Conversions 1-32
 - String Parsing 1-33
 - New Objects Resulting from Conversions 1-33

CHAPTER 2

- Using Expressions and the Expression Editor 2-1**
 - How to Access the Cisco Unified CCX Expression Editor 2-1
 - How to Use the Expression Editor 2-2
 - How To Enter Expressions in the Expression Editor 2-2
 - About the Expression Editor Toolbar 2-4
 - Toolbar Tabs 2-5
 - A Pop-Up Menu 2-7
 - Showing or Hiding the Expression Editor Toolbar 2-8
 - About the Expression Editor Syntax Buttons 2-9
 - About Expression and Java Licensing 2-9

CHAPTER 3

- Expression Editor Tool Reference Descriptions 3-1**
 - Friendly Data Types 3-2
 - Tool Tips 3-4
 - Tool Tips For the Java and Miscellaneous Tool Tabs 3-4
 - Tool Tips For All the Expression Editor Tools 3-6
 - Array 3-8
 - About Arrays 3-8
 - Array Java Specification on the Web 3-8
 - Example Array Code 3-9
 - Array Variables 3-10

Index Variables	3-10
Array Methods	3-11
Array tab Syntax Buttons	3-11
BigDecimal	3-13
About BigDecimals	3-13
BigDecimal Java Specification on the Web	3-13
Example BigDecimal Code	3-14
BigDecimal Variables	3-15
BigDecimal Constructors, Methods, and Attributes	3-15
BigDecimal tab Syntax Buttons	3-16
BigInteger	3-18
About BigIntegers	3-18
BigInteger Specification on the Web	3-18
Example BigInteger Code	3-18
BigInteger Variables	3-20
BigInteger Constructors, Methods, and Attributes	3-20
BigInteger tab Syntax Buttons	3-20
Boolean	3-23
About Booleans	3-23
Boolean Specification on the Web	3-23
Example Complex Expression Using a Boolean	3-23
Boolean Variables	3-24
Boolean Constructors, Methods, and Attributes	3-25
Boolean tab Syntax Buttons	3-25
Boolean Literals	3-27
Byte	3-27
About Bytes	3-28
Byte Java Specification on the Web	3-28
Example Simple Expression Use the Byte Data Type	3-29
Byte Constructors, Methods, and Attributes	3-29
Byte Variables	3-29
Byte tab Syntax Buttons	3-29
Character	3-32
About the Character Data Type	3-32
Character Specification on the Web	3-33
Example Character Code	3-33
Character Methods and Attributes	3-34
Character Variables	3-34
Character tab Syntax Buttons	3-35

- Character Literals 3-35
- Escape Character Literals 3-36
- Currency 3-37
 - About Currencies 3-37
 - Currency Specification and Code List on the Web 3-37
 - Example Simple Expression Using a Prompt and Currency 3-38
 - Currency Variables 3-38
 - Currency Methods and Attributes 3-38
 - Recent Currencies 3-38
 - Currency tab Syntax Button 3-39
 - Currency Literals 3-39
- Date 3-39
 - About Dates 3-39
 - Date Specification on the Web 3-40
 - Example Date Code 3-40
 - Date Variables 3-41
 - Date Constructors and Methods 3-42
 - Date tab Syntax Buttons 3-42
 - Date Literals 3-43
- Document 3-44
 - About Expression Language Documents 3-45
 - Example Expression Using a Document 3-45
 - Document Variables 3-45
 - Browse Documents Dialog Box 3-46
 - Document tab Syntax Buttons 3-46
 - Document Literals 3-48
 - Document Concatenation Operator + 3-51
 - Document Qualifier Operator 3-52
 - Time of Week, Day of Week, and Time of Day Documents 3-52
- Double 3-54
 - About Doubles 3-54
 - Double Specification on the Web 3-54
 - Example Double Code 3-55
 - Double Variables 3-56
 - Double Constructors, Methods, and Attributes 3-56
 - Double tab Syntax Buttons 3-56
- Float 3-57
 - About Floats 3-58
 - Float Specification on the Web 3-58

Example Float Code	3-58
Float Variables	3-59
Float Constructors, Methods, and Attributes	3-59
Float tab Syntax Buttons	3-60
Floating-Point Literals	3-61
Grammar	3-62
About Grammars	3-62
Grammar Specifications on the Web	3-62
Example Grammar Code	3-63
Grammar Variables	3-63
Browse Grammars Dialog Box	3-64
Grammar tab Syntax Buttons	3-64
Grammar Literals	3-65
Compound Grammar	3-68
Compound Grammar Indexing	3-69
Grammar Template File Types and Template Enhancements	3-69
Integer	3-69
About the Integer Class	3-70
Integer Specification on the Web	3-70
Example Integer Code	3-70
Integer Variables	3-71
Integer Constructors, Methods, and Attributes	3-71
Integer Operations	3-72
Integer tab Syntax Buttons	3-72
Integer Literals	3-75
Java	3-77
Java Specification on the Web	3-77
Example Java tab Code	3-77
Java tab Constructors, Methods, and Attributes	3-78
How to Access a Java Constructor, Method, or Attribute for Any Class	3-79
How to Make Custom Java Classes Available to the Cisco Unified CCX Editor	3-80
Java tab Syntax Button Descriptions	3-80
Language	3-84
Language Class and Code Specifications on the Web	3-84
Example Language Code	3-85
Language Variables	3-85
Language Methods and Attributes	3-85
Recent Languages	3-85
All Languages	3-86

- Language tab Syntax Button 3-86
- Language Literals 3-86
- Long 3-87
 - About the Long Data Type 3-87
 - Long Specification on the Web 3-87
 - Example Long Code 3-87
 - Long Variables 3-89
 - Long Constructors, Methods, and Attributes 3-89
 - Long tab Syntax Buttons 3-89
- Miscellaneous 3-91
 - Example Simple Expression Using the Miscellaneous Tab 3-92
 - Object Variables 3-92
 - DayOfWeek 3-93
 - Gender 3-93
 - The Null Literal 3-93
 - Miscellaneous tab Syntax Buttons 3-93
- Prompt 3-94
 - About Prompts 3-94
 - Prompt Variables 3-95
 - Browse Prompts Dialog Box 3-96
 - Prompt tab Syntax Buttons 3-96
 - Prompt Literals 3-98
 - Operators Used with Prompts 3-104
 - Prompt Templates 3-106
 - Prompt Conversions 3-109
- Script 3-110
 - About Scripts 3-110
 - Example Simple Expression Using a Script 3-111
 - Script Variables 3-111
 - Browse Scripts 3-111
 - Script tab Syntax Buttons 3-111
- Short 3-113
 - About the Short Data Type 3-113
 - Numeric Type Specification on the Web 3-113
 - Example Short Code 3-113
 - Short Variables 3-115
 - Short Constructors, Methods, and Attributes 3-115
 - Short tab Syntax Buttons 3-115

String	3-117
About the String Class	3-118
Java String Specification on the Web	3-118
Example Simple Expression Using a String	3-118
String Variables	3-119
String Constructors, Methods, and Attributes	3-119
String tab Syntax Buttons	3-119
String Literals	3-120
Escape Sequences for Character and String Literals	3-121
An Array of Characters is Not a String	3-121
Time	3-122
About Time Data	3-122
Time Specification on the Web	3-122
Example Simple Expression using Time and Three Script Variables	3-123
Time Variables	3-123
Time Constructors and Methods	3-123
Time tab Syntax Buttons	3-124
Time Literals	3-124
User	3-125
About Users	3-125
Example User Code	3-126
User Variables	3-126
User Syntax Button	3-126



Preface

The *Cisco Unified Contact Center Express (Cisco Unified CCX) Scripting and Development Series* contains three volumes and provides information about how to use the Cisco Unified CCX Editor to develop a wide variety of interactive scripts:

- *Volume 1, Getting Started with Scripts*, provides an overview of the Cisco Unified CCX and the Cisco Unified CCX Editor web interface.
- *Volume 2, Editor Step Reference*, describes each individual step in the Cisco Unified CCX Editor palettes.
- *Volume 3, Expression Language Reference* (this book), provides details on working with the Cisco Unified CCX Expression Editor

The information in all three volumes is included in the Cisco Unified CCX Step Editor online help. This means by searching in one location, the Cisco Unified CCX Step Editor help, you should be able to find any information contained in all three volumes.

Purpose

This document briefly describes the Expression Language and how to use its Expression Editor. The Expression Editor is an addition to the Cisco Unified CCX script application set of APIs that provides support for creating, validating, and evaluating expressions in Cisco Unified CCX scripts.

The Expression Language syntax is borrowed from other languages such as C++, C, and Java.

Audience

The *Cisco Unified Contact Center Express Expression Language Reference Guide 11.0(1)* is written for script developers who use the Cisco Unified CCX Editor to create and modify Cisco Unified CCX scripts. This guide targets developers who have the IP telephony knowledge required to create useful applications and who also have some background in programming or scripting. While readers of this guide do not need experience or training with Java, such training is useful to fully utilize the capabilities of the Expression Language.

Organization

Chapter	Describes
Chapter 1, “About the Cisco Unified CCX Expression Language”	The Expression Language
Chapter 2, “Using Expressions and the Expression Editor”	How to use the Cisco Unified CCX Expression Editor
Chapter 3, “Expression Editor Tool Reference Descriptions”	Each of the Expression Editor editing aids on the Expression Editor tool bars.

Related Documentation

Refer to the following documents for further information about Cisco Unified CCX Applications and Products:

- *Cisco Unified Contact Center Express Scripting and Development Series: Volume 1, Getting Started with Scripts*
- *Cisco Unified Contact Center Express Scripting and Development Series: Volume 2, Editor Step Reference*
- *Cisco Unified Contact Center Express Administration Guide*
- *Cisco Unified Contact Center Express Installation and Upgrade Guide*
- *Cisco Unified Contact Center Express Servicing and Troubleshooting Guide*
- *Cisco Unified Communications Manager Administration Guide*
- *Cisco Unified Communications Manager Extended Services Administrator Guide*
- *Cisco Unified Communications Manager System Guide*
- *Cisco Unified Communications Solution Reference Network Design (SRND) documents*
- *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985 (IEEE, New York)*

Conventions

This manual uses the following conventions.

Convention	Description
boldface font	<p>Boldface font is used to indicate commands, such as user entries, keys, buttons, and folder and submenu names. For example:</p> <ul style="list-style-type: none"> Choose Edit > Find. Click Finish.
<i>italic font</i>	<p><i>Italic</i> font is used to indicate the following:</p> <ul style="list-style-type: none"> To introduce a new term. Example: A <i>skill group</i> is a collection of agents who share similar skills. For emphasis. Example: <i>Do not</i> use the numerical naming convention. An argument for which you must supply values. Example: IF (<i>condition, true-value, false-value</i>) A book title. Example: See the <i>Cisco Unified CCX Installation Guide</i>.
window font	<p>Window font, such as Courier, is used for the following:</p> <ul style="list-style-type: none"> Text as it appears in code or information that the system displays. Example: <code><html><title>Cisco Systems, Inc. </title></html></code> File names. Example: <code>tserver.properties</code>. Directory paths. Example: <code>C:\Program Files\Adobe</code>
string	<p>Nonquoted sets of characters (strings) appear in regular font. Do not use quotation marks around a string or the string will include the quotation marks.</p>
[]	<p>Optional elements appear in square brackets.</p>
{ x y z }	<p>Alternative keywords are grouped in braces and separated by vertical bars.</p>
[x y z]	<p>Optional alternative keywords are grouped in brackets and separated by vertical bars.</p>
< >	<p>Angle brackets are used to indicate the following:</p> <ul style="list-style-type: none"> For arguments where the context does not allow italic, such as ASCII output. A character string that the user enters but that does not appear on the window such as a password.
^	<p>The key labeled Control is represented in screen displays by the symbol ^. For example, the screen instruction to hold down the Control key while you press the D key appears as ^D.</p>

Obtaining Documentation, Obtaining Support, and Security Guidelines

For information on obtaining documentation, obtaining support, security guidelines, and also recommended aliases and general Cisco documents, see the monthly What's New in Cisco Product Documentation, which also lists all new and revised Cisco technical documentation, at:
<http://www.cisco.com/en/US/docs/general/whatsnew/whatsnew.html>

Documentation Feedback

You can provide comments about this document by sending email to the following address:

ccbu_docfeedback@cisco.com

We appreciate your comments.



CHAPTER 1

About the Cisco Unified CCX Expression Language

This chapter section covers the following topics:

- [The Language Purpose, page 1-1](#)
- [How to Access the Language, page 1-1](#)
- [The Language Syntax, page 1-2](#)
- [The Language Classes and Interfaces, page 1-3](#)
- [Language Code Examples, page 1-3](#)
- [Expression Language Terminology, page 1-4](#)
- [Expression Language Operator Summary, page 1-7](#)
- [Operators Used with Prompts and Documents, page 1-8](#)
- [Expression Language Keywords, page 1-14](#)
- [Expression Language Literals, page 1-15](#)
- [Expression Language Data Types, page 1-16](#)
- [The Language Variables, page 1-27](#)
- [About Conversions in the Expression Language, page 1-31](#)

The Language Purpose

The Cisco Unified CCX Expression Language provides a simple, yet powerful, way for script developers to customize steps or variable values within Cisco Unified CCXscripts.

How to Access the Language

The Expression Language can be accessed from the Cisco Unified CCX Step Editor through the Expression Editor responsible for providing parsing, validating, and evaluation of expressions. See [How to Access the Cisco Unified CCX Expression Editor, page 2-1](#), for more details.

The Language Syntax

The syntax for this language started with simple operations such as additions and subtractions in Cisco CRA 2.0.

After Cisco CRA 2.0, the language syntax was enhanced to provide more programming capability by borrowing from the syntax of other languages such as C++, C, and Java. For example, the expression syntax allows for developing more complex expressions that use statements such as for, while, if-then, switch, and try/catch which are popular in many programming languages today.

The Unified CCX 8.0 Expression Language syntax is backwards compatible with the previous releases of the Expression Language.



Note

Although the notation in the Expression Language is identical to Java in many cases, it is different in other cases. Also the Expression Language adds more support for creating literals of complex objects and more operators for these complex objects.

The language was enhanced to support the following:

- All numerical operators
- All boolean operators (?:, |, ||, &, &&, ...)
- Variables manipulation operators (=, +=, -=, ^=, ...)
- All primitive Java data types (void, byte, short, float, double, int, ...)
- All numerical literals (3I, 5L, 6.3E4F, 5.54D, ...)
- All BigInteger and BigDecimal literals (23IB, 45.5ID)
- Hexadecimal numbers (0x2A4, 0x2F44FL, ...)
- Additional prompt operators (substitute prompt, random prompt, day of week prompt, time of day prompt)
- Additional document operators (day of week document, time of day document, ...)
- Additional grammar operators (compound grammars, indexing of compound grammars)
- User document representation
- Customizing some prompt generation (\${23.33}, C[FRF]), ...)
- Typecasting ((int)23.33, ...)
- Block comments and line comments within the expression similar to C++ and Java
- Creating user-specified objects using the new operator (new java.util.Vector(), ...)
- Full array creation and indexing support (new int[] {3, 4}, intVar[2], intVar.length, ...)
- Complex block expression with return statement ({return 5 * 1000L;})
- Full Java-like statement support in complex block expression (if, while, do-while, for, switch, try-catch-finally, throw, break, continue, default, ...)
- Full Java-like support for labels inside complex block expression ({loop: while(true) {while (true) {break loop;}}}, ...)
- Local variable definition inside complex block expression ({int j = 5; return j + 2;}, ...)

The Language Classes and Interfaces

The classes and interfaces of the Expression Language are drawn from the Java 2 platform. By default the Expression Language supports referencing classes and interfaces defined in the Java package `java.lang` directly without the need to specify the name of the package. This is similar to Java programming where this package is implicitly imported.

As opposed to these other programming languages, the Cisco Unified CCX expression language is tailored only to provide a value for a property of a step such as a timeout or a prompt to be played while allowing the freedom for the script designator to bridge into Java-defined classes directly instead of using the original Java steps that used to provide that type of bridge between Cisco Unified CCX scripts and custom Java classes defined by individual users.

The Expression Language contains not only the publicly available Java language classes but also has the following unique language classes:

- Contact
- Currency
- Document
- Grammar
- Prompt
- Session
- Script
- User
- Customer
- POD

See also:

- [Expression Editor Tool Reference Descriptions, page 3-1](#)
- [Expression Language Data Types, page 1-16](#) and [The Language Variables, page 1-27](#)
- “How to Use the Cisco Unified CCX Editor in *Cisco Unified CCX Scripting and Development Series: Volume 1, Getting Started with Scripts*.”
- “Cisco Unified CCX Editor Palette Step Descriptions in *Cisco Unified CCX Scripting and Development Series: Volume 2, Editor Step Reference*.”

Language Code Examples

In [Chapter 3, “Expression Editor Tool Reference Descriptions,”](#) language code examples are provided with example screen shots of the Expression Editor tool tabs. These examples can be used in any expression field of a script. You can also use the `Set` step to test most of the examples presented in this guide.

Expression Language Terminology

The following are some of the common object-oriented programming terms found in the Java programming language and used in the Expression Language.

Table 1-1 Language Terminology

Term	Description
BigDecimal	A decimal number that can be arbitrarily large.
BigInteger	An integer that can be arbitrarily large. It is not limited to the 64 bits available in the long data type.
Class	A data type that defines what a specific type of object is. You associate methods with the data type.
Comments	<p>Everything between /* and */ is ignored by the script parser and is used for entering comments. These comment separators can be spread over more than one line.</p> <p>Everything after // on one line is ignored by the script parser and is used for entering comments on one line.</p>
Constructor	Creates a new instance of a class, an object, and initializes all the fields in that instance. A constructor method has the same name as the class.
Floating-Point numbers	<p>Has a decimal point and a fractional part. They can be positive or negative and come in two sizes:</p> <ul style="list-style-type: none"> float: A four-byte number that can contain values as small as 1.40129846e-45 and as large as 3.40282347e+38 double: An eight-byte number that can contain values as small as 4.940656458412446544e-324 and as large as 1.79769313486231570e+308
Identifiers	<p>Names of variables, methods, classes, packages, and interfaces. Unlike literals, they are not the objects themselves, just ways of referring to them.</p> <p>An identifier is an unlimited-length sequence of letters and digits and underscore, the first of which must be a letter or an underscore. An identifier cannot have the same spelling as a keyword, Boolean literal, day of week literal, gender literal or the null literal.</p> <pre> Identifier: IdentifierChars but not a Keyword or BooleanLiteral or DayOfWeekLiteral or GenderLiteral or NullLiteral IdentifierChars: Letter IdentifierChars LetterOrDigit Letter: any from A to Z any from a to z _ LetterOrDigit: Letter any from 0 to 9 Examples of identifiers are: i3 _i StringMAX_VALUEisLetterOrDigit </pre>

Table 1-1 Language Terminology (continued)

Term	Description
Integers	Whole numbers. They come in four sizes: <ul style="list-style-type: none"> • int: The default type. Takes up to four bytes of memory and can hold numbers between -2,147,483,648 and +2,147,483,647 • long: Takes up to eight bytes of memory and ranges in size from -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 • short: Takes up two bytes of memory and can hold numbers between -32,768 and +32,767 • byte: The shortest integer of all. Is one byte long and ranges in value from -128 to 127.
Keywords	Reserved words that cannot be used by the programmer for variable or method names. Uses the same keywords that Java uses. For a list of these keywords, see http://java.sun.com/docs/books/tutorial/java/nutsandbolts/_keywords.html .
Literal	A value that is written directly into an expression without being stored in a variable first.
Method	A function that all objects in a class can perform. When more than one argument can be passed to a method, the successive arguments are separated by commas. Every method has a return type or else it is called a void method. A void method produces no output, begins with the void keyword and is usually used to notify an object of an event. A method can return only one value.
Object	A specific instance of a class. When you create a new object, you are said to be “instantiating” the class. To create an object, use the keyword new . The new keyword is also called the “construction operator.”
Operator	A symbol that operates on one or more arguments to produce a result.
Package	A group of functionally related classes.
Public	If a field or variable is declared to be public, it can be used by any object.
Remainder or Modulus	The remainder value of a number left over after one number is divided by another. For example, 7 divided by 2 has a remainder or modulus of 1.

Table 1-1 Language Terminology (continued)

Term	Description														
Separators	Help define the structure of your expressions.														
	<table border="1"> <thead> <tr> <th>Separator</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>()</td> <td>Parentheses. Enclose arguments in method definitions and calling. Adjusts precedence in arithmetic expressions. Surrounds cast types and delimits test expression in flow control statements.</td> </tr> <tr> <td>{ }</td> <td>Curly braces. Define a block of statements.</td> </tr> <tr> <td>[]</td> <td>Square brackets. Declare array types and used for referencing array members.</td> </tr> <tr> <td>;</td> <td>Semicolon. Terminates a code statement.</td> </tr> <tr> <td>,</td> <td>Comma. Separates successive identifiers in a variable declaration. Joins statements in the test expression of a for loop.</td> </tr> <tr> <td>.</td> <td>Period. Separates and selects a field or method from an object. Separates package names from class names and subpackages.</td> </tr> </tbody> </table>	Separator	Description	()	Parentheses. Enclose arguments in method definitions and calling. Adjusts precedence in arithmetic expressions. Surrounds cast types and delimits test expression in flow control statements.	{ }	Curly braces. Define a block of statements.	[]	Square brackets. Declare array types and used for referencing array members.	;	Semicolon. Terminates a code statement.	,	Comma. Separates successive identifiers in a variable declaration. Joins statements in the test expression of a for loop.	.	Period. Separates and selects a field or method from an object. Separates package names from class names and subpackages.
Separator	Description														
()	Parentheses. Enclose arguments in method definitions and calling. Adjusts precedence in arithmetic expressions. Surrounds cast types and delimits test expression in flow control statements.														
{ }	Curly braces. Define a block of statements.														
[]	Square brackets. Declare array types and used for referencing array members.														
;	Semicolon. Terminates a code statement.														
,	Comma. Separates successive identifiers in a variable declaration. Joins statements in the test expression of a for loop.														
.	Period. Separates and selects a field or method from an object. Separates package names from class names and subpackages.														
Thread	An independent process. A single program can have many different processes executing independently and continuously.														
Tokens	The smallest items in the language. They consist of keywords, operators, comments, identifiers, separators, white space, and literals.														
Unicode	A two-byte character code set that has characters representing almost all characters in the human alphabets and writing systems around the world, including English, Arabic, and Chinese.														
Variable	Stores a value in a computer storage location. That value can be changed by programs acting on it. Variables defined within a class are member variables or fields of that class.														
White Space	The single space, the horizontal tab, the form feed, the carriage return, and the linefeed characters. Outside of String literals, white space characters are all treated the same. Many white space characters together are also treated the same, as if they were one. They are used to separate tokens and to enhance code legibility for human reading.														

See also:

- Chapter 3, “Expression Editor Tool Reference Descriptions”
- http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html

Expression Language Operator Summary

The following is a summary of the operators in the Java language. These have the same functions in the Expression Language. See the individual Expression Editor tab descriptions for the meanings of the syntax buttons displayed on each tab.

See also:

- For a list of all the operators you can use in the Java language:
http://java.sun.com/docs/books/jls/second_edition/html/lexical.doc.html#230663 .
- For descriptions of how these operators are used:
http://java.sun.com/docs/books/jls/second_edition/html/typesValues.doc.html.

Table 1-2 Arithmetic Operators for Obtaining a Value

Operator	Meaning	Example
+	Addition	6 + 7
-	Subtraction	9 - 6
*	Multiplication	4 * 4
/	Division	9 / 3
%	Modulus (remainder)	12 % 10

Table 1-3 Assignment Operators Used to Store Values in Variables

Operator	Meaning
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x/y
x = y	x = y

Table 1-4 Relational Operators for Returning a Boolean Value

Operator	Meaning	Example
==	True if x Equals y. Otherwise, false.	x == y
!=	True if x is Not Equal to y. Otherwise false.	x != y
<	True if x is less than y. Otherwise, false.	x < y
>	True if x is Greater than y. Otherwise false.	x > y
<=	True if x is less than or equal to y.	x <= y
>=	True if x is Greater than or equal to y.	x >= y

Table 1-5 Bitwise Operators for Integer Comparisons

Operator	Meaning	Example
&	Bitwise AND. Copies a bit to the result if it exists in both operands. Otherwise it returns zero.	$x \& y$
	Bitwise OR. Copies a bit to the result if it exists in either operand. Otherwise it returns 0.	$x y$
^	Bitwise XOR. Copies the bit to the result if it is set in one operand (but not both). Otherwise it returns 0.	$x \wedge y$
<<	Left shift The SHIFT LEFT operator moves the bits to the left in the left operand by the number of bits specified in the right operand, discarding the far left bit, and assigning the right-most bit a value of 0. Each move to the left effectively multiplies operand 1 by 2. The left operands value is moved left or right by the number of bits specified by the right operand.	$x \ll y$
>>	Signed Right shift The SHIFT RIGHT operator moves the bits to the right in the left operand by the number of bits specified in the right operand, discarding the far right bit, and assigning the left-most bit a value of 0. Each move to the right effectively divides the operand on the left in half.	$x \gg y$
>>>	Unsigned (Zero fill) right shift	$x \ggg y$
~	Bitwise complement The COMPLEMENT operator has one operand and is used to invert all of the bits of that operand.	$\sim x$
<<=	Left shift assignment	$x = x \ll y$
>>=	Signed Right shift assignment	$x = x \gg y$
>>>=	Unsigned (Zero fill) right shift assignment	$x = x \ggg y$
?&=?	AND assignment	$x = x \& y$
? =?	OR assignment	$x = x y$
?^=?	XOR assignment	$x = x \wedge y$

Operators Used with Prompts and Documents

In addition to the usual operators used in Java, you should be aware of the following operators that you can use in the Expression Language with prompts and documents:

- [The Prompt Substitution Operator |||, page 1-9](#)

- [Qualifier Operators, page 1-9](#)
- [Additive Operators, page 1-9](#)
- [Escalation Operator ||, page 1-11](#)
- [Compound Assignment Operators, page 1-13](#)

The Prompt Substitution Operator |||

The operator ||| is called the prompt substitution operator. It is used to create a substitute prompt. A substitute prompt is a prompt where the first prompt is queued for playback whenever the substitute prompt is used in a media context. If a failure occurs while attempting to queue the prompt then the substitute is queued instead. For example the main prompt could represent a TTS prompt which in cases where the system has not been installed or licensed with TTS support, one would want to fallback to a pre-recorded prompt. In this case, queuing a TTS prompt would fail and the substitute would be used instead. This operator is not associative.

```
SubstituteExpression:
  PromptExpression ||| PromptExpression
```

Qualifier Operators

Qualifier operators are used to further qualify objects by assigning them new or different properties. Qualified objects can be used just as their normal objects. However, in some cases the qualification applied to an object can be used to determine what kind of container prompt or documents will result from the || operator (see [Prompt Escalation Operator ||, page 1-11](#) and [Document Escalation Operator ||, page 1-12](#)). Only one specific qualifier is not ignored if not used in conjunction with the || operator and that is the language qualification. Prompts or documents can be qualified multiple times through multiple types of qualifications.

All qualifiers have the same precedence and are syntactically left-associative (they group left-to-right). They are defined as:

```
QualifiedExpression:
  QualifiedPromptExpression
  QualifiedDocumentExpression
```

The following sections describe a qualified prompt expression and a qualified document expression:

- [Prompt Escalation Operator ||, page 1-11](#)
- [Document Escalation Operator ||, page 1-12](#)

Additive Operators

The operators + and - are called the additive operators. They have the same precedence and are syntactically left-associative (they group left-to-right).

```
AdditiveExpression:
  MultiplicativeExpression
  AdditiveExpression + MultiplicativeExpression
  AdditiveExpression - MultiplicativeExpression
```

If the type of either operand of a + operator is Prompt, then the operation is prompt concatenation. If either type is String or if both are char, then the operation is string concatenation. If the type of both operands is Document then the operation is a document concatenation.

Otherwise, the type of each of the operands of the + operator must be a primitive numeric type, or a parse-time error occurs.

In every case, the type of each of the operands of the binary - operator must be a primitive numeric type, or a compile-time error occurs.

The following topics describe additive and multiplicative expressions:

- [Document Concatenation Operator +, page 1-10](#)
- [Prompt Concatenation Operator +, page 1-10](#)
- [String Concatenation Operator +, page 1-10](#)

Document Concatenation Operator +

If both operand expressions are of type Document or a java.io.InputStream or a java.io.Reader, then the result is a reference to a newly created Document object that is the concatenation of the two operand documents. The content of the left-hand operand precedes the content of the right-hand operand in the newly created document. The concatenation is low-level and makes no assumptions as to the content of both documents. The resulting content type will be reported the same as the first document that defines a content type.

Prompt Concatenation Operator +

If only one operand expression is of type Prompt, then prompt conversion is performed on the other operand to produce a prompt at run time. The result is a reference to a newly created Prompt object that is the concatenation of the two operand prompts. The content of the left-hand operand precedes the content of the right-hand operand in the newly created prompt.

String Concatenation Operator +

If only one operand expression is of type String, then string conversion is performed on the other operand to produce a string at run time. If both are of type char, then string conversion is performed on both operands to produce strings at run time. The result is a reference to a newly created String object that is the concatenation of the two operand strings. The characters of the left-hand operand precede the characters of the right-hand operand in the newly created string.

Integer/Boolean Conditional-Or Operator ||

This Java operator when used as follows is called the conditional-or operator. This section is included here so that you can understand both the conditional-or operator and the escalation operator, which use the same symbol.

The || operator is like | but evaluates its right-hand operand only if the value of its left-hand operand is false. It is syntactically left-associative (it groups left-to-right). It is fully associative with respect to both side effects and result value; that is, for any expressions a, b, and c, evaluation of the expression ((a)||b)||c produces the same result, with the same side effects occurring in the same order, as evaluation of the expression a)||b)||c).

```
ConditionalOrExpression:
    ConditionalAndExpression
    ConditionalOrExpression || ConditionalAndExpression
```


Each operand of `||` must be of type Boolean, or a parse-time error occurs. The type of a conditional-or expression is always Boolean.

At run time, the left-hand operand expression is evaluated first; if its value is `true`, the value of the conditional-or expression is `true` and the right-hand operand expression is not evaluated. If the value of the left-hand operand is `false`, then the right-hand expression is evaluated and its value becomes the value of the conditional-or expression.

Thus, `||` computes the same result as `|` on Boolean operands. It differs only in that the right-hand operand expression is evaluated conditionally rather than always.

Escalation Operator `||`

When used as follows, the `||` operator is called the escalation operator.

The `||` operator can either be used like `|` or it can be used with Prompt, or Document (or compatible types such as `java.io.InputStream` or `java.io.Reader`) to create an escalating prompt, a day of week prompt, a time of day prompt, a random prompt or an escalating document, a day of week document or a time of day document.

The escalation `||` operator can be used to create escalation prompts/documents, day of week prompts/documents, time of day prompts/documents, time of week prompts/documents or random prompts. It is syntactically left-associative (it groups left-to-right). It is fully associative with respect to both side effects and result value; that is, for any expressions `a`, `b`, and `c`, evaluation of the expression `((a)||b)||c` produces the same result, with the same side effects occurring in the same order, as evaluation of the expression `a)||((b)||c)`.

```
EscalationExpression:
    PromptEscalationExpression
    DocumentEscalationExpression
```

The following topics describe a prompt escalation expression and a document escalation expression:

- [Prompt Escalation Operator `||`, page 1-11](#)
- [Document Escalation Operator `||`, page 1-12](#)

Prompt Escalation Operator `||`

The prompt escalation `||` operator can be used to create escalation prompts, day of week prompts, time of day prompts, time of week prompts or random prompts. If at least one of the operands is a prompt, the other is converted to a prompt according to the rules set forth by [Table 1-7](#) and the result will be a new prompt. It is syntactically left-associative (it groups left-to-right). It is fully associative with respect to both side effects and result value; that is, for any expressions `a`, `b`, and `c`, evaluation of the expression `((a)||b)||c` produces the same result, with the same side effects occurring in the same order, as evaluation of the expression `a)||((b)||c)`.

```
PromptEscalationExpression:
    PromptExpression || PromptEscalationExpression
```

The determination of the type of prompt that will result from this operator depends on how the first two prompt operands (in a sequence of `||` operators) were qualified using the `@` or `%` operators:

1. If both prompt operands are qualified with a time of day and a day of week then the resulting prompt will be a time of week prompt. All remaining operands of subsequent `||` operators are going to be added as prompts for subsequent time of week and must then be qualified with at least both a time of day and a day of week or a parse-time error will occur. Other qualifiers if present will be ignored.

2. Otherwise, if both prompt operands are qualified with a day of week then the resulting prompt will be a day of week prompt. All remaining operands of subsequent `||` operators are going to be added as prompts for subsequent day of week and must then be qualified with at least a day of week or a parse-time error will occur. Other qualifiers if present will be ignored.
3. Otherwise, if both prompt operands are qualified with a time of day then the resulting prompt will be a time of day prompt. All remaining operands of subsequent `||` operators are going to be added as prompts for subsequent time of day and must then be qualified with at least a time of day or a parse-time error will occur. Other qualifiers if present will be ignored.
4. Otherwise, if both prompt operands are qualified with a weight then the resulting prompt will be a random prompt. All remaining operands of subsequent `||` operators are going to be added as additional prompts and must then be qualified with at least a weight or a parse-time error will occur. Other qualifiers if present will be ignored.
5. Otherwise, the resulting prompt will be an escalation prompt. All remaining operands of subsequent `||` operators are going to be added as subsequent escalation and their qualifications will be ignored.

Document Escalation Operator `||`

The document escalation `||` operator can be used to create escalation documents, day of week documents, time of day documents, or time of week documents. If at least one of the operands is a document or a `java.io.InputStream` or a `java.io.Reader`, the other is converted to a document. This conversion only supports the prompt conversion as seen in [Prompt Conversions, page 1-32](#). The result of this operator will be a new document. It is syntactically left-associative (it groups left-to-right). It is fully associative with respect to both side effects and result value; that is, for any expressions `a`, `b`, and `c`, evaluation of the expression `((a)||b))||c` produces the same result, with the same side effects occurring in the same order, as evaluation of the expression `a)||((b)||c)`.

```
DocumentEscalationExpression:
    DocumentExpression || DocumentEscalationExpression
    InputStreamExpression || DocumentEscalationExpression
    ReaderExpression || DocumentEscalationExpression
```

The determination of the type of document that will result from this operator depend on how the first two document operands (in a sequence of `||` operators) were qualified using the `@` operator:

1. If both document operands are qualified with a time of day and a day of week then the resulting document will be a time of week document. All remaining operands of subsequent `||` operators are going to be added as documents for subsequent time of week and must then be qualified with at least both a time of day and a day of week or a parse-time error will occur. Other qualifiers if present will be ignored.
2. Otherwise, if both document operands are qualified with a day of week then the resulting document will be a day of week document. All remaining operands of subsequent `||` operators are going to be added as documents for subsequent day of week and must then be qualified with at least a day of week or a parse-time error will occur. Other qualifiers if present will be ignored.
3. Otherwise, if both document operands are qualified with a time of day then the resulting document will be a time of day document. All remaining operands of subsequent `||` operators are going to be added as documents for subsequent time of day and must then be qualified with at least a time of day or a parse-time error will occur. Other qualifiers if present will be ignored.
4. Otherwise, the resulting document will be an escalation document. All remaining operands of subsequent `||` operators are going to be added as subsequent escalation and their qualifications will be ignored.

Compound Assignment Operators

All compound assignment operators require both operands to be of primitive type, except for `+=`, which allows the right-hand operand to be of any type if the left-hand operand is of type `String` or of one of the type specified in [Table 1-7](#) if the left-hand operand is of type `Prompt`.

A compound assignment expression of the form `E1 op= E2` is equivalent to `E1 = (T)((E1) op (E2))`, where `T` is the type of `E1`, except that `E1` is evaluated only once.

**Note**

The implied cast to type `T` may be either an identity conversion or a narrowing primitive conversion.

For example, the following code is correct.

```
short x = 3;
x += 4.6;
```

and results in `x` having the value `7` because it is equivalent to:

```
short x = 3;
x = (short)(x + 4.6);
```

At run time, the expression is evaluated in one of two ways. If the left-hand operand expression is not an array access expression, then four steps are required:

1. First, the left-hand operand is evaluated to produce a variable. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason; the right-hand operand is not evaluated and no assignment occurs.
2. Otherwise, the value of the left-hand operand is saved and then the right-hand operand is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.
3. Otherwise, the saved value of the left-hand variable and the value of the right-hand operand are used to perform the binary operation indicated by the compound assignment operator. If this operation completes abruptly (the only possibility is an integer division by zero), then the assignment expression completes abruptly for the same reason and no assignment occurs.
4. Otherwise, the result of the binary operation is converted to the type of the left-hand variable and the result of the conversion is stored into the variable.

If the left-hand operand expression is an array access expression, then many steps are required:

1. First, the array reference subexpression of the left-hand operand array access expression is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason; the index subexpression (of the left-hand operand array access expression) and the right-hand operand are not evaluated and no assignment occurs.
2. Otherwise, the index subexpression of the left-hand operand array access expression is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and the right-hand operand is not evaluated and no assignment occurs.
3. Otherwise, if the value of the array reference subexpression is null, then no assignment occurs and a `NullPointerException` is thrown.
4. Otherwise, the value of the array reference subexpression indeed refers to an array. If the value of the index subexpression is less than zero, or greater than or equal to the length of the array, then no assignment occurs and an `ArrayIndexOutOfBoundsException` is thrown.

5. Otherwise, the value of the index subexpression is used to select a component of the array referred to by the value of the array reference subexpression. The value of this component is saved and then the right-hand operand is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs. (For a simple assignment operator, the evaluation of the right-hand operand occurs before the checks of the array reference subexpression and the index subexpression, but for a compound assignment operator, the evaluation of the right-hand operand occurs after these checks.)
6. Otherwise, consider the array component selected in the previous step, whose value was saved. This component is a variable; call its type S. Also, let T be the type of the left-hand operand of the assignment operator as determined at parse time.
 - If T is a primitive type, then S is necessarily the same as T.
 - The saved value of the array component and the value of the right-hand operand are used to perform the binary operation indicated by the compound assignment operator. If this operation completes abruptly (the only possibility is an integer division by zero), then the assignment expression completes abruptly for the same reason and no assignment occurs.
 - Otherwise, the result of the binary operation is converted to the type of the selected array component and the result of the conversion is stored into the array component.
 - If T is a reference type, then it must be String or a Prompt. Because class String is a final class, S must also be String if this is a string compound assignment operator. If it is a prompt compound assignment operator then the S can be assignment compatible with the Prompt reference type. Therefore the run-time check that is sometimes required for the simple assignment operator is never required for a compound assignment operator.
 - The saved value of the array component and the value of the right-hand operand are used to perform the binary operation (string or prompt concatenation) indicated by the compound assignment operator (which is necessarily +=). If this operation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.
7. Otherwise, the String, Prompt, or Document result of the binary operation is stored into the array component.

Expression Language Keywords

The following character sequences, formed from ASCII letters, are reserved for use as keywords and cannot be used as identifiers:

```
Keyword: one of
abstractdefaultifpackagesynchronized
booleandoimplementsprivatethis
breakdoubleimportprotectedthrow
byteelseinstanceofpublicthrows
caseextendsintreturn transient
catchfinalinterfaceshorttry
charfinallyinterruptiblestaticuninterruptible
classfloatlongstrictfpvoid
constfor nativesupervolatile
continuegotonewswitchwhile
```

Some of these keywords, though not currently used, are reserved nevertheless.

While true and false might appear to be keywords, they are technically Boolean literals. Similarly, while null might appear to be a keyword, it is technically the null literal. And the same is true for sun, mon, tue, wed, thu, fri, sat which are technically DayOfWeek literals and neutral, male, female which are Gender literals.

Expression Language Literals

This topic includes:

- [Lexical Literals, page 1-15](#)
- [Class Literals, page 1-15](#)
- [Complex Literals, page 1-16](#)

See [Expression Editor Tool Reference Descriptions](#) for further descriptions of the literals used in the Expression Language.

Lexical Literals

A literal denotes a fixed, unchanging value.

```
Literal:  
IntegerLiteral  
FloatingPointLiteral  
BooleanLiteral  
CharacterLiteral  
StringLiteral  
DayOfWeekLiteral  
GenderLiteral  
NullLiteral
```

The type of a literal is determined as follows:

- The type of an integer literal that ends with I or i is int, that ends with L or l is long and that ends with IB or ib is BigInteger; the type of any other integer literal is int.
- The type of a floating-point literal that ends with F or f is float. The type of a floating-point literal that ends with D or d is double. The type of a floating-point literal that ends with BF or bf is BigDecimal. The type of any other floating-point literal is double.
- The type of a boolean literal is boolean.
- The type of a character literal is char.
- The type of a string literal is String.
- The type of the null literal null is the null type; its value is the null reference.

Evaluation of a lexical literal always completes normally.

Class Literals

A class literal is an expression consisting of the name of a class, interface, array, or primitive type followed by a `.' and the token class. The type of a class literal is Class. It evaluates to the Class object for the named type (or for void) as defined by the defining class loader of the class of the current instance.

Complex Literals

A complex literal is the source code representation of a value of a Currency, Date, Document, Grammar, Language, Prompt, User, Script, or Time. For examples of complex literals, see the example literals in the descriptions of Currency, Date, Document, Grammar, Language, Prompt, User, Script, or Time data types.

```
ComplexLiteral:
  CurrencyLiteral
  DateLiteral
  DocumentLiteral
  GrammarLiteral
  LanguageLiteral
  PromptLiteral
  UserLiteral
  ScriptLiteral
  TimeLiteral
```

Prompt complex literals are of type Prompt. Currency complex literals are of type Currency, Date complex literal are of type Date. Document complex literals are of type Document. Grammar complex literals are of type Grammar. Language complex literals are of type Language. Prompt complex literals are of type Prompt. User complex literals are of type User. Script complex literals are of type Script. Time complex literals are of type Time.

Expression Language Data Types

The Java programming language on which the Expression Language is based is a strongly typed language, which means that every variable and every expression has a type that is known at parse time.

Types:

- Limit the values that a variable can hold or that an expression can produce.
- Limit the operations supported on those values.
- Determine the meaning of the operations.

Strong typing helps detect errors at parse time.

The types of the Expression Language are divided into two basic categories:

- **Primitive** types: The Boolean type and the numeric types. The numeric types are the integral types:
 - byte
 - short
 - int
 - long
 - BigInteger
 - char
 - floating-point types: float, double, and BigDecimal
- **Reference** types:
 - class types
 - interface types

- array types
- a special null type.

An object is a dynamically created instance of a class type or a dynamically created array. The values of a reference type are references to objects. All objects, including arrays, support the methods of class `Object`. String literals are represented by `String` objects.

The Cisco Unified CCX Expression Language also includes the **friendly** data type. A friendly data type is the Cisco Unified CCX data type that is the equivalent of a fully qualified Java class name, that is the Java data name and the package in which it is included. Friendly data types are either primitive or reference types. For more information on friendly data types, see [Friendly Data Types](#).

Names of types or friendly type names are used in declarations, casts, array creation expressions, class literals, and instanceof operator expressions.

This section includes the following topics:

- [Type Variables, page 1-17](#)
- [Type Values, page 1-17](#)
- [Primitive Values, page 1-18](#)
- [Reference Values, page 1-24](#)
- [Where Types Are Used, page 1-27](#)

See also:

- [Expression Editor Tool Reference Descriptions, page 3-1](#) for descriptions and examples of the data classes and literals you can use in the Expression Language.

Type Variables

A variable is always of a type and is a storage location for that type:

- A variable of a primitive type always holds a value of that exact type or a null reference.
- A variable of a class type `X` can hold a null reference or a reference to an instance of class `X` or of any class that is a subclass of `X`.
- A variable of an interface type can hold a null reference or a reference to any instance of any class that implements the interface.

Type Values

Since the Expression Language has two basic types (primitive and reference), it also has two kinds of data values that can be stored in variables, passed as arguments, returned by methods, and operated on: primitive and reference values.

There is also a special null type, the type of the null expression, which has no name. Because the null type has no name, it is impossible to declare a variable of the null type or to cast to the null type. The null reference is the only possible value of an expression of null type. The null reference can always be cast to any reference type. In practice, you can ignore the null type and consider that null is merely a special literal that can be of any reference type.

This section includes the following topics:

- [Integral Values](#)

- [Integer Operations](#)
- [Floating-Point Values](#)
- [Floating-Point Operations](#)
- [Boolean Values](#)

Primitive Values

A primitive type is predefined by the Expression Language and named by its reserved keyword:

```
PrimitiveType:
  NumericType
  Boolean
NumericType:
  IntegralType
  FloatingPointType
IntegralType: one of
  byte short int long BigInteger char
FloatingPointType: one of
  float double BigDecimal
```

Primitive values do not share state with other primitive values. A variable whose type is a primitive type always holds a primitive value of that same type or null. The value of a variable of primitive type can be changed only by assignment operations on that variable:

- The numeric types are the integral types and the floating-point types.
- The integral types are byte, short, int, long, and BigInteger, whose values are 8-bit, 16-bit, 32-bit, 64-bit and unlimited signed two's-complement integers, respectively, and char, whose values are 16-bit unsigned integers representing Unicode characters.
- The floating-point types are float, whose values include the 32-bit IEEE 754 floating-point numbers, and double, whose values include the 64-bit IEEE 754 floating-point numbers, and BigDecimal, whose value include an unlimited size floating-point numbers.
- The Boolean type has exactly two values: true and false.

As opposed to the Java programming language where primitive types are not considered reference types, the Expression Language treats them the same. That means the reference type Integer and the primitive data type int are one and the same. All primitive data types share the same features as a reference type. So although an attempt was made in this chapter to separate them, primitive data types should be considered as reference data types as well.

Integral Values

The values of the integral types are integers in the following ranges:

- For byte, from -128 to 127, inclusive.
- For short, from -32768 to 32767, inclusive.
- For int, from -2147483648 to 2147483647, inclusive.
- For long, from -9223372036854775808 to 9223372036854775807, inclusive.
- For BigInteger, no limits.
- For char, from '\u0000' to '\uffff' inclusive.

Integer Operations

The Expression Language provides a number of operators that act on integral values:

- The comparison operators, which result in a value of type `Boolean`:
 - Numerical Comparison Operators (<, <=, >, and >=).
 - Numeric Equality Operators (== and !=).
- The numerical operators, which result in a value of type `int` or `long` or `BigInteger`:
 - Unary Plus Operator (+) and Unary Minus Operator (-).
 - Multiplicative Operators (*, /, and %).
 - Additive Operators (+ and -) for Numeric Types.
 - Prefix Increment Operator (++) and Postfix Increment Operator (++).
 - Prefix Decrement Operator (--) and Postfix Decrement Operator (--).
 - Shift Operators (<<, >>, and >>>).
 - Bitwise Complement Operator (~).
 - Integer Bitwise Operators (&, ^, and |).
- Conditional Operator (? :)
- Field Access Using a Primary.
- Method Invocation Expressions.
- The cast operator, which can convert from an integral value or a string value to a value of any specified numeric type.
- The String Concatenation Operator +, which, when given a String operand and an integral operand, will convert the integral operand to a String representing its value in decimal form, and then produces a newly created String that is the concatenation of the two strings.
- The prompt concatenation operator +, which, when given a Prompt operand and an integral operand, will convert the integral operand to a Prompt representing its value in spoken form, and then produce a newly created Prompt that is the concatenation of the two prompts.

Except for the prompt concatenation operator, these operations are the same as those in Java. For descriptions of the operations you can have on expressions, see:

http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#44393.

Other useful constructors, methods, and constants are predefined in the classes `Byte`, `Short`, `Integer`, `Long`, `BigInteger`, and `Character`.

If an integer operator other than a shift operator has at least one operand of type `BigInteger`, then the operation is carried out using `BigInteger` precision and the result of the numerical operator is of type `BigInteger`.

If at least one operand is of type `long`, then the operation is carried out using 64-bit precision, and the result of the numerical operator is of type `long`. If the other operand is not `long` or `BigInteger`, it is first widened to type `long` or `BigInteger` by numeric promotion. Otherwise, the operation is carried out using 32-bit precision, and the result of the numerical operator is of type `int`. If either operand is not an `int`, it is first widened to type `int` by numeric promotion.

The built-in integer operators do not indicate overflow or underflow in any way. The only numeric operators that can throw an exception are the integer divide operator `/` and the integer remainder operator `%`, which throw an `ArithmeticException` in the complex expression block or an `ExpressionArithmeticException` in the script if the right-hand operand is zero.

The example:

```
{
  Prompt p = null;
  int i = 1000000;

  p = N[i * i];
  long l = i;

  p += N[l * l];
  p += [20296 / (l - i)];
  return p;
}
```

would create a concatenated prompt that would playback the spoken representation of -727379968 and 100000000000 and then encounters an ArithmeticException in the division by $l - i$, because $l - i$ is zero. The first multiplication is performed in 32-bit precision, whereas the second multiplication is a long multiplication. The value -727379968 is the decimal value of the low 32 bits of the mathematical result, 100000000000, which is a value too large for type int.

Any value of any integral type may be cast to or from any numeric type and from the String type in which case an ExpressionTargetException with a nested NumberFormatException is thrown back if the string value cannot be properly parsed into an integral type. There are no casts between integral types and the type Boolean.

Floating-Point Values

The floating-point types are float, double, and BigDecimal, which are conceptually associated with the single-precision 32-bit, double-precision 64-bit format IEEE 754 and arbitrary-precision signed decimal values and operations as specified in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985 (IEEE, New York).

The IEEE 754 standard includes not only positive and negative numbers that consist of a sign and magnitude, but also positive and negative zeros, positive and negative infinities, and special Not-a-Number values (hereafter abbreviated NaN). A NaN value is used to represent the result of certain invalid operations such as dividing zero by zero. NaN constants of both float and double type are predefined as Float.NaN and Double.NaN.

The finite nonzero values of any floating-point value set can all be expressed in the form $s \cdot m \cdot 2^{[e-N+1]}$, where s is +1 or -1, m is a positive integer less than 2^N , and e is an integer between $E_{\min} = -(2^{K-1}-2)$ and $E_{\max} = 2^{K-1}-1$, inclusive, and where N and K are parameters that depend on the value set. Some values can be represented in this form in more than one way; for example, suppose that a value v in a value set might be represented in this form using certain values for s , m , and e , then if it happened that m were even and e were less than 2^{K-1} , one could halve m and increase e by 1 to produce a second representation for the same value v . A representation in this form is called *normalized* if $m \geq 2^{[N-1]}$; otherwise the representation is said to be *denormalized*. If a value in a value set cannot be represented in such a way that $m \geq 2^{[N-1]}$, then the value is said to be a *denormalized value*, because it has no normalized representation.

The constraints on the parameters N and K (and on the derived parameters E_{\min} and E_{\max}) for the two floating-point value sets are summarized in [Table 1-6](#).

Table 1-6 Floating-Point Limit Value Sets

Parameter	Float	Double
N	24	53
K	8	11
E _{max}	+127	+1023
E _{min}	-126	-1022

Each of the two value sets includes not only the finite nonzero values that are ascribed to it above, but also NaN values and the four values positive zero, negative zero, positive infinity, and negative infinity.

The elements of the float value set are exactly the values that can be represented using the single floating-point format defined in the IEEE 754 standard. The elements of the double value set are exactly the values that can be represented using the double floating-point format defined in the IEEE 754 standard.

Except for NaN, floating-point values are *ordered*; arranged from smallest to largest, they are negative infinity, negative finite nonzero values, positive and negative zero, positive finite nonzero values, and positive infinity.

Positive zero and negative zero compare equal; thus the result of the expression `0.0== -0.0` is `true` and the result of `0.0> -0.0` is `false`. But other operations can distinguish positive and negative zero; for example, `1.0/0.0` has the value positive infinity, while the value of `1.0/-0.0` is negative infinity.

NaN is *unordered*, so the numerical comparison operators `<`, `<=`, `>`, and `>=` return `false` if either or both operands are NaN. The equality operator `==` returns `false` if either operand is NaN, and the inequality operator `!=` returns `true` if either operand is NaN. In particular, `x!=x` is `true` if and only if `x` is NaN, and `(x<y) == !(x>=y)` is `false` if `x` or `y` is NaN.

Any value of a floating-point type may be cast to or from any numeric type and from the String type in which case an `ExpressionTargetException` with a nested `NumberFormatException` is thrown back if the string value cannot be properly parsed into a floating-point type. There are no casts between floating-point types and the type `Boolean`.

Floating-Point Operations

The Expression Language provides a number of operators that act on floating-point values:

- The comparison operators, which result in a value of type `Boolean`:
 - Numerical Comparison Operators (`<`, `<=`, `>`, and `>=`).
 - Numeric Equality Operators (`==` and `!=`).
- The numerical operators, which result in a value of type `float` or `double` or `BigDecimal`:
 - Unary Plus Operator (`+`) and Unary Minus Operator (`-`).
 - Multiplicative Operators (`*` ? `%`).
 - Additive Operators (`+` and `-`).
 - Postfix Increment Operator `++` and Prefix Increment Operator (`++`).
 - Postfix Decrement Operator `--` and Prefix Decrement Operator (`--`).
- Conditional Operator (`? :`)
- Field access, using either a qualified name or a field access expression.

- Method invocation.
- The cast operator, which can convert from a floating-point value to a value of any specified numeric type.
- The string concatenation operator `+`, which, when given a `String` operand and a floating-point operand, will convert the floating-point operand to a `String` representing its value in decimal form (without information loss), and then produce a newly created `String` by concatenating the two strings.
- The prompt concatenation operator `+`, which, when given a `Prompt` operand and a floating-point operand, will convert the floating-point operand to a `Prompt` representing its value in spoken form, and then produce a newly created `Prompt` that is the concatenation of the two prompts.

Except for the prompt concatenation operator, these operations are the same as those in Java. For descriptions of the operations you can have on expressions, see:

http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#44393.

Other useful constructors, methods, and constants are predefined in the classes `Float`, `Double`, `BigDecimal`, and `Math`.

If at least one of the operands to a binary operator is of floating-point type, then the operation is a floating-point operation, even if the other is integral.

If at least one of the operands to a numerical operator is of type `BigDecimal`, then the operation is carried out using arbitrary floating-point arithmetic, and the result of the numerical operator is a value of type `BigDecimal`. If the other operand is not a `BigDecimal`, it is first widened to type `double` by numeric promotion. If at least one of the operands to a numerical operator is of type `double`, then the operation is carried out using 64-bit floating-point arithmetic, and the result of the numerical operator is a value of type `double`. (If the other operand is not a `double`, it is first widened to type `double` by numeric promotion.) Otherwise, the operation is carried out using 32-bit floating-point arithmetic, and the result of the numerical operator is a value of type `float`. If the other operand is not a `float`, it is first widened to type `float` by numeric promotion.

Operators on floating-point numbers behave as specified by IEEE 754 (with the exception of the remainder operator). The Expression Language requires support of IEEE 754 *denormalized* floating-point numbers and *gradual underflow*, which make it easier to prove the properties of some numerical algorithms. Floating-point operations do not "flush to zero" if the calculated result is a denormalized number.

The Java programming language requires that floating-point arithmetic behave as if every floating-point operator rounded its floating-point result to the result precision. *Inexact* results must be rounded to the representable value nearest to the infinitely precise result; if the two nearest representable values are equally near, the one with its least significant bit zero is chosen. This is the IEEE 754 standard's default rounding mode known as *round to nearest*.

The language uses *round toward zero* when converting a floating value to an integer, which acts, in this case, as though the number were truncated, discarding the mantissa bits. Rounding toward zero chooses as its result the format's value closest to and no greater in magnitude than the infinitely precise result.

Floating-point operators produce no exceptions. An operation that overflows produces a signed infinity, an operation that underflows produces a denormalized value or a signed zero, and an operation that has no mathematically definite result produces NaN. All numeric operations with NaN as an operand produce NaN as a result. As has already been described, NaN is unordered, so a numeric comparison operation involving one or two NaNs returns `false` and any `!=` comparison involving NaN returns `true`, including `x!=x` when `x` is NaN.

The example expression:

```
{
    // An example of overflow:
```

```

double d = 1e308;

System.out.print("overflow produces infinity: ");
System.out.println(d + "*10==" + d*10);
// An example of gradual underflow:
d = 1e-305 * Math.PI;
System.out.print("gradual underflow: " + d + "\n      ");
for (int i = 0; i < 4; i++) {
    System.out.print(" " + (d /= 100000));
}
System.out.println();
// An example of NaN:
System.out.print("0.0/0.0 is Not-a-Number: ");
d = 0.0/0.0;
System.out.println(d);
// An example of inexact results and rounding:
System.out.print("inexact results with float:");
for (int i = 0; i < 100; i++) {
    float z = 1.0f / i;

    if (z * i != 1.0f) {
        System.out.print(" " + i);
    }
}
System.out.println();
// Another example of inexact results and rounding:
System.out.print("inexact results with double:");
for (int i = 0; i < 100; i++) {
    double z = 1.0 / i;

    if (z * i != 1.0) {
        System.out.print(" " + i);
    }
}
System.out.println();
// An example of cast to integer rounding:
System.out.print("cast to int rounds toward 0: ");
d = 12345.6;
System.out.println((int)d + " " + (int)(-d));
return null;
}

```

produces the output:

```

overflow produces infinity: 1.0e+308*10==Infinity
gradual underflow: 3.141592653589793E-305
    3.1415926535898E-310 3.141592653E-315 3.142E-320 0.0
0.0/0.0 is Not-a-Number: NaN
inexact results with float: 0 41 47 55 61 82 83 94 97
inexact results with double: 0 49 98
cast to int rounds toward 0: 12345 -12345

```

This example demonstrates, among other things, that gradual underflow can result in a gradual loss of precision. The results when *i* is 0 involve division by zero, so that *z* becomes positive infinity, and *z* * 0 is NaN, which is not equal to 1.0.

Boolean Values

The Boolean type represents a logical quantity with two possible values, indicated by the case insensitive literals true and false.

The Boolean operators are:

- Boolean Equality Operators (== and !=)
- Logical Complement Operator (!)
- Boolean Logical Operators (&, ^, and |)
- Conditional-And Operator (&&) and Integer/Boolean Conditional-Or Operator (||), and Escalation Operator (||)
- Conditional Operator (? :)
- Field access, using either a qualified name or a field access expression
- Method invocation
- The String Concatenation Operator (+) which, when given a String operand and a Boolean operand, will convert the Boolean operand to a String (either "true" or "false"), and then produce a newly created String that is the concatenation of the two strings

These operations are the same as those in Java. For descriptions of the operations you can have on expressions, see: http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#44393.

Boolean expressions determine the control flow in several kinds of statements:

- if Statement
- while Statement
- do Statement
- for Statement

A Boolean expression also determines which subexpression is evaluated in the conditional ?: operator.

Only Boolean expressions can be used in control flow statements and as the first operand of the conditional operator ? :. An integer *x* can be converted to a Boolean, following the C language convention that any nonzero value is true, by the expression *x!=0*. An object reference *obj* can be converted to a Boolean, following the C language convention that any reference other than null is true, by the expression *obj!=null*.

A cast of a Boolean value to type Boolean is allowed; no other casts on type Boolean are allowed. A Boolean can be converted to a string by string conversion.

Reference Values

The section includes the following topics:

- [About Reference Values](#)
- [The Object Class](#)
- [The Currency Class](#)
- [The Date Class](#)
- [The Document Class](#)
- [The Grammar Class](#)
- [The Language Class](#)
- [The Prompt Class](#)
- [The String Class](#)
- [The Script Class](#)

- [The User Class](#)
- [The Customer Class](#)
- [The POD Class](#)
- [Where Types Are Used](#)

About Reference Values

There are three kinds of reference types:

- class
- interface
- array

```
ReferenceType:
    ClassOrInterfaceType
    ArrayType
ClassOrInterfaceType:
    ClassType
    InterfaceType
ClassType:
    TypeName
InterfaceType:
    TypeName
ArrayType:
    Type [ ]
```

The Object Class

The Object class is a super class of all other classes. A variable of type Object can hold a reference to any object, whether it is an instance of a class or an array. All class and array types inherit the methods of class Object.

The Currency Class

Instances of the Currency class represent currency designators. A Currency object has a constant (unchanging) value. Complex currency literals are references to instances of class Currency.

The Date Class

Instances of the Date class represent currency designators. A Date object has a constant (unchanging) value. Complex date literals are references to instances of class Date.

The Document Class

Instances of the Document class represent documents located somewhere that can be accessed for various reasons. A Document object has a constant (unchanging) value. Complex document literals are references to instances of class Document.

The document concatenation operator +, the time of day document, time of week document, and day of week document operators || implicitly create a new Document object.

The Grammar Class

Instances of the Grammar class represent grammars located somewhere that can be accessed for various reasons. A Grammar object has a constant (unchanging) value. Complex grammar literals are references to instances of class Grammar.

The compound grammar operator `||` implicitly creates a new Grammar object.

The Language Class

Instances of the Language class represent language designators. A Language object has a constant (unchanging) value. Complex language literals are references to instances of class Language.

The Prompt Class

Instances of the Prompt class represent audio data that can be played back to a caller. A Prompt object has a constant (unchanging) value. Complex prompt literals are references to instances of class Prompt.

The prompt concatenation operator `+`, the prompt escalation, time of day prompt, time of week prompt, day of week prompt, and random prompt operators `||`, and the prompt substitution operator `|||` implicitly create a new Prompt object.

The String Class

Instances of the String class represent sequences of Unicode characters. A String object has a constant (unchanging) value. String literals are references to instances of class String.

The string concatenation operator `+` implicitly creates a new String object.

The Script Class

Instances of the Script class represents scripts designed using the Cisco Unified CCX Script Editor and located somewhere accessible by the Cisco Unified CCX Engine such as the script repository, local or network disks, and a web server. A Script object has a constant (unchanging) value. Script literals are references to instances of class Script. Complex Script literals are references to instances of class Script.

The Time Class

Instances of the Time class represent currency designators. A Time object has a constant (unchanging) value. Complex time literals are references to instances of class Time.

The User Class

Instances of the User class represent a user configured in the Cisco Call Manager. It can represent a normal user, a Cisco Unified CCX Agent, a Cisco Unified CCX Supervisor and/or a Cisco Unified CCX Administrator. A User object has a constant (unchanging) value. Complex user literals are references to instances of class User.

The Customer Class

Instances of the Customer class represent an Organization's customer when used with reference to Context Service data, for instance the caller's data. Here, an Organization represents Cisco's customer.

The POD Class

Instances of the POD class represent a piece of data that presents an activity between the Organization and its customer. POD is referred to as an activity in the Finesse UI. Here, the Organization represents Cisco's customer.

Where Types Are Used

Types are used when they appear in declarations or in certain expressions. They can be found in expressions of the following kinds:

- Array Creation Expressions
- Cast Expressions
- Type Comparison Operator `instanceof`

The Language Variables

This topic covers the following:

- [About Language Variables, page 1-27](#)
- [Primitive Variables, page 1-28](#)
- [Reference Variables, page 1-28](#)
- [Variable Categories, page 1-28](#)
- [“final” Variables, page 1-29](#)
- [Initial Values of Variables, page 1-29](#)
- [Definite Local Variable Assignment, page 1-30](#)
- [Variable Types, Classes, and Interfaces, page 1-30](#)

About Language Variables

A variable is a storage location and has an associated type, sometimes called its *parse-time type*, which is either a primitive type or a reference type. A variable always contains a value that is assignment compatible with its type. A variable's value is changed by an assignment or by a prefix or postfix `++` (increment) or `--` (decrement) operator.

Compatibility of the value of a variable with its type is guaranteed by the Expression Language. Default values are compatible and all assignments to a variable are checked for assignment compatibility usually at parse time, but, in a single case involving arrays, a run-time check is made.

See also:

- “The Variable Pane” in the *Cisco Unified CCX Scripting and Development Series: Volume 1, Getting Started with Scripts* for how to use script variables.
- “How and When To Configure the Encoding and Decoding of Variable Types” in the *Cisco Unified CCX Scripting and Development Series: Volume 1, Getting Started with Scripts* for how variables of different data types are converted to the appropriate system type when transferred between systems in an enterprise configuration.

Primitive Variables

A variable of a primitive type always holds a value of that exact primitive type or a `null` reference.

Reference Variables

A variable of reference type can hold either of the following:

- A null reference
- A reference to any object whose class is assignment compatible with that type of the variable

Variable Categories

There are five kinds of variables:

- **Script variables.** These are defined in the Cisco Unified CCX Script Editor and are accessible from within expressions (except for prompt and grammar templates). They exist for the life of the script.
- **Array components.** These are unnamed variables that are created and initialized to default values whenever a new object that is an array is created. The array components effectively cease to exist when the array is no longer referenced.
- **Complex expression block parameters.** These are name argument values passed to a complex block expression. For every parameter declared in a method declaration, a new parameter variable is created each time that method expression is evaluated. The new variable is initialized with the corresponding argument value from the expression invocation. The block parameter effectively ceases to exist when the execution of the body of the block expression is complete.
- An **exception-handler parameter.** This is created each time an exception is caught by a `catch` clause of a `try` statement. The new variable is initialized with the actual object associated with the exception. The exception-handler parameter effectively ceases to exist when execution of the block associated with the `catch` clause is complete.
- **Local variables.** These are declared by local variable declaration statements. Whenever the flow of control enters a block or `for` statement, a new variable is created for each local variable declared in a local variable declaration statement immediately contained within that block or `for` statement. A local variable declaration statement may contain an expression which initializes the variable. The local variable with an initializing expression is not initialized, however, until the local variable declaration statement that declares it is executed. The rules of definite assignment prevent the value of a local variable from being used before it has been initialized or otherwise assigned a value. The local variable effectively ceases to exist when the execution of the block or `for` statement is complete.

Were it not for one exceptional situation, a local variable could always be regarded as being created when its local variable declaration statement is executed. The exceptional situation involves the `switch` statement, where it is possible for control to enter a block but bypass execution of a local variable declaration statement. Because of the restrictions imposed by the rules of definite assignment, however, the local variable declared by such a bypassed local variable declaration statement cannot be used before it has been definitely assigned a value by an assignment expression.

The following example contains several different kinds of variables:

```
(int count) { // Complex Expression Block variable
    int x, y; // x and y are local variables
    int[] w = new int[10]; // w[0] is an array component
}
```

“final” Variables

A variable can be declared `final`. A `final` variable may only be assigned to once. It is a parse time error if a `final` variable is assigned to unless it is definitely unassigned immediately prior to the assignment.

Once a `final` variable has been assigned, it always contains the same value. If a `final` variable holds a reference to an object, then the state of the object may be changed by operations on the object, but the variable will always refer to the same object. This applies also to arrays, because arrays are objects; if a `final` variable holds a reference to an array, then the components of the array may be changed by operations on the array, but the variable will always refer to the same array.

Declaring a variable `final` can serve as useful documentation that its value will not change and can help avoid programming errors.

In the example:

```
(String s) {
    final int max_size = 23;
    if (s.length() < max_size) {
        return s;
    } else {
        return s.substring(0, max_size);
    }
}
```

the variable `max_size` is declared `final` and holds the value 23. The value of the variable `max_size` can never change, so it always refers to the same size, the one created by its initializer.

Initial Values of Variables

Every variable in a program must have a value before its value is used:

- Each array component is initialized with a default value when it is created. For type:
 - byte, the default value is zero, that is, the value of `(byte)0`.
 - short, the default value is zero, that is, the value of `(short)0`.
 - int, the default value is zero, that is, `0`.
 - long, the default value is zero, that is, `0L`.
 - `BigInteger`, the default value is zero, that is, `0IB`.
 - float, the default value is positive zero, that is, `0.0f`.
 - double, the default value is positive zero, that is, `0.0d`.

- BigDecimal, the default value is positive zero, that is, 0.0fb.
- char, the default value is the null character, that is, '\u0000'.
- Boolean, the default value is false.
- String, the default value is the empty string, that is, "".
- Prompt, the default value is the empty prompt, that is, P[[]].
- Grammar, the default value is the empty grammar, that is, G[[]].
- Document, the default value is the empty document, that is, DOC[[]].
- Date, the default value is the current date at the time of interpretation.
- Time, the default value is the current time at the time of interpretation.
- Language, the default value is the system default language.
- Currency, the default value is the system default currency.

For all other reference types, the default value is null.

- Each complex block expression argument is initialized to the corresponding argument value provided by the invoker of the expression.
- An exception-handler parameter is initialized to the thrown object representing the exception and throw statements.
- A local variable and the for statement must be explicitly given a value before they are used, by either initialization or assignment, in a way that can be verified by the parser using the rules for definite assignment.

Definite Local Variable Assignment

Each local variable must have a *definitely assigned* value when any access of its value occurs. If accessed before its value is initialized, a run-time error is generated.

Similarly, every blank final variable must be assigned at most once; it must be *definitely unassigned* when an assignment to it occurs otherwise a run-time error occurs.

Variable Types, Classes, and Interfaces

In the Expression Language, every variable and every expression has a type that can be determined at parse time. The type may be a primitive type or a reference type. Reference types include class types and interface types. Often the term type refers to either a class or an interface.

Every object belongs to some particular class: the class that was mentioned in the creation expression that produced the object, the class whose Class object was used to invoke a reflective method to produce the object, or the String class for objects implicitly created by the string concatenation operator (+), or the Document class for objects implicitly created by the document concatenation operator (+) or the time of day, time of week, day of week document operator (||), or the Prompt class for objects implicitly created by the prompt concatenation operator (+) or the prompt escalation, time of day, time of week, day of week or random prompt operator (||) or the prompt substitution operator (|||), or the Grammar class for objects implicitly created by the grammar compound operator (||). This class is called the class of the object. (Arrays also have a class, as described at the end of this section.) An object is said to be an instance of its class and of all super classes of its class.

Sometimes a variable or expression is said to have a "run-time type". This refers to the class of the object referred to by the value of the variable or expression at run time, assuming that the value is not null.

The parse time type of a variable is always declared, and the parse time type of an expression can be deduced at parse time. The parse time type limits the possible values that the variable can hold or the expression can produce at run time. If a run-time value is a reference that is not `null`, it refers to an object or array that has a class, and that class will necessarily be compatible with the parse-time type.

Even though a variable or expression may have a parse-time type that is an interface type, there are no instances of interfaces. A variable or expression whose type is an interface type can reference any object whose class implements that interface.

Here is an example of creating new objects and of the distinction between the type of a variable and the class of an object:

```
{
    java.util.ArrayList alist = new java.util.ArrayList();
    java.util.Vector vector = new java.util.Vector();
    java.util.List list;

    list = alist;
    list = vector;
}
```

In this example:

- The local variable `list` has as its type the interface `java.util.List`, so it can hold a reference to any object whose class implements `java.util.List`; specifically it can hold a reference to either `java.util.ArrayList` or `java.util.Vector`.
- Note that an expression such as `"new java.util.List()"` is not valid because it is not possible to create an instance of an interface, only of a class.

Every array also has a class; the method `getClass`, when invoked for an array object, will return a class object (of class `Class`) that represents the class of the array.

The classes for arrays have strange names that are not valid identifiers; for example, the class for an array of `int` components has the name `"[I"` and so the value of the expression:

```
new int[10].getClass().getName()
```

is the string `"[I"`; see the specification of `Class.getName` for details.

About Conversions in the Expression Language

In addition to the conversions allowed in Java, you should be aware of the following about conversions in the Expression Language:

- [Prompt Conversions, page 1-32](#)
- [Document Conversions, page 1-32](#)
- [String Conversions, page 1-32](#)
- [String Parsing, page 1-33](#)
- [New Objects Resulting from Conversions, page 1-33](#)

Prompt Conversions

Prompt conversion applies only to the operands of the binary + operator when one of the arguments is a Prompt. In this special case only, the other argument to the + is converted to a Prompt as described in [Table 1-7](#), and a new Prompt which is the concatenation of the two prompts is the result of the +.

The prompt concatenation operator +, which, when given a Prompt operand and an integral or floating-point operand, will convert the integral or floating-point operand to a Prompt representing its value in spoken form, and then produce a newly created Prompt that is the concatenation of the two prompts.

There is a prompt conversion to type Prompt from every other type, including the null type as described in [Table 1-7](#). For the null type, the result is the empty prompt.

Document Conversions

There is a document conversion to type Document from the Prompt, String, java.io.InputStream, and java.io.Reader types. The resulting document for the last two types can only be accessed once. There is also a conversion to the type java.io.InputStream, from the Prompt and Document types. And finally there is a conversion to the type java.io.Reader type from the Document type.

Document conversion allows the Prompt type to be converted to type Document, and a conversion from type Prompt to type Document requires run-time processing to collect the content of the specified prompt and return it as a Document object. This conversion might result in an exception being thrown at run-time. It also allows the String type to be converted to type Document.

Example Document Conversion Code

```
// Prompt conversion of i and f
Prompt p = P[ValueOf.wav] + S["i"] + P[Is.wav] + i

// Document conversion of prompt p
Document d = (Document)p;
```

String Conversions

String conversion applies only to the operands of the binary + operator when one of the arguments is a String and the other is not a Prompt or both of them are of type char. In the first special case, the other argument to the + is converted to a String, and a new String which is the concatenation of the two strings is the result of the +. In the last special case, both characters are converted to new Strings and then concatenated together to return a new String as the result of the +. String conversion is specified in detail within the description of the string concatenation + operator.

Any type may be converted to type String by string conversion.

A value x of primitive type T is first converted to a reference value as if by giving it as an argument to an appropriate class instance creation expression:

- If T is Boolean, then use new Boolean(x).
- If T is char, then use new Character(x).
- If T is byte, short, or int, then use new Integer(x).
- If T is long, then use new Long(x).
- If T is float, then use new Float(x).

- If T is double, then use `new Double(x)`.

This reference value is then converted to type `String` by string conversion.

Only reference values need to be considered. If the reference is null, it is converted to the string "null" (four ASCII characters n, u, l, l). If it is a `Document` then the whole document is read and returned as a single string. Otherwise, the conversion is performed as if by an invocation of the `toString` method of the referenced object with no arguments; but if the result of invoking the `toString` method is null, then the string "null" is used instead.

The `toString` method is defined by the primordial class `Object`; many classes override it, notably `Boolean`, `Character`, `Integer`, `Long`, `Float`, `Double`, and `String`.

String conversion applies only to the operands of the binary `+` operator when one of the arguments is a `String` and the other is not a `Prompt` or both of them are of type `char`. In the first special case, the other argument to the `+` is converted to a `String`, and a new `String` which is the concatenation of the two strings is the result of the `+`. In the last special case, both characters are converted to new `Strings` and then concatenated together to return a new `String` as the result of the `+`. String conversion is specified in detail within the description of the string concatenation `+` operator.

The string concatenation operator `+`, when given a `String` operand and a floating-point operand, converts the floating-point operand to a `String` representing its value in decimal form (without information loss), and then produce a newly created `String` by concatenating the two strings.

String Parsing

There is a string parsing from type `String` to every other numeral type. This type of conversion typically parses the string operand by first replacing all occurrences of the `'*'` character to a `'.'`. This allows input of numbers through steps such as the Cisco Unified CCX Editor `GetDigitString` step where the caller might use the `'*'` character to represent a decimal point. Then the resulting string is converted to the specified numerical type. String parsing can always result in exceptions being thrown at run-time.

New Objects Resulting from Conversions

A new class instance is implicitly created when:

- The string, prompt or document concatenation operator `+` is used in an expression, resulting in a new object of type `String`, `Prompt`, or `Document`.
- The prompt escalation, time of day prompt, time of week prompt, day of week prompt or random prompt operator `||` is used in an expression.
- The prompt substitution operator `|||` is used in an expression.
- The time of day document, time of week document or day of week document operator `||` is used in an expression.
- The compound grammar operator `||` is used in an expression.
- The String Concatenation Operator `+`, which, when given a `String` operand and a reference, converts the reference to a `String` by invoking the `toString` method of the referenced object (using "null" if either the reference or the result of `toString` is a null reference) or reads the whole document and converts it as a string if the referenced object is of type `Document`, and then produces a newly created `String` that is the concatenation of the two strings.

- The [Document Concatenation Operator +](#), page 1-10, which when given a Document operand and a reference to another Document type produces a newly created Document that is the concatenation of the two documents.
- The time of day document, time of week document and day of week document operator || ([Prompt Escalation Operator ||](#), page 1-11), which when given a Document operand and a reference to another Document type produces a newly created Document based on the qualifiers having been applied to the two document operands.
- The [Prompt Concatenation Operator +](#), page 1-10, which, when given a Prompt operand and a reference to a char, Currency, Date, Document, java.io.File, java.io.InputStream, Language, Prompt, String, java.net.URL, Time or any numeral types, converts the reference to a Prompt based on [Table 1-7](#), and then produces a newly created Prompt that is the concatenation of the two prompts.



Note The types with the asterisk (*) require that the proper language pack be installed. The asterisk is not part of the type.

Table 1-7 Prompt Concatenation Conversion Result

Type	Prompt Result
char*	Spoken representation of the character
Currency*	Spoken representation of the currency designator
Date*	Spoken representation of the date
Document	Assumes document represents a properly encoded prompt and plays its content
java.io.File	Assumes file represents a properly encoded prompt and plays its content
java.io.InputStream	Assumes stream represents a properly encoded prompt and plays its content
Language*	Spoken representation of the represented language
Prompt	No conversion needed
String*	Spells back the string one character at a time
java.net.URL	Assumes referenced content represents a properly encoded prompt and plays its content
Time*	Spoken representation of the time
Any numeral types*	Spoken representation of the value

- The prompt escalation, time of day prompt, time of week prompt, and day of week prompt operator || ([Prompt Escalation Operator ||](#), page 1-11), which when given a Prompt operand and a reference to another Prompt type produces a newly created Prompt based on the qualifiers having been applied to the two prompt operands.
- [The Prompt Substitution Operator |||](#), page 1-9, which when given a Prompt operand and a reference to another Prompt type produces a newly created Prompt.

- The compound grammar operator `||`, which when given a Grammar operand and a reference to another Grammar type produces a newly created Grammar.
- The Type Comparison Operator `instanceof`
- The Reference Equality Operators `==` and `!=`
- The Conditional Operator `?:`
- [Prompt Qualifier Operator @](#), page 3-105, which accepts either a `DayOfWeekLiteral`, `Language`, `Time`, or numeral type as its right hand-side operand and results in the same prompt/document being qualified in order to be used with the prompt/document container operator `||` to create a day of week prompt/document, a time of day prompt/document or a time of week prompt/document, or simply to override the language of a specific prompt/document.
- [Prompt Weight Qualifier Operator %](#), page 3-105, which accepts a numeral type as its right hand-side operand and results in the same prompt being qualified in order to be used with the prompt container operator `||` to create a random prompt.

There may be many references to the same object. Most objects have state, stored in the fields of objects that are instances of classes or in the variables that are the components of an array object. If two variables contain references to the same object, the state of the object can be modified using one variable's reference to the object, and then the altered state can be observed through the reference in the other variable.

- Array creation expressions, string concatenation expressions, document concatenation expressions, or prompt concatenation expressions throw an `OutOfMemoryError` if there is insufficient memory available.



CHAPTER 2


Using Expressions and the Expression Editor

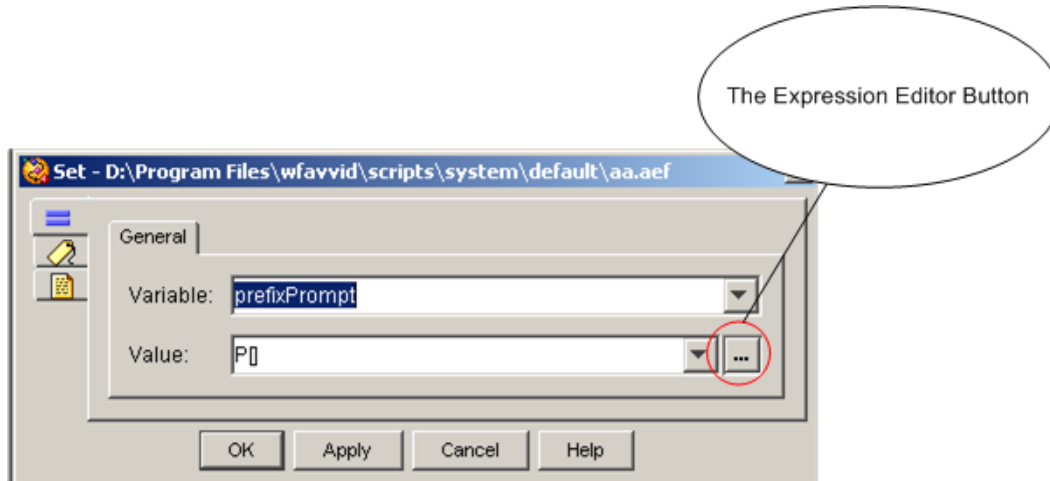
This chapter covers the following topics:

- [How to Access the Cisco Unified CCX Expression Editor, page 2-1](#)
- [How to Use the Expression Editor, page 2-2](#)
- [About the Expression Editor Toolbar, page 2-4](#)
- [About the Expression Editor Syntax Buttons, page 2-9](#)
- [About Expression and Java Licensing, page 2-9](#)

For an explanation of each toolbar on each Expression Editor tab, see [Using Expressions and the Expression Editor, page 2-1](#).

How to Access the Cisco Unified CCX Expression Editor

Whenever you see this 3-dot button  in a Cisco Unified CCX Editor step properties window, you can click on it to open the Expression Editor to edit the value of the field to the left of the button. The following figure shows the Expression Editor button in the Set step properties window.



How to Use the Expression Editor

Use the Expression Editor to enter or modify expressions in a Cisco Unified CCX script.

This section includes the following topics:

- [How To Enter Expressions in the Expression Editor, page 2-2](#)
- [About the Expression Editor Toolbar, page 2-4](#)
- [About the Expression Editor Syntax Buttons, page 2-9](#)
- [About Expression and Java Licensing, page 2-9](#)

How To Enter Expressions in the Expression Editor

Expressions are useful if you do not know an exact value at design time and instead need to enter a formula that can be evaluated at run time.

**Note**

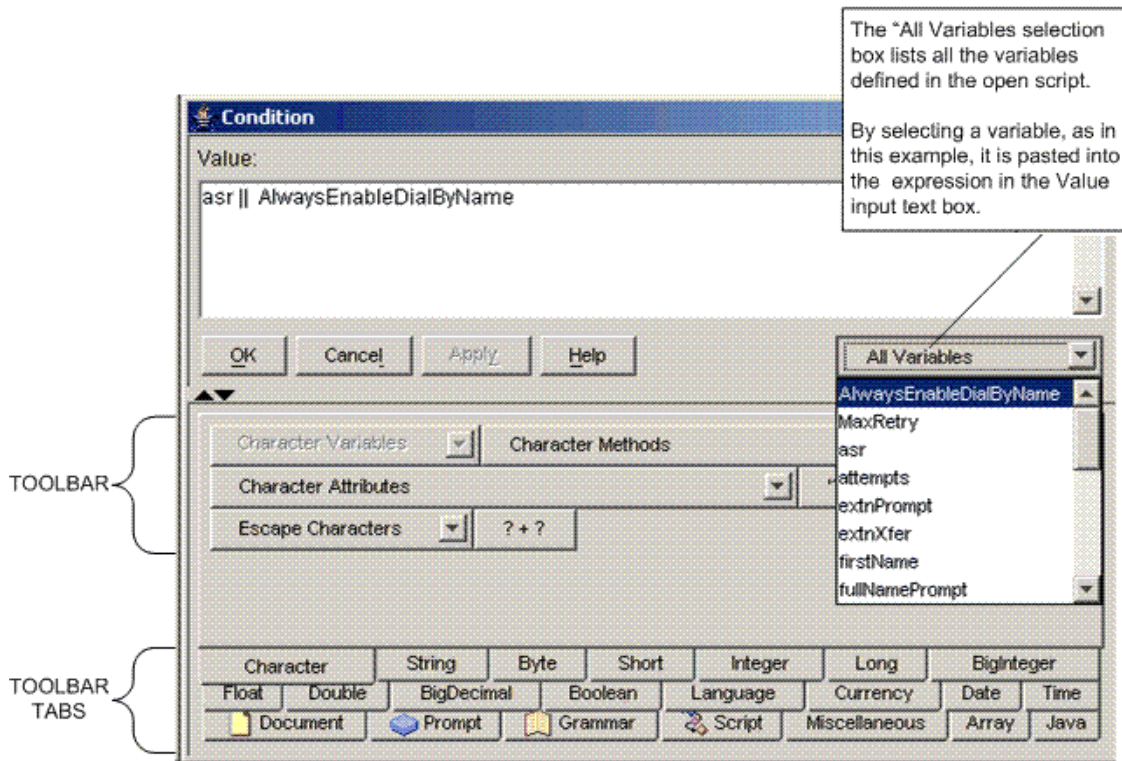
The resulting type of expression must match the expected input type or types (which you check at design time).

In the Expression Editor window, you can enter or edit an expression in the **Value** input text box and you can use the **All Variables** selection box to get quick access to a variable you have previously defined in the script to paste it into the expression.

When you choose a variable from the All Variables selection box, the variable name appears in the Value input text box.

After you enter the expression, click **OK** and the Expression Editor closes.

Figure 2-1 Example Expression Editor Window with the “All Variables” Selection box Open



About the Expression Editor Toolbar

Below the Expression Editor Value input text box and buttons is a versatile toolbar.



Note

The toolbar changes to suit the type of data or feature you select in the toolbar tabs at the bottom of the Expression Editor window.

This section includes the following topics:

- [Toolbar Tabs, page 2-5](#)
- [A Pop-Up Menu, page 2-7](#)
- [Showing or Hiding the Expression Editor Toolbar, page 2-8](#)

Toolbar Tabs

By clicking on the appropriate tab below the toolbar, the toolbar changes to include the tools useful for editing the selected type of data indicated by the selected tab. For example, in [Figure 2-2](#), the Character toolbar is selected and so tools appropriate for editing or entering character data are displayed.

The toolbar scripting tools (or aids) include:

- **Variables:** A selection box listing all the variables of the toolbar type selected (for example, character) currently contained in the open script.
- **Constructors.** A selection list of the public Java constructors available for creating and initializing new objects of the selected data type.
- **Methods.** A selection list of public Java methods for all the operations you can perform on the selected data type. A method has four basic parts:
 - The method name
 - The type of object the method returns
 - A list of parameters
 - The body of the method
- **Attributes.** A selection list of all the public Java attributes available for the selected data type. These are the things that differentiate one object from another in the selected data type. For example, color or size.
- **Constants and Keywords.** In some cases, constants and keywords for the selected data type or object are included.
- **Syntax** button. Buttons for quickly entering data of the selected type with the correct syntax. The question marks on the buttons indicate command parameters which you need to supply.
- Easy access to **Prompts, grammars, documents, and scripts** stored inside the Cisco Unified CCX repository.

When you click a button or select an item from a list, the Cisco Unified CCX Editor inserts the selected expression text at the cursor position in the text input field.

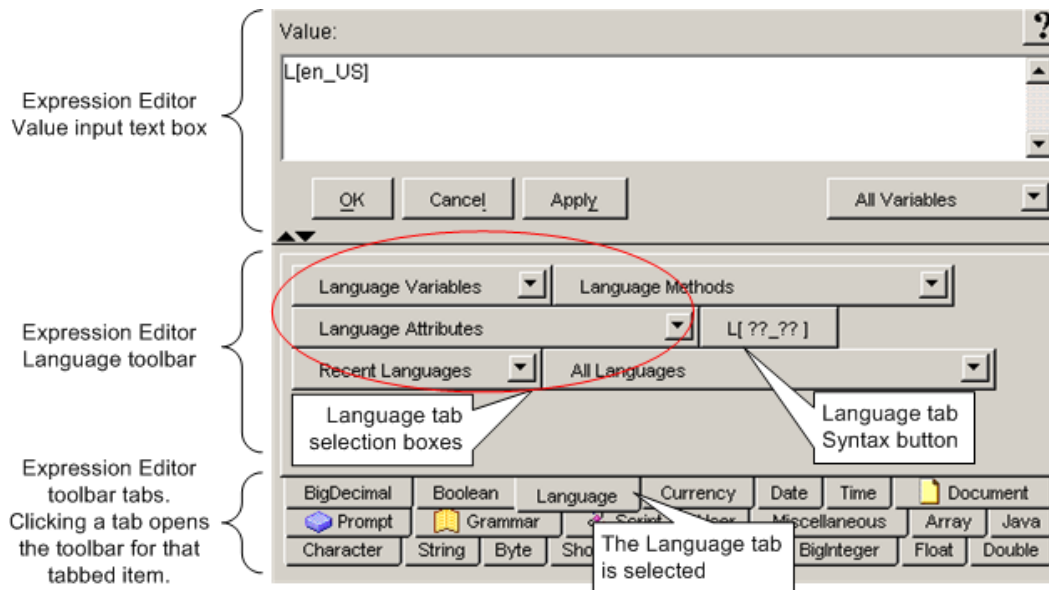
For example, if you are creating an expression that accesses the current time, on the Time tab, click the **now** button, and the Cisco Unified CCX Editor will insert the Java code that retrieves the current time when the script runs.

**Note**

The Java tab contains a selection list of the constructors, methods, attributes, and syntax buttons of the selected Java object within the open script. Therefore, the contents of this tab will vary.

The Java tab allows you to enter a class name of your own in order to have its set of constructors, methods or attributes listed in the selection boxes. This enables an easy lookup of what is available so you can paste it into the expression directly. The Java toolbar is populated with the constructors, methods or attributes of the class you enter. A selection box drop-down arrow is disabled if the class entered is invalid or does not have any constructors, methods or attributes.

Figure 2-2 Example Expression Editor Window with the Language Toolbar Selected



A Pop-Up Menu

Right click in the Expression Editor window to access the pop-up menu. This enables you to access editing functions such as Undo, Cut, and Paste. See [Figure 2-3](#).

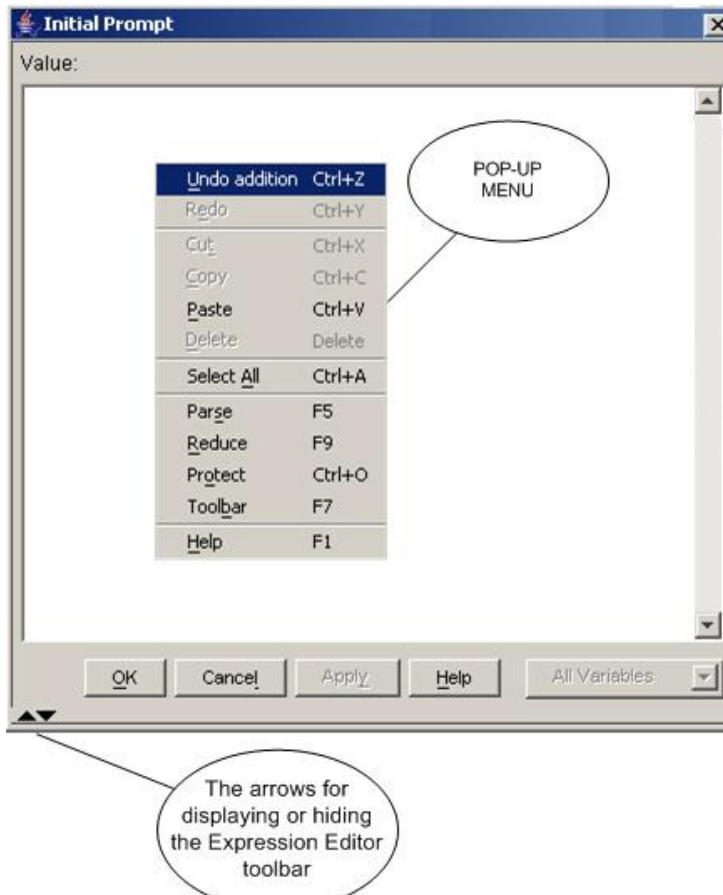
The popup menu also provides two special functions:

- One allows you to parse an expression immediately in order to pinpoint errors
- The other allows you to automatically reduce the expression to a smaller and yet equivalent expression (for example $3 + 2$ would be reduced to 5).

Showing or Hiding the Expression Editor Toolbar

To show or hide the Expression Toolbar, click on the arrow buttons on the bottom left of the Expression Editor text window. This alternately removes or displays the tabbed toolbar.

Figure 2-3 Expression Editor Window without the Toolbar but with the Pop-up Menu



About the Expression Editor Syntax Buttons

The toolbar syntax buttons indicate the different ways you can operate on a data type. This syntax is the same as the Java language syntax plus additional syntax aids for handling prompts and documents.

About Expression and Java Licensing

Beginning with Cisco CRS 4.x, expressions are validated against installed licenses to make sure that they do not violate license agreements. This validation is performed by the Cisco CRS Engine whenever a script is loaded or whenever a prompt template or grammar template is accessed and evaluated.

For script expressions containing TTS or Java features to work during runtime, you must have either a Cisco Unified IP IVR, a Cisco Unified CCX Enhanced, or a Cisco Unified CCX Premium license.

**Note**

In Cisco Unified CCX Standard, you can enter only simple expressions unless you also have a Java license. You automatically have a Java license with the other four Cisco Unified CCX products.

An example of a TTS feature is a TTS prompt complex literal. A Java feature is a complex expression block, a Java-like statement, method, constructor invocation expression, or a field access expression.

Any license violation will be recorded in the logs and prevent the scripts from being loaded in memory.



CHAPTER 3

Expression Editor Tool Reference Descriptions

Each Expression Editor tool tab helps you enter or modify script data of a specific type.

This chapter:

- Lists
 - All the [friendly data types](#) with their Java equivalents that you can use in the Expression Editor.
 - [Tips common](#) to all the Expression Editor tools.
- Describes the features, data types, and functions of the Expression Editor tool tabs:
 - [Array](#), page 3-8
 - [BigDecimal](#), page 3-13
 - [BigInteger](#), page 3-18
 - [Boolean](#), page 3-23
 - [Byte](#), page 3-27
 - [Character](#), page 3-32
 - [Currency](#), page 3-37
 - [Date](#), page 3-39
 - [Document](#), page 3-44
 - [Double](#), page 3-54
 - [Float](#), page 3-57
 - [Grammar](#), page 3-62
 - [Integer](#), page 3-69
 - [Java](#), page 3-77
 - [Language](#), page 3-84
 - [Long](#), page 3-87
 - [Miscellaneous](#), page 3-91
 - [Prompt](#), page 3-94
 - [Script](#), page 3-110
 - [Short](#), page 3-113
 - [String](#), page 3-117
 - [Time](#), page 3-122

– User, page 3-125

Friendly Data Types

All data types can be used in scripts as parameters in an expression.

The Expression Language supports any Java data types entered as a fully qualified Java class name. As with the Java language, all classes defined as part of the `java.lang` package can be named directly without having to include the package name. For example, the `java.lang.String` class can also be referred to as `String`. That is because the `String` class is both a friendly data type and also defined in the default `java.lang` package for which all classes do not need to be entered using their fully qualified class name. However, the `java.util.ArrayList` class cannot be referred to as simply `ArrayList`.

In addition to the fully qualified Java class names, the Cisco CRS expression language defines friendly data types that are equivalent to these Java class names. Entering these data types in a script is the same as entering their equivalent fully qualified Java class names.

Primitive Java data types such as `int`, `boolean`, `long`, `float`, `byte`, `char`, `short`, `double` are automatically converted into their corresponding Java object representation. You can enter into a Cisco Unified CCX script either the Java class name or the Cisco Unified CCX friendly data type name.

Table 3-3 lists the friendly data types that the Expression Language uses and their equivalents in Java. The Java equivalent class names are the Java fully qualified class name, that is the data name and the package in which it is included.

Table 3-1 Cisco Unified CCX Friendly Data Types with their Java Equivalent Class Names

Friendly Data Type	Java Class Name	Default Value
<code>int</code>	<code>java.lang.Integer</code>	<code>0</code>
<code>String</code>	<code>java.lang.String</code>	<code>""</code>
<code>char</code>	<code>java.lang.Character</code>	<code>'\u0000'</code>
<code>float</code>	<code>java.lang.Float</code>	<code>0.0f</code>
<code>long</code>	<code>java.lang.Long</code>	<code>0L</code>
<code>double</code>	<code>java.lang.Double</code>	<code>0.0d</code>
<code>byte</code>	<code>java.lang.Byte</code>	<code>(byte)0</code>
<code>short</code>	<code>java.lang.Short</code>	<code>(short)0</code>
<code>boolean</code>	<code>java.lang.Boolean</code>	<code>false</code>
<code>User</code>	<code>com.cisco.user.User</code>	<code>null</code>
<code>Contact</code>	<code>com.cisco.contact.Contact</code>	<code>null</code>
<code>Session</code>	<code>com.cisco.session.Session</code>	<code>null</code>
<code>Script</code>	<code>com.cisco.script.Script</code>	<code>null</code>
<code>Prompt</code>	<code>com.cisco.prompt.Playable</code>	<code>P[]</code>
<code>Grammar</code>	<code>com.cisco.grammar.Recognizable</code>	<code>G[]</code>
<code>Document</code>	<code>com.cisco.doc.Document</code>	<code>DOC[]</code>
<code>Language</code>	<code>java.util.Locale</code>	The system default language
<code>Currency</code>	<code>com.cisco.util.Currency</code>	The system default currency

Table 3-1 Cisco Unified CCX Friendly Data Types with their Java Equivalent Class Names (continued)

Friendly Data Type	Java Class Name	Default Value
Date	java.util.Date	The current date at the time of interpretation
Time	java.sql.Time	The current time at the time of interpretation
BigInteger	java.math.BigInteger	0IB
BigDecimal	java.math.BigDecimal	0.0fb
Iterator	java.util.Iterator	null
Customer	com.cisco.uccx.contextservice.model.InternalCustomer	null
POD	com.cisco.uccx.contextservice.model.InternalPod	null

Following table gives you the detail about the additional methods without parameters that are available in the Expression Editor of the scripts for the **Customer** and **POD** objects:

Class	Return type	Name of the Method	Description
com.cisco.uccx.contextservice.model.InternalCustomer	java.util.UUID	getCustomerId	This returns the UUID of the Customer object.
com.cisco.uccx.contextservice.model.InternalCustomer	java.util.Date	getCreateDate	This returns the creation date of the Customer object.
com.cisco.uccx.contextservice.model.InternalCustomer	java.util.Date	getLastModifiedDate	This returns the last modified date of the Customer object.
com.cisco.uccx.contextservice.model.InternalPod	java.util.UUID	getPodId	This returns the UUID of the Pod object.
com.cisco.uccx.contextservice.model.InternalPod	java.util.UUID	getCustomerId	This returns the UUID of the Customer corresponding to the POD object.

Class	Return type	Name of the Method	Description
com.cisco.uccx.contextservice.model.InternalPod	java.util.Date	getCreateDate	This returns the creation date of the Pod object. Note: Create POD step will not show the Creation date of the POD due to performance reasons. Do a Retrieve PODs step to obtain the creation date of the POD object.
com.cisco.uccx.contextservice.model.InternalPod	java.util.Date	getLastModifiedDate	This returns the last modified date of the Pod object.
com.cisco.uccx.contextservice.model.InternalPod	String	getMediaType	This returns the media type associated with the pod (For example: VOICE, CHAT, EMAIL).
com.cisco.uccx.contextservice.model.InternalPod	String[]	getContributors	This returns the contributors associated with the Pod. Format will be Contributor-ID(Contributor-Type).

Tool Tips

This section describes:

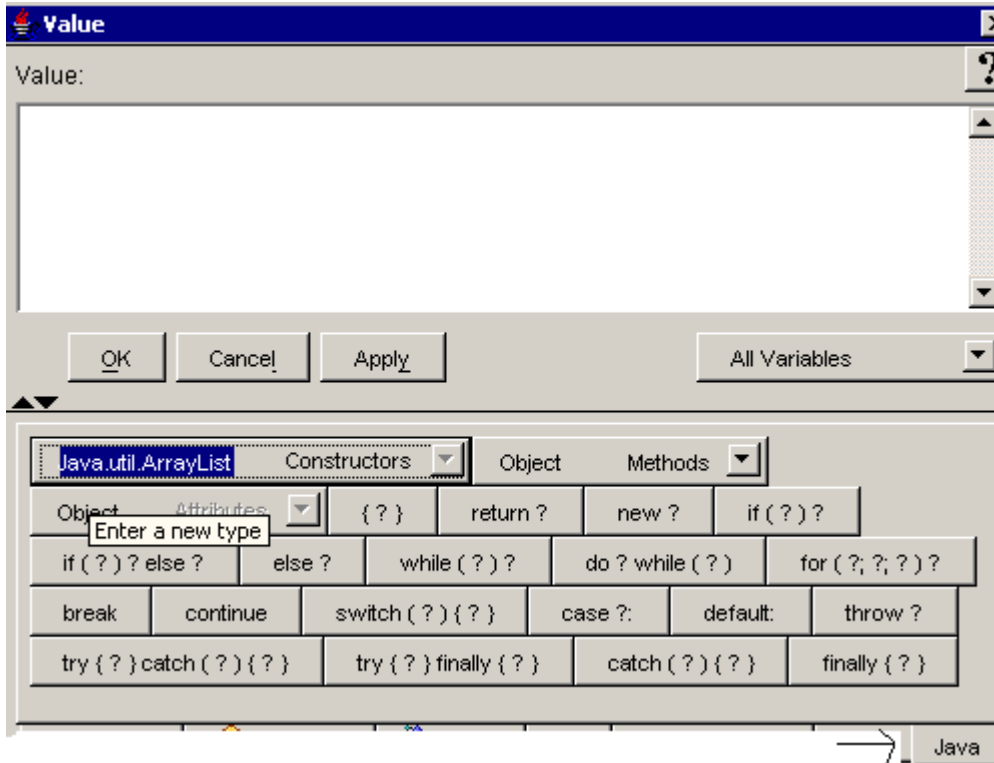
- [Tool Tips For the Java and Miscellaneous Tool Tabs, page 3-4](#)
- [Tool Tips For All the Expression Editor Tools, page 3-6](#)

Tool Tips For the Java and Miscellaneous Tool Tabs

- The Java tool tab:
 - Contains a selection list of the constructors, methods, attributes, and syntax buttons of the selected Java object within the open script. Therefore, the contents of this tab will vary.

- Allows you to enter any fully qualified Java class name of your choosing in order to have its set of constructors, methods or attributes listed in the selection boxes. Included in this list of class names are every class from the Sun JDK, all the Cisco classes, and any custom classes you might have uploaded through the Cisco Unified CCX Application Administration web pages. See [Figure 3-1](#).

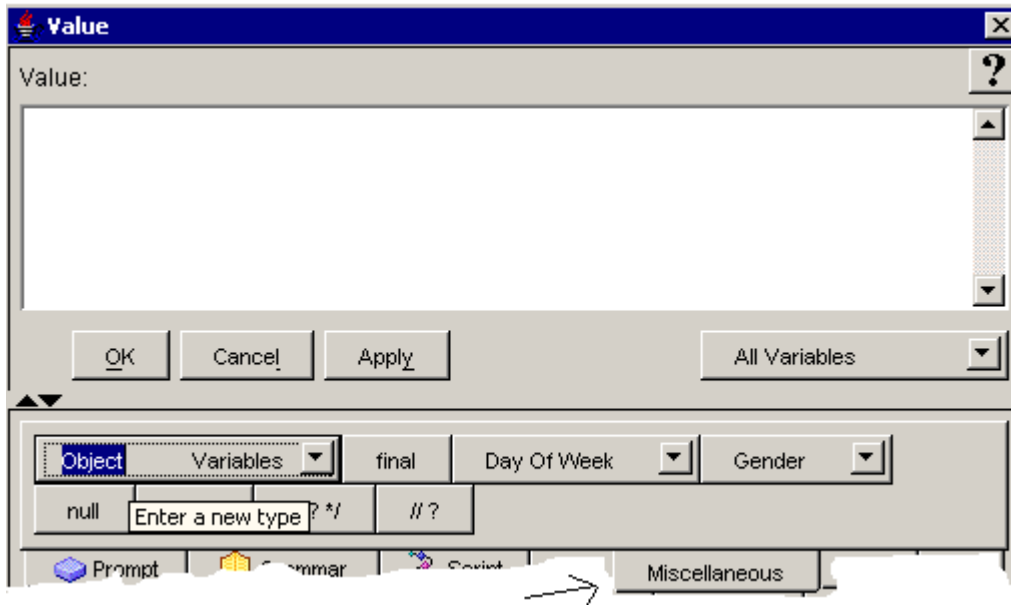
Figure 3-1 Java Tool Tab with “Java.util.ArrayList” Entered By User For its Selection List



This enables an easy lookup of what is available so you can paste it into an expression directly. The selection box drop-down arrow is disabled if the class entered is invalid or does not have any constructors, methods or attributes. For how to do this, see [How to Access a Java Constructor, Method, or Attribute for Any Class](#), page 3-79

- The Miscellaneous tool tab:
 - Provides a way to easily enter types of data into an expression that are not included in the other tabs.
 - Allows you to filter the variables selection list of all the variables in the opened script so that only those variables of the type you select are displayed in the selection list. See [Figure 3-2](#). For how to do this, see [Object Variables](#), page 3-92.

Figure 3-2 Miscellaneous Tool Tab



Tool Tips For All the Expression Editor Tools

- The example expressions in this guide include both simple and complex ones and list the script variables used in the expressions.
- In Unified IP IVR, Unified CCX Enhanced, and Unified Contact CCX Premium, you can enter both simple and complex expressions.
However, in Unified CCX Standard, you can enter only simple expressions unless you also have a Java license. You automatically have a Java license with the other four Cisco Unified CCX products.
- A complex expression is one surrounded by braces and having more than one statement, is specific to the Java language, and requires that you have a Java license. If your script contains a complex expression and you do not have a Java license, then when you load that script in the Cisco Unified CCX engine, the script is declared invalid and you will not be able to run it.
- You can paste into an expression a variable, constructor, method, and attribute that you select from the tool selection boxes.
- The constructor, method, and attribute selection boxes for each tool display the public Java constructors, methods, and attributes available for that tool.
- The variables selection box filters the variables in the opened script so that only those of the selected tool type are displayed.
- Both static and nonstatic available public Java methods and attributes are displayed:

- A static method or attribute is an operation attached to a data type rather than attached to an object. It is similar to a global function and does not require an instance Object of the type.



Note As opposed to previous releases, invoking static methods no longer requires having a dummy variable created of the proper type. Instead one can simply prefix the method name with the class name followed by a period. For example: `String.valueOf()`.

- A non static method or attribute requires an instance Object of the type.

See [Using Expressions and the Expression Editor, page 2-1](#) for further generic “how to” information.

Array

Use the Array tab to enter or modify array data in an expression.

This topic includes the following:

- [About Arrays, page 3-8](#)
- [Array Java Specification on the Web, page 3-8](#)
- [Example Array Code, page 3-9](#)
- [Array Variables, page 3-10](#)
- [Index Variables, page 3-10](#)
- [Array Methods, page 3-11](#)
- [Array tab Syntax Buttons, page 3-11](#)

About Arrays

As in the Java programming language, arrays in the Expression language are objects that are dynamically created. All methods of class `java.lang.Object` may be invoked on an array.

An array is different from the other data types listed in the Expression Editor tool tabs. For example, the other data type variables represent a single value, like a string, an integer, or a boolean. But an array variable represents a collection (array) of values of one of the other data types; for example: a collection of integers, strings, dates, or whatever object.

A component of an array is an integer, or string, or any other Java type or even another array. You can act on an array component in the same way as you can act on any other object of that data type.

Array Java Specification on the Web

For the Sun Java specification on arrays, see

http://java.sun.com/docs/books/jls/second_edition/html/arrays.doc.html#27805.

The following two sections describe the differences between arrays in the second edition of the Sun Java specification and arrays in the Cisco Unified CCX Expression Language:

- [Array Enhancements, page 3-8](#)
- [Array Exceptions, page 3-9](#)

Array Enhancements

An array iterator attribute is included in the Cisco Unified CCX Expression Language but is not in the Sun Java specification on arrays:

- The public final field `iterator`, containing a list of all components of the array (the iterator may not be null) has been added to the members of an array type.
- The iterator makes an array's components available as a final instance variable.
- All array components may be accessed through the iterator as defined by the reference type `java.util.Iterator`. This is done by following the array reference with the `iterator` field.

Example Expression: `ArrayVariableName.iterator`

This example expression returns an iterator object on which the method `next()` can be called to retrieve the next element of the array starting with the first one. The method `hasNext()` can be also called to check if there is another element to extract from the array. Once all elements of the array have been iterated, the iterator can no longer be used and throws a `java.util.NoSuchElementException` exception.

Array Exceptions

The following are array features in Java that are not in the Cisco Unified CCX Expression Language:

- As opposed to the Java language, a trailing comma may not appear after the last expression.
- As opposed to the Java programming language, the `[]` may not appear as part of the declarator for a particular variable as in this example:

```
byte matrix[];
```

This declaration would be valid in the Java programming language but in the Expression Language, it must be written as follows:

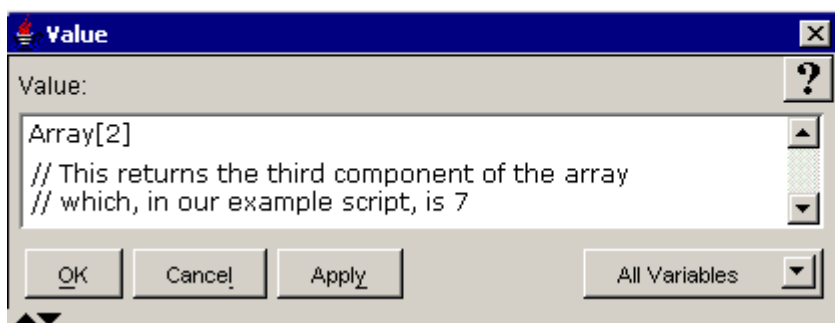
```
byte[] matrix;
```

Example Array Code

In the following two examples, the script variables used in the expressions are listed in the top right of each example.

Figure 3-3 A Simple Expression Using an Array and an Array Script Variable

Name	Type	Value
array	int[]	new int[] { 23, 4, 7 }



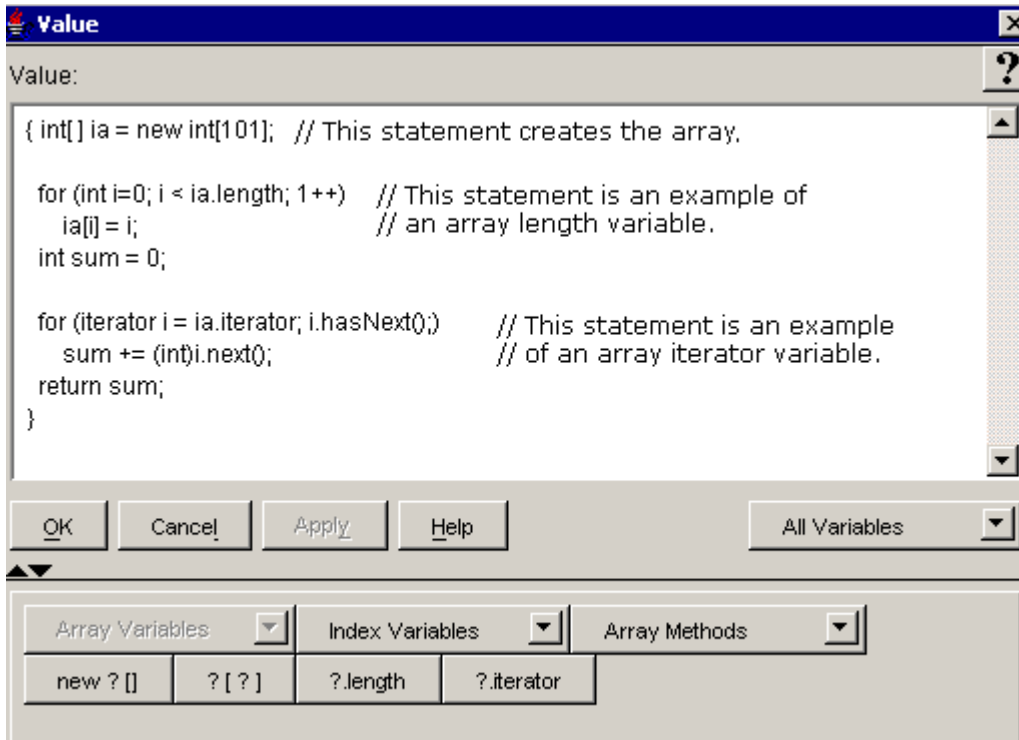
The expression

```
new int[] { 23, 4, 7, }
```

is valid in Java but not in the expression language. Instead write:

```
new int[] { 23, 4, 7 }
```

Figure 3-4 A Complex Expression Using an Array and an Array Script Variable



The following sections describe the options on the Array tool tab:

- [Array Variables, page 3-10](#)
- [Index Variables, page 3-10](#)
- [Array Methods, page 3-11](#)
- [Array tab Syntax Buttons, page 3-11](#)

Array Variables

An array variable names the object that is the array. In addition, an array variable contains other, multiple variables. These variables are called the array components and each are of the same type as the type held by the array. For example, if a component type of an array is T, then the type of the array itself is T[].

There is no maximum limit to the size of an array, nor any requirement that member variables be indexed or assigned contiguously. Only memory limits the size of an array.

The Array Variables selection box lists all the array variables contained in the currently opened script. Use this selection box to paste an already defined array variable into an expression.

Index Variables

Arrays are zero-based, that is, the first element is indexed with the number 0. If an array has n components, then n is the *length* of the array and its components are referenced using integer indices from 0 to n - 1, inclusive.

An array index variable holds an index entry for the array and is used to retrieve a specific array component. For example: `ArrayVariable[IndexVariable]=value`

The Index Variables selection box lists all the variables contained in the opened script that are of the type that can be used as an index variable. Use this selection box to paste one of these variables into an expression as an index variable if it is appropriate for such in your script.

Array Methods

For descriptions of the public Java array methods available in the selection box, see http://java.sun.com/docs/books/jls/second_edition/html/arrays.doc.html#27805.

Array tab Syntax Buttons

An array must be declared. When you declare an array variable, you suffix the **type** with `[]` to indicate that this variable is an array. This states the type of value the array holds. Each `[]` represents one dimension of the array. So the array

```
int[][]
```

represents a 2 dimensional array of integers where the components of the first dimension are of type

```
int[]
```

and the components of these components are of type

```
int
```

The variable name of the array appears in an array declaration followed by a semicolon. Here are some examples:

```
int[ ] x;  
float[ ] nt;  
String[ ] names;
```

Use the `new` keyword to create an array. For example:

```
c = new int[3];
```

In the preceding example, `c` is the array variable. The number in the brackets specifies the number of components in the array, which is called the length of the array. This allocates memory for the array.

Table 3-2 Array Syntax Button Descriptions

Button	Description
new ?[]	<p>Creates a new array. For example: <code>ArrayVariableName = new string[]</code></p> <ul style="list-style-type: none"> To create an array and at the same time assign values to the array components, use this format: <code>new type[] { value, value, value, ... }</code> For example: <code>new int[] { 3, 4, 5 }</code> creates an array of size 3 where the first component is 3, the second is 4 and the last is 5. To create an array and only specify its size, use this format: <code>new type[size]</code> For example: <code>new int[3]</code> creates an array of size 3 but initializes all components to their default value. For a list of variable default values based on type, see Initial Values of Variables, page 1-29.
? [?]	<p>Enters an array component by array variable and index. For example: <code>ArrayVariableName[array_index]</code></p>
? .length ¹	<p>Enters an array length variable. For example: <code>ArrayVariableName.length</code> The length is the number of components in the array.</p>
? .iterator ¹	<p>Enters an array iterator variable. For example: <code>ArrayVariableName.iterator</code> The Iterator is a class based on the Array. It provides methods to go through all the Array components, one at a time.</p>

1. The .length and the .iterator variables do not require the Java license.

BigDecimal

Use the BigDecimal tab to enter or modify BigDecimal data in an expression. BigDecimal is a friendly data type corresponding to the fully qualified java.math.BigDecimal class.

This topic includes the following:

- [About BigDecimals, page 3-13](#)
- [BigDecimal Java Specification on the Web, page 3-13](#)
- [Example BigDecimal Code, page 3-14](#)
- [BigDecimal Variables, page 3-15](#)
- [BigDecimal Constructors, Methods, and Attributes, page 3-15](#)
- [BigDecimal tab Syntax Buttons, page 3-16](#)
- [Floating-Point Literals, page 3-61](#)

About BigDecimals

The BigDecimal class provides a decimal, floating-point arithmetic which produces arbitrary-precision signed decimal numbers. Use BigDecimals when you do not want to be limited and you need more precision than floats allow.

For a description and comparison of the different types of floating-point numbers (floats, doubles, and BigDecimals), see [Floating-Point Literals, page 3-61](#).

The BigDecimal class does normal rounding and gives you complete control over rounding behavior, allowing you to explicitly specify a rounding behavior (scale) for operations capable of discarding precision by using:

- Java constructors to specify a scale.
- Java methods [divide(BigDecimal, int), divide(BigDecimal, int, int), and setScale(int, int)].
- Cisco Unified CCX math operators (see [BigDecimal Enhancements, page 3-14](#) and [BigDecimal tab Syntax Buttons, page 3-16](#)).

Because the BigDecimal class gives you control over rounding and the number of decimal places you are interested in, it can be useful when dealing with money or in any circumstance where the tolerance for rounding errors is low.

As specified in the Sun Java specification on such, a BigDecimal consists of an arbitrary precision integer unscaled value and a non-negative 32-bit integer scale, which represents the number of digits to the right of the decimal point. The number represented by the BigDecimal is (unscaledValue/10scale). BigDecimal provides operations for basic arithmetic, scale manipulation, comparison, hashing, and format conversion.

For examples of how you can use BigDecimals, see [Example BigDecimal Code, page 3-14](#) and [BigDecimal tab Syntax Buttons, page 3-16](#).

BigDecimal Java Specification on the Web

For the Sun Java specification on BigDecimals, see <http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigDecimal.html>.

BigDecimal Enhancements

The following Cisco Unified CCX Expression Language enhancements are not in the Sun Java specification on BigDecimals. In the Cisco Unified CCX Expression Language:

- You can specify a BigDecimal literal with the characters DB as in 26.12DB.
- You can do math operations on BigDecimals, using the standard math operators. See [BigDecimal tab Syntax Buttons](#), page 3-16.
- When entering a BigDecimal as a literal number, the scale of that BigDecimal is set as the number of digits in the fractional part of the number. For example: in the number in 1.53DB, the scale is 2. The scale is zero if there is no decimal point.
- When dividing BigDecimals, the scale of both numbers that are divided is added with a minimum of 25 to set the scale of the result value. If the scale is less than 25, then the scale is fixed at 25. If the sum of the two scales is larger than 25, then the scale is fixed at that larger number.

Example BigDecimal Code

In the following two examples, the script variables used in the expressions are listed in the top right of each example.

Figure 3-5 A Simple Expression Using a BigDecimal and Two Script Variables

Name	Type	Value
rate	BigDecimal	7.25DB
amount	String	"243697"

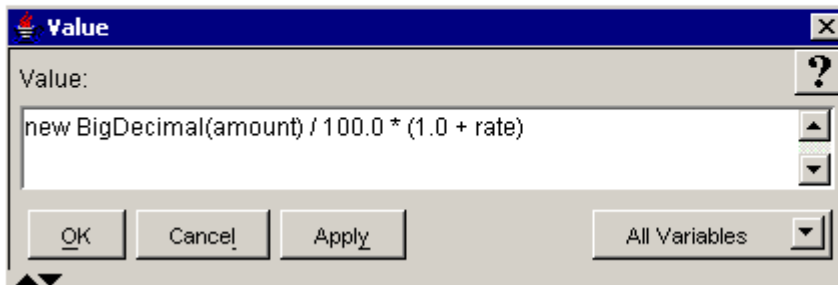


Figure 3-6 A Complex Expression Using a BigDecimal and Two Script Variables

Name	Type	Value
rate	BigDecimal	7.25DB
amount	String	"243697"


```

{
  int v = new Integer(amount); // convert amount variable into an integer value
                               // (assumes no decimal point)
  BigDecimal bd = v / 100.0DB; // convert value into a floating point value where
                               // the first 2 digits represented cents
  return bd * (1.0 + rate); // calculate total price
}

```

**Note**

The preceding example is of a complex expression. This type of expression, one surrounded by braces and having more than one statement, is specific to the Java language and requires that you have a Java license. If your script contains a complex expression and you do not have a Java license, then when you load that script in the Cisco Unified CCX Engine, the script is declared invalid and you will not be able to run it.

BigDecimal Variables

The BigDecimal Variables selection box lists all the BigDecimal variables contained in the currently opened script. Use this selection box to paste an already defined BigDecimal variable into an expression.

A BigDecimal variable consists of an arbitrary-precision integer along with a scale, where the scale is the number of digits to the right of the decimal point.

The default value of a BigDecimal variable is positive zero, that is, 0.0fb.

BigDecimal Constructors, Methods, and Attributes

Use the appropriate selection box to add BigDecimal code to your expression.

The available public methods and attributes include both static and non static ones.

For descriptions of all the public BigDecimal constructors, methods, and attributes available in the selection boxes, see the Java specification at <http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigDecimal.html>.

BigDecimal tab Syntax Buttons

The BigDecimal tab syntax buttons indicate all the ways you can add a BigDecimal to an expression.

Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate BigDecimal operand or literal.



Note All of the BigDecimal syntax listed in [Table 3-3](#) is specific to the Cisco Unified CCX Expression Language.

The use of DB to specify a BigDecimal and the use of math operators on BigDecimals is specific to the Expression Language and is not a part of the Java language syntax.

Table 3-3 *BigDecimal Syntax Button Descriptions*

Syntax Button	Name	Type	Description
?DB	literal	BigDecimal	Enters a BigDecimal literal. See Floating-Point Literals, page 3-61 . For example: 3.14159DB 2E-12DB -100DB
? + ?	addition	arithmetic	Adds two operands.
? - ?	subtraction		Subtracts the second operand from the first.
? * ?	multiplication		Multiplies two operands.
? / ?	division		Divides the first operand by the second.
? % ?	remainder		Returns the remainder of the first operand divided by the second.

Table 3-3 *BigDecimal Syntax Button Descriptions (continued)*

Syntax Button	Name	Type	Description
? *= ?	multiply and assign	assignment The operand on the left of the assignment statement (the first operand) can be any type of variable, including an array component or a public class attribute.	Multiplies the first operand by the second and assigns the result to the first operand.
? /= ?	divide and assign		Divides the first operand by the second and assigns the result to the first operand.
? %= ?	remainder and assign		Divides the first operand by the second operand and assigns the remainder to the first operand.
? += ?	add and assign		Adds the first operand to the second operand and assigns the result to the first operand.
? -= ?	subtract and assign		Subtracts the second operand from the first operand and assigns the result to the first operand.
+?	unary plus	unary	The positive value of the operand.
-?	unary minus		The negative value of the operand.

BigInteger

Use the BigInteger tab to enter or modify BigInteger data in an expression. BigInteger is a friendly data type corresponding to the fully qualified java.math.BigInteger class.

This topic includes the following:

- [About BigIntegers, page 3-18](#)
- [BigInteger Specification on the Web, page 3-18](#)
- [Example BigInteger Code, page 3-18](#)
- [BigInteger Variables, page 3-20](#)
- [BigInteger Constructors, Methods, and Attributes, page 3-20](#)
- [BigInteger tab Syntax Buttons, page 3-20](#)
- [Integer Literals, page 3-75](#)

About BigIntegers

The BigInteger class represents integers that can be arbitrarily large; that is, BigIntegers are not limited to the 64 bits available in the long data type.

Literals of type BigInteger have no maximum and minimum. Any value can be represented using the BigInteger type. For examples of how you can use BigIntegers, see [Example BigInteger Code, page 3-18](#) and [BigInteger tab Syntax Buttons, page 3-20](#).

BigInteger Specification on the Web

For the Sun Java specification on BigIntegers on the web, see <http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigInteger.html>

BigInteger Enhancement

In the Cisco Unified CCX Expression Language, you can specify a BigInteger with the characters IB as in the 234556789IB. This method of specifying a BigInteger is not in the Sun Java specification on BigInteger.

Example BigInteger Code

In the following two examples, the script variables used in the expressions are listed in the top right of each example.

Figure 3-7 Example Simple Expression Using a BigInteger and a Script Variable

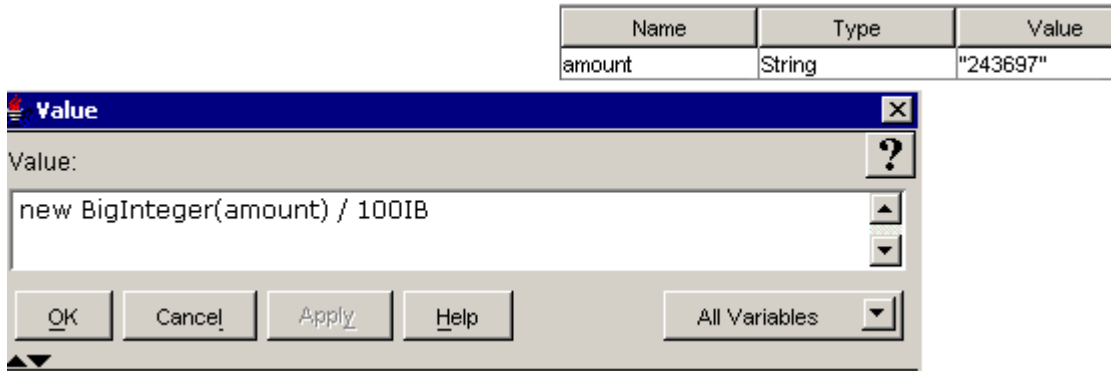
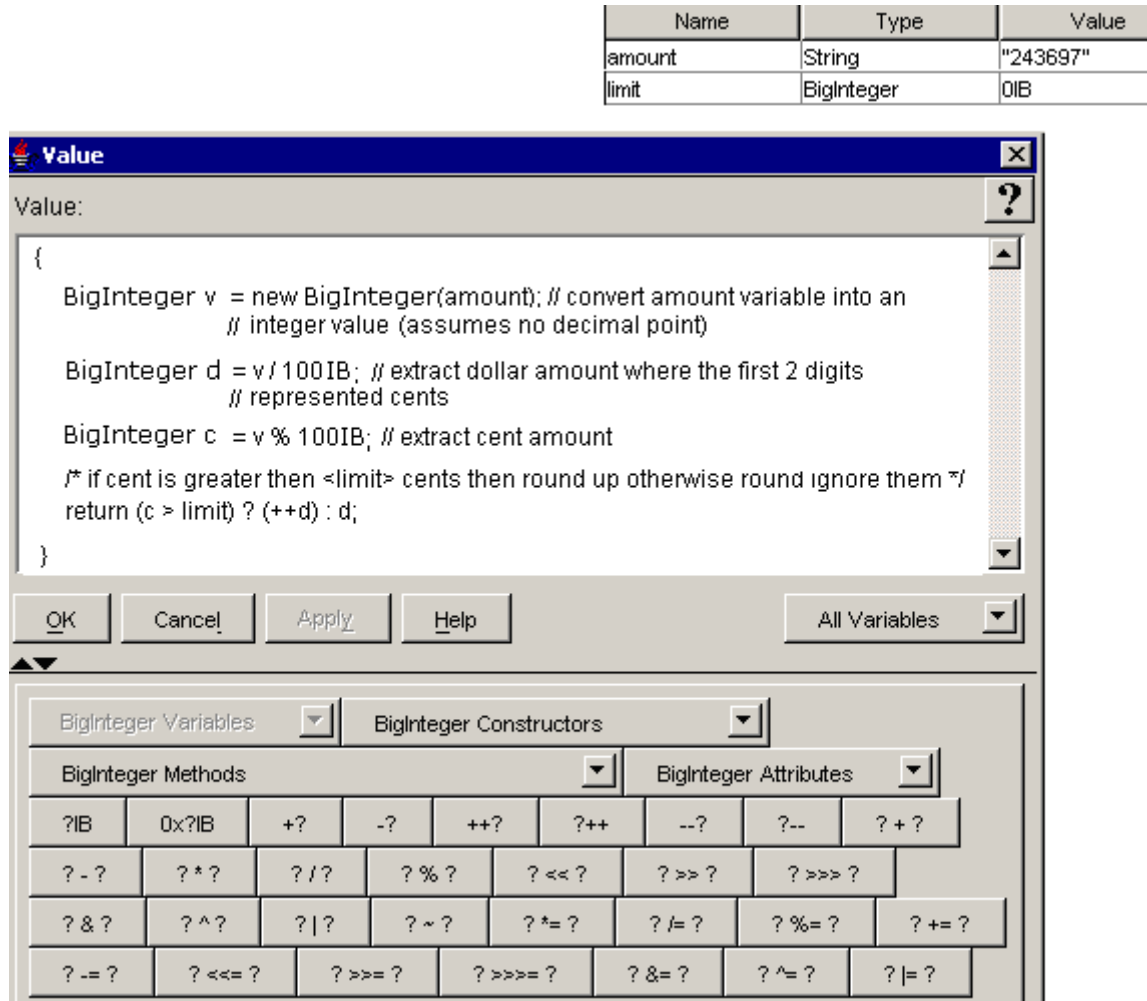


Figure 3-8 Example Complex Expression Using a BigInteger and Two Script Variables



The following sections describe how you can use the BigInteger tab:

- [BigInteger Variables](#), page 3-20
- [BigInteger Constructors, Methods, and Attributes](#), page 3-20

- [BigInteger tab Syntax Buttons, page 3-20](#)

BigInteger Variables

The BigInteger Variables selection box lists all the BigInteger variables contained in the currently opened script. Use this selection box to paste an already defined BigInteger variable into an expression.

The BigInteger variable represents arbitrary-precision integers. The default value of a BigInteger variable is zero, that is, 0IB.

BigInteger Constructors, Methods, and Attributes

Use the appropriate selection box to add a public BigInteger constructor, method, or attribute in your Cisco Unified CCX script expression.

The available public methods and attributes include both static and non static ones.

For descriptions of the public Java BigInteger constructors, methods, and attributes available in the selection box, see <http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigInteger.html>.

BigInteger tab Syntax Buttons

The BigInteger tab syntax buttons indicate all the ways you can add or modify a BigInteger in an expression in a Cisco Unified CCX script. Clicking on one of the buttons adds the indicated syntax to your expression. The Question marks are not added to the expression when you click the syntax button. You need to substitute them with the appropriate values in the expression.



Note All of the BigInteger syntax listed in [Table 3-4](#) is specific to the Cisco Unified CCX Expression Language.

The use of IB to specify a BigInteger and the use of math operators on BigIntegers is specific to the Expression Language and is not a part of the Java language syntax.

The semantics of arithmetic operations exactly mimic those of Java's integer arithmetic operators, as defined in The Java Language Specification. See the following for a summary descriptive list of all the operators you can use in the Java language:

<http://java.sun.com/docs/books/tutorial/java/nutsandbolts/opsummary.html>

Table 3-4 *BigInteger Syntax Button Descriptions*

Syntax Button	Name	Type	Description
?IB	literal	decimal	A BigInteger literal in decimal format. For example: 234556789IB 0IB -23IB 21474836482147483648IB

Table 3-4 BigInteger Syntax Button Descriptions (continued)

Syntax Button	Name	Type	Description
0x?IB	literal	hexadecimal	A BigInteger literal in hexadecimal format. For example: 0x10000000000000000000IB
+?	unary plus	unary	The positive value of the operand.
-?	unary minus		The negative value of the operand.
++? ¹	prefix increment	increment	Increments the value of the operand by one before the operand is changed in an expression.
?++ ¹	postfix increment		Increments the value of the operand by one after the operand is changed in an expression.
--? ¹	prefix decrement	decrement	Decrements the value of the operand by one before the operand is changed in an expression.
?-- ¹	postfix decrement		Decrements the value of the operand by one after the operand is changed in an expression.
? + ?	addition	arithmetic	Adds two operands.
? - ?	subtraction		Subtracts the second operand from the first.
? * ?	multiplication		Multiplies two operands.
? / ?	division		Divides the first operand by the second.
? % ?	remainder		Returns the remainder of the first operand divided by the second.
? << ?	shift left	bitwise shift (for operations on individual bits in integers only)	Shifts bits of operand 1 left by the distance of operand 2; fills with zero bits on the right-hand side.
? >> ?	shift right		Shifts bits of operand 1 right by the distance of operand 2; fills with the highest (signed) bit on the left-hand side.
? >>> ?	zero fill right shift		Shifts bits of operand 1 right by a distance of operand 2; fills with zero bits on the left-hand side.
? & ?	bitwise AND	bitwise logical (for operations on individual bits in integers only)	Compares both operands. If both operand bits are 1, the AND function sets the resulting bit to 1; otherwise, the resulting bit is 0.
? ^ ?	bitwise exclusive OR (XOR)		Compares both operands. If both operand bits are different, the resulting bit is 1; otherwise the resulting bit is 0.
? ?	bitwise inclusive OR		Compares both operands. If either of the two operand bits is 1, the resulting bit is 1. Otherwise, the resulting bit is 0.
~ ?	Bitwise complement		Inverts the value of each operand bit: If the operand bit is 1, the resulting bit is 0; if the operand bit is 0, the resulting bit is 1.

Table 3-4 BigInteger Syntax Button Descriptions (continued)

Syntax Button	Name	Type	Description
? *= ?	multiply and assign	assignment	Multiplies the first operand by the second and assigns the result to the first operand.
? /= ?	divide and assign	The operand on the left of the assignment	Divides the first operand by the second and assigns the result to the first operand.
? %= ?	remainder and assign	statement (the first operand) can be any type of variable, including an	Divides the first operand by the second operand and assigns the remainder to the first operand.
? += ?	add and assign	array component or a public class attribute.	Adds the first operand to the second operand and assigns the result to the first operand.
? -= ?	subtract and assign		Subtracts the second operand from the first operand and assigns the result to the first operand.
? <<= ?	left shift and assign		Shifts bits of operand 1 left by the distance of operand 2; fills with zero bits on the right-hand side and assigns the resulting bit to operand 1.
? >>= ?	right shift and assign		Shifts bits of operand 1 left by the distance of operand 2; fills with zero bits on the right-hand side and assigns the resulting bit to operand 1.
? >>>= ?	zero fill, right shift, and assign		Shifts bits of operand 1 right by a distance of operand 2; fills with zero bits on the left-hand side and assigns the resulting bit to operand 1.
? &= ?	AND and assign	Assignment (continued)	First, compares both operands. If both operand bits are 1, the AND function sets the resulting bit to 1; otherwise, the resulting bit is set to 0. Then, assigns the resulting bit to operand 1.
? ^= ?	XOR and assign		First, compares both operands. If both operand bits are different, the resulting bit is 1; otherwise the resulting bit is 0. Then, assigns the resulting bit to operand 1.
? = ?	OR and assign		First, compares both operands. If either of the two operand bits is 1, the resulting bit is 1. Otherwise, the resulting bit is 0. Then, assigns the resulting bit to operand 1.

1. The operand for the prefix and postfix increment operators must be a variable, an array component, or a public class attribute.

Boolean

Use the Boolean tab to add or modify Boolean data in an expression. “boolean” is a friendly data type corresponding to the fully qualified `java.lang.Boolean` class.

**Note**

In the Expression Language, `boolean` and `Boolean` can be used interchangeably as opposed to Java where `boolean` represents a primitive data type and `Boolean` represents an object.

This topic includes the following:

- [About Booleans, page 3-23](#)
- [Boolean Specification on the Web, page 3-23](#)
- [Example Complex Expression Using a Boolean, page 3-23](#)
- [Boolean Variables, page 3-24](#)
- [Boolean Constructors, Methods, and Attributes, page 3-25](#)
- [Boolean tab Syntax Buttons, page 3-25](#)
- [Boolean Literals, page 3-27](#)

About Booleans

A Boolean variable has one of two values: `true` or `false`. The words `true` and `false` are also reserved words, are case insensitive, and are called Boolean literals.

These variables are not the same as the strings `true` and `false` nor are they the same as any numeric value like 1 or 0. Booleans are not numbers or strings. They are simply Booleans.

**Note**

The `Boolean` class is spelled with an initial capital letter, but the Java `boolean` data type is all lowercase.

For examples of how you can use Booleans, see [Example Complex Expression Using a Boolean, page 3-23](#) and [Boolean tab Syntax Buttons, page 3-25](#).

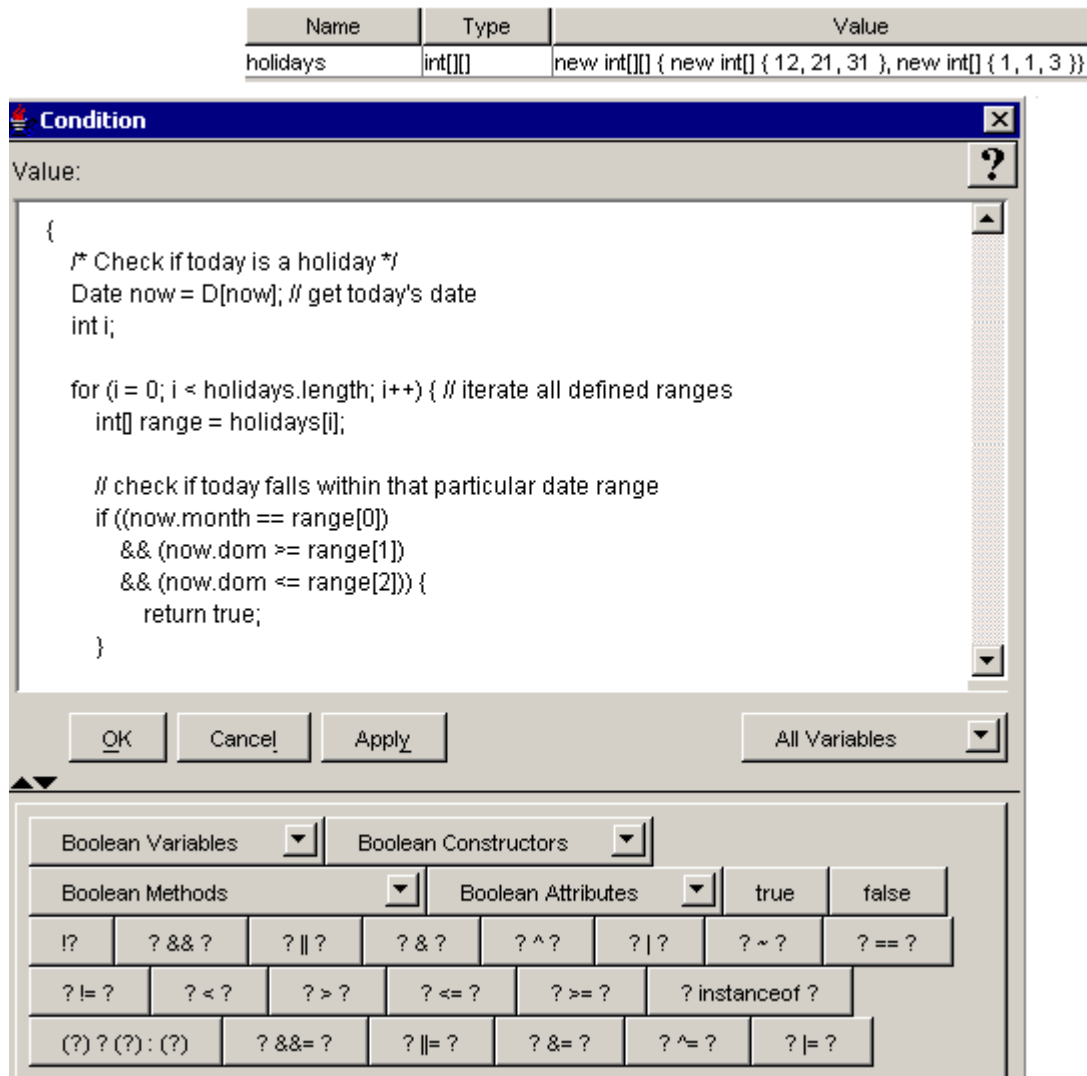
Boolean Specification on the Web

For the Sun Java specification on Booleans, see <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Boolean.html>.

Example Complex Expression Using a Boolean

In the following example, the script variable used in the expressions is listed in the top right of the example.

Figure 3-9 Example Complex Expression Using a Boolean



The following sections describe the options on the Boolean tab:

- [Boolean Variables, page 3-24](#)
- [Boolean Constructors, Methods, and Attributes, page 3-25](#)
- [Boolean tab Syntax Buttons, page 3-25](#)

Boolean Variables

The Boolean Variable selection box lists all the Boolean variables defined in the open Cisco Unified CCX script. Use this selection box to paste an already defined Boolean variable into an expression.

A Boolean variable can be either true or false, and is primarily used by the If step in the General palette of the Cisco Unified CCX Editor or any of the conditional operators.

The default value of a Boolean variable is false.

Boolean Constructors, Methods, and Attributes

Use the appropriate selection box to add a public Boolean constructor, method, or attribute in your expression.

The available public methods and attributes include both static and non static ones.

For descriptions of the public Java Boolean constructors, methods, and attributes available in the selection box, see <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Boolean.html>.

Boolean tab Syntax Buttons

The Boolean tab syntax buttons indicate all the ways you can add or use a Boolean in an expression. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values.

The semantics of Boolean operations exactly mimic those of Java's Boolean operators, as defined in The Java Language Specification. See the following for a summary descriptive list of all the operators you can use in the Java language:

<http://java.sun.com/docs/books/tutorial/java/nutsandbolts/opsummary.html>

Table 3-5 Boolean Syntax Button Descriptions

Syntax Button	Operator	Operator Type	Description
true	literal	Boolean	The Boolean literal corresponding to the primitive value true. See also Boolean Literals, page 3-27 .
false	literal	Boolean	The Boolean literal corresponding to the primitive value false. See also Boolean Literals, page 3-27 .
!?	boolean NOT	logical	Returns true if the operand is false.
? && ?	boolean AND		Compares both operands. Returns true if both operand 1 and operand 2 are true.
? ?	boolean OR		Compares both operands. Returns true if either operand 1 or operand 2 is true.

Table 3-5 Boolean Syntax Button Descriptions (continued)

Syntax Button	Operator	Operator Type	Description
? & ?	bitwise AND	bitwise logical	Compares both operands. Returns true if operand 1 and operand 2 are both boolean and both true; always evaluates operand 1 and operand 2.
? ^ ?	bitwise exclusive OR		Compares both operands. Returns true if operand 1 and operand 2 are different — that is, if one or the other of the operands, but not both, is true.
? ?	bitwise inclusive OR		Compares both operands. Returns true if both operand 1 and operand 2 are boolean and either operand 1 or operand 2 is true; always evaluates operand 1 and operand 2.
~ ?	bitwise complement		Inverts the value of each operand bit: If the operand bit is 1 (true), the resulting bit is 0 (false); if the operand bit is 0 (false), the resulting bit is 1 (true).
? == ?	equal to	conditional equality	Returns true if operand 1 and operand 2 are equal.
? != ?	not equal to	(For Java objects, equality is determined by invoking the equals() method on the first operand with the second operand as argument.)	Returns true if operand 1 and operand 2 are not equal.
? < ? ¹	less than	conditional relation	Returns true if operand 1 is less than operand 2.
? > ? ¹	greater than		Returns true if operand 1 is greater than operand 2.
? <= ? ^{1,2}	less than or equal to		Returns true if operand 1 is less than or equal to operand 2.
? >= ? ^{1,2}	greater than or equal to		Returns true if operand 1 is greater than or equal to operand 2.
? instanceof ?	instance of	conditional instance	Returns true if operand 1 is an instance of the class represented by operand 2.
(?) ? (?): (?)	if then else	conditional true and else	If operand 1 is true, returns operand 2. Otherwise, returns operand 3.

Table 3-5 Boolean Syntax Button Descriptions (continued)

Syntax Button	Operator	Operator Type	Description
? &&= ?	logical AND	relational and assignment	Returns true if operand 1 and operand 2 are both true and assigns operand 2 to operand 1; conditionally evaluates operand 2.
? = ?	logical OR and assign	(The operand on the left of the assignment statement (the first operand) can be any type of variable, including an array component or a public class attribute.)	Returns true if either operand 1 or operand 2 is true and then assigns operand 2 to operand 1; conditionally evaluates operand 2.
? &= ?	AND and assign		First, compares both operands. If both operand bits are true, the AND function sets the resulting bit to true (1); otherwise, the resulting bit is set to false (0). Then, assigns the resulting bit to operand 1.
? ^= ?	XOR and assign		First, compares both operands. If both operand bits are different, the resulting bit is true (1); otherwise the resulting bit is false (0). Then, assigns the resulting bit to operand 1.
? = ?	OR and assign		First, compares both operands. If either of the two operand bits is true (1), the resulting bit is true (1). Otherwise, the resulting bit is false (0). Then, assigns the resulting bit to operand 1.

1. For Java objects which are instances of the `java.lang.Comparable` interface, comparison is determined by invoking the `compareTo()` method on the first operand with the second operand as argument.
2. For Java objects which are not instances of the `java.lang.Comparable` interface, comparison can only verify equality as with the `==` or `!=` operators by invoking the `equals()` method on the first operand with the second operand as argument. The operator returns true only if `equals()` returns true. As such, only equality is being verified and not greater or less.

Boolean Literals

The Boolean type has two values, represented by the literals `true` and `false`, formed from ASCII letters. A Boolean literal is always of type `Boolean` and is case insensitive.

BooleanLiteral: one of
any case from: `true` or `false`

Each Boolean literal is a reference to an instance of class `Boolean`. These objects have a constant value and can be used interchangeably with its counter part Java primitive data type when calling methods that expect the primitive types or when accessing Java attributes declared using the Java primitive data type.

Byte

Use the Byte tab to enter or modify byte data in an expression. A byte is a friendly data type corresponding to the fully qualified `java.math.Byte` class name.



Note

In the Expression Language, `byte` and `Byte` can be used interchangeably as opposed to Java where `byte` represents a primitive data type and `Byte` represents an object.

This topic includes the following:

- [About Bytes, page 3-28](#)
- [Byte Java Specification on the Web, page 3-28](#)
- [Example Simple Expression Use the Byte Data Type, page 3-29](#)
- [Byte Variables, page 3-29](#)
- [Byte Constructors, Methods, and Attributes, page 3-29](#)
- [Byte tab Syntax Buttons, page 3-29](#)

About Bytes

A byte is an integral type of eight bits and is the smallest addressable numeric unit of storage.

The byte data type does not support literals. As such, one can use integer literal and type cast them to byte using the (byte) type cast operator as long as the value of the integer literal does not exceed the capacity of a byte.

The Java numeric types are the integral types and the floating-point types:

- The integral types are byte, short, int, and long, whose values are 8-bit, 16-bit, 32-bit and 64-bit signed two's-complement integers, respectively, and char, whose values are 16-bit unsigned integers representing UTF-16 code units.
- The floating-point types are float, whose values include the 32-bit IEEE 754 floating-point numbers, and double, whose values include the 64-bit IEEE 754 floating-point numbers.

You can convert a byte to a string and a string to a byte.

For examples of how you can use bytes, see [Example Simple Expression Use the Byte Data Type, page 3-29](#) and [Byte tab Syntax Buttons, page 3-29](#).

Byte Java Specification on the Web

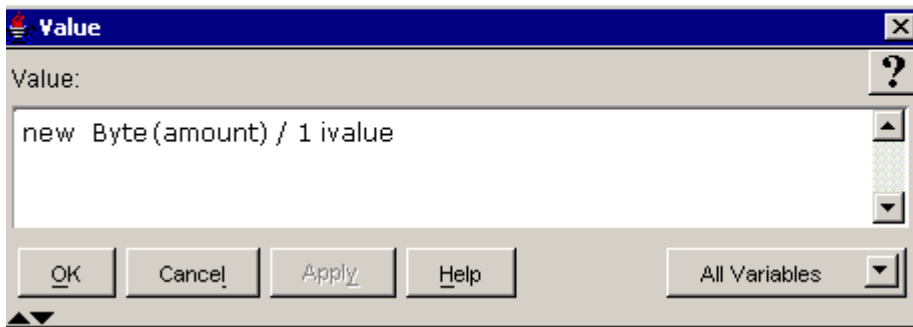
For the Sun Java specification on bytes, see

<http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Byte.html>

Example Simple Expression Use the Byte Data Type

Figure 3-10 Example Simple Expression Using a Byte and Script Variables

Name	Type	Value
amount	String	"243"
ivalue	byte	(byte)10



The following sections describe the options on the Byte tab:

- [Byte Variables, page 3-29](#)
- [Byte Constructors, Methods, and Attributes, page 3-29](#)
- [Byte tab Syntax Buttons, page 3-29](#)

Byte Constructors, Methods, and Attributes

Use the appropriate selection box, to add a public byte constructor, method, or attribute into your expression.

The available public methods and attributes include both static and non static ones.

For descriptions of the public Java Byte constructors, methods, and attributes available in the selection box, see <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Byte.html>.

Byte Variables

The Byte Variables selection box lists all the Byte variables contained in the currently opened script. Use this selection box to paste an already defined Byte variable into an expression.

A Byte variable holds the value of a Byte, which represents an 8-bit integer value with a value range from -128 to +127. The default value of a Byte variable is zero, that is, the value of (byte)0.

Byte tab Syntax Buttons

The Byte tab syntax buttons indicate all the ways you can add or use a Byte in an expression. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values.

The semantics of Byte operations exactly mimic those of Java's Byte operators, as defined in The Java Language Specification. See the following for a summary descriptive list of all the operators you can use in the Java language:

<http://java.sun.com/docs/books/tutorial/java/nutsandbolts/opsummary.html>

Table 3-6 Byte Syntax Button Descriptions

Syntax Button	Name	Type	Description
(byte)?	Byte typecast	typecast	Converts the operand value into a byte value by ignoring the information that exceeds the byte representation. For example: (byte)23 (byte)-45
+?	unary plus	unary	The positive value of the operand.
-?	unary minus		The negative value of the operand.
++? ¹	prefix increment	increment	Increments the value of the operand by one before the operand is changed in an expression.
?++ ¹	postfix increment		Increments the value of the operand by one after the operand is changed in an expression.
--? ¹	prefix decrement	decrement	Decrements the value of the operand by one before the operand is changed in an expression.
?-- ¹	postfix decrement		Decrements the value of the operand by one after the operand is changed in an expression.
? + ?	addition	arithmetic	Adds two operands.
? - ?	subtraction		Subtracts the second operand from the first.
? * ?	multiplication		Multiplies two operands.
? / ?	division		Divides the first operand by the second.
? % ?	remainder		Returns the remainder of the first operand divided by the second.
? << ?	shift left	bitwise shift (for operations on individual bits in integers only)	Shifts bits of operand 1 left by the distance of operand 2; fills with zero bits on the right-hand side.
? >> ?	shift right		Shifts bits of operand 1 right by the distance of operand 2; fills with the highest (signed) bit on the left-hand side.
? >>> ?	zero fill right shift		Shifts bits of operand 1 right by a distance of operand 2; fills with zero bits on the left-hand side.

Table 3-6 Byte Syntax Button Descriptions (continued)

Syntax Button	Name	Type	Description
? & ?	bitwise AND	bitwise logical (for operations on individual bits in integers only)	Compares both operands. If both operand bits are 1, the AND function sets the resulting bit to 1; otherwise, the resulting bit is 0.
? ^ ?	bitwise exclusive OR (XOR)		Compares both operands. If both operand bits are different, the resulting bit is 1; otherwise the resulting bit is 0.
? ?	bitwise inclusive OR		Compares both operands. If either of the two operand bits is 1, the resulting bit is 1. Otherwise, the resulting bit is 0.
~ ?	Bitwise complement		Inverts the value of each operand bit: If the operand bit is 1, the resulting bit is 0; if the operand bit is 0, the resulting bit is 1.
? *= ?	multiply and assign	assignment The operand on the left of the assignment statement (the first operand) can be any type of variable, including an array component or a public class attribute.	Multiplies the first operand by the second and assigns the result to the first operand.
? /= ?	divide and assign		Divides the first operand by the second and assigns the result to the first operand.
? %= ?	remainder and assign		Divides the first operand by the second operand and assigns the remainder to the first operand.
? += ?	add and assign		Adds the first operand to the second operand and assigns the result to the first operand.
? -= ?	subtract and assign		Subtracts the second operand from the first operand and assigns the result to the first operand.
? <<= ?	left shift and assign		Shifts bits of operand 1 left by the distance of operand 2; fills with zero bits on the right-hand side and assigns the resulting bit to operand 1.
? >>= ?	right shift and assign		Shifts bits of operand 1 left by the distance of operand 2; fills with zero bits on the right-hand side and assigns the resulting bit to operand 1.
? >>>= ?	zero fill, right shift, and assign	assignment (continued)	Shifts bits of operand 1 right by a distance of operand 2; fills with zero bits on the left-hand side and assigns the resulting bit to operand 1.

Table 3-6 Byte Syntax Button Descriptions (continued)

Syntax Button	Name	Type	Description
? &= ?	AND and assign		First, compares both operands. If both operand bits are 1, the AND function sets the resulting bit to 1; otherwise, the resulting bit is set to 0. Then, assigns the resulting bit to operand 1.
? ^= ?	XOR and assign		First, compares both operands. If both operand bits are different, the resulting bit is 1; otherwise the resulting bit is 0. Then, assigns the resulting bit to operand 1.
? = ?	OR and assign		First, compares both operands. If either of the two operand bits is 1, the resulting bit is 1. Otherwise, the resulting bit is 0. Then, assigns the resulting bit to operand 1.

1. With a prefix or postfix operator, the first operand must be a variable, an array component, or a class attribute.

Character

Use the Character tab to add or modify Character data in an expression. A char is a friendly data type corresponding to the fully qualified `java.lang.Character` class.



Note

In the Expression Language, `char` and `Character` can be used interchangeably as opposed to Java where `char` represents a primitive data type and `Character` represents an object.

This topic includes the following:

- [About the Character Data Type](#), page 3-32
- [Character Specification on the Web](#), page 3-33
- [Example Character Code](#), page 3-33
- [Character Methods and Attributes](#), page 3-34
- [Character tab Syntax Buttons](#), page 3-35
- [Character Literals](#), page 3-35
- [Escape Character Literals](#), page 3-36

About the Character Data Type

The `Character` class provides several methods for determining a character's category (lowercase letter, digit, and so on) and for converting a character from uppercase to lowercase and vice versa.

The `char` data type represents 16-bit Unicode characters. These are a superset of the ASCII character set which allow non-English language characters. Any Unicode character can be written as a literal using the Escape character (backslash `\`) and the "u" character followed by its hexadecimal representation. For example, `\u0065` represents the letter e.

The methods and data of class Character are defined by the information in the UnicodeData file standard that is part of the Unicode Character Database maintained by the Unicode Consortium. This file and its description are available from the Unicode Consortium at:

<http://www.unicode.org>

For examples of character code, see [Example Character Code](#), page 3-33 and [Character tab Syntax Buttons](#), page 3-35.

Character Specification on the Web

For the Sun Java specification on characters, see <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Character.html>.

Example Character Code

In the following two examples, the script variable used in the expression are listed in the top right of each example.

Figure 3-11 Example Simple Expression Using Character Code

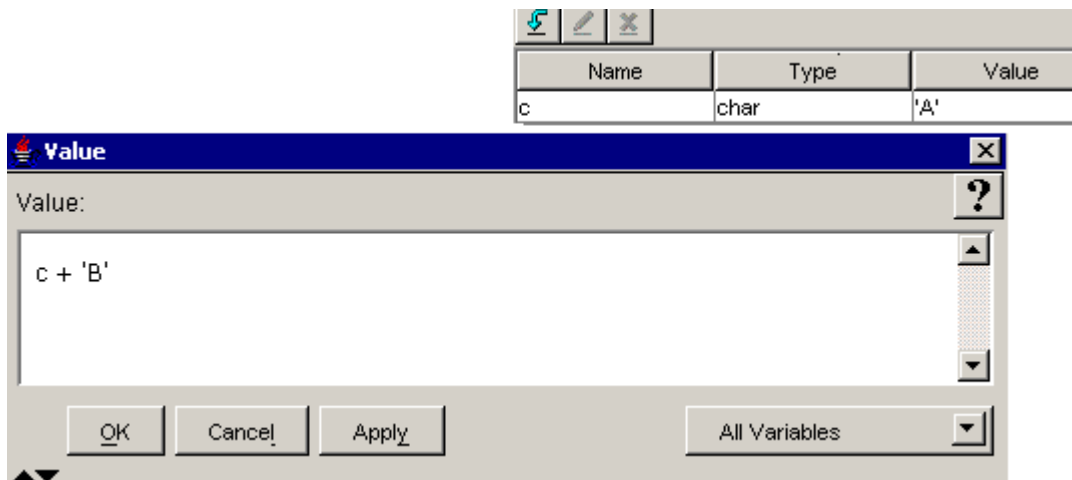
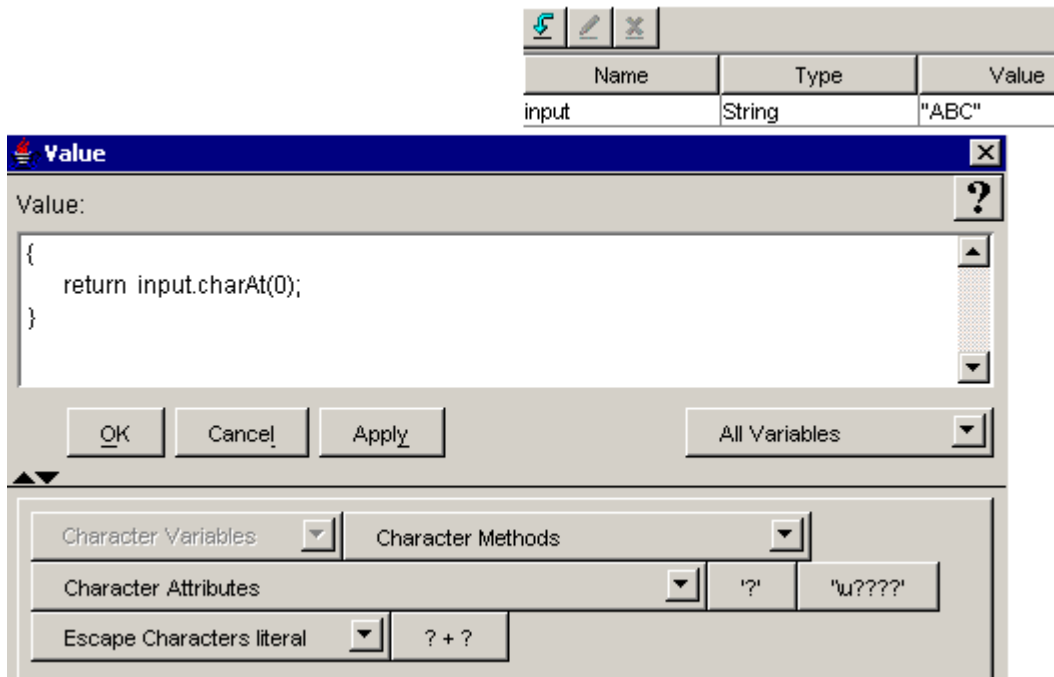


Figure 3-12 Example Complex Expression Using Character Code



The following sections describe the options on the Character tab:

- [Character Methods and Attributes, page 3-34](#)
- [Character Variables, page 3-34](#)
- [Character tab Syntax Buttons, page 3-35](#)
- [Character Literals, page 3-35](#)
- [Escape Character Literals, page 3-36](#)

Character Methods and Attributes

Use the appropriate selection box to add a character variable, method, or attribute, or an escape character to your expression.

The available public methods and attributes include both static and non static ones.

For descriptions the public Java char constructors, methods, and attributes available in the selection boxes, see <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Character.html>.

Character Variables

A character variable holds the value of a char and consists of characters, such as the letters in an alphabet. Its range of values is from '\u0000' to '\uffff' inclusive.

The default value of a Character variable is the null character, that is, '\u0000' or '\0'.

The Character Variable selection box lists all the character variables contained in the currently opened script. Use this selection box to paste an already defined character variable into an expression.

Character tab Syntax Buttons

The Character tab syntax buttons indicate all the ways you can insert or modify a char in an expression. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values.

Table 3-7 Character Syntax Button Descriptions

Syntax Button	Name	Type	Description
'?'	character literal	ASCII code for a character literal	<p>Inserts single quotes for entering a single character literal (which includes characters for escape sequences).</p> <p>For example: 'a', '%', '1', 'Z', 'Ω', '⊗', '\t', '\r', '\0', '\n', '\f', '\\', '\'</p> <p>See Character Literals, page 3-35.</p>
'\u????'		Unicode for a character literal	<p>Inserts single quotes for entering a single character literal (which includes characters for escape sequences) in a Unicode representation.</p> <p>For example: '\u03a9' '\uFFFF'</p> <p>See Character Literals, page 3-35.</p>
? + ?	concatenation (specific to Cisco Unified CCX)	string	<p>Concatenates two characters together to form a new string with these two characters in it.</p> <p>See String Concatenation Operator +, page 1-10.</p>

Character Literals

A character literal is expressed as a character or an escape sequence, enclosed in ASCII single quotes. (The single-quote, or apostrophe, character is \u0027.)

A character literal is always of type char.

```
CharacterLiteral:
  ' SingleCharacter '
  ' EscapeSequence '
SingleCharacter:
  UnicodeInputCharacter but not ' or \
```

The escape sequences are described in [Escape Character Literals, page 3-36](#).

The characters CR and LF are never an InputCharacter; they are recognized as constituting a LineTerminator.

ou will receive a parse-time error if the character following the SingleCharacter or EscapeSequence is other than a '. You will also receive a parse-time error if a line terminator appears after the opening ' and before the closing '.

The following are examples of char literals:

```
'a'
'%'
'\t'
'\'
'\''
'\u03a9'
'\uFFFF'
'\177'
'⊙'
'⊗'
```

Each char literal is a reference to an instance of class Character. These objects have a constant value and can be used interchangeably with its counter part Java primitive data type when calling methods that expect the primitive types or when accessing Java attributes declared using the Java primitive data type.

Escape Character Literals

Character literals for Escape Sequence:

```
\ b /* \u0008: backspace BS */
\ t /* \u0009: horizontal tab HT */
\ n /* \u000a: linefeed LF */
\ f /* \u000c: form feed FF */
\ r /* \u000d: carriage return CR */
\ " /* \u0022: double quote " */
\ ' /* \u0027: single quote ' */
\ \ /* \u005c: backslash \ */
\ 0 /* \u0000: null character */
\ UnicodeInputCharacter/* the actual Unicode character */
```



Note

Before you insert a new line ('\n') or a carriage return ('\r') in the Email body, ensure that you have a character; such as a period, colon, or a Cisco Unified CCX variable.

Table 3-8

Escape Character Descriptions

Escape ASCII Character	Escape Unicode Character	Description
'\t'	'\u0009'	tab
'\f'	'\u000c'	form feed
'\r'	'\u000d'	return
'\n'	'\u000a'	newline
'\0'	'\u0000'	null character
'\"'	'\u0027'	single quote
'\"'	'\u0022'	double quote

Currency

Use the Currency tab to add or modify Currencies in an expression.

The Currency friendly data type corresponds to the Java `com.cisco.util.Currency` class and not to the Java `java.util.Currency` class.

A Currency object represents a specific currency unit for a country. An operation that requires a Currency to perform its task is called currency-sensitive and uses the Currency to tailor information for the user. For example, playing back a dollar amount is a currency-sensitive operation and the amount must be formatted according to the currency conventions.

This topic includes the following:

- [About Currencies, page 3-37](#)
- [Currency Specification and Code List on the Web, page 3-37](#)
- [Example Simple Expression Using a Prompt and Currency, page 3-38](#)
- [Currency Variables, page 3-38](#)
- [Currency Methods and Attributes, page 3-38](#)
- [Recent Currencies, page 3-38](#)
- [Currency tab Syntax Button, page 3-39](#)
- [Currency Literals, page 3-39](#)

About Currencies

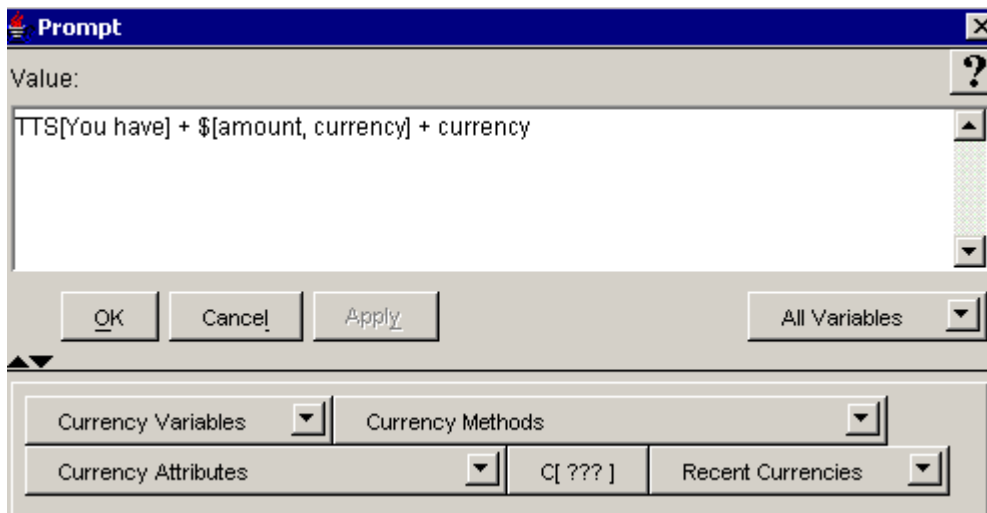
You can enter into a script a currency code from the ISO 4217 standard lists of currency codes to specify a country's currency.

The ISO 4217 standard internationally represents the currencies of countries throughout the world. In most cases, the currency code is composed of a country's two-character Internet country code plus an extra character to denote the currency unit. For example, the code for the Canadian Dollars is simply Canada's two-character Internet code ("CA") plus a one-character currency designator ("D").

Currency Specification and Code List on the Web

For a list of the ISO currency codes by country and precious metal or by code, see <http://www.xe.com/iso4217.htm>.

Example Simple Expression Using a Prompt and Currency



The following sections describe the options on the Currency tab:

- [Currency Variables, page 3-38](#)
- [Currency Methods and Attributes, page 3-38](#)
- [Recent Currencies, page 3-38](#)
- [Currency tab Syntax Button, page 3-39](#)

Currency Variables

The Currency Variable selection box lists all the currency variables contained in the currently opened script. Use this selection box to paste an already defined currency variable into an expression.

The Currency variable is used to identify a given currency, such as the American Dollar (USD), and is useful when creating generated currency prompts that need to be tailored based on a given currency.

The default value of a currency variable is the system default currency.

Currency Methods and Attributes

Use the appropriate selection box to add currency methods or attributes to your expression.

The available public methods and attributes include both static and non static ones.

For descriptions of the public Java Currency methods and attributes available in the selection boxes, see <http://java.sun.com/j2se/1.4.2/docs/api/java/util/Currency.html>.

Recent Currencies

Lists the currencies you have recently used in your script.

Currency tab Syntax Button

The C[???] button is for entering the currency of your choice with the “???” to be replaced by the ISO 4217 code for that currency.

For Example:

```
C[USD]// US Dollar currency
C[CAD]// Canadian Dollar currency
C[EUR]// Euro currency
```

Currency Literals

The currency literal is always of type Currency.

```
CurrencyLiteral:
  CurrencyDeclarator [CurrencyDesignator]
CurrencyDeclarator: one of
  c C
CurrencyDesignator:
  CurrencyLetter CurrencyLetter CurrencyLetter
CurrencyLetter:
  any from A to Z
```

The ISO 4217 standard requires the CurrencyDesignator to be defined as the upper-case three-letter code in ISO 4217.

Each currency literal is a reference to an instance of class `com.cisco.util.Currency`.

Date

Use the Date tab to enter or modify dates in an expression. Date is a friendly data type that corresponds to the `java.util.Date` class.

The Expression Editor formats the date and time according to the default locale.

This topic includes the following:

- [About Dates, page 3-39](#)
- [Date Specification on the Web, page 3-40](#)
- [Example Date Code, page 3-40](#)
- [Date Variables, page 3-41](#)
- [Date Constructors and Methods, page 3-42](#)
- [Date tab Syntax Buttons, page 3-42](#)
- [Date Literals, page 3-43](#)

About Dates

The Date class represents a specific instant in time with millisecond precision. For examples of how to enter a date, see [Example Date Code, page 3-40](#) and [Date tab Syntax Buttons, page 3-42](#).

Date Specification on the Web

For the Sun Java specification on dates, see <http://java.sun.com/j2se/1.4.2/docs/api/java/util/Date.html>.

Example Date Code

Figure 3-13 Example Simple Expression Using a Date

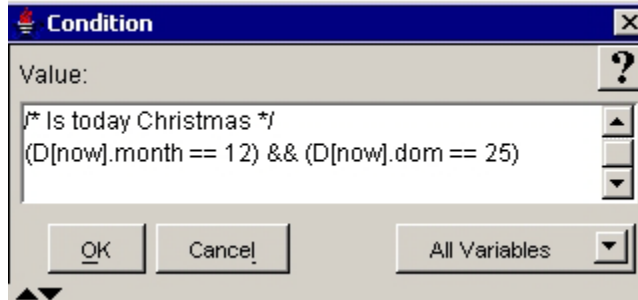


Figure 3-14 Example Complex Expression Using a Date and a Script Variable

Name	Type	Value
holidays	int[][]	new int[][] { new int[] { 12, 21, 31 }, new int[] { 1, 1, 3 } }

Condition

Value: ?

```
{
/* Check if today is a holiday */
Date now = D[now]; // get today's date
int i;

for (i = 0; i < holidays.length; i++) { // iterate all defined ranges
    int[] range = holidays[i];

    // check if today falls within that particular date range
    if ((now.month == range[0]
        && (now.dom >= range[1])
        && (now.dom <= range[2])) {
        return true;
    }
}
// if we get here then we didn't find any range in which today falls
return false;
}
```

OK Cancel Apply All Variables

Date Variables Date Constructors

Date Methods D[???] D[now] ?.year

?.month ?.woy ?.wom ?.date ?.dom ?.doy

?.dow ?.ampm ?.hour ?.hod ?.min ?.sec ?.ms

The following sections describe the options on the Date tab:

- [Date Variables, page 3-41](#)
- [Date Constructors and Methods, page 3-42](#)
- [Date tab Syntax Buttons, page 3-42](#)

Date Variables

The Date Variables selection box lists all the date variables contained in the currently opened script. Use this selection box to paste an already defined date variable into an expression.

The Date variable includes date information. The default value of the Date variable is the current date at the time of interpretation.

Date Constructors and Methods

Use the appropriate selection box to add a date constructor or method to your expression.

The available public methods and attributes include both static and non static ones.

For descriptions the public Java Date constructors and methods available in the selection boxes, see <http://java.sun.com/j2se/1.4.2/docs/api/java/util/Date.html>.

Date tab Syntax Buttons

The Date tab syntax buttons indicate all the ways you can add or use a Date in an expression. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values.



Note

The Date syntax specified by the Date tab syntax buttons is specific to Cisco Unified CCX.

Table 3-9 Date Syntax Button Descriptions

Syntax Button	Description
D[???	<p>Enters the date. You can specify a date in many different formats. See Date Literals, page 3-43 for the different formats you can use.</p> <p>Examples:</p> <pre>D[12/13/52] D[Dec 13, 1952] D[Thu, July 4, 2002] D[July 5, 2002] D[July 7, 2002] D[7/6/02] D[Thu, July 4, 2002 5:59 PM] D[July 5, 2002 5:59 PM] D[July 7, 2002 5:59 PM] D[7/6/02 5:59 PM] D[Thu, July 4, 2002 12:23:59 AM] D[July 5, 2002 12:23:59 AM] D[July 7, 2002 12:23:59 AM] D[7/6/02 12:23:59 AM] D[Thu, July 4, 2002 12:23:59 AM CST] D[July 5, 2002 12:23:59 AM CST] D[July 7, 2002 12:23:59 AM CST] D[7/6/02 12:23:59 AM CST]</pre>
D[now]	<p>Returns the current date and time at run-time. This is the date and time when the expression is evaluated and not when the expression is entered in the Cisco Unified CCX Editor.</p> <p>Returns the current date in the format Month Day, Year HH:MM:SS AM PM. For example: D[July 5, 2005 3:34:42 PM].</p>
?..year ¹	<p>Returns the current year of the date object as an int number.</p> <p>For example: 2005</p>
?..month ¹	<p>Returns a number representing the month that contains or begins with the instant in time represented by this Date object. The value returned is between 1 and 12, with the value 1 representing January.</p>

Table 3-9 Date Syntax Button Descriptions (continued)

Syntax Button	Description
?.woy ¹	Returns the week of the year of the date object. The range is 1 - 52.
?.wom ¹	Returns the week of the month of the date object. The range is 1 - 5.
?date ¹	Returns the current date of the date object. This date is the same as that specified by the .dom syntax.
?dom ¹	Returns the day of the month represented by this Date object. The value returned is between 1 and 31 representing the day of the month that contains or begins with the instant in time represented by this Date object, as interpreted in the local time zone.
?doy ¹	Returns the day of a date in a year as a number. The range is from 1 to 366.
?dow ¹	Returns the day of the week represented by this date. The returned value (1 = Sunday, 2 = Monday, 3 = Tuesday, 4 = Wednesday, 5 = Thursday, 6 = Friday, 7 = Saturday) represents the day of the week that contains or begins with the instant in time represented by this Date object, as interpreted in the local time zone.
?ampm ¹	Returns an int number of the date object; AM=0, PM=1
?hour ¹	Returns the hour represented by this Date object. The returned value is a number (0 through 12) representing the hour within the day that contains or begins with the instant in time represented by this Date object, as interpreted in the local time zone.
?hod ¹	Returns the hour represented by this Date object. The returned value is a number (0 through 23) representing the hour within the day that contains or begins with the instant in time represented by this Date object, as interpreted in the local time zone.
?min ¹	Returns the number of minutes past the hour represented by this date, as interpreted in the local time zone. The value returned is between 0 and 59.
?sec ¹	Returns the number of seconds past the minute represented by this date. The value returned is between 0 and 59.
?ms ¹	Returns the number of milliseconds since the second represented by the Date. The range is 0 to 999.

1. The .year, .month, .woy, .wom, .date, .dom, .doy, .dow, .ampm, .hour, .hod, .min, .sec, and .ms variables do not require the Java license.

Date Literals

The date literal is always of type `Date`. The following are the different syntax formats you can use to enter a date.

```
DateLiteral:
    DateDeclarator [n o w ]
    DateDeclarator [DateDesignator TimeDesignatoropt]
```

```
DateDeclarator [ DateDesignator TimeDesignatoropt ]
```

```
DateDeclarator: one of
    d D
```

```
DateDesignator:
    FullDatePattern:
```

Defined by the pattern "EEEE, MMMM d, yyyy"

LongDatePattern:

Defined by the pattern "MMMM d, yyyy"

MediumDatePattern:

Defined by the pattern "MMM d, yyyy"

ShortDatePattern:

Defined by the pattern "M/d/yy"

TimeDesignator:

FullTimePattern:

Defined by the pattern "h:mm:ss a z"

LongTimePattern:

Defined by the pattern "h:mm:ss a z"

MediumTimePattern:

Defined by the pattern "h:mm:ss a"

ShortTimePattern:

Defined by the pattern "h:mm a"

Example Date Literals:

```
D[12/13/52]
D[Dec 13, 1952]
D[Thu, July 4, 2002]
D[July 5, 2002]
D[July 7, 2002]
D[7/6/02]
D[Thu, July 4, 2002 5:59 PM]
D[July 5, 2002 5:59 PM]
D[July 7, 2002 5:59 PM]
D[7/6/02 5:59 PM]
D[Thu, July 4, 2002 12:23:59 AM]
D[July 5, 2002 12:23:59 AM]
D[July 7, 2002 12:23:59 AM]
D[7/6/02 12:23:59 AM]
D[Thu, July 4, 2002 12:23:59 AM CST]
D[July 5, 2002 12:23:59 AM CST]
D[July 7, 2002 12:23:59 AM CST]
D[7/6/02 12:23:59 AM CST]
D[now]
```

Details for the date and time patterns are available in the documentation of the `java.text.DateFormat` class. If the string `now` is used, then the literal corresponds to the current date in the server's default timezone at the time the literal is evaluated for the first time.

Each date literal is a reference to an instance of class `java.util.Date`.

Document

Use the Document tab to add documents to an expression. The Document friendly data type corresponds to the Java `com.cisco.doc.Document` class.

This topic includes the following:

- [About Expression Language Documents, page 3-45](#)
- [Example Expression Using a Document, page 3-45](#)
- [Document Variables, page 3-45](#)
- [Browse Documents Dialog Box, page 3-46](#)
- [Document tab Syntax Buttons, page 3-46](#)
- [Document Literals, page 3-48](#)

- [Time of Week, Day of Week, and Time of Day Documents, page 3-52](#)

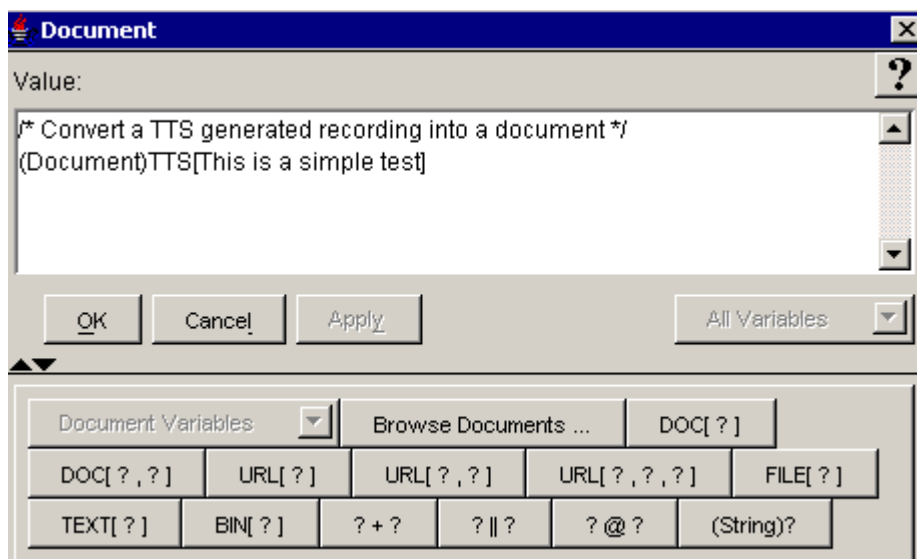
See also [Document Conversions, page 1-32](#).

About Expression Language Documents

In the Cisco Unified CCX Expression Language, instances of class Document represent documents located somewhere that can be accessed for various reasons. A Document object has a constant (unchanging) value. Complex document literals are references to instances of class Document.

A Document object can be an HTML or XML document, or a file, or a user or system document, and so on. For a list of all valid Document objects, see [Document Literals, page 3-48](#).

Example Expression Using a Document



The following sections describe the options you can use in the Expression Editor Document tab:

- [Document Variables, page 3-45](#)
- [Browse Documents Dialog Box, page 3-46](#)
- [Document tab Syntax Buttons, page 3-46](#)

Document Variables

The Document Variable selection box lists all the document variables contained in the currently opened script. Use the Document Variables selection box to paste a document variable into an expression.

A Document variable can be of any type of document, such as a file, a URL, or a recording. The default value of a Document variable is the empty document, that is, DOC[]

Browse Documents Dialog Box

Use the Browse Documents selection box to add a Document from disk or from the Document repository to your script expression.

Document tab Syntax Buttons

The Document tab syntax buttons indicate all the ways you can add a Document object to an expression. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values. For operations you can perform on documents, see [Operators Used with Prompts and Documents, page 1-8](#).

<http://java.sun.com/j2se/1.4.2/docs/api/java/net/URL.html>

Table 3-10 Document Syntax Button Descriptions

Syntax Button	Document Type or Operation	Description
DOC[?]	user-defined literal	A user-defined document in the document repository. See User Document Literals, page 3-51 . For example: DOC[AA\schedule.doc] DOC[rootTemplateDir + "templateA.txt"]
DOC[?,?]		A user-defined document in the document repository with associated contact (additional arguments, more than one is allowed). The additional argument(s) is passed into the document template (if one or more are referenced) as initial arguments for evaluation by the document template. See User Document Literals, page 3-51 . For example: DOC["vxml\application1.vxml", mainCall]

Table 3-10 Document Syntax Button Descriptions (continued)

Syntax Button	Document Type or Operation	Description
URL[?]	URL document literal	See URL Document Literals, page 3-49 . For example: URL[http://www.cisco.com/index.html] URL[http://evbuweb/mydoc.asp?number=23]
URL[?,?]		A URL document with output document and time out. See User Document Literals, page 3-51 . For example: URL[http://www.cisco.com/index.html,2000L] The first operand is the URL as in the preceding example and the second operand is either an output document to be sent to the referenced URL as a post or a time out. The time out is used only in the case of HTTP URLs as defined in the CreateURLDoc step and is in milliseconds.
URL[?,?,?]		A URL document with output document and time out. See User Document Literals, page 3-51 . For example: URL[http://www.cisco.com/index.html, MyDoc, 2000L] The first operand is the URL, the second is the output document to be sent to the referenced URL, and the third operand is the time out. The time out is used only in the case of HTTP URLs as defined in the CreateURLDoc step and is in milliseconds.
FILE[?]	file document literal	See File Document Literals, page 3-50 . For example: FILE[C:\Documents\mydoc.txt]
TEXT[?]	text document literal	See Text Document Literals, page 3-50 . For example: TEXT[Some text to be stored in a document]
BIN[?]	binary document literal	See Binary Document Literal, page 3-49 . For example: BIN[cafebabe34f3edca56b8001cdef] BIN[myArrayOfBytes]
? + ?	Concatenation operation	Concatenates characters, strings, or documents. See Additive Operators, page 1-9 and Document Concatenation Operator +, page 3-51 . For example: DOC[rootTemplateDir + "templateA.txt"]

Table 3-10 Document Syntax Button Descriptions (continued)

Syntax Button	Document Type or Operation	Description
? ?	document escalation operation	<p>The document escalation operator enables you to add document options to an expression depending on the time of the week, the day of the week, and the time of day.</p> <p>See Escalation Operator , page 1-11. See also Time of Week Document, page 3-52, Day of Week Document, page 3-53, or Time of Day Document, page 3-53.</p> <p>For example: DOC[D1.txt] @ MON @ T[10:59 AM] DOC[D2.txt] @ TUE @ T[11:58 PM] DOC[] @ MON @ T[1:00 PM]</p>
? @ ?	document qualification operation	<p>The document qualification operator @ qualifies how a document is to be run. This operator expects a qualifying expression of the following type:</p> <ul style="list-style-type: none"> • Language. Represents a language qualification and is used to temporarily override the language associated with a given document. The expression must be of type Language. Qualifying a document more than once with a language results in only the last one to be kept as the overridden language for the document. • DayOfWeekLiteral. Represents a day of week qualification and is used to specify the starting day of a possible range when the document is to be used in a day of week document or time of week document expression. <p>For example, the expression “DOC[D1.txt] @ MON” specifies that document D1.txt can be accessed on Monday:</p> <ul style="list-style-type: none"> • Number. Represents the starting day of the week where its value must evaluate to 1 for Sunday, 2 for Monday, and so on to 7 for Saturday. • Time. Represents time qualification and is used to specify the starting time of a possible range when the document is to be used in a time of day document or time of week expression. <p>For example, the expression “DOC[D1.txt] @ MON @ T[10:59 AM]” specifies that document D1.txt can be accessed on Monday from 10.59 AM:</p> <p>See also Time of Week, Day of Week, and Time of Day Documents, page 3-52.</p>
(String)?	cast conversion	Converts the specified document to a string. See Document Conversions , page 1-32.

Document Literals

The document literal is always of type Document.

```

DocumentLiteral:
  BinaryDocumentLiteral
  URLLiteral
  FileDocumentLiteral
  TextDocumentLiteral
  UserDocumentLiteral

```

Each document literal is a reference to an instance of a class that implements the interface `com.cisco.doc.Document`.

This section describes the following document literals:

- [Binary Document Literal, page 3-49](#)
- [URL Document Literals, page 3-49](#)
- [File Document Literals, page 3-50](#)
- [Text Document Literals, page 3-50](#)
- [User Document Literals, page 3-51](#)

Binary Document Literal

The binary document literal is always of type `Document`.

```

BinaryDocumentLiteral:
  BinaryDocumentDeclarator [ComplexLiteralInputChars]
  BinaryDocumentDeclarator [Expression]
BinaryDocumentDeclarator:
  any case for BIN

```

Binary document literals are used to represent a document located in memory using a hexadecimal text representation of the binary data.

The `ComplexLiteralInputChars` can include the [character as long as it has a balanced number of] characters; one for every [character found.

If the sequence of characters can be parsed to an `Expression` of type `(byte[])`, then the resulting document is a binary document where the expression specifies the content of the document.

If the sequence of characters cannot be parsed properly as described above, then it is considered to be whole binary content of the document represented in hexadecimal form where each hexadecimal character represents one nibble of data.

Example Binary Document Literals:

```

BIN[cafebabe34f3edca56b8001cdef]
BIN[myArrayOfBytes]

```

URL Document Literals

The URL document literal is always of type `Document`.

```

URLDocumentLiteral:
  URLLiteral [ComplexLiteralInputChars]
  URLLiteral [Expression]
  URLLiteral [Expression, Expression]
URLDocumentDeclarator:
  any case for URL

```

URL document literals are used to represent a document using a URL scheme.

The `ComplexLiteralInputChars` can include the [character as long as it has a balanced number of] characters: one for every [character found:

- If the sequence of characters can be parsed as an Expression of type String or java.net.URL, then the resulting document is a URL document where the expression specifies the URL from where to retrieve the document.
- If the sequence of characters can be parsed as two Expressions where the first one must have type String or java.net.URL and the second one must have type Document, then the resulting document is a URL document where the first argument specifies the URL of the document and the second one represents a document that is sent to the destination specified by the URL. This, for example, can be used to upload documents to a web server.
- If the sequence of characters cannot be parsed properly as described above, then it is considered to be the URL from where to retrieve the document.

Example URL Document Literals:

```
URL[http://localhost/index.html]
URL[ftp://12.12.12.12:8080/schedule.doc]
URL["http://www.cisco.com/index.html"]
URL[new java.net.URL("http", "www.cisco.com", "index.html")]
URL[myURI + "index.html", DOC[docs\fax.txt]]
```

File Document Literals

The file document literal is always of type Document.

```
FileDocumentLiteral:
  FileDocumentDeclarator [ComplexLiteralInputChars]
  FileDocumentDeclarator [Expression]
FileDocumentDeclarator:
  any case for FILE
```

File document literals are used to represent a document located on the local disk:

- If the sequence of characters can be parsed as an expression of type String or java.io.File, then the resulting document is a file document where the expression specifies the filename where to retrieve the document from.
- If the sequence of characters cannot be parsed properly as described above, then it is considered to be the filename where to retrieve the document from.

Example File document Literals:

```
FILE[C:\Program Files\wfvavvid\lib\CiscoUtil.jar]
FILE[schedule.doc]
FILE[root + u"\help.txt"]
FILE[new File(rootDirectory, "template.txt")]
```

Text Document Literals

The text document literal is always of type Document.

```
TextDocumentLiteral:
  TextDocumentDeclarator [ComplexLiteralInputChars]
  TextDocumentDeclarator [Expression]
TextDocumentDeclarator:
  any case for TEXT
```

Text document literals are used to represent a document located in memory using a text string:

- If the sequence of characters can be parsed as an expression of type String, then the resulting document is a text document where the expression specifies the content of the document.

- If the sequence of characters cannot be parsed properly as described above, then it is considered to be the whole content of the document.

Example Text Document Literals:

```
TEXT[This is a simple text document.]
TEXT[This is another simple text document.]
TEXT[myStringVar + u"\nAdding more information."]
```

User Document Literals

The user document literal is always of type Document.

```
UserDocumentLiteral:
  UserDocumentDeclarator [ComplexLiteralInputCharsopt]
  UserDocumentDeclarator [Expression]
  UserDocumentDeclarator [Expression, Expression]
UserDocumentDeclarator:
  any case for DOC
```

User document literals are used to represent a document located in the document repository and manageable through the document management pages which are part of the Cisco Unified CCX Application Administrator Web page.

- If the sequence of characters can be parsed as an expression of type String, then the resulting document is a user document where the expression specifies the name of the document to retrieve the document from the repository.
- If the sequence of characters can be parsed as two expressions where the first one must have type String and the second one must have type Contact, then the resulting document is a user document where the first argument specifies the name of the document to retrieve from the repository and the second one represents a contact which can be associated with the document to allow proper resolution of the document in the repository using the language context associated with the specified contact.
- If the sequence of characters cannot be parsed properly as described above, then it is considered to be the name of the user document to retrieve from the repository.

Example User Document Literals:

```
DOC[AA\schedule.doc]
DOC["vxml\application1.vxml", mainCall]
DOC[rootTemplateDir + "templateA.txt"]
```



Note

The special case of DOC [] represents an empty document.

Document Concatenation Operator +

If both operand expressions are of type Document or a java.io.InputStream or a java.io.Reader, then the result is a reference to a newly created Document object that is the concatenation of the two operand documents. The content of the left-hand operand precedes the content of the right-hand operand in the newly created document. The concatenation is low-level and makes no assumptions as to the content of both documents. The resulting content type is reported the same as the first document that defines a content type.

Document Qualifier Operator

Document qualifier operator results in an expression of the `Document` type.

```
QualifiedDocumentExpression:
  DocumentExpression
  QualifiedDocumentExpression @ Expression
```

See [Prompt Templates, page 3-106](#) for examples of prompt expressions.

Document Qualifier Operator @

The document qualifier @ expects a qualifying expression of the following type:

- `Language`
- `DayOfWeekLiteral`
- `Number`
- `Time`

The first qualifier represents a language qualification and is used to temporarily override the language associated with a given document. The expression must be of type `Language`. Qualifying a document more than once with a language results in only the last one to be kept as the overridden language for the document.

The second qualifier represents a day of week qualification and is used to specify the starting day of a possible range when the document is to be used in a day of week document or time of week document expression. The starting day can also be specified using a `Number` type as seen in the third option where its value must evaluate to 1 for Sunday, 2 for Monday ... or 7 for Saturday.

The last qualifier represents time qualification and is used to specify the starting time of a possible range when the document is to be used in a time of day document or time of week expression.

Time of Week, Day of Week, and Time of Day Documents

The following are the Document specifications:

- [Time of Week Document, page 3-52](#)
- [Day of Week Document, page 3-53](#)
- [Time of Day Document, page 3-53](#)

Time of Week Document

A time of week document contains multiple documents, each qualified with a particular time of the day and day of the week. When accessed, a time of week document evaluates the current time of the week and accesses a single document from its list. The document selected is based on a time range starting at the day and time specified until the day and time specified by the subsequent document in time or until the end of the week, if this is the last document. The week starts on Sunday morning.

The order of the operands is not important in determining the beginning or end of a range. The expression parser puts them back in the proper chronological order based on the specified day of week or time of day used when qualifying each one of the document operands.

For example, the document expression:

```
DOC[D1.txt] @ MON @ T[10:59 AM]
|| DOC[D2.txt] @ TUE @ T[11:58 PM]
|| DOC[] @ MON @ T[1:00 PM]
```

means:

- From Sunday morning to Monday 10:58:59 AM nothing can be accessed.
- From Monday 10:59:00 AM to Monday 12:59:59 PM, `DOC[D1.txt]` can be accessed. (`DOC[D1.txt] @ MON @ T[10:59 AM]`)
- From Monday 1:00:00 PM to Tuesday 11:57:00PM, nothing can be accessed. (`|| DOC[] @ MON @ T[1:00 PM]`) and (`|| DOC[D2.txt] @ TUE @ T[11:58 PM]`)
- From Tuesday 11:58:00PM until the end of the week, `DOC[D2.txt]` can be played back. (`|| DOC[D2.txt] @ TUE @ T[11:58 PM]`)

Day of Week Document

A day of week document contains multiple documents each qualified with a particular day of the week. When accessed, a day of week document evaluates the current day of the week and accesses a single prompt from its list. The document selected is based on a day range starting at the day specified until the day specified by the subsequent document in time or until the end of the week if this is the last document. The week starts on Sunday.

The order of the operands is not important in determining the beginning or end of a range. The expression parser puts them back in the proper chronological order based on the specified day of week used when qualifying each one of the document operands.

The 3-letter abbreviations for the day of the week variable to be pasted into the expression are: MON, TUE, WED, THU, FRI, SAT, SUN.

For example, the document expression:

```
DOC[D1.txt] @ MON || DOC[D2.txt] @ TUE || DOC[] @ THU
```

means:

- On Sunday nothing can be accessed.
- On Monday, `DOC[D1.txt]` can be accessed. (`DOC[D1.txt] @ MON`)
- On Tuesday and Wednesday, `DOC[D2.txt]` can be accessed. (`DOC[D2.txt] @ TUE`)
- The rest of the week nothing can be accessed. (`DOC[] @ THU`)

Time of Day Document

A time of day document contains multiple documents, each qualified with a particular time of the day. When accessed, a time of day document evaluates the current time of the day and accesses a single document from its list. The document selected is based on a time range starting at the time specified until the time specified by the subsequent document in time range or until the end of the day if this is the last document.

The order of the operands is not important in determining the beginning or end of a range. The expression parser puts them back in the proper chronological order based on the specified time of day used when qualifying each one of the document operands.

For example, the document expression:

```
DOC[D1.txt] @ T[10:59 AM] || DOC[D2.txt] @ T[11:58 PM] || DOC[] @ T[1:00 PM]
```

means:

- From the beginning of the day until 10:58:59 AM nothing can be accessed.
- From 10:59:00 AM until 12:59:59 PM, `DOC[D1.txt]` can be accessed.
(`DOC[D1.txt] @ T[10:59 AM] || DOC[D2.txt] @ T[11:58 PM]`)
- From 1:00:00 PM until 11:57:00 PM, nothing can be accessed.
(`|| DOC[] @ T[1:00 PM]`) and (`|| DOC[D2.txt] @ T[11:58 PM]`)
- From 11:58:00 PM until the end of the day, `DOC[D2.txt]` can be accessed.
(`|| DOC[D2.txt] @ T[11:58 PM]`)

Double

Use the Double tab to enter or modify double data in an expression. Double is a friendly data type corresponding to the fully qualified `java.lang.Double` class.



Note

In the Expression Language, `double` and `Double` can be used interchangeably as opposed to Java where `double` represents a primitive data type and `Double` represents an object.

This topic includes the following:

- [About Doubles, page 3-54](#)
- [Double Specification on the Web, page 3-54](#)
- [Example Double Code, page 3-55](#)
- [Double Variables, page 3-56](#)
- [Double Constructors, Methods, and Attributes, page 3-56](#)
- [Double tab Syntax Buttons, page 3-56](#)
- [Floating-Point Literals, page 3-61](#)

About Doubles

There are three kinds of floating-point numbers (numbers containing a decimal point): floats, doubles, and `BigDecimal`s. Each can be positive or negative. For a description of floats, see [Float, page 3-57](#). For a description of `BigDecimal`s, see [BigDecimal, page 3-13](#).

A double is a 64-bit floating-point primitive number. A double takes up 8 bytes and has 18 places of precision. Doubles are the default floating-point number type. So when you enter a number with a decimal point, the Expression Language treats it as a double.

Double Specification on the Web

For the Sun Java specification on doubles, see <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Double.html>.

Example Double Code

In the following two examples, the script variables used in the expressions are listed in the top right of each example.

Figure 3-15 A Simple Expression Using a Double and Two Script Variables

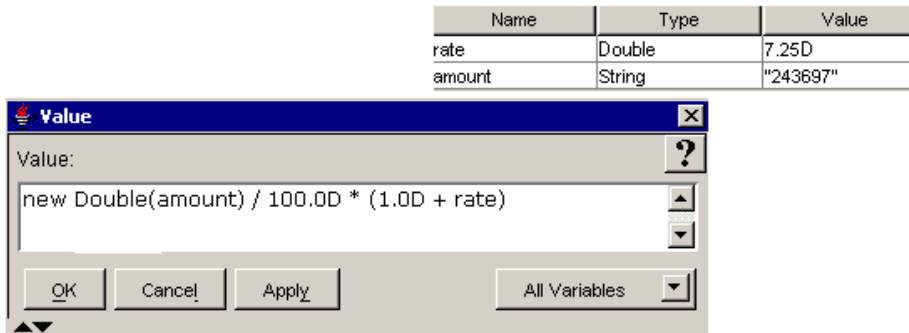
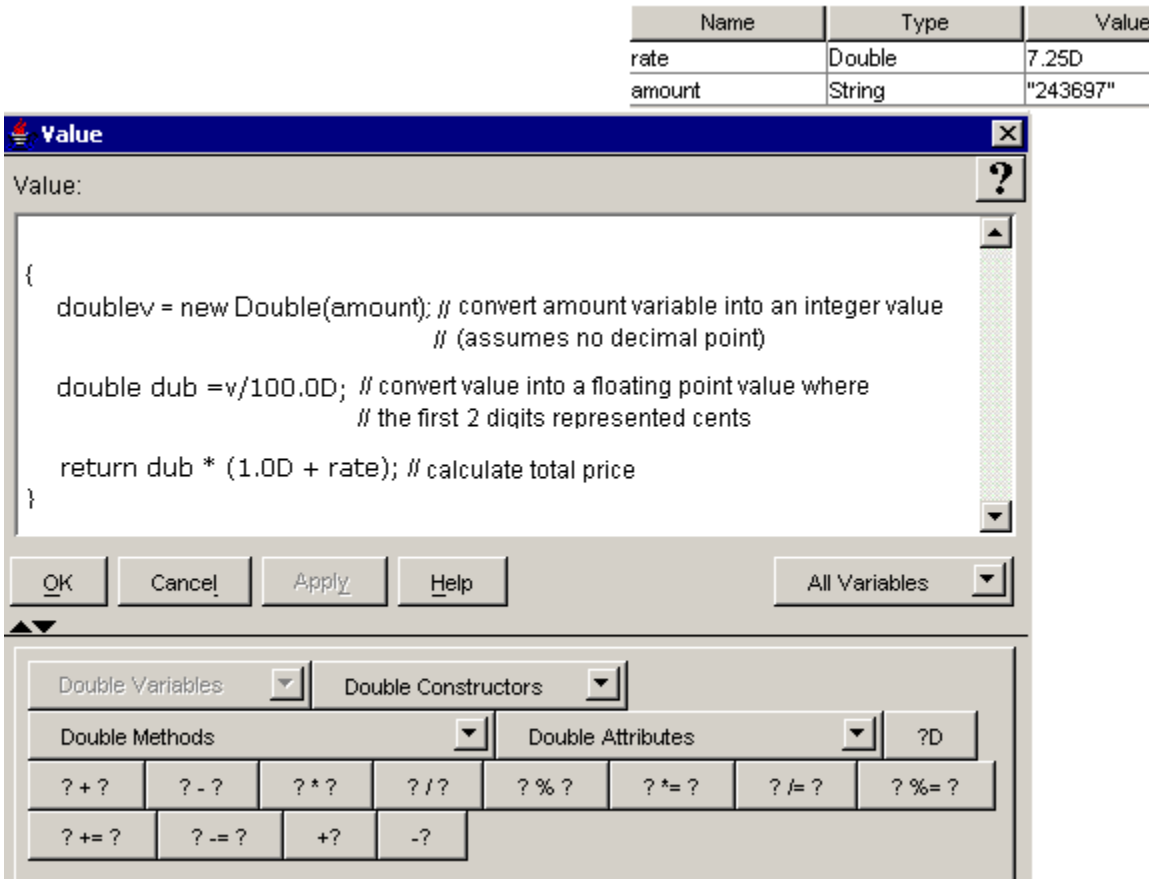


Figure 3-16 A Complex Expression Using a Double and Two Script Variables



The following sections describe the options on the Double tab:

- [Double Variables, page 3-56](#)

- [Double Constructors, Methods, and Attributes](#), page 3-56
- [Double tab Syntax Buttons](#), page 3-56

Double Variables

The Double Variables selection box lists all the double variables contained in the currently opened script. Use this selection box to paste an already defined double variable into an expression.

The double variable represents an expanded float variable. Its values include the 64-bit IEEE 754 floating-point numbers.

The default value of a double variable is positive zero, that is, 0.0d.

Double Constructors, Methods, and Attributes

Use the appropriate selection box to add a Double constructor, method, or attribute to your expression.

The available public methods and attributes include both static and non static ones.

For descriptions of the public Java Double constructors, methods, and attributes available in the selection boxes, see <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Double.html>

Double tab Syntax Buttons

The Double tab syntax buttons indicate all the ways you can add or use a double in an expression. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values.

Table 3-11 Double Syntax Descriptions

Syntax Button	Name	Type	Description
?D	literal		Enters an object of type double. For example: 3.14159D 2E-12D -100D
? + ?	addition	arithmetic	Adds two operands.
? - ?	subtraction		Subtracts the second operand from the first.
? * ?	multiplication		Multiplies two operands.
? / ?	division		Divides the first operand by the second.
? % ?	remainder		Returns the remainder of the first operand divided by the second.

Table 3-11 Double Syntax Descriptions (continued)

Syntax Button	Name	Type	Description
? *= ?	multiply and assign	assignment	Multiplies the first operand by the second and assigns the result to the first operand.
? /= ?	divide and assign	The operand on the left of the	Divides the first operand by the second and assigns the result to the first operand.
? %= ?	remainder and assign	assignment statement	Divides the first operand by the second operand and assigns the remainder to the first operand.
? += ?	add and assign	(the first operand) can be any type	Adds the first operand to the second operand and assigns the result to the first operand.
? -= ?	subtract and assign	of variable, including an array component or a public class attribute.	Subtracts the second operand from the first operand and assigns the result to the first operand.
+?	unary plus	unary	The positive value of the operand.
-?	unary minus		The negative value of the operand.

See the following for a summary descriptive list of all the operators you can use in the Java language:

<http://java.sun.com/docs/books/tutorial/java/nutsandbolts/opsummary.html>

Float

Use the Float tab to enter or modify Float data in an expression. Float is a friendly data type corresponding to the fully qualified `java.math.Float` class.



Note

In the Expression Language, `float` and `Float` can be used interchangeably as opposed to Java where `float` represents a primitive data type and `Float` represents an object.

This topic includes the following:

- [About Floats, page 3-58](#)
- [Float Specification on the Web, page 3-58](#)
- [Example Float Code, page 3-58](#)
- [Float Variables, page 3-59](#)
- [Float Constructors, Methods, and Attributes, page 3-59](#)
- [Float tab Syntax Buttons, page 3-60](#)
- [Floating-Point Literals, page 3-61](#)

About Floats

There are three types of floating-point numbers (numbers containing a decimal point): floats, doubles, and BigDecimals. Each can be positive or negative. For a description of doubles, see [Double, page 3-54](#). For descriptions of BigDecimals, see [BigDecimal, page 3-13](#).

A `java.lang.Float` is a 2-bit floating-point primitive number, takes up 4 bytes, and has 9 places of precision.

Float Specification on the Web

For the Sun Java specification on floats, see <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Float.html>.

Example Float Code

In the following two examples, the script variables used in the expressions are listed in the top right of each example.

Figure 3-17 A Simple Expression Using a Float and Two Script Variables

Name	Type	Value
amount	String	"243697"
rate	float	7.25F

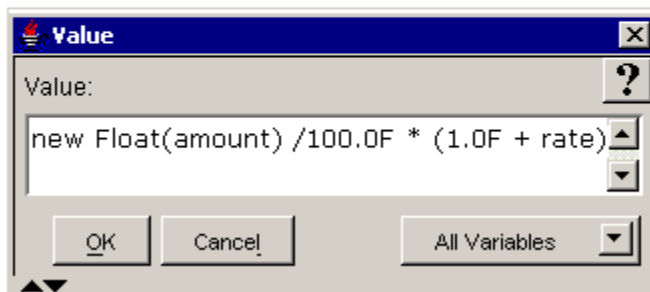
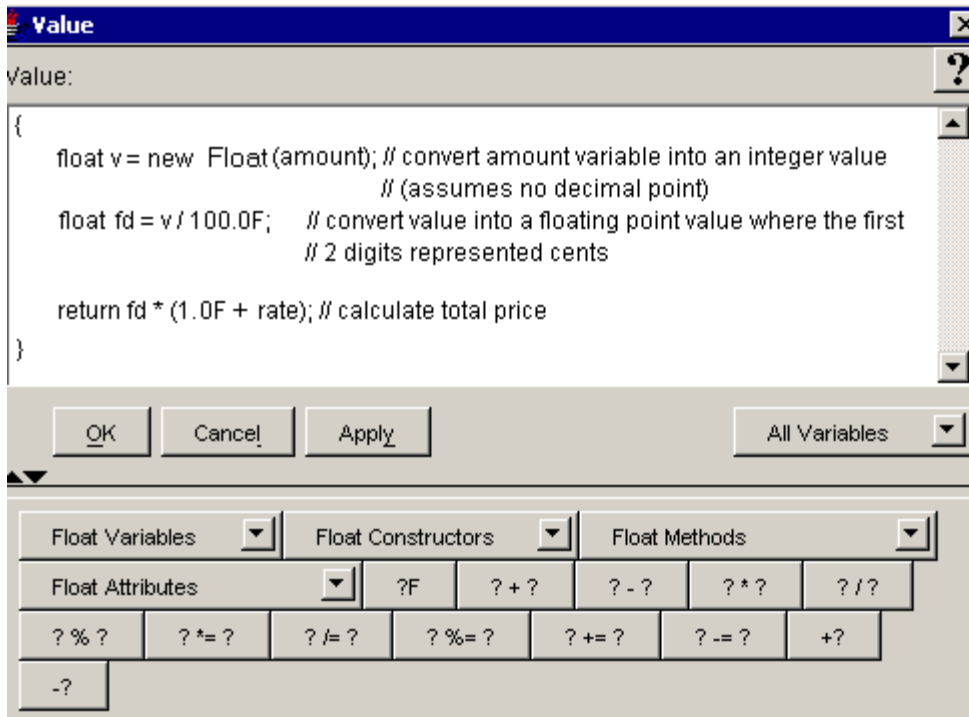


Figure 3-18 A Complex Expression Using a Float and Two Script Variables

Name	Type	Value
amount	String	"243697"
rate	float	7.25F



The following sections describe the options on the Expression Editor Float tab:

- [Float Variables](#), page 3-59
- [Float Constructors, Methods, and Attributes](#), page 3-59
- [Float tab Syntax Buttons](#), page 3-60

Float Variables

The Float Variable selection box lists all the float variables contained in the currently opened script. Use this to add a predefined float variable to your expression.

A float variable holds the value of a float. Its values include 32-bit IEEE 754 floating-point decimal numbers. A Decimal number larger than this is called a [Double](#), page 3-54 or a [BigDecimal](#), page 3-13. The default value of a float variable is positive zero, that is, 0.0f.

Float Constructors, Methods, and Attributes

Use the appropriate selection box to add a float item to your expression.

The available public methods and attributes include both static and non static ones.

For descriptions of the public Java float constructors, methods, and attributes available in the selection boxes, see <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Float.html>.

Float tab Syntax Buttons

The Float tab syntax buttons indicate all the ways you can add or use a float in an expression. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values.

Table 3-12 Float Syntax Descriptions

Syntax Button	Name	Type	Description
?F	literal		Enters an object of type float. See Floating-Point Literals, page 3-61 . For example: 3.14159F
? + ?	addition	arithmetic	Adds two operands. For example: 2E+12F
? - ?	subtraction		Subtracts the second operand from the first. For example: 2E-12F
? * ?	multiplication		Multiplies two operands.
? / ?	division		Divides the first operand by the second.
? % ?	remainder		Returns the remainder of the first operand divided by the second.
? *= ?	multiply and assign	assignment The operand on the left of the assignment statement (the first operand) can be any type of variable, including an array	Multiplies the first operand by the second and assigns the result to the first operand.
? /= ?	divide and assign		Divides the first operand by the second and assigns the result to the first operand.
? %= ?	remainder and assign		Divides the first operand by the second operand and assigns the remainder to the first operand.
? += ?	add and assign		Adds the first operand to the second operand and assigns the result to the first operand.
? -= ?	subtract and assign	component or a public class attribute.	Subtracts the second operand from the first operand and assigns the result to the first operand.
+?	unary plus	unary	The positive value of the operand.
-?	unary minus		The negative value of the operand. For example: -100F

See the following for a summary descriptive list of all the operators you can use in the Java language:
<http://java.sun.com/docs/books/tutorial/java/nutsandbolts/opsummary.html>

Floating-Point Literals

A floating-point literal has the following parts: a whole-number part, a decimal point (represented by an ASCII period character), a fractional part, an exponent, and a type suffix. The exponent, if present, is indicated by the ASCII letter e or E followed by an optionally signed integer.

At least one digit, in either the whole number or the fraction part, and either a decimal point, an exponent, or a float type suffix are required. All other parts are optional.

A floating-point literal is of type float if it is suffixed with an ASCII letter F or f, of type double if it is suffixed with an ASCII letter D or d; otherwise its type is BigDecimal and must be suffixed with the ASCII letters FB or fb.

```

FloatingPointLiteral:
  Digits . Digitsopt ExponentPartopt FloatTypeSuffixopt
  . Digits ExponentPartopt FloatTypeSuffixopt
  Digits ExponentPart FloatTypeSuffixopt
  Digits ExponentPartopt FloatTypeSuffix
ExponentPart:
  ExponentIndicator SignedInteger
ExponentIndicator: one of
  e E
SignedInteger:
  Signopt Digits
Sign: one of
  + -
FloatTypeSuffix: one of
  f F d D fb FB

```

The elements of the types float and double are those values that can be represented using the IEEE 754 32-bit single-precision and 64-bit double-precision binary floating-point formats, respectively.

The details of proper input conversion from an ASCII string representation of a floating-point number to the internal IEEE 754 binary floating-point representation are described for the methods valueOf of class Float and class Double of the package java.lang.

The largest positive finite float literal is 3.40282347e+38f. The smallest positive finite nonzero literal of type float is 1.40239846e-45f. The largest positive finite double literal is 1.79769313486231570e+308. The smallest positive finite nonzero literal of type double is 4.94065645841246544e-324.

If a nonzero floating-point literal of type float is too large or too small, the value is then represented as a floating-point literal of type double.

If a nonzero floating-point literal of type double is too large, the value is then represented as a floating-point literal of type BigDecimal.

Predefined constants representing Not-a-Number values are defined in the classes Float and Double as Float.NaN and Double.NaN.

Examples of float literals:

```
1e1f 2.f.3f 0f 3.14f 6.022137e+23f
```

Examples of double literals:

```
1e1d2.D .3d 0.0D3.14D 1e-9d 1e137
```

Examples of BigDecimal literals:

```
1e1fb2.FB.3fb0.0FB3.14FB1e-9dfb 1e133334217
```

There is no provision for expressing floating-point literals in other than decimal radix. However, method `intBitsToFloat` of class `Float` and method `longBitsToDouble` of class `Double` provide a way to express floating-point values in terms of hexadecimal or octal Integer literals.

For example, the value of:

```
Double.longBitsToDouble(0x400921FB54442D18L)
```

is equal to the value of `Math.PI`.

Each float, double, and Double literal is a reference to an instance of class `Float`, `Double` and `BigDecimal` respectively. These objects have a constant value. The `Float` and `Double` objects can be used interchangeably with their counter part Java primitive data types when calling methods that expect the primitive types or when accessing Java attributes declared using the Java primitive data type.

Grammar

Use the Grammar tab to add or modify grammars in an expression.

The Grammar friendly data type corresponds to the Java `com.cisco.grammar.Recognizable` class.

This topic includes the following:

- [About Grammars, page 3-62](#)
- [Grammar Specifications on the Web, page 3-62](#)
- [Example Grammar Code, page 3-63](#)
- [Grammar Variables, page 3-63](#)
- [Browse Grammars Dialog Box, page 3-64](#)
- [Grammar tab Syntax Buttons, page 3-64](#)
- [Grammar Literals, page 3-65](#)
- [Compound Grammar, page 3-68](#)
- [Compound Grammar Indexing, page 3-69](#)
- [Grammar Template File Types and Template Enhancements, page 3-69](#)

About Grammars

A grammar is a set of rules that define the structure or syntax of a language. A grammar can be used either to parse a sentence or to generate one. Grammars are used in Cisco Unified CCX scripts for automatic speech recognition (ASR) and touch tone (DTMF-based) interactions with a caller.

Grammar Specifications on the Web

For information on the grammar specifications Cisco Unified CCX scripts use, see the following:

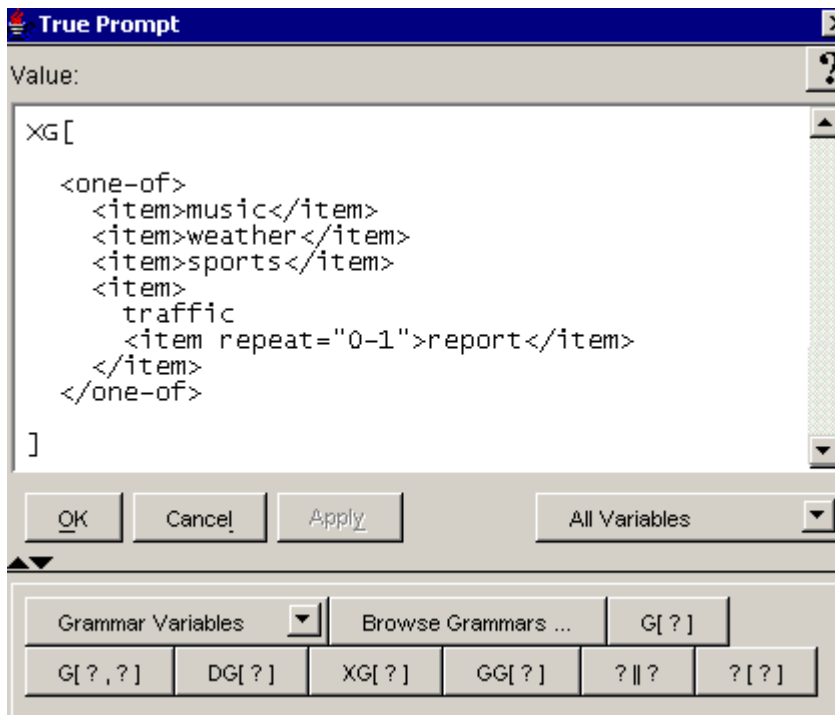
- The *Speech Recognition Grammar Specification (SRGS) Version 1.0* at <http://www.w3.org/TR/speech-grammar/> defines a standard syntax for representing grammars for use in speech recognition so that developers can specify the words and patterns of words to be listened for by a speech recognizer.

- The *Cisco Regular Expression (Regex) Grammar Specification* at http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/rel_docs/vxmlprg/refgde1.htm#1050172 defines the Cisco dual tone multiple frequency (DTMF) grammar that you can use with VoiceXML.

Example Grammar Code

The following example grammar recognizes any of the following spoken phrases:

- "music"
- "weather"
- "sports"
- "traffic"
- "traffic reports"



The following sections describe the options on the Grammar tab:

- [Grammar Variables](#), page 3-63
- [Browse Grammars Dialog Box](#), page 3-64
- [Grammar tab Syntax Buttons](#), page 3-64

Grammar Variables

The Grammar Variable selection box contains all the grammar variables contained in the currently opened script. Use this selection box to paste a grammar variable into an expression.

The grammar variable represents different options that can be selected by a caller using a Media input step (such as the Menu step). A grammar variable can represent grammars uploaded to the grammar repository or created using some of the existing steps.

The default value of a Grammar variable is the empty grammar, that is, G[].

Browse Grammars Dialog Box

Use the Browse Grammars selection box to browse grammars created on disk or in the Grammar repository. You can then add a selected grammar from this selection box to your script.


Grammar tab Syntax Buttons

The Grammar tab syntax buttons indicate all the ways you can add a Grammar object to an expression. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values.

Table 3-13 Grammar Syntax Button Descriptions

Syntax Button	Description
G[?]	Grammar object. See Grammar Literals, page 3-65 and User Grammar Literals, page 3-66 . Examples: G[DigitGrammar] G[MyGrammar.gsl]
G[?,?]	Grammar and argument(s). See Grammar Literals, page 3-65 and User Grammar Literals, page 3-66 . The first argument must be of type string and specifies the name of the grammar to retrieve from the user or system repository. The second argument (or argument list) corresponds to the expected parameterized arguments of a complex expression block defined in a grammar template file. Examples: G[SRGSGrammar,SRGS] G["myGrammar.tgl", "dtmf-2", tagValue]
DG[?]	DTMF Grammar. See Digit Grammar Literals, page 3-66 Example: DG[<grammar xml:lang="en-US" root = "pin" mode="dtmf" xmlns="http://www.w3.org/2001/06/grammar">]
XG[?]	XML Grammar (SRGS). See SRGS Grammar Literals, page 3-68 . Example: XG[<?xml version="1.0"?><grammar version="1.0" xmlns=http://www.w3.org/2001/06/grammar></grammar>]

Table 3-13 Grammar Syntax Button Descriptions (continued)

Syntax Button	Description
GG[?]	<p>GSL Grammar. See GSL Grammar Literals, page 3-67</p> <p>Example: GG[[[yes digit-1] {tag yes} [no digit-2] u{tag no}]]</p>  <p>Note Depreciated. GSL grammars are no longer supported.</p>
? ?	<p>Compound Grammar. The compound grammar operator combines multiple grammars together. See Compound Grammar, page 3-68.</p> <p>Example: G[grammar1.digit] G[grammar2.grxml]</p>
?[?]	<p>Indexing a compound grammar. The index is 0 based. See Compound Grammar Indexing, page 3-69.</p> <p>Example: Grammar[5]</p>

Grammar Literals

The grammar literal is always of type Grammar.

```
GrammarLiteral:
  DigitGrammar
  GSLGrammar (deprecated)
  SRGSGrammar
  UserGrammar
```

Each grammar literal is a reference to an instance of a class that implements the interface `com.cisco.grammar.Recognizable`.

Example Grammar Literals:

```
G[], SG[]—An empty grammar. (No value gets recognized.)
G[grammar.grxml]—A user-defined grammar located in the Grammars repository.
```

The topic describes the following grammar literals:

- [User Grammar Literals, page 3-66](#)
- [Digit Grammar Literals, page 3-66](#)
- [GSL Grammar Literals, page 3-67](#)
- [SRGS Grammar Literals, page 3-68](#)

Each grammar literal is a reference to an instance of a class that implements the interface `com.cisco.grammar.Recognizable`.



Note

The `GSLGrammar` is deprecated. That means that, although it can be used, it is restricted. This type of grammar is automatically converted to the `SRGSGrammar` (Speech Recognition Grammar) format. You must use the `SRGSGrammar` rather than the `GSLGrammar`.

User Grammar Literals

The user grammar literal is always of type `Grammar`.

```
UserGrammar:
  UserGrammarDeclarator [ComplexLiteralInputCharsopt]
  UserGrammarDeclarator [Expression]
  UserGrammarDeclarator [Expression, ArgumentList]
```

```
UserGrammarDeclarator: one of
  g G
```

User grammar literals are used to represent a grammar located in the grammar repository and manageable through the grammar management pages which are part of the Cisco Unified CCX Application Administrator web page. The `ComplexLiteralInputChars` can include the `[` character as long as it has a balanced number of `]` characters: one for every `[` character found:

- If the sequence of characters can be parsed as an Expression of type `String`, then the resulting grammar is a user grammar where the expression specifies the name of the grammar to retrieve the grammar from the repository.
- If the sequence of characters can be parsed as an Expression and an `ArgumentList` where the first one must have type `String`, then the resulting grammar is a user grammar where the first argument specifies the name of the grammar to retrieve from the repository and the argument list must correspond to the expected parameterized arguments of a complex expression block defined in a grammar template file.

The arguments are ignored if the referenced grammar is not a grammar template. If it is one, then each specified argument is evaluated and assigned as the value of a defined argument to the expression block. If the types does not match, then a runtime exception is thrown back. No errors are generated if more arguments are supplied then expected; they are ignored. No errors are generated if fewer arguments are supplied then expected unless the given argument is accessed by the complex expression block and it is not defined with a default value.

- If the sequence of characters cannot be parsed properly as described above, then it is considered to be the name of the user grammar to retrieve from the repository.

Example User Grammar Literals:

```
G[myGrammar.gsl]
G[global]
G["myGrammar.tgl", "dtmf-2", tagValue]
```

The extension of the grammar file can be omitted in which case the search attempts all supported extensions (`<.grxml>`, `<.gsl>`, `<.digit>`, `<.tgl>`) with the search starting based on the type of media supported by the call (that is, if MRCP ASR is supported, the search starts with `<.grxml>`, if MRCP, then `.grxml`, `.tgl`, `.gsl`, `.digit`; otherwise (for CMT), `.digit`, `.tgl`, `.grxml`, `.gsl`, otherwise, it starts with `<.digit>`) and then continues with the other extensions based on the order specified above.



Note

The special case of `G[]` represents an empty grammar.

Digit Grammar Literals

The digit grammar literal is always of type `Grammar`.

```
DigitGrammar:
  DigitGrammarDeclarator [ComplexLiteralInputChars]
  DigitGrammarDeclarator [Expression]
DigitGrammarDeclarator:
  any case of DG
```

Digit grammar literals are used to represent a grammar that contains solely digits to be recognized. Its format is similar to the Digit File Grammar format where each entry is separate with a | character. Each entry is defined as key=value or key where the keys are defined as dtmf-x. Where x is from the set 0123456789*#ABCD or be one of star or pound and values would be the corresponding tag to be returned when key is pressed or recognized. An optional entry defined as word=true can be used to identify that the word representation of each DTMF digit must be automatically included during a recognition:

- If the sequence of characters can be parsed as an Expression of type String, java.io.File or java.util.Properties, then the resulting grammar is a digit grammar where the expression specifies the inline content of the grammar, the filename where to retrieve the grammar or a properties object where the keys are expected to be defined as dtmf-x and the values would be the corresponding tag to be returned when the key is pressed or recognized.
- If the sequence of characters cannot be parsed properly as described above, then it is considered to be the inline content of the digit grammar.

Example Digit Grammar Literals:

```
DG[digit-1=yes|digit-2=no]
DG[word=true|digit-star=cancel]
DG["digit-1=" + tag1 + "|digit-2=" + tag2]
DG[new java.io.File(u"C:\myGrammar digit")]
```

GSL Grammar Literals



Note

The GSLGrammar is deprecated. That means that, although it can be used, it is restricted. This type of grammar is automatically converted to the SRGSGrammar (Speech Recognition Grammar) format. You must use the SRGSGrammar rather than the GSLGrammar.

The Nuance GSL (Grammar Specification Language) grammar literal is always of type Grammar.

```
GSLGrammar:
    GSLGrammarDeclarator ComplexLiteralInputChars]
    GSLGrammarDeclarator [Expression]

GSLGrammarDeclarator:
    any case of GG
```

GSL grammar literals are used to represent a grammar that supports, in a limited way, the Nuance Grammar Specification Language format. Only one expression is supported; no rule set and it must have a slot named <tag> if used as a main grammar in a recognition.

The ComplexLiteralInputChars can include the [character as long as it has a balanced number of] characters: one for every [character found:

- If the sequence of characters can be parsed as an Expression of type String or java.io.File, then the resulting grammar is a GSL grammar where the expression specifies the inline content of the grammar or the filename where to retrieve the grammar.
- If the sequence of characters cannot be parsed properly as described above, then it is considered to be the inline content of the GSL grammar.

**Note**

Unified CCX does not provide support for Nuance GSL grammars. However to remain backward compatible, the Expression Language still parses and validates these expressions. It is at run time that it is highly likely that an exception would be thrown if the system is unable to convert the GSL grammar into a digit or an SRGS grammar. For more details, see the Cisco MRCP (Media Resource Control Protocol) documentation in the *Cisco Unified Contact Center Express Administration Guide*.

Example GSL Grammar Literals:

```
GG[hello world]
GG[(i would like [one two three] hamburgers)]
GG[[yes dtmf-1]]
GG[[no dtmf-2]]
GG[[movies sports weather]]
GG[new java.io.File(u"C:\\myGrammar.gsl")]
```

SRGS Grammar Literals

The SRGS grammar literal is always of type Grammar.

SRGSGrammar:

```
SRGSGrammarDeclarator [ComplexLiteralInputChars]
SRGSGrammarDeclarator [Expression]
```

SRGSGrammarDeclarator:
any case of XG

SRGS grammar literals are used to represent a grammar that supports the Speech Recognition Grammar format. The grammar must have an ecma variable named tag if used as a main grammar in a recognition. The ComplexLiteralInputChars can include the [character as long as it has a balanced number of] characters: one for every [character found:

- If the sequence of characters can be parsed as an Expression of type String, java.io.File, java.net.URL, or org.w3c.dom.Document then the resulting grammar is a SRGS grammar where the expression specifies the inline content of the grammar or the filename where to retrieve the grammar.
- If the sequence of characters cannot be parsed properly as described above, then it is considered to be the inline content of the SRGS grammar.

Compound Grammar

A compound grammar is a grammar that combines multiple grammars together. All grammars combined together are activated at the same time when a recognition or an acquisition is performed. Priority is always given to the grammars that comes to the right of another. So if an additional grammar is combined with a first one and it defines the same choices, it is the one taken precedence in the recognition. Compound grammars may have some special treatment based on the media choosen. These would be documented in these respective documents. For CMT media termination, all DTMFs are combined together to form a single grammar to be used when acquiring DTMF digits from a caller.

For example, the grammar expression:

```
G[G1] || G[G2] || GG[Hello|dtmf-2]
```

represents a compound grammar that would activate the grammars G[G1], G[G2] and GG[Hello|dtmf-2] together with priority to GG[Hello|dtmf-2] over G[G1] and G[G2], and priority to G[G2] over G[G1].

Compound Grammar Indexing

It is possible to index a compound grammar like an array in order to reference a single grammar contained in the compound grammar. This is done using the [] operator and is similar to indexing an array. Whether the compound grammar is represented with the || operator or is a grammar from the grammar repository that would result into a compound grammar.

If the supplied index is out of bounds, a parse time or an evaluation time `ExpressionArrayIndexOutOfBoundsException` may be thrown as a result. If the grammar being indexed does not represent a compound grammar then an `ExpressionClassCastException` is thrown.

Compound Grammar Indexing Examples, Each of which Results in a Grammar Expression

```
G[grammar.tgl][1]
(G[grammar1.digit] || G[grammar2.grxml])[0]
((DG[dtmf-1|word=true] || G[grammar.tgl] || GG[hello|dtmf-3])[0][1]
```

Grammar Template File Types and Template Enhancements

There are some grammar definition and syntax changes. The Speech Recognition Grammar Specification (SRGS), the Speech Synthesis Markup Language Specification (SSML) MRCP grammars, and the Cisco DTMF RegEx specification replace the Nuance Grammar Specification Language (GSL).

There is also added support for a new type of grammar file to the user and system grammars already available. This new file has the filename extension.tgl and can be referenced in a script just like other grammar files.

In addition, not related to the expression, there is added support for one new grammar file extension:.grxml. Files ending with this extension are expected to be text files written as SRGS (Speech Recognition Grammar Specification) grammars. Since Cisco CRS 3.0, when referencing a user grammar, the extension of the file was optional and a search among valid extensions was performed to locate a file in the grammar repository. The search order is:.grxml,.gsl,.digit and .tgl.

When a user grammar with the .tgl extension is located, it is loaded as a text file and parsed, and the result is a grammar object. The expression specified in the text file does not have access to script variables. However, if defined using a complex block expression, the block can be parameterized like a method declaration, allowing for scripts to customize the evaluation of the expression. This is similar in concept to the prompt template file described in [Prompt Templates, page 3-106](#).

Integer

Use the Integer tab to enter or modify Integer data in an expression. The int friendly data type corresponds to the `java.lang.Integer` class.



Note

In the Expression Language, int and Integer can be used interchangeably as opposed to Java where int represents a primitive data type and Integer represents an object.

This topic includes the following:

- [About the Integer Class, page 3-70](#)
- [Integer Specification on the Web, page 3-70](#)
- [Example Integer Code, page 3-70](#)

- [Integer Variables](#), page 3-71
- [Integer Constructors, Methods, and Attributes](#), page 3-71
- [Integer Operations](#), page 3-72
- [Integer tab Syntax Buttons](#), page 3-72
- [Integer Literals](#), page 3-75

About the Integer Class

The Integer defines integral value between -2^{31} to $2^{31}-1$. The BigInteger class contains those integers that are larger.

An Integer literal can be expressed in decimal (base 10) or hexadecimal (base 16). The following are examples of Integer literals:

```
0 2i -23 03720xDadaCafe1996I0x00FF00FF
```

For more information on Integer literals, see [Integer Literals](#), page 3-75.

Integer Specification on the Web

For the Sun Java specification on integers on the Web, see <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Integer.html>

Example Integer Code

In the following two examples, the script variables used in the expressions are listed in the top right of each example.

Figure 3-19 Example Simple Expression Using an Integer and Script Variables

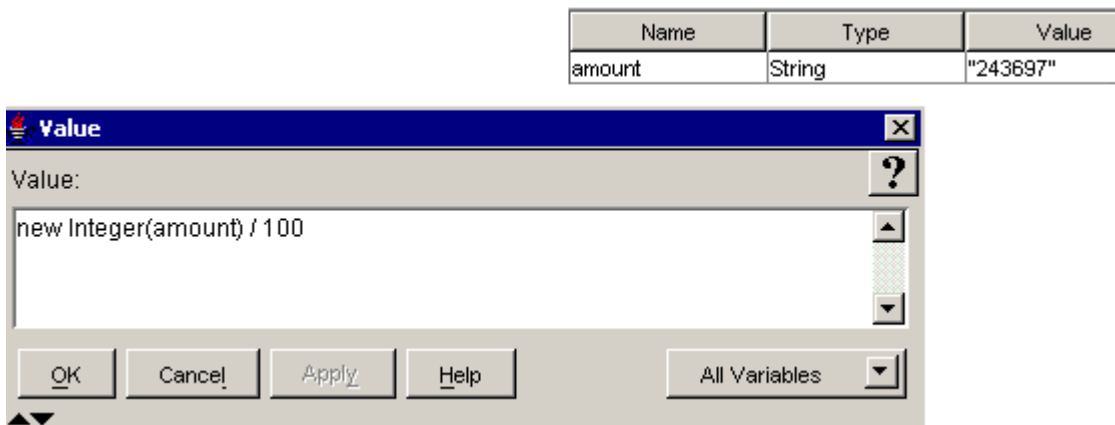
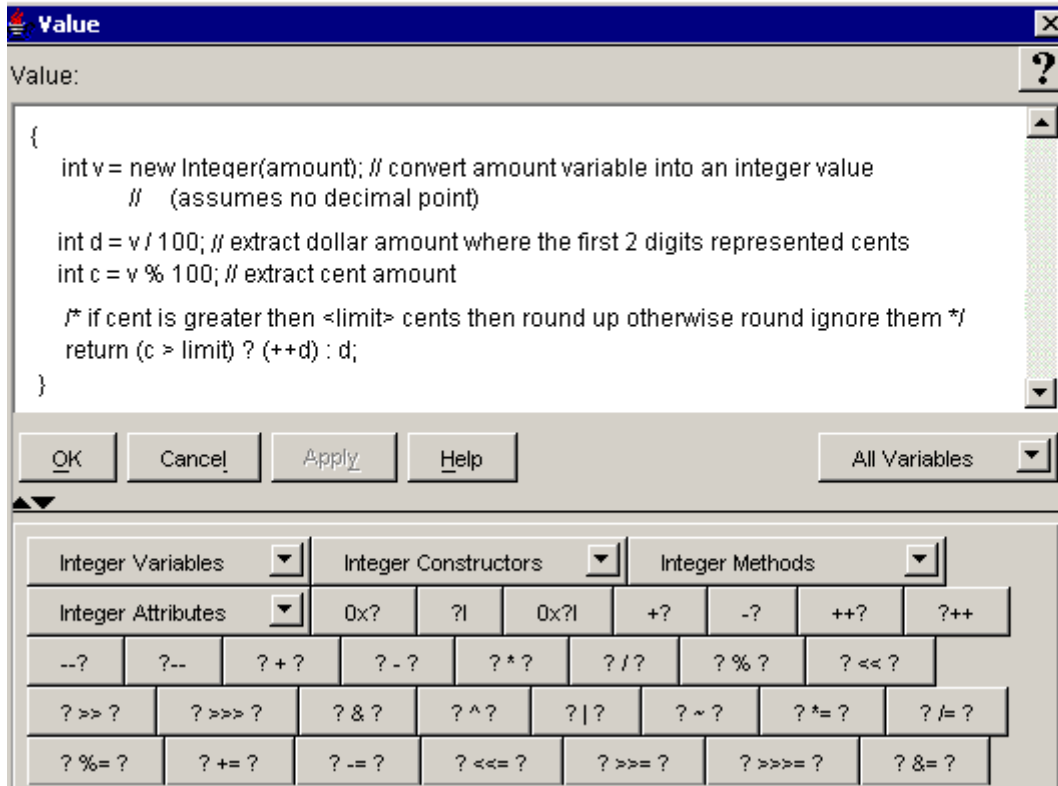


Figure 3-20 Example Complex Expression Using Integers and Two Script Variables

Name	Type	Value
amount	String	"243697"
limit	int	80



The following sections describe the options on the Integer tab:

- [Integer Variables](#), page 3-71
- [Integer Constructors, Methods, and Attributes](#), page 3-71
- [Integer Operations](#), page 3-72
- [Integer tab Syntax Buttons](#), page 3-72

Integer Variables

The Integer Variable selection box lists all the Integer variables contained in the currently opened script. Use this selection box to paste an already defined Integer variable into an expression.

An Integer variable can hold the value of a whole number from -2147483648 to 2147483647, inclusive. The default value of an Integer variable is zero, that is, 0.

Integer Constructors, Methods, and Attributes

Use the appropriate selection box to add public Integer Java code to your expression.

The available public methods and attributes include both static and non static ones.

For descriptions of all the public Integer constructors, methods, and attributes available in the selection boxes, see the Java specification at <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Integer.html>.

Integer Operations

The Expression Language provides a number of operators that act on Integers:

- The comparison operators, which result in a value of type Boolean:
 - Numerical Comparison Operators (<, <=, >, and >=)
 - Numeric Equality Operators (== and !=)
- The numerical operators, which result in a value of type int or long or BigInteger:
 - Unary Plus Operator (+) and Unary Minus Operator (-)
 - Multiplicative Operators (*, /, and %)
 - Additive Operators (+ and -) for Numeric Types
 - Prefix Increment Operator (++) and Postfix Increment Operator (++)
 - Prefix Decrement Operator (--) and Postfix Decrement Operator (-)
 - Shift Operators (<<, >>, and >>>)
 - Bitwise Complement Operator (~)
 - Integer Bitwise Operators (&, ^, and |)
- Conditional Operator (? :)
- Field Access Using a Primary
- Method Invocation Expressions
- The cast operator, which can convert from an integral value or a string value to a value of any specified numeric type

The semantics of arithmetic operations exactly mimic those of Java's Integer arithmetic operators, as defined in The Java Language Specification. See the following for a summary descriptive list of all the operators you can use in the Java language:

<http://java.sun.com/docs/books/tutorial/java/nutsandbolts/opsummary.html>

Integer tab Syntax Buttons

The Integer tab syntax buttons indicate all the ways you can add or modify an Integer in an expression in a Cisco Unified CCX script. Clicking on one of the buttons adds the indicated syntax to your expression. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values.

Table 3-14 Integer Syntax Button Descriptions

Syntax Button	Name	Type	Description
?I	literal	decimal	An Integer literal in decimal format. See Integer Literals, page 3-75 . For example: 3 or 5I
OX? OX?I	literal	hexadecimal	An Integer literal in hexadecimal format. See Integer Literals, page 3-75 . For example: 0x49 or OXFFI
+?	unary plus	unary	The positive value of the operand.
-?	unary minus		The negative value of the operand.
++? ¹	prefix increment	increment	Increments the value of the operand by one before the operand is changed in an expression.
?++ ¹	postfix increment		Increments the value of the operand by one after the operand is changed in an expression.
--? ¹	prefix decrement	decrement	Decrements the value of the operand by one before the operand is changed in an expression.
?-- ¹	postfix decrement		Decrements the value of the operand by one after the operand is changed in an expression.
? + ?	addition	arithmetic	Adds two operands.
? - ?	subtraction		Subtracts the second operand from the first.
? * ?	multiplication		Multiplies two operands.
? / ?	division		Divides the first operand by the second.
? % ?	remainder		Returns the remainder of the first operand divided by the second.
? << ?	shift left	bitwise shift (for operations on individual bits in Integers only)	Shifts bits of operand 1 left by the distance of operand 2; fills with zero bits on the right-hand side.
? >> ?	shift right		Shifts bits of operand 1 right by the distance of operand 2; fills with the highest (signed) bit on the left-hand side.
? >>> ?	zero fill right shift		Shifts bits of operand 1 right by a distance of operand 2; fills with zero bits on the left-hand side.

Table 3-14 Integer Syntax Button Descriptions (continued)

Syntax Button	Name	Type	Description
? & ?	bitwise AND	bitwise logical (for operations on individual bits in Integers only)	Compares both operands. If both operand bits are 1, the AND function sets the resulting bit to 1; otherwise, the resulting bit is 0.
? ^ ?	bitwise exclusive OR (XOR)		Compares both operands. If both operand bits are different, the resulting bit is 1; otherwise the resulting bit is 0.
? ?	bitwise inclusive OR		Compares both operands. If either of the two operand bits is 1, the resulting bit is 1. Otherwise, the resulting bit is 0.
~ ?	Bitwise complement		Inverts the value of each operand bit: If the operand bit is 1, the resulting bit is 0; if the operand bit is 0, the resulting bit is 1.
? *= ?	multiply and assign	assignment	Multiplies the first operand by the second and assigns the result to the first operand.
? /= ?	divide and assign	The operand on the left of the	Divides the first operand by the second and assigns the result to the first operand.
? %= ?	remainder and assign	assignment statement (the first operand)	Divides the first operand by the second operand and assigns the remainder to the first operand.
? += ?	add and assign	can be any type of	Adds the first operand to the second operand and assigns the result to the first operand.
? -= ?	subtract and assign	variable, including an array	Subtracts the second operand from the first operand and assigns the result to the first operand.
? <<= ?	left shift and assign	component or a public class attribute.	Shifts bits of operand 1 left by the distance of operand 2; fills with zero bits on the right-hand side and assigns the resulting bit to operand 1.
? >>= ?	right shift and assign	Assignment (continued)	Shifts bits of operand 1 left by the distance of operand 2; fills with zero bits on the right-hand side and assigns the resulting bit to operand 1.
? >>>= ?	zero fill, right shift, and assign		Shifts bits of operand 1 right by a distance of operand 2; fills with zero bits on the left-hand side and assigns the resulting bit to operand 1.
? &= ?	AND and assign		First, compares both operands. If both operand bits are 1, the AND function sets the resulting bit to 1; otherwise, the resulting bit is set to 0. Then, assigns the resulting bit to operand 1.
? ^= ?	XOR and assign		First, compares both operands. If both operand bits are different, the resulting bit is 1; otherwise the resulting bit is 0. Then, assigns the resulting bit to operand 1.
? = ?	OR and assign		First, compares both operands. If either of the two operand bits is 1, the resulting bit is 1. Otherwise, the resulting bit is 0. Then, assigns the resulting bit to operand 1.

1. The operand for the prefix and postfix increment operators must be a variable, an array component, or a public class attribute.

Integer Literals

An Integer literal can be expressed in decimal (base 10) or hexadecimal (base 16):

```
IntegerLiteral:
    DecimalIntegerLiteral
    HexIntegerLiteral

DecimalIntegerLiteral:
    DecimalNumeral IntegerTypeSuffixopt

HexIntegerLiteral:
    HexNumeral IntegerTypeSuffixopt

IntegerTypeSuffix: one of
    i I l L ib IB
```

An Integer literal is of type BigInteger if it is suffixed with the ASCII letters IB or ib, long if it is suffixed with an ASCII letter L or l (ell); otherwise it is of type int. The suffix L is preferred, because the letter l (ell) is often hard to distinguish from the digit 1 (one).

A decimal numeral is either the single ASCII character 0, representing the Integer zero, or consists of an ASCII digit from 1 to 9, optionally followed by one or more ASCII digits from 0 to 9, representing a positive Integer:

```
DecimalNumeral:
    0
    NonZeroDigit Digitsopt

Digits:
    Digit
    Digits Digit

Digit:
    0
    NonZeroDigit

NonZeroDigit: one of
    1 2 3 4 5 6 7 8 9
```

A hexadecimal numeral consists of the leading ASCII characters 0x or 0X followed by one or more ASCII hexadecimal digits and can represent a positive, zero, or negative Integer. Hexadecimal digits with values 10 through 15 are represented by the ASCII letters a through f or A through F, respectively; each letter used as a hexadecimal digit may be uppercase or lowercase.

```
HexNumeral:
    0 x HexDigits
    0 X HexDigits

HexDigits:
    HexDigit
    HexDigit HexDigits

HexDigit: one of
    0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
```

The largest decimal literal of type int is 2147483648 (2³¹). All decimal literals from 0 to 2147483647 may appear anywhere an int literal may appear, but the literal 2147483648 may appear only as the operand of the unary negation operator -.

Java

The Java tab in the Expression Editor provides you a visualized way to write an expression block. This tool also enables you to access the Java constructors, methods, and attributes for any Java class of data.

**Note**

In the example code illustrating how to use each Expression Editor tab, there are simple expression examples and complex expression examples. The complex expression examples require a Java license. The Unified CCX Enhanced, and Unified CCX Premium licences include the Java license. The Cisco Unified CCX Standard license does not include the Java license.

This topic includes the following:

- [Java Specification on the Web, page 3-77](#)
- [Example Java tab Code, page 3-77](#)
- [Java tab Constructors, Methods, and Attributes, page 3-78](#)
- [How to Access a Java Constructor, Method, or Attribute for Any Class, page 3-79](#)
- [Java tab Syntax Button Descriptions, page 3-80](#)

Java Specification on the Web

For the Sun Java specification on the Web, see:

http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html

For the Java specification on Blocks and Statements, see:

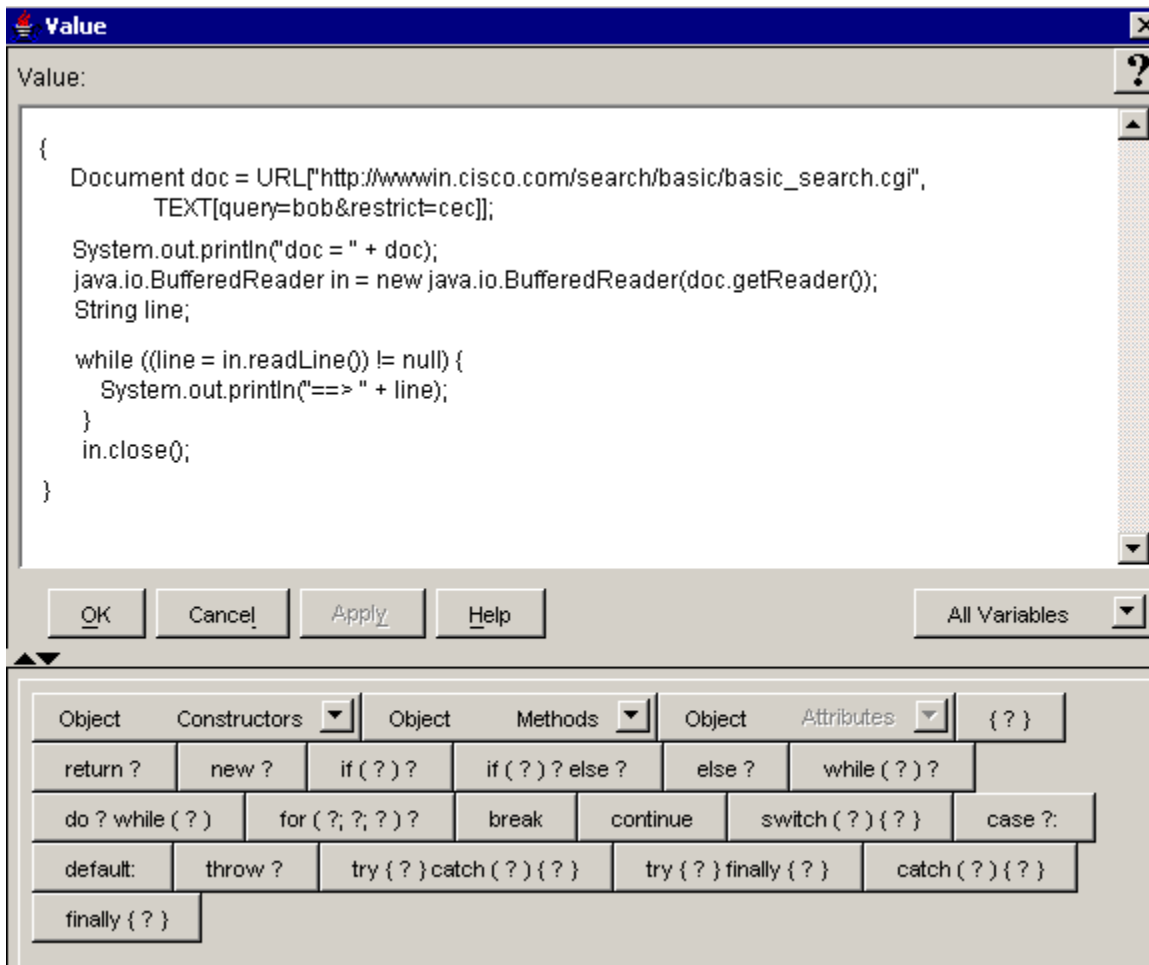
http://java.sun.com/docs/books/jls/second_edition/html/statements.doc.html#101241

For summary descriptions of the Sun Java control-flow statements, see:

<http://java.sun.com/docs/books/tutorial/java/nutsandbolts/flowssummary.html>

Example Java tab Code

The following is an example complex expression using the Java tab



The following sections describe the options on the Java tab:

- [Java tab Constructors, Methods, and Attributes, page 3-78](#)
- [Java tab Syntax Button Descriptions, page 3-80](#)

Java tab Constructors, Methods, and Attributes

Use the appropriate selection box to add a constructor, method, or attribute to your expression for any Java type entered in the expression text field.

The available public methods and attributes include both static and non static ones.



Note

The Java tab contains a selection list of the constructors, methods, attributes, and syntax buttons of the selected Java object within the open script. Therefore, the contents of this tab will vary.

See also [How to Access a Java Constructor, Method, or Attribute for Any Class, page 3-79](#).

How to Access a Java Constructor, Method, or Attribute for Any Class

The Java tab allows you to enter any fully qualified Java class name of your choosing in order to have its set of constructors, methods or attributes listed in the selection boxes. Included in this list of class names are every class from the Sun JDK, all the Cisco classes, and any custom classes you might have uploaded through the Cisco Unified CCX Application Administration web pages.

This enables an easy lookup of what is available so you can paste it into an expression directly. The selection box drop-down arrow is disabled if the class entered is invalid or does not have any constructors, methods or attributes.



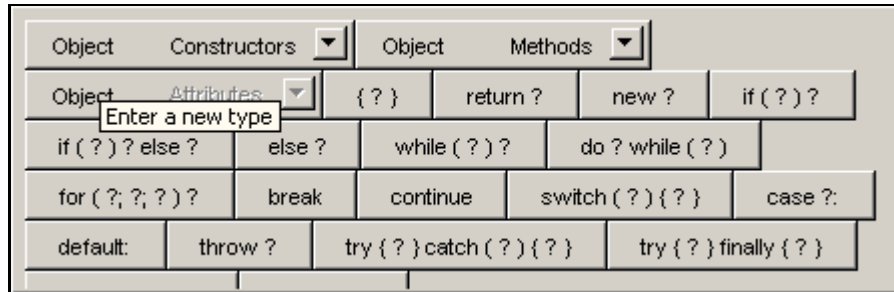
Note

This Expression Language Java functionality of allowing you to select the Java class name of your own choosing replaces and simplifies what you could do with the customizer of the deprecated Java steps.

To access the Java Constructors, Methods, and Attributes of Your Own Data Choice:

- Step 1** In the Cisco Unified CCX Expression Editor Java tab, place the cursor over the word “Object” in the Constructor, Method, or Attribute selection box, depending on which item you want.

A yellow pop-up window appears saying: Enter a new type.



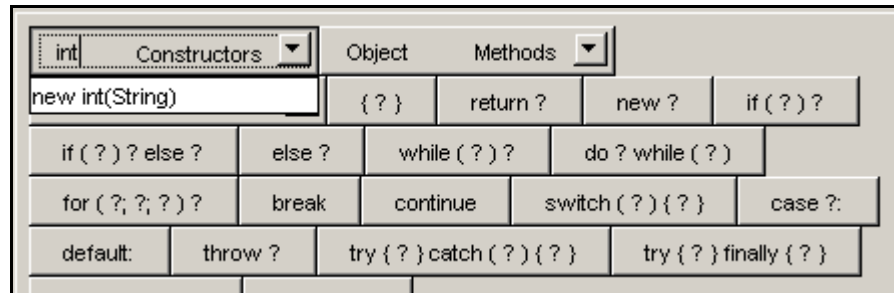
- Step 2** Double click the word **Object**.

The word Object is highlighted in blue and the arrow cursor changes to an input cursor (“I”) indicating this is a text field that you can change.

- Step 3** Enter the Java class name that you want.

The drop down list changes to the entered type.

- Step 4** Click the selection arrow to display the list of items available for that data class. In the following example, the user entered “int” to replace object in the constructor selection box.



- Step 5** Select an item in the list to enter it into the Expression Editor Value text field.

How to Make Custom Java Classes Available to the Cisco Unified CCX Editor

To make custom Java classes available to the Cisco Unified CCX Editor:

- Step 1** Using the Cisco Unified CCX Application Administration web pages, upload the jar files containing the custom classes to the document repository.
- Step 2** Using the Cisco Unified CCX Application Administration web pages, configure a custom class path to specify the jar files.
- Step 3** Restart the Cisco Unified CCX Editor to load the custom jar files and make them available.

See the *Cisco Unified Contact Center Administration Guide* for further instructions.

Java tab Syntax Button Descriptions

Use the Java tab syntax buttons to add statements to an expression. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values.

Table 3-15 Java Syntax Button Descriptions

Syntax Button	Enters into the Expression Editor palette, the syntax for a...
{ ? }	Block statement. A block statement or block is a sequence of statements and local variable declaration statements within braces. See “Blocks and Statements” at http://java.sun.com/docs/books/jls/third_edition/html/statements.html
return ?	Return statement. Returns control to the evaluator of a complex expression block. See “Branching Statements” at http://java.sun.com/docs/books/tutorial/java/nutsandbolts/branch.html
new ?	New operator. The operand represents the friendly data type or the fully qualified Java class name of the object to create. Creates and allocates space for a new object.
if (?) ?	If statement. Conducts a conditional test and executes a block of statements if the test evaluates to true. See “The if/else Statements” at http://java.sun.com/docs/books/tutorial/java/nutsandbolts/if.html .

Table 3-15 Java Syntax Button Descriptions (continued)

Syntax Button	Enters into the Expression Editor palette, the syntax for a...
if (?) ? else ?	<p>If-else statement.</p> <p>Conducts a conditional test and executes a block of statements if the test evaluates to true. Allows conditional execution of a statement or a conditional choice of two statements, executing one or the other but not both.</p> <p>See “The if/else Statements” at http://java.sun.com/docs/books/tutorial/java/nutsandbolts/if.html.</p>
else ?	<p>Else statement.</p> <p>Executes a block of statements in the case that the test condition with the "if" keyword evaluates to false.</p> <p>See “The if/else Statements” at http://java.sun.com/docs/books/tutorial/java/nutsandbolts/if.html.</p>
while (?) ?	<p>While statement.</p> <p>Executes a Statement repeatedly until the value of the Expression is false. The Expression must have type Boolean, or a parse-time error occurs.</p> <p>If the value of the Expression is false the first time it is evaluated, then the Statement is not executed.</p> <p>See “The while and do-while Statements” at http://java.sun.com/docs/books/tutorial/java/nutsandbolts/while.html.</p>
do ? while (?)	<p>Do-While statement.</p> <p>Executes a Statement and an Expression repeatedly until the value of the Expression is false. The Expression must have type Boolean, or a parse-time error occurs.</p> <p>Executing a do statement always executes the contained Statement at least once.</p> <p>See “The while and do-while Statements” at http://java.sun.com/docs/books/tutorial/java/nutsandbolts/while.html.</p>
for (? . ? . ?) ?	<p>For statement.</p> <p>Executes some initialization code, then executes an Expression, a Statement, and some update code repeatedly until the value of the Expression is false.</p> <p>See The for Statement at http://java.sun.com/docs/books/tutorial/java/nutsandbolts/for.html.</p>
break	<p>Break statement.</p> <p>Transfers control out of an enclosing statement.</p> <p>See “Branching Statements” at http://java.sun.com/docs/books/tutorial/java/nutsandbolts/branch.html.</p>
continue	<p>Continue statement.</p> <p>Passes control to the loop-continuation point of an iteration statement. A continue statement may occur only in a while, do, or for statement; statements of these three kinds are called iteration statements.</p> <p>See “Branching Statements” at http://java.sun.com/docs/books/tutorial/java/nutsandbolts/branch.html.</p>

Table 3-15 Java Syntax Button Descriptions (continued)


Syntax Button	Enters into the Expression Editor palette, the syntax for a...
switch (?) { ? }	Switch statement. The switch statement transfers control to one of several statements depending on the value of an expression. The type of the Expression must be char, byte, short, int, BigInteger, float, double, BigDecimal, String, or Language, or a parse-time error occurs. This is an extension on the Java programming language where only the types char, byte, short, or int are supported. String switch statements are case insensitive. See “The switch Statement” at http://java.sun.com/docs/books/tutorial/java/nutsandbolts/switch.html .
 Note The switch statement is similar to the Cisco Unified CCX Editor Switch step.	
case ?:	Case statement. A block of code or a code branch destinations depending on the value of an expression in a switch statement. See “The switch Statement” at http://java.sun.com/docs/books/tutorial/java/nutsandbolts/switch.html .
default:	Default statement. Optionally used after all "case" conditions in a "switch" statement. If all "case" conditions are not matched by the value of the "switch" variable, the "default" statement is executed.
throw ?	Throw statement. Causes an exception to be thrown. The result is an immediate transfer of control that may exit multiple statements and the complex expression block containing it until a try statement is found that catches the thrown value. See “Method Throws” at http://java.sun.com/docs/books/jls/third_edition/html/classes.html#8.4.6
try { ? } catch (?) { ? } }	Try-catch statement. A try statement executes a block. If a value is thrown and the try statement has one or more catch clauses that can catch it, then control is transferred to the first such catch clause. See “Exception Handling Statements” at http://java.sun.com/docs/books/tutorial/java/nutsandbolts/exception.html .
try { ? } finally { ? }	Try-finally statement. If the try statement has a finally clause, then another block of code is executed, no matter whether the try block completes normally or abruptly, and no matter whether a catch clause is first given control. See “Exception Handling Statements” at http://java.sun.com/docs/books/tutorial/java/nutsandbolts/exception.html .

Table 3-15 Java Syntax Button Descriptions (continued)

Syntax Button	Enters into the Expression Editor palette, the syntax for a...
catch (?) { ? }	<p>Catch statement.</p> <p>Encloses some code and is used to handle errors and exceptions that might occur in that code.</p> <p>See “Exception Handling Statements” at http://java.sun.com/docs/books/tutorial/java/nutsandbolts/exception.html.</p>
finally { ? }	<p>Finally statement.</p> <p>Unconditionally executed after all other error processing has occurred. This guarantees execution of cleanup code when execution of a block of code is interrupted.</p> <p>See “Exception Handling Statements” at http://java.sun.com/docs/books/tutorial/java/nutsandbolts/exception.html.</p>

Language

Use the Language tab to add, delete, or modify languages in an expression. Language is a friendly data type corresponding to the fully qualified `java.util.Locale` class.

This topic includes the following:

- [Language Class and Code Specifications on the Web](#), page 3-84
- [Example Language Code](#), page 3-85
- [Language Variables](#), page 3-85
- [Language Methods and Attributes](#), page 3-85
- [Recent Languages](#), page 3-85
- [All Languages](#), page 3-86
- [Language tab Syntax Button](#), page 3-86
- [Language Literals](#), page 3-86

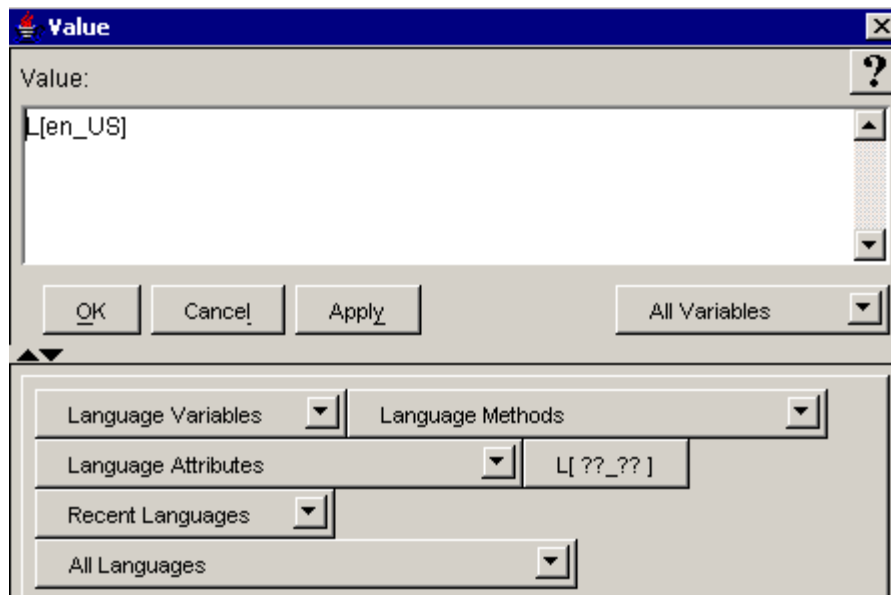
Language Class and Code Specifications on the Web

The Language class in the Cisco Unified CCX Expression Language is a friendly data type and is equivalent to the Sun Java Locale class. For the Sun Java specification on the Locale class, see <http://java.sun.com/j2se/1.4.2/docs/api/java/util/Locale.html>.

For a list of ISO language codes, see <http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>

For a list of ISO country codes, see http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html

Example Language Code



The following sections describe the options on the Language tab:

- [Language Variables, page 3-85](#)
- [Language Methods and Attributes, page 3-85](#)
- [Language tab Syntax Button, page 3-86](#)

Language Variables

The Language Variable selection box lists all the language variables contained in the currently opened script. Use this selection box to paste an already defined language variable into an expression.

A language variable is used to localize a particular resource in the system and can be associated with a contact to customize what prompts and grammars must be retrieved from the Cisco Unified CCX repository when required.

The default value of a language variable is the system default language.

Language Methods and Attributes

Use the appropriate selection box to add an available Language method or attribute to your expression.

The available public methods and attributes include both static and non static ones

For descriptions of all the public Language methods and attributes available in the selection boxes, see <http://java.sun.com/j2se/1.4.2/docs/api/java/util/Locale.html>.

Recent Languages

Lists all the languages currently used and defined since the last installation.

This list accumulates the languages you have used and defined since the last installation. As you use languages, they are appended to the list.

All Languages

Lists the languages that are defined in <http://ftp.ics.uci.edu/pub/ietf/http/related/iso639.txt>.

Language tab Syntax Button

The Language tab syntax buttons indicate all the ways you can add or use a Language in an expression. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values.

Table 3-16 Language Syntax Button Description

Syntax Button	Description
L[??_??]	language literal. See Language Literals , page 3-86. For example: L[en_US] or L[fr_CA]

Language Literals

Each friendly language literal is a reference to an instance of class `java.util.Locale`.

The language literal is always of type `Language`.

LanguageLiteral:
LanguageDeclarator [LanguageDesignator_{opt}]

LanguageDeclarator: one of
| L

LanguageDesignator:
Language Country_{opt} Variants_{opt}

Language:
any valid ISO language code defined as the lower-case two-letter codes
by ISO-639

Country:
_ CountryCode

CountryCode:
any valid ISO country code defined as the upper-case two-letter codes
by ISO-3166

Variants:
Variant
Variants Variant

Variant:
_ VariantCode

VariantCode:
VariantCharacter
VariantCode VariantCharacter

VariantCharacter:

InputCharacter but not _

The VariantCode is a free form string which is used to further qualify the language. The `_EURO` literal is normally used to specify the use of the Euro Currency for a European country. It could also be used to distinguish between different talents used for the prompt recordings (for example, `L[en_CA_John]` versus `L[en_CA_Jenna]`). The variant can be composed of many variant parts each separated with an underscore (for example, `L[de_DE_EURO_Joe]`).

Each language literal is a reference to an instance of class `java.util.Locale`.



Note

The language literal requires that the proper language pack is installed.

Long

Use the Expression Editor Long tab to enter or modify Long data in an expression. Long is a friendly data type corresponding to the fully qualified `java.lang.Long` class.



Note

In the Expression Language, `long` and `Long` can be used interchangeably as opposed to Java where `long` represents a primitive data type and `Long` represents an object.

This topic includes the following:

- [About the Long Data Type, page 3-87](#)
- [Long Specification on the Web, page 3-87](#)
- [Example Long Code, page 3-87](#)
- [Long Variables, page 3-89](#)
- [Long Constructors, Methods, and Attributes, page 3-89](#)
- [Long tab Syntax Buttons, page 3-89](#)

About the Long Data Type

The `java.lang.Long` numeric data type is a 32-bit Integer and its value can be from -9223372036854775808 to 9223372036854775807, inclusive.

Long Specification on the Web

For the Sun Java specification of the Java class `Long`, see:
<http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Long.html>

For the Sun Java specification on numeric types, see “*Types, Values, and Variables*” at
http://java.sun.com/docs/books/jls/second_edition/html/typesValues.doc.html#9164

Example Long Code

In the following two examples, the script variables used in the expressions are listed in the top right of each example.

Long

Figure 3-21 Example Simple Expression Using a Long and Script Variables

Name	Type	Value
amount	String	"243697"

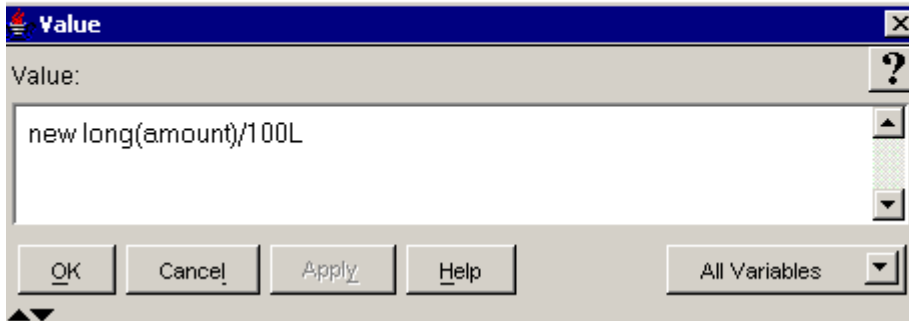
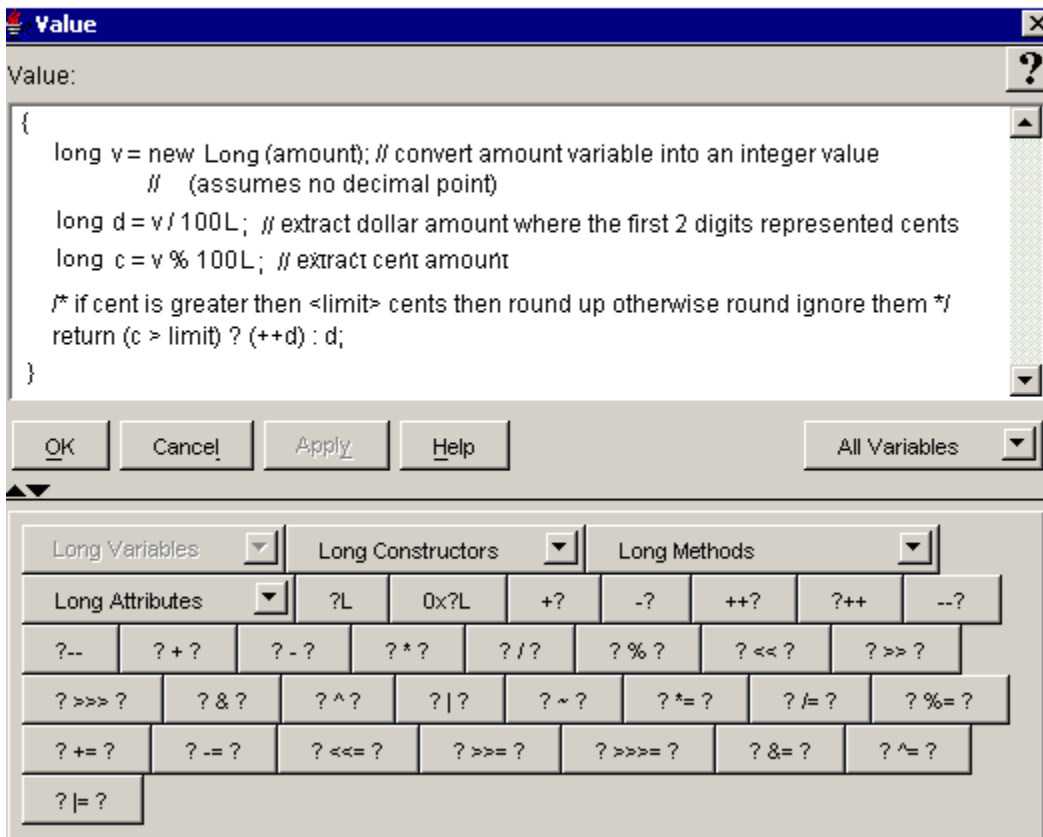


Figure 3-22 Example Complex Expression Using a Long and Two Script Variables

Name	Type	Value
amount	String	"243697"
limit	long	0L



The following sections describe the options available on the Long tab:

- [Long Variables, page 3-89](#)

- [Long Constructors, Methods, and Attributes, page 3-89](#)
- [Long tab Syntax Buttons, page 3-89](#)

Long Variables

The Long Variables selection box lists all the long variables contained in the currently opened script. Use this selection box to paste an already defined Long variable into an expression.

The Long variable holds the value of a long and is an expanded Integer variable. Its value ranges from from -9223372036854775808 to 9223372036854775807, inclusive.

The default value of a long variable is zero, that is, 0L.

Long Constructors, Methods, and Attributes

Use the appropriate selection box to add a public constructor, method, or attribute in your Cisco Unified CCXscript expression.

The available public methods and attributes include both static and non static ones.

For descriptions of all the public Long constructors, methods, and attributes available in the selection boxes, see the Java specification at:

<http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Long.html>

Long tab Syntax Buttons

The Long tab syntax buttons indicate all the ways you can add a long data type to an expression. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values.

Table 3-17 Long Syntax Button Descriptions

Syntax Button	Name	Type	Description
?L	literal	decimal	A Long literal in decimal format. See Integer Literals, page 3-75 . for example: 234556789L 0L
0x?L	literal	hexadecimal	A Long literal in hexadecimal format.
+?	unary plus	unary	The positive value of the operand.
-?	unary minus		The negative value of the operand. For example: -23L
++? ¹	prefix increment	increment	Increments the value of the operand by one before the operand is changed in an expression.
?++ ¹	postfix increment		Increments the value of the operand by one after the operand is changed in an expression.

Table 3-17 Long Syntax Button Descriptions (continued)

Syntax Button	Name	Type	Description
--? ¹	prefix decrement	decrement	Decrements the value of the operand by one before the operand is changed in an expression.
?-- ¹	postfix decrement		Decrements the value of the operand by one after the operand is changed in an expression.
? + ?	addition	arithmetic	Adds two operands.
? - ?	subtraction		Subtracts the second operand from the first.
? * ?	multiplication		Multiplies two operands.
? / ?	division		Divides the first operand by the second.
? % ?	remainder		Returns the remainder of the first operand divided by the second.
? << ?	shift left	bitwise shift (for operations on individual bits in Integers only)	Shifts bits of operand 1 left by the distance of operand 2; fills with zero bits on the right-hand side.
? >> ?	shift right		Shifts bits of operand 1 right by the distance of operand 2; fills with the highest (signed) bit on the left-hand side.
? >>> ?	zero fill right shift		Shifts bits of operand 1 right by a distance of operand 2; fills with zero bits on the left-hand side.
? & ?	bitwise AND	bitwise logical (for operations on individual bits in Integers only)	Compares both operands. If both operand bits are 1, the AND function sets the resulting bit to 1; otherwise, the resulting bit is 0.
? ^ ?	bitwise exclusive OR (XOR)		Compares both operands. If both operand bits are different, the resulting bit is 1; otherwise the resulting bit is 0.
? ?	bitwise inclusive OR		Compares both operands. If either of the two operand bits is 1, the resulting bit is 1. Otherwise, the resulting bit is 0.
~ ?	Bitwise complement		Inverts the value of each operand bit: If the operand bit is 1, the resulting bit is 0; if the operand bit is 0, the resulting bit is 1.
? *= ?	multiply and assign	assignment The operand on the left of the assignment statement (the first operand) can be any type of variable, including an array component or a public class attribute.	Multiplies the first operand by the second and assigns the result to the first operand.
? /= ?	divide and assign		Divides the first operand by the second and assigns the result to the first operand.
? %= ?	remainder and assign		Divides the first operand by the second operand and assigns the remainder to the first operand.
? += ?	add and assign		Adds the first operand to the second operand and assigns the result to the first operand.
? -= ?	subtract and assign		Subtracts the second operand from the first operand and assigns the result to the first operand.

Table 3-17 Long Syntax Button Descriptions (continued)

Syntax Button	Name	Type	Description
? <<= ?	left shift and assign		Shifts bits of operand 1 left by the distance of operand 2; fills with zero bits on the right-hand side and assigns the resulting bit to operand 1.
? >>= ?	right shift and assign		Shifts bits of operand 1 left by the distance of operand 2; fills with zero bits on the right-hand side and assigns the resulting bit to operand 1.
? >>>= ?	zero fill, right shift, and assign		Shifts bits of operand 1 right by a distance of operand 2; fills with zero bits on the left-hand side and assigns the resulting bit to operand 1.
? &= ?	AND and assign		First, compares both operands. If both operand bits are 1, the AND function sets the resulting bit to 1; otherwise, the resulting bit is set to 0. Then, assigns the resulting bit to operand 1.
? ^= ?	XOR and assign		First, compares both operands. If both operand bits are different, the resulting bit is 1; otherwise the resulting bit is 0. Then, assigns the resulting bit to operand 1.
? = ?	OR and assign		First, compares both operands. If either of the two operand bits is 1, the resulting bit is 1. Otherwise, the resulting bit is 0. Then, assigns the resulting bit to operand 1.

1. The operand for the prefix and postfix increment operators must be a variable, an array component, or a public class attribute.

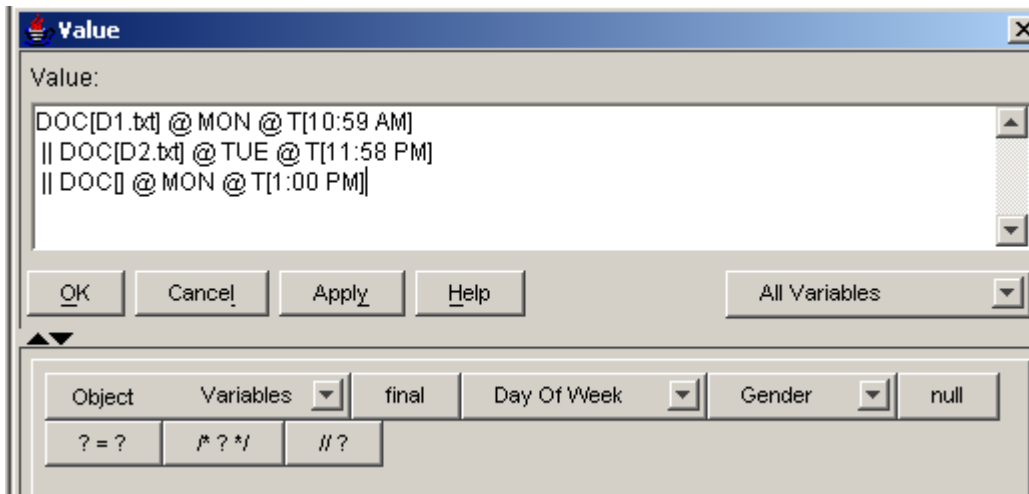
Miscellaneous

The Expression Editor Miscellaneous tab provides you a way to easily enter types of data into an expression that are not included in the other tabs.

This section covers the following topics:

- [Example Simple Expression Using the Miscellaneous Tab, page 3-92](#)
- [Object Variables, page 3-92](#)
- [Miscellaneous tab Syntax Buttons, page 3-93](#)

Example Simple Expression Using the Miscellaneous Tab



The following sections describe the options on the Miscellaneous tab:

- [Object Variables, page 3-92](#)
- [Miscellaneous tab Syntax Buttons, page 3-93](#)

Object Variables

The Object Variables selection box lists all the variables created in the currently opened script. This allows you to paste one of these variables into an expression.



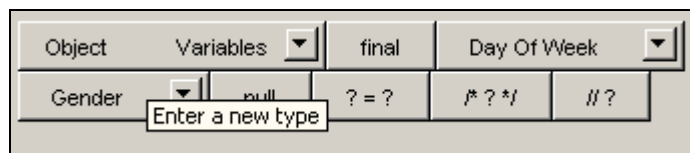
Note

By entering a data class name of your choice in the “Object” field header name of the Object Variables list selection box, you can filter out variables in a script that you want.

To access a script variable variable of your own data choice:

- Step 1** In the Cisco Unified CCX Expression Editor Miscellaneous tab, place the cursor over the word “Object” in the Object Variable selection box header.

A yellow pop-up window appears saying: Enter a new type.



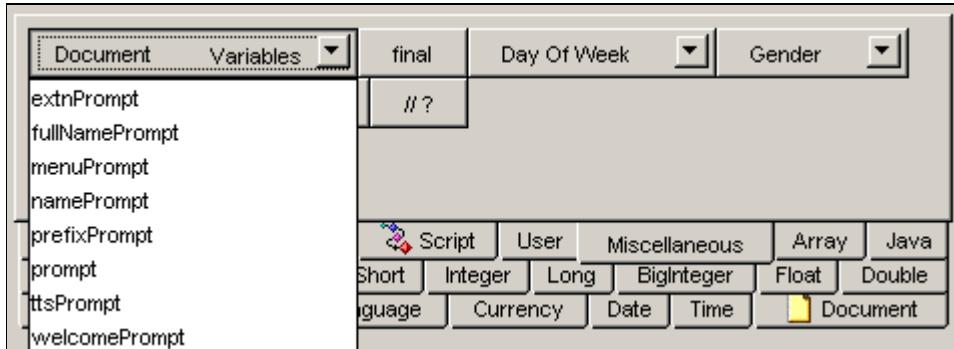
- Step 2** Double click the word **Object**.

The word Object is highlighted in blue and the arrow cursor changes to an input cursor (“I”) indicating this is a text field that you can change.

- Step 3** Enter the Java class name for the variables that you want listed.

The drop down list of variables is filtered to the entered type.

- Step 4** Click the selection arrow to display the list of variables available for that data class. In the following example, the user entered “Document” to replace “Object” in the variable selection box.



- Step 5** Select an item in the list to enter it into the Expression Editor Value text field.

DayOfWeek

Allows you to select a 3-letter abbreviation for a day of the week variable to be pasted into the expression: MON, TUE, WED, THU, FRI, SAT, SUN.

The DayOfWeek literal can be used as a qualifier for a prompt or a document expression when defining day of week prompts or documents. It has no type and cannot be used elsewhere in the grammar. The literals are case insensitive.

Gender

Allows you to select a gender type to be pasted into the expression: MALE, FEMALE, NEUTRAL.

The Gender literal can be used when defining a number prompt, ordinal, or TTS prompt to qualify the gender context into which the prompt generation must be tailored. It has no type and cannot be used elsewhere in the grammar. The literal is case insensitive.

The Null Literal

The null type has one value, the null reference, represented by the literal null, which is formed from ASCII characters. A null literal is always of the null type.

Miscellaneous tab Syntax Buttons

The Miscellaneous tab syntax buttons provides you a way to easily enter types of data into an expression that are not included in the other tabs. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values.

Table 3-18 *Miscellaneous Syntax Button Descriptions*

Syntax Button	Name	Type	Description
final	constant	final	The final local variable modifier. This marks the variable as one that cannot have its value changed. Such a variable is known as a constant and can be used to define other non-final variable initial values. The keyword final can also be prefixed to the data type for the same result, to make a variable final of type Integer, as in the following example: final int
null	null	value	The null value.
? = ?	assignment	statement	The assignment statement.
/* ? */	comment	block	A block of comments.
//?	comment	line	A comment on one line.

Prompt

Use the Prompt tab to add, delete, or modify Prompts in a Cisco Unified CCX script expression.

The Prompt friendly data type corresponds to the Java `com.cisco.prompt.Playable` class.

This section includes the following topics:

- [About Prompts, page 3-94](#)
- [Note You can localize prompts to suit your local language. For information on that, see “Localizing Cisco Unified CCX Scripts” in the Cisco Unified CCX Scripting and DExample Simple Expression Using a Prompt, page 3-95](#)
- [Prompt Variables, page 3-95](#)
- [Browse Prompts Dialog Box, page 3-96](#)
- [Prompt tab Syntax Buttons, page 3-96](#)
- [Prompt Literals, page 3-98](#)
- [Operators Used with Prompts, page 3-104](#)
- [Prompt Templates, page 3-106](#)
- [Prompt Conversions, page 3-109](#)

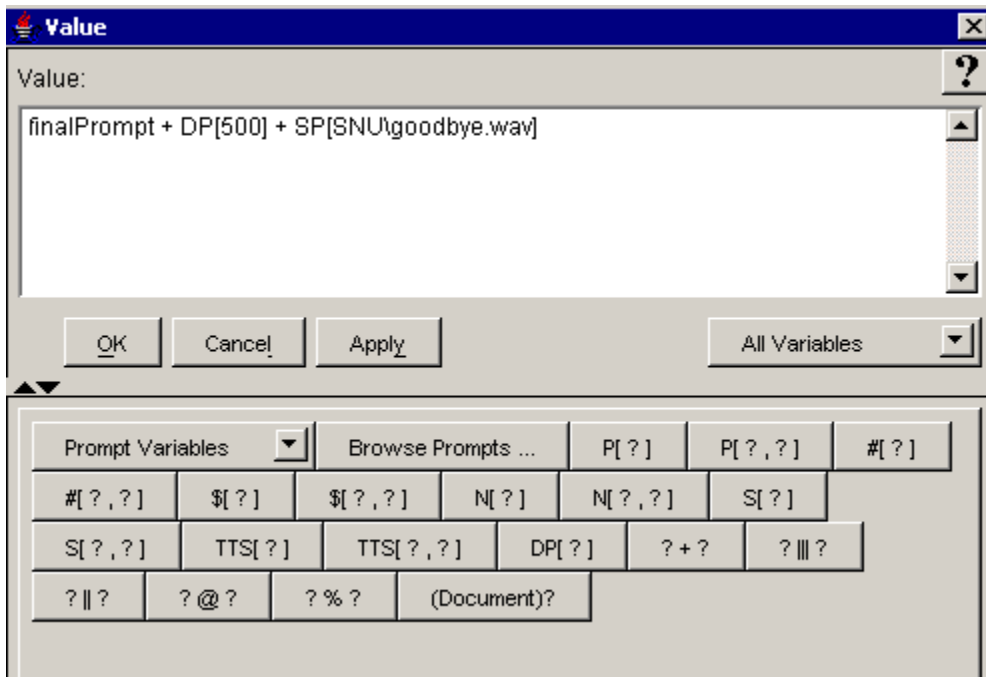
About Prompts

Instances of class `Prompt` represent audio data that can be played back to a caller. A `Prompt` object has a constant (unchanging) value. Complex prompt literals are references to instances of class `Prompt`.

The prompt concatenation operator +, the prompt escalation, time of day prompt, time of week prompt, day of week prompt, and random prompt operators ||, and the prompt substitution operator ||| implicitly create a new Prompt object. An expression that creates a prompt is called a prompt expression. See [Prompt tab Syntax Buttons](#), page 3-96 for examples of prompt expressions.

**Note**

You can localize prompts to suit your local language. For information on that, see “Localizing Cisco Unified CCX Scripts” in the *Cisco Unified CCX Scripting and DExample Simple Expression Using a Prompt*



The following sections describe the options on the Prompt tab:

- [Prompt Variables](#), page 3-95
- [Browse Prompts Dialog Box](#), page 3-96
- [Prompt tab Syntax Buttons](#), page 3-96
- [Prompt Literals](#), page 3-98
- [Operators Used with Prompts](#), page 3-104

Prompt Variables

The Prompt Variable selection box contains all the prompt variables created in the currently opened script. Use this selection box to paste a Prompt variable into an expression.

A Prompt variable contains information about what to play to a caller when a call is passed to a Media step. It can reference audio files in the prompt repository or on disk, concatenation of multiple prompts, or more complicated types of prompts.

The default value of a prompt variable is the empty prompt, that is, P[]

Browse Prompts Dialog Box

Use the Browse Prompts selection box to add a Prompt to your script expression by browsing the local disk or the Prompt repository.

Prompt tab Syntax Buttons

The Prompt tab syntax buttons indicate all the ways you can add a Prompt object to an expression. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values. See [Prompt Literals, page 3-98](#) for code examples. For operations you can perform on prompts, see [Operators Used with Prompts and Documents, page 1-8](#).

Table 3-19 Prompt Syntax Button Descriptions

Syntax Button	Prompt Type or Operator	Description
P[?]	User-Defined Prompt A prompt located in the prompt repository and manageable using the prompt management pages that are part of the Cisco Unified CCX Administration Web page.	User prompt. See Prompt Literals, page 3-98 and User Prompt Literals, page 3-99 . Examples: P[text.tts] P[AA\Welcome.wav]
P[?,?]		User prompt with optional arguments. See Prompt Literals, page 3-98 and User Prompt Literals, page 3-99 . Example: P["currency.tpl", amount, C[USD]]
#[?]	Ordinal Prompt A prompt that corresponds to the spoken ordinal position of a specified number in a parameter.	Ordinal Prompt (neutral gender). See Ordinal Prompt Literals, page 3-100 . Examples: #[2] // second #[3.3F] // third
#[?,?]		Ordinal prompt with gender argument. See Ordinal Prompt Literals, page 3-100 . Example: #[2 + i, FEMALE]
\${?}	Currency Prompt The spoken amount representation of the specified currency arguments.	Currency prompt (the system default currency). See Currency Prompt Literals, page 3-100 . Examples: \${2} // 2 dollars \${3.3F} // 3 dollars and 30 cents \${"23.33"}
\${?,?}		Currency prompt with additional arguments. See Currency Prompt Literals, page 3-100 . Examples: \${"23.33", true} \${"123.33", C[CAD]} \${"123.33", false, C[DEM]}

Table 3-19 Prompt Syntax Button Descriptions (continued)

Syntax Button	Prompt Type or Operator	Description
N[?]	Number Prompt The spoken number representation of the specified arguments.	Number prompt (neutral gender). See Number Prompt Literals, page 3-101 . Examples: <pre>N[2345.3D] N["12" + '.' + "23"]</pre>
N[?,?]		Number prompt with additional arguments. See Number Prompt Literals, page 3-101 . Examples: <pre>N[0x233, false] N["1223", Boolean.TRUE] N[45, MALE] N["3.23", NEUTRAL] N[11, 1] N[2000000, false, FEMALE] N[-2e23, Boolean.TRUE, 1] N["29.0002", true, MALE]</pre>
S[?]	Spelling Prompt A string spelled back one character at a time.	See Spelling Prompt Literals, page 3-102 . Examples: <pre>S[A] S[John Doe] S[\n b]; S["some text"] S['\f'] S['a' + " nice day"] S["b"] S[u\t"] S[java.util.Locale.US]</pre>
S[?,?]		Spelling prompt with additional arguments. See Spelling Prompt Literals, page 3-102 . Examples: <pre>S["a", true] S[java.util.Locale.US, false]</pre>
TTS[?]	TTS (Text To Speech) prompt A prompt generated from textual content using a TTS server.	TTS prompt. See TTS Prompt Literals, page 3-103 . Examples: <pre>TTS[This is an example] TTS[John Doe] TTS[URL["http://localhost/email.doc"]] TTS[new java.io.File(u"C:\\help.ssml")] TTS["Some text to be rendered", "Nuance Vocalizer 3.0"]</pre>
TTS[?,?]		TTS prompt with provider argument. See TTS Prompt Literals, page 3-103 . Example: <pre>S[java.util.Locale.US, false]</pre>

Table 3-19 Prompt Syntax Button Descriptions (continued)

Syntax Button	Prompt Type or Operator	Description
DP[?]	Delay Prompt A silence pause of the specified number of milliseconds in parameter.	Delay prompt. See Delay Prompt Literals, page 3-104 . Examples: DP[500] // 500 milliseconds of silence DP[1000.45] // 1 second of silence
? + ?	Prompt Concatenation Operator	Prompt concatenation. If only one operand expression is of type Prompt, then prompt conversion is performed on the other operand to produce a prompt at run time. The result is a reference to a newly created Prompt object that is the concatenation of the two operand prompts. The content of the left-hand operand precedes the content of the right-hand operand in the newly created prompt.
? ?	Prompt Substitution Operator	Prompt substitution. A prompt used as a substitute prompt when another is no longer available. For example, if a failure occurs while attempting to queue a prompt, then the substitute is queued instead. See Prompt Substitution Operator , page 3-104 .
? ?	Escalation Operator	Can create a Time of week, day of week, time of day, random, or escalation prompt. See Prompt Escalation Operator , page 3-105 .
? @ ?	Prompt Qualification Operator	Prompt qualifier. The prompt qualifier @ expects a qualifying expression of the following type: <ul style="list-style-type: none"> • Language • DayOfWeekLiteral • Number • Time See Prompt Qualifier Operator @, page 3-105
? % ?	Prompt Weight Qualification Operator	Prompt weight. The prompt weight qualifier % expects a qualifying expression of the Number type and is used to assign a weight to a prompt when used in a random prompt expression. See Prompt Weight Qualifier Operator %, page 3-105 and Random Prompt, page 3-109 .
(Document)?	Prompt Conversion Operation	Converts a prompt into an audio document. See Prompt Conversions, page 3-109 .

Prompt Literals

This topic covers the following:

- [About Prompt Literals, page 3-99](#)
- [User Prompt Literals, page 3-99](#)
- [Ordinal Prompt Literals, page 3-100](#)
- [Currency Prompt Literals, page 3-100](#)

- [Number Prompt Literals, page 3-101](#)
- [Spelling Prompt Literals, page 3-102](#)
- [TTS Prompt Literals, page 3-103](#)
- [Delay Prompt Literals, page 3-104](#)

About Prompt Literals

The prompt literal is always of type `Prompt`. Each prompt literal is a reference to an instance of a class that implements the interface `com.cisco.prompt.Playable`.

```
PromptLiteral:
  UserPrompt
  OrdinalPrompt
  CurrencyPrompt
  NumberPrompt
  SpellingPrompt
  TTSPrompt
  DelayPrompt
  GeneratedPrompt
```

Example Prompt Literals:

- `P[]`—An empty prompt. (No prompt gets played back.)
- `P[AA\AAWelcome.wav]`—A user-defined prompt located in the User Prompts directory.

User Prompt Literals

The user prompt literal is always of type `Prompt`.

```
UserPrompt:
  UserPromptDeclarator [ComplexLiteralInputCharsopt]
  UserPromptDeclarator [Expression]
  UserPromptDeclarator [Expression, ArgumentList]
```

UserPromptDeclarator: one of
p P

User prompt literals are used to represent a prompt located in the prompt repository and manageable using the prompt management pages which are part of the Cisco Unified CCX Administration Web page. The `ComplexLiteralInputChars` can include the [character as long as it has a balanced number of] characters: one for every [character found:

- If the sequence of characters can be parsed as an `Expression` of type `String`, then the resulting prompt is a user prompt where the expression specifies the name of the prompt to retrieve the prompt from the repository.
- If the sequence of characters can be parsed as an `Expression` and an `ArgumentList` where the first one must have type `String`, then the resulting prompt is a user prompt where the first argument specifies the name of the prompt to retrieve from the repository and the argument list must correspond to the expected parameterized arguments of a complex expression block defined in a prompt template file.

The arguments are ignored if the referenced prompt is not a prompt template. If it is one, then each specified argument is evaluated and assigned as the value of a defined argument to the expression block. If the types do not match, then a runtime exception is thrown back. No errors are generated

if more arguments are supplied than expected; they are ignored. No errors are generated if fewer arguments are supplied than expected unless the given argument is accessed by the complex expression block and it was not defined with a default value

- If the sequence of characters cannot be parsed properly as described above, then it is considered to be the name of the user prompt to retrieve from the repository.

Example User Prompts Literals:

```
P[text.tts]
P[AA\Welcome.wav]
P[currency.tpl]
P[prompt]
P["currency.tpl", amount, C[USD]]
```

The extension of the prompt file can be omitted in which case the search looks at all supported extensions (<.wav>, <.ssml>, <.tts>, <.tpl>) in order until one is found.



Note

The special case of P [] represents an empty prompt.

Ordinal Prompt Literals

The ordinal prompt literal is always of type Prompt.

```
OrdinalPrompt:OrdinalPromptDeclarator [ Expression ]
OrdinalPromptDeclarator [ Expression , Expression ]
OrdinalPromptDeclarator:
#
```

Ordinal prompts correspond to the spoken ordinal position of the specified number in a parameter. Floating point and string literals are converted to an Integer representation before being converted to an ordinal spoken representation. The first expression must be of type String or a java.lang.Number type (that is, integral or floating-point). The second expression, if supplied, must be either a predefined gender constant (for example, MALE, FEMALE, or NEUTRAL) or an int type that results to 0 for neutral, 1 for male and 2 for female. This form also assumes that the language of the text correspond to the current language of the context unless the prompt is further qualified using the @ operator.

Example Ordinal Prompt Literals:

```
#[2] // second
#[3.3F] // third
#[2 + i, FEMALE]
```

Currency Prompt Literals

The currency prompt literal is always of type Prompt.

```
CurrencyPrompt:
CurrencyPromptDeclarator [ArgumentList]

CurrencyPromptDeclarator:
$
```

Currency prompts correspond to the spoken amount representation of the specified arguments. The CurrencyPromptDeclarator is the symbol for the specified currency. For example: \$, £, or ¥. There cannot be more than four arguments in the ArgumentList and the supported combination are listed as follows.

Table 3-20 *Currency Prompt Literal Arguments*

1 st Argument	2 nd Argument	3 rd Argument	4 th Argument
Amount			
Amount	Colloquial Flag		
Amount	Currency		
Amount	Colloquial Flag	Currency	
Dollar Amount	Cent Amount		
Dollar Amount	Cent Amount	Colloquial Flag	
Dollar Amount	Cent Amount	Currency	
Dollar Amount	Cent Amount	Colloquial Flag	Currency

An Amount argument must be of type String or a java.lang.Number type (for example, integral or floating-point). The Colloquial Flag argument must be of type Boolean and specifies whether to use colloquial currencies representation like "Dollars" instead of "US Dollars". The Currency argument must be of type Currency.

This form also assumes the current language of the context unless the prompt is further qualified using the @ operator.

Example Currency Prompt Literals:

```

${2} // 2 dollars
${3.3F} // 3 dollars and 30 cents
$["23.33"]
$["23.33", true]
$["123.33", C[CAD]]
$["123.33", false, C[DEM]]
${23.33 + 2}
${amount, true}
${123.33 + 3, C[CAD]}
${balance, false, C[DEM]}
${33, 2}
${15, 7, true}
${17, 66, C[CAD]}
${3455 - 3, 88, false, C[DEM]}

```

Number Prompt Literals

The number prompt literal is always of type Prompt.

```

NumberPrompt:
  NumberPromptDeclarator [ArgumentList]
NumberPromptDeclarator: one of
  n N

```

Number prompts correspond to the spoken number representation of the specified arguments. There cannot be more than three arguments in the ArgumentList and the supported combination are listed in [Table 3-21](#).

Table 3-21 Number Prompt Literal Arguments

1 st Argument	2 nd Argument	3 rd Argument
Number		
Number	Full Format Flag	
Number	Gender	
Number	Full Format	Gender

The Number argument must be of type String or a java.lang.Number type (for example, integral or floating-point). The Full Format Flag argument must be of type Boolean and must specify to play the number in full format (that is, 709 is played as "Seven Hundred and Nine"); otherwise, plays the number in brief format that is, 709 becomes "Seven Oh Nine"). The Gender argument must be either a predefined gender constant (that is, MALE, FEMALE, or NEUTRAL) or an int type that results to 0 for neutral, 1 for male and 2 for female.

This form also assumes the current language of the context unless the prompt is further qualified using the @ operator.

Example Number Prompt Literals:

```
N[2345.3D]
N["12" + '.' + "23"]
N[0x233, false]
N["1223", Boolean.TRUE]
N[45, MALE]
N["3.23", NEUTRAL]
N[11, 1]
N[2000000, false, FEMALE]
N[-2e23, Boolean.TRUE, 1]
N["29.0002", true, MALE]
```

Spelling Prompt Literals

The spelling prompt literal is always of type Prompt.

```
SpellingPrompt:
    SpellingPromptDeclarator [ ComplexLiteralInputChars ]
    SpellingPromptDeclarator [ Expression ]
    SpellingPromptDeclarator [ Expression , Expression ]

SpellingPromptDeclarator:
    s S
```

Spelling prompts correspond to a string being spelled back one character at a time:

- If the sequence of characters can be parsed as an Expression of any type then its string representation, returned by the Java method toString() of the object, is spelled back. If an additional Expression can be parsed then it must be of type Boolean and represents whether or not special characters must be spelled back as well instead of being played back as silences if it is false.
- If the sequence of characters cannot be parsed properly as described above, then it is considered to be the text to be spelled back.

This form also assumes that the language of the text correspond to the current language of the context unless the prompt is further qualified using the @ operator.

Example Spelling Prompt Literals:

```

S[A]
S[John Doe]
S[\n b];
S["some text"]
S['\f']
S['a' + " nice day"]
S["b"]
S[u\t"]
S["a", true]
S[java.util.Locale.US]
S[java.util.Locale.US, false]

```

TTS Prompt Literals

The TTS prompt literal is always of type Prompt.

```

TTSPrompt:
  TTSPromptDeclarator [ComplexLiteralInputChars]
  TTSPromptDeclarator [Expression]
  TTSPromptDeclarator [Expression , Expression]
TTSPromptDeclarator:
  any case of TTS

```

TTS prompt literals are used to represent a prompt that is generated from textual content using a TTS server. This type of prompt requires TTS to be licensed and installed for it to work properly at run time.

If the sequence of characters can be parsed as an Expression of type String, Document, java.net.URL, or java.io.File then the resulting prompt is a TTS prompt where the expression specifies the textual content to convert, a document containing the text to convert, a URL where to get the text to convert or a file containing the text to convert respectively.

When specified in this form, the system default TTS provider, configured through the Cisco Unified CCX Application Administration Web page, is selected as the provider to be contacted to perform the resolution. This form also assumes that the language of the text correspond to the current language of the context unless the prompt is further qualified using the @ operator:

- If the sequence of characters can be parsed as two Expressions where the first one must have type String, Document, java.net.URL, or java.io.File and the second one must have type String, then the resulting TTS prompt corresponds to the content specified from the first expression as described above using the specified TTS provider if available. This form also assumes that the language of the text correspond to the current language of the context unless the prompt is further qualified using the @ operator.
- If the sequence of characters cannot be parsed properly, as described above, then it is considered to be the textual content to be converted as a TTS prompt. When specified in this form, the system default TTS provider, configured through the Cisco Unified CCX Administration Web page, is selected as the provider to be contacted to perform the resolution. This form also assumes that the language of the text corresponds to the current language of the context unless the prompt is further qualified using the @ operator.

Example TTS Prompt Literals:

```

TTS[This is an example]
TTS[John Doe]
TTS["Some text to be rendered", "Nuance Vocalizer 3.0"]
TTS[URL["http://localhost/email.doc"]]
TTS[new java.io.File(u"C:\\help.ssml")]

```

Delay Prompt Literals

The delay prompt literal is always of type Prompt.

```
DelayPrompt:
    DelayPromptDeclarator [Expression]
```

```
DelayPromptDeclarator:
    any case of DP
```

Delay prompts correspond to a silence pause of the specified number of milliseconds in parameter. Floating point literals and string are converted to an Integer representation. The expression must be of type String or a java.lang.Number type (that is, integral or floating-point).

Example Delay Prompt Literals:

```
DP[500] // 500 milliseconds of silence
DP[1000.45] // 1 second of silence
```

Operators Used with Prompts

You can use the following operators with prompts:

- [Prompt Concatenation Operator +](#), page 3-104
- [Prompt Substitution Operator |||](#), page 3-104
- [Prompt Qualifier Operators](#), page 3-105
- [Prompt Qualifier Operator @](#), page 3-105
- [Prompt Weight Qualifier Operator %](#), page 3-105
- [Prompt Escalation Operator ||](#), page 3-105

Prompt Concatenation Operator +

If only one operand expression is of type Prompt, then prompt conversion is performed on the other operand to produce a prompt at run time. The result is a reference to a newly created Prompt object that is the concatenation of the two operand prompts. The content of the left-hand operand precedes the content of the right-hand operand in the newly created prompt.

Prompt Substitution Operator |||

The operator ||| is called the prompt substitution operator. It is used to create a substitute prompt. A substitute prompt is a prompt where the first prompt is queued for playback whenever the substitute prompt is used in a media context. If a failure occurs while attempting to queue the prompt then the substitute is queued instead. For example the main prompt could represent a TTS prompt which in cases where the system has not been installed or licensed with TTS support, one would want to fallback to a pre-recorded prompt. In this case, queuing a TTS prompt would fail and the substitute would be used instead. This operator is not associative.

```
SubstituteExpression:
    PromptExpression ||| PromptExpression
```

See [Prompt Templates](#), page 3-106 for examples of prompt expressions

Prompt Qualifier Operators

Qualifier operators are used to further qualify objects by assigning them new or different properties. Qualified objects can be used just as their normal objects. However, in some cases the qualification applied to an object can be used to determine what kind of container prompt will result from the `||` operator (see [Prompt Escalation Operator ||](#), page 3-105). The language qualification is the only qualifier that is not ignored if not used in conjunction with the `||` operator. Prompts can be qualified multiple times through multiple types of qualifications.

Prompt qualifier operators results in an expression of the Prompt type.

```
QualifiedPromptExpression:
  PromptExpression
  QualifiedPromptExpression @ Expression
  QualifiedPromptExpression % Expression
```

See the following two topics for the meaning of qualified prompt expressions:

- [Prompt Qualifier Operator @](#), page 3-105
- [Prompt Weight Qualifier Operator %](#), page 3-105

Prompt Qualifier Operator @

The prompt qualifier `@` expects a qualifying expression of the following type:

- Language
- DayOfWeekLiteral
- Number
- Time

The first qualifier represents a language qualification and is used to temporarily override the language associated with a given prompt. The expression must be of type `Language`. Qualifying a prompt more than once with a language will result in only the last one being kept as the overridden language for the prompt.

The second qualifier represents a day of week qualification and is used to specify the starting day of a possible range when the prompt is to be used in a day of week prompt or time of week prompt expression. The starting day can also be specified using a `Number` type as seen in the third option where its value must evaluate to 1 for Sunday, 2 for Monday ... or 7 for Saturday.

The last qualifier represents time qualification and is used to specify the starting time of a possible range when the prompt is to be used in a time of day prompt or time of week expression.

Prompt Weight Qualifier Operator %

The prompt qualifier `%` expects a qualifying expression of the `Number` type and is used to assign a weight to a prompt when used in a random prompt expression. See also [Random Prompt](#), page 3-109.

Prompt Escalation Operator ||

The prompt escalation `||` operator can be used to create escalation prompts, day of week prompts, time of day prompts, time of week prompts or random prompts. If at least one of the operands is a prompt, the other is converted to a prompt according to the rules set forth by [Table 1-7](#) and the result will be a new prompt. It is syntactically left-associative (it groups left-to-right). It is fully associative with respect

to both side effects and result value; that is, for any expressions a, b, and c, evaluation of the expression ((a)|(b))|(c) produces the same result, with the same side effects occurring in the same order, as evaluation of the expression (a)|((b)|(c)).

```
PromptEscalationExpression:
  PromptExpression || PromptEscalationExpression
```

The determination of the type of prompt that results from this operator depend on how the first two prompt operands (in a sequence of || operators) was qualified using the @ or % operators:

- If both prompt operands are qualified with a time of day and a day of week then the resulting prompt is a time of week prompt. All remaining operands of subsequent || operators are going to be added as prompts for subsequent time of week and must then be qualified with at least both a time of day and a day of week or a parse-time error occurs. Other qualifiers if present are ignored.
- Otherwise, if both prompt operands are qualified with a day of week then the resulting prompt is a day of week prompt. All remaining operands of subsequent || operators are added as prompts for the subsequent day of week and must then be qualified with at least a day of week or a parse-time error occurs. Other qualifiers if present are ignored.
- Otherwise, if both prompt operands are qualified with a time of day then the resulting prompt is a time of day prompt. All remaining operands of subsequent || operators are added as prompts for the subsequent time of day and must then be qualified with at least a time of day or a parse-time error occurs. Other qualifiers if present are ignored.
- Otherwise, if both prompt operands are qualified with a weight then the resulting prompt is a random prompt. All remaining operands of subsequent || operators are added as additional prompts and must then be qualified with at least a weight or a parse-time error occurs. Other qualifiers if present are ignored.
- Otherwise, the resulting prompt is an escalation prompt. All remaining operands of subsequent || operators are added as subsequent escalation and their qualifications are ignored.

Prompt Templates

This section contains the following topics:

- [About Prompt Templates, page 3-106](#)
- [Escalating Prompt, page 3-108](#)
- [Time of Week Prompt, page 3-108](#)
- [Day of Week Prompt, page 3-109](#)
- [Time of Day Prompt, page 3-109](#)
- [Random Prompt, page 3-109](#)

About Prompt Templates

A prompt template is a prompt represented as an expression and evaluated at the time it is queued up for playback.

There is added support for a new type of prompt file to the user and system prompts already available. This new file has the filename extension .tpl and can be referenced in a script just like the other .wav prompt files could.

In addition, not related to the expression, there is added support for two new prompt file extensions: `.tts` and `.ssml`. Files ending with these extensions are expected to be text files containing the text to be rendered as audio using a configured TTS server.

Since Cisco CRS 3.0, when referencing a user or system prompt, the extension of the file was optional. If the extension `.wav`, `.tpl`, `.tts` or `.ssml` is specified, Cisco CRS searches for only this specific prompt. If no extension is specified, the search starts with a `.wav` file, and if none is found, then a file with an `.ssml` extension is searched, and then `.tts` extension, and finally `.tpl` extension is searched.

If no extension is located for the first language in the language context, then the search moves to its parent language or the next one in the context. This search is similar to the one that existed in Cisco CRS 3.0 for user and system grammars where files with the extension `.gsl` and `.digit` are supported.

When a user or system prompt with the `.tpl` extension is located, it is loaded as a text file and parsed by the Expression Language manager and the result must be a prompt object, or an object of a data type that can be converted to a Prompt as described in. The expression specified in the text file does not have access to script variables. However, if defined using a complex block expression, the block can be parameterized like a method declaration, allowing for the scripts to customize the evaluation of the expression.

For example, say we have the following user prompt in the user repository defined as `currency.tpl`:

```
(float amount, boolean colloquial = true) {
    int dollars = (int)amount;
    int cents = ((int)(amount * 100.0F)) - dollars * 100;
    Prompt result = N[dollars];

    if (!colloquial) {
        result += P[us.wav];
    }
    result += P[dollars.wav] + P[and] + N[cents] + P[cents.wav];
    return result;
}
```

You could take advantage of this user prompt within a script to create a very simple currency generator of the US currency. This prompt defines two arguments that can be customized inside the script: `amount`, which is mandatory, as it has no default value, represents the amount to be played back; and `colloquial`, which defaults to `true`, can be used to customized the playback of the US currency as either US `dollars` or simply `dollars`. The result is a prompt concatenation that can be queued for playback directly. Inside the script, this prompt is referenced in the following ways:

```
P["currency.tpl", 3.23F]
P["currency.tpl", BankAmount, false]
    - where BankAmount is a script variable
P["currency.tpl", BankAmount, Colloquial]
P["currency.tpl", 10.0F + BankAmount]
```

If referenced without supplying the mandatory arguments, an `ExpressionNotInitializedException` exception is thrown back. If the arguments passed in are of an invalid type, an `ExpressionClassCastException` exception is thrown back. If more arguments than declared are passed in, they are simply ignored.

See [Table 3-22](#) for other examples of prompt template files.

Table 3-22 More Prompt Template File Examples

1	P[prompt.wav]
2	P[prompt.wav] + P[prompt2]

Table 3-22 More Prompt Template File Examples (continued)

3	TTS[John Doe] @ L[en_US] S[John Doe]
4	P[P1] @ TUE P[P2] @ MON

Example 3 in the preceding table represents a substitute prompt where first an attempt is made to generate the string “*John Doe*” localized as US English using the system configured TTS provider, and if that fails for any reason, the system falls back to the second prompt which spells back the string “*John Doe*”. Example 4 represents a `TimeOfDay` prompt where no prompts are played on Sundays, the user prompt `P2` is played on Mondays, and the `P1` prompt is played on all other days.

Escalating Prompt

An escalating prompt contains multiple prompts sequenced in such a way that the first time the prompt is queued up inside a step, it queues the first prompt in the escalation. If a retry occurs within the step, then the second prompt is queued up instead of the first one, and so on until the last prompt is queued up on the previous attempt at which point it is queued up for all remaining attempts.

An escalation is always reset to the first prompt when the step exits either successfully or in error.

For example, the prompt expression:

```
P[P1] || P[P2] || TTS[Hello]
```

represents an escalation prompt that plays `P[P1]` on the first attempt in a step, `P[P2]` on the next retry and `TTS[Hello]` for all other retries in that same step.

Time of Week Prompt

A time of week prompt contains multiple prompts each qualified with a particular time of the day and day of the week. When queued up for playback, a time of week prompt evaluates the current time of the week and queues up a single prompt from its list. The prompt selected is based on a time range starting at the day and time specified until the day and time specified by the subsequent prompt in time or until the end of the week if this is the last prompt. The week starts on Sunday morning.

The order of the operands is not important in determining the beginning or end of a range. The expression parser puts them back in the proper chronological order based on the specified day of week and time of day used when qualifying each one of the prompt operands.

For example, the prompt expression:

```
P[P1] @ MON @ T[10:59 AM]
|| P[P2] @ TUE @ T[11:58 PM]
|| P[] @ MON @ T[1:00 PM]
```

means that from Sunday morning to Monday 10:58:59 AM nothing is played back, from Monday 10:59:00 AM to Monday 12:59:59 PM, `P[P1]` is played back, from Monday 1:00:00 PM to Tuesday 11:57:00 PM, nothing is played back, and from Tuesday 11:58:00 PM until the end of the week, `P[P2]` is played back.

Day of Week Prompt

A day of week prompt contains multiple prompts each qualified with a particular day of the week. When queued up for playback, a day of week prompt evaluates the current day of the week and queues up a single prompt from its list. The prompt selected is based on a day range starting at the day specified until the day specified by the subsequent prompt in time or until the end of the week if this is the last prompt. The week starts on Sunday.

The order of the operands is not important in determining the beginning or end of a range. The expression parser puts them back in the proper chronological order based on the specified day of week used when qualifying each one of the prompt operands.

For example, the prompt expression:

```
P[P1] @ MON || P[] @ THU || P[P2] @ TUE
```

means that on Sunday is be played back, on Monday, P[P1] is played back, on Tuesday and Wednesday, P[P2] is played back, and the rest of the week nothing is played back.

Time of Day Prompt

A time of day prompt contains multiple prompts each qualified with a particular time of the day. When queued up for playback, a time of day prompt evaluates the current time of the day and queues up a single prompt from its list. The prompt selected is based on a time range starting at the time specified until the time specified by the subsequent prompt in time or until the end of the day if this is the last prompt.

The order of the operands is not important in determining the beginning or end of a range. The expression parser puts them back in the proper chronological order based on the specified time of day used when qualifying each one of the prompt operands.

For example, the prompt expression:

```
P[P2] @ T[11:58 PM] || P[P1] @ T[10:59 AM] || P[] @ T[1:00 PM]
```

means that from the beginning of the day until 10:58:59 AM nothing is played back, from 10:59:00 AM until 12:59:59 PM, P[P1] is played back, from 1:00:00 PM until 11:57:00PM, nothing is played back, and from 11:58:00 PM until the end of the day, P[P2] is played back.

Random Prompt

A random prompt contains multiple prompts each qualified with a given weight. When queued up for playback, a random prompt randomly chooses and queues up a single prompt from its list. The random selection is affected by the weight given to each prompt. A prompt qualified with a bigger weight has a higher chance of being selected. This type of prompt is typically used for playing back advertisements or slogans.

For example, the prompt expression:

```
P[P1] % 1 || P[P2] % 1 || P[] % 2
```

Prompt P[1] and P[2] have 25% of chances of being played back while nothing (P[]) is played back 50% of the times.

Prompt Conversions

Prompt conversion applies only to the operands of the binary + operator when one of the arguments is a Prompt. In this special case only, the other argument to the + is converted to a Prompt as described in [Table 1-7](#), and a new Prompt which is the concatenation of the two prompts is the result of the +.

The prompt concatenation operator `+`, which, when given a Prompt operand and an integral or floating-point operand, converts the integral or floating-point operand to a Prompt representing its value in spoken form, and then produces a newly created Prompt that is the concatenation of the two prompts.

There is a prompt conversion to type Prompt from every other type, including the null type as described in [Table 1-7](#). For the null type, the result is the empty prompt.

The [Prompt Concatenation Operator `+`, page 3-104](#), which, when given a Prompt operand and a reference to a char, Currency, Date, Document, `java.io.File`, `java.io.InputStream`, Language, Prompt, String, `java.net.URL`, Time or any numeral types, converts the reference to a Prompt based on [Table 1-7](#), and then produces a newly created Prompt that is the concatenation of the two prompts.

Script

Use the Script tab to reference a script from within a Cisco Unified CCXscript expression.

Script is a friendly data type that corresponds to the Java `com.cisco.script.Script` class.

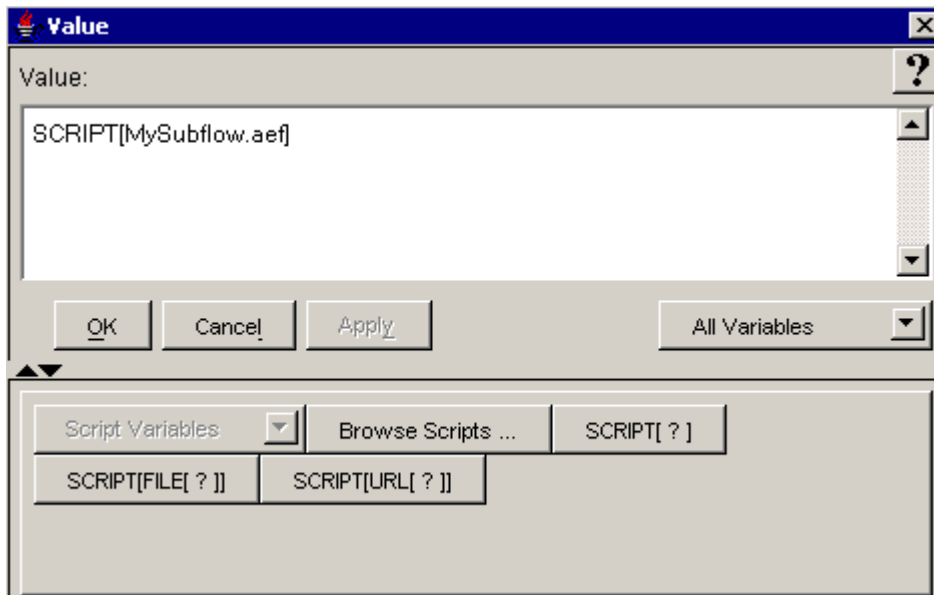
This topic includes the following:

- [About Scripts, page 3-110](#)
- [Example Simple Expression Using a Script, page 3-111](#)
- [Script Variables, page 3-111](#)
- [Browse Scripts, page 3-111](#)
- [Script tab Syntax Buttons, page 3-111](#)

About Scripts

You can reference other Cisco Unified CCX scripts from within a Cisco Unified CCX script.

Example Simple Expression Using a Script



The following section describes the options on the Script tab:

- [Script Variables, page 3-111](#)
- [Browse Scripts, page 3-111](#)
- [Script tab Syntax Buttons, page 3-111](#)

Script Variables

The Script Variables selection box lists all the script variables contained in the currently opened script. Use this selection box to paste an already defined Script variable into an expression. The default value of a script variable is null.

Browse Scripts

Use the Browse Scripts selection box to paste a script reference into your expression by browsing the local drive or the script repository.

Script tab Syntax Buttons

The Script tab syntax buttons indicate all the ways you can add a script object to an expression. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values.

Table 3-23 *Script Syntax Button Descriptions*

Syntax Button	Description
SCRIPT[?]	User script Syntax: SCRIPT[filename.aef] Example: SCRIPT[aa.aef]
SCRIPT[FILE[?]]	User script file Syntax: SCRIPT[FILE[drive:\\directorylocation\filename.aef]] Example: SCRIPT[FILE[C:\\Windows\aa.aef]]
SCRIPT[URL[?]]	User URL-based script Syntax: SCRIPT[URL[http://UrlAddress/filename.aef]] Example: SCRIPT[URL[http://localhost/aa.aef]]

Short

Use the Expression Editor Short tab to enter or modify short data in an expression. Short is a friendly data type corresponding to the fully qualified `java.lang.Short` class.

**Note**

In the Expression Language, `short` and `Short` can be used interchangeably as opposed to Java where `short` represents a primitive data type and `Short` represents an object.

This section covers the following topics:

- [About the Short Data Type, page 3-113](#)
- [Numeric Type Specification on the Web, page 3-113](#)
- [Example Short Code, page 3-113](#)
- [Short Constructors, Methods, and Attributes, page 3-115](#)
- [Short tab Syntax Buttons, page 3-115](#)

About the Short Data Type

The `java.lang.Short` numeric data type is a 16-bit Integer and its value can be from -32768 to 32767, inclusive. For type `short`, the default value is zero, that is, the value of `(short)0`.

Numeric Type Specification on the Web

For the Sun Java specification on the Short data type, see:
<http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Short.html>

Example Short Code

In the following two examples, the script variables used in the expressions are listed in the top right of each example.

Figure 3-23 Example Simple Expression Using a Short and Script Variables

Name	Type	Value
amount	String	"243697"
ivalue	short	(short)100

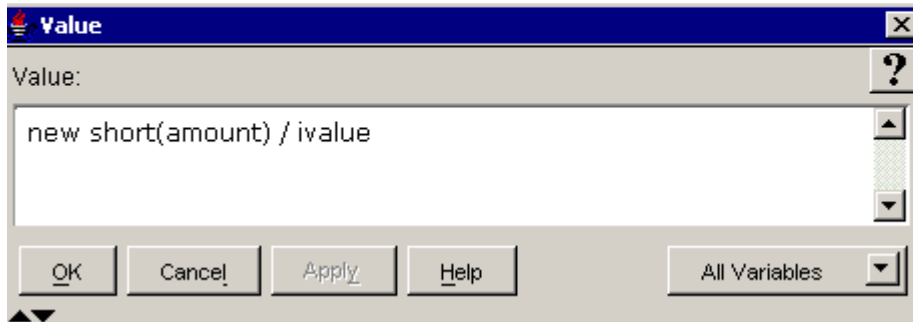
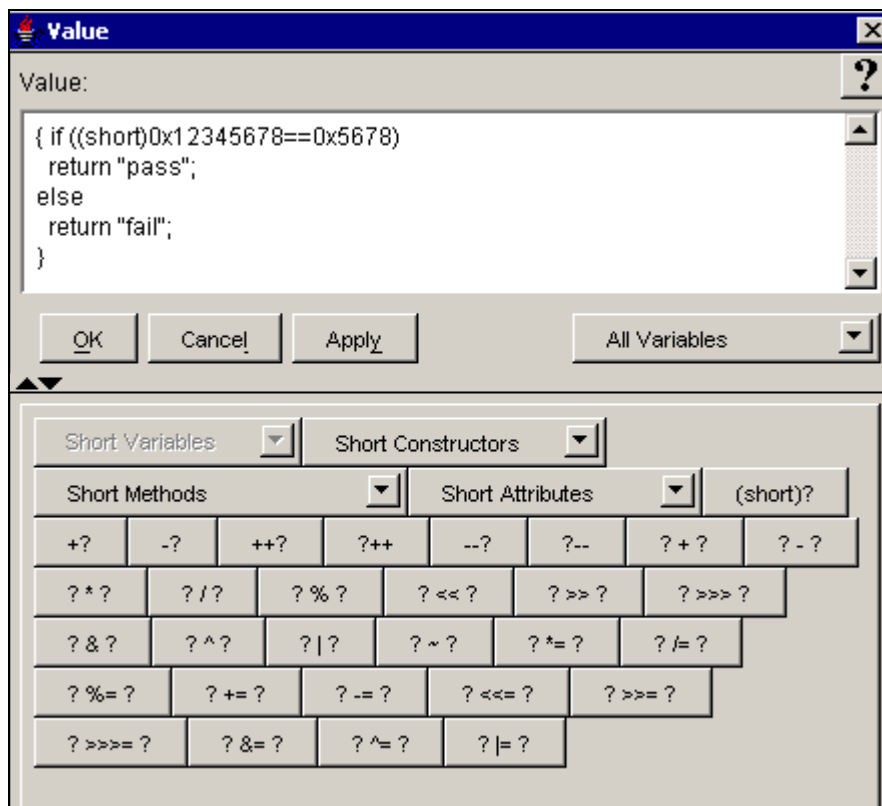


Figure 3-24 Example Complex Expression Using a Short



The following sections describe the options on the Short tab:

- [Short tab Syntax Buttons, page 3-115](#)
- [Short Variables, page 3-115](#)
- [Short Constructors, Methods, and Attributes, page 3-115](#)
- [Short tab Syntax Buttons, page 3-115](#)

Short Variables

The Short Variables selection box lists all the short variables contained in the currently opened script. Use this selection box to paste an already defined short variable into an expression.

A short variable holds the value of an short, which is a 16-bit Integer, with value range from -32768 to 32767, inclusive.

The default value of a short variable is zero, that is, the value of (short)0.

Short Constructors, Methods, and Attributes

Use the appropriate selection box to add to a short constructor, method, or attribute to your Cisco Unified CCX script expression.

The available public methods and attributes include both static and non static ones.

For descriptions of all the public Java Short constructors, methods, and attributes available in the selection boxes, see the Java specification at

<http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Short.html>.

Short tab Syntax Buttons

The Short tab syntax buttons indicate all the ways you can add or modify a short in an expression in a Cisco Unified CCX script. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values.

The semantics of arithmetic operations exactly mimic those of Java's Long arithmetic operators, as defined in The Java Language Specification. See the following for a summary descriptive list of all the operators you can use in the Java language:

<http://java.sun.com/docs/books/tutorial/java/nutsandbolts/opsummary.html>

Table 3-24 Short Syntax Button Descriptions

Syntax Button	Name	Type	Description
(short)?	short typecast	typecast	Assigns the operand a short type. This means a variable of type Integer, byte, and so on can be converted to a short type. The rule of the type casting is the same as in Java. See also Integer Literals, page 3-75 . Example: (short)3456
+	unary plus	unary	The positive value of the operand.
-	unary minus		The negative value of the operand.
++	prefix increment	increment	Increments the value of the operand by one before the operand is changed in an expression.
?++	postfix increment		Increments the value of the operand by one after the operand is changed in an expression.

Table 3-24 Short Syntax Button Descriptions (continued)

Syntax Button	Name	Type	Description
--?	prefix decrement	decrement	Decrements the value of the operand by one before the operand is changed in an expression.
?--	postfix decrement		Decrements the value of the operand by one after the operand is changed in an expression.
? + ?	addition	arithmetic	Adds two operands.
? - ?	subtraction		Subtracts the second operand from the first.
? * ?	multiplication		Multiplies two operands.
? / ?	division		Divides the first operand by the second.
? % ?	remainder		Returns the remainder of the first operand divided by the second.
? << ?	shift left	bitwise shift (for operations on individual bits in Integers only)	Shifts bits of operand 1 left by the distance of operand 2; fills with zero bits on the right-hand side.
? >> ?	shift right		Shifts bits of operand 1 right by the distance of operand 2; fills with the highest (signed) bit on the left-hand side.
? >>> ?	zero fill right shift		Shifts bits of operand 1 right by a distance of operand 2; fills with zero bits on the left-hand side.
? & ?	bitwise AND	bitwise logical (for operations on individual bits in Integers only)	Compares both operands. If both operand bits are 1, the AND function sets the resulting bit to 1; otherwise, the resulting bit is 0.
? ^ ?	bitwise exclusive OR (XOR)		Compares both operands. If both operand bits are different, the resulting bit is 1; otherwise the resulting bit is 0.
? ?	bitwise inclusive OR		Compares both operands. If either of the two operand bits is 1, the resulting bit is 1. Otherwise, the resulting bit is 0.
~ ?	Bitwise complement		Inverts the value of each operand bit: If the operand bit is 1, the resulting bit is 0; if the operand bit is 0, the resulting bit is 1.
? *= ?	multiply and assign	assignment The operand on the left of the assignment statement (the first operand) can be any type of variable, including an array component or a public class attribute.	Multiplies the first operand by the second and assigns the result to the first operand.
? /= ?	divide and assign		Divides the first operand by the second and assigns the result to the first operand.
? %= ?	remainder and assign		Divides the first operand by the second operand and assigns the remainder to the first operand.
? += ?	add and assign		Adds the first operand to the second operand and assigns the result to the first operand.
? -= ?	subtract and assign		Subtracts the second operand from the first operand and assigns the result to the first operand.

Table 3-24 Short Syntax Button Descriptions (continued)

Syntax Button	Name	Type	Description
? <<= ?	left shift and assign	Assignment (continued)	Shifts bits of operand 1 left by the distance of operand 2; fills with zero bits on the right-hand side and assigns the resulting bit to operand 1.
? >>= ?	right shift and assign		Shifts bits of operand 1 left by the distance of operand 2; fills with zero bits on the right-hand side and assigns the resulting bit to operand 1.
? >>>= ?	zero fill, right shift, and assign		Shifts bits of operand 1 right by a distance of operand 2; fills with zero bits on the left-hand side and assigns the resulting bit to operand 1.
? &= ?	AND and assign		First, compares both operands. If both operand bits are 1, the AND function sets the resulting bit to 1; otherwise, the resulting bit is set to 0. Then, assigns the resulting bit to operand 1.
? ^= ?	XOR and assign		First, compares both operands. If both operand bits are different, the resulting bit is 1; otherwise the resulting bit is 0. Then, assigns the resulting bit to operand 1.
? = ?	OR and assign		First, compares both operands. If either of the two operand bits is 1, the resulting bit is 1. Otherwise, the resulting bit is 0. Then, assigns the resulting bit to operand 1.

String

Use the String tab to enter and modify strings in an expression. String is a friendly data type corresponding to the fully qualified `java.lang.String` class name.

This topic includes the following:

- [About the String Class, page 3-118](#)
- [Java String Specification on the Web, page 3-118](#)
- [Example Simple Expression Using a String, page 3-118](#)
- [String Variables, page 3-119](#)
- [String Constructors, Methods, and Attributes, page 3-119](#)
- [String tab Syntax Buttons, page 3-119](#)
- [String Literals, page 3-120](#)
- [Escape Sequences for Character and String Literals, page 3-121](#)
- [An Array of Characters is Not a String, page 3-121](#)

See also:

- [String Concatenation Operator +, page 1-10](#)
- [String Conversions, page 1-32](#)

- [String Parsing](#), page 1-33

About the String Class

Instances of class String represent sequences of Unicode characters. Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language. For a list of all the language Unicode charts, see <http://www.unicode.org/charts/>

Using [Google.com](http://www.google.com), you should be able to find tutorials on Unicode characters.

As in Java, the class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase.

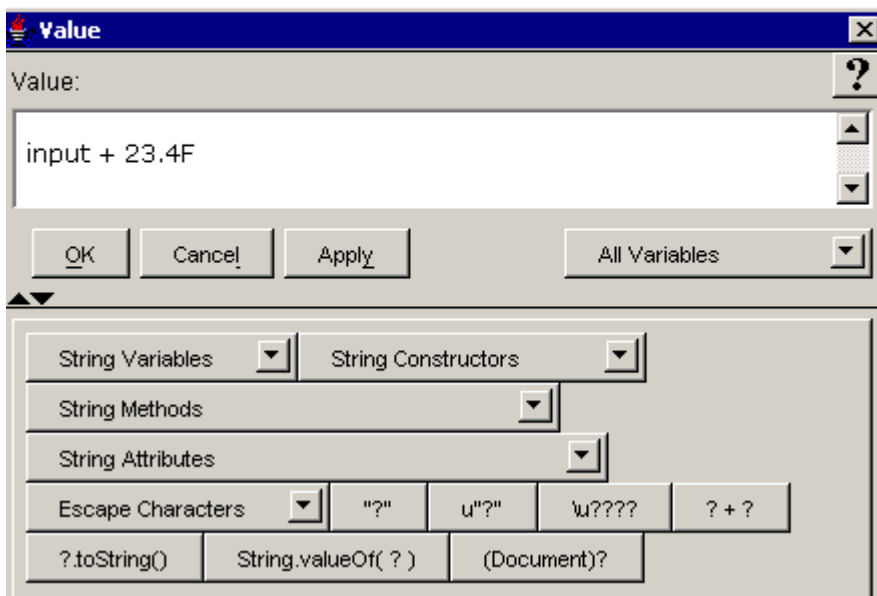
The string concatenation operator + implicitly creates a new String object and a String object has a constant (unchanging) value. Once they are created, they cannot be changed.

Java String Specification on the Web

The Cisco Unified CCX Expression Language uses strings in the same way Java uses them. For the Sun Java specification on strings, see <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/String.html>.

Example Simple Expression Using a String

Name	Type	Value
input	String	"12 or 34"



The following sections describe the options on the String tab:

- [String Variables, page 3-119](#)
- [String Constructors, Methods, and Attributes, page 3-119](#)
- [String tab Syntax Buttons, page 3-119](#)

See also:

- [String Conversions, page 1-32](#)
- [String Parsing, page 1-33.](#)

String Variables

The String Variable selection box lists all the string variables contained in the currently opened script. Use this selection box to paste a String variable into an expression.

A String variable has a constant (unchanging) value and the string concatenation operator + implicitly creates a new String variable. A String variable consists of the set of Unicode characters from “\u0000” to “\uffff” inclusive.

The default value of a String variable is the empty string, that is, "".

String Constructors, Methods, and Attributes

Use the appropriate selection box to add a String constructor, method, or attribute to your expression.

The available public methods and attributes include both static and non static ones.

For descriptions of the public Java String constructors, methods, and attributes available in the selection boxes, see <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/String.html>.

See also:

- [Character tab Syntax Buttons, page 3-35.](#)

String tab Syntax Buttons

The String tab syntax buttons indicate all the ways you can add or use a String in an expression. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values.

Table 3-25 *String Syntax Button Descriptions*

Syntax Button	Type	Description
"?"	string literal	A string. See String Literals, page 3-120
u"?"	extended string literal	An extended string. The Unicode character codes for the characters from 128 and higher. Since ASCII is a seven-bit code and most computers manipulate data in eight-bit bytes, many extensions use the additional Unicode codes available by using all eight bits of each byte.
\u????	Unicode literal	A Unicode character.

Table 3-25 String Syntax Button Descriptions (continued)

Syntax Button	Type	Description
? + ?	string concatenation	Concatenates two strings into one. See also String Concatenation Operator + , page 1-10
?.toString()	cast	Returns the string representation of the specified object.
String.valueOf(?)	cast	Returns the string representation of the specified object. If the object is null, it returns “null”.
(Document)?	cast	Converts a string into a text document.

String Literals

A string literal consists of zero or more characters enclosed in double quotes. For backward compatibility with the original release of Cisco Unified CCX software, we must maintain the first string literal format which did not have any support for escape sequence. As such, string literals since Cisco CRS 3.0 can be represented in two ways. The first one using one pair of double quotes characters has no support for escape sequence so the \ character can be used to represent itself. The second format introduced in Cisco CRS 3.0 uses two double quote characters where the first one is represented by the lowercase letter `u` where each character may be represented by an escape sequence.

A string literal is always of type `String`. A string literal always refers to the same instance of class `String`.

```
StringLiteral:
    " NoEscapeStringCharactersopt "
    u " StringCharactersopt "

NoEscapeStringCharacters:
    NoEscapeStringCharacter
    NoEscapeStringCharacters NoEscapeStringCharacter

NoEscapeStringCharacter:
    InputCharacter but not "

StringCharacters:
    StringCharacter
    StringCharacters StringCharacter

StringCharacter:
    UnicodeInputCharacter but not " or \
    EscapeSequence
```

The escape sequences are described in [Escape Sequences for Character and String Literals](#), page 3-121.

Neither of the characters CR and LF is ever considered to be an `InputCharacter`; each is recognized as constituting a `LineTerminator`.

You will receive a parse-time error if a line terminator appears after the opening `"` and before the closing matching `"`. A long string literal can always be broken up into shorter pieces and written as a (possibly parenthesized) expression using the string concatenation operator `+`.

Example string literals:

```
" " // the empty string
"\ " // a string containing the \ character alone
u"" // a string containing " alone
"This is a string" // a string containing 16 characters
"This is a " + // a string-valued constant expression,
"two-line string" // formed from two string literals
```

Each string literal is a reference to an instance of class `String`. `String` objects have a constant value.

Escape Sequences for Character and String Literals

The character and string escape sequences allow for the representation of some non-graphic characters as well as the single quote, double quote, and backslash characters in character literals and string literals.

```
Escape Sequences:
\ b /* \u0008: backspace BS */
\ t /* \u0009: horizontal tab HT */
\ n /* \u000a: linefeed LF */
\ f /* \u000c: form feed FF */
\ r /* \u000d: carriage return CR */
\ " /* \u0022: double quote " */
\ ' /* \u0027: single quote ' */
\ \ /* \u005c: backslash \ */
\ 0 /* \u0000: null character */
\ UnicodeInputCharacter/* the actual Unicode character */
```

If the character following a backslash in an escape is not an ASCII `b`, `t`, `n`, `f`, `r`, `"`, `'`, `\`, or `0`, then the escape sequence is replaced with the actual character and the backslash is simply omitted.

An Array of Characters is Not a String

In the Expression Framework language as in the Java programming language, unlike C, an array of `char` is not a `String`, and neither a `String` nor an array of `char` is terminated by `'\u0000'` (the NULL character).

A `String` object is immutable, that is, its contents never change, while an array of `char` has mutable elements. The method `toCharArray` in class `String` returns an array of characters containing the same character sequence as a `String`. The class `StringBuffer` implements useful methods on mutable arrays of characters.

Time

Use the Time tab to enter or modify time data in an expression. Time is a friendly data type corresponding to the fully qualified `java.sql.Time` class name.

The Expression Editor formats the date and time according to the default locale.

This topic includes the following:

- [About Time Data, page 3-122](#)
- [Time Specification on the Web, page 3-122](#)
- [Example Simple Expression using Time and Three Script Variables, page 3-123](#)
- [Time Constructors and Methods, page 3-123](#)
- [Time tab Syntax Buttons, page 3-124](#)
- [Time Literals, page 3-124](#)

About Time Data

The Time class deals with hours, minutes, and seconds. When you create a new Time object, you must pass it the hour, minute, and second. The Time class uses the ISO hh-mm-ss format. Hours are represented by numbers between 0 and 23. Minutes and seconds are represented by numbers between 0 and 59. Use the `Time.valueOf(String)` method to convert an “hh-mm-ss” string to a Time object and use the `?.toString()` method to return a Time object to its string (hh-mm-ss) representation.

The Time class extends the Date class. Both the Time and Date class use most of the same methods. For a description of Time literals, see [Time Literals, page 3-124](#).

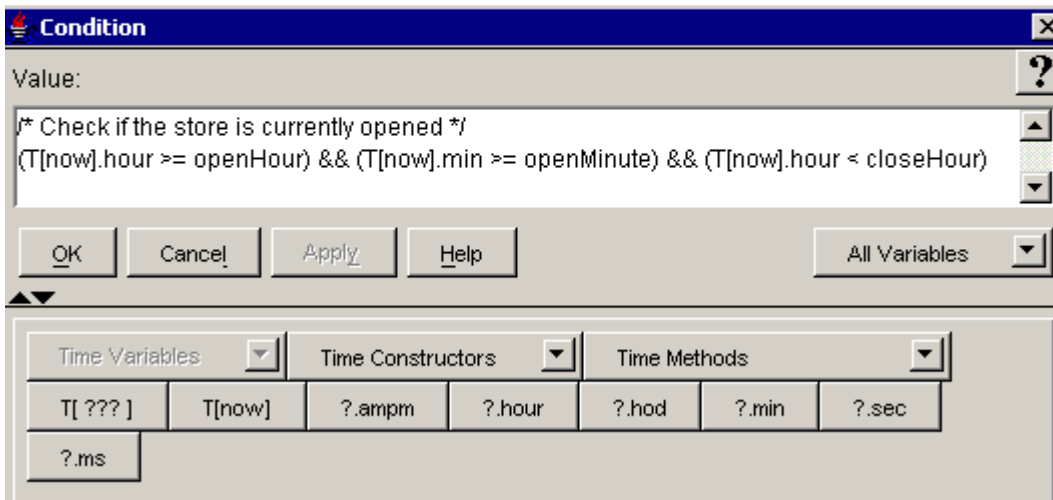
Time Specification on the Web

For the Sun Java specification on the Time class, see <http://java.sun.com/j2se/1.4.2/docs/api/java/sql/Time.html>.

For a list of world time zones, see <http://www.worldtimezone.com>.

Example Simple Expression using Time and Three Script Variables

Name	Type	Value
closeHour	int	8
openHour	int	8
openMinute	int	30



The following sections describe the options on the Time tab:

- [Time Variables, page 3-123](#)
- [Time Constructors and Methods, page 3-123](#)
- [Time tab Syntax Buttons, page 3-124](#)

Time Variables

The Time Variables selection box lists all the time variables contained in the currently opened script. Use this selection box to paste a time variable into an expression.

A Time variable contains time information. The default value of the Time variable is the current time at the time of interpretation.

Time Constructors and Methods

Use the appropriate selection box to add a public Time constructor or method to your expression.

The available public methods include both static and non static ones.

For descriptions of the public Java Time constructors and methods available in the selection boxes, see <http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigDecimal.html>.

Time tab Syntax Buttons

The Time tab syntax buttons indicate all the ways you can add a Time object to an expression. Clicking on one of the buttons adds the indicated syntax (minus the question marks) to your expression. In the spaces left by the question marks, enter the appropriate values.

Table 3-26 Time Syntax Button Descriptions

Syntax Button	Description
T[???	Returns the time in the format HH:MM:SS AM PM. For example: T[3:39 AM] T[3:34:42 PM] T[11:59:58 PM EST]
T[now]	Returns the current time in the format HH:MM:SS AM PM. For example, returns: T[3:34:42 PM].
? .ampm ¹	Returns an int number of the date object; AM=0, PM=1
? .hour ¹	Returns the hour represented by this Date object. The returned value is a number (0 through 23) representing the hour within the day that contains or begins with the instant in time represented by this Date object, as interpreted in the local time zone.
? .hod ¹	Returns the hour represented by this Date object. The returned value is a number (0 through 23) representing the hour within the day that contains or begins with the instant in time represented by this Date object, as interpreted in the local time zone.
? .min ¹	Returns the number of minutes past the hour represented by this date, as interpreted in the local time zone. The value returned is between 0 and 59.
? .sec ¹	Returns the number of seconds past the minute represented by this date. The value returned is between 0 and 61. The values 60 and 61 can only occur on those Java Virtual Machines that take leap seconds into account.
? .ms ¹	Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.

1. The .ampm, .hour, .hod, .min, .sec, and .ms variables do not require the Java license.

Time Literals

The time literal is always of type `Time`.

```
TimeLiteral:
    TimeDeclarator [n o w]
    TimeDeclarator [TimeDesignator]
TimeDeclarator: one of
    t T
```

```
TimeDesignator:
    FullTimePattern
    LongTimePattern
    MediumTimePattern
    ShortTimePattern
```


FullTimePattern:
Defined by the pattern "h:mm:ss a z"

LongTimePattern:
Defined by the pattern "h:mm:ss a z"

MediumTimePattern:
Defined by the pattern "h:mm:ss a"

ShortTimePattern:
Defined by the pattern "h:mm a"

If the string `now` is used, then the literal corresponds to the current time in the server's default time zone when the literal is first evaluated. See the documentation of the `java.text.DateFormat` class for descriptions of the available time patterns.

Example Time Literals:

```
T[5:59 PM]
T[12:23:59 AM]
T[12:23:59 AM CST]
T[now]
```

Each time literal is a reference to an instance of class `java.sql.Time`.

User

Use the User tab to enter a user literal directly into a Cisco Unified CCX script expression.

The User class is specific to the Expression Language and is defined at `com.cisco.user.User`

This topic includes the following:

- [About Users, page 3-125](#)
- [Example User Code, page 3-126](#)
- [User Variables, page 3-126](#)
- [User Syntax Button, page 3-126](#)

About Users

A user object identifies and represents anyone configured in the Cisco Unified CallManager. That someone could be an agent, a supervisor, and administrator, or anyone configured in the Cisco Unified CallManager.

You specify a user object in a user variable that you can create in a Cisco Unified CCX script. For example, in an Unified CCX application, when a requested agent is available, the Select Resource step can return a user object that is the requested agent. The Connect step can then pass the user object as an argument to connect the call to the selected agent.

Another example application that can have user objects is the Cisco Auto Attendant. For example, An Auto Attendant application, might ask you the name or extension of the person to whom you want to speak so you can be transferred to that person.

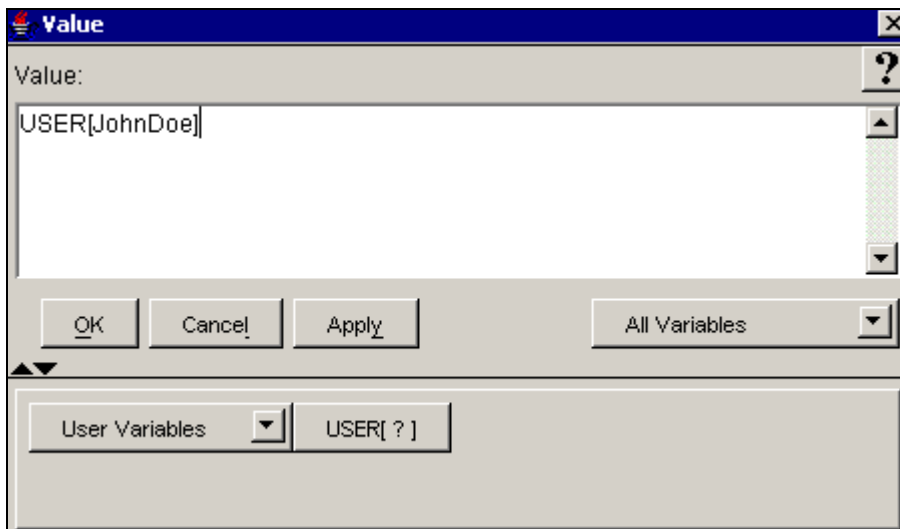
The Name to User step and the Get User step the step are two example steps that can return a user object as in the Auto Attendant application. The Name to User step prompts for either the spoken or spelled name of a user. The Get User step specifies the extension of a user so it will return that particular user to a caller.

See also *Cisco Unified Contact Center Express Scripting and Development Series: Volume 2, Editor Step Reference* and the chapter on creating IP IVR scripts in *Cisco Unified Contact Center Express Scripting and Development Series: Volume 1, Getting Started with Scripts*.

To play back the spoken name of a user:

- You can replace the Get User and the Get User Info steps with one step by using a USER[userID] variable. Then, when a user is requested in the Get User Info step, instead of selecting a “generic” variable of type user, you can directly enter the specific user variable that holds the USER[userID] you want.

Example User Code



The following sections describe the options on the User tab:

- [User Variables, page 3-126](#)
- [User Syntax Button, page 3-126](#)

User Variables

The User Variables selection box lists all the user variables contained in the currently opened script.

User Syntax Button

Use the USER[?] button to specify a user ID in an expression. For example: USER[JohnSmith].



INDEX

A

additive operators

- about [1-9](#)
- document concatenation operator + [1-10, 3-51](#)
- prompt concatenation operator + [1-10, 3-104](#)
- string concatenation operator + [1-10](#)

array

- about [3-8](#)
- components [1-28](#)
- example code [3-9](#)
- Expression Editor tab [3-8](#)
- methods [3-11](#)
- of characters [3-121](#)
- specification on web [3-8](#)
- syntax [3-11](#)
- variables [3-10](#)

assignment operators

- compound assignment operators [1-13](#)

B

BigDecimal

- about [3-13](#)
- description [1-4](#)
- example code [3-14](#)
- Expression Editor tab [3-13](#)
- specification [3-13](#)
- syntax [3-16](#)
- variables [3-15](#)

BigInteger

- about [3-18](#)
- description [1-4](#)

example code [3-18](#)

Expression Editor tab [3-18](#)

specification [3-18](#)

syntax [3-20](#)

variables [3-20](#)

Binary Document Literal [3-49, 3-52](#)

Boolean

- about [3-23](#)
- example code [3-23](#)
- Expression Editor tab [3-23](#)
- literals [3-27](#)
- specification [3-23](#)
- syntax [3-25](#)
- type [1-21, 1-23](#)
- values [1-21, 1-23](#)
- variables [3-24](#)

Byte

- about [3-27](#)
- example code [3-29](#)
- Expression Editor tab [3-27](#)
- specification [3-28](#)
- syntax [3-29](#)
- variables [3-29](#)

C

Character

- about [3-32](#)
- escape [3-36](#)
- example code [3-34](#)
- Expression Editor tab [3-32](#)
- literals [3-35](#)
- specification [3-33](#)

- syntax [3-35](#)
 - variables [3-34](#)
- class
 - currency [1-25](#)
 - date [1-25](#)
 - definition [1-4](#)
 - document [1-25](#)
 - grammar [1-26](#)
 - language [1-26](#)
 - object [1-25](#)
 - prompt [1-26](#)
 - string [1-26](#)
 - time [1-26](#)
- comments
 - definition [1-4](#)
- complex
 - expression block
 - parameter [1-28](#)
- compound assignment operators [1-13](#)
- constructor
 - definition [1-4](#)
- conversions
 - document [1-32](#)
 - string [1-32](#)
 - string parsing [1-33](#)
- Currency
 - about [3-37](#)
 - example code [3-38](#)
 - Expression Editor tab [3-37](#)
 - literal [3-39](#)
 - prompt literals [3-100](#)
 - recent [3-38](#)
 - specification [3-37](#)
 - syntax [3-39](#)
 - variables [3-38](#)
- custom Java class
 - how make available to editor [3-80](#)

D

- Date
 - about [3-39](#)
 - example code [3-40](#)
 - Expression Editor tab [3-39](#)
 - literals [3-43](#)
 - specification [3-40](#)
 - syntax [3-42](#)
 - variables [3-41](#)
- DayOfWeek [3-93](#)
 - document [3-53](#)
- day of week
 - prompt [3-109](#)
- definite assignment [1-30](#)
- delay prompt literals [3-104](#)
- digit grammar literals [3-66](#)
- Document
 - about [3-44](#)
 - conversions [1-32](#)
 - day of week [3-53](#)
 - escalation operator || [1-12](#)
 - example code [3-45](#)
 - Expression Editor tab [3-44](#)
 - literals [3-48](#)
 - file [3-50](#)
 - text [3-50](#)
 - URL [3-49](#)
 - user [3-51](#)
 - qualificator operator @ [3-52](#)
 - syntax [3-46](#)
 - time of day [3-53](#)
 - time of week [3-52](#)
 - variables [3-45](#)
- double
 - about [3-54](#)
 - example code [3-55](#)
 - Expression Editor tab [3-54](#)
 - specification [3-54](#)

syntax [3-56](#)
 variables [3-56](#)

E

escalating prompt [3-108](#)
 escalation operator `||`
 about [1-11](#)
 document escalation [1-12](#)
 prompt escalation [3-105](#)
 escape characters [3-36](#)
 escape sequences
 for literals [3-121](#)
 exception-handler parameter [1-28](#)
 Expression Editor
 All Variables selection box [2-3](#)
 pop-up menu [2-7](#)
 tabbed toolbar [2-7](#)
 using [2-2](#)
 Expression Language
 description [1-1](#)
 operator summary [1-7](#)
 expressions
 licensing [2-9](#)

F

file document literals [3-50](#)
 final
 variable modifier [3-94](#)
 variables [1-29](#)
 float
 about [3-57](#)
 example code [3-58](#)
 Expression Editor tab [3-57](#)
 specification [3-58](#)
 variables [3-59](#)
 floating-point
 numbers [1-4](#)

operations [1-21](#)
 types, formats, and values [1-20](#)

G

gender [3-93](#)
 grammar
 about [3-62](#)
 example code [3-63](#)
 syntax [3-64](#)
 templates [3-69](#)
 variables [3-64](#)
 GSL grammar literals [3-67](#)

I

identifiers
 description [1-4](#)
 index variables [3-10](#)
 Integer (int)
 about [3-69](#)
 class [3-70](#)
 example code [3-19, 3-41, 3-71, 3-77, 3-88, 3-114](#)
 operations [3-72](#)
 specification [3-70](#)
 syntax [3-72](#)
 variables [3-71](#)
 integer (int)
 definition [1-5](#)
 Expression Editor tab [3-69](#)
 operations [1-19](#)
 integer/Boolean conditional-or operator `||` [1-10](#)
 integral types and values [1-18](#)

J

Java
 example code [3-77](#)
 Expression Editor tab [3-77](#)

how to use [3-77](#)
 licensing [2-9](#)
 specification [3-77](#)
 syntax buttons [3-80](#)

Java classes

custom classes [3-80](#)
 how to access any Java class [3-79](#)

K

keywords

definition [1-5](#)

L

language

about [3-84](#)
 all [3-85](#)
 example code [3-85](#)
 Expression Editor tab [3-84](#)
 recent [3-86](#)
 specifications [3-84](#)
 syntax [3-86](#)
 variables [3-85](#)

licensing, expressions [2-9](#)

literals

and escape sequences [3-121](#)
 Boolean [3-27](#)
 character [3-35](#)
 currency [3-39](#)
 currency prompt [3-100](#)
 date [3-43](#)
 definition [1-5](#)
 delay prompt [3-104](#)
 digit grammar [3-66](#)
 floating-point [3-61](#)
 GSL grammar [3-67](#)
 integer [3-75](#)
 null [3-93](#)

number prompt [3-101](#)
 ordinal prompt [3-100](#)
 prompt [3-98](#)
 spelling prompt [3-102](#)
 SRGS grammar [3-68](#)
 string [3-120](#)
 time [3-124](#)
 TTS prompt [3-103](#)
 user grammar [3-66](#)
 user prompt [3-99](#)

local variables [1-28](#)

long

about [3-87](#)
 example code [3-87](#)
 Expression Editor tab [3-87](#)
 specification [3-87](#)
 syntax [3-89](#)
 variables [3-89](#)

M

method

definition [1-5](#)

Miscellaneous tab

about [3-91](#)
 example code [3-92](#)
 tool tip [3-5](#)

modulus

definition [1-5](#)

N

null literal [3-93](#)

number prompt literals [3-101](#)

O

objects

class [1-25](#)

definition [1-5](#)
 operator
 definition [1-5](#)
 ordinal prompt literals [3-100](#)

P

package
 definition [1-5](#)
 primitive
 types and values [1-18](#)
 prompt
 about [3-94](#)
 day of week [3-109](#)
 escalating [3-108](#)
 escalation operator || [3-105](#)
 example code [3-95](#)
 Expression Editor tab [3-94](#)
 literals [3-98](#)
 qualificator operator @ [3-105](#)
 qualificator operator % [3-105](#)
 qualificator operators [3-105](#)
 random [3-109](#)
 spelling literals [3-102](#)
 substitution operator ||| [1-9, 3-104](#)
 syntax [3-96](#)
 templates [3-106](#)
 time of day [3-109](#)
 time of week [3-108](#)
 variables [3-95](#)
 public
 definition [1-5](#)

Q

qualificator operators [1-9](#)
 document [3-52](#)
 document qualificator operator @ [3-52](#)
 prompt qualification [3-105](#)

prompt qualificator operator @ [3-105](#)
 prompt qualificator operator % [3-105](#)

R

random prompt [3-109](#)
 remainder
 definition [1-5](#)
 removing or showing
 Expression Editor toolbar [2-8](#)

S

script
 example code [3-111](#)
 Expression Editor tab [3-110](#)
 syntax [3-111](#)
 variables [1-28, 3-111](#)
 separators
 definition [1-6](#)
 Short
 about [3-113](#)
 example code [3-113](#)
 Expression Editor tab [3-113](#)
 specification [3-113](#)
 syntax [3-115](#)
 variables [3-115](#)
 spelling prompt literals [3-102](#)
 SRGS grammar literals [3-68](#)
 String
 about [3-117](#)
 concatenation operator + [1-10](#)
 conversion [1-32](#)
 example code [3-118](#)
 Expression Editor tab [3-117](#)
 literals [3-120](#)
 parsing [1-33](#)
 specification [3-118](#)
 syntax [3-119](#)

variables selection box [3-119](#)

prompt literals [3-99](#)

T

text document literals [3-50](#)

thread

definition [1-6](#)

time

about [3-122](#)

example code [3-123](#)

Expression Editor tab [3-122](#)

literals [3-124](#)

specification [3-122](#)

syntax [3-124](#)

variables [3-123](#)

time of day

document [3-53](#)

prompt [3-109](#)

time of week

document [3-52](#)

prompt [3-108](#)

tokens

definition [1-6](#)

TTS prompt literals [3-103](#)

types

and classes and interfaces [1-30](#)

kinds [1-17](#)

where used [1-27](#)

U

Unicode

definition [1-6](#)

URL document literals [3-49](#)

User

Expression Editor tab [3-125](#)

user

document literals [3-51](#)

grammar literals [3-66](#)

V

values [1-17](#)

variables [3-95](#)

about [1-27](#)

array [3-10](#)

array components [1-28](#)

BigDecimal [3-15](#)

BigInteger [3-20](#)

Boolean [1-21, 1-23, 3-24](#)

byte [3-29](#)

character [3-34](#)

complex expression block parameter [1-28](#)

Currency [3-38](#)

Date [3-41](#)

definite assignment [1-30](#)

definition [1-6](#)

Document [3-45](#)

double [3-56](#)

exception-handler parameter [1-28](#)

final [1-29](#)

float [3-59](#)

Grammar selection box [3-64](#)

how to quickly access [3-92](#)

index [3-10](#)

initial values [1-29](#)

integer (int) [3-71](#)

integral [1-18](#)

kinds [1-28](#)

language [3-85](#)

local [1-28](#)

long [3-89](#)

primitive [1-18, 1-28](#)

Prompt selection box [3-95](#)

reference [1-24, 1-28](#)

script [1-28](#)

Short [3-115](#)

String [3-119](#)
Time [3-123](#)
types [1-17](#)
values [1-17](#)
where types used [1-27](#)

W

white space
definition [1-6](#)

