Oracle
Primavera
Gateway Groovy Scripting Guide

Version 20
February 2020

Contents

Overview

Groovy is a java-compliant scripting language used to customize the processing of objects and fields in Gateway. Use Groovy expressions to manipulate objects and fields in:

- Field mapping templates
- File Parsers
- File Generators

> **Note**: Groovy syntax in Gateway is fully compatible with Primavera Cloud Expression Language (PEL) used in Oracle Primavera Cloud. If you have created Groovy scripts for a previous version of Gateway, you will need to review and update Groovy scripts.

Gateway developers responsible for creating any of the above artifacts should use this guide.

The following sections describe how to write Groovy code to include objects and fields in Gateway to address your business needs. For additional information on supported types, methods, and operators, see the ***Language Reference*** (on page 9).

Within our documentation, some content might be specific for cloud deployments while other content is relevant for on-premises deployments. Any content that applies to only one of these deployments is labeled accordingly.

In This Section

About Personal Information (PI)

Personal information (PI) is any piece of data which can be used on its own or with other information to identify, contact, or locate an individual or identify an individual in context. This information is not limited to a person's name, address, and contact details. For example, a person's IP address, phone IMEI number, gender, and location at a particular time could all be personal information. Depending on local data protection laws, organizations may be responsible for ensuring the privacy of PI wherever it is stored, including in backups, locally stored downloads, and data stored in development environments.

> **Caution**: Personal information (PI) may be at risk of exposure. Depending on local data protection laws, organizations may be responsible for mitigating any risk of exposure.

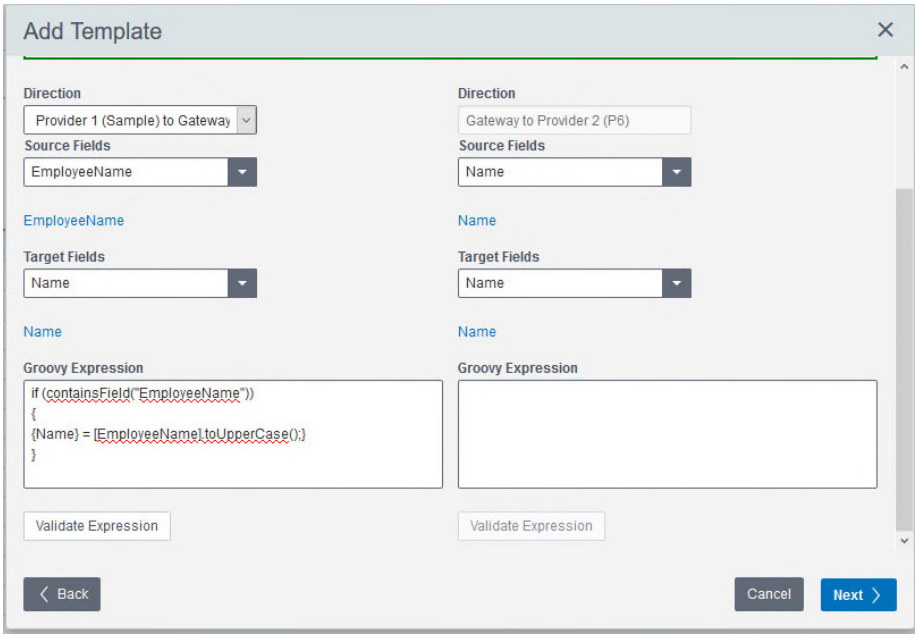
Working with the Script Editor

Gateway includes a script editor to help you create valid Groovy scripts for defining objects and fields.

For Field Mapping Templates

The script editor allows you to

- Select the source provider and the destination provider thereby indicating the direction of the data flow
- Select fields in the source provider and / or the destination provider
- Enter Groovy expressions for the selected fields
- Use the **Validate Expression** button to help ensure the scripts follow Groovy syntax rules



On successful validation, the following message displays: *Syntax OK.*

For File Converters

Add or edit a File converter (parser or generator) to process specific objects using Groovy.

Accessing the Script Editor

Access the Groovy script editor in Gateway depends on the purpose for which it is being used.

For Field Mapping Templates

To add or edit field mapping templates with Groovy code as follows:

1) In the sidebar, select **Configuration**.
2) Select the **Customization** tab.

3) In the **Field Mapping Templates** section, *Select Business Object* from the list.
4) Select ＋ **Add...** or ✎ **Edit...**.
   The **Template** wizard displays.
5) If you add a mapping template, in the **General** tab, select the **Template Type** as *Groovy*.
6) Follow the sequence to specify the Groovy code in the **Mappings** tab of the wizard.

For File Converters

1) In the sidebar, select **Configuration**.
2) Select the **File Converters** tab.
3) Select ＋ **Add...** or ✎ **Edit...**.
4) The File Converter dialog box displays.
5) In the **Script** section, enter the Groovy code and select **Validate** to check for errors.
6) Select **Save**.

Writing a Groovy Script

The following is a code example for a Resource Generator which generates a file containing only resource objects.

```
setCurrentObjectName(context, "Resource");
            def fieldNameCount = getFieldNameCount(context);
            if (fieldNameCount > 0) {

                def fieldNameIndex = 0;
                while (fieldNameIndex < fieldNameCount) {
                    def fieldName = getFieldNameByIndex(context,
fieldNameIndex);
                    writeCell(context, fieldName);
                    fieldNameIndex = fieldNameIndex + 1;
                }
                newline(context);
                def objectCount = getObjectCount(context);
                def objectIndex = 0;
                while (objectIndex < objectCount) {
                    setCurrentObject(context, objectIndex);

                    fieldNameIndex = 0;
                    while (fieldNameIndex < fieldNameCount) {
                        def fieldName = getFieldNameByIndex(context,
fieldNameIndex);
                        def value = getFieldValue(context, fieldName);
                        writeCell(context, value);
                        fieldNameIndex = fieldNameIndex +1;
                    }
                    newline(context);
                    objectIndex = objectIndex + 1;
                }
            }
```

Language Reference

The following sections provide detailed information on Groovy scripting syntax and supported features.

In This Section

Supported Data Types, Operators and Statements

Groovy is a lightweight scripting language optimized for security and performance within Primavera Gateway. Groovy syntax includes additional syntax for working with PEL (Primavera Cloud expression language).

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

Supported Data Types

The following table lists Groovy data types supported in Gateway.

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

Operators

Groovy provides operators for computation and comparison. Operators can only be applied to data types they support. The following tables list Groovy supported operators in Gateway.

- Supported Numeric Operators
- Supported Logical 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`
**	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

Supported 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`
&&	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.`

>=	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`

<=	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 Groovy code and specify how Gateway should interpret and execute the 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.`

=	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. */`

Supported Groovy Statements

Groovy supports a variety of statements you can use to determine the control flow of Groovy code. The following tables list Groovy statements.

Statement	Description	Example

def	Specifies a variable. Variables in Groovy 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 execution. If no return statement is specified, Groovy 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 execution ends when a return statement is evaluated.`

{ } (block)	Specifies a sequence of expressions to evaluate. Used to structure Groovy 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 executed if condition is true`
		`return "True";`
		`} else {`
		`//block executed if condition is false`
		`return "Not true!";`
		`}`

Unsupported Programming Constructs

For improved efficiency and security, the following Groovy programming constructs are not supported in Gateway:

- Loops (for example: for, for each)

> **Note**: Only while loop is supported in Gateway

- 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 Groovy scripts contain unsupported data types or other unsupported control structures.

Supported Methods

Groovy supports a variety of methods for each of its data types, as well as some additional Java methods. The following sections list Groovy supported methods for each data type. Refer to the official Java documentation for more information on each method.

String Methods

The following string methods are supported to manipulate textual data. Refer to the official Java documentation for more information on each method.

> - **Notes**:
> - Doubles are converted to Int types when calling methods that require Int arguments.
> - Types are indicated in **bold** in each method signature.

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"`

Method Signature	Description	Parameters	Example
`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"`
`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`
`String.length()`	Returns the length of the String calling the length method.	No method parameters	`"Hello".length(); //returns 5`

Method Signature	Description	Parameters	Example
`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`
`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"`

Method Signature	Description	Parameters	Example
`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	Example
`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.`

Method Signature	Description	Parameters	Example

`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

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`

`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`

Method Signature	Description	Parameters	Example
`35.4Double.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"`
`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 Methods

Method Signature	Description	Parameters	Example
`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`

Method Signature	Description	Parameters	Example

`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

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`

`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	Example
`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`

Method Signature	Description	Parameters	Example
`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`

Method Signature	Description	Parameters	Example
`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`

Method Signature	Description	Parameters	Example
`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

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`

Method Signature	Description	Parameters	Example

`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`

`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`

Method Signature	Description	Parameters	Example

`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`

Method Signature	Description	Parameters	Example
`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`

Method Signature	Description	Parameters	Example
`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

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`

Method Signature	Description	Parameters	Example

`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`

`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`

Method Signature	Description	Parameters	Example

`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`

`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 7.389056098930665`

`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`

Method Signature	Description	Parameters	Example
`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`
`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`

Method Signature	Description	Parameters	Example

`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`

`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(.005); //returns 0.0050000416670833376`
			`Math.tan(35.4); //returns 1.1212163300856046`

`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

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`

Method Signature	Description	Example

`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`

`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`

Method Signature	Description	Example

`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`

`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.`

Copyright

Oracle Primavera Gateway Groovy Scripting Guide