Oracle
Primavera Cloud Expression Language Guide

April 2024

Contents

About this Guide

This guide describes how to use Primavera Cloud Expression Language, a scripting language within Primavera Cloud. Primavera Cloud Expression Language provides a variety methods you can use to generate data for configured fields and measures based on the contents of objects and fields in Primavera Cloud. Application administrators and other users responsible for configuring measures and configured fields should read this guide.

Primavera Cloud Expression Language Overview

Primavera Cloud Expression Language is a scripting language used to define formulas in Oracle Primavera Cloud. Formulas are scripts that can be run within the application to compute and return values. You can write formulas using Primavera Cloud Expression Language to set values for measures and configured fields based on other data within the application. For example, you can define a formula type configured field that uses a Primavera Cloud Expression Language script to calculate and return a portfolio's Net Present Value based on its current approved budget.

The practical set of language features in Primavera Cloud Expression Language ensure custom fields and measures remain flexible. For example, Primavera Cloud Expression Language provides semantic and syntactic constructs that support the following:

- Mathematical Operations
- Variable Assignment
- Date and Duration Calculation
- Field Value Referencing

The following sections introduce Primavera Cloud Expression Language and describe how to begin writing measure and configured field formulas to address your business needs. For additional information on the Primavera Cloud Expression Language, including supported types, methods, and operators, see the ***Language Reference*** (on page 17).

Defining Formulas

Use Primavera Cloud Expression Language to define Oracle Primavera Cloud configured field or measure formulas. Formulas are a sequence of Primavera Cloud Expression Language statements and expressions, called scripts, that programmatically determine the values of configured fields and portfolio and strategy measures. Only formulas containing well-formed Primavera Cloud Expression Language scripts are valid. A Primavera Cloud Expression Language script is well-formed if it conforms to Primavera Cloud Expression Language syntax rules. To learn more about Primavera Cloud Expression Language syntax, see the ***Language Reference*** (on page 17).

The following objects support formula based measures or configured fields:

- Activity

- Budget
- Budget Changes
- Budget Details
- Budget Transfers
- Budget Details
- Changes
- Change Estimates
- Change Order
- Commitment
- Contract
- Custom Logs
- Fund
- Payment Application
- Portfolio
- Potential Change Order
- Program Budget
- Program Budget Changes
- Program Budget Details
- Program Budget Transfers
- Project
- Project Actuals
- Project Cost Sheet
- Resource Assignment
- RFI
- Risk
- Scope Assignment
- Scope Item
- Strategy
- Submittal
- Tasks
- WBS
- Work Package
- Workspace Cost Sheet

Example Primavera Cloud Expression Language Script Formula

The following example contains a formula that returns a string indicating whether project delays are likely based on a project's current **Percent Complete** value and the number of days until the project's **Planned Finish Date**.

```
def currentDate = new Date();
def weekFromFinish = minusDays(object.Project_planEndDate, 7);
if(object.Project_percentComplete < 95 && currentDate >= weekFromFinish){
      return "Project delay likely";
} else {
      return "No delays anticipated.";
}
```

This example showcases the basic structure of a formula defined using Primavera Cloud Expression Language. The first and second lines demonstrates variable assignment:

```
def currentDate = new Date();
def weekFromFinish = minusDays(object.Project_planEndDate, 7);
```

Variables are declared using the `def` keyword and are assigned values using the = operator.

Note: Primavera Cloud Expression Language variables must maintain the same type as the value they are assigned in their initial declaration and can only be reassigned values of the same type. If you attempt to reassign a declared variable a value of a different type, the script will fail to validate. For more information, see the ***Language Reference*** (on page 17).

Lines three to seven of the script illustrate an conditional `if` statement written in Primavera Cloud Expression Language:

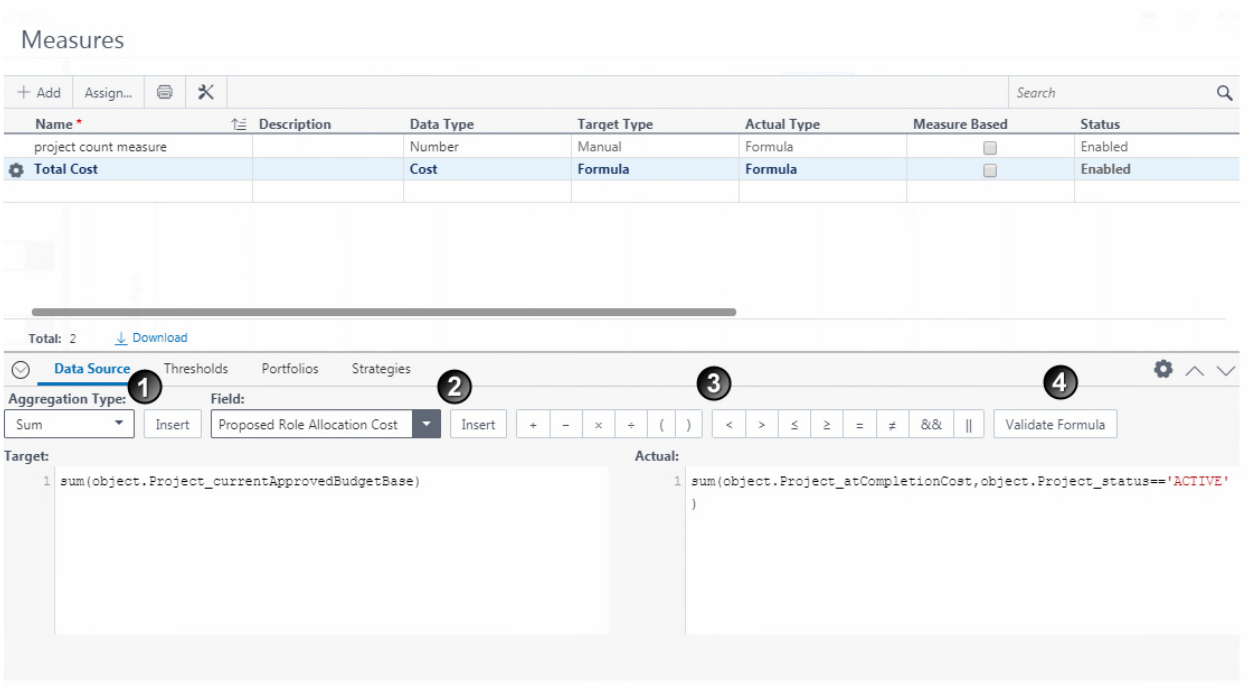
```
if(object.Project_percentComplete < 95 && currentDate >= weekFromFinish){
      return "Project delay likely";
} else {
      return "No delays anticipated.";
}
```

The conditional statement is organized in blocks indicated by opening and closing brackets. As in many other programming languages, if the expression contained in the `if` statement evaluates to true, the first block of the conditional is evaluated. If the expression evaluates to false, the `else` block is evaluated.

In general, Primavera Cloud Expression Language closely follows many of the same syntactic rules and conventions as the Java programming language. To learn more about the types, operators, and methods included in Primavera Cloud Expression Language, see the ***Language Reference*** (on page 17).

Working with the Formula Editor

Primavera Cloud includes a formula editor designed to help you create valid Primavera Cloud Expression Language scripts. The formula editor provides a field selector, script validation, and operator buttons to help ensure your scripts follow Primavera Cloud Expression Language syntax rules. You can use the formula editor when defining configured fields or measures that support formulas.



Name *	Description	Data Type	Target Type	Actual Type	Measure Based	Status
project count measure		Number	Manual	Formula	☐	Enabled
Total Cost		Cost	Formula	Formula	☐	Enabled

Target:

```
sum(object.Project_currentApprovedBudgetBase)
```

Actual:

```
sum(object.Project_atCompletionCost,object.Project_status=='ACTIVE'
)
```

Table of Screen Highlights

Item	Description
1	**Aggregation Type**: Provides a template for formulas that use Sum, Min, Max, Count, and Average methods. To use a template, select the Aggregation Type, and then select Insert. For example, the Sum template inserts the following code snippet: **sum(< Field >,< Filter >)** Replace <Field> with the name of the field to be summed. Optionally, replace <Filter> with a n expression that returns a Boolean value to filter results. Add any additional filter requirements using the provided mathematical operators.

Item	Description
2	**Field**: Provides all the fields that can be used in formulas. If Measure Based is selected, only existing measures display. Select Referenced Measures in the Field list to view measures that are currently assigned to Portfolios or Strategies. Select a field name, and select **Insert Field**. The proper formatting for the field displays. If you are defining a configured field formula, after selecting a field, insert a "." after the field name and press **Ctrl** + **Spacebar** to view a list of available field values. Select a value from the **Field Values** list to enter it into your formula. For example, enter **object.Activity_activityStatus.** and press **Ctrl** + **Spacebar** to view a list of Activity Status field values.
3	**Mathematical Symbols**: Provides access to Primavera Cloud Expression Language supported operators. Use operators in your formulas to calculate values and form complex expressions.
4	**Validate Formula**: Verifies the formula works and is well formed. The formula editor indicates syntax errors upon validation. Formulas are also validated when you save configured field or measure settings.

For more information on the formula editor, refer to the *Oracle Primavera Cloud Help*.

Writing a Primavera Cloud Expression Language Script

Write Primavera Cloud Expression Language scripts to dynamically determine the values of configured fields and measures that support formulas. A formula-based field defaults to a read-only calculated value. It displays the value resulting from the evaluation of the Primavera Cloud Expression Language script that defines the formula. The expected return type of the formula field script must be compatible with the configured field or measure type (Number, Date, or Text).

The following example shows how to calculate the Net Present Value (NPV) of a portfolio using a formula.

To define a formula that calculates Net Present Value:

1) In the object selector, select a workspace.
2) In the sidebar, select ⚙ **Summary & Settings**.
3) On the **Summary & Settings** menu, select **Defaults & Options**, and then select **Portfolio**.
4) On the **Portfolio** page, select the **Configured Fields** tab.
5) In the table, select + **Add**.
    a. In the **Column Label** column, enter **Net Present Value**.
    b. In the **View Column Name** column, enter **NPV**.
    c. In the **Data Type** column, select **Number**.
    d. In the **Type** column, select **Formula**.

6) In the **Formula** detail window, in the **Formula Editor**, enter the following:

```
//Assume a rate of return of 10%
//Calculate NPV for a 2 year period.
def npv = 0;
def i = 0.10;
def year = 0;
if (object.Portfolio_currentApprovedBudget <= 0) {
    return 0;
} else {
      npv = npv + ((-1 * object.Portfolio_currentApprovedBudget) /
Math.pow((1 + i), year));
     year = year + 1;
     npv = npv + ((-1 * object.Portfolio_currentApprovedBudget) /
Math.pow((1 + i), year));
      year = year + 1;
      npv = npv + ((-1 * object.Portfolio_currentApprovedBudget) /
Math.pow((1 + i), year));
}
return npv;
```

Note: Because Primavera Cloud Expression Language does not support loops or function definition, you must write calculations that require recursion or iteration in an imperative style as shown in the previous example.

Note: When adding a null check to a formula, you must specify the null check at the beginning of the if condition statement.

7) In the **Formula** detail window, select **Validate Formula**.
8) In the **Formula Validation** dialog box, select **OK**.
9) Select **Save**.

Date and Duration Calculation

Primavera Cloud Expression Language provides several methods for date calculations. Use date and duration methods when specifying fields or measures that modify data from date fields such as **Project Planned Start** or **Project Planned Finish**. For example, you may want to define a configured field for the Project object that returns the date six months after the current project's **Planned Start Date**. To do so, use the **plusMonths** date method as shown in the following example:

```
return plusMonths(object.Project_planStartDate, 6);
```

You can also use Primavera Cloud Expression Language date methods to perform calculations and set field values based on the contents of multiple date fields. Complete the following task to practice using Primavera Cloud Expression Language date methods.

To define a configured field that returns a message based on a project's **Planned End Date** and **Percent Complete**:

1) In the object selector, select a workspace.
2) In the sidebar, select **Summary & Settings**.
3) On the **Summary & Settings** menu, select **Defaults & Options**, and then select **Project**.
4) On the **Projects** page, select the **Configured Fields** tab.
5) In the table, select **Add**:
    a. In the **Column Label** column, enter **Anticipated Delays**.
    b. In the **View Column Name** column, enter **AD**.
    c. In the **Data Type** column, select **Text**.
    d. In the **Type** column, select **Formula**.
6) In the **Formula** detail window, in the **Formula Editor**, enter the following:

```
def currentDate = new Date();
def weekFromFinish = minusDays(object.Project_planEndDate, 7);
if(object.Project_percentComplete < 95 && currentDate >= weekFromFinish){
      return "Project delay likely";
} else {
      return "No delays anticipated.";
}
```

7) In the **Formula** detail window, select **Validate Formula**.
8) In the **Formula Validation** dialog box, select **OK**.
9) Select **Save**.

To learn more about Primavera Cloud Expression Language date and duration methods, see ***Date Methods*** (on page 42) and ***Duration Methods*** (on page 51).

Measures Formulas

You can use Primavera Cloud Expression Language to define custom portfolio and strategy measures that enable you to track your organization's performance metrics. Primavera Cloud Expression Language supports the creation of two measure types, **Field Based** or **Measure Based**.

Field Based Measures

Field based measures aggregate object field values to set measure actuals and targets. Primavera Cloud Expression Language provides aggregation types that specify how a measure is calculated. For example, the aggregation type **Sum** calculates the result of adding the values of its arguments. Aggregation methods accept two types of arguments: **fields** and an optional **filter**. Field arguments determine what field values will be aggregated. The **filter** argument limits the set of aggregated fields to those matching the conditions specified by the filter. For example, the expression `count(object.Project_projectCode, object.Project_status == 'ACTIVE')` returns the number of projects in the application with a current status of active.

You can also use Primavera Cloud Expression Language operators to combine aggregation types or pass complex field values as arguments.

The following examples illustrate a number of expressions you can use when defining measures. Doubles are valid outside of parenthesis as long as one of the following numeric operators is used: +, -, *, /.

Note: Text surrounded in <> indicates a variable. Variable values must be replaced by valid aggregation methods, measures, or field.

<operator> indicates any valid Primavera Cloud Expression Language numeric operator, such as +, -, *, /.

<double> indicates a value of the Primavera Cloud Expression Language data type Double.

Expression Format	Example Expression
`<aggregationType>(<fieldA>, <filter>)`	`count(object.Project_projectCode, object.Project_status == 'ACTIVE');`
`<aggregationType>(<UDFField>, <filter>)`	`count(object.ProjectCPUDF_APPROVALRATING, object.ProjectCPUDF_APPROVALRATING == 100);`
`<aggregationType>(<field> <operator> <Double>)`	`sum(object.Project_atCompletionCost + 10000);`
`<aggregationType>(<fieldA> <operator> <fieldB>)`	`max(object.Project_currentApprovedBudgetBase - object.Project_currentApprovedBudgetBaseDistributed);`
`<aggregationType>((<fieldA> <operator> <fieldB>) * <Double>`	`average((object.Project_estToCompleteCost + object.Project_currentApprovedBudgetBaseUndistributed) * 100)`

Expression Format	Example Expression
`<aggregationType>(<fieldA> <operator> <Double> <operator> <fieldB> <operator> <Double>)`	`max(object.Project_costVariance * 100 + object.Project_scheduleVariance * 100);`
`<aggregationType>(<fieldA>) <operator> <Double>`	`max(object.Project_percentComplete) * 100`
`<aggregationType>(<fieldA> <operator> <fieldB>) <operator> <Double>`	`average(object.Project_proposedBudgetBase - object.Project_plannedBudget) * 100`
`<aggregationType>((<fieldA> <operator> <fieldB>) <operator> <Double>) <operator> <Double>`	`min((object.Project_plannedBudget + object.Project_proposedBudgetBase) * 100) / 2`

Filters Using Numeric Fields and Numbers

Expression Format	Example Expression
`<aggregationType>(<fieldA>, <fieldB> > <Double>)`	`min(object.Project_projectCode, object.Project_netPresentValue > 100000);`
`<aggregationType>(<fieldA>, <fieldB> > <Double> || <fieldA> > <Double>)`	`max(object.Project_plannedBudget, object.Project_currentApprovedBudgetBaseUndistributed > 100000 || object.ProjectCPUDF_APPROVALRATING > 10000);`
`<aggregationType>(<fieldA>, (<fieldB> + <fieldC>) > <Double>)`	`average(object.Project_currentApprovedBudgetBase, (object.Project_plannedBudget + object.Project_proposedBudget) > 100000);`
`<aggregationType>(<fieldA>, (<fieldB> + <fieldC>) > <fieldD>)`	`count(object.Project_projectCode, (object.Project_currentApprovedBudgetBaseDistributed + object.Project_estToCompleteCost) > object.Project_currentApprovedBudgetBase);`

Expression Format	Example Expression
`<aggregationType>(<fieldA>, (<fieldB> + <fieldC>) > <fieldD> * <Double>)`	`average(object.Project_netPresentValue, (object.ProjectforecastCost + object.Project_estToCompleteCost) > object.Project_currentApprovedBudgetBase * 100);`
`<aggregationType>(<fieldA>, (<fieldB> + <fieldC>) > (<fieldD> * <fieldE>))`	`sum(object.Project_currentApprovedBudgetBase, (object.Project_plannedBudget + object.Project_estToCompleteCost) > (object.Project_atCompletionCost * object.Project_spendCost))`

Filters Using Text Fields

Expression Format	Example Expression
`<aggregationType>(<fieldA>, <fieldB> == 'String1' && <fieldC> != 'String2' )`	`count(object.Project_projectCode, object.Project_status == 'ACTIVE' && object.Project_projectName != 'Barr Harbor Office Refurbishment');`

Filters Using Date Fields

Expression Format	Example Expression
`<aggregationType>(<fieldA>, <dateFieldA> > <dateFieldB>)`	`count(object.Project_projectCode, object.Project_actualStartDate > object.Project_planStartDate)`
`<aggregationType>(<fieldA>, <dateField> > Date)`	`count(object.Project_projectCode, object.Project_actualStartDate > Date.parse('dd/mm/yyyy', '02/06/2017'))`

Expression Format	Example Expression
`<aggregationType>(<fieldA>, <dateFieldA> == Date || <dateFieldB> >= Date)`	`count(object.Project_projectCode, object.Project_planFinishDate == Date.parse('dd-MMM-yyyy', '02-JUN-2017') || object.Project_actualFinishDate >= Date.parse('dd/MMM/yyyy', '02/JUN/2017'))`

Measure Based Measures

A measure based measure leverages existing measures to define actual and target values for a custom measure. Primavera Cloud Expression Language provides **get** methods that enable you to use measure data from various objects within the application. **Get** methods have the following general method signature:

```
<object>.get("<workspaceName>",
"<objectName>").'<measure>_<measureName>'
```

Note: Get methods only accept string literals as arguments. If you pass a variable as an argument to a get method, your Primavera Cloud Expression Language script will fail to validate.

Regular Measure or a Measure-Based Measure from the Current Workspace

Expression Format	Example Expression

`object.'PortfolioMeasure_<Measure Name>'`
 	
`object.'PortfolioMeasure_<Measure Name> .Target'` or `object.'PortfolioMeasure_<Measure Name> .Actual'`
	`object.'PortfolioMeasure_Project Delays'`
	
	`object.'PortfolioMeasure_Project Delays.Target'`
	
	`object.'PortfolioMeasure_Project Delays.Actual'`

`object.'PortfolioMeasure_<Measure Name>' <operator> <Double>`	`object.'PortfolioMeasure_Project Delays' * 100`
`object.'PortfolioMeasure_<Measure Name>' <operator> object.'PortfolioMeasure_<Measure Name>'`	`object.'PortfolioMeasure_Project Delays' + object.'PortfolioMeasure_Additional Expenses'`

Workspace Could be the Current Workspace and Child Workspaces

Expression Format	Example Expression
`<portfolioMeasure>`	`Portfolios.get("JM001","Active Projects").'PortfolioMeasure_Project Delays'`
`<strategyMeasure>`	`Strategies.get("JM001","Maximize NPV").'StrategyMeasure_NPV Aggregate'`
`<portfolioMeasureA> <operator> <portfolioMeasureB>`	`Portfolios.get("JM001","Active Projects").'PortfolioMeasure_Under Budget' - Portfolios.get("KM002","Planned Projects").'PortfolioMeasure_Insufficient Funds'`
`<strategyMeasureA> <operator> <strategyMeasureB>`	`Strategies.get("JM001","Maximize NPV").'StrategyMeasure_NPV Aggregate' + Strategies.get("KM002","Maximize ROI").'StrategyMeasure_ROI Aggregate'`

Combination of Referenced Measures and a Simple Measure-Based Measure from the Current Workspace

Expression Format	Example Expression
`(<portfolioMeasureA> <operator> <portfolioMeasureB>) <operator> <measureA>`	`(Portfolios.get("JM001","Active Projects").'PortfolioMeasure_Active Budgets' + Portfolios.get("KM002","Planned Projects").'PortfolioMeasure_Proposed Budgets') / object.'PortfolioMeasure_Planned Budgets'`

Using Weights in Calculations

Expression Format	Example Expression
`<Double> <operator> <strategyMeasureA> <operator> <Double> <operator> <strategyMeasureB> <operator> <Double> <operator> <strategyMeasureC>`	`0.50 * Strategies.get("JM001","Maximize NPV").'StrategyMeasure_NPV Aggregate' + 0.25 * Strategies.get("KM002","Maximize ROI").'StrategyMeasure_ROI Aggregate' + 0.25 * Strategies.get("IM001","Low Budget").'StrategyMeasure_Budget Aggregate'`
`<Double> <operator> (<portfolioMeasureA> <operator> <portfolioMeasureB>) <operator> <Double> <operator> <portfolioMeasureC>`	`(Portfolios.get("JM001","Active Projects").'PortfolioMeasure_Active Budgets' + Portfolios.get("KM002","Planned Projects").'PortfolioMeasure_Planned Budgets') * 0.50 + Portfolios.get("IM001","Inactive Projects").'PortfolioMeasure_Proposed Budgets' * 0.50`

Language Reference

The following sections provide detailed information on Primavera Cloud Expression Language syntax and supported features. For more information on using Primavera Cloud Expression Language, see ***Primavera Cloud Expression Language Overview*** (on page 5).

Supported Data Types, Operators, and Statements

Primavera Cloud Expression Language is a lightweight scripting language optimized for security and performance within Oracle Primavera Cloud. Primavera Cloud Expression Language syntax is similar to popular programming languages such as Java, and includes additional syntax for working with data in Primavera Cloud.

Primavera Cloud Expression Language supports several data types, operators, and statements. Refer to the sections below for more information on Primavera Cloud Expression Language supported data types and programming constructs.

Data Types

The following table lists Primavera Cloud Expression language data types.

Supported Data Types

Type	Example
Boolean	`true`
Date	`new Date()`
Double	`5.0`
String	`"Hello World"`

Operators

Primavera Cloud Expression Language provides operators for computation and comparison. Operators can only be applied to data types they support. The following tables list Primavera Cloud Expression Language supported operators.

Supported Numeric Operators

Use numeric operators to perform calculations on numeric data types, such as doubles.

Operator	Name	Description	Applicable Data Types	Return Data Type	Example
+	Addition	Sums two Double values.	Double	Double	`2 + 5; //returns 7`
-	Subtraction	Subtracts two Double values.	Double	Double	`5 - 3; //returns 2`
*	Multiplication	Multiplies two Double values.	Double	Double	`2 * 5; //returns 10`
/	Division	Divides two Double values.	Double	Double	`10 / 2; //returns 5` `3 / 2; //returns 1.5`
%	Modulo (remainder)	Returns the remainder resulting from the division of two Doubles.	Double	Double	`5 % 2; //returns 1`

Operator	Name	Description	Applicable Data Types	Return Data Type	Example
**	Exponential	Returns the result of raising one Double to the power of another.	Double	Double	`5 ** 2; //returns 25`
Unary -	Unary Minus	Indicates a negative value.	Double	Double	`-5; //returns -5.` `-5 + 2; //returns -3.`

Supported Logical Operators

Use logical operators to manipulate and combine Boolean values and to construct conditional statements.

Operator	Name	Description	Applicable Data Types	Return Data Type	Example
||	OR	Returns the result of combining Boolean values using a logical OR. If one value or expression contained in the OR statement evaluates to true, the OR expression returns true.	Boolean	Boolean	`true || false;` `//returns true`

Operator	Name	Description	Applicable Data Types	Return Data Type	Example
&&	AND	Returns the result of combining Boolean values using a logical AND. If both values or expressions in the AND statement evaluate to true, the AND expression returns true.	Boolean	Boolean	`true && false;` `//returns false`
!	NOT	Negates the specified Boolean value. Applying the NOT operator to an expression that evaluates to true will return false.	Boolean	Boolean	`!true;` `//returns false`

Supported Comparison Operators

Use comparison operators to measure values against each other. The data types of compared values must match, otherwise the application will return an error. The result of a comparison operation is a Boolean, true or false.

Operator	Name	Description	Applicable Data Types	Return Data Type	Example

==	Equal	Tests if two values are equal.	All	Boolean	`1.0 == 1.0; //returns true`
					`1.0 == 2.3 //returns false`
					`"Hello" == "Hello" //returns true`

!=	Does Not Equal	Tests if two values are not equal.	All	Boolean	`2 + 2 != 5; //returns true`
					`2 + 2 != 4; //returns false`
					`"Hello" != "Goodbye" //returns true`

>	Greater Than	Tests if one value is greater than another.	All	Boolean	`50 > 100; //returns false`
					`50 > 5; //returns true`
					`"2" > "10" //returns true`
					`"Hello" > "World" //returns false`
					`5 > "4" //error. The types of compared values must match.`

Operator	Name	Description	Applicable Data Types	Return Data Type	Example
>=	Greater Than or Equal To	Tests if one value is greater than or equal to another.	All	Boolean	`60 >= 50; //returns true` `60 >= 70 //returns false` `60 >= 60 //returns true` `"Hello" >= "World" //returns false`
<	Less Than	Tests if one value is less than another.	All	Boolean	`50 < 100; //returns true` `50 < 40 //returns false` `"2" < "10" //returns false` `"Hello" < "World"` `//returns true` `"3/1/17" < new Date() //error the types of compared values must match`

Operator	Name	Description	Applicable Data Types	Return Data Type	Example

<=	Less Than or Equal To	Tests if one value is less than or equal to another.	All	Boolean	`60 <= 50; //returns false`
					`60 <= 90 // returns true`
					`60 <= 60 //returns true`
					`"10" <= "2" //returns true`

Supported Special Operators

Use special operators to create structure in your Primavera Cloud Expression Language code and specify how Primavera Cloud should interpret and run your Primavera Cloud Expression Language expressions.

Operator	Name	Description	Applicable Data Types	Example

+	Concatenation	Combines a String with another data type, yielding a new String. String concatenation is applied when the leftmost operand of the + operator is of the data type String. The right operand can be a value of any type. When concatenated, Date values may not serialize in the format anticipated.	String	`"Hello" + " World" //returns "Hello World"`
				`"Lucky Number " + 7; //returns "Lucky Number 7"`
				`"a" + 1 + 2 //returns "a12"`
				`5 + "Hello"; //returns an error. To perform concatenation, the leftmost operand must be a String.`

Operator	Name	Description	Applicable Data Types	Example
=	Assignment	Assigns a value of a supported data type to a variable.	All	`def myVar = 50;`

( )	Parentheses	Specifies logical grouping and evaluation order. Statements in parentheses have increased precedence.	All	`(5 + 2) * 2;`
				`//returns 14`

new	Constructor	Instantiates a new instance of a class, resulting in an object of that class.	Date	`def currDate = new Date();`
//	Comment	Indicates a single-line comment. Commented lines are not evaluated. Use comments to describe and annotate your formula script. Comments extend to the end of a line.	All	`//def a = 5; //because this code is commented, it will not be evaluated.`

/**/	Multi-line Comment	Indicates an extended comment. Multi-line comments are not evaluated. Multi-line comments may span multiple lines.	All	`/* Multi-line comments span`
				`multiple lines.`
				`*/`

Statements

Primavera Cloud Expression Language supports a variety of statements you can use to determine the control flow of Primavera Cloud Expression Language code. The following tables list Primavera Cloud Expression Language statements.

Supported Statements

Statement	Description	Example

def	Specifies a variable. Variables in Primavera Cloud Expression Language are strongly typed and restricted to the data type of their initial assignment. Variables are not accessible outside the scope of the code block they are declared in. Variables declared outside of code blocks or other control structures are global and may be accessed from anywhere within the script.	`def x = "My Variable";`
		`def y = 2;`
		`def z = new Date();`
		`x = y; //error, after the initial declaration and assignment, x can only have a data type of String. Tried to assign a data type of Double.`

return	Specifies a value that should be returned as the result of a script and serve as the default value of the configured field associated with the formula script. The return statement ends script processing. If no return statement is specified, Primavera Cloud Expression Language scripts automatically return the last statement evaluated.	`return "Hello World" // script exits, the text Hello World is returned as the configured field value.`
		`5 + 2; //unreachable code--will not be evaluated. Script processing ends when a return statement is evaluated.`

Statement	Description	Example

{ } (block)	Specifies a sequence of expressions to evaluate. Used to structure Primavera Cloud Expression Language scripts, establish variable scope, and improve readability.
	Variables defined and assigned values within a code block are accessible only within the code block and other structures the code block contains.
		`def a = 8;`
		`{`
		`    def b = 10;`
		`}`
		`return a + b; // error. B is not defined within scope.`

if/else	Code contained in an if block will only be evaluated if the condition specified by the if statement evaluates to true. Code in the else block will be evaluated if the specified condition evaluates to false.	`if (1 + 1 == 2) {`
		`    //block processed if condition is true`
		`    return "True";`
		`} else {`
		`   //block processed if condition is false`
		`    return "Not true!";`
		`}`

Unsupported Programming Constructs

For improved efficiency and security, Primavera Cloud Expression Language does not support the following programming constructs:

- Loops (for example: for, for each, while).
- Function or method definition.
- Arrays, hashes, or collections.
- Mixed assignment operators (for example: ++, --, +=, -=).
- String interpolation.
- Regular expressions.

Note: The application generates a validation error if your Primavera Cloud Expression Language scripts contain unsupported data types or other unsupported control structures.

Supported Methods

Primavera Cloud Expression Language supports a variety of methods for each of its data types, as well as some additional Java methods. The following sections list Primavera Cloud Expression Language supported methods for each data type. Refer to the official ***Java documentation*** (***https://docs.oracle.com/javase/8/docs/api/overview-summary.html***) for more information on each method.

String Methods

Use Primavera Cloud Expression Language supported string methods to manipulate textual data.

Refer to the official ***Java documentation*** (***https://docs.oracle.com/javase/8/docs/api/overview-summary.html***) for more information on each method.

Notes:

- Primavera Cloud Expression Language automatically converts Doubles to Int types when calling methods that require Int arguments.
- Types are indicated in **bold** in each method signature.

String Methods

Method Signature	Description	Parameters	Example
`String.codePointAt(`**Int** `index)`	Returns the Unicode code point value for the character within the string at the index matching the Int passed as an argument.	**Int** index - Index value of a string character.	`"Hello".codePointAt(1); //returns 101, the Unicode code point value of "e"`
`String.codePointBefore(`**Int** `index)`	Returns the Unicode code point value for the character within the string at the index value immediately before the Int passed as an argument.	**Int** index - Index value of a string character.	`"Hello".codePointBefore(1); //returns 72, the Unicode code point value of "H"`

Method Signature	Description	Parameters	Example
`String.codePointCount(`**Int** `indexStart,` **Int** `indexEnd)`	Returns the number of Unicode code points within an index range specified by the Int values passed as arguments.	**Int** indexStart - Value to specify the start of an index range. The number of Unicode points within the specified range is returned by this method. **Int** indexEnd - Value to specify the end of an index range.	`"Hello".codePointCount(1, 4); //returns 3, the number of character in the index range 1 to 4, namely "ell";`

Method Signature	Description	Parameters	Example
`String.compareTo(`**String**` string)`	Compares the lexicographic value of a String to the String value passed as an argument. Returns 0 if the Strings are lexicographically equivalent. Returns a value greater than 0 if the argument String is lexicographically greater, returns a value less than 0 if the argument String is lexicographically lesser. Lexicographic equivalence is based on the order of strings in a lexicon or dictionary. For example, "ant" is lexicographically lesser than "bat" because ant occurs first in an alphabetically ordered lexicon.	**String** string - String to lexicographically compare with the String calling the compareTo method.	`"Hello".compareTo("Goodbye"); //returns a value greater than 0 because "Hello" occurs after "Goodbye" in an ordered dictionary.` `"Hello".compareTo("Hello"); //returns 0 because "Hello" and "Hello" are lexicographically equivalent; they occupy the same position in an ordered lexicon or dictionary.` `"Hello".compareTo("hello"); //returns a value less than 0 because "Hello" occurs before "hello" in an ordered lexicon or dictionary; all capital letter forms precede their lowercase counterparts.`

Method Signature	Description	Parameters	Example

`String.compareToIgnoreCase(`**String** `string)`	Compares the lexicographic value of a String to the String value passed as an argument and ignores differences in case. Returns 0 if the Strings are lexicographically equivalent. Returns a value greater than 0 if the argument String is lexicographically greater, returns a value less than 0 if the argument String is lexicographically lesser.
	Note: Lexicographic equivalence is based on the order of strings in a lexicon or dictionary. For example, "ant" is lexicographically lesser than "bat" because ant occurs first in an alphabetically ordered lexicon.
		String string - String to lexicographically compare with the String calling the compareToIgnoreCase method.	`"Hello".compareToIgnoreCase("hello");`
			`//returns 0.`

`String.contains(`**String** `string)`	Returns a Boolean value indicating whether the String passed as an argument appears within the String calling the contains method.	**String** string - String to find within the String calling the contains method.	`"Hello".contains("lo");`
			`//returns true`
			`"Hello".contains("World");`
			`//returns false`

Method Signature	Description	Parameters	Example
`String.endsWith(`**String** `string)`	Returns a Boolean value indicating whether the String calling the endsWith method ends with the String passed as an argument.	**String** string - String to find at the end of the String calling the endsWith method.	`"Hello".endsWith("lo"); //returns true` `"Hello".endsWith("He"); //returns false`
`String.equalsIgnoreCase(`**String** `string)`	Returns a Boolean indicating whether the String passed as an argument is equal to the String calling the equals method ignoring case. Strings are equivalent if they are a representation of the same sequence of characters.	**String** string - String to check against the String calling the equalsIgnoreCase method.	`"Hello".equalsIgnoreCase("hello"); //returns true`
`String.hashCode()`	Returns a numeric hash code value representing the String calling the hashCode method.	No method parameters.	`"Hello".hashCode(); //returns 69609650`
`String.lastIndexOf(`**String** `string)`	Returns the last index value at which the String passed as an argument occurs in the String calling the lastIndexOf method.	**String** string - String to find the index of the last occurrence of in the String calling the lastIndexOf method.	`"Hello".lastIndexOf("l"); //returns 3`

Method Signature	Description	Parameters	Example
`String.length()`	Returns the length of the String calling the length method.	No method parameters.	`"Hello".length(); //returns 5`
`String.replace(`**String** `stringToFind,` **String** `stringToReplace)`	Replaces occurrences of the first String passed as an argument in the String calling the method with the second String passed as an argument.	**String** stringToFind - String to find within the String calling the replace method. **String** stringToReplace - String to use to replace the String passed as the first argument to the method.	`"Hello World".replace("World", "Universe!"); //returns "Hello Universe!"`
`String.startsWith(`**String** `string)`	Returns a Boolean value indicating whether the String calling the startsWith method starts with the String passed as an argument.	**String** string - String to find at the end of the String calling the startsWith method.	`"Hello".startsWith("He"); //returns true`

Method Signature	Description	Parameters	Example
`String.substring(`**Int** `index)`	Returns a substring of the String calling the substring method. The resulting substring begins with the character stored at the String index location matching the passed Int argument and ends with the last character of the String.	**Int** index - Index value specifying where to begin extracting the substring from the String calling the substring method.	`"Hello".substring(2); //returns "llo"`
`String.toLowerCase()`	Returns the result of converting all of the characters of the String calling the toLowerCase method to their lowercase form.	No method parameters.	`"Hello".toLowerCase(); //returns "hello"`
`String.toUpperCase()`	Returns the result of converting all of the characters of the String calling the toUpperCase method to their uppercase form.	No method parameters.	`"Hello".toUpperCase(); //returns "HELLO"`
`String.trim()`	Returns the result of removing space characters that appear at the beginning or the end of the String calling the trim method.	No method parameters.	`"  Hello World ".trim(); //returns "Hello World"`

Additional String Methods

Method Signature	Description	Parameters	Examples
`String.concat(`**String** `string)`	Returns the concatenation of the String passed as an argument and the String calling this method.	**String** string - A value of type String to append to the end of the String calling the method.	`"Hello".concat(" World"); //returns "Hello World"`

`String.equals(`**Object** `obj)`	Returns the result of comparing the String calling this method to the object passed as an argument. Returns true if the Object value passed as an argument represents a String equivalent to the String calling the method, otherwise returns false.	**Object** object - An Object against which the String calling the method is compared.	`"Hello".equals("Hello"); //returns true.`
			`"Hello".equals("World"); // returns false.`

`String.indexOf(`**Int** `character)`	Returns the indexical position of the specified character value in the String calling this method. Arguments may be passed as a literal quote enclosed character or as a Unicode code point Integer representing that character.	**Int** character - A value of type Int in Unicode code point format representing an alphanumeric character.	`"Hello".indexOf("H"); //returns 0`
			`"Hello".indexOf(72); //returns 0`
			`"Hello".indexOf("i"); //returns -1`
			`"Hello".indexOf(105); //returns -1`

`String.isEmpty()`	Returns true if the length of the string calling the method is equal to 0, otherwise returns false.	No method parameters.	`"Hello".isEmpty(); //returns false`
			`"".isEmpty(); //returns true`

`String.offsetByCodePoints(`**Int** `index,` **Int** `codePointOffset)`	Returns the index within the String calling this method that is offset from the index argument by the number of code points specified by the codePointOffset argument.	**Int** index - Initial index value within the Strign calling this method. **Int** codePointOffset - Number of points to offset from the index argument.	`"Hello".offsetByCodePoints(1, 2); //returns 3`
`String.toString()`	Returns the String calling this method.	No method parameters.	`"Hello".toString(); //returns "Hello"`
`String.valueOf(`**Object** `obj)`	Returns the string representation of the Object passed as an argument.	**Object** obj - An object to convert to a String value.	`String.valueOf(3); //returns "3"`

Double Methods

Use Primavera Cloud Expression Language supported methods to manipulate decimal numbers of the type double.

Refer to the official ***Java documentation*** (***https://docs.oracle.com/javase/8/docs/api/overview-summary.html***) for more information on each method.

Notes:

- Primavera Cloud Expression Language automatically converts Doubles to Int types when calling methods that require Int arguments.
- Types are indicated in **bold** in each method signature.

Double Methods

Method Signature	Description	Parameters	Example
`Double.byteValue()`	Returns the value of the Double calling the byteValue method represented as a byte.	No method parameters.	`def x = 65.3;` `def y = 77.8;` `x.byteValue(); //returns 65` `y.byteValue(); //returns 77`

Method Signature	Description	Parameters	Example

`Double.compare(`**Double** `valueOne,` **Double** `valueTwo)`	Compares the value of two Double arguments. Returns 0 if the arguments are equal, less than 0 if the first argument is less than the second, and greater than 0 if the first argument is greater than the second.	**Double** valueOne - First value to compare.
		Double valueTwo - Second value to compare.
			`Double.compare(35.4, 22.1); //returns 1`
			`Double.compare(35.4, 35.4); //returns 0`
			
			`Double.compare(35.4, 42.1); //returns -1`

`Double.compareTo(`**Double** `value)`	Compares the value of the Double calling the compareTo method with a Double passed as an argument. Returns 0 if the argument is equal to the Double calling the method, less than 0 if the Double calling the method is less than the argument, and greater than 0 if the Double calling the method is greater than the argument.	**Double** value - Value to compare against the Double calling the method.	`def x = 35.4;`
			`x.compareTo(100.0); // returns -1`
			`x.compareTo(35.4); // returns 0`
			`x.compareTo(22.1); // returns 1`

`Double.parseDouble(`**String** `string)`	Returns a Double value matching the numeric value contained in the String passed as an argument.	**String** string - String to convert to a Double value.	`Double.parseDouble("35.4"); //returns 35.4`
`Double.toString(`**Double** `value);`	Returns a String representing the Double value passed as an argument.	**Double** value - Double value to convert to a String.	`Double.toString(35.4); //returns "35.4"`

Method Signature	**Description**	**Parameters**	**Example**

`Double.valueOf(`**String** `string)`
`Double.valueOf(`**Double** `value)`
	Returns a Double value representing the value of a String or Double passed as an argument.	**String** string - String from which to extract a Double value.
		Double value - Double object form which to extract a Double value.
			`Double.valueOf("35.4");`
			`//returns 35.4`
			`Double.valueOf(35.4);`
			`//returns 35.4`

Additional Double Methods

Method Signature	Description	Parameters	Examples
Double.doubleValue()	Returns the value of the Double calling this method.	No method parameters.	def x = 12.5; x.doubleValue(); //returns 12.5

Double.equals(**Object** obj)	Compares the double calling this method to the Object passed as an argument. Returns true if the argument is a Double value representing the same value as the Double calling the argument, otherwise returns false.	**Object** obj - An Object against which the Double calling the method is compared.	def x = 12.5;
			x.equals(12.5); //returns true
			x.equals(34.8); //returns false
			x.equals("Hello"); //returns false

Double.floatValue()	Returns the float value of the Double calling this method.	No method parameters.	def x = 12.5; x.floatValue(); //returns 12.5
Double.hashCode()	Returns a hash code for the Double calling this method.	No method parameters.	def x = 12.5; x.hashCode(); //returns 1076428800
Double.intValue()	Returns the value of the Double calling this method as an Int type.	No method parameters.	def x = 12.5; x.intValue(); //returns 12
Double.isInfinite()	Returns true if the Double calling this method is infinitely large, otherwise returns false.	No method parameters.	def x = 12.5; x.isInfinite(); //returns false.

Double.isNaN()	Returns true if the Double calling this method is not a number, otherwise returns false.	No method parameters.	def x = 12.5 x.isNaN(); //returns false
Double.longValue()	Returns the value of the Double calling this method as a Long type.	No method parameters.	def x = 12.5; x.longValue(); //returns 12
Double.shortValue()	Returns the value of the Double calling this method as a Short type.	No method parameters.	def x = 12.5; x.shortValue(); //returns 12
Double.toHexString( )	Returns the Double passed as an argument as a hexadecimal string.	**Double** value - Double to convert to a hexidecimal string.	Double.toHexString(12.5); //returns 0x1.9p3

Integer Methods

Use Primavera Cloud Expression Language supported integer methods to manipulate numeric data of the integer type.

Refer to the official ***Java documentation*** (***https://docs.oracle.com/javase/8/docs/api/overview-summary.html***) for more information on each method.

Notes:

- Primavera Cloud Expression Language automatically converts Doubles to Int types when calling methods that require Int arguments.
- Types are indicated in **bold** in each method signature.

Integer Methods

Method Signature	Description	Parameters	Example
`Integer.parseInt(`**String**` string)`	Returns an Integer value representing the contents of the String passed as an argument.	**String** string - String from which to extract an integer value.	`Integer.parseInt("45"); //returns 45`

Method Signature	Description	Parameters	Example

`Integer.valueOf(`**String** `string)`
 `Integer.valueOf(`**String** `string,` **Int** `radix)`
	Returns an Integer value representing the value of a String passed as an argument.	**String** string - String from which to extract an Integer value.
		 Int radix - Specifies the radix to use when converting the String argument into an Integer, for example, 2, 8, 16.
			`Integer.valueOf("45"); //returns 45`
			 `Integer.valueOf("101101", 2); //returns 45`

Additional Integer Methods

Method Signature	Description	Parameters	Examples
`Integer.bitCount(`**Int** `value)`	Returns the number of one bits in the binary representation of the Int value passed as an argument.	**Int** value -Int value containing one bits to be counted.	`Integer.bitCount(104); //returns 3`

`Integer.Compare(`**Int** `x,` **Int** `y)`	Compares the value of the first Int passed as an argument to the value of the second Int passed as an argument. Returns 0 if the values are equal, returns less than 0 if the first argument is less than the second, and returns a value greater than 0 if the first argument is greater than the second.	**Int** x - The first Int value to compare.
		 Int y - The second Int value to compare.
			Integer.compare(3,3); //returns 0
			 Integer.compare(3, 4); //returns -1
			 Integer.compare(4,3); //returns 1

`Integer.decode(`**String** `number)`	Returns the result of converting the String passed as an argument into an Int value. The argument String must be in decimal, hexadecimal, or octal format.	**String** number - String to convert into an Int value.	Integer.decode("24"); //returns 24
`Integer.hashCode()`	Returns a hash code representing the Integer calling this method.	No method parameters.	def x = 12; x.hashCode(); //returns 12
`Integer.highestOneBit(`**Int** `value)`	Returns an Int value with at most a single one bit in the position of the leftmost one bit contained in the binary representation of the Int value passed as an argument.	**Int** value - Integer value containing one bits in its binary representation.	Integer.highestOneBit(12); //returns 8
`Integer.lowestOneBit(`**Int** `value)`	Returns an Int value with at most a single one bit in the position of the rightmost one bit contained in the binary representation of the Int value passed as an argument.	**Int** value - Integer value containing one bits in its binary representation.	Integer.lowestOneBit(12); //returns 4
`Integer.numberOfLeadingZeros(`**Int** `value)`	Returns the number of zeros preceding the leftmost one bit in the binary representation of the Int value passed as an argument.	**Int** value - Integer value containing one bits in its binary representation.	Integer.numberOfLeadingZeros(12); //returns 28

`Integer.numberOfTrailingZeros(`**Int** `value)`	Returns the number of zeros following the rightmost one bit in the binary representation of the Int value passed as an argument.	**Int** value - Integer value containing one bits in its binary representation.	Integer.numberOfTrailingZeros(12); //returns 2
Integer.reverse(**Int** value)	Returns the result of reversing the order of bits contained in the binary representation of the Int value passed as an argument.	**Int** value - Integer value containing one bits in its binary representation.	Integer.reverse(12); //returns 805306368
`Integer.rotateLeft(`**Int** `value,` **Int** `distance)`	Returns the result of rotating the binary representation of the Int value passed as an argument left by the value specified by the second Int value passed as an argument.	**Int** value - Integer value containing one bits in its binary representation. **Int** distance - Number of places to rotate the binary representation of the first argument left.	Integer.rotateLeft(12, 4); //returns 192
`Integer.rotateRight(`**Int** `value,` **Int** `distance)`	Returns the result of rotating the binary representation of the Int value passed as an argument right by the value specified by the second Int value passed as an argument.	**Int** value - Integer value containing one bits in its binary representation. **Int** distance - Number of places to rotate the binary representation of the first argument right.	Integer.rotateRight(12, 4); //returns -1073741824
`Integer.signum(`**Int** `value)`	Returns the signum function of the Int value passed as an argument. The signum value indicates whether an integer is positive, negative, or zero.	**Int** value - Int value against which to calculate signum.	Integer.signum(12); //returns 1

`Integer.toBinaryString(`**Int**` value)`	Returns a string representation of the binary representation of the Int value passed as an argument.	**Int** value - Integer value to convert to a binary representation.	Integer.toBinaryString(12); //returns "1100"
`Integer.toHexString(`**Int**` value)`	Returns a string representation of the hexadecimal representation of the Int value passed as an argument.	**Int** value - Integer value to convert to a hexadecimal representation.	Integer.toHexString(12); //returns "c"
`Integer.toOctalString(`**Int**` value)`	Returns a string representation of the octal representation of the Int value passed as an argument.	**Int** value - Integer value to convert to an octal representation.	Integer.toOctalString(); //returns 14

Date Methods

Use Primavera Cloud supported date methods to manipulate date data.

Refer to the official ***Java documentation*** (***https://docs.oracle.com/javase/8/docs/api/overview-summary.html***) for more information on each method.

Notes:

- Primavera Cloud Expression Language automatically converts Doubles to Int types when calling methods that require Int arguments.
- Types are indicated in **bold** in each method signature.

Date Methods

Method Signature	Description	Parameters	Example

Method Signature	**Description**	**Parameters**	**Example**

`Date.after(`**Date** `when)`	Returns a Boolean value indicating whether the Date calling the after method occurs after the Date passed as an argument.	**Date** when - Date to compare against.	`def now  = new Date();`
			`def later = new Date("2999 Nov 27");`
			`now.after(later); //returns false`
			`later.after(now); //returns true`

`Date.before(`**Date** `when)`	Returns a Boolean value indicating whether the Date calling the before method occurs before the Date passed as an argument.	**Date** when - Date to compare against.	`def now  = new Date();`
			`def later = new Date("2999 Nov 27");`
			`now.before(later); //returns true`
			`later.before(now); //returns false`

`Date.compareTo(`**Date** `anotherDate)`	Compares the value of the Date passed as an argument to the Date calling the compareTo method. Returns a 0 if the dates are equal, less than 0 if the Date calling the compareTo method occurs before the argument, and greater than 0 if the Date calling the compareTo method occurs after the argument.	**Date** anotherDate - Date to compare against.	`def now = new Date();`
			`def then = now;`
			`def later = new Date("2999 Nov 27");`
			`now.compareTo(then); //returns 0`
			`now.compareTo(later); //returns -1`

Method Signature	Description	Parameters	Example
`Date.equals(`**Date** `anotherDate)`	Checks if the Date calling the equals method is equivalent to the Date passed as an argument. Dates are equivalent if they represent the same point in time to the millisecond. Returns true if the Dates are equal and returns false otherwise.	**Date** anotherDate - Date value to compare against.	`def now = new Date();` `def then = now;` `def later = new Date("2999 Nov 27");` `now.equals(then); //returns true` `now.equals(later); //returns false`
`Date.getDate()`	Returns a value between 1 and 31 representing the day of the Date calling the getDate method.	No method parameters.	`def now = new Date();` `now.getDate(); //returns 20`
`Date.getTime()`	Returns the value of the Date calling the getTime method represented as the number of milliseconds since January 1, 1970, 00:00:00 GMT.	No method parameters.	`def now = new Date();` `now.getTime(); //returns 1490039162479`
`Date.hashCode()`	Returns a hash code representing the Date calling the hashCode method.	No method parameters.	`def now = new Date();` `now.hashCode(); //returns a hash code value representing this date, for example -313684330`

Additional Date Methods

Method Signature	Description	Parameters	Example

`Date.getHours()`	Returns a value between 0 and 23 representing the hours of the Date calling this method.	No method parameters.	`x = new Date(0;` `x.getHours(); //returns a value similar to 15`
`Date.getMinutes()`	Returns a value between 0 and 59 representing the minutes of the Date calling this method.	No method parameters.	`x = new Date();` `x.getMinutes(); //returns a value similar to 41`
`Date.getSeconds()`	Returns a value between 0 and 61 representing the minutes of the Date calling this method.	No method parameters.	`x = new Date(); x.getSeconds(); //returns a value similar to 32`
`Date.getTimezoneOffset()`	Returns the offset between the Date calling this method and UTC represented in minutes.	No method parameters.	`x = new Date(); x.getTimezoneOffset(); //returns a value similar to 0`
`Date.getYear()`	Returns the result of subtracting 1900 from the year contained in the Date calling this method.	No method parameters.	`x = new Date();` `x.getYear(); //returns a value similar to 117`
`Date.parse(`**String** `string)`	Returns the result of converting the String passed as an argument to a Date value.	**String** string - String representation of a date.	`Date.parse("3/4/17"); //returns 1488585600000`
`Date.toGMTString()`	Returns a String representing the Date calling this method in GMT format.	No method parameters.	`x = new Date();` `x.toGMTString(); //returns "17 May 2017 15:41:04 GMT"`
`Date.toLocaleString()`	Returns a String representing the Date calling this method in an implementation independent form.	No method parameters.	`x = new Date();` `x.toLocaleString(); //returns a value similar to "May 17, 2017 3:41:04 PM"`

`Date.toString()`	Returns a String representing the Date calling this method in the form dow mon dd hh:mm:ss zzz yyyy.	No method parameters.	`x = new Date();` `x.toString(); //returns a value similar to "Wed May 17 15:41:04 UTC 2017"`

Math Methods

Use Primavera Cloud supported math methods to evaluate common mathematical functions, such as sine and cosine.

Refer to the official ***Java documentation*** (***https://docs.oracle.com/javase/8/docs/api/overview-summary.html***) for more information on each method.

Notes:

- Primavera Cloud Expression Language automatically converts Doubles to Int types when calling methods that require Int arguments.
- Types are indicated in **bold** in each method signature.

Math Methods

Method Signature	Description	Parameters	Example
`Math.abs(`**Double** `value)`	Returns a value representing the absolute value of the Double passed as an argument.	**Double** value - Value against which the absolute value will be computed.	`Math.abs(-34.5); //returns 34.5` `Math.abs(34.5); //returns 34.5`
`Math.acos(`**Double** `value)`	Returns a value representing the result of computing the arc cosine of the Double passed as an argument.	**Double** value - Value against which arc cosine will be computed.	`Math.acos(35.4); //returns NaN` `Math.acos(.005); //returns 1.565796305961329`

Method Signature	Description	Parameters	Example
`Math.asin(`**Double** `value)`	Returns a value representing the result of computing the arc sine of the Double passed as an argument.	**Double** value - Value against which arc sine will be computed.	`Math.asin(34.5); //returns NaN` `Math.asin(.005); //returns 0.005000020833567712`
`Math.atan(`**Double** `value)`	Returns a value representing the result of computing the arc tangent of the Double passed as an argument.	**Double** value - Value against which the arc tangent will be computed.	`Math.atan(34.5); //returns 1.5418189329433354` `Math.atan(.005); //returns 0.0049999583339583225`
`Math.ceil(`**Double** `value)`	Returns a value representing the smallest mathematical integer that is greater than or equal to the Double passed as an argument.	**Double** value - Value against which the mathematical ceiling will be computed.	`Math.ceil(35.4); //returns 36.0` `Math.ceil(-35.4); //returns -35.0`
`Math.cos(`**Double** `value)`	Returns a value representing the result of computing the cosine of the Double passed as an argument.	**Double** value - Value against which cosine will be computed.	`Math.cos(35.4); //returns -0.6656134553337595` `Math.cos(.005); //returns 0.9999875000260416`
`Math.cosh(`**Double** `value)`	Returns a value representing the result of computing the hyperbolic cosine of the Double passed as an argument.	**Double** value - Value against which the hyperbolic cosine will be computed.	`Math.cosh(35.4); //returns 1.1830270194762338E15` `Math.cosh(.005); //returns 1.0000125000260416`

Method Signature	Description	Parameters	Example
`Math.exp(`**Double** `value)`	Returns a value representing the result of raising Euler's number to the power of the value of the Double passed as an argument.	**Double** value - Value representing the power to which Euler's number will be raised.	`Math.exp(2); //returns 7.38905609893065`
`Math.floor(`**Double** `value)`	Returns a value representing the largest mathematical integer that is less than or equal to the Double passed as an argument.	**Double** value - Value against which the mathematical floor will be computed.	`Math.floor(35.4); //returns 35.0` `Math.floor(-35.4); //returns -36.0`
`Math.log(`**Double** `value)`	Returns a value representing the result of computing the natural logarithm of the Double passed as an argument.	**Double** value - Value against which logarithm will be computed.	`Math.log(35.4); //returns 3.5409593240373143`
`Math.log10(`**Double** `value)`	Returns a value representing the result of computing the base 10 logarithm of the Double passed as an argument.	**Double** value - Value against which base 10 logarithm will be computed.	`Math.log10(35.4); //returns 1.5490032620257879`
`Math.max(`**Double** `valueOne,` **Double** `valueTwo)`	Returns the greater of the two Doubles passed as arguments.	**Double** valueOne - First value to compare. **Double** valueTwo - Second value to compare.	`Math.max(35.4, 22.1); //returns 35.4`
`Math.min(`**Double** `valueOne,` **Double** `valueTwo)`	Returns the lesser of the two Doubles passed as arguments.	**Double** valueOne - First value to compare. **Double** valueTwo - Second value to compare.	`Math.min(35.4, 22.1); //returns 22.1`

Method Signature	Description	Parameters	Example
`Math.pow(`**Double** `valueOne,` **Double** `valueTwo)`	Returns a value representing the result of raising the first Double passed as an argument to the power of the second argument.	**Double** valueOne - Value to raise to the power specified by the second argument. **Double** valueTwo - Value specifying the power to which the first argument will be raised.	`Math.pow(5.0, 2.0); //returns 25.0`
`Math.random()`	Returns a random positive Double value greater than 0.0 and less than 1.0.	No method parameters.	`Math.random(); //returns, for example, 0.9086713591277327`
`Math.round(`**Double** `value)`	Returns an Integer representing the result of rounding the Double passed as an argument to the nearest mathematical integer.	**Double** value - Value to round to the nearest integer.	`Math.round(35.4); //returns 35`

`Math.signum(`**Double** `value)`	Returns an Integer value representing the signum function of the Double passed as an argument. Returns 0 if the argument is zero, returns -1 if the argument is negative, and returns 1 if the argument is positive.	**Double** value - Value against which signum will be computed.	`Math.signum(35.4); //returns 1.0`
			`Math.signum(0); //returns 0.0`
			`Math.signum(-35.4); //returns -1.0`

`Math.sin(`**Double** `value)`	Returns a value representing the result of computing the sine of the Double passed as an argument.	**Double** value - Value against which sine will be computed.	`Math.sin(35.4); //returns -0.7462966756449163`
			`Math.sin(.005); //returns 0.004999979166692708`

Method Signature	Description	Parameters	Example
`Math.sqrt(`**Double** `value)`	Returns a value representing the result of computing the square root of the Double passed as an argument.	**Double** value - Value against which square root will be computed.	`Math.sqrt(25.0); //returns 5.0`
`Math.tan(`**Double** `value)`	Returns a value representing the result of computing the tangent of the first Double passed as an argument.	**Double** value - Value against which tangent will be computed.	`Math.tan(35.4); //returns 1.1212163300856046` `Math.tan(.005); //returns 0.0050000416670833376`
`Math.toDegrees(`**Double** `value)`	Returns the result of converting the Double passed as an argument measured in radians to a value measured in degrees.	**Double** value - Value to convert to degrees.	`Math.toDegrees(35.4); //returns 2028.2705947631143`
`Math.toRadians(`**Double** `value)`	Returns the result of converting the Double passed as an argument measured in degrees to a value measured in radians.	**Double** value - Value to convert to radians.	`Math.toRadians(35.4); //returns 0.6178465552059926`

Additional Math Methods

Method Signature	Description	Parameters	Example
`Math.cbrt(`**Double** `value)`	Returns the cube root of the Double passed as an argument.	**Double** value - Value against which the cube root will be calculated.	`Math.cbrt(8.0); //returns 2.0`

`Math.IEEEremainder(`**Double** `dividend` **Double** `divisor)`	Computes the remainder, as prescribed by the IEEE 754 standard, of the Doubles passed as arguments. The first argument represents the dividend, and the second argument represents the divisor.	**Double** dividend - A double value that represents the dividend of a division operation. **Double** divisor - A double that represents the divisor of a division operation.	`Math.IEEEremainder(10.0, 1.4) //returns 0.20000000000000062`
`Math.rint(`**Double** `value)`	Returns a Double that is closest to the Double value passed as an argument and is also an Integer.	**Double** value - Value against which rint will be computed.	`Math.rint(34.78); //returns 35.0`
`Math.ulp(`**Double** `value)`	Returns the positive distance between the floating-point value of the Double passed as an argument and the Double value that is next largest in magnitude.	**Double** value -Value against which positive distance to a value of the next magnitude will be computed.	`Math.ulp(12.2) //returns 1.7763568394002505E-15`

Duration Methods

Primavera Cloud Expression Language provides additional methods for modifying objects of the Date data type.

The following table lists methods you can use to modify Date values in your Primavera Cloud Expression Language scripts:

Method Signature	Description	Example

`plusHours(`**Date** `date,` **Int** `value)`	Add hours to the date field.	`def now = new Date();`
		`def nextDay = plusHours(now, 24);`
		`return nextDay; //returns the value of now increased by a duration of 24 hours since the initial value of now`

`plusDays(`**Date** `date,` **Int** `value)`	Add days to the date field.	`def now = new Date();`
		`def nextDay = plusDays(now, 1);`
		`return nextDay //returns the value of now increased by a duration of one day since the initial value of now`

`plusWeeks(`**Date** `date,` **Int** `value)`	Add weeks to the date field.	`def now = new Date();`
		`def nextWeek = plusWeeks(now, 1);`
		`return nextWeek; //returns the value of now increased by a duration of one week since the initial value of now`

`plusMonths(`**Date** `date,` **Int** `value)`	Add months to the date field.	`def now = new Date();`
		`def nextMonth = plusMonths(now, 1);`
		`return nextMonth; //returns the value of now increased by a duration of one month since the initial value of now`

`plusYears(`**Date** `date,` **Int** `value)`	Add years to the date field.	`def now = new Date();`
		`def nextYear = plusYears(now, 1);`
		`return nextYear; //returns the value of now increased by a duration of one year since the initial value of now`

Method Signature	Description	Example

`minusHours(`**Date** `date,` **Int** `value)`	Subtract hours from the date field.	`def now =  new Date();`
		`def yesterday = minusHours(now, 24);`
		`return yesterday; // returns the value of now decreased by a duration of 24 hours since the initial value of now`

`minusDays(`**Date** `date,` **Int** `value)`	Subtract days from the date field.	`def now = new Date();`
		`def yesterday = minusDays(now, 1);`
		`return yesterday; //returns the value of now decreased by a duration of one day since the initial value of now`

`minusWeeks(`**Date** `date,` **Int** `value)`	Subtract weeks from the date field.	`def now = new Date();`
		`def lastWeek = minusWeeks(now, 1);`
		`return lastWeek; //returns the value of now decreased by a duration of one week since the initial value of now`

`minusMonths(`**Date** `date,` **Int** `value)`	Subtract months from the date field.	`def now = new Date();`
		`def lastMonth = minusMonths(now, 1);`
		`return lastMonth; //returns the value of now decreased by a duration of one month since the initial value of now`

`minusYears(`**Date** `date,` **Int** `value)`	Subtract years from the date field.	`def now = new Date();`
		`def lastYear = minusYears(now, 1);`
		`return lastYear; //returns the value of now decreased by a duration of one year since the initial value of now`

Method Signature	Description	Example
minusDate(**Date** dateOne, **Date** dateTwo)	Subtract one date from another. Returns the difference in days between the dates passed as arguments. If the second date passed as an argument occurs after the first date argument, the returned value is negative.	`def now = new Date();` `def tomorrow = plusHours(now, 24);` `return minusDate(now, tomorrow); //returns the difference in number of days between the value of now decreased by the value of tomorrow. In this case -1.0.`

Object Naming Conventions

The following are naming conventions for properties on the Primavera Cloud Expression Language default object.

Standard Fields

```
object.<Prefix>_<logicalPropertyName>
```

For example:

```
object.Project_plannedStart
object.Portfolio_currentApprovedBudget
```

Configured Fields

```
object.<Prefix>_<VIEW_COLUMN_NAME_OF_CONFIGURED_FIELD>
```

For example:

```
object.Project_CONFIGURED_FIELD
object.Portfolio_CONFIGURED_FIELD
```

You can determine the column name of a configured field needed for the expression by viewing the configured fields for a particular object in the Summary & Settings panel.

To view the configured fields for an object:

1) In the object selector, select a workspace.
2) In the sidebar, select ⚙ **Summary & Settings**.
3) On the **Summary & Settings** menu, select **Defaults & Options**.

4) Select an object, and then select the **Configured Fields** tab. The name in the **View Column Name** field is the name to use in the expression.

Code Types and Measures

Codes and measures may have names that include characters that are not valid in identifiers, such as spaces, quotation marks, and reserved words. Use quotation marks around identifiers containing invalid characters to ensure your Primavera Cloud Expression Language scripts are interpreted correctly.

```
object.'<Prefix>_<Unique Name>'
```

Note: Single or double quotation marks must surround the property name if it contains characters that are not valid in identifiers.

For example:

```
object.'ProjectCode_Project Type'
object.'PortfolioMeasure_# Active Projects'
```