

COMSOL Multiphysics

Application Programming Guide



Application Programming Guide

© 1998-2024 COMSOL

Protected by patents listed on www.comsol.com/patents, or see Help > About COMSOL Multiphysics on the File menu in the COMSOL Desktop for less detailed lists of U.S. Patents that may apply. Patents pending.

This Documentation and the Programs described herein are furnished under the COMSOL Software License Agreement (www.comsol.com/sla) and may be used or copied only under the terms of the license agreement.

COMSOL, the COMSOL logo, COMSOL Multiphysics, COMSOL Desktop, COMSOL Compiler, COMSOL Server, and LiveLink are either registered trademarks or trademarks of COMSOL AB. All other trademarks are the property of their respective owners, and COMSOL AB and its subsidiaries and products are not affiliated with, endorsed by, sponsored by, or supported by those trademark owners. For a list of such trademark owners, see www.comsol.com/ trademarks.

Version: COMSOL 6.3

Contact Information

Visit the Contact COMSOL page at www.comsol.com/contact to submit general inquiries or search for an address and phone number. You can also visit the Worldwide Sales Offices page at www.comsol.com/contact/offices for address and contact information.

If you need to contact Support, an online request form is located on the COMSOL Access page at www.comsol.com/support/case. Useful links:

- Support Center: www.comsol.com/support
- Product Download: www.comsol.com/product-download
- Product Updates: www.comsol.com/product-update
- COMSOL Blog: www.comsol.com/blogs
- Discussion Forum: www.comsol.com/forum
- Events: www.comsol.com/events
- COMSOL Video Gallery: www.comsol.com/videos
- Support Knowledge Base: www.comsol.com/support/knowledgebase
- · Learning Center: https://www.comsol.com/support/learning-center

Part number: CM020012

Contents

Introduction
Syntax Primer
Data Types
Declarations
Built-in Elementary Math Functions
Control Flow Statements
Important Programming Tools
Ctrl+Space for Code Completion21
Recording Code23
Model Methods and Application Methods26
Global Methods, Form Methods, and Local Methods 27
Method Names27
The Java Shell and Data Viewer Windows
The Chatbot Window
Chatbot Functionality in the Method Editor
Introduction to the Model Object
Model Object Tags37
Creating a Model Object
Creating Model Components and Model Object Nodes 40
Get and Set Methods for Accessing Properties41
Parameters and Variables48
Unary and Binary Operators in the Model Object50
Geometry
Mesh53
Physics

Material
Study
Results63
Multiphysics
Working with Model Objects
The Model Object Class Structure
The Application Object76
Shortcuts
Accessing the Application Object
The Name of User Interface Components
Important Classes
Get and Set Methods for the Color of a Form Object 79
General Properties
The Main Application Methods82
Main Window
Form
Form Object
ltem
Data Source
AppEvent Class
AppEventHandler Class
Method Class
Form, Form Object, and Item List Methods
The Built-in Method Library for the Application Builder I I 8
Model Utility Methods
License Methods
File Methods
Operating System Methods

	Email Methods
	Email Class Methods
	GUI-Related Methods
	GUI Command Methods
	Debug Methods
	Methods for External C Libraries
	Progress Methods
	Date and Time Methods
	Conversion Methods
	Array Methods
	String Methods
	Collection Methods
	Model Builder Methods for Use in Add-ins
Pr	ogramming Examples
	Running the Examples
	Visualization Without Solution Data: Grid Datasets178
	Visualization of Points, Curves, and Surfaces
	Reading and Writing Data to File
	Converting Interpolation Curve Data
	Plotting Points on a Parametric Surface
	Defining a Parametric Sweep
	Using Selections
	Measuring Model Quantities
	Using Numerical Results in a Model or Application 240
	Getting Numerical Data240
	Recursion and Recursively Defined Geometry Objects275
	Mesh Information and Statistics

Accessing System Matrices and Vectors	283
Data Validation	288
Using Selections in Add-ins	300
Using Built-In Methods from an External Java Library	306
Measuring the Java Heap Space Memory	307
Time-Limited and Hardware-Locked Applications	307
Get and Set of 3D Camera Parameters	312
Index	317

Introduction

This book is a guide to writing code for COMSOL[®] models and applications using the Method Editor. The Method Editor is an important part of the Application Builder and is available in the COMSOL Desktop[®] environment in the Windows[®] version of COMSOL Multiphysics. For an introduction to using the Application Builder and its Form Editor and Method Editor, see the book *Introduction to Application Builder*.

Writing a method is needed when an action is not already available in the standard run commands associated with functionality in the model tree nodes of the Model Builder. A method may, for example, contain loops, process inputs and outputs, and send messages and alerts to the user of the application.

In the Model Builder, the model tree is a graphical representation of the data structure that represents a model. This data structure is called the model object and stores the state of the underlying COMSOL Multiphysics model that is embedded in an application.

The contents of the application tree in the Application Builder is accessed through the application object, which is an important part of the model object. You can write code using the Method Editor to directly access and change the user interface of a running application, for example, to update button text, icons, colors, and fonts.

In the COMSOL Multiphysics environment, you use the Java[®] programming language to write methods, which means that you can utilize the extensive collection of Java[®] libraries. In addition to the Java[®] libraries, the Application Builder includes a built-in library for building applications and modifying the model object. A number of tools and resources are available to help you automatically create code for methods. For more information on autogeneration of code, see the book *Introduction to Application Builder*.

This book assumes no prior knowledge of the Java[®] programming language. However, some familiarity with a programming language is helpful.

Syntax Primer

If you are not familiar with the Java[®] programming language, read this section to quickly get up to speed with its syntax. When creating applications, it is useful to know the basics of Java such as how to use the if, for, and while control statements. The more advanced aspects of Java will not be covered in this book. For more detail, see any dedicated book on Java programming or one of the many online resources. You can also learn a lot by reviewing the methods in the example applications available in the Application Libraries. Note that the Method Editor supports Java 11 syntax.

Data Types

PRIMITIVE DATA TYPES

Java contains eight primitive data types, listed in the table below.

DATA TYPE	DESCRIPTION	NUMBER OF BYTES	EXAMPLE
byte	Integer between -127 and 128		byte b=33;
char	Unicode character; integer between 0 and 65535 (0 and 2 ¹⁶ -1)	2	char c='a'; char c=97;
short	Integer between -32768 and 32767 (-2 15 -1 and 2 15 -1)	2	short s=-1025;
int	Integer between -2 ³¹ and 2 ³¹ -1	4	int i=15;
long	Integer between -2 ⁶³ and 2 ⁶³ -1	8	long I=15;
float	32-bit floating point number	4	float f =4.67f;
double	64-bit floating point number	8	double d=4.67;
boolean	Boolean with values false or true	N/A	boolean b=true;

Other data types such as strings are classes, which are also referred to as composite data types.

In methods, you can use any of the primitive or composite data types available in Java and the Java libraries. Many of the Application Builder built-in methods make use of primitive or composite data types. For example, the timeStamp() method provides a long integer as its output.

Assignments and Literals

A few examples of using literals in assignments are:

int i=5; // initialize i and assign the value 5 double d=5.0; // initialize d and assign the value 5.0 boolean b=true; // initialize b and assign the value true

The constants 5, 5.0, and true are literals. Java distinguishes between the literals 5 and 5.0, where 5 is an integer and 5.0 is a double (or float).

UNARY AND BINARY OPERATORS IN METHODS (JAVA SYNTAX)

You can perform calculations and operations using primitive data types just like with many other programming languages. The table below describes some of the most common unary and binary operators used in Java code.

PRECEDENCE LEVEL	SYMBOL	DESCRIPTION
1	++	unary: postfix addition and subtraction
2	++ + - !	unary: addition, subtraction, positive sign, negative sign, logical not
3	* / %	binary: multiplication, division, modulus
4	+ -	binary: addition, subtraction
5	!	Logical NOT
6	< <= > >=	comparisons: less than, less than or equal, greater than, greater than or equal
7	== !=	comparisons: equal, not equal
8	&&	binary: logical AND
9		binary: logical OR
10	?:	conditional ternary
	= += -= *= /= %= >>= <<= &= ^= =	assignments
12	,	element separator in lists

TYPE CONVERSIONS AND TYPE CASTING

When programming in Java, conversion between data types is automatic in many cases. For example, the following lines convert from an integer to a double:

```
int i; // initialize i double d; //initialize d i=41; d=i; // the integer i is assigned to the double d and d is 41.0
```

However, the opposite will not work automatically (you will get a compilation error). Instead you can use explicit type casting as follows:

```
int i; // initialize i
double d; //initialize d
d=41.0;
i=(int) d; // the double d is assigned to the integer i and i is 41
```

You can convert between integers and doubles within arithmetic statements in various ways, however you will need to keep track of when the automatic type conversions are made. For example:

```
int i; // initialize i
double d; //initialize d
i=41;
d=14/i; // d is 0
```

In the last line, 14 is seen as an integer literal and the automatic conversion to a double is happening after the integer division 14/41, which results in 0.

Compare with:

int i; // initialize i
double d; //initialize d
i=41;
d=14.0/i; // d is 0.3414...

In the last line, 14.0 is seen as a double literal and the automatic conversion to a double is happening before the division and is equivalent to 14.0/41.0.

You can take charge over the type conversions with explicit casting by using the syntax (int) or (double):

```
int i; // initialize i
double d,e; //initialize d and e
i=41;
d=((int) 14.0)/i; // d is 0
e=14/((double) i); // e is 0.3414...
```

STRINGS AND JAVA OBJECTS

The String data type is a Java object. This is an example of how to declare a string variable:

```
String a="string A";
```

When declaring a string variable, the first letter of the data type is capitalized. This is a convention for composite data types (or object-oriented classes).

After you have declared a string variable, a number of methods are automatically made available that can operate on the string in various ways. Two such methods are concat and equals as described below, but there are many more methods available in the String class. See the online Java documentation for more information.

Concatenating Strings

To concatenate strings, you can use the method concat as follows:

```
String a = "string A";
String b = " and string B";
a.concat(b);
```

The resulting string a is "string A and string B". From an object-oriented perspective, the variable a is an instance of an object of the class String. The method concat is defined in the String class and available using the a.concat() syntax.

Alternatively, you can use the + operator as follows:

a = a + b;

which is equivalent to:

a = "string A" + " and string B";

and equivalent to:

a = "string A" + " " + "and string B";

where the middle string is a string with a single whitespace character.

Comparing Strings

Comparing string values in Java is done with the equals method and not with the == operator. This is due to the fact that the == operator compares whether the strings are the same when viewed as class objects and does not consider their values. The code below demonstrates string comparisons:

```
boolean streq = false;
String a = "string A";
String b = "string B";
streq = a.equals(b);
// In this case streq == false
streq = (a == b);
// In this case streq == false
b = "string A";
streq = a.equals(b);
// In this case streq == true
```

Special Characters

If you would like to store, for example, a double quotation mark or a new line character in a string you need to use special character syntax preceded by a

backslash ($\$). The table below summarizes some of the most important special characters.

SPECIAL CHARACTER	DESCRIPTION
\'	Single quotation mark
\ "	Double quotation mark
11	Backslash
\t	Tab
\b	Backspace
\r	Carriage return
\f	Form feed
\n	Newline

Note that in Windows the new line character is the composite $\n\n$ whereas in Linux and macOS \n is used.

The example below shows how to create a string in Windows that you later on intend to write to file and that consists of several lines.

```
String contents = "# Created by me\r\n"
+"# Version 1.0 of this file format \r\n"
+"# Body follows\r\n"
+"0 1 \r\n"
+"2 3\r\n"
+"4 5\r\n";
```

The string is here broken up into several lines in the code for readability. However, the above is equivalent to the following:

```
\label{eq:string} String \ contents = "# \ Created \ by \ me\r\n# \ Version \ 1.0 \ of \ this \ file \ format \ \n# \ Body \ follows\r\n0 \ 1 \ \n^2 \ \n^7;
```

which is clearly less readable.

Some Useful String Methods

The following table contains some of the most commonly used built-in methods for the String class:

METHOD	DESCRIPTION
<pre>int length()</pre>	Returns the length of the string, for example: int len=str.length();
boolean equals(Object obj)	Compares two strings for equality, for example: boolean streq = a.equals(b)

METHOD	DESCRIPTION
<pre>boolean equalsIgnoreCase(String anotherString)</pre>	Compares two strings, ignoring case differences, for example: boolean streq = a.equalsIgnoreCase(b)
boolean startsWith(String prefix)	Checks if a string starts with a specified prefix, for example: boolean stw = a.startsWith(b)
<pre>boolean endsWith(String suffix)</pre>	Checks if a string ends with a specified suffix
<pre>int indexOf(String str)</pre>	Returns the index of the first occurrence of a specified substring
<pre>int lastIndexOf(String str)</pre>	Returns the index of the last occurrence of a specified substring
<pre>String substring(int beginIndex)</pre>	Returns a substring from the specified index to the end
String substring(int beginIndex, int endIndex)	Returns a substring within a specified range
<pre>String strip()</pre>	Removes leading and trailing whitespace
<pre>String stripLeading()</pre>	Removes leading whitespace
<pre>String stripTrailing()</pre>	Removes trailing whitespace
String concat(String str)	Concatenates a string
<pre>boolean isEmpty()</pre>	Checks if the string is empty (length() == 0)
boolean isBlank()	Checks if the string is empty or contains only whitespace

In addition, the following string conversion methods are built-in to the COMSOL API:

METHOD	DESCRIPTION
boolean toBoolean(String str)	Converts a string to a boolean
<pre>int toInt(String str)</pre>	Converts a string to an int
<pre>double toDouble(str)</pre>	Converts a string to a double
<pre>String toString(boolean b)</pre>	Converts a boolean to a string
<pre>String toString(int i)</pre>	Converts an int to a string
<pre>String toString(double d)</pre>	Converts a double to a string

Note that in Java, you can convert a string to a double using the standard double parseDouble(String s) method. However, the toDouble method provided by the COMSOL API offers additional functionality. Specifically, it can handle arithmetic expressions that evaluate to constants. These expressions can include:

- Built-in numerical constants: eps, i, j, inf, Inf, NaN, nan, pi.
- Elementary functions such as sin, exp, gamma, max, min.
- Certain operators that make sense in a constant-evaluation context, such as error, if, isinf, isnan, nif, realdot.

ARRAYS

In the application tree, the **Declarations** node directly supports 1D and 2D arrays of type string (String), integer (int), Boolean (boolean), or double (double). A 1D array may be referred to as a vector and a 2D array referred to as a matrix, provided that the array is rectangular. A nonrectangular array is called jagged or ragged. In methods, you can define higher-dimensional arrays as well as arrays of data types other than string, integer, Boolean, or double.

ID Arrays

If you choose not to use the **Declarations** node to declare an array, then you can use the following syntax in a method:

double dv[] = new double[12];

This declares a double array of length 12.

The previous line is equivalent to the following two lines:

```
double dv[];
dv = new double[12];
```

When a double vector has been declared in this way, the value of each element in the array will be zero.

To access elements in an array you use the following syntax:

```
double e;
e = dv[3]; // e is 0.0
```

Arrays are indexed starting from 0. This means that dv[0] is the first element of the array in the examples above, and dv[11] is the last element.

You can simultaneously declare and initialize the values of an array by using curly braces:

double dv[] = $\{4.1, 3.2, 2.93, 1.3, 1.52\};$

In a similar way you can create an array of strings as follows:

String sv[] = {"Alice", "Bob", "Charles", "David", "Emma"};

2D Arrays

2D rectangular arrays can be declared as follows:

```
double dm[][] = new double[2][3];
```

This corresponds to a matrix of doubles with 2 rows and 3 columns. The row index comes first.

You can simultaneously declare and initialize a 2D array as follows:

double dm[][] = {{1.32, 2.11, 3.43}, {4.14, 5.16, 6.12}};

where the value of, for example, dm[1][0] is 4.14. This array is a matrix since it is rectangular (it has same number of columns for each row). You can declare a ragged array as follows:

double dm[][] = {{1.32, 2.11}, {4.14, 5.16, 6.12, 3.43}};

where the value of, for example, dm[1][3] is 3.43.

Copying Arrays

Note that when copying arrays, this code:

```
for (int i1 = 0; i1 <= 11; i1++) {
  for (int i2 = 0; i2 <= 2; i2++) {
    input_array[i1][i2] = init_input_array[i1][i2];
  }
}</pre>
```

is not equivalent to:

input_array = init_input_array;

The latter only assigns the same reference, meaning both variables will point to the exact same array in memory.

Instead, you can use the copy method:

```
input_table = copy(init_input_table);
```

This call creates a new array under the hood and then copies the values into it.

Declarations

Variables defined in the **Declarations** node in the application tree are directly available as global variables in a method and need no further declarations.

✓ ▲ helical_sta	tic_mixer.mph (roo	t)	
🖶 Inputs	;		
🔁 Them	es		
> 🔄 Main 🛛	Window		
> 🔁 Forms	;		
🖳 Events	5		
✓	rations		
abc Sti	ring		
8.5 Do	ouble		
⊡ Bo	olean		
	umber of blades Lis	t {numberOfBladesChoicelist}	
;— ⊃n > ि— Metho	oncuis		
> IIII Librari	ies		
Settings			•
String			
List of Variables	5		
* Name	Initial value	Description	
solutionState	nosolution	Solution state	
device	computer	Preferred device	
material	steel	Material	
			_

Variables declared in methods will have local scope unless you specify otherwise. The **Declarations** node directly supports integers (int), doubles (double), and Booleans (boolean). In addition, strings are supported (see "Strings and Java Objects" on page 10). In the **Declarations** node, variables can be scalars, 1D arrays, and 2D arrays.

To simplify referencing form objects as well as menu, ribbon, and toolbar items by name, you can create shortcuts with a custom name. These names are available in the **Declarations** node under **Shortcuts**. They are directly available in methods along with the other global variables defined under **Declarations**. For more information on shortcuts, see "Shortcuts" on page 76.

In addition, under the **Declarations** node, you can add declarations for: **Choice List**, **File**, **Unit Set**, **File Type**, and **Graphics Data**. For more information on these types of declarations, see also the *Introduction to Application Builder*.

FORM DECLARATIONS

Variables can also be defined as **Form Declarations** under each respective form node in the application tree.



Form declarations can be of the types **Scalar**, **Array 1D**, **Array 2D**, and **Choice List**. Global declarations are exposed to all user-interface components of the application whereas form declarations are only exposed to the form that they are defined in and the form objects within that form. Form declarations are used to limit the scope of variables and thereby logically separate the different parts of an application.

Built-in Elementary Math Functions

Elementary math functions for use in methods are available in the Java math library. Some examples:

```
double a = Math.PI; // the mathematical constant pi
double b = Math.sin(3*a); // trigonometric sine function
double c = Math.cos(4*a); // trigonometric cosine function
double d = Math.random(); // random number uniformly distributed in [0,1)
double e = Math.exp(2*a); // exponential function
double f = Math.log(1+e); // natural base e logarithm
double g = Math.pow(10,3); // power function
double h = Math.log10(2.5); // base 10 logarithm
double k = Math.sqrt(81.0); // square root
```

There are several more math functions available in the Java math library. For additional information, see any Java book or online resource.

Control Flow Statements

Java supports the usual control flow statements if-else, for, and while. You can use the **Language Elements** tool to insert template code corresponding to a number of control flow, of or block, statements.



The following examples illustrate some of the most common uses of control flow statements.

THE IF-ELSE STATEMENT

This is an example of a general if-else statement:

```
if (a < b) {
    alert("Value too small.");
} else {
    alert("Value is just right.");
}</pre>
```

Between curly braces {} you can include multiple lines of code, each terminated with a semicolon. If you only need one line of code, such as in the example above, this shortened syntax is available:

```
if (a < b)
    alert("Value too small.");
else
    alert("Value is just right.");</pre>
```

THE FOR STATEMENT

Java supports several different types of for statements. This example uses the perhaps most conventional syntax:

```
// Iterate i from 1 to N:
int N = 10;
for (int i = 1; i <= N; i++) {
    // Do something
}
```

In an alternative syntax, shown in the example on page 86, the loop is over all form objects in a list of form objects:

```
for (FormObject formObject : app.form("form1").formObject()) {
   if ("Button".equals(formObject.getType())) {
     formObject.set("enabled", false);
   }
}
```

where the local iteration variable looped over is formObject of the type, or class, FormObject. The collection of objects, in this case

app.form("form1").formObject(), can be an array or other types of lists of objects. Using this syntax, the iteration variable loops over all entries in the collection, from start to finish. Another example can be found on page 117.

THE WHILE STATEMENT

This example shows a while statement.

```
double t = 0, h = 0.1, tend = 10;
while(t < tend) {
    // do something with t
    t = t + h;
}
```

For a more advanced example of a while statement, see "Creating and Removing Model Tree Nodes" on page 56.

Note that Java also supports do-while statements.

THE WITH STATEMENT

When writing methods in the Method Editor, in addition to the standard Java control flow statement, there is also an optional with statement, specific to the Method Editor, that can be used to make Application Builder code more compact and easier to read (you enable this in **File** > **Preferences**). A simple example is shown below:

```
// Set the global parameter L to a fixed value
with(model.param());
   set("L", "10[cm]");
endwith();
```

The code above is equivalent to:

model.param().set("L", "10[cm]");

In this case using the with statement has limited value since just one parameter is assigned but for multiple assignments readability increases. See "Parameters and Variables" on page 48 for an example with multiple assignments.

Note that the with statement is only available when writing code in the Method Editor. It is not available when using the COMSOL API for use with Java[®]. You can turn off the use of with statements in the section for **Methods** in **Preferences**.

The method descr returns the variable description for the last parameter or variable in a with statement:

```
with(model.param());
   set("L", "10[cm]");
   String ds = descr("L");
endwith();
```

Assuming that the parameter description of the parameter L is Length. The string ds will have the value Length.

EXCEPTION HANDLING

An exception is an error that occurs at runtime. The Java[®] programming language has a sophisticated machinery for handling exceptions and each exception generates an object of an exception class. The most common way to handle exceptions is by using try and catch, as in the example below.

```
double d[][] = new double[2][15];
try {
    d = readMatrixFromFile("common:///my_file.txt");
} catch (Exception e) {
    error("Cannot find the file my_file.txt.");
}
```

where an error dialog is shown in case the file my_file.txt is not found in the application file folder common.

To inform the user of the underlying cause you can use an additional input argument to the error method, as shown in the example below.

```
double d[][] = new double[2][15];
try {
    d = readMatrixFromFile("common:///my_file.txt");
} catch (Exception e) {
    error("Cannot find the file my_file.txt.",e);
}
```

This can be used to "wrap" native COMSOL Multiphysics error messages with custom error messages. See the Java[®] documentation for more information about using try and catch.

The Application Builder includes several tools for automatically generating code as well as debugging. These tools include code completion, **Record Method**, **Record Code**, **Convert to New Method**, **Editor Tools**, **Language Elements**, **Copy as Code to Clipboard**, **Call Stack**, and **Variables**, and are described in the book *Introduction to Application Builder*. These utilities allow you to quickly get up and running with programming tasks even if you are not familiar with the syntax.

The following sections describes two of the most important tools: code completion using Ctrl+Space and **Record Code**. Using these tools will make you more productive, for example, by allowing you to copy-paste or autogenerate blocks of code.

Ctrl+Space for Code Completion

While typing code in the Method Editor, the Application Builder can provide suggestions for code completions. The list of possible completions are shown in a separate completion list that opens while typing. In some situations, detailed information appears in a separate window when an entry is selected in the list. Code completion can always be requested with the keyboard shortcut Ctrl+Space. Alternatively Ctrl+/ can be used to request code completion, which is useful if Ctrl+Space is in use by the Windows operating system such as for certain languages. When accessing parts of the model object, you will get a list of possible completions, as shown in the figure below:



Select a completion by using the arrow keys to choose an entry in the list and double-click, or press the Tab or Enter key, to confirm the selection.

If the list is long, you can filter by typing the first few characters of the completion you are looking for.

For example, if you enter the first few characters of a variable or method name, and press Ctrl+Space, the possible completions are shown:



In the example above, only variables that match the string iv are shown. This example shows that variables local to the method also appear in the completion suggestions.

You can also use Ctrl+Space to learn about the syntax for the built-in methods that are not directly related to the model object. Type the name of the command and use Ctrl+Space to open a window with information on the various calling signatures available.



Additional information is also available in the form of tool tips that are displayed when hovering over the different parts of the code.

The Method Editor also supports code completion for properties, including listing the properties that are available for a given model object feature node, and providing a list of allowed values that are available for a given property.

The figure below shows an example of code completion for the mesh element size property, where a list of the allowed values for the predefined element sizes is presented.



COMSOL Multiphysics and its add-on modules contain thousands of physics features that you can learn about by using, for example, **Record Code**, **Save as** >

Model File for Java, and code completion. The figure below shows code completion for a particular feature in the **Electric Currents** interface.



Recording Code

Click the **Record Code** button in the **Code** section of the Method Editor ribbon to record a sequence of operations that you perform using the model tree, as shown in the figure below.



Certain operations in the application tree can also be recorded, for example, code that changes the color of a text label in a running application may be generated.

To record a new method, click the **Record Method** button in the **Main** section of the **Home** tab.



While recording code, the COMSOL Desktop windows are surrounded by a red frame:





24 |

To stop recording code, click one of the **Stop Recording** buttons in the ribbon of either the Model Builder or the Application Builder.



By using **Data Access**, you selectively access various settings in the Model Builder, from the Application Builder. For example, you can set the values of the **Heat transfer coefficient** and the **External temperature** properties of the busbar tutorial model used in the documents *Introduction to COMSOL Multiphysics* and *Introduction to Application Builder*.



To generate similar code using **Record Code** (**Data Access** is not used when recording code), follow these steps:

- Create a simple application based on the busbar model (MPH file).
- In the Model Builder window, in the **Developer** tab, click **Record Method**, or with the Method Editor open, click **Record Code**.

- Change the value of the **Heat transfer coefficient** to 5.
- Change the value of the External temperature to 300[K].
- Click Stop Recording.
- If it is not already open, open the method with the recorded code.

The resulting code is listed below:

```
model.component("comp1").physics("ht").feature("hf1").set("h", 5);
model.component("comp1").physics("ht").feature("hf1").set("Text",
"300[K]");
```

To generate code corresponding to changes to the application object, use **Record Code** or **Record Method**, then go to the Form Editor, and, for example, change the appearance of a form object. The following code corresponds to changing the color of a text label from the default **Inherit** to **Blue**:

```
app.form("form1").formObject("textlabel1").set("foreground", "blue");
```

Built-in methods that changes the application object are only available when running applications and not when running methods from the Model Builder.

Use the tools for recording code to quickly learn how to interact with the model object or the application object. The autogenerated code shows you the names of properties, parameters, and variables. Use strings and string-number conversions to assign new parameter values in model properties. By using **Data Access** while recording, you can, for example, extract a parameter value using get, process its value in a method, and save it back into the model object using set. For more information on **Data Access**, see the *Introduction to Application Builder*.

Model Methods and Application Methods

Methods called from the Model Builder are referred to as model methods, while those called from an application are referred to as application methods. When the context is clear, this distinction is often omitted, and they are simply called methods. Model methods directly modify the model object represented in the Model Builder during the current session and are typically used to automate modeling tasks that would otherwise involve several manual steps. Application methods, on the other hand, modify the embedded model within the application. When you click Test Application, the methods executed by the application do not modify the model currently seen in the Model Builder, but instead operate on a copy of the model—the embedded model of the application.

For example, a typical use of a model method is to solve multiple studies in sequence. In a model with several studies, you can record code that first computes

Study 1, then Study 2, which may depend on the results of Study 1, and so on. This recorded method can then be called to automate the entire process.

To organize a customized workflow in the Model Builder you can create an addin based on methods by using a **Method Call**, **Settings Form**, or a ribbon tab. For an introductory example of using methods from the Model Builder and for information on how to create add-ins, see the *Introduction to Application Builder*.

You can learn how to write methods to manipulate the model object by reading the method code in the add-ins that are available in the Add-in Libraries. To load an MPH file from these libraries, browse to the addins folder in the installation folder. This is typically located at:

```
C:\Program Files\COMSOL\COMSOL63\COMSOL_Multiphysics\addins
```

In these examples you can learn how to identify a model component, how to check the spatial dimension of a model component, how to work with selections, and more.

Global Methods, Form Methods, and Local Methods

There are global methods, form methods, and local methods. Global methods are displayed in the application tree and are accessible from all methods and form objects. Form methods are displayed in the application tree as child nodes to the form it belongs to. A local method is associated with a form object or event and can be opened from the corresponding **Settings** window.

Global methods are exposed to all components of the application whereas form methods are only exposed to the form that they are defined in and the form objects within that form. You can use form methods to provide a logical separation of the different parts of an application.

Method Names

A method name has to be a text string without spaces. The string can contain letters, numbers, and underscores. Java® programming language keywords cannot be used. The name must not begin by a number (this is also true for the name of a form object, variable, and method).

A global method cannot have the same name as a form method and vice versa. In addition, the following names are reserved since they are used as names of local methods:

- onActivate
- onClick
- onClose
- onDataChange
- onEvent
- onFocusGained
- onFocusLost
- onLoad
- onPickingChanged
- onStartup
- onShutdown

The Java Shell and Data Viewer Windows

The **Java Shell** window is an interactive environment that provides a command prompt for running Java code, supporting all the features of the Method Editor such as code completion, syntax highlighting, and more. This functionality can be used to quickly modify a model through the COMSOL API for Java, without needing to create a method first. It can be used to, for example, prototype code for methods, bulk create features, inspect properties, or change declaration variables. It is also useful for debugging methods, allowing you to perform code evaluations while stopped at a breakpoint.

The **Java Shell** window can be accessed from the Model Builder, Application Builder, and Model Manager, by selecting it from the **Windows** menu in the **Home**

tab. The figure below shows the Java Shell window used to debug the buildGeometry method in the Transmission Line Calculator tutorial app.

	🕼 🕼 🔚 🗴 🕺 🖏 🙀 🦕 🕫 transmission_line_calculate.reph - COMSOL Maliphysics	- a ×
File Home Method		2
Ubiity Cless Lenge Leng	argebrane (by ∰ 1 → 0 β - C t ² (b) → 1→ L (c) 1 = 1 H H B → 1 → 1 → 1 → 1 → 1 → 1 → 1 → 1 → 1 →	
Lance La La	Control Data The Prove Static State of the particle	v to the second
sendimal	 gettigge() Bounding bits calculates (pmin, smar, sprin, s	
> 💼 utanes	> model.eson(eson).est	
	 Bon (m + 1) 	
	* 100 U 100 U 1	
	1.77 GB 12.27 GB	

The **Data Viewer** window is used to display and edit parameters, declarations, and Java variables when stopped at a breakpoint. The **Data Viewer** window is always accessible, including in the Model Builder workspace, allowing for quick viewing and updating of parameters and declarations no matter where you are in the user interface. The window also shows Java variables from the **Java Shell** window. The Java Shell window can also be used to define temporary Java variables, as illustrated in the figure below where the temporary integer variables a and b are initialized and used.

This functionality can be helpful when debugging code. Note that all Application Builder variables as well as temporary **Java Shell** window variables are displayed in the **Data Viewer** window, as shown in the figure below.

a 🗅 🕼 🗟 🔌 🕨 🗠 🐇	9 10 9 0 8 8 8 6	⊂ transmission_line,	calculator.mph - CC	MSOL Multip	hysics												- 0	. >
File Home Method																		
Andel Manager Form v Method v 19	Data Access Complex Access Record Method Complex	R Add-in Definition	Application Argument	Event	abc and a scalar Ar	abc ny Amy 1+ 20 +	More Declarations +	Images Sounds Files	More Libraries ~	Main Window	First Application	(‡) Apply Changes	Test in Web Browner v	compare Compare	Compane with Saved	Windows v	Tile -	tep
Workspace	Main	Add-in	Inputs	Events		Declaratio	11	Librar	ries	Main Window		Test		Comp	oare		View	
	Concern Research									Company of	. Data Lila							
Application Builder * 💷	Contraction of the second	<u> </u>								- official of the	C Data view	NEI A						
	2		e Bernerià Breb							Name			Value					
Sype filter text	3 checkBadAspectRati	00								> P) Pati	ameters 1							
v 🔝 transmission_line_calculator.mph (root)	4 States and									v = Dec	larations							
teputa 1	inputs 0 if if (type.couple)("couple1")) (*** actionMenuState			, Beoureph,						
S Thernes	7 geom = "geom";							*** emailBody			"Please find the report from Transmission Line Calculator application attached."							
Main Window	*L)							+++ emailSubject			"Transmission Line Calculator results"							
- C Forms	<pre>>> else if (type.equals("tutnlesd")) (</pre>								abr.	emailTo								
- manine	12 J							+++ inputOutputState			"coasialGeometry"							
hand Marris	12 else if (type.equals("microstrip")) {							isCpwBackplane			true							
submenu	13 geom = "geom3";							+++ reportName			"rpt1"							
> 😋 Action Menu	14 [] 15] also if (type equals("contener")) (+++ solutionStateCoaxial			"neSolution"							
> 😂 Geometry Input	16 geome - Country Country Country 17 1							+++ solutionStateCoplanar			"neSelution"							
> 🧮 Physics Input	12 3							+++ solutionStateMicrostrip			"Nebeluben"							
> 🔁 Netations	11 E else (*** solutionStateTwinlead			"nobeluben"						
Simulation Information	<pre>is error("Invalid type");</pre>							+++ type			"coasial"							
visualizationSnaphics	21 b								+++ vioualizationState			"geometry"						
> computedParameters	22								> 🔄 Java Shel									
ensitialog	23 visualizationState = "geometry";							v +hi geom			, decuri,							
E enalemonulareg	24	0								123	coder		0					
> Declarations	23 model.geom(geom), run(); 24 27 useGruphics(model.geom(geom), "/visualizationGraphics/graphicsGeometrySolution");							122	123 hash			0						
 The Methods 									 III value 			byte[] Length: 5 (id=640)						
i highlight lopMenuitem	23 zoosExtents("/visu	alizationGraphics/grap	hiceGeometrySol	ution");						128 [0]			103					
highlightSubmenultem																		
showinputOutputitem	1000									123 [2]			111					
buildGeometry	100. •										122 [3]		109					
checkEadAspectFatio	Java Shell 🗙 🛛 Call Stack :								× 0.1		122 [4]		49					
deckQuesistaticApproximation	50 De 101 101 20 101 X																	
1 checkfrequencyKange	The last set at a																	
E afrect/broad	* cean																	
1 showhetations																		
a repetPerameters	sec Beou> ,Beoul,																	
computeSolution	visualizationState																	
plotSolution	de la contra d																	
createReport	- Fernery																	
aveReport 5	<pre>int a = 5;</pre>																	
H measuration	123 a> 5																	
(a) openermativesog																		
) III Libories	1nt b = a+1;																	
133 b ==> 6																		
	3																	
	- man (
										3.07 GB	3.4 68							

At the top and bottom of the **Java Shell** window are two toolbars used to access different features.



These toolbars contain the following buttons:

- Go to Node: Jumps to the corresponding node in the model tree.
- Data Viewer: Opens the Data Viewer window.
- Collapse All: Collapses all content.
- Expand All: Expands all content.
- Only List Input: Displays only the input commands in the output area.
- Display Session Content: Displays the entire session content in the output area.
- **Clear Output Area**: Clears all content from the output area; however, no shell variables are removed.

- **Reset Session**: Clears all content from the output area and removes shell variables.
- **Stop**: Stops code execution.
- **Run**: Runs code entered at the command prompt. Alternatively, you can press Enter.
- **Multiline Input**: Makes it possible to add and then run multiple lines of commands. To run the code you can click the **Run** button or press Ctrl+ Enter.
- Previous in History: Moves up in the command history.
- Next in History: Moves down in the command history.

Note that you get additional options by right-clicking the output area.

The **Java Shell** session is reset when you open or start a new model and no content is saved between sessions. To make persistent changes, you need to change the model object and then save the model or use declarations in the Application Builder.

To interact with the model tree, commands should begin with model. and continue with the appropriate methods. Code completion is available as a tooltip, suggesting possible completions. For example, if you type model.bas, the tooltip displays model.baseSystem() and model.baseSystem(String name) as options. You can also activate code completion using Ctrl+Space.

For instance, if you enter

```
model.baseSystem("mpa");
```

and run the command, the **Unit System** in the root node updates from SI to MPa.

The **Java Shell** automatically generates local shell variables for output associated with the model object. By default, these variables have the same names as the tags of the model object. For example, if you type:

```
model.geom().create("geom1",3)
```

you will see the following output:

```
geom1 ==> Geometry 1
```

A local variable named geom1 is created in the shell, allowing you to use:

```
geom1.angularUnit("deg")
```

instead of:

```
model.geom("geom1").angularUnit("deg")
```

from that point forward.

You can select text in a message, then right-click and choose **Copy** (or press Ctrl+C) to copy the text. To copy the content of any output from that input, select **Copy**

Children. You can also right-click to delete it from the output area by choosing **Delete** (or press Delete).

To re-run code, right-click and select **Run**. Additionally, you can send the message content to the input area or the **Chatbot** window by choosing **Send to Input** or **Send to Chatbot**, respectively. For a detailed view, right-click and select **Details** to open the message in a separate window that can be resized. The **Details** window supports both inputs and outputs. Input code is displayed with syntax highlighting and search functionality is provided in a panel that you open by pressing Ctrl+F.

If you encounter errors, sending them to the **Chatbot** window can help you get improved code suggestions. Note that when writing to standard output or standard error in the **Java Shell** window, the output appears in the same window. For example:

System.out.println("Foo");

However, this behavior differs when running a method created in the Method Editor. In this case, standard output and standard error are not displayed. In an upcoming version, these outputs will instead be directed to the **Debug Log** window.

The Chatbot Window

From the **Windows** menu on the **Home** toolbar, select **Chatbot** to open a window for communicating with an OpenAI or Azure OpenAI GPT model. This feature assists with generating and correcting COMSOL API for Java® code directly within COMSOL Multiphysics, as well as answering modeling questions related to the software. Using the **Chatbot** window can help with both programming tasks using the COMSOL API for Java and general modeling tasks within the COMSOL Desktop. The **Chatbot** window is available in both the Model Builder and Application Builder workspaces.

The **Chatbot** window is only available with the Windows version of the COMSOL Desktop. The COMSOL software must also be installed with the CHATBOT feature included in the installation and enabled using the Enable **Chatbot** checkbox on the **Chatbot** page in the **Preferences** window. By default, the CHATBOT feature is not included, but if it is selected and installed, the **Enable Chatbot** checkbox is selected by default.

The **Chatbot** window functionality requires an OpenAI or Azure API key. The first time that you use the **Chatbot** window you need to configure a chatbot provider. In the **Chatbot** window, click **Configure Chatbot Provider**, which opens the **Preferences** window. Enter your key in the **API key** field and click **OK**. If you are using Azure OpenAI, also enter an endpoint URL in the **Endpoint** field and a model deployment name in the **Deployment name** field. Once configured, you can

start a chatbot conversation. Note that obtaining an API key typically requires signing up for an API key payment plan. For more information, see the *Reference Manual*.

The **Chatbot** window functionality connects to an external AI system. A subscription agreement with the AI system host is needed to obtain an API key. COMSOL is not a party in this agreement, and use of the external AI system may be subject to additional fees.

It is your responsibility to ensure that your use of the AI system is in accordance with laws and policies that may apply. It is also your responsibility to review the suitability of any suggestions made by the AI system.

The **Chatbot** window includes a set of toolbar buttons and menus, as shown in the figure below. In addition, information on the number of tokens used is displayed in the **Chatbot** window's top-right corner, next to **Tokens** used:



From the list in the top-left corner of the **Chatbot** window, you can select a subject: **General**, **Programming**, and **Modeling**, as shown in the figure below.



The selected subject influences how the chatbot interprets your prompts. The chatbot understands Java syntax but not all aspects of the COMSOL API. However, you can guide the chatbot to help you write code by using the **Record Code** functionality of the Method Editor.

For example, let's use the **Chatbot** window to add annotations to the Pacemaker Electrode tutorial model which is available in the Application Libraries under **COMSOL Multiphysics>Electromagnetics** with the file name

pacemaker_electrode.mph.

If we open this model and record the creation of one annotation feature for the **Electric Potential** plot, then the following code is produced:

```
model.result("pg1").create("ann1", "Annotation");
model.result("pg1").feature("ann1").set("posxexpr", 0.001);
model.result("pg1").feature("ann1").set("posyexpr", 0.002);
model.result("pg1").feature("ann1").set("poszexpr", 0.003);
model.result("pg1").feature("ann1").set("text", "eval(V)");
```

Let's say that you would like to create 10 such annotations along a line. While having the **Programming** subject selected, the following prompt will then guide the chatbot to write functional code:

```
This is how you create an annotation in COMSOL Multiphysics:
```

```
model.result("pg1").create("ann1", "Annotation");
model.result("pg1").feature("ann1").set("posxexpr", 0.001);
model.result("pg1").feature("ann1").set("posyexpr", 0.002);
model.result("pg1").feature("ann1").set("poszexpr", 0.003);
model.result("pg1").feature("ann1").set("text", "eval(V)");
```

```
Can you create a for loop that places these annotations at 10 locations along a line from (x,y,z)=(0,0,-0.02) to (x,y,z)=(0,0,0).
```

The figure below shows the resulting output from the chatbot, which can be used to automate this task by entering the code in a method or in the **Java Shell** window.

```
Messages \times
                    Progress
                                Loa
                                                                                                                                      - D I
                                            Java Shell 	imes
                                                               Chatbot X
🖹 Programming 🔻 📴 🕂 🛅 📐
Conversation:
               This is how you create an anno... (Oct 10, 2024, 8:14:12 PM)

    Tokens used: 313 | 1 K

    This is how you create an annotation in COMSOL Multiphysics:
        model.result("pg1").create("ann1", "Annotation");
        model.result("pg1").feature("ann1").set("posxexpr", 0.001);
        model.result("pg1").feature("ann1").set("posyexpr", 0.002);
        model.result("pg1").feature("ann1").set("poszexpr", 0.003);
        model.result("pg1").feature("ann1").set("text", "eval(V)");
        Can you create a for loop that places these annotations at 10 locations along a line from (x,y,z)=(0,0,-0.02) to (x,y,z)=(0,0,0).
    for (int i = 0; i < 10; i++) {</pre>
             double z = -0.02 + i * 0.002; // Calculate z position
             String tag = "ann" + (i + 1); // Create unique tag for each annotation model.result("pg1").create(tag, "Annotation");
             model.result("pg1").feature(tag).set("posxexpr", 0.0);
             model.result("pg1").feature(tag).set("posyexpr", 0.0);
model.result("pg1").feature(tag).set("poszexpr", z);
             model.result("pg1").feature(tag).set("text", "eval(V)");
        3
Enter prompt here
⊖l
```

To run the code in the **Java Shell** window, right-click in the output area and select **Send to Java Shell**. Alternatively, you can click the **Send to Java Shell** button in the toolbar.



Then, in the Java Shell window, click Run. The result is shown in the figure below.



For more details about the Chatbot window, see the Reference Manual.

Chatbot Functionality in the Method Editor

The Method Editor includes certain chatbot functionality that can be useful for programming or debugging purposes. You can right-click in a method window and, from the **Send to Chatbot** submenu, choose one of the following options (assuming we are working with a method named method1):

• **Conversation About method1**: Attach the method to a prompt in the **Chatbot** window and then ask about some aspect of the method.

- Find Bugs in method I: Attach the method to a prompt in the Chatbot window with a suitable text for getting help with debugging.
- **Suggest Improvements to method1**: Attach the method to a prompt with a suitable text for getting suggestions for improvements.

The figure below shows these menu options for a method named compute.

Q Pre	view 🕃 changelnput 🛛 📳	com	pute ×										
1	/**												
2	2 * Computes the solution, updates the solutionsState variable, and plots the solution.												
3	*/												
4	if (icSendEmail && (emailT	ī:	Go to Node	F11									
6	alert("Specify a recipie	Ξ÷	Go to Method	Ctrl+Alt+Double-click		box")							
7	return;			carrier bouble circle			,						
8	}	∽.	Undo	Ctrl+Z									
9		0	Redo	Ctrl+Y									
10	// Temporary change the so	÷.,											
11	solutionState = "nosolutio		Zoom 100%		•								
12	// Salue alat and est as	×	Cut	Ctrl+X									
13	model study("std1") cun():		Conv	Ctrill C									
15	useGraphics(model.result("		сору	cui+c									
16	useGraphics(model.result("	th	Paste	Ctrl+V									
17	<pre>useGraphics(model.result("</pre>	m	Delete	Del									
18	<pre>useGraphics(model.result("</pre>			~									
19		3	Select All	Ctrl+A									
20	solutionState = "solutione	•	Indent and Format	Tab									
22 🕀	<pre>if (isSendEmail) {</pre>	₹ ^{a=}	Create Local Variable	Ctrl+1									
23	<pre>model.result().report("r</pre>	72	Toggle Comment	Ctrl+7									
24	sendReportByEmail();	=	loggie comment	carri									
25	}		Send to Chatbot		•	U C	onversation About compute						
26	if (icPlayTada) (S Fi	nd Bugs in compute									
28	<pre>if (ISFIGyraud) { playSound("success 1 way) </pre>												
29	}		LE <u>M</u> SU	uggest improvements to compute									
	-												
Introduction to the Model Object

The model object is the data structure that stores the state of the COMSOL Multiphysics model. The model object contents are reflected in the COMSOL Desktop user interface by the structure of the Model Builder and its model tree. The model object is associated with a large number of methods for setting up and running sequences of operations such as geometry sequences, mesh sequences, and study steps. As an alternative to using the Model Builder, you can write programs in the Method Editor that directly access and change the contents of the model object.

The model object methods are structured in a tree-like way, similar to the nodes in the model tree. The top-level methods just return references that support further methods. At a certain level the methods perform actions, such as adding data to the model object, performing computations, or returning data.

For a complete list of methods used to edit the model object, see the *Programming Reference Manual*. For an introduction to using the Model Builder, see the book *Introduction to COMSOL Multiphysics*.

The contents of the application tree in the Application Builder are accessed through the application object, which is an important part of the model object. You can write code using the Method Editor to alter, for example, button text, icons, colors, and fonts in the user interface of a running application.

This section gives an overview of the model object. The section "The Application Object" on page 76 gives an overview of the application object.

Model Object Tags

In the model tree and when working with the model object from methods, tags are used as handles to different parts of the model object. These tags can also be made visible in the Model Builder by first clicking the Model Builder toolbar menu **Model Tree Node Text** and then choosing **Tag**, as shown in the figure below.



The figures below show an example of a model tree without tags shown in the left figure and with tags shown in the right figure.



In code, the tags are referenced using double quotes. For example, in the following line

model.geom("geom1").create("r1", "Rectangle");

geom1 is a tag for a geometry object and r1 is a tag for a rectangle object. The following sections contain multiple examples of using tags to create and edit parts of a model object.

The option **Name**, available in the **Model Tree Node Text** menu in the Model Builder toolbar, represents the name used for scoping. The scope names are used to access the different parts of the model object. This is important, for example, when working with global variables for defining the constraints and objective functions

for an optimization study. In the figure below, the variables mass and maxStress are referenced by scope names: comp1.mass and comp1.maxStress, respectively.

Settings			•
Optimization			
= Compute ⊣→ Continue			
abel: Optimization			
Optimization Solver			
Objective Function			+• 🗐
Expression	Description		Evaluate for
comp1.mass	Bracket mass		Stationary {ref2}
			Eigenfrequency {ref}
A 1			
Ť ↓ ≔x ≡D ▼ None:			
Minimization			
Aultiple objectives:			
Sum of objectives			
olution:			
Use first			
Objective scaling:			
None			
Control Variables and Param	neters		
 Constraints 			+• 🖻
₩			
Expression	Lower bound	Upper bound	Evaluate for
real(ireq)	minFreq		Eigentrequency {ref} •
comp1.maxStress/maxStressLimit			Stationary {ref2}
d_O_Cmp	3[mm]		Eigenfrequency {ref} 🔹

Using scope names avoids name collisions in cases where there are multiple model components or multiple physics interfaces with identical variable names.

Creating a Model Object

If you create an application using the Model Builder and the Application Builder, then a model object model is automatically created the first time you enter the

Model Builder. This is then available as a reserved variable name. When using the Model Wizard, the creation of the model tag is automatically handled.

You can create additional model objects with calls to the createModel method. Such additional model objects will not be visible in the Model Builder (only model is).

For more information on working with several model objects, see the section "Working with Model Objects" on page 67.

Creating Model Components and Model Object Nodes

A model contains one or more model components. You can create a model component using the following command:

```
model.modelNode().create("comp1", true);
```

The second argument is a boolean that determines whether separate geometry, mesh, material, and spatial frames should be created. If unsure whether these frames will be needed, it is recommended to set this value to true to ensure they are included.

The component is given a definite spatial dimension when you create a geometry node:

```
model.geom().create("geom1", 2);
```

where the second argument can be 0, 1, 2, or 3, depending on the spatial dimension. In the example above, the spatial dimension is 2.

In addition to creating model components and geometry nodes, there are create methods for many of the nodes in the model tree.

Whether the geometry should be interpreted as being axisymmetric or not is determined by a Boolean property that you can assign as follows:

```
boolean makeaxi = true;
model.geom("geom1").axisymmetric(makeaxi);
```

The axisymmetric property is only applicable to models of spatial dimension 1 or 2.



Using the Model Wizard, if you first create a **Blank Model** and then add a component using the Model Builder, you will be prompted to choose the space dimension of the component. This operation will, in addition to creating a component, also create a **Geometry** and **Mesh** node. For example, creating a 2D component corresponds to the following lines of code:

```
model.component().create("comp1", true);
model.component("comp1").geom().create("geom1", 2);
model.component("comp1").mesh().create("mesh1");
```

Get and Set Methods for Accessing Properties

The get and set methods are used to access and assign, respectively, property values in the different parts of the model object. To assign individual elements of a vector or matrix, the setIndex method is used. The property values can be of the basic data types: String, int, double, and boolean, as well as vectors or matrices of these types (1D or 2D arrays).

The get, set, and create methods (described in the previous section) are also accessible from the model tree by right-clicking and selecting **Copy as Code to Clipboard**.

> 🚺 Mat	erials		
🗸 🝋 Hea	t Transfer in Solids (ht) {ht}		
P :	Solid 1 {solid 1}		
	Initial Values 1 {init1}		
	Thermal Insulation 1 {ins1}		
	Temperature 1 {temp 1}		
	Heat Flux 1 {hf1}		
> 🚠 Mu	Load Group	•	
> \infty Study 1	Copy as Code to Clipboard	۲	Create
> 🔎 Results	† Move Up	Ctrl+Up	🗄 Get
	🔳 Сору	Ctrl+C	🖁 😋 Set All
	🔄 Duplicate	Ctrl+Shift+D	

The get Methods

The family of get methods is used to retrieve the values of properties. For example, the getDouble method can be used to retrieve the value of the predefined element size property hauto for a mesh and store it in a variable hv:

```
double hv = model.mesh("mesh1").feature("size").getDouble("hauto");
```

See the section "Example Code" on page 46 for more information on the property hauto.

The syntax for the family of get methods for the basic data types is summarized in the following table:

TYPE	SYNTAX
String	getString(String name)
String array	getStringArray(String name)
String matrix	getStringMatrix(String name)
Integer	<pre>getInt(String name)</pre>
Integer array	getIntArray(String name)
Integer matrix	<pre>getIntMatrix(String name)</pre>
Double	getDouble(String name)
Double array	getDoubleArray(String name)
Double matrix	<pre>getDoubleMatrix(String name)</pre>
Boolean	getBoolean(String name)
Boolean array	getBooleanArray(String name)
Boolean matrix	getBooleanMatrix(String name)

All arrays are returned as copies of the data; writing to a retrieved array does not change the data in the model object. To change the contents of an array in the model object, use one of the methods set or setIndex.

Automatic type conversion is attempted from the property type to the requested return type.

The set Method

The syntax for assignment using the set method is exemplified by this line of code, which sets the title of a plot group pg1:

```
model.result("pg1").set("title", "Temperature T in Kelvin");
```

The first argument is a string with the name of the property, in the above example "title". The second argument is the value and can be a basic type as indicated by the table below.

TYPE	SYNTAX
String	<pre>set(String name,String val1)</pre>
String array	<pre>set(String name,new String[]{"val1","val2"})</pre>
String matrix	<pre>set(String name,new String[][]{{"1","2"},{"3","4"}})</pre>
Integer	<pre>set(String name,17)</pre>
Integer array	<pre>set(String name,new int[]{1,2})</pre>

TYPE	SYNTAX
Integer matrix	<pre>set(String name,new int[][]{{1,2},{3,4}})</pre>
Double	<pre>set(String name,1.3)</pre>
Double array	<pre>set(String name,new double[]{1.3,2.3})</pre>
Double matrix	<pre>set(String name,new double[][]{{1.3,2.3},{3.3,4.3}})</pre>
Boolean	<pre>set(String name,true)</pre>
Boolean array	<pre>set(String name,new boolean[]{true,false})</pre>
Boolean matrix	<pre>set(String name,new boolean[][]{{true, false},{false, false}})</pre>

Using the set method for an object returns the object itself. This allows you to append multiple calls to set as follows:

```
model.result("pg1").set("edgecolor", "black").set("edges", "on");
```

The previous line of code assigns values to both the edgecolor and edges properties of the plot group pg1 and is equivalent to the two lines:

```
model.result("pg1").set("edgecolor", "black");
model.result("pg1").set("edges", "on");
```

In this case, the set method returns a plot group object.

Yet another equivalent way is illustrated by the following example, which stores a reference to the plot group object in pg1, an instance of the ResultFeature class. To automatically determine and insert the correct type, you can click on a line of code such as:

```
model.result("pg1").set("edgecolor", "black").set("edges", "on");
```

and then press Ctrl+1. This will automatically refactor the code to:

```
ResultFeature pg1 = model.result("pg1");
pg1.set("edgecolor", "black").set("edges", "on");
```

This approach may improve readability and avoids repeated lookups of "pg1" in the model.

The same technique of chaining multiple calls to the set method also applies to other parts of the model object. For example, to set multiple properties for a circle in a single statement, you can write:

```
model.geom("geom1").create("c1","Circle").set("pos", new
double[]{0,1}).set("r", 5);
```

Alternatively, you can store the circle in a PropFeature object:

```
PropFeature circ1 = model.geom("geom1").create("c1", "Circle").set("pos",
new double[]{0,1}).set("r", 5);
```

To make a model parametric using a parameter defined under **Global Definitions**, use the following string-array syntax:

```
model.geom("geom1").create("c1","Circle").set("pos", new
String[]{"0","b"}).set("r", 5);
```

Here, b is a global parameter. This is the recommended syntax if you intend to perform a parametric sweep using a study.

Note that this syntax differ from the double-array syntax:

```
model.geom("geom1").create("c1","Circle").set("pos", new
double[]{0,c}).set("r", 5);
```

In this case, c is interpreted as a Java variable. The expression double[]{0, c} is evaluated numerically (for example, to {0, 1.3} if c is 1.3), which means the reference to c is lost and cannot be used for the built-in parametric sweep. However, if you are making the model parametric purely through Java code rather than using the built-in parametric sweep functionality, then this syntax works as intended.

Automatic type conversion is attempted from the input value type to the property type. For example, consider a model parameter a that is just a decimal number with no unit. Its value can be set with the statement:

```
model.param().set("a", "7.54");
```

where the value "7.54" is a string. In this case, the following syntax is also valid: model.param().set("a",7.54);

The setIndex Method

The setIndex method is used to assign a value to a 1D or 2D array element at a position given by one or two indices (starting from index 0).

The following line illustrates using setIndex with one index:

```
model.physics("c").feature("cfeq1").setIndex("f", "2.5", 0);
```

The following line illustrates using setIndex with two indices:

```
model.physics("c").feature("cfeq1").setIndex("c", "-0.1", 0, 1);
```

For the setIndex method in general, use one of these alternatives to set the value of a single element:

```
setIndex(String name,String value,int index);
setIndex(String name,String value,int index1,int index2);
```

The name argument is a string with the name of the property. The value argument is a string representation of the value. The indices start at 0, for example:

setIndex(name,value,2);

sets the third element of the property name to value.

The setIndex method returns an object of the same type, which means that setIndex methods can be appended just like the set method.

If the index points beyond the current size of the array, then the array is extended as needed before the element at index is set. The values of any newly created intermediate elements are undefined.

The method setIndex and set can both be used to assign values in ragged arrays. For example, consider a ragged array with 2 rows. The code statements:

```
setIndex(name,new String[]{"1","2","3"},0);
setIndex(name,new String[]{"4","5"},1);
```

sets the first and second row of the array and are equivalent to the single statement:

```
set("name",new String[][]{{"1","2","3"},{"4","5"}});
```

See the section "Example Code" below for examples of using setIndex.

METHODS ASSOCIATED WITH SET AND GET METHODS

For object types for which the set, setIndex, and get methods can be used, the following additional methods are available, exemplified by the case of a Heat Transfer in Solids physics interface:

```
// String[] properties();
String[] props =
model.component("comp1").physics("ht").feature("solid1").properties();
```

returns the names of all available properties,

```
// boolean hasProperty(String name);
boolean b =
model.component("comp1").physics("ht").feature("solid1").hasProperty("k mat");
```

returns true if the feature has the named property,

```
// String[] getAllowedPropertyValues(String name);
String[] vals =
model.component("comp1").physics("ht").feature("solid1").getAllowedPropertyValu
es("k_mat");
```

returns the allowed values for named properties, if it is a finite set. The figure below illustrates how to use the **Java Shell** window to retrieve these properties and

values. This example assumes that a heat transfer model is open in the user interface.



EXAMPLE CODE

The following code can be used to warn an application's user of excessive simulation times based on the element size:

```
if (model.mesh("mesh1").feature("size").getDouble("hauto") <= 3) {
    exp_time = "Solution times may be more than 10 minutes for finer element
    sizes.";
}</pre>
```

In the above example, getDouble is used to retrieve the value of the property hauto, which corresponds to the **Element Size** parameter **Predefined** in the **Settings** window of the **Size** node under the **Mesh** node. This setting is available when the **Sequence type** is set to **User-controlled mesh**, in the **Settings** window of the **Mesh** node.

The following code retrieves an array of strings corresponding to the legends of a 1D point graph.

```
String[] legends =
model.results("pg3").feature("ptgr1").getStringArray("legends");
```

The figure below shows an example of a vector of legends in the **Settings** window of the corresponding **Point Graph**.

 Legen 	ds
🛛 Show I	egends
Legends:	Manual 🗸
" Legend	s
a=0.02 m	
a=0.025 m	
a=0.03 m	
a=0.035 m	
a=0.04 m	

The following code sets the **Dataset** dset1 for the **Plot Group** pg1:

```
model.result("pg1").set("data", "dset1");
```

The following code set the anisotropic diffusion coefficient for a Poisson's equation problem on a block geometry.

```
model.geom("geom1").create("blk1", "Block");
with(model.geom("geom1").feature("blk1"));
set("size", new String[]{"10", "1", "1"});
endwith();
model.geom("geom1").run();
with(model.physics("c").feature("cfeq1"));
setIndex("c", "-0.1", 0, 1);
setIndex("c", "-0.2", 0, 6);
setIndex("f", "2.5", 0);
endwith();
```

The 3-by-3 diffusion coefficient matrix indices follow column-first ordering.

The following code sets the global parameter L to a fixed value.

```
model.param().set("L", "10[cm]");
```

The following code sets the material link index to the string variable alloy, defined under the **Declarations** node.

```
model.material("matlnk1").set("link", alloy);
```

The following code sets the coordinates of a cut point dataset cpt1 to the values of the 1D array samplecoords[].

```
with(model.result().dataset("cpt1"));
  set("pointx", samplecoords[0]);
  set("pointy", samplecoords[1]);
  set("pointz", samplecoords[2]);
endwith();
```

The following code sets the components of a deformation plot, applied to a volume plot, to the contents of the strings uStr, vStr, and wStr.

```
model.result("pg1").feature("vol1").feature("def").setIndex("expr", uStr,
0).setIndex("expr", vStr, 1).setIndex("expr", wStr, 2);
```

The code below sets the title and color legend of a plot group pg2 and then regenerates the plot.

```
with(model.result("pg2"));
   set("titletype", "auto");
endwith();
with(model.result("pg2").feature("surf1"));
   set("colorlegend", "on");
endwith();
model.result("pg2").run();
```

Parameters and Variables

This code defines a global parameter L with **Expression** 0.5[m] and **Description** Length:

model.param().set("L", "0.5[m]"); model.param().descr("L", "Length");

There is an alternative syntax using three input arguments:

```
model.param().set("L", "0.5[m]", "Length");
```

You can also use the with syntax to set the **Expression** and **Description** for several parameters, for example:

```
with(model.param());
  set("L", "0.5[m]");
  descr("L", "Length");
  set("wd", "10[cm]");
  descr("wd", "Width");
  set("T0", "500[K]");
  descr("T0", "Temperature");
endwith();
```

which corresponds to the following **Settings** window for **Global Definitions** > **Parameters**:

Setting	gs s			▼ □1
Label: Par	rameters 1			Ē
~ Param	eters			
* Name	Expression	Value	Description	
L	0.5[m]	0.5 m	Length	
wd	10[cm]	0.1 m	Width	
	50011/3	500 K	T	

Note the with syntax is not supported in the Java Shell window.

ACCESSING A GLOBAL PARAMETER

You would typically use the **Editor Tools** window for generating code for setting the value of a global parameter. While in the Method Editor, right-click the parameter and select **Set**.

To set the value of the global parameter L to 10 cm:

model.param().set("L", "10[cm]");

To get the global parameter L and store it in a double variable Length:

```
double Length = model.param().evaluate("L");
```

Note that if you have multiple parameter nodes, the syntax for evaluation is still the same. For example, this code sets the parameter L2 in a **Parameters 2** node, with tag par2:

model.param("par2").set("L2", "5[cm]");

To get the parameter L2 and store it in a double variable Length2:

```
double Length2 = model.param().evaluate("L2");
```

without any reference to the tag par2.

The evaluation is in these cases with respect to the base **Unit System** defined in the model tree root node.

To return the unit of the parameter L, if any, use:

```
String Lunit=model.param().evaluateUnit("L");
```

To write the value of a model expression to a global parameter, you typically need to convert it to a string. The reason is that model expressions may contain units.

Multiply the value of the variable $\tt Length$ with 2 and write the result to the parameter $\tt L$ including the unit of cm.

```
Length = 2*Length;
model.param().set("L", toString(Length) + "[cm]");
```

To return the value of a parameter in a different unit than the base **Unit System**, use:

```
double Length_real = model.param().evaluate("L","cm");
```

For the case where the parameter is complex valued, the real and imaginary parts can be returned as a double vector of length 2:

```
double[] realImag = model.param().evaluateComplex("Ex", "V/m");
```

For parameters that are numbers without units, you can use a version of the set method that accepts a double instead of a string. For example, the lines

```
double a_double = 7.65;
model.param().set("a_param", a_double);
```

assigns the value 7.65 to the parameter a_param.

VARIABLES

The syntax for accessing and assigning variables is similar to that of parameters. For example, the code:

```
with(model.variable("var1"));
   set("F", "150[N]");
   descr("F", "Force");
endwith();
```

assigns the Expression 150[N] to the variable with Name F.

The following code assigns a model expression to the variable f:

```
with(model.variable("var1"));
   set("f", "(1 - alpha)^2/(alpha^3 + epsilon) + 1");
endwith();
```

and the following code stores the model expression for the same variable in a string fs.

```
String fs = model.variable("var1").get("f");
```

Unary and Binary Operators in the Model Object

The table below describes the unary and binary operators that can be used when accessing a model object, such as the model expressions used when defining

PRECEDENCE LEVEL	SYMBOL	DESCRIPTION
1	() {} .	grouping, lists, scope
2	^	power
3	! - +	unary: logical not, minus, plus
4	[]	unit
5	* /	binary: multiplication, division
6	+ -	binary: addition, subtraction
7	< <= > >=	comparisons: less-than, less-than or equal, greater-than, greater-than or equal
8	== !=	comparisons: equal, not equal
9	&&	logical and
10		logical or
11	,	element separator in lists

parameters, variables, material properties, and boundary conditions, as well as in expressions used in results for postprocessing and visualization.

The following example code creates a variable to indicate whether the effective von Mises stress exceeds 200 MPa by using the inequality solid.mises>200[MPa]:

```
model.variable().create("var1");
model.variable("var1").model("comp1");
model.variable("var1").set("hi_stress", "solid.mises>200[MPa]");
```

The following code demonstrates using this variable in a surface plot:

```
model.result().create("pg3", "PlotGroup3D");
model.result("pg3").create("surf1", "Surface");
with(model.result("pg3").feature("surf1"));
  set("expr", "hi_stress");
endwith();
model.result("pg3").run();
```

The same plot can be created by directly using the inequality expression in the surface plot expression as follows:

```
with(model.result("pg3").feature("surf1"));
  set("expr", "solid.mises > 200[MPa]");
endwith();
model.result("pg3").run();
```

Once the **Geometry** node is created (see "Creating Model Components and Model Object Nodes" on page 40) you can add geometric features to the node. For example, add a square using default position (0, 0) and default size 1:

```
model.geom("geom1").create("sq1", "Square");
```

The first input argument "sq1" to the create method is a tag, a handle, to the square. The second argument "Square" is the type of geometry object.

Add another square with a different position and size:

```
model.geom("geom1").create("sq2", "Square");
with(model.geom("geom1").feature("sq2"));
  set("pos", new String[]{"0.5", "0.5"});
  set("size", "0.9");
endwith();
```

The with statement in the above example is used to make the code more compact and, without using with, the code statements above are equivalent to:

```
model.geom("geom1").feature("sq2").set("pos", new String[]{"0.5", "0.5"});
model.geom("geom1").feature("sq2").set("size", "0.9");
```

Take the set difference between the first and second square:

```
model.geom("geom1").create("dif1", "Difference");
with(model.geom("geom1").feature("dif1").selection("input"));
  set(new String[]{"sq1"});
endwith();
with(model.geom("geom1").feature("dif1").selection("input2"));
  set(new String[]{"sq2"});
endwith();
```

To build the entire geometry, you call the method run for the Geometry node:

```
model.geom("geom1").run();
```

The above example corresponds to the following **Geometry** node settings:



In this way, you have access to the functionality that is available in the geometry node of the model tree. Use **Record Code** or any of the other tools for automatic generation of code to learn more about the syntax and methods for other geometry operations.

REMOVING MODEL TREE NODES

You can remove geometry objects using the remove method:

```
model.geom("geom1").feature().remove("sq2");
```

Remove a series of geometry objects (circles) with tags c1, c2, ..., c10:

```
for (int n = 1; n <= 10; n = n+1) {
    model.geom("geom1").feature().remove("c"+n);
}</pre>
```

The syntax "c"+n automatically converts the integer n to a string before concatenating it to the string "c".

To remove all geometry objects:

```
for (String tag : model.geom("geom1").feature().tags()) {
  model.geom("geom1").feature().remove(tag);
}
```

However, the same can be achieved with the shorter:

```
model.geom("geom1").feature().clear();
```

In a similar way, you can remove other model tree nodes.

Mesh

The following line adds a **Mesh** node, with tag mesh1, linked to the geometry with tag geom1:

```
model.mesh().create("mesh1", "geom1");
```

You can control the mesh element size either by a preconfigured set of sizes or by giving low-level input arguments to the meshing algorithm.

The following line:

model.mesh("mesh1").autoMeshSize(6);

corresponds to a mesh with **Element size** set to **Coarse**. The argument to the method autoMeshSize ranges from 1-9, where 1 is **Extremely fine** and 9 is **Extremely coarse**.

To generate the mesh, you call the run method for the **Mesh** node:

```
model.mesh("mesh1").run();
```

Use Record Code to generate code for other mesh operations.

The code below shows an example where the global mesh parameters have been changed.

```
model.mesh("mesh1").automatic(false); // Turn off Physics-controlled mesh
with(model.mesh("mesh1").feature("size"));
   set("custom", "on"); // Use custom element size
```

```
set("hmax", "0.09"); // Maximum element size
set("hmin", "3.0E-3"); // Minimum element size
set("hgrad", "1.2"); // Maximum element growth rate
set("hcurve", "0.35"); // Curvature factor
set("hnarrow", "1.5"); // Resolution of narrow regions
endwith();
model.mesh("mesh1").run();
```

The above example corresponds to the following **Mesh** node settings:



Note that you can also set local element size properties for individual points, edges, faces, and domains. Use **Record Code** or any of the other tools for automatic generation of code to learn more about the syntax and methods for other mesh operations.

Physics

Consider analyzing stationary heat transfer in the solid rectangular geometry shown earlier. To create a physics interface, for **Heat Transfer in Solids**, use:

model.physics().create("ht", "HeatTransfer", "geom1");

The first input argument to the create method is a physics interface tag that is used as a handle to this physics interface. The second input argument is the type of physics interface. The third input argument is the tag of the geometry to which the physics interface is assigned.

To set a fixed temperature boundary condition on a boundary, you first create a TemperatureBoundary feature using the following syntax:

```
model.physics("ht").create("temp1", "TemperatureBoundary", 1);
```

The first input argument to create is a feature tag that is used as a handle to this boundary condition. The second input argument is the type of boundary condition. The third input argument is the spatial dimension for the geometric entity that this boundary condition should be assigned to. Building on the previous example of creating a 2D rectangle, the input argument being 1 means that the dimension of this boundary is 1 (that is, an edge boundary in 2D).

The next step is to define which selection of boundaries this boundary condition should be assigned to. To assign it to boundary 1 use:

```
model.physics("ht").feature("temp1").selection().set(new int[]{1});
```

To assign it to multiple boundaries, for example 1 and 3, use:

```
model.physics("ht").feature("temp1").selection().set(new int[]{1, 3});
```

To set the temperature on the boundary to a fixed value of 400 K, use:

```
model.physics("ht").feature("temp1").set("T0", "400[K]");
```

The following lines of code show how to define a second boundary condition for a spatially varying temperature, varying linearly with the coordinate y:

```
model.physics("ht").create("temp2", "TemperatureBoundary", 1);
model.physics("ht").feature("temp2").selection().set(new int[]{4});
model.physics("ht").feature("temp2").set("TO", "(300 + 10[1/m]*y)[K]");
```

The resulting model tree structure is shown in the figure below.



Use **Record Code** or any of the other tools for automatic generation of code to learn more about the syntax and methods for other physics interface features and other physics interfaces.

CREATING AND REMOVING MODEL TREE NODES

Below is a larger block of code that removes, creates, and accesses physics interface feature nodes. It uses the Iterator class and methods available in the java.util package. For more information, see the Java[®] documentation.

```
String[] flowrate = column1:
String[] Mw = column2;
iava.util.Iterator<PhysicsFeature> iterator =
    model.physics("pfl").feature().iterator();
while (iterator.hasNext()) {
  if (iterator.next().getType().equals("Inlet"))
    iterator.remove():
}
if (flowrate != null) {
  for (int i = 0; i < flowrate.length; i++) {</pre>
    if (flowrate[i].length() > 0) {
      if (Mw[i].length() > 0) {
        int d = 1 + i:
        model.physics("pfl").create("inl" + d, "Inlet");
        model.physics("pfl").feature("inl" + d).setIndex("spec", "3", 0);
        model.physics("pfl").feature("inl" + d).set("qsccm0", flowrate[i]);
        model.physics("pfl").feature("inl" + d).set("Mn", Mw[i]);
        model.physics("pfl").feature("inl" + d).selection().set(new int[]{d});
      }
   }
 }
}
```

The need to remove and create model tree nodes is fundamental when writing methods because the state of the model object is changing each time a model tree node is run. In the method above, the number of physics feature nodes are dynamically changing depending on user inputs. Each time the simulation is run, old nodes are removed first and then new nodes are added.

RETRIEVING THE TYPE OF A PHYSICS FEATURE

The Model Builder always shows the label for the model tree nodes. To get more information about each node, in the Model Builder toolbar click **Model Tree Node Text**. Then select any combination of options from the list: **Name**, **Tag**, and **Type**.

- Name: The descriptive, human-readable label shown in the user interface to help you identify and distinguish it.
- **Tag**: The unique internal identifier (an alphanumeric string) used in method code and model files to reference that feature programmatically.
- **Type**: The class of a feature, which determines its available settings, behavior, and applicable operations.

Since the **Name** can be changed by the user, and thereby vary from model to model even if the physics is identical, it is not very useful when programming using the API. Instead, the **Tag** is most frequently used, as in many of the previous examples.

However, sometimes the **Type** is also useful. It gives a human-readable description of a feature that cannot be changed by the user. For example, in the case of a **Temperature** boundary condition for a **Heat Transfer in Solids** interface, a call to

```
model.component("comp1").physics("ht").feature("hs1").getType();
```

will return the string HeatSource, regardless of which Name is displayed in the Model Builder.

Material

A material, represented in the Model Builder by a **Materials** node, is a collection of property groups, where each property group defines a set of material properties, material functions, and model inputs that can be used to define, for example, a temperature-dependent material property. A property group usually defines properties used by a particular material model to compute a fundamental quantity.

To create a Materials node:

```
model.material().create("mat1", "Common", "comp1");
```

You can give the material a name, for example, Aluminum, as follows:

```
model.material("mat1").label("Aluminum");
```

The following lines of code shows how to create a basic material property group for heat transfer:

```
with(model.material("mat1").propertyGroup("def"));
  set("thermalconductivity", new String[]{"238[W/(m*K)]"});
  set("density", new String[]{"2700[kg/m^3]"});
  set("heatcapacity", new String[]{"900[J/(kg*K)]"});
endwith();
```

The built-in property groups have a read-only tag. In the above example, the tag def represents the property group **Basic** in the model tree.



The resulting model tree and Material node settings are shown in the figure below.

Model Builder	Settings Basic					- D #
Type filter text C* 	Label: Basic Name: def					=
Pi Parameters 1	 Output Properties 					
Omponent 1 (comp 1)	Property	Variable	Expression	Unit	Size	Info
> E Definitions	Thermal conductivity	k_iso ; kii	238[W/(m*K)]	W/(m·K)	3x3	
> Materials	Density	rho	2700[kg/m^3]	kg/m ³	1x1	
Aluminum (mat1)	Heat capacity at const	Ср	900[J/(kg*K)]	J/(kg·K)	1x1	
Basic (def)						
 Image Heat Transfer in Solids (<i>ht</i>) Mesh 1 Results 						

Note that some physics interfaces do not require a material to be defined. Instead, the corresponding properties can be accessed directly in the physics interface. This is also the case if the physics model settings are changed from **From material** to **User defined**. For example, for the **Heat Transfer in Solids** interface, this setting can be found in the **Settings** window of the subnode **Solid**, in the sections **Heat Conduction**, **Solid** and **Thermodynamics**, **Solid**, as shown in the figure below.

~	 Heat Conduction, Solid 						
The	rmal conductivity:						
k	User defined 🔹						
	0	0	0				
	0	0	0	W/(m·K)			
	0	0	0				
	Full 🔹 🐳						
~	 Thermodynamics, Solid 						
Der	nsity:						
ρ	From material 🔹						
Hea	Heat capacity at constant pressure:						
C _p	C _p From material						

Use **Record Code** or any of the other tools for automatic generation of code to learn more about the syntax and methods for materials.

Study

The **Study** node in the model tree contains one or more study steps, instructions that are used to set up solvers and solve for the dependent variables. The settings

for the **Study** and the **Solver Configurations** nodes can be quite complicated. Consider the simplest case for which you just need to create a study, add a study step, and run it.

Building on the example from the previous sections regarding stationary heat transfer, let's add a **Stationary** study step.

```
model.study().create("std1"); // Study with tag std1
model.study("std1").create("stat", "Stationary");
model.study("std1").run();
```

The call to the method run automatically generates a solver sequence in a data structure model.sol and then runs the corresponding solver. The settings for the solver are automatically configured by the combination of physics interfaces you have chosen. You can manually change these settings, as shown later in this section. The data structure model.sol roughly corresponds to the contents of the **Solver Configurations** node under the **Study** node in the model tree.

All low-level solver settings are available in model.sol. The structure model.study is used as a high-level instruction indicating which settings should be created in model.sol when a new solver sequence is created.

For backward compatibility, the low-level settings in model.sol can be automatically generated when using **Record Code**. This behavior is controlled by the **Store complete solver history** checkbox, available in the **Settings** window of the each Study node.



The example below shows a slightly more detailed approach to programming a study setup, based on the stationary heat transfer example shown earlier. These instructions more closely reflect the autogenerated output produced when **Record Code** is enabled and **Store complete solver history** is selected.

First, create instances of the **Study** node (with tag std1) and a **Stationary** study step subnode:

```
model.study().create("std1");
model.study("std1").create("stat", "Stationary");
```

The actual settings that determine how the study is run are contained in a sequence of operations in the **Solution** data structure, with tag sol1, which is linked to the study:

```
model.sol().create("sol1");
model.sol("sol1").study("std1");
```

The following code defines the sequence of operations contained in sol1.

First, create a **Compile Equations** node under the **Solution** node to determine which study and study step will be used:

```
model.sol("sol1").create("st1", "StudyStep");
model.sol("sol1").feature("st1").set("study", "std1");
model.sol("sol1").feature("st1").set("studystep", "stat");
```

Next, create a **Dependent Variables** node, which controls the scaling and initial values of the dependent variables and determines how to handle variables that are not solved for:

```
model.sol("sol1").create("v1", "Variables");
```

Now create a **Stationary Solver** node. The **Stationary Solver** contains the instructions that are used to solve the system of equations and compute the values of the dependent variables.

```
model.sol("sol1").create("s1", "Stationary");
```

Add subnodes to the **Stationary Solver** node to choose specific solver types. In this example, use an **Iterative** solver:

```
model.sol("sol1").feature("s1").create("i1", "Iterative");
```

Add a Multigrid preconditioner subnode:

```
model.sol("sol1").feature("s1").feature("i1").create("mg1", "Multigrid");
```

You can have multiple **Solution** data structures in a study node (such as sol1, sol2, and so on) defining different sequences of operations. The process of notifying the study of which one to use is done by "attaching" the **Solution** data structure sol1 with study std1:

```
model.sol("sol1").attach("std1");
```

The attachment step determines which **Solution** data structure sequence of operations should be run when selecting **Compute** in the COMSOL Desktop user interface.

Finally, run the study, which is equivalent to running the **Solution** data structure sol1:

```
model.sol("sol1").runAll();
```

The resulting **Study** node structure is shown in the figure below. Note that there are several additional nodes added automatically. These are default nodes and you

can edit each of these nodes by explicit method calls. You can edit any of the nodes while using **Record Code** to see the corresponding methods and syntax used.



QUICK WAY OF USING A STUDY

An alternative for quickly using a study in method code is to use:

```
model.study(studyTag).createAutoSequences("all");
```

where studyTag equals "std1", or similar, depending on the model's configuration.

This will generate the solver sequence automatically.

To run the study, you can use:

```
model.study(studyTag).run();
```

For example, in a model with a study that has the study tag std1, the corresponding code would be:

```
model.study("std1").createAutoSequences("all");
```

```
model.study("std1").run();
```

In a typical case, when **Record Code** or **Record Method** is enabled and **Store complete solver history** is disabled, the recorded code will appear as follows:

```
model.study().create("std1");
model.study("std1").create("stat", "Stationary");
model.study("std1").feature("stat").setSolveFor("/physics/ht", true);
model.study("std1").createAutoSequences("all");
model.sol("sol1").runAll();
```

The line calling setSolveFor is used to specify whether a particular physics interface should be included in the computation. In this example, setSolveFor("/physics/ht", true) ensures that the Heat Transfer interface is enabled in the study.

MODIFYING LOW-LEVEL SOLVER SETTINGS

To illustrate how some of the low-level solver settings can be modified, consider a case where the settings for the **Fully Coupled** node are modified. This subnode controls the type of nonlinear solver used.

The first line below may not be needed depending on whether the **Fully Coupled** subnode has already been generated or not (it could have been automatically generated by code similar to what was shown above).

```
model.sol("sol1").feature("s1").create("fc1", "FullyCoupled");
SolverFeature fc1 = model.sol("sol1").feature("s1").feature("fc1");
fc1.set("dtech", "auto"); // Nonlinear method (Newton solver)
fc1.set("initstep", "0.01"); // Initial damping factor
fc1.set("minstep", "1.0E-6"); // Minimum damping factor
fc1.set("rstep", "10"); // Restriction for step-sized update
fc1.set("useminsteprecovery", "auto"); // Use recovery damping factor
fc1.set("ninsteprecovery", "0.75"); // Recovery damping factor
fc1.set("ntermauto", "tol"); // Termination technique
fc1.set("ntolfact", "1"); // Tolerance factor
fc1.set("termonres", "auto"); // Termination criterion
fc1.set("reserfact", "1000"); // Residual factor
```

For more information on the meaning of these and other low-level solver settings, see the *Solver* section of the *Programming Reference Manual*.

Changing the low-level solver settings requires that model.sol has first been created. It is always created the first time you compute a study, however, you can trigger the automatic generation of model.sol as follows:

```
model.study().create("std1");
model.study("std1").create("stat", "Stationary");
model.study("std1").showAutoSequences("sol");
```

where the call to showAutoSequences corresponds to the option Show Default Solver, which is available when right-clicking the Study node in the model tree. This can be used if you do not want to take manual control over the settings in model.sol (the solver sequence) and are prepared to rely on the physics interfaces to generate the solver settings. If your application makes use of the automatically generated solver settings, then updates and improvements to the solvers in future versions are automatically included. Alternatively, the automatically generated model.sol can be useful as a starting point for your own edits to the low-level solver settings.

CHECKING IF A SOLUTION EXISTS

When creating an application it is often useful to keep track of whether a solution exists or not. The method model.sol("sol1").isEmpty() returns a boolean and is true if the solution structure sol1 is empty. Consider an application where the solution state is stored in a string solutionState. The following code sets the state depending on the output from the isEmpty method:

```
if (model.sol("sol1").isEmpty()) {
   solutionState = "noSolution";
}
else {
   solutionState = "solutionExists";
}
```

Alternatively, solutionState can be initialized to noSolution and the following code is used to indicate a state change corresponding to the input values having changed:

```
if (solutionState.equals("solutionExists")) {
   solutionState = "inputChanged";
}
```

Almost all of the example applications in the Application Libraries use this technique.

Results

The **Results** node contains nodes for **Datasets**, **Derived Values**, **Tables**, **Plot Groups**, **Export**, and **Reports**. As soon as a solution is obtained, a set of **Plot Group** nodes are automatically created. In the example of **Heat Transfer in Solids**, when setting up such an analysis in the Model Builder, two **Plot Group** nodes are added automatically. The first one is a **Surface** plot of the **Temperature** and the second one is a **Contour** plot showing the isothermal contours. Below you will see how to set up the corresponding plots manually.

First create a 2D plot group with tag pg1:

```
model.result().create("pg1", "PlotGroup2D");
```

Change the Label of the Plot Group:

```
model.result("pg1").label("Temperature (ht)");
```

Use the dataset dset1 for the **Plot Group**:

```
model.result("pg1").set("data", "dset1");
```

Create a **Surface** plot for pg1 with settings for the color table used, the intraelement interpolation scheme, and the dataset referring to the parent of the **Surface** plot node, which is the pg1 node:

```
ResultFeature rf = model.result().create("pg1", "PlotGroup2D");
rf.create("surf1", "Surface");
rf.label("Surface");
ResultFeature surf1 = model.result("pg1").feature("surf1");
surf1.set("colortable", "ThermalLight");
surf1.set("smooth", "internal");
surf1.set("data", "parent");
```

If you want to define a variable such as rf and surf1, you don't need to know its type in advance. Simply write the code you want to assign to a variable, for example:

```
model.result().create("pg1", "PlotGroup2D");
```

Then press **Ctrl+1**, which will replace the line with something like:

```
ResultFeature var1 = model.result().create("pg1", "PlotGroup2D");
```

You can then rename the autogenerated var1 to, for example, rf.

Now create a second 2D plot group with contours for the isotherms:

```
model.result().create("pg2", "PlotGroup2D");
model.result("pg2").label("Isothermal Contours (ht)");
ResultFeature pg2 = model.result("pg2");
pg2.set("data", "dset1");
pg2.feature().create("con1", "Contour");
pg2.feature("con1").label("Contour");
ResultFeature con1 = pg2.feature("con1");
con1.set("colortable", "ThermalLight");
con1.set("smooth", "internal");
con1.set("data", "parent");
```

Finally, generate the plot for the **Plot Group** pg1:

```
model.result("pg1").run();
```

or, alternatively, using the variable defined earlier:

rf.run();

To find the maximum temperature, add a **Surface Maximum** subnode to the **Derived Values** node as follows:

First create the **Surface Maximum** node with tag max1:

```
model.result().numerical().create("max1", "MaxSurface");
```

Note that in this context the method corresponding to the **Derived Values** node is called numerical.

Next, specify the selection. In this case there is only one domain 1:

```
model.result().numerical("max1").selection().set(new int[]{1});
```

Create a **Table** node to hold the numerical result and write the output from max1 to the **Table**:

```
model.result().table().create("tbl1", "Table");
model.result().table("tbl1").comments("Surface Maximum 1 {max1} (T)");
model.result().numerical("max1").set("table", "tbl1");
model.result().numerical("max1").setResult();
```

Use **Record Code** or any of the other tools for automatic generation of code to learn more about the syntax and methods for **Results**.

Using Parameterized Solutions in Results

The code below changes the visualization of a plot group pg1 by setting the property looplevel, which controls the solution parameter, to the string variable svar.

```
model.result("pg1").set("looplevel", new String[]{svar});
model.result("pg1").run();
```

The property looplevel has a central role in accessing parameterized solutions. Its argument is a 1D string array with one index per "loop level" in a study. The different loop levels correspond to the different nested parameters in a parametric sweep with multiple parameters. Again, use **Record Code** or **Record Method** for a parametric model to learn more about this syntax.

Loading Data to Tables

By using the loadFile method you can import data into a table and then display it using a results table form object or a table surface plot. The example below demonstrates loading data from an Excel file into a table and visualizing the contents using a table surface plot. The file in this example is assumed to be imported, in an application, using a file import form object with a file declaration file1 as the **File Destination**.

```
model.result().table("tbl1").loadFile("upload:///file1", "", cells);
/*
  The string variable cells contains the spreadsheet selection to be
  imported, for example A1:J7.
  The following code creates a plot group pg1 with a table surface plot.
  This code is not needed if the embedded model already contains a table
  and a table surface plot.
  */
model.result().create("pg1", 2);
model.result("pg1").create("tbls1", "TableSurface");
```

```
model.result("pg1").feature("tbls1").set("table", "tbl1");
model.result("pg1").feature("tbls1").set("dataformat", "cells");
model.result("pg1").feature("tbls1").create("hght1", "TableHeight");
model.result("pg1").feature("tbls1").feature("hght1").set("view",
"view1");
ViewFeature cam = model.view("view1").camera();
cam.set("viewscaletype", "manual");
cam.set("xscale", "1");
cam.set("zscale", "1");
cam.set("zscale", "1");
// The following line is needed to update the plot
model.result("pg1").run();
```

Similar to before, you can use **Ctrl+1** to automatically generate the variable cam of the type ViewFeature.

Multiphysics

Some of the physics interfaces define a multiphysics analysis by themselves without being coupled to any other interface. This is the case when the physics interface is used for a coupling that is so strong that it does not easily lend itself to be separated into several physics interfaces. In other cases, a set of single physics interfaces, typically two, can be combined by the use of the **Multiphysics** node. For example, a **Joule Heating** analysis is defined as the combination of an **Electric Currents** interface and a **Heat Transfer in Solids** interface with an additional **Electromagnetic Heat Source** node under the **Multiphysics** node. The following lines of code illustrate the corresponding method calls, in a 2D case.

```
model.physics().create("ec", "ConductiveMedia", "geom1");
model.physics().create("ht", "HeatTransfer", "geom1");
model.multiphysics().create("emh1", "ElectromagneticHeatSource",
"geom1", 2);
model.multiphysics("emh1").selection().all();
model.multiphysics("emh1").set("EMHeat_physics", "ec");
model.multiphysics("emh1").set("Heat_physics", "ht");
```

For a 3D case, change "geom1",2) to "geom1",3).

When using the Model Builder to set up a **Joule Heating** analysis, nodes in addition to those shown above will be created corresponding to Joule heating in thin shells, should they exist in the model, and temperature couplings if there are multiple field variables for electric potential and temperature.

When using the Model Builder in the COMSOL Desktop interface, an embedded model with variable name model is automatically created. The embedded model has a special status. For example, the automatic code generation tools only consider the embedded model. In addition, when you save to or load from an MPH file, only the embedded model is saved or loaded. General tools include the **Save Application As** command in the Application Builder and **File > Save As**, from the **File** menu of the COMSOL Desktop environment.

However, in an application you are allowed to create and edit multiple models. Saving and loading such models is done by using the built-in methods saveModel and loadModel. An MPH file can only contain a single embedded model object.

If you need to create model objects, in addition to the embedded model, use the built-in method createModel.

To create a new model you use:

```
Model extmodel = createModel();
```

A unique model tag is created automatically and assigned to the model. If you want to explicitly control the model tag, use a syntax like:

```
Model extmodel = createModel("My_model_1");
```

where My_model_1 is a unique tag. It is recommended that you do not use the names Model1, Model2, Model3, and so on, since these names are used by the mechanism that automatically generates model tags for the embedded model when loading and saving MPH files.

The following example retrieves the model tag of the embedded model:

```
String my_modeltag = model.tag();
```

however, you rarely need to use the model tag of the embedded model object.

Instead of creating and building up the contents of a model from scratch, you can load an existing model and edit it.

For example in the Windows operating system, load a model my_model.mph from the folder C:\COMSOL_Work, by using the built-in method loadModel:

```
Model extmodel = loadModel("C:\\COMSOL_Work\\my_model.mph");
```

A unique model tag is created automatically and is assigned to the model upon load. Note the double-backslash syntax in the filename. Backslash (\setminus) is a special character in Java and the double backslash is needed in this case.

To make your application portable, you can use the file scheme syntax available in the Application Builder. Assuming you stored the MPH file in the common folder, the call to loadModel should be:

```
Model extmodel = loadModel("My_Model_1", "common:///my_model.mph");
```

In this example, the tag My_Model_1 is important because it is used to reference the model from other methods. Once loaded, the model object extmodel exists in the workspace of the current COMSOL Multiphysics or COMSOL Server session.

Note that an MPH file can only contain a single embedded model object, so there is no ambiguity about which model is being referenced when loading an MPH file.

Assume that you, in one method, have loaded the model extmodel with the tag My_Model_1, such as in the example above. The model variable extmodel is not available in other methods. In order to retrieve the model from another method use:

```
Model mymodel = getModel("MyModel_1");
```

The contents of mymodel and extmodel are the same, but these variables exist in the variable space of two different methods.

The tag My_Model_1 uniquely identified and retrieved the model object from the current COMSOL Multiphysics or COMSOL Server session.

To reset your model before running a model method, call the built-in clearModel method. For example:

```
clearModel(model);
```

Placing this at the top of your model method automatically clears the Model Builder tree contents on each run.

For a list of model utility methods, see "Model Utility Methods" on page 118.

A COMPLETE EXAMPLE IN ID

The following code listing uses all the elements of the previous sections including material definitions, geometry setup, physics configuration, and boundary conditions— and demonstrates how to set up and run a 1D stationary heat transfer simulation in a 10 cm long steel rod, focusing on its lengthwise thermal behavior. In this model, the left side of the rod is cooled while a random heat source is applied throughout the rod.

To use this code:

- Start from the Model Wizard and select a Blank Model.
- 2 Copy and paste the code into a method, in the Application Builder, or the Java Shell window.
- 3 Choose the corresponding **Run** option to start the simulation.

```
// Set up and run a 1D stationary heat transfer simulation
clearModel(model);
model.component().create("comp1", true);
model.component("comp1").geom().create("geom1", 1);
model.component("comp1").mesh().create("mesh1");
```

```
model.component("comp1").geom("geom1").create("i1", "Interval");
model.component("comp1").geom("geom1").feature("i1").setIndex("coord", 0.1, 1);
model.component("comp1").geom("geom1").run();
model.component("comp1").physics().create("ht", "HeatTransfer", "geom1");
model.component("comp1").material().create("mat1", "Common");
model.component("comp1").material("mat1").label("Steel AISI 4340");
model.component("comp1").material("mat1").propertyGroup("def").set("density",
"7850[kg/m^3]");
model.component("comp1").material("mat1").propertyGroup("def").set("heatcapacit
v", "475[J/(kq*K)]");
model.component("comp1").material("mat1").propertyGroup("def")
  .set("thermalconductivity", new String[]{"44.5[W/(m*K)]", "0", "0", "0",
"44.5[W/(m*K)]", "0", "0", "0", "44.5[W/(m*K)]"});
model.func().create("rn1", "Random");
model.func("rn1").set("type", "uniform");
model.func("rn1").set("mean", 0.5);
model.component("comp1").physics("ht").create("temp1", "TemperatureBoundary",
0);
model.component("comp1").physics("ht").feature("temp1").selection().set(1);
model.component("comp1").physics("ht").create("hs1", "HeatSource", 1);
model.component("comp1").physics("ht").feature("hs1").selection().set(1);
model.component("comp1").physics("ht").feature("hs1").set("Q0", "1e6*(1+
rn1(x))");
model.study().create("std1");
model.study("std1").create("stat", "Stationary");
model.study("std1").feature("stat").setSolveFor("/physics/ht", true);
model.study("std1").createAutoSequences("all");
model.sol("sol1").runAll();
model.result().create("pg1", "PlotGroup1D");
model.result("pg1").label("Temperature (ht)");
model.result("pg1").feature().create("lngr1", "LineGraph");
model.result("pg1").feature("lngr1").set("xdata", "expr");
model.result("pg1").feature("lngr1").set("xdataexpr", "x");
model.result("pg1").feature("lngr1").set("data", "parent");
model.result("pq1").feature("lnqr1").selection().geom("geom1", 1);
model.result("pg1").feature("lngr1").selection().set(1);
model.result("pg1").run();
model.result("pg1").run();
```

A COMPLETE EXAMPLE IN 3D

The following code listing is similar to the 1D case, but simulates heat transfer in a 3D cylinder. This example demonstrates how to set up and run a stationary 3D heat transfer simulation in a 10 cm long cylindrical steel rod with a radius of 0.5 cm. One end of the rod is cooled, while a random heat source is applied throughout its volume.

```
//\ensuremath{ Set up and run a 3D stationary heat transfer simulation
```

```
clearModel(model);
model.component().create("comp1", true);
model.component("comp1").geom().create("geom1", 3);
model.component("comp1").geom("geom1").geomRep("comsol");
model.component("comp1").mesh().create("mesh1");
model.component("comp1").geom("geom1").create("cyl1", "Cylinder");
model.component("comp1").geom("geom1").feature("cyl1").set("h", 0.1);
model.component("comp1").geom("geom1").feature("cyl1").set("r", 0.005);
model.component("comp1").geom("geom1").run();
model.component("comp1").physics().create("ht", "HeatTransfer", "geom1");
model.component("comp1").material().create("mat1", "Common");
model.component("comp1").material("mat1").label("Steel AISI 4340");
model.component("comp1").material("mat1").propertyGroup("def").set("density",
"7850[kg/m^3]");
model.component("comp1").material("mat1").propertyGroup("def").set("heatcapacit
y", "475[J/(kg*K)]");
model.component("comp1").material("mat1").propertyGroup("def")
  .set("thermalconductivity", new String[]{"44.5[W/(m*K)]", "0", "0", "0",
"44.5[W/(m*K)]", "0", "0", "0", "44.5[W/(m*K)]"});
model.func().create("rn1", "Random");
model.func("rn1").set("type", "uniform");
model.func("rn1").set("nargs", 3);
model.func("rn1").set("mean", 0.5);
model.component("comp1").physics("ht").create("temp1", "TemperatureBoundary",
2);
model.component("comp1").physics("ht").feature("temp1").selection().set(3);
model.component("comp1").physics("ht").create("hs1", "HeatSource", 3);
model.component("comp1").physics("ht").feature("hs1").selection().set(1);
model.component("comp1").physics("ht").feature("hs1").set("Q0", "1e6*(1+
rn1(x,y,z))");
model.study().create("std1");
model.study("std1").create("stat", "Stationary");
model.study("std1").feature("stat").setSolveFor("/physics/ht", true);
model.study("std1").createAutoSequences("all");
model.sol("sol1").runAll();
model.result().create("pg1", "PlotGroup3D");
model.result("pg1").label("Temperature (ht)");
model.result("pg1").feature().create("vol1", "Volume");
model.result("pg1").feature("vol1").set("colortable", "HeatCameraLight");
model.result("pg1").run();
```

```
model.result("pg1").run();
```

Note: When using the **Record Code** or **Record Method** options, additional code may be generated automatically. In the examples above, any unnecessary or extraneous code has been removed for clarity.

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

heat_transfer_in_cylindrical_rod.mph

TURNING OFF AND RESETTING THE MODEL HISTORY

When running model method code to automate modeling tasks, the stored model history can grow excessively large. The model history is used, for example, when saving to a Model M-file or Model Java-file. Depending on the repetitive nature of your code, you may want to temporarily disable history logging to keep the model history manageable. This can be done as shown below:

```
model.hist().disable();
// some code
model.hist().enable();
```

To reset the model history to an almost minimal sequence of commands that reproduces the current state of the model object, you can use:

```
model.resetHist();
```

In the File menu, this action is referred to as Compact History.

Note that model history logging is only used for model methods and is automatically turned off when running an application. Also, when history logging is disabled, certain features are not available, such as saving to a Model M-file or Java-file and using the **Copy to Clipboard** functionality.

OPTIMIZING PHYSICS FEATURE CREATION PERFORMANCE

Repeated create operations for features in the Model Builder can become slow when you add many of them. Different feature types vary in complexity—for example, the TemperatureBoundary boundary condition is relatively simple, while, for example, a Thermal Contact boundary condition performs more advanced operations. Although certain optimizations are possible, identifying and resolving bottlenecks can be time-consuming.

A common technique to speed up these processes is to temporarily disable updates, as shown below:

```
try {
    // Turn off updates
    model.disableUpdates(true);
    // Perform the required operations here,
    // typically involving repeated manipulations of physics features
} finally {
    // Important: re-enable updates before leaving the method
```

```
// The try-finally block ensures this is done even if errors occur
model.disableUpdates(false);
}
```

```
// Note: Some operations cannot be performed while updates are disabled.
```

By using this approach, certain methods can show significant reduction in processing time.

Disabling model updates can lead to unexpected side effects. For example, parameters in a physics feature, such as model inputs ending in _src, may remain invalid until the model is updated. Attempting to set any of these parameters while updates are disabled can produce errors. In addition, generated variables may be unavailable to the unit evaluator, and the equation view may show incomplete data. As soon as updates are re-enabled, the program performs a full variable refresh and returns the model to a fully functional state.

The following example uses this technique to test the speed when adding a large number of model parameters. Change the value of the disableFlag variable to toggle between the disabled and enabled states.

```
// Enable or disable updates
boolean disableFlag = true;
// Number of parameter updates to perform
int maxIterations = 2000;
// Clear model
clearTheModel(); // This may take several seconds and is not part of the time
measured
// Record start time
long startTime = timeStamp();
debugLog("Start of test.");
debugLog("Disable is: "+disableFlag);
// Disable model updates (returns previous state) to speed up the loop
boolean updatesWereEnabled = model.disableUpdates(disableFlag);
try {
  // Bulk-set parameters Val1, Val2, ...
  for (int i = 1; i <= maxIterations; i++) {</pre>
    model.param("default").set("Val"+i, i);
  }
} finally {
  // Always restore the original update state, even if an error occurs
  model.disableUpdates(updatesWereEnabled);
}
// Compute and log elapsed time in h:mm:ss format
long endTime = timeStamp();
debugLog("End of test.");
debugLog("Data reading time: "+formattedTime(endTime-startTime, "h:min:s"));
```

This example is part of a collection available for download:
https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

• disable_updates.mph

LIMITATIONS WITH LOADING AND SAVING MODELS

If you use the loadModel method to load another model into your application, then the usual functionality for displaying the geometry, mesh, and results, for the loaded model, is not directly available in the application since that functionality is reserved for the embedded model. However, you can use the API to call geometry, mesh, study, and results functionality and extract numerical results from the loaded model. For example, you can change the value of parameters or variables of the loaded model, run a study, and extract numerical results.

However, in an application, you can display plots generated from models other than the embedded model by writing specific method code. First, you will need to use the loadModel method to load the desired model. Following this, the useGraphics method can be used to display a particular plot group from the loaded model. Here is a simplified example to illustrate this process:

```
Model m = loadModel(...);
useGraphics(m.result("pg1"), "/form1/graphics1");
```

In this example, loadModel is used to load a model, and useGraphics is used to display a plot group "pg1" from the loaded model in the graphics component

"/form1/graphics1"

Note that useGraphics exclusively supports the display of plot groups. This method cannot be used for displaying other graphical elements like geometry or mesh. Note, however, that functionality for plotting the CAD model as well as the mesh is possible also from plot groups. For example, by plotting a constant expression or a fixed color for a surface or volume plot. Or, by using a dedicated mesh plot.

The toolbar accompanying these plots is limited compared to the one available for plots from the embedded model. This limitation exists as certain actions associated with the full toolbar are not supported in this scenario.

Note that the loadModel and saveModel methods are not supported in standalone applications that have been compiled with COMSOL Compiler.

The Model Object Class Structure

For a full description of the class structure and method signatures, see the HTML document *Java Documentation* available in the *COMSOL Documentation*. You

find the Java documentation under COMSOL API for use with Java[®] at the bottom of the Documentation window.

The figure below shows the document as displayed in the COMSOL documentation viewer.



If you encounter a class that you are unfamiliar with you will get help by a tooltip as in the figure below for the output of the getView method.



By using the keyboard shortcut Ctrl+1 the correct type declaration is assigned to the variable as shown in the figure below.



This way you can avoid having to consult the Java Documentation.

The Application Object

The application object is a part of the model object and is the data structure that allows access to the user interface features of an application from within a method. The state of the application object is reflected in the COMSOL Desktop user interface by the contents of the Application Builder and its application tree.

You can write code using the Method Editor to directly access and change the features presented in a running application, including button text, icons, colors, and fonts.

The application object gives you access to a subset of the features and settings available in the Application Builder. You can use the application object methods for runtime modifications to the user interface, but not for building a complete user interface. For building the user interface of an application, you need to use the Form Editor as described in the book *Introduction to Application Builder*.

This section gives an overview of the application object.

Shortcuts

Form objects and other user interface components are referenced in methods by using a certain syntax. For example, using the default naming scheme form3/ button5 refers to a button with name button5 in form3 and form2/graphics3 refers to a graphics object with name graphics3 in form2. You can also change the default names of forms and form objects. For example, if form1 is your main form then you can change its name to main.

To simplify referencing form objects as well as menu, ribbon, and toolbar items by name, you can create shortcuts with a custom name. In the **Settings** window of an object or item, click the button to the right of the **Name** field and type the name of your choice.



To create or edit a shortcut, you can also use the keyboard shortcut Ctrl+K. All shortcuts that you create are made available in a **Shortcuts** node under **Declarations** in the application tree.

> 🗒 Events
✓
Boolean
123 Integer
8.5 Double
abc String
Material {materialList}
Simulation Type List {simulationTypeList}
📃 Shortcuts
> 晴 Methods
> III Libraries

In the **Settings** window for **Shortcuts** shown below, a number of shortcuts were created for a various form objects.

Settings Shortcuts		* E	1 # >
List of Shortcuts			
* Name	Target	Description	
reportButtonTablet	toolbarTablet/reportButtonTablet	Shortcut to Button	~
targetFrequencyTextSmart	mainSmartphone/targetFrequencyTextSmartphone	Shortcut to Text label	
targetFrequencyInputSma	mainSmartphone/targetFrequencyInputSmartphone	Shortcut to Input field	
frequencyToleranceTextS	mainSmartphone/frequencyToleranceTextSmartphone	Shortcut to Text label	
frequencyToleranceInputS	mainSmartphone/frequencyToleranceInputSmartphone	Shortcut to Input field	
prongLengthInputSmartp	mainSmartphone/prongLengthInputSmartphone	Shortcut to Input field	
targetFrequencyUnitSmart	mainSmartphone/targetFrequencyUnitSmartphone	Shortcut to Unit	
frequencyToleranceUnitS	mainSmartphone/frequencyToleranceUnitSmartphone	Shortcut to Unit	
plotButtonSmartphone	toolbarSmartphone/plotButtonSmartphone	Shortcut to Toolbar	
playSoundButtonSmartph	toolbarSmartphone/playSoundButtonSmartphone	Shortcut to Toolbar	
reportButtonSmartphone	toolbarSmartphone/reportButtonSmartphone	Shortcut to Toolbar	

The shortcuts can be referenced in other form objects or in code in the Method Editor. The example below shows a shortcut tempVis used as an input argument to a temperature plot.



Shortcuts are automatically updated when objects are renamed, moved, copied, and duplicated. They are available in application methods as read-only Java[®] variables, similar to string, integer, double, and Boolean declarations.

Using shortcuts is recommended because it avoids the need to update methods when the structure of the application user interface changes.

Shortcuts can also be created for most objects in the model builder tree.

EXAMPLE CODE

If the application contains a button named button1 in a form named form1, and the button has a shortcut named b1, the following two ways to change the button text to red are equivalent:

```
b1.set("foreground", "red");
app.form("form1").formObject("button1").set("foreground", "red");
```

Accessing the Application Object

In the Method Editor you can directly access the application object part of the model object by using the app variable. This variable is a shorthand for model.app().

The Name of User Interface Components

Access the various parts of the application object by using the *name* of a form object, form, item, and so on. A *name* in the application object has the same function as the *tag* in the model object omitting the model.app part.

For example, in the line of code

app.form("form1").formObject("button1").set("enabled", false);

the string form1 is the *name* of a form and button1 is the *name* of a button.

Important Classes

THE MAIN APPLICATION CLASS

When working with an application object, the main application class is AppModel, which is the type of model.app().

DECLARATION CLASSES

In addition to the basic data types and shortcut declarations, the **Declaration** node may include Choice List and Unit List declarations. The corresponding classes are ChoiceList and UnitSet. The parent class to ChoiceList and UnitSet is called DataSource. In addition, Scalar, Array 1D, and Array 2D data types are handled by the DataSource class. For more information, see "Data Source" on page 109.

METHOD CLASS

The Method class is used to represent methods. For more information, see "Method Class" on page 116.

MAIN USER INTERFACE COMPONENT CLASSES

In an application object, the main user interface components correspond to the following classes:

- MainWindow
 - The class representing the Main Window node in the application tree.
- Form
 - The class representing a form.
- FormObject
 - The class representing a form object.
- Item
 - The class representing, for example, a menu, toolbar, or ribbon item.
- AppEvent
 - The class used for application user-interface events.

Each class has a set of associated methods that are used to edit the corresponding user interface component at runtime. These are described in the following sections.

In addition to the main user interface component classes, there are also list versions of the Form, FormObject, and Item classes. These are: FormList, FormObjectList, and ItemList.

Get and Set Methods for the Color of a Form Object

The get and set methods described in the section "Get and Set Methods for Accessing Properties" on page 41 are applicable to the model object as well as the

model.app part of the model object. In addition, the following methods are available for changing the color of a form object:

NAME	SYNTAX	DESCRIPTION
getColor	<pre>int[] getColor(String prop)</pre>	Get the value of a color property as an array of red, green, and blue values.
setColor	<pre>setColor(String prop, int r, int g, int b)</pre>	Set a color property using red, green, and blue values.

Not all methods are applicable to all properties. Use Ctrl+Space to use code completion to find out what methods are applicable for a certain object, and what property names and property values are applicable for a certain method.

General Properties

The following table lists properties that are available for several different user interface components, including form objects. In the table below, a user interface component is referred to as an object.

PROPERTY	VALUE	DEFAULT	DESCRIPTION
background	String	default	The background color for the corresponding user interface element.
enabled	true false	true	If the value is true , the corresponding object is enabled in the user interface, which means that the user can interact with the object.
font	String	default	The font family name. The special value default means that the font to use is determined by the parent object, which is the corresponding setting in the Settings window of the Forms node.
fontbold	true false	false	If true the font uses boldface style.
fontitalic	true false	false	If true the font uses italic style.
fontunderline	true false	false	If true the font uses underline style.

PROPERTY	VALUE	DEFAULT	DESCRIPTION
fontsize	String	-1	The font size in points. The special value -1 represents the default size, which means that the size is taken from the parent object (the Forms node) or from the system default size if no parent object defines the size.
foreground	String	default	The foreground color for the corresponding user interface element.
visible	true false	true	If the value is true , the corresponding object is visible in the user interface.

A foreground or background color property is represented by a string value. The available colors are: black, blue, cyan, gray, green, magenta, red, white, and yellow, or a custom color may also be defined. The special value default means that the color is taken from the parent object. Depending on the parent type, this could mean that default is **Inherit** or **Transparent**, referring to the corresponding setting in the **Settings** window in the Form Editor. An arbitrary RGB color can be represented by a string of the form rgb(red,green,blue) where red, green, and blue are integers between 0 and 255. Color properties can also be manipulated using the getColor and setColor methods to directly access the red, green, and blue color components. If a color property has the value default, it does not have red, green, and blue values. In this case, the getColor method returns the array [0,0,0].

EXAMPLE CODE

The following example reads the current background color for a form, makes the color darker, and applies the modified color to the same form.

```
int[] rgb = app.form("form1").getColor("background");
for (int i = 0; i < 3; i++)
  rgb[i] /= 2;
app.form("form1").setColor("background", rgb[0], rgb[1], rgb[2]);
```

The following line of code sets the background color to black:

```
app.form("form1").set("background", "black");
```

The following line of code sets the background color to default which in the case of the background color property corresponds to the Form Editor setting **Transparent**.

```
app.form("form1").set("background", "default");
```

The following line of code sets the background color to the RGB values 125, 45, and 43.

```
app.form("form1").set("background", "rgb(125,45,43)");
```

NAME	SYNTAX	DESCRIPTION
declaration	Declaration declaration()	Returns the list of declarations.
declaration	DataSource declaration(String name)	Returns the declaration object (Scalar, Array 1D, Array 2D, ChoiceList, or UnitSet) with the specified name.
event	<pre>AppEventHandlerList event()</pre>	Returns the list of event handlers.
event	AppEventHandler event(String name)	Returns the event handler with the specified name.
form	FormList form()	Returns the list of forms.
form	Form form(String name)	Returns the form with the specified name.
hasProperty	boolean hasProperty(String name)	Returns true if there is a modifiable property with the specified name.
mainWindow	MainWindow mainWindow()	Returns the MainWindow object.

The main application class AppModel has the following methods:

The AppModel class has the following properties:

PROPERTY	VALUE	DEFAULT	DESCRIPTION
asktosave	true false	false	If true, ask user if changes should be saved before the application is closed.
startmode	edit run	edit	Determines whether the application is opened for editing or running when you double-click the MPH file, including Windows desktop icons.

EXAMPLE CODE

```
app.set("asktosave", true);
```

The following code appends a text string to the application window title.

```
String oldTitle = app.mainWindow().getString("title");
app.mainWindow().set("title", oldTitle + " modified");
```

The following examples show how to query the list of declarations in an application.

```
// Get the declaration list
Declaration list = app.declaration();
// Get the names of all DataSource objects in the list.
String[] names = list.names();
```

```
// Get the number of DataSource objects in the list.
int size = list.size();
// Get the DataSource with the name "svar".
DataSource src = list.get("svar");
// Get the index within the list of the DataSource with the name "svar".
int index = list.index("svar");
// Get the DataSource at a certain index within the list.
DataSource src = list.get(index);
// Get the DataSource objects defined in a given form.
Declaration formDeclarations = app.form("form1").declaration();
// Iterate over DataSource objects within the list.
for(DataSource dt : list) {
    // Get the type of the DataSource.
    String type = dt.getType();
}
```

Main Window

NAME	SYNTAX	DESCRIPTION
fileMenu	<pre>ItemList fileMenu()</pre>	Returns the list of items in the file menu.
fileMenu	Item fileMenu(String name)	Returns the file menu item with the specified name.
hasProperty	boolean hasProperty(String name)	Returns true if there is a modifiable property with the specified name.
menuBar	<pre>ItemList menuBar()</pre>	Returns the list of items in the menu bar.
menuBar	Item menuBar(String name)	Returns the menu bar item with the specified name.
ribbon	<pre>ItemList ribbon()</pre>	Returns the list of items in the ribbon.
ribbon	Item ribbon(String name)	Returns the ribbon item with the specified name.
toolBar	<pre>ItemList toolBar()</pre>	Returns the list of items in the toolbar.
toolBar	Item toolBar(String name)	Returns the toolbar item with the specified name.

The MainWindow class has the following methods:

The menuBar and toolBar items are visible in the application user interface if the menu type is set to **Menu bar** in the **Settings** window of the **Main Window**. The ribbon and fileMenu items are visible in the user interface if the menu type is set to **Ribbon**. It is possible to access and modify items that are not visible based on

the menu type setting, but doing so will not have any visible effect in the user interface.

PROPERTY	VALUE	DEFAULT	DESCRIPTION
imagetheme	\$default \$light \$dark	\$default	Set the color theme for image export.
showfilename	true false	true	If true the filename is included in the window title bar title.
title	String	My application	The text to display in the window title bar.
theme	\$default \$light \$dark	\$default	Set the color theme for the application user interface.

The MainWindow class has the following properties:

EXAMPLE CODE

```
// Do not show the filename in the application user interface window bar.
app.mainWindow().set("showfilename", false);
// Set dark application theme.
app.mainWindow().set("theme", "$dark");
// Set light image export theme.
app.mainWindow().set("imagetheme", "$light");
```

Form

The Form class has the following methods:

NAME	SYNTAX	DESCRIPTION
declaration	DataSource declaration(String name)	Returns a form declaration object (Scalar, Array ID, Array 2D, or ChoiceList) with the specified name.
formObject	<pre>FormObjectList formObject()</pre>	Returns the list of form objects.
formObject	FormObject formObject(String name)	Returns the form object with the specified name.
getName	<pre>String getName()</pre>	Returns the name of this form.
getParentForm	Form getParentForm()	Returns the parent form that contains this form. Useful for local cards in a card stack.

NAME	SYNTAX	DESCRIPTION
hasProperty	boolean hasProperty(String name)	Returns true if there is a modifiable property with the specified name.
method	MethodList method()	Returns the list of methods.
method	Method method(String name)	Gets a method with the specified name.

The Form class has the following properties:

PROPERTY	VALUE	DEFAULT	DESCRIPTION
icon	String		The name of the background image. Valid values are images defined in the Images > Libraries node in the Application Builder.
iconhalign	left center right fill repeat	center	Horizontal alignment for the background image.
iconvalign	top center bottom fill repeat	center	Vertical alignment for the background image.
title	String	Form N	The form title for an integer N.

EXAMPLE CODE

```
app.form("form1").set("icon", "compute.png");
app.form("form1").formObject("button1").set("enabled", false);
DataSource ds = app.form("form1").declaration("var");
```

For examples of how to use the declaration method, see "The Main Application Methods" on page 82.

Form Object

The FormObject class has the following methods:

NAME	SYNTAX	DESCRIPTION
declaration	DataSource declaration(String name)	Returns a form declaration object (Scalar, Array ID, Array 2D, or ChoiceList) with the specified name.
expanded	void expanded(String form, boolean expanded)	Sets the expanded state of the form when using the sections type.

NAME	SYNTAX	DESCRIPTION
form	<pre>FormList form()</pre>	For a CardStack form object, returns the list of local cards.
form	Form form(String name)	For a CardStack form object, returns the local card with the specified name.
getName	<pre>String getName()</pre>	Returns the name of this form object.
getParentForm	<pre>Form getParentForm()</pre>	Returns the parent form that contains this form object.
getType	<pre>String getType()</pre>	Returns the form object type name, as defined in the following sections.
getValue	Value getValue()	Returns the current value of a FormObject when applicable. The method is supported by the following types of FormObjects: input field, toggle button, checkbox, combo box, file import, array input, radio button, selection input, text, list box, table, slider, knob.
hasProperty	boolean hasProperty(String name)	Returns true if there is a modifiable property with the specified name.
item	<pre>ItemList item()</pre>	For a Toolbar, Graphics, or Table form object, returns the list of user-defined buttons.
item	Item item(String name)	For a Toolbar, Graphics, or Table form object, return the user-defined button with the specified name.
visible	void visible(String form, boolean visible)	Sets the visible state of the form.

Most form objects have one or more of the properties listed in "General Properties" on page 80. A form object has a certain property if the corresponding setting is available in the Form Editor. Additional properties are supported for several types of form objects. The general properties that are supported and any additional properties for form objects are listed in the following sections.

EXAMPLE CODE

The following code loops over all buttons and disables them:

```
for (FormObject formObject : app.form("form1").formObject()) {
   if ("Button".equals(formObject.getType())) {
     formObject.set("enabled", false);
   }
}
```

The getType method retrieves the type of form object. In the above example the type of form object is Button and the statement

"Button".equals(formObject.getType()) represents a string comparison between the output of the getType method and the string "Button". For an example of using getValue, see "Data Validation" on page 288. The following table lists all form object types that can be returned by getType:

FORM OBJECT TYPE		
ArrayInput	Hyperlink	SelectionInput
Button	Image	Slider
CardStack	InformationCardStack	Spacer
CheckBox	InputField	Table
ComboBox	Knob	Text
DataDisplay	Line	TextLabel
Equation	ListBox	ToggleButton
FileImport	Log	Toolbar
Form	MessageLog	Unit
FormCollection	ProgressBar	Video
Gauge	RadioButton	WebPage
Graphics	ResultsTable	

ARRAY INPUT

Property	Value	Default	Description
background enabled font fontbold fontitalic fontsize foreground visible			See ''General Properties'' on page 80.

Example Code

```
app.form("form1").formObject("arrayinput1").set("enabled", false);
```

BUTTON

PROPERTY	VALUE	DEFAULT	DESCRIPTION
enabled font fontbold fontitalic fontsize foreground visible			See ''General Properties'' on page 80.
icon	String		The button icon. Valid values are images defined in the Images > Libraries node in the Application Builder.
text	String	Generated automatically	The button text. The text must not be an empty string.
tooltip	String		The button tooltip text.

In the Form Editor, if a button has its **Size** setting set to **Large**, it always displays its text property. If the button is **Small**, it either displays the icon or the text according to the following rule: if the icon property is empty, the text is displayed, if the icon property is not empty, the icon is displayed.

Example Code

app.form("form1").formObject("button1").set("enabled", false);

CARD STACK

PROPERTY	VALUE	DEFAULT	DESCRIPTION
enabled visible			See "General Properties" on page 80.

Example Code

app.form("form1").formObject("cardstack1").set("visible", false);

To access objects in a local card, either use shortcuts or use the form method:

```
app.form("form1").formObject("cardstack1").form("card1")
    .formObject("button1").set("enabled", false);
```

Снесквох

PROPERTY	VALUE	DEFAULT	DESCRIPTION
background enabled font fontbold fontitalic fontsize fontunderline foreground visible			See ''General Properties'' on page 80.
text	String	Generated automatically	The checkbox label text.
tooltip	String	н н	The checkbox tooltip text.

Example Code

```
app.form("form1").formObject("checkbox1").set("tooltip", "tooltip text");
```

Сомво Вох

PROPERTY	VALUE	DEFAULT	DESCRIPTION
enabled font fontbold fontitalic fontsize foreground visible			See ''General Properties'' on page 80.

Example Code

```
app.form("form1").formObject("combobox1").set("foreground", "blue");
```

DATA DISPLAY

PROPERTY	VALUE	DEFAULT	DESCRIPTION
background enabled font fontbold fontitalic fortsize foreground visible			See ''General Properties'' on page 80.
exponent	superscript E	superscript	When set to superscript , exponents are displayed using superscript font. When set to E , exponents are displayed using the character E followed by the exponent value.
notation	auto scientific decimal	auto	When the value is scientific, numbers are always displayed using scientific notation. When the value is decimal, numbers are never displayed using scientific notation. When the value is auto, the notation depends on the size of the number.
precision	Integer	4	The number of significant digits that are displayed.
showunit	true false	true	Controls if the unit is displayed in addition to numerical values.
tooltip	String		The tooltip text.

Example Code

app.form("form1").formObject("datadisplay1")
 .setColor("background", 192, 192, 192);

EQUATION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
enabled fontsize foreground visible			See ''General Properties'' on page 80.
equation	String		The string in LaTeX syntax defining the equation.

Example Code

app.form("form1").formObject("equation1").set("visible", false);

FILE IMPORT

PROPERTY	VALUE	DEFAULT	DESCRIPTION
background enabled font fontbold fontitalic fortsize foreground visible			See "General Properties" on page 80.
buttontext	String	Browse	Text to display on the button that opens the file browser.
dialogtitle	String	File import	Text to display as dialog title for the file browser dialog. Also displayed as a tooltip for the FileBrowser form object.
filetypes	String[]	{"ALLFILES"}	Defines the list of file types that can be selected in the file browser.

Example Code

```
app.form("form1").formObject("fileimport1")
    .set("filetypes", new String[]{"ALL2DCAD"});
```

Form

A form used as a subform does not have any modifiable properties.

FORM COLLECTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
enabled font fontbold fontitalic fontsize foreground visible			See ''General Properties'' on page 80.
formvisible	boolean[]	Array with each entry having the value true	Defines the visible state of forms in a form collection.
sectionexpanded	boolean[]	Array with each entry having the value true	Defines the expanded state of forms in a form collection when using the sections type.

To modify the active pane, change the corresponding declaration variable.

Example Code

This line of code changes the font:

```
app.form("form1").formObject("collection1").set("font", "Arial");
```

The expanded state of sections in form collections can be controlled by:

app.form("form1").formObject("formcollection1").expanded("form2", false);

The expanded method is only supported by form collections which use sections. The first argument is the tag of the form which is represented by the section. The second argument determines if the sections should be expanded or collapsed.

GRAPHICS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
datapick	true false	false	If true , data picking is enabled in the graphics form object.
datapicktarget	ProbeFeature GraphicsData		Defines where the picked data is stored. Valid values are probe features and GraphicsData declarations.
enabled visible			See ''General Properties'' on page 80.
source	ModelEntity		Defines the type of model entity (Plot Group, Geometry, Mesh, Explicit Selection or Player Animation) used to plot in the graphics form object.

Example Code

This line of code displays plot group 5 (pg5) in the graphics object graphics1 in the form with the name Temperature:

```
app.form("Temperature").formObject("graphics1")
    .set("source", model.result("pg5"));
```

The following line of code using useGraphics is equivalent to the above example:

```
useGraphics(model.result("pg5"), "Temperature/graphics1");
```

Either method changes the source of the graphics form object and then plots the contents.

To clear the contents of a graphics object, use:

```
app.form("Temperature").formObject("graphics1")
    .set("source", (ModelEntity) null);
```

or equivalently

```
useGraphics(null, "Temperature/graphics1");
```

The code below displays the mesh in the model tree node mesh1 in the graphics object graphics1 contained in the card of a card stack:

```
app.form("mesh").formObject("cardstack1").form("card1")
    .formObject("graphics1").set("source", model.mesh("mesh1"));
```

The code below enables data picking for a graphics object and sets the data picking target to a domain point probe:

If a shortcut graphics 1 has been created for the graphics object and a shortcut pdom1 has been created for the probe object, the above can be shortened to:

```
graphics1.set("datapick", true);
graphics1.set("datapicktarget", pdom1);
```

Hyperlink

PROPERTY	VALUE	DEFAULT	DESCRIPTION
background enabled font fontbold fontitalic fontsize visible			See ''General Properties'' on page 80.
text	String	Generated automatically	The text to display on the HyperLink form object.
url	String	н н	The URL to open when the HyperLink is activated.

Example Code

```
with (app.form("form1").formObject("hyperlink1"));
  set("text", "COMSOL");
  set("url", "www.comsol.com");
endwith();
```

IMAGE

PROPERTY	VALUE	DEFAULT	DESCRIPTION
enabled visible			See ''General Properties'' on page 80.
icon	String	cube_large.png	Defines the icon name to display in the Image form object. Valid values are images defined in the Images > Libraries node in the Application Builder.

Example Code

```
app.form("form1").formObject("image1").set("icon", "compute.png");
```

INFORMATION CARD STACK

PROPERTY	VALUE	DEFAULT	DESCRIPTION
background enabled font fontbold fontitalic fontsize fontunderline visible			See ''General Properties'' on page 80.

Example Code

app.form("form1").formObject("infocard1").set("fontunderline", true);

INPUT FIELD

PROPERTY	VALUE	DEFAULT	DESCRIPTION
background enabled font fontbold fontitalic fontsize foreground visible			See ''General Properties'' on page 80.
editable	true false	true	If true then the text in the input field can be edited by the user.
exponent	superscript E	superscript	When set to superscript , exponents are displayed using superscript font. When set to E , exponents are displayed using the character E followed by the exponent value.
inputformatting	on off	off	If the value is on , then numerical values in the input field are formatted according to the exponent, notation, and precision properties. When the user is editing the text in the input field, the formatting is temporarily disabled so that the original text can be edited.
maxdouble	double	1000	The maximum allowed double value. This property is only accessible when the Filter setting is set to Double and the corresponding checkbox is enabled in the Data Validation section.
mindouble	double	0	The minimum allowed double value. This property is only accessible when the Filter setting is set to Double and the corresponding checkbox is enabled in the Data Validation section.

PROPERTY	VALUE	DEFAULT	DESCRIPTION
maxinteger	Integer	1000	The maximum allowed integer value. This property is only accessible when the Filter setting is set to Integer and the corresponding checkbox is enabled in the Data Validation section.
mininteger	Integer	0	The minimum allowed integer value. This property is only accessible when the Filter setting is set to Integer and the corresponding checkbox is enabled in the Data Validation section.
notation	auto scientific decimal	auto	When the value is scientific, numbers are always displayed using scientific notation. When the value is decimal, numbers are never displayed using scientific notation. When the value is auto, the notation depends on the size of the number.
precision	Integer	4	The number of significant digits displayed.
tooltip	String	N N	The tooltip displayed when the mouse pointer is located over the input field.

Example Code

app.form("form1").formObject("inputfield1").set("precision", 6);

Line

PROPERTY	VALUE	DEFAULT	DESCRIPTION
enabled font fontbold fontitalic fontsize foreground visible			See ''General Properties'' on page 80.
text	String	и и	Text to display on the line. The text is only displayed for horizontal lines that have Include divider text enabled in the Line object Settings window.
thickness	Integer	1	The line thickness.
linecolor	String	default	The line color.

Example Code

```
app.form("form1").formObject("line1").set("text", "divider text");
app.form("form1").formObject("line1").set("thickness", 10);
app.form("form1").formObject("line1").set("linecolor", blue);
```

LIST BOX

PROPERTY	VALUE	DEFAULT	DESCRIPTION
enabled font fontbold fontitalic fontsize foreground visible			See ''General Properties'' on page 80.

Example Code

```
app.form("form1").formObject("listbox1").set("foreground", "red");
```

To change the list box contents, modify the corresponding choice list:

app.declaration("choicelist1").appendListRow("new value", "new name");

Log

PROPERTY	VALUE	DEFAULT	DESCRIPTION
background enabled font fontbold fontitalic fortsize foreground visible			See ''General Properties'' on page 80.

Example Code

```
app.form("form1").formObject("log1").set("fontsize", "20");
```

Message Log

PROPERTY	VALUE	DEFAULT	DESCRIPTION
background enabled font fontbold fontitalic fortsize foreground visible			See ''General Properties'' on page 80.

Example Code

```
app.form("form1").formObject("messages1").set("background", "gray");
```

PROGRESS BAR

PROPERTY	VALUE	DEFAULT	DESCRIPTION
enabled visible			See ''General Properties'' on page 80.

To create and update progress information see "Progress Methods" on page 153.

Example Code

```
app.form("form1").formObject("progressbar1").set("visible", false);
```

RADIO BUTTON

PROPERTY	VALUE	DEFAULT	DESCRIPTION
background enabled font fontbold fontitalic fontsize fontunderline foreground visible			See ''General Properties'' on page 80.

To change the display name for a radio button, modify the value in the corresponding choice list.

For a choice list that is used by a radio button, it is not possible to change the value of any row, or to add or remove rows. Only the display name can be changed.

Example Code

```
app.form("form1").formObject("radiobutton1").set("fontitalic", true);
app.declaration("choicelist1").setDisplayName("new name", 0);
```

RESULTS TABLE

PROPERTY	VALUE	DEFAULT	DESCRIPTION
enabled font fontbold fontitalic fontsize foreground visible			See ''General Properties'' on page 80.
source	TableFeature		Set the contents of the results table.

To change the contents of the results table use the method useResultsTable or evaluateToResultsTable. See also "GUI-Related Methods" on page 137.

```
Example Code
```

```
app.form("form1").formObject("resultstable1").set("visible", true);
app.form("form1").formObject("resultstable1").set("source",
    model.result().table("tbl2"));
useResultsTable(model.result().table("tbl2"), "/form1/resultstable1");
```

Selection Input

PROPERTY	VALUE	DEFAULT	DESCRIPTION
active	true false	false	The active property controls whether the Selection Input is active. Changing the property value selects or deselects the Activate Selection button in the Selection Input.
background enabled font fontbold fontitalic fortsize foreground visible			See ''General Properties'' on page 80.
graphics	FormObject		Defines the graphics form object to use when the selection form object is active.
source	SelectionFeature		Defines the model selection the selection form object is connected to.

Example Code

```
app.form("form1").formObject("selectioninput1").set("graphics",
    "graphics1");
```

Alternatively, if there are shortcuts sel1 and g1 to the selectioninput1 and graphics1 form objects:

sel1.set("graphics", g1);

To change the model selection, assuming sel1 is a shortcut to the selection input form object:

```
sel1.set("source", model.selection("sel2"));
```

SLIDER

PROPERTY	VALUE	DEFAULT	DESCRIPTION
enabled visible			See ''General Properties'' on page 80.
max	Double	1	The largest possible slider value.
min	Double	0	The smallest possible slider value.
steps	Integer	5	The number of steps between the min and max values. The number of tick marks is one more than the number of steps.
tooltip	String		The tooltip text.

The min value is allowed to be larger than the max value, in which case the slider behaves as if the values were swapped. The smallest value always corresponds to the left side of the slider.

Example Code

```
app.form("form1").formObject("slider1").set("min", 1);
app.form("form1").formObject("slider1").set("max", 12);
app.form("form1").formObject("slider1").set("steps", 11);
```

Клов

PROPERTY	VALUE	DEFAULT	DESCRIPTION
background enabled font fontbold fontitalic fontsize foreground visible			See ''General Properties'' on page 80.
exponent	superscript E	superscript	When set to superscript , exponents are displayed using superscript font. When set to E , exponents are displayed using the character E followed by the exponent value.
knobmaincolor	String	default	The main background color.
knobradius	Integer		When knobradiusmanual has the value on , this is the value of the knob. The tick marks appear outside this radius.
knobradiusmanual	on off	off	When the value is off the radius is calculated automatically. When the value is on , the radius is given by the knobradius property.
knobsecondarycolor	String	default	The secondary background color.
labelformatting	true false	false	When true this setting enables custom label formatting.
max	Double	1	The largest possible knob value.
min	Double	0	The smallest possible knob value.
mouse	distance vertical circular	distance	The kind of mouse movement to change the knob value. When set to distance, the distance the mouse is moved, either horizontally or vertically, controls the value. When set to vertical, only the vertical distance the mouse is moved controls the value. When set to circular, the value is changed using a circular motion.
needlecolor	String	default	The color of the knob's needle.

PROPERTY	VALUE	DEFAULT	DESCRIPTION
notation	auto scientific decimal	auto	When the value is scientific, numbers are always displayed using scientific notation. When the value is decimal, numbers are never displayed using scientific notation. When the value is auto, the notation depends on the size of the number.
precision	Integer	4	The number of significant digits that are displayed.
scale	none marks markslabels marksalllabels	marks	To control whether the scale should have tick marks or labels. When set to none , the knob has no tick marks or labels. When set to marks , the knob has tick marks but no labels. When set to markslabels , the knob has tick marks and labels for the minimum and maximum values. When set to marksalllabels , the knob has ticks marks and all major ticks has a label.
steps	Integer	5	The number of steps between the min and max values. The number of tick marks is one more than the number of steps.
tooltip	String		The tooltip text.

Example Code

```
app.form("form1").formObject("knob1").set("min", 1);
app.form("form1").formObject("knob1").set("max", 12);
app.form("form1").formObject("knob1").set("steps", 11);
```

Spacer

A spacer object does not have any modifiable properties.

TABLE

PROPERTY	VALUE	DEFAULT	DESCRIPTION
enabled font fontbold fontitalic fontsize foreground visible			See ''General Properties'' on page 80.
headers	String[]	Generated automatically	Column headers.
showheaders	boolean	Generated automatically	Show column headers.

To change the contents of the table, change the declaration variables or model entities the table is displaying.

Example Code

```
app.form("form1").formObject("table1").set("enabled", false);
```

PROPERTY	VALUE	DEFAULT	DESCRIPTION
background enabled font fontbold fontitalic fortsize foreground visible			See "General Properties" on page 80.
editable	on off	off	If the value is on , the text can be edited by the user of the application. If the value is off , the text can only be changed programmatically.
textalign	left center right	left	Defines how the text is aligned within the text area.
wrap	on off	on	If the value is on , word wrapping is used to break lines that are too long to fit within the text area. If the value is off , long lines may not be completely visible.

Text

```
Example Code
app.form("form1").formObject("text1").set("textalign", "center");
```

TEXT LABEL

PROPERTY	VALUE	DEFAULT	DESCRIPTION
background enabled font fontbold fontitalic fontsize fontunderline foreground visible			See ''General Properties'' on page 80.
text	String	Generated automatically	The text to display in the label when the label is not in multiline mode.
textmulti	String	Generated automatically	The text to display in the label when the label is in multiline mode.

Example Code

```
app.form("form1").formObject("textlabel1").set("text", "custom text");
```

TOGGLE BUTTON

PROPERTY	VALUE	DEFAULT	DESCRIPTION
enabled font fontbold fontitalic fontsize foreground visible			See ''General Properties'' on page 80.
icon	String	н и	The button icon. Valid values are images defined in the Images > Libraries node in the Application Builder.
text	String	Generated automatically	The button text. The text must not be an empty string.
tooltip	String		The button tooltip text.

A button with size large always displays the text, a button with size small displays either the icon or the text. If the icon property is empty, the text is displayed. If the icon property is not empty, the icon is displayed.

Example Code

```
app.form("form1").formObject("togglebutton1").set("icon",
     "about_information.png");
```

TOOLBAR

PROPERTY	VALUE	DEFAULT	DESCRIPTION
background enabled font fontbold fontitalic fontsize foreground visible			See ''General Properties'' on page 80.

Example Code

```
app.form("form1").formObject("toolbar1").set("background", "gray");
```

UNIT

PROPERTY	VALUE	DEFAULT	DESCRIPTION
background enabled font fontbold fontitalic fortsize foreground visible			See ''General Properties'' on page 80.

Example Code

```
app.form("form1").formObject("unit1").set("visible", false);
```

VIDEO

PROPERTY	VALUE	DEFAULT	DESCRIPTION
visible			See ''General Properties'' on page 80.

Example Code

```
app.form("form1").formObject("video1").set("visible", false);
```

Web Page

PROPERTY	VALUE	DEFAULT	DESCRIPTION
file	String		The file to display. File scheme syntax is supported.
html	String	<html></html>	The HTML code to display.
report	ReportFeature or String		The report feature to display.
type	page url type report	page	Determines which property is used to specify the browser display contents.
url	String	https://www.comsol.com	The URL to display.
visible			See "General Properties" on page 80.

Example Code

```
app.form("form1").formObject("webpage1").set("type", "report");
app.form("form1").formObject("webpage1").set("report", "rpt1");
model.result().report("rpt1");
model.result().report("rpt1").run();
```

ltem

Item objects represent items, toggle items, user defined buttons in Toolbar, Graphics, and Table form objects, and submenus in the menu bar, toolbar, ribbon, and file menus. The following methods are available:

NAME	SYNTAX	DESCRIPTION	
getParentItem	<pre>Item getParentItem()</pre>	Returns the parent item, or null for a top- level item.	
hasProperty	boolean hasProperty(String name)	Returns true if there is a modifiable property with the specified name.	
item	<pre>ItemList item()</pre>	Returns the list of subitems.	
item	<pre>Item item(String name)</pre>	Returns the subitem with the specified name.	
PROPERTY	VALUE	DEFAULT	DESCRIPTION
----------	----------	----------------------------	---
enabled	on off	on	If the value is on , the item can be activated by the user. If the value is off , the item cannot be activated.
icon	String	Generated automatically	The icon name. Valid values are images defined in the Images > Libraries node in the Application Builder.
text	String	Generated automatically	The text for a menu or ribbon item.
title	String	Generated automatically	The title text for a menu or submenu.
tooltip	String		The tooltip text.
visible	on off	on	Controls whether the item is visible or not.

The Item class contains the following properties:

In order for an item to be enabled, the enabled property needs to have the value on for the item itself as well as for all of its parents. In other words, disabling an item also disables all of its subitems.

Item objects also include separators. However, separators do not have any accessible properties.

Item objects for user-defined buttons do not have the title and tooltip properties. For Table form objects, predefined items such as "move up" and "move down" do not have any modifiable properties.

EXAMPLE CODE

```
app.mainWindow().menuBar("menu1").set("title", "new title");
app.mainWindow().menuBar("menu1").item("toggle_item1").set("text",
        "test");
```

Data Source

A DataSource object is either a Scalar, Array 1D, Array 2D, ChoiceList a UnitSet or a GraphicsData.

SCALAR, ARRAY ID, AND ARRAY 2D METHODS

The methods described in the following table are applicable for Scalar, Array 1D, and Array 2D objects (of types String, Boolean, Integer, Double). These methods are used to manipulate such variable objects during runtime.

NAME	SYNTAX	DESCRIPTION
getBoolean	boolean getBoolean()	Gets the Boolean value stored in the data source.
getBooleanArray	<pre>boolean[] getBooleanArray()</pre>	Gets the Boolean array value stored in the data source.
getBooleanMatrix	<pre>boolean[][] getBooleanMatrix()</pre>	Gets the Boolean matrix value stored in the data source.
getDouble	<pre>double getDouble()</pre>	Gets the double value stored in the data source.
getDoubleArray	<pre>double[] getDoubleArray()</pre>	Gets the double array value stored in the data source.
getDoubleMatrix	<pre>double[][] getDoubleMatrix()</pre>	Gets the double matrix value stored in the data source.
getInt	<pre>int getInt()</pre>	Gets the int value stored in the data source.
getIntArray	<pre>int[] getIntArray()</pre>	Gets the int array value stored in the data source.
getIntMatrix	<pre>int[][] getIntMatrix()</pre>	Gets the int matrix value stored in the data source.
getString	<pre>String getString()</pre>	Gets the String value stored in the data source.
getStringArray	<pre>String[] getStringArray()</pre>	Gets the String array value stored in the data source.
getStringMatrix	<pre>String[][] getStringMatrix()</pre>	Gets the String matrix value stored in the data source.
set	<pre>set(boolean value) set(boolean[] value) set(boolean[][] value) set(double[] value) set(double[] value) set(double[][] value) set(int value) set(int[] value) set(int[][] value) set(String[] value) set(String[] value)</pre>	Set the value stored in the data source. Available methods depend on the type of the underlying declaration.

EXAMPLE CODE

```
// Get a scalar double declaration.
DataSource ds = app.declaration("var");
// The 'var' declaration is a scalar double so we use the getDouble method
// to read its value.
double cur = ds.getDouble();
// Modifying the local field 'cur' does not affect the value stored in the
// data source 'ds'.
cur = cur + 1;
// Set the value of the data source.
ds.set(cur);
```

CHOICE LIST AND UNIT SET METHODS

The methods described in the following table are applicable for both ChoiceList and UnitSet objects. These methods are used to manipulate choice lists and unit sets during runtime.

NAME	SYNTAX	DESCRIPTION
addListRow	addListRow(String value, String displayName, int row)	Inserts a new row with the given value and display name at the specified row (0-based).
appendListRow	appendListRow(String value, String displayName)	Inserts a new row with the given value and display name at the end of the list.
getValue	String getValue(int row)	Returns the value for the given row (0-based).
getDisplayName	<pre>String getDisplayName(int row)</pre>	Returns the display name for the given row (0-based).
getDisplayName	String getDisplayName(String value)	Returns the display name for the row with the given value.
getValues	<pre>String[] getValues()</pre>	Returns all values as an array.
getDisplayNames	<pre>String[] getDisplayNames()</pre>	Returns all display names as an array.
removeListRow	<pre>removeListRow(int row)</pre>	Removes the given row (0-based) from the list.
setListRow	<pre>setListRow(String value, String displayName, int row)</pre>	Sets the value and display name for the given row (0-based). If the row is equal to the length of the list, a new row is added.
setList	<pre>setList(String[] values, String[] displayNames)</pre>	Sets all of the values and display names, replacing the contents of the choice list or unit set.

NAME	SYNTAX	DESCRIPTION
setValue	setValue(String value, int row)	Sets the value for the given row (0-based). If the row is equal to the length of the list, a new row is added with the value and an empty display name.
setDisplayName	setDisplayName(String displayName, int row)	Sets the display name for the given row (0-based). If the row is equal to the length of the list, a new row is added with the display name and an empty value.

Example Code

The code below adds the string Aluminum 3004 to a choice list. Note that the choice list index starts at 0, whereas the material tags start at 1 (mat1, mat2, mat3, and mat4).

```
ChoiceList choiceList = getChoiceList("choiceList1");
choiceList.setListRow("mat4", "Aluminum 3004", 3);
```

For more information on using choice lists for changing materials, see the book *Introduction to Application Builder*.

Unit Set Methods

When the object is a UnitSet the following additional methods are also available:

NAME	SYNTAX	DESCRIPTION
getString	<pre>String getString()</pre>	Returns the currently selected value for the unit set.
getString	String getString(String unitList)	Returns the selected unit for the given unit list.
set	<pre>set(String value)</pre>	Switch unit for the unit set.

GRAPHICSDATA METHODS

When the object is a GraphicsData the following methods are available:

NAME	SYNTAX	DESCRIPTION
getString	String getString(String name)	Returns the value of the specified property as a string.
getDouble	double getDouble(String name)	Returns the value of the specified property as a double value.

NAME	SYNTAX	DESCRIPTION
getDoubleArray	<pre>double[] getDoubleArray(String name)</pre>	Returns the value of the specified property as a double array value.
set	<pre>set(String name, String value) set(String name, double value) set(String name, double[] value)</pre>	Sets a new value for the property with the specified name.

The GraphicsData class has the following properties:

PROPERTY	VALUE	DEFAULT	DESCRIPTION
coord	double[]	0, 0, 0	Point location of picked data. Array of 1, 2, 3, or 6 elements, corresponding to 1D, 2D, 3D, or Smith plot, respectively.
depth	double	0	Point along a line when data picking in a domain. Valid values are between 0 and 1, inclusive.
edim	domain boundary	domain	Geometric entity level.
eval	double	0	Evaluated value of picked data.
method	pointnormal pointdir twopoints none	pointnormal	Method of selecting a point in a 3D geometry.
twopoint	first second	first	Point being picked in the twopoints line entry method.

Example Code

The following code enables data picking for the graphics1 object, connects it to the graphicsdata1 object and sets some properties on the graphicsdata1 object:

AppEvent Class

The AppEvent class contains information regarding an application user-interface events. The event field, accessible in application methods, is initialized with an

instance of this class when a FormObject event is being triggered. The getUserControl method retrieves the FormObject where the event originated.

NAME	SYNTAX	DESCRIPTION
getNewValue	Value getNewValue()	New value corresponding to a form object having an On data change event type.
getOldValue	Value getOldValue()	Old value corresponding to a form object having an On data change event type.
getUserControl	IPropertyAccess getUserControl()	To make it convenient to access a user control that triggered an event the method event.getUserControl returns the form or form object that triggered the event. The return type of the methods is IPropertyAccess and the object can be cast to an instance of the FormObject or Form class depending on what type of event is triggered.

VALUE CLASS

The methods getNewValue and getOldValue returns an object of type Value which has methods according to the table below. The Value class represents a value which can be retrieved in different formats depending on the form object. For example, an event triggered from an input field is a scalar variable (boolean, int, double, or String). This means that only a subset of the methods in the table below is valid for a particular form object event.

NAME	SYNTAX	DESCRIPTION
getBoolean	<pre>boolean getBoolean()</pre>	Gets the Boolean value.
getBooleanArray	boolean[] getBooleanArray()	Gets the Boolean array value.
getBooleanMatrix	boolean[][] getBooleanMatrix()	Gets the Boolean matrix value.
getDouble	<pre>double getDouble()</pre>	Gets the double value.
getDoubleArray	double[] getDoubleArray()	Gets the double array value.
getDoubleMatrix	double[][] getDoubleMatrix()	Gets the double matrix value.
getInt	<pre>int getInt()</pre>	Gets the int value.
getIntArray	<pre>int[] getIntArray()</pre>	Gets the int array value.
getIntMatrix	int[][] getIntMatrix()	Gets the int matrix value.
getString	<pre>String getString()</pre>	Gets the string value.

INALITE	STINTAA	DESCRIFTION
getStringArray	<pre>String[] getStringArray()</pre>	Gets the string array value.
getStringMatrix	String[][] getStringMatrix	Gets the string matrix value.

Example Code

The following code exemplifies how to use the event field, for the class AppEvent, and the methods getNewValue and getOldValue:

```
String value = event.getNewValue().getString();
String oldValue = event.getOldValue().getString();
```

For an example of using the event field and its associated methods, see "Data Validation" on page 288.

AppEventHandler Class

This class represents the **Event** and **Timer** nodes found under **Events** in the Application Builder. An AppEventHandler object can be either an Event or a Timer.

The following methods are available for an AppEventHandler object:

NAME	Syntax	DESCRIPTION
getName	<pre>String getName()</pre>	Returns the name of this event handler.
getType	<pre>String getType()</pre>	Returns the event handler type name: Event or Timer.

An AppEventHandler object has the following properties:

PROPERTY	VALUE	DEFAULT	DESCRIPTION
delay	double	1	The interval of a Timer in seconds.
enabled			See ''General Properties'' on page 80.

Example code

The following code sets the interval of a Timer to 1 hour:

app.event("timer1").set("delay","1[h]");

Method Class

The Method class is used to represent a method. The following method is available for a Method object:

NAME	SYNTAX	DESCRIPTION
run	Object run(Object input)	Run the method with the specified input parameters.

The following code exemplifies using a Method object:

```
// Get a Method object representing a Form Method.
Method m = app.form("form1").method("method1");
// The method takes a String parameter and returns a double.
double res = (Double) m.run("input");
```

Form, Form Object, and Item List Methods

The AppEventHandlerList, Declaration, FormList, FormObjectList, and ItemList classes have the following methods:

NAME	SYNTAX	PURPOSE
get	Form get(int index) FormObject get(int index) Item get(int index) DataSource get(int index) AppEventHandler get(int index)	Returns the object at a certain index.
get	Form get(String name) FormObject get(String name) Item get(String name) DataSource get(String name) AppEventHandler get(String name)	Returns the object with a given name.
index	int index(String name)	Returns the 0-based position of the object with a given name in the list.
names	<pre>String[] names()</pre>	Returns an array of names for all objects in the list.
size	int size()	Returns the number of objects in the list.

Additionally the FormList class has the following methods:

NAME	SYNTAX	PURPOSE
hasProperty	boolean hasProperty(String name)	Returns true if there is a modifiable property with the specified name.

It is also possible to use a list in an enhanced for loop to operate on all objects in the list.

In the following example, the background color is set to red in all forms, by looping over all forms:

```
for (Form f : app.form()) { // app.form() is of type FormList
    f.set("background", "red");
}
```

The Built-in Method Library for the Application Builder

This section lists built-in methods available in the Method Editor in addition to the methods that operate on the model and application objects. For more information on the model object and its methods, refer to earlier sections of this book and the *Programming Reference Manual*. For more information on the application object, see "The Application Object" on page 76. Some of the listings have associated example code but for more extensive programming examples, see "Programming Examples" on page 178.

The syntax rules are those of the Java[®] programming language. Note that each line of code needs to end with a semicolon (;), but the semicolon is omitted in the listings below.

Model Utility Methods

The following table summarizes the model utility methods for querying, creating, loading, and saving model objects. The model object is stored on the MPH file format. These methods should not be confused with utility classes, defined in the application tree under libraries.

NAME	SYNTAX	DESCRIPTION
clearModel	clearModel(Model model) clearModel(String tag)	Remove everything from the model except for the application part and the things you can change on the root node in the Model Builder, such as thumbnail, comment, and author.
createModel	Model createModel(String tag)	Creates a new model with the given tag.
createModel	Model createModel()	Create a new model with a unique tag.
getModel	Model getModel(String tag)	Returns the model with a specified tag.
loadModel	Model loadModel(String location)	Loads a model with a specified tag from a file on the file system or in a database.
loadModel	Model loadModel(String tag, String location)	Loads a model from a file. The model is given a unique tag.

NAME	SYNTAX	DESCRIPTION
loadProtectedModel	Model loadProtectedModel(String tag, String filename, String password)	Loads a password protected model with a specified tag from a file.
loadProtectedModel	Model loadProtectedModel(String filename, String password)	Loads a password protected model from a file. The model is given a unique tag.
loadRecoveryModel	Model loadRecoveryModel(String tag, String foldername)	Loads a model from a recovery directory/folder structure.
loadRecoveryModel	Model loadRecoveryModel(String foldername)	Loads a model from a recovery directory/folder structure. The model is given a unique tag.
modelTags	<pre>String[] modelTags()</pre>	Returns an array of model tags for all loaded models, including the embedded model.
removeModel	removeModel(String tag) removeModel(Model model)	Removes a model. The embedded model cannot be removed.
saveModel	<pre>saveModel(Model model, String filename)</pre>	Saves a model to a file. The filename can be a file scheme path or (if allowed by security settings) a server file path.
uniqueModeltag	String uniqueModeltag(String prefix)	Returns a model tag that is not in use.
getComsolVersion	getComsolVersion()	Returns the current software version as a string.

For the built-in application methods loadModel(*<location>*) and loadModel(*<tag>*, <location>) the *<location>* argument scan either be the path to an MPH file found on the file system or a model location URI referencing a model version in a Model Manager database.

The model utility methods provide convenient shortcuts to commonly used methods in the ModelUtil class. In addition to these shortcuts, the ModelUtil class offers functionality for managing client-server operations. This includes methods for connecting to or disconnecting from a COMSOL Multiphysics server or COMSOL Server. For a complete list of available methods, refer to the *Programming Reference Manual*.

Example Code

The code below loads a model using loadModel, presented in the table above. It extracts the x-, y-, and z-coordinates of all **Mesh** nodes and stores them in a 2D

double array coords[3][N], where N is the number of **Mesh** nodes. The individual x-y-, and z- coordinates are available as the length-N 1D arrays coords[0], coords[1], coords[2], respectively. (The node locations can be plotted by using the Cut Point 3D dataset in combination with a 3D Point Trajectories plot.)

```
Model extmodel = loadModel("C:\\Paul\\pacemaker_electrode.mph");
SolverFeature step = extmodel.sol("sol1").feature("v1");
XmeshInfo xmi = step.xmeshInfo();
XmeshInfoNodes nodes = xmi.nodes();
double[][] coords = nodes.coords();
```

For more information on methods operating on the model object, see the *Programming Reference Manual*.

Note that to make the code above platform independent for use in an application you can instead use the common application file folder:

```
Model extmodel = loadModel("common:///pacemaker_electrode.mph");
```

License Methods

The license methods read the license number and check out or test the licenses of the current session or for an MPH file. This functionality can be used, for example, to limit the use of an application to one or a few license numbers or to dynamically adapt functionality of an application depending on which product licenses are available.

NAME	SYNTAX	DESCRIPTION
checkoutLicense	boolean checkoutLicense(String product)	Checks out licenses for all specified products. If not all licenses can be checked out, no licenses are checked out.
checkoutLicenseFor File	boolean checkoutLicenseForFile(String file)	Checks out licenses required to use a given MPH file. If not all required licenses can be checked out, no licenses are checked out.
checkoutLicenseFor FileOnServer	boolean hasLicenseForFileOnServer(String file);	Checks out licenses required to use a given MPH file. If not all required licenses can be checked out, no licenses are checked out.
getLicenseNumber	<pre>String license = getLicenseNumber()</pre>	Returns a string with the license number for the current session.

NAME	SYNTAX	DESCRIPTION
hasProduct	boolean hasProduct(String product)	Returns true if the COMSOL installation contains the software components required for running the specified products. Code completion can be used to get a list of valid product names.
hasProductForFile	boolean hasProductForFile(String file)	Returns true if the COMSOL installation contains the software components required for running the products required by a given MPH file.
hasProductForFileO nServer	<pre>boolean checkoutLicenseForFileOnServer(String file);</pre>	Returns true if the COMSOL installation contains the software components required for running the products required by a given MPH file.

EXAMPLE CODE

The following code tries to check out a license for the AC/DC Module, and displays an error message if it fails:

```
if (!checkoutLicense("ACDC"))
    alert("There seems to be a problem. Please contact Alice and Bob at 123-
456-7890.");
```

You can use this to customize license error messages by calling a method containing this code before any add-on product specific features are used by the application.

In the Application Builder root node you can further select the **Ignore license errors during launch** checkbox. This will make it possible for users to start an application regardless of which licenses are available. However, this will not work if the application, at startup, uses features required by an add-on product. The application has to be created in such a way that the add-on product specific features are not used in the startup phase of the application but instead are deferred to later in the workflow.

To emulate a scenario where there are not enough available licenses you can, for example, disable one or more products in the **Licensed and Used Products in Session** dialog, available from the **File** menu in the COMSOL Desktop environment.

The following code tests if the COMSOL installation has the capability to use an MPH file:

```
boolean ok = hasProductForFile("model_file.mph");
```

Note that even if hasProductForFile return true, in a floating network license situation there may not be any free licenses to check out. If your application is going to process several MPH files and you want to make sure all licenses are checked out before the processing starts, instead use the checkoutLicense or checkoutLicenseForFile methods.

LICENSE FEATURE STRINGS

The following table contains the product strings for all add-on products in the COMSOL 6.3 product suite that can be used by the method checkoutLicense:

PRODUCT/FEATURE	FEATURE NAME
AC/DC Module	ACDC
Acoustics Module	ACOUSTICS
Battery Design Module	BATTERYDESIGN
CAD Import Module	CADIMPORT
CFD Module	CFD
Chemical Reaction Engineering Module	CHEM
Corrosion Module	CORROSION
Design Module	DESIGN, CADIMPORT
ECAD Import Module	ECADIMPORT
Electric Discharge Module	ELECTRICDISCHARGE
Electrochemistry Module	ELECTROCHEMISTRY
Electrodeposition Module	ELECTRODEPOSITION
Fatigue Module	FATIGUE
File Import for CATIA V5	CATIA5
Fuel Cell & Electrolyzer Module	FUELCELLANDELECTROLYZER
Geomechanics Module	GEOMECHANICS
Heat Transfer Module	HEATTRANSFER
Liquid & Gas Properties Module	LIQUIDANDGASPROPERTIES
LiveLink™ for AutoCAD [®]	LLAUTOCAD, CADIMPORT
LiveLink [™] for PTC [®] Creo [®] Parametric [™]	LLCREOPARAMETRIC, CADIMPORT
LiveLink™ <i>for</i> Excel [®]	LLEXCEL
LiveLink™ <i>for</i> Inventor [®]	LLINVENTOR, CADIMPORT
LiveLink™ for MATLAB [®]	LLMATLAB
LiveLink™ <i>for</i> Revit [®]	LLREVIT, CADIMPORT

PRODUCT/FEATURE	FEATURE NAME
LiveLink [™] for Simulink [®]	LLSIMULINK
LiveLink™ <i>for</i> Solid Edge [®]	LLSOLIDEDGE, CADIMPORT
LiveLink TM for SOLIDWORKS [®]	LLSOLIDWORKS, CADIMPORT
Material Library	MATLIB
MEMS Module	MEMS
Metal Processing Module	METALPROCESSING
Microfluidics Module	MICROFLUIDICS
Mixer Module	MIXER
Molecular Flow Module	MOLECULARFLOW
Multibody Dynamics Module	MULTIBODYDYNAMICS
Nonlinear Structural Materials Module	NONLINEARSTRUCTMATERIALS
Optimization Module	OPTIMIZATION
Particle Tracing Module	PARTICLETRACING
Pipe Flow Module	PIPEFLOW
Plasma Module	PLASMA
Polymer Flow Module	POLYMERFLOW
Porous Media Flow Module	POROUSMEDIAFLOW
Ray Optics Module	RAYOPTICS
RF Module	RF
Rotordynamics Module	ROTORDYNAMICS
Semiconductor Module	SEMICONDUCTOR
Structural Mechanics Module	STRUCTURALMECHANICS
Subsurface Flow Module	SUBSURFACEFLOW
Uncertainty Quantification Module	UQ
Wave Optics Module	WAVEOPTICS

File Methods

File methods are used to read and write data to a file or portions of a file. Note that higher-level techniques for reading and writing to files are available from within the Application Builder user interface. For more information, see the book

Introduction to Application Builder and "GUI Command Methods" on page 149.

NAME	SYNTAX	DESCRIPTION
readFile*	String readFile(String name)	Returns the contents in the given filename as a string. The string name is the absolute path to a file or a path given by the file scheme syntax.
openFileStreamReader*	CsReader openFileStreamReader(String name)	Returns a CsReader that can be used to read line-by-line or character-by-character from the given filename.
openBinaryFileStreamReader*	CsBinaryReader openBinaryFileStreamReader(Stri ng name)	Returns a CsBinaryReader that can be used to read from the given file byte-by-byte.
readMatrixFromFile*	double[][] readMatrixFromFile(String name)	Reads the contents of the given file into a double matrix. The file has the same spreadsheet type format as available in the model tree Export node.
readMatrixFromFile*	double[][] readMatrixFromFile(String name, char delimiter)	Reads the contents of the given file into a double matrix, using a user-defined delimiter character. The file has the same spreadsheet type format as available in the model tree Export node.
readStringMatrixFromFile*	<pre>String[][] readStringMatrixFromFile(String name)</pre>	Reads the contents of the given file into a string matrix. The file has the same spreadsheet type format as available in the model tree Export node.
readStringMatrixFromFile*	String[][] readStringMatrixFromFile(String name, char delimiter)	Reads the contents of the given file into a string matrix, using a user-defined delimiter character The file has the same spreadsheet type format as available in the model tree Export node.
readCSVFile*	<pre>String[][] readCSVFile(String name)</pre>	Reads a file with comma- separated values (CSV file) into a string matrix. Expects file to use the RFC 4180 format for CSV.

NAME	SYNTAX	DESCRIPTION
readCSVFile*	<pre>String[][] readCSVFile(String name, char delimiter)</pre>	Reads a file with values, separated by a user-defined delimiter character, into a string matrix.
writeFile*	writeFile(String name, String contents)	Writes the given string contents to the given filename.
writeFile*	<pre>writeFile(String name, String contents, boolean append)</pre>	Writes the given string contents to the given filename. If append is true, then the contents are appended instead of overwritten.
writeFile*	<pre>writeFile(String name, double[][] data)</pre>	Writes the array data to the given file. The spreadsheet format is used, which means it can be read by readMatrixFromFile .
writeFile*	writeFile(String name, double[][] data, boolean append)	Writes the array data to the given file. The spreadsheet format is used, which means it can be read by readMatrixFromFile. If append is true, then the contents are appended instead of overwritten.
writeFile*	<pre>writeFile(String name, String[][] data)</pre>	Writes the array data to the given file. The spreadsheet format is used, which means it can be read by readStringMatrixFromFile.
writeFile*	<pre>writeFile(String name, String[][] data, boolean append)</pre>	Writes the array data to the given file. The spreadsheet format is used, which means it can be read by readStringMatrixFromFile. If append is true, then the contents are appended instead of overwritten.
openFileStreamWriter*	CsWriter openFileStreamWriter(String name)	Returns a CsWriter that can write to the given file.
openFileStreamWriter*	CsWriter openFileStreamWriter(String name, boolean append)	Returns a CsWriter that can write to the given file. If append is true, then the contents are appended instead of overwritten.

NAME	SYNTAX	DESCRIPTION
openBinaryFileStreamWriter*	CsBinaryWriter openBinaryFileStreamWriter(Stri ng name)	Returns a CsBinaryWriter that can be used to write to the given file byte-by-byte.
openBinaryFileStreamWriter*	CsBinaryWriter openBinaryFileStreamWriter(Stri ng name, boolean append)	Returns a CsBinaryWriter that can be used to write to the given file byte by byte. If append is true, then the contents are appended instead of overwritten.
writeCSVFile*	<pre>writeCSVFile(String name, String[][] data)</pre>	Writes the given string array data to a CSV file. The RFC 4180 format is used for the CSV.
writeCSVFile*	writeCSVFile(String name, String[][] data, boolean append)	Writes the given string array data to a CSV file. The RFC 4180 format is used for the CSV. If append is true, then the contents are appended instead of overwritten.
writeCSVFile*	<pre>writeCSVFile(String name, double[][] data)</pre>	Writes the given double array data to a CSV file. The RFC 4180 format is used for the CSV.
writeCSVFile*	<pre>writeCSVFile(String name, double[][] data, boolean append)</pre>	Writes the given double array data to a CSV file. The RFC 4180 format is used for the CSV. If append is true, then the contents are appended instead of overwritten.
exists*	boolean exists(String name)	Tests whether a file with the given name exists. If the name is not a file scheme path name or an absolute path, then the method first finds out whether a file with file scheme path embedded:/// + argument exists. If such a file does not exist, then it tests whether there is a file with a matching name in the current working directory.
deleteFile*	deleteFile(String file)	Delete a file with the given name if it exists. The file is deleted on the server,. The name can use a file scheme path.

NAME	SYNTAX	DESCRIPTION
copyFile*	copyFile(String sourceFile, String destFile)	Copies a file on the server. Both the source and target names can use file scheme paths.
importFile	<pre>importFile(String name) importFile(String name, String[] fileTypes)</pre>	Displays a file browser dialog and uploads the selected file to the file declaration with the given name. After this, the uploaded file can be accessed with upload:/// <name>. The optional fileTypes argument can be used to filter which file types are available for selection in the file browser. The easiest way to get a list of valid file types is to use code completion in the Method Editor. The file types defined by File Type declarations can also be used.</name>
importFile	importFile(ModelEntity entity, String name)	Displays a file browser dialog and uploads the selected file to the Filename text field in the given model object entity. This defines an input file that the application will need at a later stage. For example, the Filename of an interpolation function accessed with model.func(' <tag>')). The uploaded file can be accessed with upload:///<tag>/ filename.</tag></tag>
writeExcelFile*	writeExcelFile(String name, String[][] data)	Writes the given string array data starting from the first cell in the first sheet of an Excel file. This method requires LiveLink™ for Excel [®] .
writeExcelFile*	writeExcelFile(String name, String sheet, String cell, String[][] data)	Writes the given string array data starting from the specified cell in the specified sheet of an Excel file. This method requires LiveLink [™] for Excel [®] .

NAME	SYNTAX	DESCRIPTION
readExcelFile*	<pre>String[][] readExcelFile(String name)</pre>	Reads the first sheet of an Excel file, starting from the first cell, into a String[][]. This method requires LiveLink™ for Excel®.
readExcelFile*	<pre>String[][] readExcelFile(String name, String sheet, String cell)</pre>	Reads the specified sheet of an Excel file, starting from the specified cell, into a String[]]. This method requires LiveLink [™] for Excel [®] .
getFilePath*	String getFilePath(String name)	Returns the absolute server file path of the server proxy file corresponding to a certain file scheme path, or null if the server proxy file for the given path does not exist. This method can be used to pass the path to, for example, a file using the temp:/// scheme to external code or an application. In addition, this method is used to retrieve the filename of an uploaded file when using the file scheme upload:///inputFile, for example by using a File declaration together with a File Import form object.
getClientFileName	<pre>String getClientFileName(String name)</pre>	Returns the original name of an uploaded file on the client file system (or null if there is no uploaded file matching the given file scheme path). This method is only useful for providing user interface feedback. For example, to get information on which uploaded file is being used. There is no guarantee that the original file would still exist on the client or even that the current client would be the same as the original client.

NAME	SYNTAX	DESCRIPTION
getClientFilePath	String getClientFilePath(String name)	Returns the original path of an uploaded file on the client file system (or null if there is no uploaded file matching the given file scheme path). Returns only the filename part of the path when called from an application running in a web browser.
		This method is only useful for providing user interface feedback. For example, to get information on which uploaded file is being used. There is no guarantee that the original file would still exist on the client or even that the current client would be the same as the original client.
createDirectory*	createDirectory(String name)	Creates a file directory name , see examples below in the Example Code section.

All file methods in the table above that are marked with an asterisk (*) support working with data files stored in a Model Manager database. Note that all of these methods automatically load the data file to a working copy directory located on the same computer that COMSOL Multiphysics runs on. This means that any updates made to a data file is not persisted to the database unless explicitly saved as a new file version via the Model Manager database API.

EXAMPLE CODE

This line of code copies the uploaded file file1 to the temp folder with new filename file2.mphbin and then prompts the user to save the file to any location.

```
copyFile("upload:///file1", "temp:///file2.mphbin");
fileSaveAs("temp:///file2.mphbin");
```

This line of code deletes the file file2.mphbin from the temp folder.

deleteFile("temp:///file2.mphbin");

This line of code creates a directory in the user folder, as specified in Preferences. createDirectory("user:///a/b");

This line of code creates a directory in the Temp folder under C:.

```
createDirectory("C:\\Temp\\a\\b");
```

Note that the method fileSaveAs returns a boolean that indicates if saving the file was successful or not. Saving a file will fail if, for example, the user does not have access to the target folder.

Operating System Methods

Operating system methods are used for accessing operating system information and commands from an application.

NAME	SYNTAX	DESCRIPTION
executeOSCommand	String executeOSCommand(String command, String params)	Executes the OS command with the given command (full path) and parameters. Execution times out after a default 180 seconds. Returns everything the process printed to its out stream as a string. When applicable, the command is run server side.
executeOSCommand	<pre>String executeOSCommand(String command, int timeoutSec, String params)</pre>	Executes the OS command with the given command (full path) and parameters. Returns everything the process printed to its out stream as a string. The execution is forcibly stopped after timeoutSec seconds if the command has not finished. To disable the timeout functionality, timeoutSec value 0 can be used. When applicable, the command is run server side.
fileOpen	fileOpen(String name)	Opens the file represented by name with the associated program on the client. Also see the section "Example code".
getUser	<pre>String username = getUser()</pre>	Returns the username of the user that is running the application. If the application is not run from COMSOL Server, then the value of the preference setting General>Username>Name is returned.

NAME	SYNTAX	DESCRIPTION
openURL	openURL(String url)	Opens a URL in the default browser on the client.
playSound	playSound(String name)	Plays the sounds in the given file on the client. Only .wav files are supported; no external libraries are required.
playSound	playSound(double hz, int millis)	Plays a signal at a given frequency hz and with given duration millis in milliseconds on the client.

EXAMPLE CODE

The line of code below plays one of the sounds available in the data/sounds folder of the COMSOL installation and has been embedded in the application and stored in the Sounds library.

playSound("embedded:///success_1.wav");

In the command sequence of a form object, this is equivalent to selecting a sound node under Libraries and clicking **Run**.

The line of code below opens a PDF file embedded in the application and stored in the File library.

```
fileOpen("embedded:///li_ion_battery_impedance.pdf");
```

In the command sequence of a form object, this is equivalent to selecting an **Open File** node under **GUI Commands** > **File Commands** and clicking **Run**, as shown in the figure below.

E Settings X					1
Name:	help			Į.	3
Text:	Help				
lcon:	🛛 help_32	2.png	•	+ [3
Size:	Large				•
Tooltip:					
Keyboard shortcut:					
State					
Visible					
 Choose Comm 	ands to Ru	n		^	Ê
 Image: Save Save Save	Application Application A Application of Application of Pile File As Application Commands ommands	As Copy A on Serv ands	ks ver ver As		
Idit Node ► Run I Plot I Set Value Show Show as Dialog ↑ Import File I Enable Ø Disable					
Command Icon Arguments					
Open file	Open file 📂 embedded:///li_ion_battery_i				
1 ↓ ☴ ५व 🕅	- 🛒				_

This line of code opens the COMSOL home page in the default browser:

openURL("https://www.comsol.com");

This line of code runs an application by means of an OS command:

```
executeOSCommand("C:\\COMSOL63\\Multiphysics\\bin\\win64\\comsol.exe",
    "-run", "C:\\work\\tubular_reactor.mph");
```

Email Methods

Email methods are used for sending emails from an application, typically with attachments containing results from a simulation.

NAME	SYNTAX	DESCRIPTION
emailFromAddress	<pre>String emailFromAddress()</pre>	Returns the email from address from the COMSOL Server or preferences setting.
sendEmail	<pre>sendEmail(String subject, String bodyText)</pre>	Sends an email to the default recipient(s) with the specified subject and body text.
sendEmail	<pre>sendEmail(String subject, String bodyText, ModelEntity modelEntity)</pre>	Sends an email to the default recipient(s) with the specified subject, body text, and zero or more attachments created from Report, Export, and Table nodes in the embedded model.
sendEmail	sendEmail(String toAddress, String subject, String bodyText, ModelEntity modelEntity)	Sends an email to the specified recipient(s) with the specified subject, body text, and zero or more attachments created from Report, Export, and Table nodes in the embedded model.
userEmailAddress	String userEmailAddress()	Returns the user email address(es) corresponding to the currently logged in user, or an empty string if the user has not configured an email address.

Email Class Methods

The class EmailMessage can be used to create custom email messages.

NAME	SYNTAX	DESCRIPTION
EmailMessage	EmailMessage mail = new EmailMessage()	Creates a new EmailMessage object.
EmailMessage.setSer ver	<pre>mail.setServer(String host, int port)</pre>	Sets the email (SMTP) server host and port to use for this email message.

NAME	SYNTAX	DESCRIPTION
EmailMessage.setUse r	mail.setUser(String name, String password)	Sets the username and password to use for email (SMTP) server authentication. This method must be called after the setServer method.
EmailMessage.setSec urity	<pre>mail.setSecurity(String security)</pre>	Sets the connection security type for email (SMTP) server communication. Valid values are 'none', 'starttls', and 'tls'. This method must be called after the setServer method.
EmailMessage.setFro m	<pre>mail.setFrom(String fromAddress)</pre>	Sets the from address.
EmailMessage.setTo	<pre>mail.setTo(String to)</pre>	Sets the to addresses.
EmailMessage.setCc	<pre>mail.setCc(String cc)</pre>	Sets the cc addresses.
EmailMessage.setBcc	<pre>mail.setBcc(String bcc)</pre>	Sets the bcc addresses.
EmailMessage.setSub ject	<pre>mail.setSubject(String subject)</pre>	Sets the email subject line. Note that newline characters are not allowed.
EmailMessage.setBod yText	<pre>mail.setBodyText(String body)</pre>	Sets the email body as plain text. An email can contain both a text and an HTML body.
EmailMessage.setBod yHtml	<pre>mail.setBodyHtml(String body)</pre>	Sets the email body as HTML text. An email can contain both a text and an HTML body.
EmailMessage.attach File	<pre>mail.attachFile(String filename)</pre>	Adds an attachment from a file. The attachment MIME type is determined by the filename extension.
EmailMessage.attach File	<pre>mail.attachFile(String filename, String mimeType)</pre>	Adds an attachment from a file with the specified MIME type.
EmailMessage.attach FromModel	<pre>mail.attachFromModel(ModelEntity modelEntity)</pre>	Adds an attachment created from a report, export, or table feature in the model.
EmailMessage.attach Text	<pre>mail.attachText(String text, String mimeSubType)</pre>	Adds a text attachment with a specified sub-MIME type, such as plain or HTML.

NAME	SYNTAX	DESCRIPTION
EmailMessage.attach Binary	<pre>mail.attachBinary(byte[] binary, String mimeType)</pre>	Adds an attachment from a byte array with the specified MIME type.
EmailMessage.send	mail.send()	Sends the email to the email (SMTP) server. An email object can only be sent once.

Each to, cc, and bcc address string can contain multiple email addresses separated by a comma or a semicolon character. Whitespace is allowed before and after the separator character.

EMAIL PREFERENCES

To set general email preferences, open the **Email** page of the **Preferences** window. There you can specify a **From address** and a **Default to address**. To set preferences for an outgoing email (SMTP) server, open the **Outgoing Server (STMP)** page under the **Email** page, as shown in the figure below.

D P	Preferences			×
	heferences Application Builder Chatbot Client-Server Computing Email Outgoing Server (SMTP) Files Geometry Graphics Help Libraries Libraries Libraries Libraries Save Security Updates User Interface	Outgoing Server (S Host: Port: Connection security: User: Password:	MTP) sever.myorganization.com 25 None • paul@myorganization.com	×
Fa	ctory Settings for All Import	Export		Factory Settings OK Cancel

COMSOL Server provides a similar set of email preferences.

EXAMPLE CODE

The following code configures the email server settings, sends an email, and attaches a report:

```
/**
* Sends an email with the simulation report attached.
*/
EmailMessage mail = new EmailMessage();
// Custom email server settings used
if (isOverrideEmail) {
 mail.setServer(emailServerHost, emailServerPort);
  mail.setUser(emailUser, util1.password);
  mail.setSecurity(emailSecurity);
  mail.setFrom(emailFromAddress);
}
mail.setTo(emailTo);
mail.setSubject(translate("Tubular reactor simulation", true));
mail.setBodyText(translate("The computation has finished. please find the
report attached"));
mail.attachFromModel(model.result().report("rpt1"));
mail.send();
```

This code is run in the Tubular Reactor application, which is available as an application example in the Application Libraries. The figure below shows part of the user interface with an input field for the email address.

✓ When Solved	
✓ Play sound Email report to:	
	දිුාි Email Settings



The figure below shows the corresponding form object and Settings window.

GUI-Related Methods

The graphical user interface (GUI) related methods are used for displaying dialogs with messages, editing form objects and user interface content, getting run-time properties of the application user interface, and running methods.

NAME	SYNTAX	DESCRIPTION
Call a method directly	<methodname>()</methodname>	Call a method from the Methods list by using its name, for example, method1(), method2().
callMethod	callMethod(String name)	Alternate way to call a method from the Methods list; used internally and in cases of name collisions.

NAME	SYNTAX	DESCRIPTION
useGraphics	useGraphics(ModelEntity entity, String name) useGraphics(ModelEntity entity, FormObject graphics)	Plots the given entity (Plot Group, Geometry, Mesh, Explicit Selection or Player Animation) in the graphics form object given by the name, name path, or graphics form object in the second argument.
openForm	openForm(String name)	Shows the form with the given name in the current main window. In a single window application, the form replaces the current one. In an application with subwindows, the form must exist in the main window layout. If not, it will not be opened. Showing a form that is already open will only activate the form.
closeForm	closeForm(String name)	Closes the form with the given name. Closing forms is only possible in applications using subwindows. This method is not applicable for single window applications.
closeDialog	closeDialog(String name)	Closes the form, shown as a dialog, with the given name.
dialog	dialog(String name)	Shows the form with the given name as a dialog. Equivalent to the dialog method of a Form object; see below.
alert	alert(String text)	Stops execution and displays an alert message with the given text.
alert	alert(String text, String title)	Stops execution and displays an alert message with the given text and title.

NAME	SYNTAX	DESCRIPTION
confirm	String confirm(String text)	Stops execution and displays a confirmation dialog with the given text. It also displays two buttons, "Yes" and "No". The method returns "Yes" or "No" depending on what the user clicks.
confirm	String confirm(String text, String title)	Stops execution and displays a confirmation dialog with the given text and title. It also displays two buttons, "Yes" and "No". The method returns "Yes" or "No" depending on what the user clicks.
confirm	String confirm(String text, String title, String yes, String no)	Stops execution and displays a confirmation dialog with the given text and title. It also displays two buttons with the given strings as labels. The method returns the label of the button that the user clicks.
confirm	String confirm(String text, String title, String yes, String no, String cancel)	Stops execution and displays a confirmation dialog with the given text and title. It also displays three buttons with the given strings as labels. The method returns the label of the button that the user clicks.
error	error(String message)	Stops execution and opens an error dialog with the given message.

NAME	SYNTAX	DESCRIPTION
error	error(String message, Throwable cause)	Stops execution and opens an error dialog with the given message including the underlying cause of class Throwable , the general Java exception class, or one of its subclasses, such as RuntimeException . This can be used to "wrap" native COMSOL Multiphysics error messages with custom error messages.
request	String request(String text)	Stops execution and displays a dialog with a text field, requesting input from the user. The given text is the label of the text field. The method returns the entered text or null if the cancel button is clicked.
request	String request(String text, String defaultString)	Stops execution and displays a dialog with a text field, requesting input from the user. The given text is the label of the text field and the default string is the text initially shown in the text field. The method returns the entered text or null if the cancel button is clicked.
request	String request(String text, String title, String defaultString)	Stops execution and displays a dialog with a text field, requesting input from the user. The given text is the label of the text field, the default string is the text initially shown in the text field, and the title is the title of the dialog. The method returns the entered text or null if the cancel button is clicked.

NAME	SYNTAX	DESCRIPTION
message	message(String message)	Sends a message to the message log if available in the application.
message	message(arg)	Sends a message arg to the message log. For an application this requires that a message log is added to the application user interface. The input argument arg can be a scalar, 1D array, or 2D array of the types string, double, int, or Boolean.
clearLog	clearLog()	Clears the log window.
clearMessageLog	clearMessageLog()	Clears the message log window.
evaluateToResultsTable	evaluateToResultsTable(NumericalFe ature entity, String name, boolean clear) evaluateToResultsTable(NumericalFe ature entity, FormObject graphics, boolean clear)	Evaluates the given entity, a Derived Value, in the table object given by the name, name path, or graphics form object in the second argument, which will then be the default target for the evaluations of the Derived Value. If the third argument is true, the table is cleared before adding the new data, otherwise the data is appended.
evaluateToDoubleArray2D	double[][] evaluateToDoubleArray2D(NumericalF eature entity)	Evaluates the given entity, a Derived Value, and returns the nonparameter column part of the real table that is produced as a double matrix. All settings in the numerical feature are respected, but those in the current table connected to the numerical feature are ignored.

NAME	SYNTAX	DESCRIPTION
evaluateToIntegerArray2D	int[][] evaluateToIntegerArray2D(Numerical Feature entity)	Evaluates the given entity, a Derived Value, and returns the nonparameter column part of the real table that is produced as an integer matrix. All settings in the numerical feature are respected, but those in the current table connected to the numerical feature are ignored.
evaluateToStringArray2D	<pre>String[][] evaluateToStringArray2D(NumericalF eature entity)</pre>	Evaluates the given entity, a Derived Value, and returns the nonparameter column part of the, potentially complex, table that is produced as a string matrix. All settings in the numerical feature are respected but those in the current table connected to the numerical feature are ignored.
useResultsTable	useResultsTable(TableFeature tableFeature, String resultsTable) useResultsTable(TableFeature tableFeature, FormObject resultsTable)	Shows the values from the tableFeature in the resultsTable form object.
getChoiceList	ChoiceList getChoiceList(String name)	Returns an object of the type ChoiceList, representing a choice list node under the declarations branch. The type ChoiceList has methods that make it easier to change the matrix value with respect to changing and accessing values and display names individually.
setFormObjectEnabled	<pre>setFormObjectEnabled(String name, boolean enabled)</pre>	Sets the enable state for the form object specified by the name or name path.
setFormObjectVisible	setFormObjectVisible(String name, boolean visible)	Sets the visible state for the form object specified by the name or name path.

NAME	SYNTAX	DESCRIPTION
setFormObjectText	setFormObjectText(String name, String text)	Sets the text for the form object specified by the name or name path in the second argument. This method throws an error if it is impossible to set a text for the specified form object.
setFormObjectEditable	setFormObjectEditable(String name, boolean editable)	Sets the editable state for the form object specified by the name or name path. This functionality is only available for text field objects.
setMenuBarItemEnabled	setMenuBarItemEnabled(String name, boolean enabled)	Sets the enable state for the menu bar item specified by the name or name path (from menu bar) in the first argument.
setMainToolbarItemEnabled	setMainToolbarItemEnabled(String name, boolean enabled)	Sets the enable state for the main toolbar item specified by the name or name path (from main toolbar) in the first argument.
setFileMenuItemEnabled	<pre>setFileMenuItemEnabled(String name, boolean enabled)</pre>	Sets the enable state for the file menu item specified by the name or name path (from file menu) in the first argument.
setRibbonItemEnabled	<pre>setRibbonItemEnabled(String name, boolean enabled)</pre>	Sets the enable state for the ribbon item specified by the name or name path (from main window) in the first argument.
setToolbarItemEnabled	setToolbarItemEnabled(String name, boolean enabled)	Sets the enable state for the toolbar form object item specified by the name or name path in the first argument.
useView	useView(View view, String name) useView(View view, FormObject graphics)	Applies a view to the graphics contents given by the name, name path, or graphics form object in the second argument.

NAME	SYNTAX	DESCRIPTION
resetView	resetView(String name) resetView(FormObject graphics)	Resets the view to its initial state in the graphics contents given by the name, name path, or graphics form object in the second argument.
getView	ViewBase getView(String name) ViewBase getView(FormObject graphics)	Returns the view currently used by the graphics contents given by the name, name path, or graphics form object in the second argument.
goToView	goToView(String name);	Goes to a standard view in main graphics window. The parameter name is one of the view orientation strings in the following list: "xy", "xz", "yx", "yz", "zx" , "zy".
goToView	goToView(String name, String graphicsname);	Goes to a standard view in the given graphics form object graphicsname.
goToView	goToView(String name, FormObject graphics);	Goes to standard view in the given graphics form object graphics.
setWebPageSource	setWebPageSource(String name, String source)	Sets the source for the form object specified by the name or name path in the first argument. This method throws an error if the name does not refer to a Web Page form object.
getScreenHeight	int getScreenHeight()	Returns the height in pixels of the primary screen on client system, or of the browser window if Web Client is used.
NAME	SYNTAX	DESCRIPTION
----------------	---------------------------	---
getScreenWidth	int getScreenWidth()	Returns the width in pixels of the primary screen on client system, or of the browser window if Web Client is used.
storeChanges	storeChanges(String form)	Commits the values entered in a dialog having On request enabled. The method takes the name of the form as its argument.

Alerts and Messages

The methods alert, confirm, and request display a dialog with a text string and optional user input. The following example uses confirm to ask the user if a direct or an iterative solver should be used in an application. Based on the answer, the alert function is then used to show the estimated memory requirement for the selected solver type in a message dialog:

```
String answer = confirm("Which solver do you want to use?",
"Solver Selection","Direct", "Iterative");
if (answer.equals("Direct")) {
    alert("Using the direct solver will require about 4GB of memory when solving.");
} else {
    alert("Using the iterative solver will require about 2GB of memory when
    solving.");
}
```

Example Code

The following code changes the camera zoom angle and updates the graphics for each change.

```
useView(model.view("view1"), "/form1/graphics1");
for (int i = 0; i < 25; i++) {
    sleep(2000);
    model.view("view1").camera().set("zoomanglefull", 12-i*5.0/25);
    useGraphics(model.geom("geom1"), "/form1/graphics1");
}
```

This line of code displays plot group 5 (pg5) in the graphics object graphics1 in the form with the name Temperature:

```
useGraphics(model.result("pg5"), "/Temperature/graphics1");
```

The code below displays the mesh in the model tree node mesh1 in the graphics object graphics1 contained in the card of a card stack. The second line runs a zoom extents command to ensure proper visualization of the mesh.

```
useGraphics(model.mesh("mesh1"), "/mesh/cardstack1/card1/graphics1");
```

```
zoomExtents("/mesh/cardstack1/card1/graphics1");
```

To clear the contents of a graphics object use a call such as

```
useGraphics(null, "/form1/graphics1");
```

The code below displays a request dialog that lets the user type in a filename for an HTML report. If the user has typed a filename, then a report is generated.

```
String answerh = request("Enter filename", "Filename", "Untitled.html");
if (answerh != null) {
  model.result().report("rpt1").set("format", "html");
  model.result().report("rpt1").set("filename", "user:///"+answerh);
  model.result().report("rpt1").run();
}
```

The code below is similar to the code above, but in this case the report is saved in $Microsoft^{\$}$ Word^{\$} format (.docx).

```
String answerw = request("Enter filename", "Filename", "Untitled.docx");
if (answerw != null) {
  model.result().report("rpt2").set("format", "docx");
  model.result().report("rpt2").set("filename", "user:///"+answerw);
  model.result().report("rpt2").run();
}
```

This line of code sets the view of the graphics object form1/graphics1 to **View 5**, as defined in the model tree:

```
useView(model.view("view5"), "form1/graphics1");
```

You can use **Data Access** in combination with **Editor Tools** to create a slider or an input field that sets the transparency level (alpha) of a plot group. The figure below shows a **Settings** window of a slider with the transparency level as **Source**.

Settings ×	Editor Tools × 🔹 🖛 🖛	
Slider		
Name:	slider1 📃	
Minimum value:	0	
Maximum value:	1	
Number of steps:	5	
Orientation:	Horizontal 🔹	
Tooltip:		
 Source 	⊕ + ℃	
 Global Definitions Thermal Actuator (comp1) {comp1} E Definitions View 1 {view1} Transparency (transparency) Transparency (transparencylevel) 		
Use as Source Selected source:	₹ y Edit Node	
8.5 View 1 {vie	w1}⇒Transparency (transparencylevel)	
Initial value: Fr	om data source 🔹 🔻	

In this case you need to create a method for updating the view that is called to handle an event from the slider or form object. In the example above, the slider uses a **Local method** defined in the **Events** section. This method contains one line of code that updates the view:

```
useView(getView("/form1/graphics1"), "/form1/graphics1");
```

Note that different transparency levels are not supported when accessing an application from a browser using COMSOL Server.

Note that you can also set a view from the command sequence of, for example, a button: select a view subnode under the **Views** node in the editor tree and click the **Plot** button under the tree.

 Choose Commands to Rule 	n		~8
> 🖻 Forms			
> 🖻 Local Forms			
> 💽 GUI Commands			
> iiii Libraries			
🗸 < Model (root)			
> 🌐 Global Definitions			
 Thermal Actuator (cor 	mp1) {	comp1}	
✓			
> 🗞 Selections			
> 🧄 View 1 {view1}			
✓ √↓ View 6 {view6}			
Directiona	l Light	1 {lgt1}	
Directional Light 2 {lgt2}			
Directional Light 3 {lgt3}			
> 🕁 View 7 {view7}			
> 🖄 Geometry 1 {geon	n1}		\sim
🖅 Edit Node 🕨 Run 🕅 Plot	🖉 Se	Value Show	
🔚 Show as Dialog 🕇 Import Fi	le 💿	Enable ⊘ Disable	
** Command	lcon	Arguments	
Use View 6 {view6}		graphics/graphics1	
↑↓≒७⊒▼∦			

To go to one of the standard views in the main Graphics window, for example in an add-in, you can use:

```
goToView("xy");
```

In an application you can similarly use one of:

```
goToView("xz", "form1/graphics1");
goToView("yz", app.form("form1").formObject("graphics1"));
```

to go to the graphics object graphics1 in the form form1.

This line of code sets the URL source of the form object webpage1 to the COMSOL web page:

```
setWebPageSource("/form1/webpage1", "https://www.comsol.com");
```

This line of code forms a string containing the screen width and height:

```
screenSize = toString(getScreenWidth()) + "-by-" +
toString(getScreenHeight());
```

You can present the string with an input field or a data display object using this string as a source (the string screenSize needs to be declared first).

The GUI command methods correspond to the **GUI Commands** node in the editor tree. The editor tree is displayed in, for example, the **Choose Commands to Run** section in the **Settings** window for a button object in the Application Builder.

NAME	SYNTAX	DESCRIPTION
clearAllMeshes	clearAllMeshes()	Clears all meshes.
clearAllSolutions	clearAllSolutions()	Clears all solutions.
clearSelection	<pre>clearSelection(String graphics) clearSelection(FormObject graphics)</pre>	Clears the selection in the given graphics object.
environmentReflections	environmentReflections(String graphicsName) environmentReflections(FormObject graphics)	Adds environment reflections to the given graphics object. This method is run as a toggle action.
exit	exit()	Exits the application.
fileOpen	fileOpen(String name)	Opens a file with the associated program on the client.
fileSaveAs	boolean fileSaveAs(String file)	Downloads a file to the client. See also the section "Example code". The method returns a boolean that indicates if the operation was successful or not.
printGraphics	<pre>printGraphics(String graphicsName) printGraphics(FormObject graphics)</pre>	Prints the given graphics object.
rotateEnvironment	rotateEnvironment(String graphicsName) rotateEnvironment(FormObject graphics)	Rotates the environment, that is used for realistic reflections, to the given graphics object. This method is run as a toggle action.
saveApplication	<pre>saveApplication(boolean confirm)</pre>	Saves the application. The boolean argument determines if the user is prompted with a confirmation dialog before saving.
saveApplicationAs	boolean saveApplication()	Saves the application under a different name. If the return value is false then the user canceled the operation and the application was not saved.

NAME	SYNTAX	DESCRIPTION
saveApplicationCopyAs	boolean saveApplicationCopyAs()	Saves a copy of the application. When running the method the user is presented with a file browser dialog where they can select where to save the copy. The method returns a boolean value where true indicates a successful save.
saveApplicationOnServe r	saveApplicationOnServer(boolean confirm)	Saves the application on server. The boolean argument determines if the user is prompted with a confirmation dialog before saving.
saveApplicationOnServe rAs	boolean saveApplicationOnServerAs()	Saves the application on server under a different name. (Or as an MPH file.) If the return value is false then the user canceled the operation and the application was not saved.
scenelight	<pre>sceneLight(String graphicsName) sceneLight(FormObject graphics)</pre>	Toggles scene light in the given graphics object.
selectAll	<pre>selectAll(String graphics) selectAll(FormObject graphics)</pre>	Sets the selection to all entities in the given graphics object.
skybox	skybox(String graphicsName) skybox(FormObject graphics)	Adds visualization of the skybox, used for realistic reflections, to the given graphics object. This method is run as a toggle action.
transparency	<pre>transparency(String graphicsName) transparency(FormObject graphics)</pre>	Toggles transparency in the given graphics object.
zoomExtents	<pre>zoomExtents(String graphicsName) zoomExtents(FormObject graphics)</pre>	Makes the entire model visible within the extent of the given graphics object.

EXAMPLE CODE

For examples of how to use fileSaveAs, see the Introduction to Application Builder. This method is frequently needed for saving files in general.

You can create an application that saves and exits automatically by running the following lines of code, for example, after solving:

```
saveModel(model,"C:\\COMSOL\\file.mph");
exit();
```

```
Or
   saveModel(model, "common:///file.mph");
   exit();
```

This is useful in a COMSOL Server setting since using exit() in this way will free up any licenses that are checked out.

Debug Methods

The debug method is used to display variable contents in the **Debug Log** window.

NAME	Syntax	DESCRIPTION
clearDebugLog	<pre>clearDebugLog()</pre>	Clears the Debug Log window.
debugLog	debugLog(arg)	Prints the value of arg to the Debug Log window. The input argument arg can be a scalar, ID array, or 2D array of the types string, double, int, or Boolean.

EXAMPLE CODE

The code below prints strings and doubles to the Debug Log window.

```
xcoords[i] = Math.cos(2.0*Math.PI*divid);
ycoords[i] = Math.sin(2.0*Math.PI*divid);
debugLog("These are component values for case 1:");
debugLog("x:");
debugLog(xcoords[i]);
debugLog("y:");
debugLog(ycoords[i]);
```

Note: As an alternative to debugLog, you can use the message method to send display strings to the **Message** window in COMSOL Desktop or to a **Message Log** object in an app.

Methods for External C Libraries

The methods for external C libraries are used for linking Application Builder methods with compiled C-code.

External Method

NAME	SYNTAX	DESCRIPTION
external	External external(String name)	Returns an interface to an external C (native) library given by the name of the library feature. The External class uses the Java Native Interface (JNI) framework.

For more information, see the Application Builder Reference Manual.

METHODS RETURNED BY THE EXTERNAL METHOD

The external method returns an object of type External with the following methods:

NAME	SYNTAX	DESCRIPTION
invoke	long invoke(String method, Object arguments)	Invokes the named native method in the library with the supplied arguments. Strings are converted to char *. Returns the value returned by the method.
invokeWideString	long invokeWideString(String method, Object arguments)	Invokes the named native method in the library with the supplied arguments. Strings are converted to wchar_t *. Returns the value returned by the method.
close	void close()	Releases the library and frees resources. If you do not call this method, it is automatically invoked when the external library is no longer needed.

Progress methods are used to create and update progress information in the Status bar, in a progress form object, and in a dialog.

NAME	SYNTAX	DESCRIPTION
setProgressInterval	<pre>setProgressInterval(String message, int intervalStart, int intervalEnd)</pre>	Sets a progress interval to use for the top-level progress and display message at that level. The top level will go from intervalStart to intervalEnd as the second level goes from 0 to 100. As the second level increases, the top level is increased by (intervalEnd - intervalStart) * (second level progress (0-100) / 100). The value for intervalStart must be between 0 and intervalEnd, and the value for intervalStart and 100. Calling this method implicitly resets any manual progress previously set by calls to setProgress().
setProgress	<pre>setProgress(int value, String message)</pre>	Sets a value for the user- controlled progress level. By default, this is the top level, but if a progress interval is active (setProgressInterval has been called and resetProgress has not been called after that), then it is the second level.
setProgress	<pre>setProgress(int value)</pre>	Same as setProgress(message, value), but uses the latest message or an empty string (if no message has been set).

NAME	SYNTAX	DESCRIPTION
resetProgress	resetProgress()	Removes all progress levels and resets progress to 0 and the message to an empty string.
showIndeterminateProgr ess	<pre>showIndeterminateProgress(String message)</pre>	Shows a progress dialog with an indeterminate progress bar, given message and a cancel button.
showIndeterminateProgr ess	<pre>showIndeterminateProgress(String message, boolean cancelButton)</pre>	Shows a progress dialog with an indeterminate progress bar, given message and an optional cancel button.
showProgress	showProgress()	Shows a progress dialog with a cancel button. No model progress is included.
showProgress	<pre>showProgress(boolean modelProgress)</pre>	Shows a progress dialog with a cancel button and an optional model progress.
showProgress	showProgress(boolean modelProgress, boolean addSecondLevel)	Shows a progress dialog with a cancel button, optional model progress, and one or two levels of progress information. Two levels can only be used if modelProgress is true.
showProgress	showProgress(boolean modelProgress, boolean addSecondLevel, boolean cancelButton)	Shows a progress dialog with optional model progress, one or two levels, and possibly a cancel button. Two levels can only be used if modelProgress is true.
closeProgress	closeProgress()	Closes the currently shown progress dialog.

NAME	SYNTAX	DESCRIPTION
startProgress	startProgress(String name) startProgress(FormObject progressBar)	Resets the value of the given progress bar form object name to 0. The progress bar to control can be specified with an absolute path, such as form1/progressbar1, or a name relative to the context from which the method was called. Nothing is done if no progress bar corresponding to the given name is found.
setProgressBar	<pre>setProgressBar(String name, int workDone, String message) setProgressBar(FormObject progressBar, int workDone, String message)</pre>	Sets the value of the given progress bar form object name in the range 0-100 and the associated progress message. Values out of range are converted to 0 or 100. The progress bar to control can be specified with an absolute path, such as form1/progressbar1, or a name relative to the context from which the method was called. Nothing is done if no progress bar corresponding to the given name is found, or if the progress bar is used for showing model progress.
setProgressBar	<pre>setProgressBar(String name, int workDone) setProgressBar(FormObject progressBar, int workDone)</pre>	Same as above, but does not update the progress message.

EXAMPLE CODE

```
showProgress(true, true, true);
/* Opens a progress dialog with cancel button showing two levels of progress.
The values shown in progress dialog will be updated to match the two levels
of progress. */
```

```
setProgressInterval("Preparing application", 0, 20);
/* Sets the current progress scale to go from 0 to 20. This means that the
top-level progress will go from 0 to 20 when second-level progress goes from
0 to 100. */
```

```
setProgress(0, "Init step 1");
  /* Sets the second-level progress to 0 and the second-level progress message
  to "Init step 1". */
  // do some work
  setProgress(40):
  /* Sets the second-level progress to 40, this causes the top-level progress
  to be updated to 8 (40% of 20). */
  // do some work
  setProgress(80, "Init step 2");
  /* Sets the second-level progress to 80 and the progress message to "Init
  step 2". The top-level message is still "Preparing application" and top-
  level progress is now 16. */
  // do some work
  setProgressInterval("Meshing", 20, 40);
  /* Sets the top-level interval to 20 - 40 and the progress message to
  "Meshing" at this point the value shown at the top-level will be 20. The
  second-level progress is cleared when the top-level interval is changed. */
  <call-meshing algorithm here>
  /* The progress messages and values from the meshing algorithm are shown at
  the second-level progress. The top-level progress message will be "Meshing",
  but the top-level progress advances from 20 to 40 while second-level
  progress advances from 0 to 100. */
  setProgressInterval("Solving", 40, 100);
  /* The top-level progress message is changed to "Solving" and its value to
  40.
  <call-solver>
  /* Similar to meshing, the progress messages and values from the solver are
  shown in the second-level progress bar and the top-level progress value goes
  from 40 to 100 while the solver progress goes from 0 to 100. */
  closeProgress();
Application Progress Information
```

Progress information can be displayed in three different ways: in the Status bar, in a progress form object, and in a dialog. Application progress information is controlled by the setProgress methods, which take as their input an integer between 0 and 100 and an optional message. The integer represents how far the displayed progress bar has progressed. If no message is supplied, the last message provided is used. For example:

```
setProgress(10, "Computing data");
setProgress(25);
```

This will keep Computing data as the progress message.

Use the setProgress method by itself if you want to display custom progress in the task and status bar. Once you have done this, that progress bar will no longer be updated by progress information from the COMSOL model, but will be completely dependent on further calls to setProgress for changes in its value. Precede it with a call to showProgress to also display the built-in progress dialog, see below.

Note that progress information from the COMSOL model will not be shown in between calls to setProgress. Progress is reset between method calls. If you want to combine custom steps of progress in methods with built-in model progress, then use setProgressInterval instead.

With setProgressInterval, you can control the top two levels of progress information. The second level can be displayed in a progress dialog and a progress bar form object, see the code segment below. The second progress level, controlled by your own custom progress calculation, is connected to the first level such that one interval at the top level corresponds to the entire second level. Thus if the interval is 0–50, when the second level progress reaches 40, for example, the first level will be set to $20 \ (=(40/100)*50)$.

Important uses of the method setProgressInterval are listed below:

- Combining calls to the COMSOL model so that you get continuous progress going from 0–100.
- Computing several studies as well as evaluating several plots. Call setProgressInterval before each call to the built-in methods with an interval that approximates how much time each model computation takes. For example:

```
setProgressInterval("Computing solution", 0, 80);
model.study("std1").run();
setProgressInterval("Plotting", 80, 100);
useGraphics(model.result("pg3"), "energy_response_plot/graphics1");
```

• Combining one or more calls to built-in COMSOL methods with custom methods that in themselves take significant time. In this case, use setProgressInterval as in the previous example, followed by your own custom code with appropriate calls to setProgress. These calls should run from 0 to 100 as they are controlling the second progress level. For example:

```
setProgressInterval("Computing solution", 0, 60);
model.study("std1").run();
setProgressInterval("Working", 60, 80);
setProgress(0, "Specific message about what I'm doing");
// ...
// Code that does something
// ...
setProgress(60);
```

If you, in a running application, wish to no longer use progress intervals, call resetProgress to return to the original state. This will also reset progress to 0.

The Progress Dialog

A progress dialog can be used to display application progress as described in the previous section. The progress dialog has the following options:

- Whether to show model progress or not. When off, no progress from the model part of the application is forwarded to the progress dialog.
- Whether to show one or two progress levels in the progress dialog.
- Whether to include a cancel button. Cancel also works for user-defined methods, as it halts execution when the next line in the method is reached.

Use the showProgress methods to enable or disable these options. To close the progress dialog, use the closeProgress method.

You can show a progress dialog with an indeterminate progress bar that keeps spinning until you close the progress dialog. Only one progress dialog can be shown at a time. Use the showIndeterminateProgress methods to display this progress dialog.

The Progress Bar Form Object

The **Progress Bar** form object can either show overall application progress information or customized partial progress information. If you have selected the **Include model progress** checkbox in the **Settings** window of the **Main Window** node, then the overall application progress information becomes available.

When **Include model progress** is selected, the progress bar will show the same information as the progress dialog. That is, one or two levels of progress information and a cancel button, depending on the settings in the form object.

When **Include model progress** is cleared, you control the progress bar through the setProgressBar methods. These take the path name of the progress bar form object, for example, main/progressbar1.

The date and time methods are used to retrieve the current date and time as well as information on computation times.

NAME	SYNTAX	DESCRIPTION
currentDate	<pre>String currentDate()</pre>	Returns the current date as a string (formatted according to the server's defaults) for the current date.
currentTime	<pre>String currentTime()</pre>	Returns the current time as a string (not including date, and formatted according to the server defaults).
formattedDateTime	<pre>String dateString = formattedDateTime(long epochInMs)</pre>	Returns a formatted and readable date and time from an input given as milliseconds since the epoch.
formattedTime	String formattedTime(long timeInMs, String format)	Returns a formatted time using the given format. The format can either be a time unit or a text describing a longer format. Supported formats are:
		'hr:min:sec', which returns the time in hours, minutes, and seconds in the form X hr Y min Z sec.
		'h:min:s', which returns the time in hours, minutes, and seconds in the form X h Y min Z s.
		'detailed', which returns the time in seconds and also includes more readable units for longer times.
sleep	<pre>sleep(long timeInMs)</pre>	Sleep for the specified number of milliseconds.
timeStamp	long timeStamp()	Current time in milliseconds since midnight, January 1, 1970 UTC.

NAME	SYNTAX	DESCRIPTION
getExpectedComputa tionTime	<pre>model.setExpectedComputationTime(String format)</pre>	Returns a string describing the approximate computation time of the application. The string can be altered by the method setExpectedComputationTi me.
setLastComputation Time	<pre>model.setLastComputationTime(long time)</pre>	Set the last computation time, overwriting the automatically generated time. You can use the timeStamp method to record time differences and set the measured time in ms (a long integer).
getLastComputation Time	<pre>String model.getLastComputationTime (String format)</pre>	Returns the last computation time in the given format. The format can either be a time unit or text describing a longer format. Currently supported formats are: hr:min:sec Returns the time in hours, minutes, and seconds in the format X h Y min Z sec. h:min:s Returns the time in hours, minutes, and seconds in the format X h Y min Z s. detailed Returns the time in seconds and also includes more readable units for longer times. This format is localized and the output is translated to the current language setting. For example, you can retrieve the time in ms by using getLastComputationTime ("m s").

EXAMPLE CODE

The following code overrides the built-in computation time that is available in the information nodes in the model tree.

```
long t0 = timeStamp(); // initialize record of computation time
// code and computations
```

model.setLastComputationTime(timeStamp()-t0); // record computation time

If it is possible to give a rough estimate of the computation time based on the given inputs of an application, you can update the expected computation time and display it in an information card stack or a text object. Assume that there is an integer input called objects that controls the number of objects in a geometry array and that the computation roughly increases linearly with this number. The following code adjusts the expected computation time accordingly.

```
// Number of minutes of computation time per object
int minutes = objects*2.1;
model.setExpectedComputationTime("About " + minutes + " minutes" );
```

Sleep

The code below makes the application idle for 1000 ms.

long delay = 1000; sleep(delay);

This technique can be used to display graphics in a sequence.

For more information on information nodes and information cards, as well as the sleep method, see the book *Introduction to Application Builder*.

Conversion Methods

Conversion methods are used to convert between the different data types Booleans, integers, doubles, strings, and arrays. These methods are shorthand versions of conversion methods in the standard Java libraries.

NAME	SYNTAX	DESCRIPTION
toBoolean	boolean toBoolean(String str)	Converts the given string to a Boolean. (' true ' returns true, all other strings return false).
toBoolean	boolean [] toBoolean(String[] strs)	Converts all the strings in the given array to Booleans (' true ' retums true, all other strings return false) and returns a Boolean array.
toBoolean	boolean [][] toBoolean(String[][] strs)	Converts all the strings in the given matrix to Booleans (' true ' returns true, all other strings return false) and returns a Boolean matrix.
toDouble	double toDouble(String str)	Converts the given string to a double.
toDouble	<pre>double[] toDouble(String strs)</pre>	Converts all the strings in the given array to doubles and returns a double array.
toDouble	<pre>double[][] toDouble(String[] strs)</pre>	Converts all the strings in the given matrix to doubles and returns a double matrix.
toDouble	double toDouble(float flt)	Converts the given float to a double.
toDouble	<pre>double[] toDouble(float flt)</pre>	Converts all the floats in the given array to doubles and returns a double array.
toDouble	<pre>double[][] toDouble(float[] flt)</pre>	Converts all the floats in the given matrix to doubles and returns a double matrix.
toInt	<pre>int toInt(String str)</pre>	Converts the given string to an integer.
toInt	<pre>int[] toInt(String strs)</pre>	Converts all the strings in the given array to integers and returns an integer array.

NAME	SYNTAX	DESCRIPTION
toInt	<pre>int[][] toInt(String[] strs)</pre>	Converts all the strings in the given matrix to integers and returns an integer matrix.
toString	<pre>String toString(int value)</pre>	Converts the given integer to a string.
toString	<pre>String toString(double value)</pre>	Converts the given double to a string.
toString	<pre>String toString(boolean value)</pre>	Converts the given Boolean to a string.
toString	String toString(double value, int digits)	Converts the given double to a string with the given number of significant digits.
toString	String toString(double value, int digits,boolean remove)	Converts the given double to a string with the given number of significant digits with trailing zeros removed if the Boolean remove is true. For example, 10.000001 with number of digits set to 3 will return 10 rather than 10.000.
toString	String toString(double value, String format)	Converts the given double to a string using the given format specifier, which is the same as java.util.Formatter. See the corresponding Java format string documentation for more information.
toString	<pre>String[] toString(double[] darray)</pre>	Converts all the doubles in the given array to strings and returns a string array.
toString	<pre>String[][] toString(double[][] dmatrix)</pre>	Converts all the doubles in the given matrix to strings and returns a string matrix.
toString	<pre>String[] toString(int[] iarray)</pre>	Converts all the integers in the given array to strings and returns a string array.
toString	<pre>String[][] toString(int[][] imatrix)</pre>	Converts all the integers in the given matrix to strings and returns a string matrix.

NAME	SYNTAX	DESCRIPTION
toString	<pre>String[] toString(boolean[] barray)</pre>	Converts all the Booleans in the given array to strings and returns a string array.
toString	<pre>String[][] toString(boolean[][] bmatrix)</pre>	Converts all the Booleans in the given matrix to strings and returns a string matrix.

Array Methods

Array methods are used to add, remove, insert, and extract subsets of 1D and 2D arrays.

NAME	SYNTAX	DESCRIPTION
getColumn	<pre>String[] getColumn(String[][] matrix, int column)</pre>	Returns a String[] for a specified column in the matrix. Useful when values have been read from a file and only certain columns should be shown in a table.
getColumn	<pre>double[] getColumn(double[][] matrix, int column)</pre>	Returns a double[] for a specified column in the matrix.
getColumn	<pre>int[] getColumn(int[][] matrix, int column)</pre>	Returns an int[] for a specified column in the matrix.
getColumn	<pre>boolean[] getColumn(boolean[][] matrix, int column)</pre>	Returns a boolean[] for a specified column in the matrix.
getSubMatrix	<pre>String[][] getSubMatrix(String[][] matrix, int startCol, int endCol, int startRow, int endRow)</pre>	Returns a rectangular submatrix of the input matrix spanning columns from startCol to endCol, and rows from startRow to endRow.
getSubMatrix	<pre>double[][] getSubMatrix(double[][] matrix, int startCol,int endCol, int startRow, int endRow)</pre>	Returns a rectangular submatrix of the input matrix spanning columns from startCol to endCol, and rows from startRow to endRow.

NAME	SYNTAX	DESCRIPTION
getSubMatrix	<pre>int[][] getSubMatrix(int[][] matrix, int startCol, int endCol, int startRow, int endRow)</pre>	Returns a rectangular submatrix of the input matrix spanning columns from startCol to endCol, and rows from startRow to endRow.
getSubMatrix	<pre>boolean[][] getSubMatrix(boolean[][] matrix, int startCol, int endCol, int startRow, int endRow)</pre>	Returns a rectangular submatrix of the input matrix spanning columns from startCol to endCol, and rows from startRow to endRow.
insert	<pre>String[] insert(String[] array, String value, int index)</pre>	Inserts an element at position index in an array and returns the expanded array.
insert	<pre>double[] insert(double[] array, double value, int index)</pre>	Inserts an element at position index in an array and returns the expanded array.
insert	<pre>int[] insert(int[] array, int value, int index)</pre>	Inserts an element at position index in an array and returns the expanded array.
insert	<pre>boolean[] insert(boolean[] array, boolean value, int index)</pre>	Inserts an element at position index in an array and returns the expanded array.
insert	<pre>String[] insert(String[] array, String[] value, int[] index)</pre>	Inserts elements in an array at positions given by the index array and returns the expanded array.
insert	<pre>double[] insert(double[] array, double[] value, int[] index)</pre>	Inserts elements in an array at positions given by the index array and returns the expanded array.
insert	<pre>int[] insert(int[] array, int[] value, int[] index)</pre>	Inserts elements in an array at positions given by the index array and returns the expanded array.
insert	<pre>boolean[] insert(boolean[] array, boolean[] value, int[] index)</pre>	Inserts elements in an array at positions given by the index array and returns the expanded array.
append	<pre>String[] append(String[] array, String value)</pre>	Adds an element to the end of an array and returns the expanded array.
append	<pre>double[] append(double[] array, double value)</pre>	Adds an element to the end of an array and returns the expanded array.

NAME	SYNTAX	DESCRIPTION
append	<pre>int[] append(int[] array, int value)</pre>	Adds an element to the end of an array and returns the expanded array.
append	<pre>boolean[] append(boolean[] array, boolean value)</pre>	Adds an element to the end of an array and returns the expanded array.
append	<pre>String[] append(String[] array, String[] value)</pre>	Adds elements to the end of an array and returns the expanded array.
append	<pre>double[] append(double[] array, double[] value)</pre>	Adds elements to the end of an array and returns the expanded array.
append	<pre>int[] append(int[] array, int[] value)</pre>	Adds elements to the end of an array and returns the expanded array.
append	<pre>boolean[] append(boolean[] array, boolean[] value)</pre>	Adds elements to the end of an array and returns the expanded array.
remove	<pre>String[] remove(String[] array, int index)</pre>	Removes an element from an array and returns the shortened array.
remove	<pre>double[] remove(double[] array, int index)</pre>	Removes an element from an array and returns the shortened array.
remove	<pre>int[] remove(int[] array, int index)</pre>	Removes an element from an array and returns the shortened array.
remove	<pre>boolean[] remove(boolean[] array, int index)</pre>	Removes an element from an array and returns the shortened array.
remove	<pre>String[] remove(String[] array, int[] index)</pre>	Removes elements from an array and returns the shortened array.
remove	<pre>double[] remove(double[] array, int[] index)</pre>	Removes elements from an array and returns the shortened array.
remove	<pre>int[] remove(int[] array, int[] index)</pre>	Removes elements from an array and returns the shortened array.
remove	<pre>boolean[] remove(boolean[] array, int[] index)</pre>	Removes elements from an array and returns the shortened array.

	SYNTAX	
insertRow	<pre>String[][] insertRow(String[][] matrix, String[] value, int rowIndex)</pre>	Inserts a row into a rectangular 2D array and returns the
		expanded array.
insertRow	<pre>double[][] insertRow(double[][] matrix, double[] value, int rowIndex)</pre>	Inserts a row into a rectangular 2D array and returns the expanded array.
insertRow	<pre>int[][] insertRow(int[][] matrix, int[] value, int rowIndex)</pre>	Inserts a row into a rectangular 2D array and returns the expanded array.
insertRow	<pre>boolean[][] insertRow(boolean[][] matrix, boolean[] value, int rowIndex)</pre>	Inserts a row into a rectangular 2D array and returns the expanded array.
insertRow	<pre>String[][] insertRow(String[][] matrix, String[][] value, int[] rowIndex)</pre>	Adds rows to a rectangular 2D array and returns the expanded array.
insertRow	<pre>double[][] insertRow(double[][] matrix, double[][] value, int[] rowIndex)</pre>	Adds rows to a rectangular 2D array and returns the expanded array.
insertRow	<pre>int[][] insertRow(int[][] matrix, int[][] value, int[] rowIndex)</pre>	Adds rows to a rectangular 2D array and returns the expanded array.
insertRow	<pre>boolean[][] insertRow(boolean[][] matrix, boolean[][] value, int[] rowIndex)</pre>	Adds rows to a rectangular 2D array and returns the expanded array.
replaceRow	<pre>String[][] replaceRow(String[][] matrix, String[] value, int rowIndex)</pre>	Replaces a row in a rectangular 2D array and returns the array.
replaceRow	<pre>double[][] replaceRow(double[][] matrix, double[] value, int rowIndex)</pre>	Replaces a row in a rectangular 2D array and returns the array.
replaceRow	<pre>int[][] replaceRow(int[][] matrix, int[] value, int rowIndex)</pre>	Replaces a row in a rectangular 2D array and returns the array.
replaceRow	<pre>boolean[][] replaceRow(boolean[][] matrix, boolean[] value, int rowIndex)</pre>	Replaces a row in a rectangular 2D array and returns the array.
replaceRow	<pre>String[][] replaceRow(String[][] matrix, String[][] value, int[] rowIndex)</pre>	Replaces rows in a rectangular 2D array and returns the array.
replaceRow	<pre>double[][] replaceRow(double[][] matrix, double[][] value, int[] rowIndex)</pre>	Replaces rows in a rectangular 2D array and returns the array.
replaceRow	<pre>int[][] replaceRow(int[][] matrix, int[][] value, int[] rowIndex)</pre>	Replaces rows in a rectangular 2D array and returns the array.

NAME	SYNTAX	DESCRIPTION
replaceRow	<pre>boolean[][] replaceRow(boolean[][] matrix, boolean[][] value, int[] rowIndex)</pre>	Replaces rows in a rectangular 2D array and returns the array.
appendRow	<pre>String[][] appendRow(String[][] matrix, String[] value)</pre>	Adds a row to the end of a rectangular 2D array and returns the expanded array.
appendRow	<pre>double[][] appendRow(double[][] matrix, double[] value)</pre>	Adds a row to the end of a rectangular 2D array and returns the expanded array.
appendRow	<pre>int[][] appendRow(int[][] matrix, int[] value)</pre>	Adds a row to the end of a rectangular 2D array and returns the expanded array.
appendRow	<pre>boolean[][] appendRow(boolean[][] matrix, boolean[] value)</pre>	Adds a row to the end of a rectangular 2D array and returns the expanded array.
appendRow	<pre>String[][] appendRow(String[][] matrix, String[][] value)</pre>	Adds rows to the end of a rectangular 2D array and returns the expanded array.
appendRow	<pre>double[][] appendRow(double[][] matrix, double[][] value)</pre>	Adds rows to the end of a rectangular 2D array and returns the expanded array.
appendRow	<pre>int[][] appendRow(int[][] matrix, int[][] value)</pre>	Adds rows to the end of a rectangular 2D array and returns the expanded array.
appendRow	<pre>boolean[][] appendRow(boolean[][] matrix, boolean[][] value)</pre>	Adds rows to the end of a rectangular 2D array and returns the expanded array.
removeRow	<pre>String[][] removeRow(String[][] matrix, int rowIndex)</pre>	Removes a row from a 2D array and returns the smaller array.
removeRow	<pre>double[][] removeRow(double[][] matrix, int rowIndex)</pre>	Removes a row from a 2D array and returns the smaller array.
removeRow	<pre>int[][] removeRow(int[][] matrix, int rowIndex)</pre>	Removes a row from a 2D array and returns the smaller array.
removeRow	<pre>boolean[][] removeRow(boolean[][] matrix, int rowIndex)</pre>	Removes a row from a 2D array and returns the smaller array.
removeRow	<pre>String[][] removeRow(String[][] matrix, int[] rowIndex)</pre>	Removes rows from a 2D array and returns the reduced array.
removeRow	<pre>double[][] removeRow(double[][] matrix, int[] rowIndex)</pre>	Removes rows from a 2D array and returns the reduced array.

NAME	SYNTAX	DESCRIPTION
removeRow	<pre>int[][] removeRow(int[][] matrix, int[] rowIndex)</pre>	Removes rows from a 2D array and returns the reduced array.
removeRow	<pre>boolean[][] removeRow(boolean[][] matrix, int[] rowIndex)</pre>	Removes rows from a 2D array and returns the reduced array.
insertColumn	<pre>String[][] insertColumn(String[][] matrix, String[] value, int columnIndex)</pre>	Adds a column into a rectangular 2D array and returns the expanded array.
insertColumn	<pre>double[][] insertColumn(double[][] matrix, double[] value, int columnIndex)</pre>	Adds a column into a rectangular 2D array and returns the expanded array.
insertColumn	<pre>int[][] insertColumn(int[][] matrix, int[] value, int columnIndex)</pre>	Adds a column into a rectangular 2D array and returns the expanded array.
insertColumn	<pre>boolean[][] insertColumn(boolean[][] matrix, boolean[] value, int columnIndex)</pre>	Adds a column into a rectangular 2D array and returns the expanded array.
insertColumn	<pre>String[][] insertColumn(String[][] matrix, String[][] value, int[] columnIndex)</pre>	Adds columns to a rectangular 2D array and returns the expanded array.
insertColumn	<pre>double[][] insertColumn(double[][] matrix, double[][] value, int[] columnIndex)</pre>	Adds columns to a rectangular 2D array and returns the expanded array.
insertColumn	<pre>int[][] insertColumn(int[][] matrix, int[][] value, int[] columnIndex)</pre>	Adds columns to a rectangular 2D array and returns the expanded array.
insertColumn	<pre>boolean[][] insertColumn(boolean[][] matrix, boolean[][] value, int[] columnIndex)</pre>	Adds columns to a rectangular 2D array and returns the expanded array.
replaceColumn	<pre>String[][] replaceColumn(String[][] matrix, String[] value, int columnIndex)</pre>	Replaces a column in a rectangular 2D array and returns the array.
replaceColumn	<pre>double[][] replaceColumn(double[][] matrix, double[] value, int columnIndex)</pre>	Replaces a column in a rectangular 2D array and returns the array.
replaceColumn	<pre>int[][] replaceColumn(int[][] matrix, int[] value, int columnIndex)</pre>	Replaces a column in a rectangular 2D array and returns the array.

NAME	SYNTAX	DESCRIPTION
replaceColumn	<pre>boolean[][] replaceColumn(boolean[][] matrix, boolean[] value, int columnIndex)</pre>	Replaces a column in a rectangular 2D array and returns the array.
replaceColumn	<pre>String[][] replaceColumn(String[][] matrix, String[][] value, int[] columnIndex)</pre>	Replaces columns in a rectangular 2D array and returns the array.
replaceColumn	<pre>double[][] replaceColumn(double[][] matrix, double[][] value, int[] columnIndex)</pre>	Replaces columns in a rectangular 2D array and returns the array.
replaceColumn	<pre>int[][] replaceColumn(int[][] matrix, int[][] value, int[] columnIndex)</pre>	Replaces columns in a rectangular 2D array and returns the array.
replaceColumn	<pre>boolean[][] replaceColumn(boolean[][] matrix, boolean[][] value, int[] columnIndex)</pre>	Replaces columns in a rectangular 2D array and returns the array.
appendColumn	<pre>String[][] appendColumn(String[][] matrix, String[] value)</pre>	Adds a column at the end of a rectangular 2D array and returns the expanded array.
appendColumn	<pre>double[][] appendColumn(double[][] matrix, double[] value)</pre>	Adds a column at the end of a rectangular 2D array and returns the expanded array.
appendColumn	<pre>int[][] appendColumn(int[][] matrix, int[] value)</pre>	Adds a column at the end of a rectangular 2D array and returns the expanded array.
appendColumn	<pre>boolean[][] appendColumn(boolean[][] matrix, boolean[] value)</pre>	Adds a column at the end of a rectangular 2D array and returns the expanded array.
appendColumn	<pre>String[][] appendColumn(String[][] matrix, String[][] value)</pre>	Adds columns to the end of a rectangular 2D array and returns the expanded array.
appendColumn	<pre>double[][] appendColumn(double[][] matrix, double[][] value)</pre>	Adds columns to the end of a rectangular 2D array and returns the expanded array.
appendColumn	<pre>int[][] appendColumn(int[][] matrix, int[][] value)</pre>	Adds columns to the end of a rectangular 2D array and returns the expanded array.
appendColumn	<pre>boolean[][] appendColumn(boolean[][] matrix, boolean[][] value)</pre>	Adds columns to the end of a rectangular 2D array and returns the expanded array.
removeColumn	<pre>String[][] removeColumn(String[][] matrix, int columnIndex)</pre>	Removes a column from a rectangular 2D array and returns the smaller array.

NAME	SYNTAX	DESCRIPTION
removeColumn	<pre>double[][] removeColumn(double[][] matrix, int columnIndex)</pre>	Removes a column from a rectangular 2D array and returns the smaller array.
removeColumn	<pre>int[][] removeColumn(int[][] matrix, int columnIndex)</pre>	Removes a column from a rectangular 2D array and returns the smaller array.
removeColumn	<pre>boolean[][] removeColumn(boolean[][] matrix, int columnIndex)</pre>	Removes a column from a rectangular 2D array and returns the smaller array.
removeColumn	<pre>String[][] removeColumn(String[][] matrix, int[] columnIndex)</pre>	Removes columns from a rectangular 2D array and returns the reduced array.
removeColumn	<pre>double[][] removeColumn(double[][] matrix, int[] columnIndex)</pre>	Removes columns from a rectangular 2D array and returns the reduced array.
removeColumn	<pre>int[][] removeColumn(int[][] matrix, int[] columnIndex)</pre>	Removes columns from a rectangular 2D array and returns the reduced array.
removeColumn	<pre>boolean[][] removeColumn(boolean[][] matrix, int[] columnIndex)</pre>	Removes columns from a rectangular 2D array and returns the reduced array.
matrixSize	<pre>int[] matrixSize(String[][] matrix)</pre>	Retums the number of rows and columns of a matrix as an integer array of length 2.
matrixSize	<pre>int[] matrixSize(double[][] matrix)</pre>	Retums the number of rows and columns of a matrix as an integer array of length 2.
matrixSize	<pre>int[] matrixSize(int[][] matrix)</pre>	Retums the number of rows and columns of a matrix as an integer array of length 2.
matrixSize	<pre>int[] matrixSize(boolean[][] matrix)</pre>	Returns the number of rows and columns of a matrix as an integer array of length 2.
transpose	<pre>String[][] transpose(String[][] matrix)</pre>	Returns the transpose of a matrix.
transpose	<pre>double[][] transpose(double[][] matrix)</pre>	Returns the transpose of a matrix.

NAME	SYNTAX	DESCRIPTION
transpose	<pre>int[][] transpose(int[][] matrix)</pre>	Returns the transpose of a matrix.
transpose	boolean[][] transpose(boolean[][] matrix)	Retums the transpose of a matrix.

String Methods

String methods are used to process string variables and string arrays.

NAME	SYNTAX	DESCRIPTION
concat	String concat(String separator, String strs)	Concatenates the given varargs- array of strings into a single string using the given separator.
concat	<pre>String[] concat(String colSepar, String rowSepar, String[] matr)</pre>	Concatenates the given string matrix (which can be given as a varargs of rows) into a single string. Puts colSepar between values of columns of a row, and rowSepar between rows.
contains	<pre>boolean contains(String[] strs, String str)</pre>	Returns true if the given string array strs contains the given string str .
find	<pre>int[] find(String[] strs, String str)</pre>	Returns an array with the indices to all occurrences of str in strs.
findIn	<pre>int findIn(String[] strs, String str)</pre>	Returns the index to the first occurrence of str in strs or -1 if no match.
findIn	int findIn(String str, String toFind)	Returns the first index of str that is the start of the substring toFind. If there is no substring matching toFind in str, -1 is returned.
length	<pre>int length(String str)</pre>	Returns the length of the string str .
replace	String replace(String str, String orig, String replacement)	Returns a string where orig has been replaced by replacement .
split	<pre>String[] split(String str)</pre>	Returns an array of strings by splitting the given string at spaces.

NAME	SYNTAX	DESCRIPTION
split	<pre>String[] split(String str, String separator)</pre>	Returns an array of strings by splitting the given string at the given separator.
substring	String substring(String str, int start, int length)	Returns a substring with the given length starting at the given position.
unique	<pre>String[] unique(String[] strs)</pre>	Returns an array of strings with the unique values in the given array of strings.

Collection Methods

Collection methods are used to copy, compare, sort, and merge variables and arrays.

NAME	SYNTAX	DESCRIPTION
сору	<pre>String[] copy(String toCopy)</pre>	Returns a copy of the given array of strings, which can also be specified as a varargs of strings.
сору	<pre>String[][] copy(String[] toCopy)</pre>	Returns a copy of the given string matrix, which can also be specified as a varargs of rows (string arrays).
сору	<pre>double[] copy(double toCopy)</pre>	Returns a copy of the given array of doubles, which can also be specified as a varargs of doubles.
сору	<pre>double[][] copy(double[] toCopy)</pre>	Returns a copy of the given double matrix, which can also be specified as a varargs of rows (double arrays).
сору	<pre>int[] copy(int toCopy)</pre>	Returns a copy of the given array of integers, which can also be specified as a varargs of integers.
сору	<pre>int[][] copy(int[] toCopy)</pre>	Returns a copy of the given integer matrix, which can also be specified as a varargs of rows (integer arrays).

NAME	SYNTAX	DESCRIPTION
сору	<pre>boolean[] copy(boolean toCopy)</pre>	Returns a copy of the given array of booleans, which can also be specified as a varargs of booleans.
сору	boolean[][] copy(boolean[] toCopy)	Returns a copy of the given boolean matrix, which can also be specified as a varargs of rows (boolean arrays).
equals	<pre>boolean equals(String[] str1, String[] str2)</pre>	Returns true if all strings in the given array are equal and they have the same number of elements.
equals	<pre>boolean equals(String[][] matr1, String[][] matr2)</pre>	Returns true if all strings in the given matrix are equal and they have the same number of elements.
equals	<pre>boolean equals(int[] ints1, int[] ints2)</pre>	Returns true if all integers in the given array are equal and they have the same number of elements.
equals	boolean equals(int[][] ints1, int[][] ints2)	Returns true if all integers in the given matrix are equal and they have the same number of elements.
equals	boolean equals(double dl1, double dl2, double relErrorTolerance)	Compares whether the relative error of two doubles is within allowed tolerance using abs((a - b) / b), where b is the larger of the doubles (by absolute value).
equals	boolean equals(double dl1, double dl2)	Same as above, but uses a default relErrorTolerance of 0.0001.
equals	<pre>boolean equals(double[] dbls1, double[] dbls2, double relErrorTolerance)</pre>	Compares the relative errors (~ abs((a - b) / b) of elements in the arrays pairwise and returns true if all relative errors are below relErrorTolerance and the arrays have the same number of elements.
equals	<pre>boolean equals(double[] dbls1, double[] dbls2)</pre>	Same as above, but uses a default relErrorTolerance of 0.0001.

NAME	SYNTAX	DESCRIPTION
equals	<pre>boolean equals(double[][] dbls1, double[][] dbls2, double relErrorTolerance)</pre>	Compares the relative errors (\sim abs((a - b) / b) of elements in the matrices pairwise and returns true if all relative errors are below relErrorTolerance and the matrices have the same number of elements.
equals	<pre>boolean equals(double[][] dbls1, double[][] dbls2)</pre>	Same as above, but uses a default relErrorTolerance of 0.0001.
sort	<pre>sort(String[] strs)</pre>	Sorts the given array of strings. NOTE: The array is sorted in place.
sort	<pre>sort(int[] ints)</pre>	Sorts the given array of integers. NOTE: The array is sorted in place.
sort	<pre>sort(double[] doubles)</pre>	Sorts the given array of doubles. NOTE: The array is sorted in place.
sort	<pre>sort(String[][] strs)</pre>	Sorts the given 2D array of strings. The columns are sorted by their row values from top to bottom. NOTE: The array is sorted in place.
sort	<pre>sort(int[][] ints)</pre>	Sorts the given 2D array of integers. The columns are sorted by their row values from top to bottom. NOTE: The array is sorted in place.
sort	<pre>sort(double[][] doubles)</pre>	Sorts the given 2D array of doubles. The columns are sorted by their row values from top to bottom. NOTE: The array is sorted in place.
merge	<pre>merge(String[] toMerge)</pre>	Returns an array of strings with all strings merged from the given arrays.

NAME	SYNTAX	DESCRIPTION
merge	<pre>merge(int[] toMerge)</pre>	Returns an array of integers with all integers merged from the two given arrays.
merge	<pre>merge(double[] toMerge)</pre>	Returns an array of doubles with all doubles merged from the two given arrays.

Model Builder Methods for Use in Add-ins

For writing add-in method code that operates on the current component, current mesh, current physics, and so on, use the methods in the table below.

NAME	SYNTAX	DESCRIPTION
getCurrentComponent	getCurrentComponent()	Returns an object of the type ModelNode for the current component.
getCurrentMesh	getCurrentMesh()	Returns an object of the type MeshSequence for the current mesh.
getCurrentNode	getCurrentNode()	Returns an object of the type ModelEntity for the current component.
getCurrentPhysics	<pre>getCurrentPhysics()</pre>	Returns an object of the type Physics for the current physics interface.
getCurrentPlotGroup	getCurrentPlotGroup()	Returns an object of the type ResultFeature for the current component.
getCurrentStudy	getCurrentStudy()	Returns an object of the type Study for the current component.
selectNode	selectNode(ModelEntity entity)	Selects a model tree node and displays its Settings window after the execution of a method from the Model Builder.

These methods return the corresponding entity such that the method code in an add-in can operate on it. When called from an application a method in this

category returns null. Also, null is returned if no entity of the corresponding type exists such that nothing is current.

NAME	SYNTAX	DESCRIPTION
createAddinForm	FormFeature createAddinForm(String tag, String definition)	Creates a Settings Form node with the given tag based on the Form Definition node with the given name definition. The Form Definition must be configured to show as Settings form.
createAddinMethodCall	MethodCallFeature createAddinMethodCall(String tag, String definition)	Creates a Method Call node with the given tag based on the Method Definition node with the given name definition. The Method Definition must be configured to use as Method call.

General utility methods for writing methods for add-ins are listed in the table below.

To learn more about using these methods you can review the Application Builder settings for one of the built-in add-ins by opening the corresponding MPH file. In a typical Windows[®] installation the built-in add-in library is located at

C:\Program Files\COMSOL\COMSOL63\COMSOL_Multiphysics\addins

This section contains examples that illustrate solving practical tasks by accessing and manipulating the model object and using the built-in methods. Note that additional examples of user-defined methods are provided in the Application Libraries, accessible from the **File** menu.

Running the Examples

To run the code in the examples below, you can create a method for use in a model or application. You can, for example, choose the option **Blank Model** in the Model Wizard. To create a new method, go to the **Developer** tab in the Model Builder and click the **New Method** button in the ribbon. In the Application Builder, paste the code into the new method. Finally, you can run the code from the **Developer** tab in the Model Builder by choosing the method you just created from the **Run Method** toolbar menu. Alternatively, for running a single block of code, you can use the **Java Shell** window, with the benefit of not having to switch between the Model Builder and the Application Builder.

Visualization Without Solution Data: Grid Datasets

The section "Results" on page 63 shows how to write code for various parts of the Results node in the model tree, including **Datasets**, **Tables**, and **Plot Groups**. These examples assume that you have solution data available from solving, for example, a heat transfer, CFD, or structural mechanics problem.

You can also create visualizations without having associated solution data by either using grid datasets or using low-level functionality only available through methods. You can, for example, write code for plotting points and triangles without any associated solution data. These techniques are useful when creating applications where customized plot functionality is needed. This section shows how to use grid datasets, and the next section shows how to use low-level functionality.

PLOTTING A UNIT SPHERE USING A GRID DATASET

Grid datasets are available in the Model Builder and can be used in applications for the sole purpose of visualization without any associated solution data. The code below creates a visualization of a unit sphere as an isosurface with the *z*-coordinate as color data.

```
model.func().create("an1", "Analytic");
model.result().dataset().create("grid1", "Grid3D");
with(model.result().dataset("grid1"));
  set("source", "data");
set("parmin1", -1);
set("parmax1", +1);
  set("parmin2", -1);
  set("parmax2", +1);
  set("parmin3", -1);
  set("parmax3", +1);
  set("source", "function");
  set("function", "an1");
endwith();
model.result().create("pg1", "PlotGroup3D");
model.result("pg1").create("iso1", "Isosurface");
with(model.result("pq1").feature("iso1"));
  set("expr", "x^2+y^2+z^2-1");
  set("levelmethod", "levels");
  set("levels", 0.0);
endwith();
model.result("pg1").feature("iso1").create("col1", "Color");
with(model.result("pg1").feature("iso1").feature("col1"));
  set("expr", "z");
endwith();
model.result("pq1").run();
```

Comments

If there is no solution-based **Dataset** available, then the **Grid3D** dataset needs to have a **Function** as its **Source**. In the example above, a default Analytic function is created with tag an1. A default **Analytic** function corresponds to f(x) = x, and its only purpose is to give the grid dataset an evaluation context.

Note: The alternative is to solve a physics problem on a mesh and reference the corresponding solution dataset. The method of referencing an arbitrary **Function** makes it possible to create visualizations without solution data.

The source for the **Grid3D** dataset with tag grid1 is set to function, and finally, the function property of grid1 is set to an1.

The **Grid3D** dataset has options for max and min parameter bounds, shown in the example code above. An additional grid resolution option is not shown in this example. However, you can learn about its syntax by using **Record Code** from the Model Builder.

The following examples describe low-level functionality for visualization that is only available from methods and is not associated with any solution data. For visualization based on solution data, see the section "Results" on page 63.

The examples below illustrate using the following plot types:

- Point Data
- Line Data
- Surface Data
- Tube Data
- Arrow Data
- Annotation Data

Once created, the plot type names are visible in the **Settings** window of each plot. In addition to the properties modified by the examples below, in the **Settings** window of these plot types, you can see the number of geometric entities created, such as number of points, line segments, and triangles. Just as for other types of plots, you can also change the **Range** of color and data, as well as **Coloring and Style**.

POINTS IN 2D

The following code plots a circle of points using the Point Data plot type.

```
// A circle of points
String pgTag = model.result().uniquetag("pg");
ResultFeature pg = model.result().create(pgTag, 2);
ResultFeature plot = pg.create("pt1", "PointData");
int N = 17;
double[][] p = new double[2][N];
double[] color = new double[N];
double R = 1000;
for (int i = 0; i < N; i++) {
  double angle = i*2*Math.PI/N;
  p[0][i] = R*Math.cos(angle);
  p[1][i] = R*Math.sin(angle);
  color[i] = p[1][i];
}
plot.set("pointdata", p)
  .set("colordata", color)
.set("coloring", "colortable");
plot.run();
```

Comments

The first line

String pgTag = model.result().uniquetag("pg");
creates a unique tag for the plot group to be created. This is useful if you intend to add a varying number of plot groups in your model or application.

The line

ResultFeature pg = model.result().create(pgTag, 2);

creates a **2D Plot Group** using the newly created unique tag. The second argument to create defines the dimension of the plot group (2 for 2D, 3 for 3D, and so on).

The line

```
ResultFeature plot = pg.create("pt1", "PointData");
```

creates a plot of the type PointData. This plot type is only available through methods.

The middle part of the example code generates the points making up the circle. The point coordinates are stored in the 2-by-N array p, along with color data in the array color of length N. The color data is, in this example, simply based on the index of the points and is used to control the coloring of each point based on a color table.

The last few lines populate the fields of the **Point Data** plot.

```
plot.set("pointdata", p)
   .set("colordata", color)
   .set("coloring", "colortable");
```

The property pointdata takes the 2-by-N array p as its input. The options for the coloring property are colortable or uniform.

To learn about the syntax for the additional properties available for a **Point Data** plot, you can run the above code in a blank model, browse to the **Settings** window for the **Point Data** plot, click **Record Code**, and change the corresponding plot properties. Note that the name of the plot type in the **Settings** window of the plot in the model tree is **Point Data**.

TURNING OFF MODEL HISTORY

When using this type of low-level functionality for larger sets of data, such as a large number of points, the stored model history may become excessively large. Because of this, it is recommended to temporarily turn off model history recording when using this type of functionality; see "Turning Off and Resetting The Model History" on page 71.

POINTS IN 3D

The following code plots points in an undulating pattern in 3D using the **Point Data** plot type.

// Undulating points in 3D

```
String pgTag = model.result().uniquetag("pg");
ResultFeature pg = model.result().create(pgTag, 3);
ResultFeature plot = pg.create("pt1", "PointData");
int N = 37;
double[][] p = new double[3][N];
double[] color = new double[N];
double R = 1000;
for (int i = 0; i < N; i++) {
  double angle = i*2*Math.PI/N;
  p[0][i] = R*Math.cos(angle);
  p[1][i] = R*Math.sin(angle);
  p[2][i] = R*Math.cos(3*angle);
  color[i] = p[1][i];
}
plot.set("pointdata", p)
  .set("colordata", color)
  .set("coloring", "colortable")
  .set("sphereradiusscale", 1);
plot.run();
selectNode(pg);
```

When plotting 3D points the line

```
model.result().create(pgTag, 3);
```

has the second argument set to 3 in order to create a **3D Plot Group**. In 3D, the point coordinates, p is a 3-by-N array.

The line

.set("sphereradiusscale", 1);

controls the radius of the sphere used to render each point.

To automatically display the newly created plot, the line

selectNode(pg);

is added last in the code segment.

To get a denser set of points, you can increase the integer N to, say, 370.

CURVE IN 3D

The following code plots line segments in the shape of a 3D helix using the Line **Data** plot type.

// A 3D helix from line segments

```
String pgTag = model.result().uniquetag("pg");
ResultFeature pg = model.result().create(pgTag, 3);
ResultFeature plot = pg.create("line1", "LineData");
int N = 100;
double[][] p = new double[3][N];
int[][] t = new int[2][N-1];
for (int i = 0; i < N; i++) {
    double s = 4*Math.PI*i/N;
```

```
p[0][i] = s/5;
p[1][i] = Math.sin(s);
p[2][i] = Math.cos(s);
if (i > 0) {
   t[0][i-1] = i-1;
   t[1][i-1] = i;
}
}
plot.set("pointdata", p)
   .set("elementdata", t);
plot.run();
selectNode(pg);
```

The line

```
ResultFeature plot = pg.create("line1", "LineData");
```

creates a plot of the type LineData. This plot type is only available through methods. Just as for **Point Data** plots, the point coordinates p is a 3-by-N array. In addition to pointdata, the LineData plot type takes elementdata as its input. In the example, this is represented by the 2-by-N array t and contains indexes to the columns of p, corresponding to the start and end points of the lines.

In a similar way, line segments can be plotted in 2D by creating a 2D plot group and by letting the point coordinates be a 2-by-N array. See also "Points in 2D" on page 180.

TRIANGULATED SHAPE IN 2D

The following code plots triangles in the shape of a 2D pentagon by using the **Surface Data** plot type.

```
// A 2D pentagon from triangles
```

```
String pgTag = model.result().uniquetag("pg");
ResultFeature pg = model.result().create(pgTag, 2);
ResultFeature plot = pg.create("surf1", "SurfaceData");
int N = 5;
double[][] p = new double[2][N+1];
int[][] t = new int[3][N];
p[0][0] = 0;
p[1][0] = 0;
for (int i = 0; i < N; i++) {
  double angle = i*2*Math.PI/N;
  p[0][i+1] = Math.cos(angle);
 p[1][i+1] = Math.sin(angle);
 t[0][i] = 0;
  t[1][i] = i+1;
 t[2][i] = 1+(i+1)%N;
}
plot.set("pointdata", p)
  .set("elementdata", t);
plot.run();
```

```
selectNode(pg);
```

The line

```
ResultFeature plot = pg.create("surf1", "SurfaceData");
```

creates a plot of the type SurfaceData. This plot type is only available through methods. Just as for 2D **Point Data** plots, the point coordinates p is a 2-by-N array. In addition to pointdata, and similar to the LineData plot type, the SurfData plot type takes elementdata as its input. In the example, this is represented by the 3-by-N array t and contains indexes to the columns of p, corresponding to the vertexes of the triangles. The ordering of the point indexes in the array t is not important for 2D Surface Data plots.

FUNCTION SURFACE IN 3D

The following code plots triangles in the shape of a 3D rotationally symmetric sinc-function surface by using the **Surface Data** plot type.

// A 3D sinc(r) function surface

```
String pgTag = model.result().uniquetag("pg");
ResultFeature pg = model.result().create(pgTag, 3);
ResultFeature plot = pg.create("surf1", "SurfaceData");
int Nx = 51;
int Ny = 51;
double[][] p = new double[3][Nx*Ny];
int[][] t = new int[3][2*(Nx-1)*(Ny-1)];
double[] color = new double[Nx*Ny];
int pos = 0;
for (int i = 0; i < Ny; i++) {</pre>
  for (int j = 0; j < Nx; j++) {
    double x = 20*(j-Nx/2)/Nx;
    double y = 20*(i-Ny/2)/Ny;
    double r = Math.sqrt(x*x+y*y);
    double z = 4*((r == 0) ? 1 : (Math.sin(r)/r));
    p[0][pos] = x;
    p[1][pos] = y;
    p[2][pos] = z;
    color[pos] = z;
    pos++;
  }
}
pos = 0;
for (int i = 0; i < Ny-1; i++) {</pre>
  for (int j = 0; j < Nx-1; j++) {
    int p00 = Nx*i+j;
    int p01 = Nx*i+j+1;
    int p10 = Nx*(i+1)+j;
    int p11 = Nx^{(i+1)+j+1};
    t[0][pos] = p00;
    t[1][pos] = p01;
    t[2][pos] = p11;
```

```
pos++;
t[0][pos] = p00;
t[1][pos] = p11;
t[2][pos] = p10;
pos++;
}
plot.set("pointdata", p)
.set("elementdata", t)
.set("colordata", color)
.set("coloring", "colortable");
plot.run();
selectNode(pg);
```

This example is similar to "Triangulated Shape in 2D" on page 183, but with the point array being a 3-by-N array for 3D surfaces. For **Surface Data** plots in 3D, the ordering of the indexes in the elementdata array t matters. It determines the direction of the surface normal, which is used for the lighting effect when using **Scene Light** in the **Graphics** window. The surface normal of a triangle is determined according to the "right-hand rule". In mathematical terms, the surface normal is defined as the vector product:

 $\mathbf{n} = (\mathbf{p}[t[1]] - \mathbf{p}[t[0]]) \times (\mathbf{p}[t[2]] - \mathbf{p}[t[0]])$

where the indexes into t represent the rows in one of the columns of t and \mathbf{p} represents a column in the array of points p.

To ensure that the lighting effect produces expected results, the triangle surface normal directions need to consistently point in the same direction as the intended overall surface normal direction. As an alternative to making sure that the indexes come in the correct order, the normal direction may be given as an additional input to a **Surface Data** plot. This is shown in the next example section, Sphere in 3D.

SPHERE IN 3D

The following code plots triangles in the shape of a 3D sphere by using the **Surface Data** plot type.

```
// A coarse sphere with user-supplied normals
int Nx = 20;
int Ny = 10;
String pgTag = model.result().uniquetag("pg");
ResultFeature pg = model.result().create(pgTag, 3);
ResultFeature plot = pg.create("surf1", "SurfaceData");
double[][] p = new double[3][Nx*Ny];
double[][] normals = new double[3][Nx*Ny];
int[][] t = new int[3][2*(Nx-1)*(Ny-1)];
double[] color = new double[Nx*Ny];
```

```
int pos = 0;
double R = 10;
for (int i = 0; i < Ny; i++) {</pre>
  for (int j = 0; j < Nx; j++) {
    double theta = Math.PI*i/(Ny-1);
    double phi = 2*Math.PI*j/(Nx-1);
    double x = R*Math.sin(theta)*Math.cos(phi);
    double y = R*Math.sin(theta)*Math.sin(phi);
    double z = R*Math.cos(theta);
    p[0][pos] = x;
    p[1][pos] = y;
    p[2][pos] = z;
    normals[0][pos] = x;
    normals[1][pos] = y;
    normals[2][pos] = z;
    color[pos] = z;
    pos++;
  }
}
pos = 0;
for (int i = 0; i < Ny-1; i++) {</pre>
  for (int j = 0; j < Nx-1; j++) {
    int p00 = Nx*i+j;
    int p01 = Nx*i+j+1;
    int p10 = Nx^{(i+1)+j};
    int p11 = Nx*(i+1)+j+1;
    t[0][pos] = p00;
    t[1][pos] = p01;
    t[2][pos] = p11;
    pos++;
    t[0][pos] = p00;
    t[1][pos] = p11;
    t[2][pos] = p10;
   pos++;
  }
}
plot.set("pointdata", p)
  .set("elementdata", t)
  .set("colordata", color)
  .set("normaldata", normals)
  .set("coloring", "colortable");
plot.run();
selectNode(pg);
```

In this example, information about the surface normal direction is not given implicitly by the triangle orientation, but instead explicitly by the parameter normaldata by means of the 3-by-Nx*Ny array normals containing surface normal vectors at each point. The normal vectors do not need to be normalized; only the direction is used. The coloring of the sphere is based on the *z*-coordinate of each triangle point and is stored for each point in the 3-by-Nx*Ny array color.

The sphere is constructed from a discrete grid defined in terms of spherical coordinate angles, where each grid cell is divided into two triangles. The number of triangles t is then given by 2*(Nx-1)*(Ny-1).

TUBE PLOT IN 3D, LOGARITHMIC SPIRAL

The following code plots a tube in 3D in the shape of a logarithmic spiral by using the **Tube Data** plot type.

```
// A logarithmic tube spiral in 3D
String pgTag = model.result().uniquetag("pg");
ResultFeature pg = model.result().create(pgTag, 3);
ResultFeature plot = pg.create("tube1", "TubeData");
int N = 1000;
double[][] p = new double[3][N];
double[] radius = new double[N];
double[] color = new double[N];
for (int i = 0; i < N; i++) {
  double par = 0.005*i;
  p[0][i] = Math.exp(par)*Math.cos(10*par);
  p[1][i] = Math.exp(par)*Math.sin(10*par);
  p[2][i] = 0.1*i;
  radius[i] = 0.2*Math.sqrt(i+1);
  color[i] = i;
}
plot.set("pointdata", p)
  .set("radiusdata", radius)
  .set("colordata", color)
  .set("coloring", "colortable");
plot.run();
selectNode(pg);
```

Comments

A **Tube Data** plot is similar to a **Point Data**, plot but with an absolute radius array given as an argument to radiusdata. For the **Point Data** plot type, there is a similar sphereradiusscale.

ARROWS IN 2D

The following code plots arrows in a circular pattern by using the **Arrow Data** plot type.

```
// Arrows in a circular pattern in 2D
String pgTag = model.result().uniquetag("pg");
ResultFeature pg = model.result().create(pgTag, 2);
ResultFeature plot = pg.create("arrow1", "ArrowData");
int N = 17;
double[][] p = new double[2][N];
double[][] vec = new double[2][N];
double[][] vec = new double[2][N];
for (int i = 0, 2;
for (int i = 0; i < N; i++) {</pre>
```

```
double angle = 2*Math.PI*i/N;
p[0][i] = Math.cos(angle);
p[1][i] = Math.sin(angle);
vec[0][i] = -len*p[0][i];
vec[1][i] = -len*p[1][i];
}
plot.set("pointdata", p)
   .set("vectordata", vec);
plot.run();
selectNode(pg);
```

An **Arrow Data** plot associates an array of vectors, in the example vec, to each point p.

ARROWS IN 3D

The following code plots arrows in a logarithmic spiral pattern by using the **Arrow Data** plot type.

```
// Arrows in a logarithmic spiral pattern in 3D
String pgTag = model.result().uniquetag("pg");
ResultFeature pg = model.result().create(pgTag, 3);
ResultFeature plot = pg.create("arrow1", "ArrowData");
int N = 1000;
double[][] p = new double[3][N];
double[][] vec = new double[3][N];
double[] color = new double[N];
for (int i = 0; i < N; i++) {
  double par = 0.005*i;
  p[0][i] = Math.exp(par)*Math.cos(10*par);
  p[1][i] = Math.exp(par)*Math.sin(10*par);
  p[2][i] = 0.1*i;
  double len = Math.sqrt(p[0][i]*p[0][i]+p[1][i]*p[1][i]*p[2][i]*p[2][i]);
  for (int j = 0; j < 3; j++) {
    vec[j][i] = 4*p[j][i]/len;
  }
  color[i] = i;
}
plot.set("pointdata", p)
  .set("vectordata", vec)
  .set("colordata", color)
.set("coloring", "colortable");
plot.run();
selectNode(pg);
```

Comments

In this example, in addition to the example in the section "Arrows in 2D", color data is used based on the point index.

Annotations in 2D

The following code renders text strings in a circular pattern by using the **Annotation Data** plot type.

```
// Letters in a circular pattern in 2D
String pgTag = model.result().uniquetag("pg");
ResultFeature pg = model.result().create(pgTag, 2);
for (int i = 0; i < 26; i++) {
    double angle = 2*Math.PI*i/26;
    ResultFeature plot = pg.create("ann"+i, "AnnotationData");
    plot.set("pos", new double[]{Math.cos(angle), Math.sin(angle)})
    .set("text", "ABCDEFGHIJKLMNOPQRSTUVWXYZ".substring(i, i+1))
    .set("showpoint", false);
}
pg.run();
selectNode(pg);</pre>
```

Comments

The property pos takes as its input an array of length 2 representing 2D coordinates for the position of the string to be rendered. The property text takes as its input the string to be rendered. The Boolean property showpoint determines if a point, at the 2D coordinate position, should be rendered or not.

Annotations in 3D with LaTeX Syntax

The following code renders text strings with Greek letters of different colors at the corners of a cube by using the **Annotation Data** plot type.

```
// Greek letters at the corners of a cube
String pgTag = model.result().uniquetag("pg");
ResultFeature pg = model.result().create(pgTag, 3);
String[] texts = {"\\alpha", "\\beta", "\\gamma", "\\delta", "\\epsilon",
"\\zeta", "\\eta", "\\theta"};
String[] colors = {"black", "blue", "cyan", "gray", "green", "magenta", "red",
"yellow"};
for (int x = 0; x < 2; x++) {
  for (int y = 0; y < 2; y++) {
    for (int z = 0; z < 2; z++) {
      int index = x+2*y+4*z;
      ResultFeature plot = pg.create("ann"+index, "AnnotationData");
      plot.set("pos", new double[]{x, y, z})
        .set("text", "$"+texts[index]+"$")
        .set("latexmarkup", true)
        .set("color", colors[index]);
    }
 }
}
pg.run();
selectNode(pg);
```

The Boolean property latexmarkup determines if the text should be interpreted using LaTeX syntax or not.

Reading and Writing Data to File

The Application Builder provides several built-in methods for reading and writing different types of files: text files, CSV-files, Excel® files (requires LiveLink[™] *for* Excel[®]), and binary files. These file methods are listed in the table "File Methods" on page 123.

Note that easy-to-use user-interface-based techniques for reading and writing to file are available in the Form Editor of the Application Builder. It is recommended that you consider those techniques first before using the programming-based ways described in this section. For more information, see the book *Introduction to Application Builder* and "GUI Command Methods" on page 149. There, you can also find information on the various file schemes used in the Application Builder for reading and writing files when running applications in a web browser.

READING AND WRITING TEXT AND SPREADSHEET FILES OVERVIEW

The following built-in methods are available for reading and writing text files:

- readFile
- readMatrixFromFile
- readStringMatrixFromFile
- readCSVFile
- writeFile
- writeCSVFile

In addition, you can use the low-level methods available in the class CsReader to read text files line by line or character by character. See the next section, "Processing Text Files using the CsReader and CsWriter Classes" on page 205, for more information.

If you have a LiveLink[™] *for* Excel[®] license, then the following methods are available for reading and writing Microsoft Excel Workbook files:

- readExcelFile
- writeExcelFile

INTRODUCTION TO READING FILES WITH A CSV-FILE EXAMPLE

Assume that you want to automate a certain thermal analysis of a circuit board by creating an application that reads in data from a spreadsheet. Further assume that information about the circuit board components is given by a proprietary format in a spreadsheet with columns for component type, heat dissipation, locations, and sizes. Assume that such a file looks like:

```
B,0,0,0,-1.57,350,200,1.57
B,3,30,10,0,40,10,2
C,1,100,30,0,3,10,
C,1,110,30,0,3,10,
B,4,30,30,0,40,10,2
...
C,1,200,30,0,3,10,
B,10,100,150,0,10,20,30
B,10,160,150,0,10,20,30
```

Each row of the spreadsheet represents a different component. The first column can contain a letter, either B or C, denoting that the component can be modeled as either a **Block** or a **Cylinder** primitive. The next column is the total heat dissipation within the component (measured in watts). The next three columns represent the location of the component in the global Cartesian coordinate system (measured in millimeters). Lastly, if the row contains a block component, there are three more columns that denote the width, depth, and height of the block. If the row contains a cylinder component, then there are two more columns that contain the radius and height information, respectively.

For the example shown above, the first row of the spreadsheet represents the circuit board itself, which is 1.57 mm thick and 350 mm-by-200 mm. It is offset from the origin by -1.57 mm in the *z* direction and does not dissipate any heat.

You can write the data in the spreadsheet out to a comma-delimited text file, also known as a CSV-file. The user interface of the application used to read the data is shown in the figure below.



The **Settings** window for the **File Import** form object is shown in the figure below.

Preview E populateBoard × form1 ×	Settings		* □‡×
Process File	Name: fileimport	t1	F
Q Q Q + ⊡	Style: Raised		•
	Icon: None		• + 🖻
	Button text: Process Fi	ile	
	Dialog title: File impor	rt	
	File types:		
	CSV File (*.csv)		
	Allow entering files	ame	
	A non chicking neuro	une .	
	 File Destination 		Ð
	✓	File}	
	The Use as Source The F	Edit Node	
	Selected source:		
	File 1 {inputFile}	}	
	Access using: upload:/	///inputFile	
	> Position and Size		
	> Appearance		
	✓ Events		
	On data change: po	opulateBoard •	≣ <u>1</u> + •
	On focus gained: No	one 🔹	≣1 + •

In the **Settings** window, **CSV File (*.csv)** is added to the **File types** list. When browsing for the file, this setting will filter out any file that is not a CSV-file.

There is also a **File Declaration** called **File I**, which is referenced by the file scheme syntax upload:///inputFile in the method populateBoard, which is used to read and process the data. The method is called as an event shown at the bottom of the **Settings** window of the **File Import** form object in the **Events** section.

Note that if you would like to open a file browser from a button or a menu item, instead of using a **File Import** object, you can create a method that calls the built-in method importFile; for example

```
importFile("file1");
```

assuming there is a file declaration file1.

The method populateBoard is listed below.

```
String[][] D = readCSVFile("upload:///inputFile");
model.geom("geom1").feature().clear();
for (int k = 0; k < D.length; k++) {
    if (D[k][0].equals("B")) { // Read in a block
        model.geom("geom1").create("P"+k, "Block").set("pos", new
        String[]{D[k][2], D[k][3], D[k][4]});
        model.geom("geom1").feature("P"+k).set("size", new String[]{D[k][5],
        D[k][6], D[k][7]});
    }
```

```
else if (D[k][0].equals("C")) { // Read in a cylinder
    model.geom("geom1").create("P"+k, "Cylinder").set("pos", new
    String[]{D[k][2], D[k][3], D[k][4]});
    model.geom("geom1").feature("P"+k).set("r", D[k][5]);
    model.geom("geom1").feature("P"+k).set("h", D[k][6]);
  }
  model.geom("geom1").feature("P"+k).set("selresult", "on");
  model.variable().remove("var"+k);
  model.variable().create("var"+k).model("comp1");
  model.variable("var"+k).selction().named("geom1_P"+k+"_dom");
  model.variable("var"+k).set("Q", D[k][1]);
}
model.geom("geom1").run();
zoomExtents("/form1/graphics1");
```

In the first line, the data read from the CSV-file is stored in the 2D array D. The rest of the code parses this array and populates the various parts of a model object. The application allows you to save the result as an MPH file with variables defined for the heat sources and geometry objects defined for the components, as shown in the figures below.



B B Q Q → 5 - C - X Q C	🖶 🍵 🎉 💥 🖳 🗢 layout_model.mph - COMSOL Multip Physics: Mesh Study: Results: Developer: Colored Selec	tions	×
Agşladisin Minager Workspace Model	Pri 3- Variables - Parameters 100 Functions - Definitions - Def	Add Interial	State Au Base Compate Sectors Sectors<
Model Builder + □ =	Settings verifieles	× 0.8	Graphics
Transfilter text			
	Geometric Entity Selection		-
Pi Parameters 1	Geometric entity level: Domain		
Component 1 (comp 7)	Selection: 19 Black 29		
Definitions			
3 Variables	E	à -	200
8+ Variables 2	_	ñ 🗵	150
a+ Variables 3		₽	
3 Variables 4			109
B: Wrinkles 6			
8+ Variables 7	 Variables 		50
3 • Variables 8			
8 Variables 9	Name Expression Unit Description		
ar Variables 10	Q 10		
3+ Variables 12			20 mm
8+ Variables 13			
8+ Variables 14			
3- Variables 15	↑ ↓ 団・ 逆 ≕ 🔪 🍉・ 😡 岐・		
8+ Variables 17	Name		
B+ Variables 18			
3+ Variables 19	Expression		
8+ Veriables 21			Affilus -
8+ Variables 22			RUUS 200
a- Variables 23			· · · ·
3+ Variables 24 De Variables 25			
8+ Variables 25	Description		
a+ Variables 27			100
3 - Variables 28			
3* Write 20			*
a+ Variables 31			
Boundary System 1 (sys 7)			
View 1			
Materiak			z
A Mesh 1			Y Lax
> 🦓 Results			
			Missaes × Promiss Log
			196 (91) 7.37 (9)

You can download the MPH file for this app from:

https://www.comsol.com/model/using-text-files-to-automate-model-preprocessing-46721

Reading Excel Files

The application described above can easily be extended to also read Microsoft Excel[®] Workbook files. Note that this requires LiveLinkTM for Excel[®]. In the **Settings** window for the **File Import** form object, you can add **Microsoft Excel**

Workbook (*.xlsx) and Microsoft Excel Workbook (*.xls) to the File types section, as shown in the figure below.

	File Import	
Process File	Name: fileimport1	
Q, Q, ∯, • ፼ ↓ • ⋈ ⋈ ⋈ ⋈ ⋈ ♂ • ● • ₪ • ₪ ₪	Style: Raised	
	Icon: None	• +
	Button text: Process File	
	Dialog title: File import	
	File types:	
	CSV File (*.csv) Microsoft Excel Workbook (*.xlsx Microsoft Excel Workbook (*.xls)	
	↑ ↓ ≔ + ⊕ ₹	
	Allow entering filename	
	 File Destination 	
	✓	
	🗐 Use as Source 🖅 Edit Node	
	Selected source:	
	Leg File 1 {inputFile}	
	Access using: upload:///inputFile	
	> Position and Size	
	> Appearance	
	 Events 	
	On data changernonulateRoa	
	on auta change. populateboa	

The next step is to add a few lines of code in the beginning of the method populateBoard, as shown below.

```
String file_name = getFilePath("upload:///inputFile");
if (file_name.endsWith(".xls") || file_name.endsWith(".xlsx"))
D = readExcelFile("upload:///inputFile");
else if (file_name.endsWith(".csv"))
D = readCSVFile("upload:///inputFile");
else
error("Unknown file type.");
```

Comments

The 2D array D can be defined as a global array in the **Declarations** node in the application tree. Alternatively, it can be declared as an array that is local to the method by adding the line

String[][] D = null;

before the if statement. Which option to choose depends on how you would like to use the 2D array data after having read the file.

The method getFilePath returns the full path and name of the uploaded file. The if statements control which method is used to read the file based on its file extension. The file extension is retrieved with the Java® method endsWith(), which belongs to the String class. Note that you can see which methods are

available for a string by typing the name of the string followed by a period and Ctrl+Space, as shown in the figure below.



WRITING CSV-FILES

You can write to a CSV-file using four different call syntaxes for the method writeCSVFile, depending on if the contents are strings or doubles and whether you would like to overwrite an already existing file or appending to its contents.

In the case above, the contents are a mix of numbers and characters, so the 2D array storing the information needs to be a string array.

Assume that we would like to move one of the components, say, the second to last one, in the file listed above. We would like to change the corresponding line in the file from

B,10,130,150,0,10,20,30

to

B,10,130,140,0,10,20,30

This corresponds to a change in the y-coordinate of one of the blocks from 150 to 140.

The following code shows how to make this change and then write data on this format, assuming that the array D has been declared as a global variable in the **Declarations** node, as described above.

```
int[] sz = matrixSize(D);
D[sz[0]-2][3] = "140.0";
writeCSVFile("temp:///my_layout.csv", D);
fileSaveAs("temp:///my_layout.csv");
```

Comments

The first line stores the size of the 2D array (or matrix) D in a 1-by-2 array (or vector) sz. The second line sets the string value of the *y*-coordinate of the block of the second-to-last row in D.

The line writeCSVFile("temp:///my layout.csv", D);

writes the data to a file my_layout.csv in a temporary folder whose location is determined by the **Preferences** of either COMSOL Multiphysics or COMSOL Server, depending on which software is used to run the application. For example, in a typical Windows® installation of COMSOL Multiphysics, the location will be similar to

```
C:\Users\paul\AppData\Local\Temp\
```

where the username is paul.

WRITING FILES IN GENERAL

Note that as a first step in the example above, the file is written to a temporary file using the writeCSVFile method. This step is done automatically by the application. In the second step, the method fileSaveAs opens a file browser and lets the user of the application choose the file location; for example, a folder on the computer's local file system or to a network folder. This extra step is needed in order for the application to function in a web browser. Due to the security settings of a typical web browser, the application is not permitted to automatically save a file to an arbitrary location. Instead, the application is allowed to save to a few specific locations, including the temp folder, whose location is specified in the **Preferences** window. The other locations are the user and common folders, also specified in the **Preferences** window. For more information, see the *Introduction to Application Builder*.

WRITING EXCEL FILES

If you have licensed LiveLink[™] for Excel[®], then you can write to a Microsoft Excel Workbook file in a way that is similar to that of a CSV-file, with the exception that the append option is not available. The following code, corresponding to the previous CSV-file example, shows how to write to an Excel file.

```
int[] sz = matrixSize(D);
D[sz[0]-2][3] = "140.0";
writeExcelFile("temp:///my_layout.xlsx", D);
fileSaveAs("temp:///my_layout.xlsx");
```

READING MATRIX FILES

Reading files with numerical data in matrix format is easiest when using the readMatrixFromFile method. This method assumes that the file has the spreadsheet format, as available in the model tree **Export** node. The example below shows a file on the spreadsheet format.

%	Model:	my_model.mph	
%	Version:	COMSOL 6.0.0.278	8

```
% Date:
                     Nov 1 2020, 8:00
% Dimension:
                     1
% Nodes:
                     5
% Expressions:
                    1
% Description:
                     Line graph
% X
                     ٧
1.2 -0.45
1.11 -0.3
1.0440468877558806 -0.38655264416650392
1.0416666666666667 -0.491666666666666666
1.02 -0.15
```

The first few lines with comments start with the character % and are ignored by the readMatrixFromFile method. You can optionally omit such lines and just have the numerical part of a file read by readMatrixFromFile. Assume that this file is uploaded to an application using a **File Import** form object and a **File** declaration file1. The following code can then be used to read the data into a double array p.

double p[][] = readMatrixFromFile("upload:///file1");

The code below shows how to both import and visualize these points in an application that, in addition to a **File Import** form object and a **File** declaration file, has a form form1 and a graphics object graphics1.

```
double p[][] = readMatrixFromFile("upload:///file1");
double pt[][] = transpose(p);
String pgTag = model.result().uniquetag("pg");
ResultFeature pg = model.result().create(pgTag, 2);
ResultFeature plot = pg.create("pt1", "PointData");
plot.set("pointdata", pt);
plot.run();
useGraphics(model.result(pgTag), "form1/graphics1");
```

Reading files on the spreadsheet format as a string array can be done with the method readStringMatrixFromFile. Also, in this case, the comment lines will be ignored. The code below shows how you can replace the first few lines in the above example using readStringMatrixFromFile instead of readMatrixFromFile.

```
String p[][] = readStringMatrixFromFile("upload:///file1");
double pt[][] = transpose(toDouble(p));
```

The method readStringMatrixFromFile is most useful when parts of the read file contains text.

WRITING MATRIX FILES

To write numerical matrix data to file, you can use the method writeFile. Assume that you want to write a matrix of random 2D coordinate values to a file on the spreadsheet format; for example:

```
        -0.3604014719437022
        0.06964952539192892

        -0.043869911848460674
        -0.14152948348300798

        0.08279441507358754
        0.3101282179426158

        ...
        0.4139106589169702

        0.15830016844077643
        -0.08445989494858042

        0.38226182707603905
        0.4837445802317204
```

The code below shows how to do this.

```
int N = 100;
double[][] p = new double[N][2];
for (int k = 0; k < N; k++) {
    p[k][0] = Math.random()-0.5;
    p[k][1] = Math.random()-0.5;
}
writeFile("temp:///my_data.txt", p);
fileSaveAs("temp://my_data.txt");
```

The resulting file can now be read back in and plotted by using the code of the previous example. The result, in an application, may look like the figure below.



This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant files for this example are:

- writing_matrix_file.mph
- my_data.txt

Note that you can append data to an already existing file by providing an additional Boolean input argument; for example:

```
writeFile("temp:///my_data.txt", p,true);
fileSaveAs("temp:///my_data.txt");
```

If you would like to export a matrix with a mix of numeric and text data, you can use the writeFile method with a string array instead of a double array. The syntax for this case is otherwise identical to that of the double array shown in the example above.

READING A TEXT FILE TO A STRING

For reading text files into a string, you can use the method readFile. A straightforward use of readFile is for previewing a text file; for example, before importing and parsing it, as illustrated by the example application in the figure below.

Untitled.mph – E		×
C:\COMSOL\text_data_file.txt	Brows	e
File Contents		
File Contents % Model: thermal_actuator_simplified.mph % Version: COMSOL 6.3.0.335 % Date: Apr 18 2025, 10:32 % Dimension: 3 % Nodes: 11733 % Expressions: 1 % Description: Temperature % Length unit: μm % x y z 7 (K) 0.3810970544231538 0.4551612607522232 44.838540844690876 0.2863313080606898 293.9336690609043 0.4551612607522232 44.838540844690876 0.2863313080606898 293.93350613709716 0.66277901355601577 44.994270829182081 0.5113795674167958 293.9355094054882 0.16763099977086143 44.55422183957100 0.532925543284397 293.9355094054882 0.6691613598655677 44.943687339214634 0.7589349672417191 293.9321240078373 0.460809242418805 44.38257051203564 0.795040024343849 293.933212400778373 0.2928932188134586 44.70710678118656 0 293.9342128741982 0.07646339000421445 44.38351027365312 0 <		<
0.616487263468868 44.92535660999578 0 293.931872423403 0.8641989740731759 44.67677571879551 0 293.9318737423403		
0.999999999999998 44.999999999999 0 293,9291901468935 0.6224751340766550 AA.627162600010526 0 202.0220010600000	Abou	.∨ ut

This application has two form objects: a **File Import** form object referencing a **File** declaration file1 and a **Text** form object referencing a string str declared in the **Declarations** node as a global variable.

Q Previe	ew form1 X	$\begin{tabular}{ c c c }\hline \hline \hline \hline \end{array}$ readString $\mbox{$\times$}$	Main Window	× □ ₹	Settings	5	+ □ # ×
					Text		
			Bro	owse	Name: text1		E
Fi	ile Contents				Editable		
\uparrow					✓ Wrap text		
					 Source 		⊕ + ~≌
					✓	rations	
					v and St	str	
					> 🔇 Mode	el (root)	
					📑 Use as So	urce 🖅 Edit Node	
					Selected source	ce:	
					abc String	ı⇒str	
					Initial value:	From data source	•

The **File Import** form object has an **Event** that calls the method readString upon data change.

This method has one line of code, as shown below.

```
str = readFile("upload:///file1");
```

Since the **Text** object is referencing the global string str, the contents of the file are displayed in the **Text** object immediately after import.

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant files for this example are:

- reading_text_file_to_string.mph
- text_data_file.txt

Parsing of smaller text files can be done with readFile in combination with the many text processing methods available in the String class. However, it is often more efficient to use methods in the CsReader class, as described in the section "Processing Text Files using the CsReader and CsWriter Classes" on page 205, especially for larger text files. The reason is that when using the readFile method, the entire file is read into a string with all its contents kept in memory; whereas when using the CsReader class methods, only small portions of the file are kept in memory at any given time.

If you would like to parse smaller text files using readFile, then the built-in "String Methods" on page 172 are useful. The example code below illustrates using the built-in methods findIn, substring, split, as well as the regular Java® methods System.getProperty and String.startsWith. The example parses the header of a text file containing polygon information to retrieve information about the number of points of each polygon in the main body of the file (not shown) as well as the number of properties (for example, color or material property). The header portion of the file may look like the example below.

```
Demo file for string parsing
Created on May 1st 2018
begin_header
number_of_points 4
number_of_properties 4
end_header
```

The figure below shows an extension of the previous demonstration app that parses this type of file.

Untitled.mph	-		\times
C:\COMSOL\simple_string_parsing_demo_file.txt		Brow Parse	vse e File
File Summary			
Number of points:	4		
Number of properties:	4		
File Contents Demo file for string parsing Created on May 1st 2018 begin_header number_of_points 4 number_of_properties 4 end_header			
		Abo	out

The code for parsing the header is listed below. It stores the number of points and properties in the variables numPoints and numProperties, respectively. To keep things simple, no error handling is done. For example, the code assumes that there is exactly one instance of begin_header and end_header.

```
numPoints = 0; // Integer variable declared in Declarations
numProperties = 0; // Integer variable declared in Declarations
fileContents = readFile("upload:///file1"); // String variable declared in
Declarations
String eol = System.getProperty("line.separator"); // Finds the system end of
line
int headerBeginIndex = findIn(fileContents, "begin header");
int headerEndIndex = findIn(fileContents, "end header");
String headerContents = substring(fileContents, headerBeginIndex,
headerEndIndex-headerBeginIndex); // Converts to string array by splitting at
each line
String[] headerContentsArr = split(headerContents, eol);
int ix = 1;
String[] headerRowArr = new String[2];
do {
  // Split each line at space.
  headerRowArr = split(headerContentsArr[ix], " ");
  if (headerRowArr.length == 2) {
    if (headerRowArr[0].trim().equalsIgnoreCase("number of points"))
      numPoints = toInt(headerRowArr[1]);
    if (headerRowArr[0].trim().equalsIgnoreCase("number of properties"))
      numProperties = toInt(headerRowArr[1]);
  }
  ix++:
} while (ix < headerContentsArr.length);</pre>
```

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant files for this example are:

- simple_string_parsing.mph
- simple_string_parsing_demo_file.txt

The Java® String class has many methods for text processing. See the online Java® documentation for more information.

WRITING A STRING TO A TEXT FILE

The following example shows how to use the method writeFile to write a string to file.

```
String contents = "# Created by me\r\n"
    +"# Version 1.0 of this file format \r\n"
    +"# Body follows\r\n"
    +"0 1 \r\n"
    +"2 3\r\n"
    +"4 5\r\n";
writeFile("temp:///my_data.txt", contents);
fileSaveAs("temp://my_data.txt");
```

The use of the file scheme syntax temp:/// is described above in the earlier examples of this section. The end-of-line characters of this example are for Windows®; see also "Special Characters" on page 11.

To append additional data to the same file, for example:

```
String contents_2 = "6 7\r\n"
+"8 9\r\n"
+"10 11\r\n";
```

use an additional Boolean input argument, which appends data when set to true:

```
writeFile("temp:///my_data.txt", contents_2, true);
fileSaveAs("temp:///my_data.txt");
```

PROCESSING TEXT FILES USING THE CSREADER AND CSWRITER CLASSES

The most efficient and flexible way to read and write to a text file is to use the methods in the CsReader and CsWriter classes, respectively. However, using the methods of these classes is more complicated than using any of the built-in methods described above.

The CsReader class inherits all public methods of the abstract Java® class Reader. In a similar way, the CsWriter class inherits all public methods of the abstract Java® class Writer. This means that when using these classes, you get access to a large number of methods for processing text files. These methods are not documented here, but you can find a lot of information with regards to using these methods online as well as in books on Java® programming. In addition, you can see which methods are available by using code-completion Ctrl+Space.

READING TEXT FILES USING THE CSREADER CLASS

The example code below shows how to parse the text file header of the earlier example by using the CsReader class instead of readFile. Just like in the previous example, the header may look like:

```
Demo file for string parsing
Created on May 1st 2018
begin_header
number_of_points 4
number_of_properties 4
end_header
```

and the corresponding code is listed below (compare with the example "Reading a Text File to a String" on page 201).

```
numPoints = 0; // Integer variable declared in Declarations
numProperties = 0; // Integer variable declared in Declarations
maxHeaderLength = 100; // Integer variable declared in Declarations
CsReader reader = openFileStreamReader("upload:///file1");
String line; // Each line in the file
```

```
String[] lineArr; // The contents of each line in an array
  int li = 0; // Line counter
  boolean begin header found = false;
  boolean end header found = false;
  while (!begin header found && li < maxHeaderLength && ((line = reader.readLine())
  != null)) {
    if (line.trim().startsWith("begin header"))
      begin header found = true;
    li++;
  }
  while (begin header found && !end header found && li < maxHeaderLength && ((line
  = reader.readLine()) != null)) {
    lineArr = split(line, " ");
    if (lineArr[0].trim().equalsIgnoreCase("number of points"))
      numPoints = toInt(lineArr[1]);
    if (lineArr[0].trim().equalsIgnoreCase("number of properties"))
      numProperties = toInt(lineArr[1]);
    if (line.trim().startsWith("end header"))
      end header found = true;
    li++;
  }
  reader.close();
  if (!begin header found || !end header found)
    error("File does not have the right format.");
Comments
```

The line

CsReader reader = openFileStreamReader("upload:///file1");

opens a Java $\$ character stream and assigns it to the object reader belonging to the class CsReader.

The while loop condition contains the statement

(line = reader.readLine()) != null)

which is reading a line from the character stream and storing the result in the string line. A line is considered to be terminated by one of the characters carriage return \r, line feed \n, or the composite \r\n. If there are no more lines to read, then null is returned.

For more information on the string methods used in this and earlier examples, including findIn, substring, and split, see the section "Writing a String to a Text File" on page 204.

The line

```
reader.close();
```

closes the stream permanently.

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant files for this example are:

- text_file_parsing.mph
- simple_string_parsing_demo_file.txt

Note that you can see which additional methods are available for the reader object by using Ctrl+Space, as shown in the figure below.



By using Ctrl+Space following a string, you can see the many additional methods available for strings, including the trim method used in the example above:



WRITING TEXT FILES USING THE CSWRITER CLASS

The example of the section "Reading Matrix Files" on page 198 uses an example on the spreadsheet data format. This section contains an example that writes a file

on the sectionwise format, the other primary format in COMSOL Multiphysics for saving results data. Data on the sectionwise format can, for example, be read into an **Interpolation Curve** geometry primitive. A file on the sectionwise format may look like this:

% Version:		COMSC	DL 6.0	.0.2	278
% Date:	5 Date:		2020,	8:0	00
% Descripti	ion:	Inter	polat:	ion	curve
% Coordinat	tes				
-1.1			-0.8		
1.2			-0.9		
0.9			1.3		
-0.8			1.05		
% Elements	(segments)				
1	2				
2	3				
3	4				

The first few lines with comments start with the character % and are ignored when imported as an **Interpolation Curve**. The first section containing data starts on the line after % Coordinates. The second section containing data starts on the line after % Elements (segments). Note that the strings Coordinates and Elements (segments) are not necessary but each section containing data will be assumed to start after each block of comments, regardless of what comes after the character %. There may be additional blocks of data when, for example, exporting **Contour** plot data.

The following example code uses a CsWriter stream to write interpolation curve data to a text file. A template point set p is copied in a circular pattern for a given radius R and number of copies nCopies.

```
CsWriter writer = openFileStreamWriter("temp:///my curve.txt");
int nCopies = 10;
double[][] p = {{-1.2, -0.9}, {0.9, -1.1}, {1.3, 0.8}, {-0.9, 1.0}}; // template
int template length = p.length;
double R = 10;
double px, py;
double pi = Math.PI;
String line;
int i1, i2;
String header = "% Version:\tCOMSOL 6.0.0.278\r\n"
  +"% Date:\tMay 5 2018, 8 : 00\r\n"
  +"% Description:\tInterpolation curve\r\n"
  +"% Coordinates:\r\n";
writer.append(header);
for (int j = 0; j < nCopies; j++) {
  for (int i = 0; i < template length; i++) {</pre>
    px = p[i][0];
    py = p[i][1];
    px = px+R*Math.cos(2*pi*j/nCopies);
```

```
py = py+R*Math.sin(2*pi*j/nCopies);
    line = toString(px)+"\t"+toString(py)+"\r\n";
    writer.append(line);
  }
}
writer.append("% Elements (segments):\r\n");
for (int j = 0; j < nCopies; j++) {
  for (int i = 0; i < template length; i++) {</pre>
    i1 = i+1;
    i2 = (i+1)%template length+1;
    i1 = i1+j*template length;
    i2 = i2+j*template length;
    line = toString(i1)+"\t"+toString(i2)+"\r\n";
    writer.append(line);
  }
}
writer.flush();
writer.close();
fileSaveAs("temp:///my curve.txt");
```

The line

```
CsWriter writer = openFileStreamWriter("temp:///my_curve.txt");
```

opens a Java® character stream and assigns it to the object writer belonging to the class CsWriter.

The line

```
writer.append(header);
```

appends the contents of the string header to the (empty) file my_curve.txt.

The line

```
writer.flush();
```

writes the contents of the character stream buffer to file and empties the buffer but does not close the stream permanently. At this point, you can still write more data to the stream.

The line

```
writer.close();
```

closes the stream permanently. If you wish to write additional data to the file, you have to open the stream again and append additional data.

Just as described above for the reader object, you can see which additional methods are available for the writer object by using Ctrl+Space, as shown in the figure below.



You can import the resulting interpolation data as an **Interpolation Curve** by selecting the **Sectionwise** option for **Data format**. This can be done for a 2D geometry object or for a **Work Plane** in 3D. The figure below shows the data imported to a 2D model.



This example is part of a collection available for download: https://www.comsol.com/model/application-programming-guide-examples-140771 The relevant file for this example is:

• write_interpolation_curve_data.mph

WRITING BINARY FILES

You write data to a binary file by using the methods of the class CsBinaryWriter in a way that is somewhat similar to that of writing text using CsWriter. However, instead of writing strings and characters, you are writing bytes. To see how many bytes each data type requires, see the table in the section "Primitive Data Types" on page 8.

The example code below writes random 3D point data to a binary file. Each point coordinate is stored as a double and takes 8 bytes to store. The first 4 bytes of the file stores the number of points in the file as an int.

To conveniently convert between the regular data types, such as double, int, and byte arrays, the Java® library method java.nio.ByteBuffer is needed. This method is not part of the standard methods available in the Method Editor and you need to use the fully qualified Java® class name java.nio.ByteBuffer, as shown in the example code below.

```
byte[] bytes8 = new byte[8];
byte[] bytes4 = new byte[4];
CsBinaryWriter bwriter = openBinaryFileStreamWriter("temp:///
my binary file.dat");
int N = 1000;
java.nio.ByteBuffer.wrap(bytes4).putInt(N);
bwriter.write(bytes4);
double p[][] = new double[N][3];
for (int k = 0; k < N; k++) {
  p[k][0] = Math.random();
  java.nio.ByteBuffer.wrap(bytes8).putDouble(p[k][0]);
  bwriter.write(bytes8);
  p[k][1] = Math.random();
  java.nio.ByteBuffer.wrap(bytes8).putDouble(p[k][1]);
  bwriter.write(bytes8);
  p[k][2] = Math.random();
  java.nio.ByteBuffer.wrap(bytes8).putDouble(p[k][2]);
  bwriter.write(bytes8);
  bwriter.flush();
}
bwriter.close();
fileSaveAs("temp:///my binary file.dat");
```

Comments

The first two lines declare byte arrays of size 8 and 4, respectively The line CsBinaryWriter bwriter = openBinaryFileStreamWriter("temp:///
my binary file.dat");

opens a Java® byte stream.

The line

int N = 1000; // The number of points

denotes the number of points written to file.

The line

ByteBuffer.wrap(bytes4).putInt(N);

uses the imported ByteBuffer method to convert the integer N to a byte array bytes4 of length 4.

The line

bwriter.write(bytes4);

writes the value of N to file.

The for-loop creates N points and writes each x-, y-, and z-coordinate as doubles using a byte array bytes8 of length 8.

The line

bwriter.flush();

empties the byte buffer and the last two lines

bwriter.close();
fileSaveAs("temp:///my_binary_file.dat");

close the byte stream and display a file browser to the user to select a location to save the binary file.

READING BINARY FILES

Based on the data format of the previous example, the code below reads a corresponding binary file and plots the points as 3D point data.

```
byte[] bytes8 = new byte[8];
byte[] bytes4 = new byte[4];
CsBinaryReader breader = openBinaryFileStreamReader("upload:///file1");
breader.read(bytes4);
int N = java.nio.ByteBuffer.wrap(bytes4).getInt();
double p[][] = new double[N][3];
for (int k = 0; k < N; k++) {
    breader.read(bytes8);
    p[k][0] = java.nio.ByteBuffer.wrap(bytes8).getDouble();
    breader.read(bytes8);
    p[k][1] = java.nio.ByteBuffer.wrap(bytes8).getDouble();
```

```
breader.read(bytes8);
p[k][2] = java.nio.ByteBuffer.wrap(bytes8).getDouble();
}
breader.close();
double pt[][] = transpose(p);
String pgTag = model.result().uniquetag("pg");
ResultFeature pg = model.result().create(pgTag, 3);
ResultFeature plot = pg.create("pt1", "PointData");
plot.set("pointdata", pt);
plot.run();
useGraphics(model.result(pgTag), "form1/graphics1");
```

The line

CsBinaryReader breader = openBinaryFileStreamReader("upload:///file1");

opens a Java® byte stream based on a **File** declaration file1, typically referenced in a **File Browser** form object, as in the earlier examples on reading text files.

The two lines

```
breader.read(bytes4);
int N = ByteBuffer.wrap(bytes4).getInt();
```

read the first 4 bytes and convert them to an int N.

The following for-loop reads chunks of 8 bytes into the byte array bytes8, converts them and stores the results in a 2D double array p.

The line

```
breader.close();
```

closes the byte stream.

The last section of the example code plots the data and is similar to the example in "Reading Matrix Files" on page 198.

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

• reading_and_writing_binary_data.mph

This files demonstrates reading and writing binary point data using a **Settings Form** in the Model Builder, as shown in the figure below.



Additional Comments on Reading and Writing Binary Formats

When processing binary files, there are two formats in which bytes can be stored: little endian and big endian, respectively. By default, Java® uses the big endian format. For example, the line

p[k][2] = ByteBuffer.wrap(bytes8).getDouble();

is the same as

p[k][2] = ByteBuffer.wrap(bytes8).order(ByteOrder.BIG_ENDIAN).getDouble();

In case the format you are reading is on the little endian format, the corresponding line should be

p[k][2] = ByteBuffer.wrap(bytes8).order(ByteOrder.LITTLE_ENDIAN).getDouble();

Converting Interpolation Curve Data

The following method converts a geometry **Interpolation Curve** to an **Interpolation** function (by creating an interpolation table). The method demonstrates extracting

geometry information from the underlying parameterization of an edge. The edge does not have to be an **Interpolation Curve** but can be any single edge.



The curve is checked for being a function curve with monotonously growing x coordinates, which is required in order to be able to convert to an interpolation function. Note that since an interpolation curve is represented using splines, even though the interpolation points form a monotonous sequence, the resulting curve may not; hence, the second consistency check (the first check can potentially be skipped).

```
// Convert using N points
int N = 100;
double monoTol = 1e-6;
int edgeNum=1;
// Update and get geometry information
model.component("comp1").geom("geom1").run("fin");
GeomSequence geom1 = model.component("comp1").geom("geom1");
GeomFeature ic1 = geom1.feature("ic1");
double[][] curvePoints = ic1.getDoubleMatrix("table");
int len = curvePoints.length;
double minX = curvePoints[0][0];
double maxX = curvePoints[len-1][0];
double scale = maxX-minX;
double scaledTol = monoTol*scale;
for (int i = 1; i < len; i++) {</pre>
  if ((curvePoints[i][0]-curvePoints[i-1][0]) < scaledTol) {</pre>
    error("Curve needs to be a function curve with monotonously growing x
      coordinates.");
```

```
}
}
double minMaxS[] = geom1.edgeParamRange(edgeNum);
double minS = minMaxS[0]:
double maxS = minMaxS[1];
double sList[] = new double[N];
for (int k = 0; k < N; k++) {
 sList[k] = (double) (N-1-k)/(double) (N-1)*minS+k/(double) (N-1)*maxS;
}
double[][] XY = geom1.edgeX(1, sList);
for (int j = 1; j < N; j++) {
  if ((XY[j][0]-XY[j-1][0]) < scaledTol) {
    error("Curve needs to be a function curve with monotonously growing x
      coordinates.");
 }
}
// Create interpolation table
model.func().create("int1", "Interpolation");
with(model.func("int1"));
  set("funcname", "int1");
  set("interp", "cubicspline");
  set("extrap", "linear"):
endwith();
model.func("int1").set("table", toString(XY));
```

The method assumes that there is a geometry sequence geom1 with an interpolation curve ic1. It further assumes that there are no other geometry features and that the geometry object has a single edge. The integer N determines how granular the interpolation table should be. It is assumed that there is only one edge in the geometry sequence (edgeNum). Note that the curve parameter range may not be the unit interval (minS does not have to be 0.0 and maxS does not have to be 1.0). To run the method more than once, you can create a cleanup method that contains the lines:

```
model.func().remove("int1");
model.result().remove("pg1");
```

for removing previously created model tree nodes.

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

converting_curve_to_function.mph
The following method retrieves coordinate values for a regularly sampled parametric surface and plots the corresponding points using a **Point Data** plot. The method demonstrates extracting geometry information from the underlying parameterization of a surface. The example assumes that a parametric surface geometry object has already been created.



```
// Sample and plot N-by-N points on a parametric surface
int N = 20:
int faceNum = 1;
// Update and get geometry information
model.component("comp1").geom("geom1").run("fin");
GeomSequence geom1 = model.component("comp1").geom("geom1");
GeomFeature ps1 = geom1.feature("ps1");
double minMaxS[] = geom1.faceParamRange(faceNum);
double minS1 = minMaxS[0];
double maxS1 = minMaxS[1];
double minS2 = minMaxS[2];
double maxS2 = minMaxS[3];
double s1List[] = new double[N];
double s2List[] = new double[N];
for (int k = 0; k < N; k++) {
  s1List[k] = (double) (N-1-k)/(double) (N-1)*minS1+k/(double) (N-1)*maxS1;
  s2List[k] = (double) (N-1-k)/(double) (N-1)*minS2+k/(double) (N-1)*maxS2;
}
```

```
double s12List[][] = new double[N*N][2];
for (int i = 0; i < N; i++) {
  for (int j = 0; j < N; j++) {
    s12List[i+N*i][0] = s1List[i];
   s12List[i+N*j][1] = s2List[j];
  }
}
double[][] XY = geom1.faceX(faceNum, s12List);
// Plot points
String pgTag = model.result().uniquetag("pg");
ResultFeature pg = model.result().create(pgTag, 3);
ResultFeature plot = pg.create("pt1", "PointData");
double[][] p = new double[3][N*N];
double[] color = new double[N*N];
for (int i = 0; i < N*N; i++) {
  p[0][i] = XY[i][0];
  p[1][i] = XY[i][1];
  p[2][i] = XY[i][2];
  color[i] = p[1][i];
}
plot.set("pointdata", p)
  .set("colordata", color)
.set("coloring", "colortable")
  .set("sphereradiusscale", 1);
plot.run();
selectNode(pg);
```

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

• surface_points.mph

Defining a Parametric Sweep

This example shows how to automate the setup of a **Parametric Sweep** by specifying the sweep parameters and their values programmatically.

Assume that we start from a file output.txt on the COMSOL Spreadsheet format which corresponds to the output of a design of experiments (DoE) study, as shown in the figure below.

% Model:	thermal_actuator_simplified	_DOE.mph	
% Version:	COMSOL 6.3.0.335		
% Date:	May 20 2025, 14:33		
% Table:	Design Data (QoI1-2) {tbl3}	-	
% dw (um)	DV (V)	comp1.maxop1(T)	<pre>comp1.intop1(solid.disp)</pre>
25.50454949697162	3.005687642725162	373.2212548702561	3.0257349372546806E-7
17.468671849531766	1.4697397345513465	313.25048202324643	7.732485967203544E-8
18.78860807641181	3.3210276804843897	391.896993669015	3.786589895801099E-7
21.66778761561682	2.1665085434119487	335.85873515691037	1.658148166252236E-7
38.040516156333474	2.670382681941675	354.9167620164062	2.0653056700467287E-7
31.95512040322993	0.7523835418710179	298.25359663640467	1.8960247212796185E-8
35.037880127623026	3.5011362289401093	398.2182778850843	3.5444520992936724E-7
22.558409974305555	3.976987601767621	431.1540698514672	5.313041594422551E-7
23.996500036971995	1.1987190998966106	306.3242984238318	5.049130792448468E-8
29.080872423188556	3.777720027812895	416.4002744511956	4.697575161529692E-7
15.884205970813458	2.4734059589372084	349.38420381835334	2.1562253214056397E-7
37.27766257708829	0.997925418377444	301.998331429077	2.994182257772048E-8
39.762213823779284	4.206468621289461	440.7023720312756	4.766185329309543E-7
33.34611501537075	4.496618241322228	462.91292779565015	6.166921400446574E-7
26.30385892147541	4.915878834948295	497.0555106106453	7.55961507288908E-7
34.52792435543927	1.9094486637294343	325.4096362763589	1.1091056424223514E-7
20.147673897527678	0.5120698403332385	295.5923925271519	9.525459006292485E-9
18.128507793834395	4.717349251563936	485.55128494418966	7.292399411173065E-7
31.129300996017907	2.771422142921289	360.6356683764387	2.5230588848432044E-7
28.291617977455072	1.8001928508733944	322.3141760241563	1.06726886513393E-7

The first two columns are the input parameters and values to this study and the last two columns are the output. Let's say that we now would like to create a parametric sweep based on the two input columns. How do we bring this data into a **Parametric Sweep**?

There are several ways of doing this and below are two examples.

The first example reads parameter values for the parameters dw and DV from a text file. It converts each column into a space-separated string and then configures a parametric sweep for the **Sweep type** option **Specified combinations**. It uses the built-in readStringMatrixFromFile method and the inserts the contents into the parametric sweep node in the current study by setting the properties plistarr, pname, and punit accordingly. In this case the file output.txt is stored in the model, under Libraries > Files, and is accessed using the file scheme syntax embedded:///output.txt.

```
String[][] p = readStringMatrixFromFile("embedded:///output.txt");
String[] pardwArray = getColumn(p, 0);
String[] parVArray = getColumn(p, 1);
String pardw = String.join(" ", pardwArray);
String parV = String.join(" ", parVArray);
model.study("std1").feature("param").set("plistarr", new String[]{pardw, parV});
model.study("std1").feature("param").set("pname", new String[]{"dw", "DV"});
model.study("std1").feature("param").set("punit", new String[]{"dw", "DV"});
```

The corresponding file is available for download (see below).

The second example reads the parameter values for the parameters dw and DV from the same text file and formats them into the parametric sweep file format readable from the user interface. Each row is constructed as:

```
<param_name> "val1 val2 ..." [unit]
```

The resulting two-line table is written to a file that can be loaded as a sweep definition from the **Parametric Sweep** table.

```
double[][] p = readMatrixFromFile("embedded:///output.txt");
double[] pardw = getColumn(p, 0);
double[] parV = getColumn(p, 1);
int len = pardw.length+2; // +2 for parameter name and unit string, respectively
String[][] parSweep = new String[2][len];
parSweep[0][0] = "dw \"";
parSweep[1][0] = "DV \"";
for (int k = 1; k < len-1; k++) {</pre>
  parSweep[0][k] = toString(pardw[k-1],4);
  parSweep[1][k] = toString(parV[k-1],4);
}
parSweep[0][len-1] = "\" [um]";
parSweep[1][len-1] = "\" [V]";
String[][] fileContents = new String[2][1];
fileContents[0][0] = ""; // Not null
fileContents[1][0] = "";
for (int k = 0; k < len; k++) {
 fileContents[0][0] = fileContents[0][0]+" "+parSweep[0][k];
  fileContents[1][0] = fileContents[1][0]+" "+parSweep[1][k];
}
writeFile("temp:///parfile.txt", fileContents);
fileSaveAs("temp:///parfile.txt");
```

The figure below shows the results of importing the parametric sweep file.

Settings Parametric Sweep = Compute C Undate Solution			- □‡
Label: Parametric Sweep			F
 Study Settings 			
Sweep type: Specified combina	tions		•
Parameter name	Parameter value list	Parameter unit	
dw (Height of the cold arm)	 25.50 17.47 18.79 21.67 38.04 31.96 35.04 22.56 24.00 29.08 15.88 	um	
DV (Applied voltage)	 3.006 1.470 3.321 2.167 2.670 0.7524 3.501 3.977 1.199 3.778 2.473 	v	
↑ ↓ + ≔ \ ► • □ ↓ ↓			

Note: the syntax

```
String out = toString(double value, int digits)
```

is used to format the numeric value in a string to a specified number of significant digits.

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771 The relevant file for this example is:

• thermal_actuator_simplified_read_parameters.mph

Using Selections

USING SELECTIONS FOR EDITING GEOMETRY OBJECTS

The following method generates a plate with an array of cylinders. The cylinders may be used, for example, in a difference operation to create an array of holes in the plate.

```
GeomSequence geom = model.component("comp1").geom("geom1");
geom.create("blk1", "Block");
geom.feature("blk1").set("size", new int[]{10, 10, 1});
geom.create("start_cyl1", "Cylinder");
geom.feature("start_cyl1").set("pos", new double[]{2.5, 2.5, 0});
geom.create("arr1", "Array");
geom.feature("arr1").selection("input").set("start_cyl1");
geom.feature("arr1").set("fullsize", new int[]{2, 2, 1});
geom.feature("arr1").set("disp1", new int[]{5, 5, 0});
```

The resulting geometry is shown in the figure below.



Assume now that the resulting geometry, from the previous step, corresponds to an imported geometry object and that you would like to replace the cylinders with larger cylinders before subtracting and generating the holes. Furthermore, assume that the cylinder objects are generated by an external software in such a way that, although you know these objects are cylinders, they are represented as generic geometry objects with no information on radius, height, or position. The following method finds the array of cylinders, extracts coordinate information for each cylinder, deletes the cylinders, creates a new array of wider cylinders, and finally subtracts the cylinders from the plate. The resulting geometry is shown in the figure below.



The method exemplifies retrieval of geometry object names, coordinate information, and the use of selections. To start from another geometry object (which is also necessarily a plate with cylinders), you can replace the string tag in the variable plateAndCylinders accordingly.

```
double selTol = 1e-2; // Selection tolerance
double newR = 1.25; // New cylinder radius
double newH = 1; // New cylinder height
double plateThickness = 1; // Plate thickness
String plateAndCylinders = "arr1";
// Update and split geometry
model.component("comp1").geom("geom1").run("fin");
GeomSequence geom = model.component("comp1").geom("geom1");
geom.run(plateAndCylinders);
geom.create("spl1", "Split");
geom.feature("spl1").selection("input").set(plateAndCylinders);
geom.run("spl1");
// Find extents of geometry in x,y,z directions
double[] bBox = geom.getBoundingBox();
double MinX = bBox[0];
double MaxX = bBox[1];
double MinY = bBox[2];
double MaxY = bBox[3];
double MinZ = bBox[4];
double MaxZ = bBox[5];
// Define scaled coordinate tolerance
```

```
double scale = Math.max(Math.max(MaxX-MinX, MaxY-MinY), MaxZ-MinZ);
double scaleSelTol = scale*selTol;
// Create box selection based on geometry extents
geom.create("boxsel1", "BoxSelection");
with(geom.feature("boxsel1"));
  // Select boundaries inside box in X-Y direction using tolerance
  set("xmin", MinX+scaleSelTol);
  set("xmax", MaxX-scaleSelTol);
  set("ymin", MinY+scaleSelTol);
  set("ymax", MaxY-scaleSelTol);
  set("zmin", MinZ-scaleSelTol);
  set("zmax", MaxZ+scaleSelTol);
  set("condition", "inside");
  set("entitydim", -1); // Select objects
endwith();
geom.run("boxsel1");
// Get object names
String[] so = geom.selection("boxsel1").objects();
int nso = so.length;
// Extract cylinder parameters
double[] MinXC = new double[nso];
double[] MaxXC = new double[nso];
double[] MinYC = new double[nso];
double[] MaxYC = new double[nso];
double[] MinZC = new double[nso];
double[] MaxZC = new double[nso];
double[] bBoxC = new double[6];
for (int i = 0; i < nso; i++) {</pre>
  bBoxC = geom.obj(so[i]).getBoundingBox();
  MinXC[i] = bBoxC[0];
  MaxXC[i] = bBoxC[1];
  MinYC[i] = bBoxC[2];
  MaxYC[i] = bBoxC[3];
  MinZC[i] = bBoxC[4];
  MaxZC[i] = bBoxC[5];
}
double[] radius = new double[nso];
double[] xc = new double[nso];
double[] yc = new double[nso];
double[] zc = new double[nso];
double[] hc = new double[nso];
double ry;
double tol = scale*1e-6; // Tolerance check for skew cylinders, optional
consistency check
for (int i = 0; i < nso; i++) {</pre>
  xc[i] = (MaxXC[i]+MinXC[i])/2;
  yc[i] = (MaxYC[i]+MinYC[i])/2;
  zc[i] = (MaxZC[i]+MinZC[i])/2;
  hc[i] = MaxZC[i]-MinZC[i];
  radius[i] = (MaxXC[i]-MinXC[i])/2;
  ry = (MaxYC[i]-MinYC[i])/2;
```

```
if (Math.abs(radius[i]-ry) > tol)
    error("Object is not a circular cylinder.");
}
// Delete all cylinder objects
geom.create("del1", "Delete");
geom.feature("del1").selection("input").init();
geom.feature("del1").selection("input").set(so);
geom.run("del1");
// Add new cylinders
String[] cylname = new String[nso];
for (int i = 0; i < nso; i++) {</pre>
  cylname[i] = "cyl"+toString(i+1);
  geom.create(cylname[i], "Cylinder");
 with(geom.feature(cylname[i]));
    set("r", newR);
    set("h", newH+2*scaleSelTol);
    set("pos", new double[]{xc[i], yc[i], plateThickness-2*zc[i]-scaleSelTol});
  endwith();
}
// Combine all cylinders into one object
geom.create("uni1", "Union");
geom.feature("uni1").selection("input").set(cylname);
geom.run("uni1");
// Difference between all objects and cylinders
geom.create("dif1", "Difference");
String[] objs = geom.objectNames();
geom.feature("dif1").selection("input").set(objs);
geom.feature("dif1").selection("input2").set("uni1");
geom.run("dif1");
```

Comments

The method assumes that the input geometry objects have the same structure as the plate with cylinders example above. Note that to clear the geometry sequence, you can create a method with the line

```
model.component("comp1").geom("geom1").feature().clear();
```

This can be useful if you are running the main method from above repeatedly, since you need to clear the geometry sequence before each run.

The first of the lines

```
geom.feature("del1").selection("input").init();
geom.feature("del1").selection("input").set(so);
```

initializes the selection to be empty of type object. The second line then selects all objects with names in the array so. In general, a call to init() without input argument means that the selection is for objects and init(n), where n=0,1,2, or 3, means that the selection is for points, edges, faces, and domains, respectively. Note

that in some cases n=-1 is used to denote the object level (instead of an empty input argument); see the *Programming Reference Manual*.

The figure below shows an example based on a larger array of cylinders.



This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant files for this example are:

- plate_with_four_cylinders.mph
- plate_with_four_hundred_cylinders.mph

SELECTING AND PARTITIONING EDGES FOR A CYLINDER OBJECT

The following two examples show how to make selections for a geometry feature that partitions the edges in a geometry. In this case the geometry is a cylinder object but the method is applicable to any geometry object as shown in the subsequent section.

The first example generates a cylinder and then inserts a PartitionEdges feature ("pare1") to subdivide four specific cylinder edges at parametric positions 0.25, 0.50, and 0.75 along their lengths, runs the partition, and then selects the partition feature in the Model Builder for further operations. You can, for example, use such a partition, under the **Mesh** node, to select edges for mesh refinement.

```
clearModel(model);
model.component().create("comp1", true);
ModelNode comp1 = model.component("comp1");
```

```
comp1.geom().create("geom1", 3);
GeomSequence geom1 = comp1.geom("geom1");
geom1.create("cyl1", "Cylinder");
geom1.feature("cyl1").set("r", 0.5);
geom1.feature("cyl1").set("h", 2);
geom1.feature("cyl1").set("pos", new double[]{0, 0, -1});
geom1.run("cyl1");
geom1.create("pare1", "PartitionEdges");
geom1.feature("pare1").selection("edge").set("cyl1", 4, 5, 8, 11);
geom1.feature("pare1").setIndex("param", 0.25, 0);
geom1.feature("pare1").setIndex("param", 0.5, 1);
geom1.feature("pare1").setIndex("param", 0.75, 2);
geom1.run("pare1");
```

The second example is identical apart from the selection part, which is now replaced by a call to the method all, which selects all available edges.

```
clearModel(model);
model.component().create("comp1", true);
ModelNode comp1 = model.component("comp1");
comp1.geom().create("geom1", 3);
GeomSequence geom1 = comp1.geom("geom1");
geom1.create("cyl1", "Cylinder");
geom1.feature("cyl1").set("r", 0.5);
geom1.feature("cyl1").set("h", 2);
geom1.feature("cyl1").set("pos", new double[]{0, 0, -1});
geom1.run("cyl1");
geom1.create("pare1", "PartitionEdges");
geom1.feature("pare1").selection("edge").all();
geom1.feature("pare1").setIndex("param", 0.25, 0);
geom1.feature("pare1").setIndex("param", 0.5, 1);
geom1.feature("pare1").setIndex("param", 0.75, 2);
geom1.run("pare1");
selectNode(geom1.feature("pare1"));
```

The all method can be used for most geometry features for selecting all geometric entities (domains, boundaries, edges, points). The following picture shows the results of the second example, zoomed in on one of the edges.



To view all available selection methods, type

geom1.feature("pare1").selection("edge")

then press Ctrl + Space. The code-completion popup below will list the options, as shown in the figure below:



This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

• cylinder_edge_partition_and_selections.mph

SELECTING AND PARTITIONING EDGES FOR GENERAL OBJECTS

The following examples are similar to the ones in the previous section but show how to make selections for a geometry feature that partitions the edges in one of several geometry objects. This example features a cone and a block object.

The first example shows how to select and partition all edges of all objects.

```
clearModel(model);
  // Create a 3D component and geometry
  model.component().create("comp1", true);
  ModelNode comp1 = model.component("comp1");
  comp1.geom().create("geom1", 3);
  GeomSequence geom1 = comp1.geom("geom1");
  // Create a cylinder object
  geom1.create("cyl1", "Cylinder");
  geom1.feature("cyl1").set("r", 0.5);
  geom1.feature("cyl1").set("h", 2);
  geom1.feature("cyl1").set("pos", new double[]{0, 0, -1});
  geom1.run("cyl1");
  // Create a block object
  geom1.create("blk1", "Block");
  geom1.run("blk1");
  // Get the names of all objects
  String[] objs = model.component("comp1").geom("geom1").objectNames();
  debugLog(objs); // Display all object names in the Debug Log window
  // Create an edge partition feature
  geom1.create("pare1", "PartitionEdges");
  geom1.feature("pare1").selection("edge").all(objs);
  // Note, also the following syntax would work:
  // geom1.feature("pare1").selection("edge").all();
  geom1.feature("pare1").setIndex("param", 0.25, 0);
  geom1.feature("pare1").setIndex("param", 0.5, 1);
  geom1.feature("pare1").setIndex("param", 0.75, 2);
  geom1.run("pare1");
  selectNode(geom1.feature("pare1"));
The call to:
  geom1.feature("pare1").selection("edge").all(objs);
is very similar to using Ctrl+A in the user interface for selecting all edges.
```

```
To instead select all edges of just the second object (the block), use:
```

```
geom1.feature("pare1").selection("edge").all(objs[1]);
```

Finally, to partition just 4 of the second object's edges, use:

```
int[] edges = new int[]{1, 2, 3, 4};
geom1.feature("pare1").selection("edge").set(objs[1], edges);
```

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

• geom_edge_partition_and_selections.mph

USING SELECTIONS TO MEASURE GEOMETRIC OBJECTS

The following example creates a cone of base radius 1 and height 1 and measures its various geometric properties. This code uses selections based on geometric entity numbers, however, the selection techniques from the previous section can also be used.

```
// Clear the model
clearModel(model):
// Create a 3D component and geometry
model.modelNode().create("comp1");
model.geom().create("geom", 3);
// Add a cone to the geometry
ModelNode comp1 = model.component("comp1");
GeomSequence geom = comp1.geom("geom");
geom.create("cone1", "Cone");
geom.feature("cone1").set("specifytop", "radius");
geom.feature("cone1").set("rtop", 0);
geom.run("cone1");
// Build the geometry
model.geom("geom").run();
// Measure the properties of the geometry
// Select the volumetric domain of the cone object
geom.measure().selection().init(3);
geom.measure().selection().set("cone1", 1);
// Measure properties from domain selection
double vol = geom.measure().getVolume(); // The volume of the cone
double volArea = geom.measure().getBoundaryArea(); // The total boundary area of
the cone based on the domain selection
// Select all of the boundaries of cone object
geom.measure().selection().init(2);
geom.measure().selection().all();
// Measure properties from boundary selection (all)
double bndArea = geom.measure().getArea(); // The total boundary area of the cone
based on selecting all boundaries
```

```
// Select one of the boundaries of the cone object (the bottom surface of the cone)
geom.measure().selection().init(2);
geom.measure().selection().set("cone1", 2);
// Measure properties from boundary selection (boundary 2)
double boundaryPerimeterLength = geom.measure().getBoundaryVolume();
// Select the edges of the bottom surface of the cone
geom.measure().selection().init(1);
int[] edges = new int[]{1, 3, 5, 8}; // The list of edges stored in an array
(vector)
geom.measure().selection().set("cone1", edges);
// geom.measure().selection().set("cone1", 1, 3, 5, 8); // Alternative syntax
// Measure properties from edge selection (edges 1,3,5,8)
double edgePerimeterLength = geom.measure().getLength(); // The total boundary
area of the cone based on selecting all boundaries
// Select two of the vertices: the apex and one of the bottom surface vertices
geom.measure().selection().init(0);
int[] vertices = new int[]{1, 3}; // The list of vertices stored in an array
(vector)
geom.measure().selection().set("cone1", vertices);
// Measure properties from vertex selection (vertices 1,3)
double[] dist = geom.measure().getVtxDistance(); // d[0] is the distance, and
d[i] is the distance in the ith coordinate (i = 1, 2, 3).
// Select the apex vertex
geom.measure().selection().set("cone1", 3);
// Get the coordinates for the apex vertex (3)
double[] coord = geom.measure().getVtxCoord();
// Display the results in the Debug Log window
debugLog("Cone volume = "+vol);
debugLog("Cone area, from domain selection = "+volArea);
debugLog("Cone area, from boundary selection = "+bndArea);
debugLog("Cone bottom surface perimeter length from boundary selection = "+
boundaryPerimeterLength);
debugLog("Cone bottom surface perimeter length from edge selection = "+
edgePerimeterLength);
debugLog("Distance between two points = "+dist[0]);
debugLog("X-distance between two points = "+dist[1]);
debugLog("Y-distance between two points = "+dist[2]);
debugLog("Z-distance between two points = "+dist[3]);
debugLog("X-coordinate of apex vertex = "+coord[0]);
debugLog("Y-coordinate of apex vertex = "+coord[1]);
debugLog("Z-coordinate of apex vertex = "+coord[2]);
// Note that you can also display the results in the Messages window, for example:
message("Cone volume = "+vol);
```

The syntax:

```
geom.measure().selection().init(0);
geom.measure().selection().init(1);
```

```
geom.measure().selection().init(2);
geom.measure().selection().init(3);
```

is used to change the selection mode to vertices, edges, boundaries, and domains, respectively.

The syntax:

```
geom.measure().selection().set("cone1", 1, 3, 5, 8);
```

is equivalent to:

int[] edges = new int[]{1, 3, 5, 8}; geom.measure().selection().set("cone1", edges);

and follows the general pattern:

geom.measure().selection().set(objectName, entityNumbers);

where objectName is the name of the geometry object and entityNumbers is an integer vector of geometric entity numbers (vertices, edges, boundaries, or domains).

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

• measure_cone.mph

For examples of measuring similar quantities for the entire model using Results tools, see "Measuring Model Quantities" on page 235.

USING SELECTIONS ON THE FINALIZED GEOMETRY

Selections applied to the finalized geometry differ from those applied to individual geometry objects within the geometry sequence under the **Geometry** node. Finalized geometry selections are used by Model Builder nodes such as **Materials**, **Mesh**, and physics interfaces like **Heat Transfer in Solids**. They are also used under a model component, such as **Component I**, for defining **Variables**, **Nonlocal Couplings**, and other features that operate on the finalized geometry.

The following example sets up and runs a 3D stationary heat transfer simulation, using selections defined by direct reference to geometric entity numbers.

```
clearModel(model);
model.component().create("comp1", true);
model.component("comp1").geom().create("geom1", 3);
model.component("comp1").geom("geom1").geomRep("comsol");
model.component("comp1").mesh().create("mesh1");
model.component("comp1").geom("geom1").create("cyl1", "Cylinder");
model.component("comp1").geom("geom1").feature("cyl1").set("h", 0.1);
model.component("comp1").geom("geom1").feature("cyl1").set("r", 0.005);
model.component("comp1").geom("geom1").run();
```

```
model.component("comp1").geom("geom1").create("arr1", "Array");
model.component("comp1").geom("geom1").feature("arr1").selection("input").set("
cyl1");
model.component("comp1").geom("geom1").feature("arr1").set("fullsize", new
int[]{3, 3, 1});
model.component("comp1").geom("geom1").feature("arr1").set("displ", new
double[]{0.02, 0.02, 0});
model.component("comp1").geom("geom1").run();
model.component("comp1").physics().create("ht", "HeatTransfer", "geom1");
model.component("comp1").material().create("mat1", "Common");
model.component("comp1").material("mat1").label("Steel AISI 4340");
model.component("comp1").material("mat1").propertyGroup("def").set("density",
"7850[kg/m^3]");
model.component("comp1").material("mat1").propertyGroup("def").set("heatcapacit
y", "475[J/(kg*K)]");
model.component("comp1").material("mat1").propertyGroup("def")
  .set("thermalconductivity", new String[]{"44.5[W/(m*K)]", "0", "0", "0",
"44.5[W/(m*K)]", "0", "0", "0", "44.5[W/(m*K)]"});
model.func().create("rn1", "Random");
model.func("rn1").set("type", "uniform");
model.func("rn1").set("nargs", 3);
model.func("rn1").set("mean", 0.5);
model.component("comp1").physics("ht").create("temp1", "TemperatureBoundary",
2);
model.component("comp1").physics("ht").feature("temp1").set("T0", "300[K]");
int[] bnds = new int[]{3, 7, 11, 21, 25, 29, 39, 43, 47};
model.component("comp1").physics("ht").feature("temp1").selection().set(bnds);
model.component("comp1").physics("ht").create("hs1", "HeatSource", 3);
int[] doms = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9};
model.component("comp1").physics("ht").feature("hs1").selection().set(doms);
model.component("comp1").physics("ht").feature("hs1").set("Q0", "1e6*(1+
rn1(x,y,z))");
model.study().create("std1");
model.study("std1").create("stat", "Stationary");
model.study("std1").feature("stat").setSolveFor("/physics/ht", true);
model.study("std1").createAutoSequences("all");
model.sol("sol1").runAll();
model.result().create("pg1", "PlotGroup3D");
model.result("pg1").label("Temperature (ht)");
model.result("pg1").feature().create("vol1", "Volume");
model.result("pg1").feature("vol1").set("colortable", "HeatCameraLight");
model.result("pg1").run();
```

The following lines illustrate how selections are applied to boundary surfaces in a physics interface:

```
model.component("comp1").physics("ht").create("temp1", "TemperatureBoundary",
2);
model.component("comp1").physics("ht").feature("temp1").set("TO", "300[K]");
int[] bnds = new int[]{3, 7, 11, 21, 25, 29, 39, 43, 47};
```

```
model.component("comp1").physics("ht").feature("temp1").selection().set(bnds);
```

In the first line, the final argument 2 specifies that the selection targets entities of dimension 2, that is, boundary surfaces. Aside from this dimension specification, the syntax is similar to selections applied to individual geometry objects.

Similarly, selections for volumetric domains are specified as follows:

```
model.component("comp1").physics("ht").create("hs1", "HeatSource", 3);
int[] doms = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9};
model.component("comp1").physics("ht").feature("hs1").selection().set(doms);
model.component("comp1").physics("ht").feature("hs1").set("Q0", "1e6*(1+
rn1(x,y,z))");
```

Here, the argument 3 in the first line refers to entities of dimension 3, corresponding to domains (volumes) in the geometry.

To use named explicit selections instead of directly referencing entity numbers in physics features, replace the previous code blocks with the following:

```
model.component("comp1").selection().create("sel1", "Explicit");
model.component("comp1").selection("sel1").geom(2);
model.component("comp1").selection("sel1").label("Temperature Boundaries");
int[] bnds = new int[]{3, 7, 11, 21, 25, 29, 39, 43, 47};
model.component("comp1").selection("sel1").set(bnds);
model.component("comp1").physics("ht").create("temp1", "TemperatureBoundary",
2);
model.component("comp1").physics("ht").feature("temp1").set("T0", "300[K]");
model.component("comp1").physics("ht").feature("temp1").selection().named("sel1
");
model.component("comp1").selection().create("sel2", "Explicit");
model.component("comp1").selection("sel2").geom(3);
int[] doms = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9};
model.component("comp1").selection("sel2").label("Heat Source Domains");
model.component("comp1").selection("sel2").set(doms);
model.component("comp1").physics("ht").create("hs1", "HeatSource", 3);
model.component("comp1").physics("ht").feature("hs1").selection().named("sel2")
model.component("comp1").physics("ht").feature("hs1").set("Q0", "1e6*(1+
rn1(x,y,z))");
```

Selections created with model.component("comp1").selection() can be placed anywhere earlier in the code, as long as they are defined before the finalized geometry is created. This approach is generally preferred, as named explicit selections promote reuse and improve the clarity of the model structure.

To retrieve the geometric entities associated with physics feature selections, you can use the inputEntities() method. This should be done after either of the selection creation methods, whether by direct entity numbers or named explicit selections, has been called. The following code demonstrates this:

 $\ensuremath{{//}}$ Retrieve the geometric entities for the Temperature Boundary selection

```
int[] entitiesT =
model.component("comp1").physics("ht").feature("temp1").selection().inputEntiti
es();
debugLog("Geometric entities for the temperature boundary selection:");
debugLog(entitiesT);
// Retrieve the geometric entities for the Heat Source selection
int[] entitiesHS =
model.component("comp1").physics("ht").feature("hs1").selection().inputEntities
();
debugLog("Geometric entities for the heat source domain selection:");
debugLog(entitiesHS);
```

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

• heat_transfer_in_cylindrical_rod_array_selections.mph

Measuring Model Quantities

For measuring geometric and mass properties of a finalized geometry, use the **Measure** options under **Results** > **Evaluation Group** or **Results** > **Derived Values**. Note that, for historical reasons, **Derived Values** are referred to as **Numerical** in the COMSOL API. The **Evaluation Group** option is a more recent addition to COMSOL Multiphysics and is the recommended option. The examples below applies to the Thermal Microactuator Simplified model, which you can find in the Application Libraries at **COMSOL Multiphysics** > **Multiphysics**.

The complete example, including the method code, is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

thermal_actuator_simplified_measure.mph

The two code listings below computes the following quantities, using **Evaluation Group** and **Derived Values**, respectively:

- Volume of the entire model (domain 1)
- Surface are of boundary 3
- Length of edge (line) 173
- Distance between vertices 155 and 167
- Mass of the entire model (domain 1)
- Center of mass coordinates

The first four quantities use purely geometry measure features: MeasureVolume, MeasureSurface, MeasureLine, MeasureDistance.

The last two use the MassProperties feature, which also recomputes volume, so you see the volume twice (once from MeasureVolume, once from MassProperties).

Unit sources:

- Length and distance units are inherited from the **Geometry** node's unit setting.
- Volume and area units come from the model's **Unit System**, defined at the root node

Using Evaluation Group

```
// Create the Evaluation Group
model.result().evaluationGroup().create("eg1", "EvaluationGroup");
// ---- 1) Volume -
model.result().evaluationGroup("eq1").create("meas1", "MeasureVolume");
model.result().evaluationGroup("eq1").feature("meas1").set("unit", "um^3");
model.result().evaluationGroup("eg1").feature("meas1").selection().set(1);
model.result().evaluationGroup("eg1").run();
double[][] volumeResult = model.result().evaluationGroup("eg1").getReal();
double volume = volumeResult[0][0];
String volumeUnit =
model.result().evaluationGroup("eq1").feature("meas1").getString("unit");
debugLog("Volume: "+volume+" "+volumeUnit);
// ---- 2) Surface area ---
model.result().evaluationGroup("eg1").create("meas2", "MeasureSurface");
model.result().evaluationGroup("eq1").feature("meas2").set("unit", "um^2");
model.result().evaluationGroup("eg1").feature("meas2").selection().set(3);
model.result().evaluationGroup("eq1").run();
double[][] areaResult = model.result().evaluationGroup("eg1").getReal();
double area = areaResult[0][1];
String areaUnit =
model.result().evaluationGroup("eg1").feature("meas2").getString("unit");
debugLog("Surface area: "+area+" "+areaUnit);
// ---- 3) Line length ----
model.result().evaluationGroup("eg1").create("meas3", "MeasureLine");
model.result().evaluationGroup("eg1").feature("meas3").set("unit", "um");
model.result().evaluationGroup("eg1").feature("meas3").selection().set(173);
model.result().evaluationGroup("eg1").run();
double[][] lengthResult = model.result().evaluationGroup("eg1").getReal();
double length = lengthResult[0][2];
String lengthUnit =
model.result().evaluationGroup("eg1").feature("meas3").getString("unit");
debugLog("Line length: "+length+" "+lengthUnit);
// ---- 4) Distance between two points ----
```

```
model.result().evaluationGroup("eg1").create("meas4", "MeasureDistance");
model.result().evaluationGroup("eg1").feature("meas4").set("unit", "um");
model.result().evaluationGroup("eg1").feature("meas4").selection().set(155,
167);
// Alternative syntax:
// int[] vertices = new int[]{155, 167};
11
model.result().evaluationGroup("eg1").feature("meas4").selection().set(vertices
);
model.result().evaluationGroup("eq1").run();
double[][] distResult = model.result().evaluationGroup("eq1").getReal();
double distance = distResult[0][3];
String distUnit =
model.result().evaluationGroup("eg1").feature("meas4").getString("unit");
debugLog("Distance: "+distance+" "+distUnit);
// ---- 5) Mass properties -
model.result().evaluationGroup("eg1").create("mass1", "MassProperties");
model.result().evaluationGroup("eg1").feature("mass1").selection().set(1);
model.result().evaluationGroup("eg1").feature("mass1").setIndex("unit", "um^3",
0);
model.result().evaluationGroup("eg1").feature("mass1").setIndex("unit", "mg",
1);
model.result().evaluationGroup("eg1").feature("mass1").setIndex("unit", "um",
2);
// Enable volume, mass and center-of-mass contributions
model.result().evaluationGroup("eg1").feature("mass1").setIndex("contributionen
abled", true, 0);
model.result().evaluationGroup("eg1").feature("mass1").setIndex("contributionen
abled", true, 1);
model.result().evaluationGroup("eg1").feature("mass1").setIndex("contributionen
abled", true, 2);
model.result().evaluationGroup("eg1").run();
double[][] massResult = model.result().evaluationGroup("eg1").getReal();
double massVolume = massResult[0][4];
double massMass = massResult[0][5];
double cmX = massResult[0][6];
double cmY = massResult[0][7];
double cmZ = massResult[0][8];
// Retrieve the unit strings
String[] massUnits =
model.result().evaluationGroup("eg1").feature("mass1").getStringArray("unit");
debugLog("Mass volume: "+massVolume+" "+massUnits[0]);
debugLog("Mass: "+massMass+" "+massUnits[1]);
debugLog("CofM x: "+cmX+" "+massUnits[2]);
debugLog("CofM y: "+cmY+" "+massUnits[2]);
debugLog("CofM z: "+cmZ+" "+massUnits[2]);
selectNode(model.result().evaluationGroup("eg1").feature("meas1"));
```

```
Using Derived Values
```

```
// ---- 1) Volume ----
```

```
model.result().numerical().create("meas1", "MeasureVolume");
model.result().numerical("meas1").set("unit", "um^3");
model.result().numerical("meas1").selection().set(1);
double[][] volumeResult = model.result().numerical("meas1").getReal();
double volume = volumeResult[0][0];
String volumeUnit = model.result().numerical("meas1").getString("unit");
debugLog("Volume: "+volume+" "+volumeUnit);
// ---- 2) Surface area --
model.result().numerical().create("meas2", "MeasureSurface");
model.result().numerical("meas2").set("unit", "um^2");
model.result().numerical("meas2").selection().set(3);
double[][] surfaceResult = model.result().numerical("meas2").getReal();
double area = surfaceResult[0][0];
String areaUnit = model.result().numerical("meas2").getString("unit");
debugLog("Surface area: "+area+" "+areaUnit);
// ---- 3) Line length --
model.result().numerical().create("meas3", "MeasureLine");
model.result().numerical("meas3").set("unit", "um");
model.result().numerical("meas3").selection().set(173);
double[][] lineResult = model.result().numerical("meas3").getReal();
double length = lineResult[0][0];
String lengthUnit = model.result().numerical("meas3").getString("unit");
debugLog("Line length: "+length+" "+lengthUnit);
// ---- 4) Distance between two points ----
model.result().numerical().create("meas4", "MeasureDistance");
model.result().numerical("meas4").set("unit", "um");
model.result().numerical("meas4").selection().set(155, 167);
double[][] distanceResult = model.result().numerical("meas4").getReal();
double distance = distanceResult[0][0];
String distanceUnit = model.result().numerical("meas4").getString("unit");
debugLog("Distance: "+distance+" "+distanceUnit);
// ---- 5) Mass properties ----
model.result().numerical().create("mass1", "MassProperties");
model.result().numerical("mass1").selection().set(1);
model.result().numerical("mass1").setIndex("unit", "um^3", 0);
model.result().numerical("mass1").setIndex("unit", "mg", 1);
model.result().numerical("mass1").setIndex("unit", "um", 2);
// Enable volume, mass and center?of?mass contributions
model.result().numerical("mass1").setIndex("contributionenabled", true, 0);
model.result().numerical("mass1").setIndex("contributionenabled", true, 1);
model.result().numerical("mass1").setIndex("contributionenabled", true, 2);
double[][] massResult = model.result().numerical("mass1").getReal();
double massVolume = massResult[0][0];
double massMass = massResult[1][0];
double cmX = massResult[2][0];
double cmY = massResult[3][0];
double cmZ = massResult[4][0];
// Retrieve the unit strings
String[] massUnits = model.result().numerical("mass1").getStringArray("unit");
```

```
debugLog("Mass volume: "+massVolume+" "+massUnits[0]);
debugLog("Mass: "+massMass+" "+massUnits[1]);
debugLog("CofM x: "+cmX+" "+massUnits[2]);
debugLog("CofM y: "+cmY+" "+massUnits[2]);
debugLog("CofM z: "+cmZ+" "+massUnits[2]);
selectNode(model.result().numerical("meas1"));
```

Comments

Note that when using **Derived Values**, the massResult array comes back transposed compared to the array returned by an **Evaluation Group**.

Instead of using the ASCII letter u for the micro-prefix, you can embed the Unicode micro sign (\u00b5) in your unit strings. For example:

This forces the output to render μ m³ rather than um³.

Rather than hard-coding geometric entity numbers (for example 155, 167), define a named **Explicit** selection and reuse it. For example:

```
// Create the Evaluation Group
model.result().evaluationGroup().create("eg1", "EvaluationGroup");
// Create Explicit named selections
model.component("comp1").selection().create("sel4", "Explicit");
model.component("comp1").selection("sel4").geom(0);
model.component("comp1").selection("sel4").label("Short Distance");
int[] vertices = new int[]{155, 167};
model.component("comp1").selection("sel4").set(vertices);
// ---- Distance between two points ----
model.result().evaluationGroup("eg1").create("meas4", "MeasureDistance");
model.result().evaluationGroup("eg1").feature("meas4").set("unit", "\u00b5m");
model.result().evaluationGroup("eq1").feature("meas4").selection().named("sel4"
);
model.result().evaluationGroup("eq1").run();
double[][] distResult = model.result().evaluationGroup("eq1").getReal();
double distance = distResult[0][0];
String distUnit =
model.result().evaluationGroup("eg1").feature("meas4").getString("unit");
debugLog("Distance: "+distance+" "+distUnit);
```

Note: If you use the **Record Code** or **Record Method** tools, the generated code will match the default user interface behavior, it writes all evaluated results into **Evaluation Group** tables or **Table** objects. It will not include direct assignments into double[][] arrays or double variables as shown above. The 2D array format is needed because each evaluation feature can compute multiple expressions or perform parametric sweeps.

To account for the circumferential contribution in revolved geometries, set the intvolume flag on a surface measure (and similar for other measures):

```
model.result().evaluationGroup("eg1").feature("meas2").set("intvolume", true);
```

This treats the surface integral as a revolved volume.

Using Numerical Results in a Model or Application

The numerical results retrieved using the API are stored in standard Java variables, such as double and String. This allows you to use them directly in custom model method code to automate modeling workflows. Similarly, these variables can be linked to **Declarations** in the Application Builder for use in simulation apps. For example, a double variable storing a length can be connected to a **Data Display** or **Input Field** form object, making it visible or editable to the app user. To see an example of this in practice, open the Tuning Fork demo application from the Application Libraries at **COMSOL Multiphysics** > **Applications**.

Getting Numerical Data

This section provides several examples of how to retrieve general numerical data from models using method code. The examples cover different types of studies including parametric sweeps, time-dependent (transient) simulations, and eigenfrequency analyses.

GETTING VALUES AT A POINT

This example is based on the Steady-State 2D Heat Transfer with Conduction tutorial model, which you can find in the Application Libraries at **COMSOL Multiphysics** > **Heat Transfer**.



Because it is a stationary (steady-state) simulation, it represents one of the simplest scenarios for demonstrating results evaluation.

The examples below, including method code, is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

heat_convection_2d_get_value_at_point.mph

The following code evaluates the temperature at the location defined by a **Cut Point 2D** feature, demonstrating how to use both an **Evaluation Group** and a **Derived Values** feature:

```
Evaluation Group
```

```
// Define a 2D point dataset
model.result().dataset().create("cpt1", "CutPoint2D");
model.result().dataset("cpt1").set("pointx", 0.6);
model.result().dataset("cpt1").set("pointy", 0.2);
// Create an Evaluation Group
model.result().evaluationGroup().create("eg1", "EvaluationGroup");
```

```
// Add an EvalPoint feature to that group
  model.result().evaluationGroup("eg1").create("pev1", "EvalPoint");
  model.result().evaluationGroup("eg1").feature("pev1").set("data", "cpt1");
  // Set the expression to be the temperature T
  model.result().evaluationGroup("eq1").feature("pev1").setIndex("expr", "T", 0);
  // Set the unit using Unicode
  model.result().evaluationGroup("eg1").feature("pev1").
    setIndex("unit", "\u00b0C", 0);
  // (Or ASCII: "degC" instead)
  // Run the group so the feature gets evaluated
  model.result().evaluationGroup("eq1").run();
  // Read back the number and unit
  double[][] pointResult = model.result().evaluationGroup("eg1").getReal();
  double pointValue = pointResult[0][0];
  String pointValueUnit = model.result().evaluationGroup("eg1")
    .feature("pev1").getString("unit");
  // Display result
  debugLog("Value at point: "+pointValue+" "+pointValueUnit);
  // Alternative Messages window syntax: message("Value at point: "+pointValue+"
  "+pointValueUnit);
  // (Optional) select it in the Model Builder
  selectNode(model.result().evaluationGroup("eg1").feature("pev1"));
Derived Values
  // Define a 2D point dataset
  model.result().dataset().create("cpt1", "CutPoint2D");
  model.result().dataset("cpt1").set("pointx", 0.6);
  model.result().dataset("cpt1").set("pointy", 0.2);
  // Add an EvalPoint feature to Derived Values
  model.result().numerical().create("pev1", "EvalPoint");
  model.result().numerical("pev1").set("data", "cpt1");
  // Set the expression to be the temperature T
  model.result().numerical("pev1").setIndex("expr", "T", 0);
  // Set the unit using Unicode
  model.result().numerical("pev1").setIndex("unit", "\u00b0C", 0); // Unicode unit
  syntax
  // (Or ASCII: "degC" instead)
  // Read back the number and unit
  double[][] pointResult = model.result().numerical("pev1").getReal();
  double pointValue = pointResult[0][0];
  String pointValueUnit = model.result().numerical("pev1").getString("unit");
  // Display result
  debugLog("Value at point: "+pointValue+" "+pointValueUnit);
```

```
// Alternative Messages window syntax: message("Value at point: "+pointValue+"
"+pointValueUnit);
```

```
// (Optional) select it in the Model Builder
selectNode(model.result().numerical("pev1"));
```

Note: If you use the **Record Code** or **Record Method** tools, the generated code will match the default user interface behavior, it writes all evaluated results into COMSOL table objects. It will not include direct assignments into double[][] arrays or double variables as shown above. The 2D array format is needed because each evaluation feature can compute multiple expressions or perform parametric sweeps.

Evaluating Two Quantities with an Evaluation Group

You can evaluate multiple quantities at once by using an **Evaluation Group**. In this example, both the temperature and the effective thermal conductivity are evaluated at the point by modifying the middle portion of the previous example:

```
// Set the expressions to be the temperature T and the effective thermal
conductivity
model.result().evaluationGroup("eq1").feature("pev1").setIndex("expr", "T", 0);
model.result().evaluationGroup("eq1").feature("pev1").setIndex("expr",
"ht.kmean", 1);
// Set the unit for Temperature using Unicode
model.result().evaluationGroup("eg1").feature("pev1").
  setIndex("unit", "\u00b0C", 0);
// (Or ASCII: "degC" instead)
// Set the unit for thermal conductivity
model.result().evaluationGroup("eg1").feature("pev1").
  setIndex("unit", "W/(m*K)", 1);
// Run the group so the feature gets evaluated
model.result().evaluationGroup("eq1").run();
// Read back the numbers and units
double[][] pointResult = model.result().evaluationGroup("eg1").getReal();
double pointValue1 = pointResult[0][0]; // Temperature
double pointValue2 = pointResult[0][1]; // Thermal conductivity
String[] pointValueUnitArray = model.result().evaluationGroup("eg1")
  .feature("pev1").getStringArray("unit");
// Display result
debugLog("Value at point: "+pointValue1+" "+pointValueUnitArray[0]);
debugLog("Value at point: "+pointValue2+" "+pointValueUnitArray[1]);
```

The results and units are retrieved as elements of arrays, corresponding to the evaluated expressions.

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

heat_convection_2d_get_value_at_point_2_expression.mph

This file also includes a corresponding method that demonstrates the use of **Derived Values**.

Evaluating Quantities For a Parametric Sweep

Building on the previous example, the figure below shows the results of a parametric sweep where the rectangle height parameter h1 is varied in four steps: 0.9, 1.0, 1.1, and 1.2 m.

Messa	iges $ imes$	Progress	Log	Evaluation Group 1 ×
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	5 алто 8.5 2 алто е-1	850 0.85	1	💼 📺 📕 🧐 📧 📾 🕞 📾 🗏 🏛 🔻
h1 (m)	Tempera	ture (°C), Poin	t: (0.6, 0.2)	Mean effective thermal conductivity (W/(m*K)), Point: (0.6, 0.2)
0.90000	18.187			52.000
1.0000	18.265			52.000
1.1000	18.272			52.000
1.2000	18.340			52.000

For parametric sweeps, such output can also be generated programmatically using a custom method, as demonstrated in the figure below where the results are written to the **Debug Log**.

```
Debug Log

Parameter: 0.9 m

Temperature at point: 18.187109113310555 °C

Conductivity at point: 52.0 W/(m*K)

Parameter: 1.0 m

Temperature at point: 18.265040746500574 °C

Conductivity at point: 52.0 W/(m*K)

Parameter: 1.1 m

Temperature at point: 18.272132734965794 °C

Conductivity at point: 52.0 W/(m*K)

Parameter: 1.2 m

Temperature at point: 18.33985062607462 °C

Conductivity at point: 52.0 W/(m*K)
```

The following code snipped shows how this can be achieved for an **Evaluation Group** in a method.

```
// Create a CutPoint2D dataset at (x=0.6, y=0.2) using the parametric sweep
dataset dset2
model.result().dataset().create("cpt1", "CutPoint2D");
model.result().dataset("cpt1").set("pointx", 0.6);
model.result().dataset("cpt1").set("pointy", 0.2);
model.result().dataset("cpt1").set("data", "dset2"); // Use data from parametric
sweep
// Create an Evaluation Group named eg1
```

```
model.result().evaluationGroup().create("eg1", "EvaluationGroup");
```

```
// Add an EvalPoint feature pev1 to the group
model.result().evaluationGroup("eg1").create("pev1", "EvalPoint");
model.result().evaluationGroup("eg1").feature("pev1").set("data", "cpt1");
// Set expressions to evaluate: temperature and effective thermal conductivity
model.result().evaluationGroup("eq1").feature("pev1").setIndex("expr", "T", 0);
model.result().evaluationGroup("eq1").feature("pev1").setIndex("expr",
"ht.kmean", 1);
// Set the unit for temperature (Unicode)
model.result().evaluationGroup("eg1").feature("pev1").
  setIndex("unit", "\u00b0C", 0);
// (Or ASCII: "degC" instead)
// Set the unit for thermal conductivity
model.result().evaluationGroup("eq1").feature("pev1").
  setIndex("unit", "W/(m*K)", 1);
// Run the Evaluation Group for all parametric values
model.result().evaluationGroup("eq1").run();
// Retrieve evaluated values and their corresponding units
double[][] pointResult = model.result().evaluationGroup("eq1").getReal();
String[] pointValueUnitArray = model.result().evaluationGroup("eq1")
  .feature("pev1").getStringArray("unit");
// Retrieve the unit for the sweep parameter
String parUnit = model.study("std1").feature("param").getString("punit");
// Loop over the sweep and print results
int lengthSweep = pointResult.length;
for (int k = 0; k < \text{lengthSweep}; k++) {
  // Extract values
  double pointValue0 = pointResult[k][0]; // Sweep parameter value (rectangle
height)
  double pointValue1 = pointResult[k][1]; // Temperature
  double pointValue2 = pointResult[k][2]; // Thermal conductivity
  // Display the evaluated results
  debugLog("Parameter: "+pointValue0+" "+parUnit);
  debugLog("Temperature at point: "+pointValue1+" "+pointValueUnitArray[0]);
  debugLog("Conductivity at point: "+pointValue2+" "+pointValueUnitArray[1]);}
// (Optional) select it in the Model Builder
selectNode(model.result().evaluationGroup("eg1").feature("pev1"));
```

Note that the evaluation results are stored in the 2D array pointResult in the same order as shown in the **Evaluation Group** table in the user interface. The first index corresponds to the sweep parameter index, and the second index corresponds to the evaluated expressions. By default, each row begins with the sweep parameter value, followed by the results of the specified expressions.

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

heat_convection_2d_get_value_at_point_one_parameter_sweep.mph

Parametric Sweep with Specific Combinations: Two or More Parameters Continuing with the Steady-State 2D Heat Transfer with Conduction tutorial model, suppose a parametric sweep is performed over two parameters: the rectangle height h1 and the boundary temperature T0. Assume the **Sweep type** is set to **Specific combinations**, with the following values:

- Rectangle height h1: 0.9, 1.0, 1.1, and 1.2 m
- Boundary temperature T0: 100, 150, 200, and 200 degC (°C)

Note: When using **Specific combinations**, the parameter lists must have the same length, in this case 4 simulation runs, with each entry representing one combination.

The figure below shows the Parametric Sweep settings used for this configuration

Settings Parametric Sweep = Compute C Update Solution	1		▼ □ ‡
Label: Parametric Sweep			F
 Study Settings Sweep type: Specified comb 	inations		•
Parameter name	Parameter value list	Parameter unit	
h1 (Height)	• 0.9 1.0 1.1 1.2	m	
T0 (Boundary temperature)	100 150 200 200	°C	
↑↓+≔↓ ► • 😡			

We can now use a breakpoint in the **Method Editor** to stop the code execution at the assignment of the pointResult array and display the results using the **Data Viewer** window:

Data Viewer	▼ □ .
-12	
Name	Value
> Pi Parameters 1	
> 🔇 model	Model (id=630)
✓ ^{8.5} ₈₈₈ pointResult	double[][] Size: 4 × 4 (id=631)
✓ 8.5 [0]	double[] Length: 4 (id=843)
8.5 [0]	0.9
8.5 [1]	100.0
8.5 [2]	18.187109113310555
8.5 [3]	52.0
✓ 8.5 [1]	double[] Length: 4 (id=844)
8.5 [0]	1.0
8.5 [1]	150.0
8.5 [2]	27.39756111975089
8.5 [3]	52.0
✓ 8.5 [2]	double[] Length: 4 (id=845)
a.s [0]	1.1
s.s [1]	200.0
8.5 [2]	36.544265469931645
8.5 [3]	52.0
✓ 8.5 [3]	double[] Length: 4 (id=846)
8.5 [0]	1.2
8.5 [1]	200.0
8.5 [2]	36.67970125214907
8.5 [3]	52.0

For information on using breakpoints, see the *Introduction to Application Builder*.

The code in this case will resemble the one-parameter sweep case. Only the latter part of the code, where the results are processed, needs to be modified, as shown below:

```
// Retrieve units for sweep parameters
String[] parUnit = model.study("std1").feature("param").getStringArray("punit");
// For specific combinations, each row of pointResult already contains h1, T0,
T, and ht.kmean
for (int k = 0; k < pointResult.length; k++) {
    double h1 = pointResult[k][0];
    double T0 = pointResult[k][1];
    double temperature = pointResult[k][2];
    double conductivity = pointResult[k][3];
    debugLog("h1 = "+h1+" "+parUnit[0]);
    debugLog("T0 = "+T0+" "+parUnit[1]);
```

```
debugLog(" Temperature at point: "+temperature+" "+pointValueUnitArray[0]);
debugLog(" Thermal conductivity: "+conductivity+" "+pointValueUnitArray[1]);
debugLog("");
}
```

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant files for this example are:

- heat_convection_2d_get_value_at_point_two_parameter_sweep_specific_combinations.
 mph
- heat_convection_2d_get_value_at_point_four_parameter_sweep_specific_combinations
 .mph

The four-parameter example includes two dummy parameters to demonstrate that the same approach applies when using the **Specific combinations** option with more than two parameters.

Parametric Sweep with All Combinations: Two Parameters

Continuing with the Steady-State 2D Heat Transfer with Conduction tutorial model, assume now that the **Sweep type** is set to **All combinations**, with the following values:

- Rectangle height h1: 0.9, 1.0, 1.1, and 1.2 m
- Boundary temperature T0: 100, 150, and 200 degC (°C)

This results in $4 \times 3 = 12$ simulation runs, covering all combinations of the two parameters. The figure below shows the **Parametric Sweep** settings used for this configuration.

Settings Parametric Sweep = Compute C Update Solution		*□‡
Label: Parametric Sweep		
 Study Settings 	_	_
Sweep type: All combination	5	•
Parameter name	Parameter value list	Parameter unit
h1 (Height) 🔹	0.9 1.0 1.1 1.2	m
T0 (Boundary temperature) 🔹	100 150 200	degC
↑↓+≔↓⊳►・▣⊾	m	

Programmatically, this corresponds to a nested for-loop. This loop is indexed, starting from 1. Once an **Evaluation Group** has been created, the indices can be retrieved using the following call:

```
double[][] sweepIndices =
model.result().evaluationGroup("eg1").feature("pev1").getDoubleMatrix("loopleve
1");
debugLog(sweepIndices);
```

The corresponding output for the array sweepIndices in the **Debug Log** window is:

 $\{\{1, 2, 3\}, \{1, 2, 3, 4\}\}$

However, in the following example this array will not be needed.

We can use a breakpoint in the **Method Editor** to stop the code execution at the assignment of the pointResult array and display the results using the **Data Viewer** window:

85	
Name	Value
 Pi Parameters 1 	
> 8.5 W	0.6 m
> 8.5 h1	1.0 m
> 8.5 TO	373.15 K
> 🔇 model	Model (id=725)
✓ ^{8.5} / _{8.5} pointResult	double[][] Size: 12 × 4 (id=726)
✓ 8.5 [0]	double[] Length: 4 (id=945)
8.5 [0]	0.9
8.5 [1]	100.0
8.5 [2]	18.187109113310555
8.5 [3]	52.0
✓ 8.5 [1]	double[] Length: 4 (id=946)
e.s [0]	0.9
8.5 [1]	150.0
8.5 [2]	27.280663669965747
8.5 [3]	52.0
✓ 8.5 [2]	double[] Length: 4 (id=947)
8.5 [0]	0.9
8.5 [1]	200.0
8.5 [2]	36.37421822662094
8.5 [3]	52.0
✓ 8.5 [3]	double[] Length: 4 (id=948)
8.5 [0]	1.0
8.5 [1]	100.0
8.5 [2]	18.265040746500574
8.5 [3]	52.0
> 8.5 [4]	double[] Length: 4 (id=949)
> 8.5 [5]	double[] Length: 4 (id=950)
> 8.5 [6]	double[] Length: 4 (id=951)
> 8.5 [7]	double[] Length: 4 (id=952)
> 8.5 [8]	double[] Length: 4 (id=953)
> 8.5 [9]	double[] Length: 4 (id=954)
> 8.5 [10]	double[] Length: 4 (id=955)
> 8.5 [11]	double[] Length: 4 (id=956)

Note that the pointResult array has dimensions [12][4], where:

• The first index corresponds to the parameter combination, with the sweep ordered such that T0 varies fastest (that is, inner loop), and h1 varies slowest (outer loop).

- Each row contains:
 - Column 0: h1
 - Column 1: T0
 - Column 2: Computed temperature at the point
 - Column 3: Effective thermal conductivity at the point

In the output, you can confirm this organization:

- Rows [0-2] all have h1 = 0.9, with T0 = 100, 150, 200
- Rows [3-5] have h1 = 1.0, again sweeping over the same TO values
- and so on...

This structure follows the row-major order of all combinations: for each value of h1, all values of T0 are iterated through.

The following code snippet shows how data can be retrieved in a method.

```
// Optionally make sure to use Unicode label for temperature in parametric sweep
model.study("std1").feature("param").setIndex("punit", "\u00b0C", 1);
```

```
// Create a CutPoint2D dataset at (x=0.6, y=0.2) using the parametric sweep
dataset dset2
model.result().dataset().create("cpt1", "CutPoint2D");
model.result().dataset("cpt1").set("pointx", 0.6);
model.result().dataset("cpt1").set("pointy", 0.2);
model.result().dataset("cpt1").set("data", "dset2"); // Use data from parametric
sweep
// Create an Evaluation Group named eq1
model.result().evaluationGroup().create("eg1", "EvaluationGroup");
// Add an EvalPoint feature pev1 to the group
model.result().evaluationGroup("eg1").create("pev1", "EvalPoint");
model.result().evaluationGroup("eg1").feature("pev1").set("data", "cpt1");
// Set expressions to evaluate: temperature and effective thermal conductivity
model.result().evaluationGroup("eg1").feature("pev1").setIndex("expr", "T", 0);
model.result().evaluationGroup("eg1").feature("pev1").setIndex("expr",
"ht.kmean", 1);
// Use description labels that are easy to use with Java's string utilities (using
underscore)
model.result().evaluationGroup("eq1").feature("pev1").setIndex("descr",
"Point temperature", 0);
model.result().evaluationGroup("eq1").feature("pev1").setIndex("descr",
"Point thermal conductivity", 1);
// Set the unit for temperature (Unicode)
model.result().evaluationGroup("eq1").feature("pev1").
  setIndex("unit", "\u00b0C", 0);
// (Or ASCII: "degC" instead)
```

```
// Set the unit for thermal conductivity
```

```
model.result().evaluationGroup("eq1").feature("pev1").
    setIndex("unit", "W/(m*K)", 1);
  // Run the Evaluation Group for all parametric values
  model.result().evaluationGroup("eq1").run();
  // Retrieve evaluated values and their corresponding units
  double[][] pointResult = model.result().evaluationGroup("eq1").getReal();
  String[] pointValueUnitArray = model.result().evaluationGroup("eg1")
    .feature("pev1").getStringArray("unit");
  // Retrieve the units for the sweep parameters
  String[] parUnits =
  model.study("std1").feature("param").getStringArray("punit");
  // Retrieve the description labels from the Evaluation Group column headers,
  assuming they use underscore in their names
  String[] columnHeaders =
  model.result().evaluationGroup("eg1").getColumnHeaders();
  String[] headerNames = new String[columnHeaders.length];
  for (int i = 0; i < columnHeaders.length; i++) {</pre>
    headerNames[i] = columnHeaders[i].split(" ")[0];
  }
  // Loop over each row of results
  for (int i = 0; i < pointResult.length; i++) {</pre>
    double h1 = pointResult[i][0];
    double T0 = pointResult[i][1];
    double pointTemperature = pointResult[i][2];
    double pointConductivity = pointResult[i][3];
    debugLog(headerNames[0]+" = "+h1+" "+parUnits[0]+" , "+headerNames[1]+" = "+
  TO+" "+parUnits[1]);
    debugLog(headerNames[2]+": "+pointTemperature+" "+pointValueUnitArray[0]);
    debugLog(headerNames[3]+": "+pointConductivity+" "+pointValueUnitArray[1]);
    debugLog("");
  }
  // Optionally select it in the Model Builder
  selectNode(model.result().evaluationGroup("eg1").feature("pev1"));
The (abridged) output from this code is as follows:
```

```
h1 = 0.9 m , T0 = 100.0 °C
Point_temperature: 18.187109113310555 °C
Point_thermal_conductivity: 52.0 W/(m*K)
h1 = 0.9 m , T0 = 150.0 °C
Point_temperature: 27.280663669965747 °C
Point_thermal_conductivity: 52.0 W/(m*K)
h1 = 0.9 m , T0 = 200.0 °C
Point_temperature: 36.37421822662094 °C
Point_thermal_conductivity: 52.0 W/(m*K)
h1 = 1.0 m , T0 = 100.0 °C
Point_temperature: 18.265040746500574 °C
```
```
Point_thermal_conductivity: 52.0 W/(m*K)
...
h1 = 1.2 m , T0 = 200.0 °C
Point_temperature: 36.67970125214907 °C
Point_thermal_conductivity: 52.0 W/(m*K)
```

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

 heat_convection_2d_get_value_at_point_two_parameter_sweep_all_combinat ions.mph

Parametric Sweep with All Combinations: Three or More Parameters

Expanding on the previous example, let us now introduce a dummy parameter par1, for demonstration purposes.

= Compute C Update Solution			
Label: Parametric Sweep			Ē
 Study Settings 			
Sweep type: All combination	ns		•
Parameter name	Parameter value list	Parameter unit	
h1 (Height)	• 0.9 1.0 1.1 1.2	m	
T0 (Boundary temperature)	• 100 150 200	degC	
par1 (Dummy parameter)	12	kg	

As before, we can use a breakpoint in the **Method Editor** to pause execution at the assignment of the pointResult array and inspect the contents using the **Data Viewer** window.

Data Viewer	
8.85 e-12	
Name	Value
✓ Pi Parameters 1	
> 8.5 W	0.6 m
> ≋.≊ h1	1.0 m
> 8.5 TO	373.15 K
> 8.5 par1	1.0 kg
> 🔇 model	Model (id=915)
✓ ^{8.5} ₈₈₈ pointResult	double[][] Size: 24 × 5 (id=916)
✓ ^{8.5} _{man} [0]	double[] Length: 5 (id=920)
8.5 [O]	0.9
8.5 [1]	100.0
8.5 [2]	1.0
8.5 [3]	18.187109113310555
8.5 [4]	52.0
✓ 8.5 [1]	double[] Length: 5 (id=921)
8.5 [0]	0.9
8.5 [1]	100.0
8.5 [2]	2.0
8.5 [3]	18.187109113310555
8.5 [4]	52.0
✓ ^{8.5} / _{8.5} [2]	double[] Length: 5 (id=922)
e.s [0]	0.9
8.5 [1]	150.0
8.5 [2]	1.0
8.5 [3]	27.280663669965747
8.5 [4]	52.0
✓ 8.5 [3]	double[] Length: 5 (id=923)
8.5 [0]	0.9
8.5 [1]	150.0
8.5 [2]	2.0
8.5 [3]	27.280663669965747
8.5 [4]	52.0
> 8.5 [4]	double[] Length: 5 (id=924)

Note that the pointResult array has dimensions [24][5], where:

• The first index corresponds to the parameter combination, with the sweep ordered such that par1 varies fastest (that is, inner loop), then T0 varies intermediately, and h1 varies slowest (outer loop).

- Each row contains:
 - Column 0: h1
 - Column 1: T0
 - Column 2: par1
 - Column 3: Computed temperature at the point
 - Column 4: Effective thermal conductivity at the point

The code used for a two-parameter sweep with the **All Combinations** option can be generalized as follows, by modifying the latter part of the loop:

```
// Previous code
// ...
/// Loop over each row of results
int numSweepParams = 3; // or headerNames.length - number of result expressions
for (int i = 0; i < pointResult.length; i++) {</pre>
  // Print all sweep parameters in one line
  String paramLine = "";
  for (int p = 0; p < numSweepParams; p++) {</pre>
    if (p > 0) paramLine += ", ";
    paramLine += headerNames[p]+" = "+pointResult[i][p]+" "+parUnits[p];
  }
  debugLog(paramLine);
  // Print evaluated results
  for (int j = 0; j < pointValueUnitArray.length; j++) {</pre>
    debugLog(" "+headerNames[numSweepParams+j]+": "+
             pointResult[i][numSweepParams+j]+" "+pointValueUnitArray[j]);
  debugLog("");
}
```

This code generalizes to more than three parameters.

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant files for this example are:

- heat_convection_2d_get_value_at_point_three_parameter_sweep_all_combinations.mph
- heat_convection_2d_get_value_at_point_four_parameter_sweep_all_combinations.mph

The second example demonstrates the case of four parameters.

Parametric Sweep that Includes an Auxiliary Sweep

A model can include nested parametric sweeps, consisting of an outer sweep and an inner sweep, where the inner sweep typically represents physics-related parameters, such as material properties or boundary conditions, rather than geometric dimensions or mesh settings. One common type of inner sweep is the auxiliary sweep. For example, in the Steady-State 2D Heat Transfer with Conduction tutorial model, the boundary temperature can be assigned as an auxiliary sweep parameter, whereas the rectangle height, being a geometric property, cannot.

The process of extracting results using an **Evaluation Group** remains essentially the same, even when an auxiliary sweep is used.

Assume again that the parameters take the following values:

- Rectangle height h1: 0.9, 1.0, 1.1, and 1.2 m
- Boundary temperature T0: 100, 150, and 200 degC (°C)

The outer sweep is defined as shown in the figure below.

Settings Parametric Sweep = Compute C Update Solution				
Label: Parametric Sweep			, =	
 Study Settings 				
Sweep type: Specifi	ed combinations		•	
Parameter name	Parameter value list	Parameter unit		
h1 (Height) • 0.9 1.0 1.1 1.2 m				
† + 🕂 🎽 🏲	• 🔲 🛄			

The auxiliary sweep (inner sweep) is defined in settings window for **Step 1: Stationary** as shown in the figure below.

weep type:	Specified	combinations	
Parameter name		Parameter value list	Parameter unit
T0 (Boundary temperature) 🔹		100 150 200	°C

The following code snippet shows how data can be retrieved in a method.

```
// Optionally make sure to use Unicode label for temperature in parametric sweep
model.study("std1").feature("stat").setIndex("punit", "\u00b0C", 0); //
Auxiliary sweep unit
```

```
// Create a CutPoint2D dataset at (x=0.6, y=0.2) using the parametric sweep
dataset dset2
model.result().dataset().create("cpt1", "CutPoint2D");
model.result().dataset("cpt1").set("pointx", 0.6);
model.result().dataset("cpt1").set("pointy", 0.2);
```

```
model.result().dataset("cpt1").set("data", "dset2"); // Use data from parametric
sweep
// Create an Evaluation Group named eq1
model.result().evaluationGroup().create("eq1", "EvaluationGroup");
// Add an EvalPoint feature pev1 to the group
model.result().evaluationGroup("eg1").create("pev1", "EvalPoint");
model.result().evaluationGroup("eq1").feature("pev1").set("data", "cpt1");
// Set expressions to evaluate: temperature and effective thermal conductivity
model.result().evaluationGroup("eq1").feature("pev1").setIndex("expr", "T", 0);
model.result().evaluationGroup("eg1").feature("pev1").setIndex("expr",
"ht.kmean", 1);
// Use description labels that are easy to use with Java's string utilities (using
underscore)
model.result().evaluationGroup("eq1").feature("pev1").setIndex("descr",
"Point temperature", 0);
model.result().evaluationGroup("eq1").feature("pev1").setIndex("descr",
"Point thermal conductivity", 1);
// Set the unit for temperature (Unicode)
model.result().evaluationGroup("eq1").feature("pev1").
  setIndex("unit", "\u00b0C", 0);
// (Or ASCII: "degC" instead)
// Set the unit for thermal conductivity
model.result().evaluationGroup("eg1").feature("pev1").
  setIndex("unit", "W/(m*K)", 1);
// Run the Evaluation Group for all parametric values
model.result().evaluationGroup("eq1").run();
// Retrieve evaluated values and their corresponding units
double[][] pointResult = model.result().evaluationGroup("eq1").getReal();
String[] pointValueUnitArray = model.result().evaluationGroup("eq1")
  .feature("pev1").getStringArray("unit");
// Retrieve the unit for the sweep parameters
// Since h1 is the only parameter in the outer sweep, retrieve its unit directly
as a string
String parUnit = model.study("std1").feature("param").getString("punit");
// Retrieve the unit for the auxiliary sweep parameter
String auxUnit = model.study("std1").feature("stat").getString("punit");
// Retrieve the description labels from the Evaluation Group column headers,
assuming they use underscore in their names
String[] columnHeaders =
model.result().evaluationGroup("eg1").getColumnHeaders();
String[] headerNames = new String[columnHeaders.length];
for (int i = 0; i < columnHeaders.length; i++) {</pre>
 headerNames[i] = columnHeaders[i].split(" ")[0];
}
```

```
// Loop over each row of results
for (int i = 0; i < pointResult.length; i++) {
    double h1 = pointResult[i][0];
    double T0 = pointResult[i][1];
    double pointConductivity = pointResult[i][2];
    double pointConductivity = pointResult[i][3];

    debugLog(headerNames[0]+" = "+h1+" "+parUnit+" , "+headerNames[1]+" = "+T0+"
"+auxUnit);
    debugLog(headerNames[2]+": "+pointTemperature+" "+pointValueUnitArray[0]);
    debugLog(headerNames[3]+": "+pointConductivity+" "+pointValueUnitArray[1]);
    debugLog("");
}
// Optionally select it in the Model Builder
selectNode(model.result().evaluationGroup("eg1").feature("pev1"));</pre>
```

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

 heat_convection_2d_get_value_at_point_one_parameter_sweep_plus_auxilia ry_sweep.mph

COMPUTING THE AVERAGE ALONG A BOUNDARY IN A PARAMETRIC SWEEP

Continuing with the Steady-State 2D Heat Transfer with Conduction tutorial model, consider the case of a parametric sweep over one parameter: the rectangle height. Suppose now that we want to compute the average temperature along the right boundary of the rectangle.

In this case, a point dataset is not required, but aside from that, the code remains very similar to the approach demonstrated earlier. One important detail is to ensure that the correct dataset corresponding to the parametric sweep is used. In this example, that is the dataset with the tag dset2.

The following code snippet shows how to use an **Evaluation Group** to retrieve the average temperature on a boundary within a method.

```
// Create the Evaluation Group and set it to use the parametric sweep dataset
model.result().evaluationGroup().create("eg1", "EvaluationGroup");
model.result().evaluationGroup("eg1").set("data", "dset2"); // Set sweep dataset
// Add an AvLine feature to the Evaluation Group to compute average values along
a boundary
model.result().evaluationGroup("eg1").create("av1", "AvLine");
// model.result().evaluationGroup("eg1").feature("av1").set("intsurface",
true); // Needed if an axisymmetric model
model.result().evaluationGroup("eg1").feature("av1").selection().set(4); //
Apply to boundary 4
// Set expressions and units
```

```
model.result().evaluationGroup("eg1").feature("av1").setIndex("expr", "T", 0);
model.result().evaluationGroup("eq1").feature("av1").setIndex("expr",
"ht.kmean", 1);
model.result().evaluationGroup("eq1").feature("av1").setIndex("unit",
"\u00b0C", 0);
model.result().evaluationGroup("eq1").feature("av1").setIndex("unit", "W/
(m*K)", 1);
// Set optional labels
model.result().evaluationGroup("eg1").feature("av1").setIndex("descr",
"Avg temperature", 0);
model.result().evaluationGroup("eg1").feature("av1").setIndex("descr",
"Avg thermal conductivity", 1);
// Run the Evaluation Group
model.result().evaluationGroup("eg1").run();
// Retrieve results and units
double[][] resultArray = model.result().evaluationGroup("eq1").getReal();
String[] resultUnits = model.result().evaluationGroup("eg1")
  .feature("av1").getStringArray("unit");
// Retrieve sweep parameter unit
String parUnit = model.study("std1").feature("param").getString("punit");
// Loop over results
for (int i = 0; i < resultArray.length; i++) {</pre>
  double sweepVal = resultArray[i][0]; // Sweep parameter (h1)
  double avgTemp = resultArray[i][1]; // Average temperature
  double avgCond = resultArray[i][2]; // Average conductivity
  debugLog("Parameter: "+sweepVal+" "+parUnit);
 debugLog(" Average temperature: "+avgTemp+" "+resultUnits[0]);
  debugLog(" Average conductivity: "+avgCond+" "+resultUnits[1]);
}
```

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

heat_convection_2d_get_value_at_point_one_parameter_sweep_global.mph

The feature type used in the **Evaluation Group** for computing an average along a line is AvLine. Similar feature types exist for other combinations of geometric entities and evaluation operations, as summarized in the table below.

FEATURE TYPE	DESCRIPTION
AvVolume	Computes the average of an expression over a domain
AvSurface	Computes the average over a boundary (2D surface in 3D)
AvLine	Computes the average over an edge (curve in 2D/3D)

FEATURE TYPE	DESCRIPTION
IntVolume	Integrates an expression over a domain
IntSurface	Integrates over a boundary surface
IntLine	Integrates over an edge
MaxVolume	Computes the maximum value within a domain
MaxSurface	Computes the maximum over a boundary
MaxLine	Computes the maximum along an edge
MinVolume	Computes the minimum value within a domain
MinSurface	Computes the minimum over a boundary
MinLine	Computes the minimum along an edge

COMPUTING GLOBAL QUANTITIES IN A PARAMETRIC SWEEP

Continuing with the Steady-State 2D Heat Transfer with Conduction tutorial model and a parametric sweep over the rectangle height, suppose we now want to compute a global quantity, for example, the number of degrees of freedom (DOFs) in the model for each parameter value. (A global quantity is one that is not associated with a specific geometric entity.)

This value will vary throughout the sweep because, as the rectangle height changes, the mesh is regenerated and the number of mesh elements, and thus the number of DOFs, changes.

The variable name for the number of DOFs is numberofdofs.

The following code snippet shows how to use an **Evaluation Group** to retrieve the number of degrees of freedom at each step of a parametric sweep within a method.

```
// Create the Evaluation Group and set it to use the parametric sweep dataset
model.result().evaluationGroup().create("eg1", "EvaluationGroup");
model.result().evaluationGroup("eg1").set("data", "dset2"); // Set sweep dataset
// Add a Global Evaluation feature to the Evaluation Group
model.result().evaluationGroup("eg1").create("gev1", "EvalGlobal");
// Set the expression to evaluate: number of DOFs
model.result().evaluationGroup("eg1").feature("gev1").set("expr", new
String[]{"numberofdofs"});
// Optional: Set a label and unit
model.result().evaluationGroup("eg1").feature("gev1").set("descr", new
String[]{"Number of degrees of freedom"});
model.result().evaluationGroup("eg1").feature("gev1").set("unit", new
String[]{"1"});
// Run the evaluation group for all sweep points
model.result().evaluationGroup("eg1").run();
```

```
// Retrieve the results (1 column: DOFs, 1 row per sweep point)
double[][] dofResults = model.result().evaluationGroup("eg1").getReal();
// Retrieve the unit for the sweep parameter
String parUnit = model.study("std1").feature("param").getString("punit");
// Loop through results
for (int i = 0; i < dofResults.length; i++) {
    double h1 = dofResults[i][0]; // Sweep parameter (h1)
    int ndof = (int) dofResults[i][1]; // Number of DOFs
    debugLog("h1 = "+h1+" "+parUnit);
    debugLog(" Number of DOFs: "+ndof);
}</pre>
```

Note: As an alternative to using **Evaluation Group** features such as AvLine and MaxVolume, you can define a nonlocal coupling operator under **Component** > **Definitions** > **Nonlocal Couplings**, and then evaluate it as a global quantity.

This example is part of a collection available for download and is included in the same file as the previous example, "Computing the Average Along a Boundary in a Parametric Sweep" on page 258.

NESTED PARAMETRIC SWEEP NODES

Instead of using a single **Parametric Sweep** feature with the **All combinations** or **Specified combinations** option, you can use multiple nested **Parametric Sweep** nodes. Accessing results from a nested sweep is very similar to retrieving data from a sweep that uses only one **Parametric Sweep** node.

Consider the earlier example where the sweep type is set to **All combinations**, with the following parameter values:

- Rectangle height h1: 0.9, 1.0, 1.1, and 1.2 m
- Boundary temperature T0: 100, 150, and 200 degC (°C)

This sweep can also be performed by nesting two **Parametric Sweep** nodes, as illustrated in the figures below.



The following code snippet shows how to retrieve data from such a nested sweep. While the process is similar to the single-node case, there are some minor differences.

```
// Optionally make sure to use Unicode label for temperature in parametric sweep
model.study("std1").feature("param").setIndex("punit", "\u00b0C", 0);
model.study("std1").feature("param2").setIndex("punit", "m", 0);
// Create a CutPoint2D dataset at (x=0.6, y=0.2) using the parametric sweep
dataset dset2
model.result().dataset().create("cpt1", "CutPoint2D");
model.result().dataset("cpt1").set("pointx", 0.6);
model.result().dataset("cpt1").set("pointy", 0.2);
model.result().dataset("cpt1").set("data", "dset2"); // Use data from parametric
sweep
// Create an Evaluation Group named eg1
model.result().evaluationGroup().create("eg1", "EvaluationGroup");
// Add an EvalPoint feature pev1 to the group
model.result().evaluationGroup("eg1").create("pev1", "EvalPoint");
```

```
model.result().evaluationGroup("eq1").feature("pev1").set("data", "cpt1");
// Set expressions to evaluate: temperature and effective thermal conductivity
model.result().evaluationGroup("eq1").feature("pev1").setIndex("expr", "T", 0);
model.result().evaluationGroup("eq1").feature("pev1").setIndex("expr",
"ht.kmean", 1);
// Use description labels that are easy to use with Java's string utilities (using
underscore)
model.result().evaluationGroup("eq1").feature("pev1").setIndex("descr",
"Point temperature", 0);
model.result().evaluationGroup("eq1").feature("pev1").setIndex("descr",
"Point thermal conductivity", 1);
// Set the unit for temperature (Unicode)
model.result().evaluationGroup("eq1").feature("pev1").
  setIndex("unit", "\u00b0C", 0);
// (Or ASCII: "degC" instead)
// Set the unit for thermal conductivity
model.result().evaluationGroup("eq1").feature("pev1").
  setIndex("unit", "W/(m*K)", 1);
// Run the Evaluation Group for all parametric values
model.result().evaluationGroup("eg1").run();
// Retrieve evaluated values and their corresponding units
double[][] pointResult = model.result().evaluationGroup("eg1").getReal();
String[] pointValueUnitArray = model.result().evaluationGroup("eg1")
  .feature("pev1").getStringArray("unit");
// Retrieve the units for the sweep parameters
String parUnit1 = model.study("std1").feature("param").getString("punit");
String parUnit2 = model.study("std1").feature("param2").getString("punit");
// Retrieve the description labels from the Evaluation Group column headers,
assuming they use underscore in their names
String[] columnHeaders =
model.result().evaluationGroup("eg1").getColumnHeaders();
String[] headerNames = new String[columnHeaders.length];
for (int i = 0; i < columnHeaders.length; i++) {</pre>
 headerNames[i] = columnHeaders[i].split(" ")[0];
}
// Loop over each row of results
for (int i = 0; i < pointResult.length; i++) {</pre>
  double h1 = pointResult[i][0];
  double T0 = pointResult[i][1];
  double pointTemperature = pointResult[i][2];
  double pointConductivity = pointResult[i][3];
 debugLog(headerNames[0]+" = "+h1+" "+parUnit2+" , "+headerNames[1]+" = "+T0+"
"+parUnit1);
  debugLog(headerNames[2]+": "+pointTemperature+" "+pointValueUnitArray[0]);
  debugLog(headerNames[3]+": "+pointConductivity+" "+pointValueUnitArray[1]);
  debugLog("");
```

```
}
// Optionally select it in the Model Builder
selectNode(model.result().evaluationGroup("eq1").feature("pev1"));
```

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

 heat_convection_2d_get_value_at_point_two_parameter_sweep_nested_featu re_nodes.mph

NESTED PARAMETRIC SWEEP NODES, RAGGED SWEEP

In certain cases, the nested node option gives you additional flexibility. For example, you can perform sweeps where the inner parameter is a function of the outer parameter. This results in an irregular sweep where some parameter combinations are missing. Other names for this type of sweep are jagged or ragged sweep.

The following example of a spinning aluminum disk demonstrates a sweep over a geometry with a varying number of holes depending on the hole radius. The example is available for download (see below) and demonstrates accessing results for a ragged sweep.

```
// Set number of displayed digits
int digits = 4;
// Set parametric sweep units
model.study("std1").feature("param").setIndex("punit", "1", 0);
model.study("std1").feature("param1").setIndex("punit", "m", 0);
// Create an Evaluation Group named eg1 and set it to the parametric sweep dataset
model.result().evaluationGroup().create("eg1", "EvaluationGroup");
model.result().evaluationGroup("eg1").set("data", "dset2");
// Create an average surface evaluation feature
model.result().evaluationGroup("eg1").create("av1", "AvSurface");
model.result().evaluationGroup("eg1").feature("av1").selection().set(1);
// Set expressions to evaluate and their units
model.result().evaluationGroup("eq1").feature("av1").setIndex("expr",
"solid.disp", 0);
model.result().evaluationGroup("eg1").feature("av1").setIndex("expr",
"solid.mises", 1);
model.result().evaluationGroup("eg1").feature("av1").setIndex("unit",
"\u00b5m", 0);
model.result().evaluationGroup("eg1").feature("av1").setIndex("unit", "MPa",
1);
```

 $\ensuremath{//}$ Use description labels that are easy to use with Java's string utilities (using underscore)

```
model.result().evaluationGroup("eq1").feature("av1").setIndex("descr",
"Displacement magnitude", 0);
model.result().evaluationGroup("eq1").feature("av1").setIndex("descr",
"von Mises stress", 1);
// Run the Evaluation Group for all parametric values
model.result().evaluationGroup("eq1").run();
// Retrieve evaluated values and their corresponding units
double[][] pointResult = model.result().evaluationGroup("eq1").getReal();
String[] pointValueUnitArray = model.result().evaluationGroup("eg1")
  .feature("av1").getStringArray("unit");
// Retrieve the units for the sweep parameters
// String parUnit1 = model.study("std1").feature("param").getString("punit"); /
/ parUnit1 is not needed since it is 1 (the unit of the number of holes)
String parUnit2 = model.study("std1").feature("param1").getString("punit");
// Retrieve the description labels from the Evaluation Group column headers,
assuming they use underscore in their names
String[] columnHeaders =
model.result().evaluationGroup("eg1").getColumnHeaders();
String[] headerNames = new String[columnHeaders.length];
for (int i = 0; i < columnHeaders.length; i++) {</pre>
 headerNames[i] = columnHeaders[i].split(" ")[0];
}
// Loop over each row of results
for (int i = 0; i < pointResult.length; i++) {</pre>
 int numHoles = (int) (pointResult[i][0]);
 String holeRadius = toString(pointResult[i][1], digits);
 String avDisp = toString(pointResult[i][2], digits);
 String avStress = toString(pointResult[i][3], digits);
 debugLog(headerNames[0]+" = "+numHoles+" , "+headerNames[1]+" = "+holeRadius+
" "+parUnit2);
 debugLog(headerNames[2]+": "+avDisp+" "+pointValueUnitArray[0]);
 debugLog(headerNames[3]+": "+avStress+" "+pointValueUnitArray[1]);
 debugLog("");
}
// Optionally select it in the Model Builder
selectNode(model.result().evaluationGroup("eg1").feature("av1"));
```

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

spinning_disk_ragged_sweep_nested_feature_nodes.mph

RETRIEVING DATA FROM A TRANSIENT SIMULATION

This example is based on the Axisymmetric Transient Heat Transfer tutorial model, which you can find in the Application Libraries at **COMSOL Multiphysics** > **Heat Transfer**.

This is a transient (time-dependent) simulation and retrieving transient data is very similar to that of a parametric sweep. You will notice that the code below is very similar to the example in "Evaluating Quantities For a Parametric Sweep" on page 244.

```
// Remove any existing features
model.result().evaluationGroup().remove("eg1");
model.result().dataset().remove("cpt1");
// Create a CutPoint2D dataset at (R=0.1 (pointx), Z=0.3 (pointy))
// Note, this is an axisymmetric model
model.result().dataset().create("cpt1", "CutPoint2D");
model.result().dataset("cpt1").set("pointx", 0.1);
model.result().dataset("cpt1").set("pointy", 0.3);
// Create an Evaluation Group named eq1
model.result().evaluationGroup().create("eg1", "EvaluationGroup");
// Add an EvalPoint feature pev1 to the group
model.result().evaluationGroup("eg1").create("pev1", "EvalPoint");
model.result().evaluationGroup("eg1").feature("pev1").set("data", "cpt1");
// Set expressions to evaluate: temperature and effective thermal conductivity
model.result().evaluationGroup("eg1").feature("pev1").setIndex("expr", "T", 0);
model.result().evaluationGroup("eg1").feature("pev1").setIndex("expr",
"ht.kmean", 1);
// Set or confirm the unit for time (optional, defaults to seconds)
model.study("std1").feature("time").set("tunit", "s");
// Set the unit for temperature (Unicode)
model.result().evaluationGroup("eg1").feature("pev1").
  setIndex("unit", "\u00b0C", 0);
// (Or ASCII: "degC" instead)
// Set the unit for thermal conductivity
model.result().evaluationGroup("eg1").feature("pev1").
  setIndex("unit", "W/(m*K)", 1);
// Run the Evaluation Group for all parametric values
model.result().evaluationGroup("eg1").run();
// Retrieve evaluated values and their corresponding units
double[][] pointResult = model.result().evaluationGroup("eg1").getReal();
String[] pointValueUnitArray = model.result().evaluationGroup("eg1")
  .feature("pev1").getStringArray("unit");
// Retrieve the unit for the transient parameter
String timeUnit = model.study("std1").feature("time").getString("tunit");
```

```
// Loop over output times (that is, times stored in the solution), which may
differ from internal solver steps
int lengthTimeList = pointResult.length;
for (int k = 0; k < lengthTimeList; k++) {</pre>
  // Extract values
  double pointValue0 = pointResult[k][0]; // Time value (similar to a sweep
parameter value)
  double pointValue1 = pointResult[k][1]; // Temperature
  double pointValue2 = pointResult[k][2]; // Thermal conductivity
  // Display the evaluated results
  debugLog("Time: "+pointValue0+" "+timeUnit);
  debugLog("Temperature at point: "+pointValue1+" "+pointValueUnitArray[0]);
  debugLog("Conductivity at point: "+pointValue2+" "+pointValueUnitArray[1]);
  debugLog("");
}
// (Optional) select it in the Model Builder
selectNode(model.result().evaluationGroup("eq1").feature("pev1"));
```

The (abridged) output from this code is as follows:

```
Time: 0.0 s
Temperature at point: 5.307062156134634E-5 °C
Conductivity at point: 52.0 W/(m*K)
Time: 10.0 s
Temperature at point: 0.002430733497874371 °C
Conductivity at point: 52.0 W/(m*K)
Time: 20.0 s
Temperature at point: 0.056482830296829434 °C
Conductivity at point: 52.0 W/(m*K)
...
Time: 380.0 s
Temperature at point: 410.03241468409624 °C
Conductivity at point: 52.0 W/(m*K)
```

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

```
    heat_transient_axi_get_value_at_point_.mph
```

Accessing The Length of a Parametric List

There are multiple ways to retrieve the number of parameters or time steps in a solution. In the previous examples, calls such as:

```
double[][] pointResult = model.result().evaluationGroup("eg1").getReal();
int lengthSweep = pointResult.length;
```

can be used to retrieve the length of the sweep.

An alternative for an Evaluation Group is:

```
int nOfRows = model.result().evaluationGroup("eg1").getNRows();
```

which returns the number of rows in the corresponding result table.

If you instead have access to a Plot Group (for example, with the tag pg1), you can use:

```
int lengthOfSweep = model.result("pg1").getStepCount(0);
```

Here, the argument to getStepCount represents the parameter index. However, note that getStepCount belongs to a different part of the API that is more tightly linked to the solution data structures. As a result, the ordering of parameters differs from that of an Evaluation Group.

For sweeps involving 1, 2, or 3 parameters, the order is simply the reverse of that used in an Evaluation Group. For 4 parameters, however, the last two parameters are combined into a single sequence. Alternatively, you can think of it as the first two parameters being grouped together.

To illustrate this, consider a sweep over the parameters h1, T0, par1, and par2, as shown in the figure below.

Settings Parametric Sweep = Compute C Update Solution Label: Parametric Sweep			+
			_
 Study Settings 			
Sweep type: All combinations 🔹			
Parameter name	Parameter value list	Parameter unit	
h1 (Height) 🔹	0.9 1.0 1.1 1.2	m	
T0 (Boundary temperature) 🔹 🔻	100 150 200	°C	
par1 (Dummy parameter 1) 🔹 🔻	12	kg	
par2 (Dummy parameter 2) 🔹 🔻	3 4	kg	
↑ ↓ + ≔ ∖ ► • 🛛 📖			

A corresponding 2D Plot Group setting is shown in the figure below.



Here, we can see that the first two parameters h1 and T0 are grouped together. An expanded view of this list is presented in the following figure.

Settings 2D Plot Group Plot I $\leftarrow \rightarrow \rightarrow \equiv \bullet$	v	□ #
Label: Temperature (ht) 1		,=
✓ Data		
Dataset:	Study 1/Parametric Solutions 1 (sol2) {dset2} 🔻	Ē1
Parameter value (h1 (m),T0 (°C)):	1: h1=0.9 m, T0=100°C	•
Parameter value (par1 (kg)):	1: h1=0.9 m, T0=100°C	
- oronieter forde (part (kgy)	2: h1=0.9 m, T0=150°C	
Parameter value (par2 (kg)):	3: h1=0.9 m, T0=200°C	
> Selection	4: h1=1 m, T0=100°C	
> Selection	5: h1=1 m, T0=150°C	
> Title	6: h1=1 m, T0=200°C	
V Plot Sottings	7: h1=1.1 m, T0=100°C	
 Plot Settings 	8: h1=1.1 m, T0=150°C	
View: Automatic	9: h1=1.1 m, T0=200°C	
u evis labels 🔲	10: h1=1.2 m, T0=100°C	
x-axis label:	11: h1=1.2 m, T0=150°C	
y-axis label:	12: h1=1.2 m, T0=200°C	

This list contains 4x3=12 entries, which is reflected by the value returned by: int lengthOfSweep = model.result("pg1").getStepCount(2); The figure below show all the returned values for the inputs 0, 1, and 2, using the Java Shell window.



ACCESSING A SUBSET OF A LIST OF OUTPUT TIMES

A list of output times can be very long and there could be reasons to access only the solutions for a subset. Consider the example in the section "Retrieving Data from a Transient Simulation" on page 266. Recall that this example uses a point evaluation at a cut point.

To access a subset of output times based on index, you can use the following syntax:

```
model.result().evaluationGroup("eg1").feature("pev1").setIndex("looplevelinput"
, "manualindices", 0);
model.result().evaluationGroup("eg1").feature("pev1").setIndex("looplevelindice
s", "range(5,10,45)", 0);
```

To access a subset of output times based on interpolated times, you can use the following syntax:

```
model.result().evaluationGroup("eg1").feature("pev1").setIndex("looplevelinput"
, "interp", 0);
model.result().evaluationGroup("eg1").feature("pev1").setIndex("interp",
"range(11.5,0.5,25.5)", 0);
```

Note that for interpolated times, the time values given as input do not need to match the output times of the **Time Dependent** study step.

If you want to clear the **Debug Log** window before running the method, you can call the clearDebugLog() method.

However, keep in mind that the primary purpose of these examples is not to print results to the **Debug Log** window, but rather to illustrate how data can be retrieved, processed, and used in other methods or incorporated into simulation apps.

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

heat_transient_axi_get_value_at_point_subset.mph

As an alternative to using an **Evaluation Group**, one can instead use a SolutionInfo object. This technique is more general but also more advanced.

The example below illustrates how the first and last time output times can be retrieved from a model using this technique.

```
// Use a SolutionInfo object to retrieve the first and last output times
SolverSequence sol = model.sol("sol1");
SolutionInfo info = sol.getSolutioninfo();
int[] indx1 = info.getIndices(0, new int[]{0}); // Not needed but used for
demonstration purposes
double[][] values = info.getVals(0, new int[]{0});
double firstTime = values[0][0];
double lastTime = values[0][indx1.length-1];
```

The the methods getIndices and getVals are described in more detail in the *Programming Reference Manual* in the section *SolutionInfo Object and its Methods*.

UPDATE A CHOICELIST DYNAMICALLY FOR OUTPUT TIMES

The following method updateOutputTimeList is part of an app where a Combo Box form object is used to select the output time used for evaluation and visualization.

```
// Set dummy expression 1 and use an Evaluation Group to evaluate first and last
times
model.result().evaluationGroup().create("eg1", "EvaluationGroup");
EvaluationGroupFeature eg1 = model.result().evaluationGroup("eg1");
eg1.create("gev1", "EvalGlobal");
eg1.feature("gev1").setIndex("expr", 1, 0);
eg1.run();
// Get first and last output times
eg1.setIndex("looplevelinput", "first", 0);
eg1.run();
double firstTime = eg1.getReal()[0][0];
eg1.setIndex("looplevelinput", "last", 0);
eq1.run();
double lastTime = eg1.getReal()[0][0];
// Calculate N evenly spaced times
int N = numberOfOutputTimes;
double[] times = new double[N];
for (int i = 0; i < N; i++) {
  times[i] = firstTime+i*(lastTime-firstTime)/(N-1);
}
```

```
// Convert to string and interpolate
String[] timeStrs = new String[N];
for (int i = 0; i < N; i++) {</pre>
  timeStrs[i] = toString(times[i], digits);
}
String interpList = String.join(" ", timeStrs);
// Set expression to evaluate (average)
model.result().evaluationGroup().create("eg2", "EvaluationGroup");
EvaluationGroupFeature eq2 = model.result().evaluationGroup("eq2");
eg2.create("av1", "AvVolume");
eg2.feature("av1").setIndex("expr", "T", 0);
eg2.set("data", "rev1");
eq2.setIndex("looplevelinput", "interp", 0);
eg2.setIndex("interp", interpList, 0);
eg2.run();
// Update choicelist and the current output time used for plot and evaluation
app.declaration("choicelist1").setList(timeStrs, timeStrs);
outputTimeStr = timeStrs[N-1];
changeOutputTime();
```

The method evaluates a dummy expression, "1", to retrieve the first and last output times. As an alternative, one can use the SolutionInfo method described in the section "Accessing a Subset of a List of Output Times" on page 270.

The call

app.declaration("choicelist1").setList(timeStrs, timeStrs);

is used to dynamically update choicelist1, which is used by a Combo Box in the app's user interface.

The code for the changeOutputTime method is as follows.

```
EvaluationGroupFeature eg2 = model.result().evaluationGroup("eg2");
eg2.setIndex("interp", outputTimeStr, 0);
eg2.run();
double[][] pointResult = eg2.getReal();
pointValue = toString(pointResult[0][1], digits);
model.result("pg4").setIndex("looplevel", "interp", 0);
model.result("pg4").set("interp", outputTimeStr);
model.result("pg4").run();
model.result("pg2").setIndex("looplevel", "interp", 0);
model.result("pg2").set("interp", outputTimeStr);
model.result("pg2").run();
```

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant files for this example are:

- heat_transient_axi_get_average_value_for_subset_evaluation_group_app.m
 ph
- heat_transient_axi_get_average_value_for_subset_solution_info_app.mph

ACCESSING EIGENFREQUENCY DATA

This example is based on the Tuning Fork tutorial model, which you can find in the Application Libraries at **COMSOL Multiphysics > Structural Mechanics**.

This is a combined parametric and eigenfrequency simulation that is also very similar to that of a parametric sweep. You will notice that the code below is very similar to the example in "Evaluating Quantities For a Parametric Sweep" on page 244.

The code below evaluates the eigenfrequencies for a subset of parameter values from a parametric sweep, using a Global Evaluation feature inside an Evaluation Group.

```
// Remove existing features if needed
model.result().evaluationGroup().remove("eq1");
// Create the Evaluation Group and set it to use the parametric sweep dataset
model.result().evaluationGroup().create("eq1", "EvaluationGroup");
model.result().evaluationGroup("eg1").set("data", "dset2"); // Set sweep dataset
// Add a Global Evaluation feature to the Evaluation Group
model.result().evaluationGroup("eg1").create("gev1", "EvalGlobal");
// Set the expression to evaluate a dummy expression 1
model.result().evaluationGroup("eg1").feature("gev1").setIndex("expr", "1", 0);
model.result().evaluationGroup("eg1").setIndex("looplevelinput",
"manualindices", 0);
model.result().evaluationGroup("eg1").setIndex("looplevelindices", "7 8 9", 0);
model.result().evaluationGroup("eq1").setIndex("looplevelinput", "manual", 1);
model.result().evaluationGroup("eg1").setIndex("looplevel", "1 2 3", 1);
// Run the evaluation group for all sweep points
model.result().evaluationGroup("eg1").run();
// Retrieve the unit for the sweep parameter
String parUnit = model.study("std1").feature("param").getString("punit");
// Retrieve the unit for the eigenfrequency parameter
//String auxUnit = model.study("std1").feature("stat").getString("punit");
String efqUnit = model.study("std1").feature("eig").getString("eigunit");
// Retrieve the results
double[][] efqResults = model.result().evaluationGroup("eg1").getReal();
// Loop through results
```

```
for (int i = 0; i < efqResults.length; i++) {
   double L = efqResults[i][0]; // Sweep parameter (L)
   double efq = efqResults[i][1]; // Eigenfrequency (resonant frequency)
   debugLog("L = "+L+" "+parUnit);
   debugLog(" Eigenfrequency: "+efq+" "+efqUnit);
   debugLog("");
}</pre>
```

Specifically it creates an Evaluation Group named eg1 and sets its dataset to dset2, which contains results from a parametric eigenfrequency sweep.

It uses dummy expression 1 to trigger evaluation at the selected sweep points. It manually selects which parametric indices and eigenmode indices to evaluate:

- looplevelindices = "7 8 9" selects specific parameter values (e.g., geometry or material settings).
- looplevel = "1 2 3" selects specific eigenmodes at each sweep point.

The Evaluation Group is then run, and the output table contains:

- The parameter value (L) used in the sweep.
- The corresponding eigenfrequency for each selected mode.

Finally, the code logs each parameter and eigenfrequency pair. The eigenfrequencies vary with the parameter L, due to the fact that the resonance frequency is a function of the length of the tuning fork.

The (abridged) output from this code is as follows:

```
L = 0.078 m
Eigenfrequency: 451.12805182824405 Hz
L = 0.078 m
Eigenfrequency: 692.8198415933232 Hz
L = 0.078 m
Eigenfrequency: 1678.173090553073 Hz
...
L = 0.07875 m
Eigenfrequency: 1657.9561032429424 Hz
```

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

• tuning_fork_get_eigenfrequency_data.mph

Recursion and Recursively Defined Geometry Objects

Methods can support recursion by having a function calling itself in a recursive loop. The following examples create recursive CAD geometry models of a Sierpinski carpet in 2D and a Menger sponge in 3D.

SIERPINSKI CARPET

The following method, create_carpet, initiates the recursion to a certain recursive level according to a user-defined **Parameter** mslevel, defined under **Global Definitions** in the **Model Builder**. It sets a limit at 5 levels in order to avoid creating an exceedingly large geometry. The method assumes that you have created a 2D Component and that you have declared an integer scalar variable counter.

```
int level = (int) model.param().evaluate("mslevel");
if (level < 1)
  error("Carpet level needs to be at least 1.");
if (level > 5)
  error("Carpet level needs to be at most 5.");
counter = 0;
model.component("comp1").geom("geom1").feature().clear();
model.component("comp1").geom("geom1").autoRebuild("off");
double cx0 = 0, cy0 = 0;
double si0 = 1;
carpet(level, cx0, cy0, si0);
model.component("comp1").geom("geom1").runPre("fin");
```

The method create_carpet in turn calls the main recursive function carpet, listed below, using four input arguments for the recursion level, the center x- and y-coordinates, and the current side length.

The input arguments are defined in the method's **Settings** window, as shown below.

Chowin M	et Andel Puilder			
	iouer builder			
Inputs a	nd Output			
Inputs				
Name	Туре	Default	Description	Unit
evel	Integer	•	Recursive level	
x	Double	•	Center x coordinate	
y	Double	•	Center y coordinate	
			Courses side low oth	

The code for the method carpet is shown below.

```
int l = level;
  double posx, posy, si1;
  String strix;
  int l1;
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
      posx = cx+i*si-si;
      posy = cy+j*si-si;
      strix = toString(i)+toString(j);
      if ((Math.abs((i-1))+Math.abs((j-1))) > 0) {
        if (1 == 1) {
          counter = counter+1;
          model.component("comp1").geom("geom1").create("sq"+strix+"C"+
            toString(counter), "Square");
          with(model.component("comp1").geom("geom1").feature("sq"+strix+"C"+
            toString(counter)));
            set("base", "center");
            set("size", new double[]{si});
            set("pos", new double[]{posx, posy});
          endwith();
          model.component("comp1").geom("geom1").feature("sq"+strix+"C"+
            toString(counter)).label("Square"+strix+"C"+toString(counter));
        } else {
          11 = 1 - 1;
          si1 = si/3;
          carpet(l1, posx, posy, si1);
        }
      }
    }
276
```

The figure below shows the resulting geometry for a level-3 carpet.



Note that the number of square geometry objects defined for the level 3 carpet is 512. The number of geometry objects for a level-N carpet is 8^N . This means that for the maximum level 5 set by the method create_carpet, the resulting geometry has 32,768 geometry objects.

Menger Sponge

The methods for the Sierpinski carpet readily generalize to 3D. However, in 3D, you need to be careful not to generate more objects than your computer can handle. The method assumes that you have created a 3D Component and that you have declared an integer scalar variable counter.

The following method, create_sponge, initiates the recursion.

```
int level = (int) model.param().evaluate("mslevel");
if (level < 1)
error("Sponge level needs to be at least 1.");
if (level > 3)
error("Sponge level needs to be at most 3.");
counter = 0;
model.component("comp1").geom("geom1").feature().clear();
model.component("comp1").geom("geom1").autoRebuild("off");
double cx0 = 0, cy0 = 0, cz0 = 0;
double si0 = 1;
sponge(level, cx0, cy0, cz0, si0);
model.component("comp1").geom("geom1").runPre("fin");
```

The method sponge, shown below, is called by the above method and recursively creates a Menger sponge.

```
int l = level;
double posx, posy, posz, si1;
String strix;
int l1;
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 3; j++) {
    for (int k = 0; k < 3; k++) {
      posx = cx+i*si-si;
      posy = cy + j * si - si;
      posz = cz+k*si-si;
      strix = toString(i)+toString(j)+toString(k);
      if ((Math.abs((i-1))+Math.abs((j-1))+Math.abs((k-1))) > 1) {
        if (1 == 1) {
          counter = counter+1;
          model.component("comp1").geom("geom1").create("blk"+strix+"C"+
            toString(counter), "Block");
          with(model.component("comp1").geom("geom1").feature("blk"+strix+"C"+
              toString(counter)));
            set("base", "center");
            set("size", new String[]{toString(si), toString(si),
              toString(si)});
            set("pos", new double[]{posx, posy, posz});
          endwith();
          model.component("comp1").geom("geom1").feature("blk"+strix+"C"+
              toString(counter)).label("Block"+strix+"C"+toString(counter));
        } else {
          11 = 1 - 1;
          si1 = si/3;
          sponge(11, posx, posy, posz, si1);
        }
     }
   }
 }
}
```

The figure below shows the resulting geometry for a level-2 sponge.



In this case, the number of geometry objects grows with the level N as 20^N and the level-2 sponge shown above has 400 block geometry objects.

Note that if you have any of the add-on products for additional CAD functionality, you can export these geometry objects on standard CAD formats. Without add-on products, you can export the meshed geometry to any of the supported mesh formats.

You can download the MPH files for these examples from:

https://www.comsol.com/model/recursion-and-recursively-defined-geometry-objects-75461

Mesh Information and Statistics

You can retrieve a variety of mesh information and statistical quantities, such as element quality, the total number of elements, the total mesh volume, and so on. The figure below shows part of an application displaying such information.

Number of elements:	79492
Tet min quality:	0.1405
Tet mean quality:	0.6532
Tet min volume:	2.815E-13
Mesh volume:	6.354E-6

Assuming that you would like to link the various mesh quantities to variables declared under the **Declarations** node, the corresponding method code would include the following lines of code:

```
a = model.component("comp1").mesh("mesh1").getNumElem();
```

b = model.component("comp1").mesh("mesh1").getMinQuality("tet");

c = model.component("comp1").mesh("mesh1").getMeanQuality("tet");

d = model.component("comp1").mesh("mesh1").getMinVolume("tet");

e = model.component("comp1").mesh("mesh1").getVolume("tet");

For more information on available mesh quantities, see the COMSOL Multiphysics Programming Reference Manual.

Accessing Higher-Order Finite Element Nodes

The extended mesh data structure contains information related to the finite element method including, for example, the placement of higher-order element nodes. The extended mesh information is contained in the class XmeshInfo and provides information about the numbering of elements, nodes, and degrees of freedom (DOFs) in the extended mesh and in the matrices returned by the Assemble feature and the solvers. For detailed information on XmeshInfo, see the *Programming Reference Manual*.

The following example method illustrates how to use the extended mesh information to plot higher-order nodes in a few important special cases. Note that general functionality for this is built-in when creating a **Mesh Plot** under **Results** while also using a **Study** as the referenced **Dataset**.

The example below covers cases with one model component, one geometry, and a subset of physics combinations. If you apply it to other cases, you will get an error message.



 $//\ Note that this method is only implemented for one component and one geometry and does not work for all physics combinations.$

```
String stdTag = model.study().uniquetag("stdfe");
model.study().create(stdTag);
model.study(stdTag).label("FE Nodes Study "+stdTag.substring(5));
model.study(stdTag).showAutoSequences("sol");
String solTag = model.sol().uniquetag("sol");
model.sol().create(solTag);
model.sol(solTag).create("st1", "StudyStep");
model.sol(solTag).create("v1", "Variables");
SolverFeature step = model.sol(solTag).feature("v1");
XmeshInfo xmi = step.xmeshInfo();
try {
  XmeshInfoNodes testnodes = xmi.nodes();
} catch (Exception e) {
  error("Cannot access finite element data. Only implemented for one geometry and
stationary studies.");
}
XmeshInfoNodes mynodes = xmi.nodes();
double[][] coords = mynodes.gCoords();
int[] coordsize = matrixSize(coords);
int sdim = 0;
if (coordsize[0] == 3) {
```

```
sdim = 3:
} else if (coordsize[0] == 2) {
  sdim = 2:
} else
  error("The geometry of the first component is not 2D or 3D.");
String mesh = "mesh"+stdTag;
model.result().dataset().create(mesh, "Mesh");
String pgTag = model.result().uniquetag("pgfe");
ResultFeature pg = model.result().create(pgTag, sdim);
model.result(pgTag).label("FE Nodes Plot "+pgTag.substring(4));
String nodes = pgTag;
model.result(nodes).create("mesh", "Mesh");
if (sdim == 3) {
  with(model.result(nodes).feature("mesh"));
    set("data", mesh);
    set("meshdomain", "volume");
  endwith();
} else {
  with(model.result(nodes).feature("mesh"));
    set("data", mesh);
    set("meshdomain", "surface");
  endwith();
}
with(model.result(nodes).feature("mesh"));
  set("elemcolor", "none");
  set("wireframecolor", "gray");
  set("elemscale", 0.999);
endwith();
with(model.result(nodes));
  set("edges", true);
  set("data", mesh);
endwith();
ResultFeature plot = pg.create("pt1", "PointData");
plot.set("pointdata", coords)
  .set("coloring", "uniform")
  .set("color", "red");
plot.run();
selectNode(pg);
```

Comments

The first few lines of the method set up a solver step object step, which is used to extract the extended mesh information. The extended mesh information, which contains information on the higher-order nodes, is extracted in the line

XmeshInfo xmi = step.xmeshInfo();

The lines

```
XmeshInfoNodes mynodes = xmi.nodes();
double[][] coords = mynodes.gCoords();
```

int[] coordsize = matrixSize(coords);

access and store the finite element node coordinates in a 2-by-coordsize (2D) or 3-by-coordsize (3D) array.

The following code segments set up a mesh dataset and an associated mesh plot.

The last section uses the low-level PointData plot type to visualize the finite element nodes. For more information on this plot type, see "Points in 3D" on page 181.

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant file for this example is:

```
    loaded_spring.mph
```

Accessing System Matrices and Vectors

You can gain low-level access to the finite element system matrices and vectors by adding nodes of the types Assemble and Input Matrix under a Study node.

The example below shows how to set up and solve a 2D electrostatics problem on the unit square [0,1]-by-[0,1]. After the original problem is solved, the load vector is modified at a user-defined coordinate. The code searches for the degree of freedom closest to the target user-defined coordinate and modifies the load vector to a user-defined value. The physical interpretation of the modified load is that of an added volume charge.

To run the example code below, first use the Model Wizard to create a blank model. Then, add a new method and paste the example code below. Finally, run

the method. You can try changing the variable values in the Initializations section at the beginning of the code and run again.



```
// Initializations
double x_load = 0.2; // Target x-coordinate for load
double y load = 0.2; // Target y-coordinate for load
double load = 1e-9; // Load, volume charge
double dist = 10.0; // Distance to (x load,y load) from degree of freedom
int index = 0; // Index of the degree of freedom closest to (x load,y load)
// Clear any previous model
clearModel(model);
// Create a new model component
model.modelNode().create("comp1");
// Create the 2D geometry
model.geom().create("geom1", 2);
model.geom("geom1").feature().create("sq1", "Square");
model.geom("geom1").run();
// Create the mesh
model.mesh().create("mesh1", "geom1");
model.mesh("mesh1").feature().create("fre1", "FreeTri");
model.mesh("mesh1").run();
// Setup the electrostatics physics problem
model.physics().create("es", "Electrostatics", "geom1");
model.physics("es").feature().create("gnd1", "Ground", 1);
model.physics("es").feature("gnd1").selection().set(new int[]{1});
model.physics("es").feature().create("sfcd1", "SurfaceChargeDensity", 1);
model.physics("es").feature("sfcd1").selection().set(new int[]{4});
// Add a varying distributed charge density along the rightmost boundary
```

```
model.physics("es").feature("sfcd1").set("rhogs", "1e-9*v");
// The following two lines using the ccn1 feature are only needed in version 6.2
and earlier versions
//model.component("comp1").physics("es").feature("ccn1").set("epsilonr mat".
"userdef");
//model.component("comp1").physics("es").feature("ccn1").set("epsilonr", "1");
// Change to first-order shape functions, to keep things simple
model.component("comp1").physics("es").prop("ShapeProperty").
set("order electricpotential", 1);
// Create and run the study.
model.study().create("std1");
model.study("std1").feature().create("stat1", "Stationary");
model.study("std1").run();
// Create a 2D plot group with a surface plot for the original problem
model.result().create("pg1", 2);
model.result("pg1").set("data", "dset1");
model.result("pg1").feature().create("surf1", "Surface");
selectNode(model.result("pq1")); // Set focus on the plot node
// Create a reusable solver feature variable
SolverFeature solft;
model.study().create("std2"); // Create a Study 2 node
model.sol().create("sol2"); // Create a dataset Solution 2
// Create a Solver configurations node under Study 2
model.sol("sol2").study("std2");
model.sol("sol2").create("st1", "StudyStep"); // Create a Compile Equations node
solft = model.sol("sol2").feature("st1"); // Assign solver step to variable
solft.set("study", "std2");
model.sol("sol2").create("v1", "Variables"); // Create a Dependent Variables node
solft = model.sol("sol2").feature("v1");
model.sol("sol2").attach("std2");
model.sol("sol2").create("a1", "Assemble"); // Add an Assemble node
solft = model.sol("sol2").feature("a1");
// Now define which system matrices should be output (Noneliminated Output)
// L=Load vector, K=Stiffness matrix, M=Constraint vector, N=Constraint Jacobian
// For more information see the Programming Reference Manual
solft.set("L", "on");
solft.set("K", "on");
solft.set("M", "on");
solft.set("N", "on");
// Create a Stationary Solver 2 node: Study 2 > Solver Configurations > Solution 2
model.sol("sol2").create("s2", "Stationary");
// Create an Input Matrix node under Stationary Solver 2
solft = model.sol("sol2").feature("s2").create("im1", "InputMatrix");
// Define which system matrices should be input
```

```
solft.set("L", "on");
solft.set("K", "on");
solft.set("M", "on");
solft.set("N", "on");
// Find the degree of freedom coordinate closest to the target coordinate
solft = model.sol("sol2").feature("v1");
XmeshInfo xmi = solft.xmeshInfo();
XmeshInfoDofs mydofs = xmi.dofs();
double[][] coords = mydofs.gCoords();
int[] coordsize = matrixSize(coords);
double new dist = dist;
for (int k = 0; k < coordsize[1]; k++) {</pre>
  new dist = Math.sqrt((coords[0][k]-x load)*(coords[0][k]-x load)+
(coords[1][k]-y load)*(coords[1][k]-y load));
  if (new dist < dist) {
    index = k;
    dist = new_dist;
  }
}
// Run the solver sequence up to and including the Assemble node
model.sol("sol2").runFromTo("st1", "a1");
// Extract system matrices and vectors
solft = model.sol("sol2").feature("a1");
// K
int KM = solft.getM("K");
int KN = solft.getN("K");
int KNnz = solft.getNnz("K");
int[] Ki = solft.getSparseMatrixRow("K");
int[] Kj = solft.getSparseMatrixCol("K");
double[] Kv = solft.getSparseMatrixVal("K");
// For more information, see the Programming Reference Manual
// L
double[] Lv = solft.getVector("L");
// N
int NM = solft.getM("N");
int NN = solft.getN("N");
int NNnz = solft.getNnz("N");
int[] Ni = solft.getSparseMatrixRow("N");
int[] Nj = solft.getSparseMatrixCol("N");
double[] Nv = solft.getSparseMatrixVal("N");
// M
double[] Mv = solft.getVector("M");
// Modify the load
Lv[index] = load;
// Put the system matrices and vectors back in again
solft = model.sol("sol2").feature("s2").feature("im1");
```

```
// K
solft.createSparseMatrix("K", KM, KN, KNnz, true);
solft.addSparseMatrixVal("K", Ki, Kj, Kv);
// L
solft.createVector("L", Lv.length, true);
solft.setVector("L", Lv);
// N
solft.createSparseMatrix("N", NM, NN, NNnz, true);
solft.addSparseMatrixVal("N", Ni, Nj, Nv);
// M
solft.createVector("M", Mv.length, true);
solft.setVector("M", Mv);
// Solve Stationary Solver 2 with the modified system
model.sol("sol2").runFromTo("s2", "s2");
// Plot the results
model.result().create("pg2", "PlotGroup2D");
model.result("pg2").set("data", "dset2");
model.result("pg2").create("surf1", "Surface");
// Plot electric potential and original mesh overlayed with no smoothing
model.result("pg2").feature("surf1").set("resolution", "norefine");
model.result("pg2").feature("surf1").set("smooth", "none");
model.result("pg2").create("surf2", "Surface");
model.result("pg2").feature("surf2").set("resolution", "norefine");
model.result("pq2").feature("surf2").set("coloring", "uniform");
model.result("pg2").feature("surf2").set("color", "gray");
model.result("pg2").feature("surf2"). set("wireframe", true);
model.result("pg2").run();
selectNode(model.result("pg2")); // Set focus on the plot node
```

Comments

In the previous example, "Accessing Higher-Order Finite Element Nodes" on page 280, the XmeshInfoNodes methods are used to access finite element nodes that have the same length as the number of finite element nodes. In this example, the XmeshInfoDofs methods are used to access the degrees of freedom vector, which has the same length as the load vector.

Note that only the load vector is modified. The other matrices and vectors are merely extracted and then put back into the system again.

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771 The relevant file for this example is:

accessing_system_matrices.mph

Data Validation

In the **Settings** window for input fields, the section for **Data Validation** has a subsection **Numerical validation** with settings that allow you to validate user inputs with respect to values. The figure below shows an example of settings for entering a radius.

 Data Validation 	1	
Unit dimension chec	k None +	
- Numerical validation		
Filter:	Double 🔹	
V Lower bound		
Comparison type:	Greater than or equal 🔹	
Value:	7.5	
Vpper bound -		
Comparison type:	Less than or equal 🔹	
Value:	23	
Error message:		
The radius must be greater than or equal to eval(MINVALUE) and less than or equal to eval(MAXVALUE).		

The Filter options, the Lower bound and Upper bound settings are only visible if the selection in the Unit dimension check combo box is set to one of None, Append unit to number, or Append unit from unit set.

The **Value** input fields for **Lower bound** and **Upper bound** allow you to use global parameters or scalar declaration variables. Using scalar declaration variables in this field supports the same type of data conversions as when using declaration names as a method input argument in a command sequence. This means that scalar **Integer** and **Double** declaration variable will always work and scalar **String** declarations will work if they can be converted to a **Double**.

Using declaration variable names or global parameters in the **Value** input fields can only be used when **Unit dimension check** is set to **Append unit to number** or **Append unit from unit set**. In that case, the bounds are checked based on the numerical value entered by the user in the input field. In a **Value** input field, you cannot use expressions in terms of declared variables but only a single declaration name. If a parameter exists with the same name as a declaration variable, then the value of the global parameter will be used.
The scope of declaration names used in a **Value** field is limited to the parent form of the input field.

Note that the bounds are updated dynamically as the value of the parameters or declaration variable is changed. This means that several input fields, not actively changed by the user, can fail numerical validation at once caused by a change in another input field, which causes the value of a parameter used in a validation bound to change.

The **Error message** text field allows you to write a customized error message to be shown when the input values are out of bounds. The error message text can contain eval(variable), which is also allowed in the title for plot groups in the Model Builder. In addition, the **Tooltip** of the input field allows use of eval(variable). The variable can be the name of any global parameter or scalar declaration. If a parameter and a declaration with the same name exists, then the parameter is used. In addition to parameter and declaration names the special strings MINVALUE and MAXVALUE can be passed to eval and will return the value entered as lower bound or upper bound.

When using eval for a parameter, it will return the evaluated value of the parameter using the same unit that is used to enter the bounds in the **Value** input field. For a scalar declaration the string value of the declaration is returned which will be true or false for a boolean declaration.

Access to Old Value and New Value in Event Methods

You can create a method for an **On data change** event directly from the **Events** section of an input field, as shown in the figure below.



In such a case, the method will automatically get a scalar String input argument with the name newValue, as shown in the figure below.



The variable newValue will get the new value entered in the input field by the user of the app.

In addition to the new value, you may need to access to the old value entered in the input field, for example, to be able to restore it if your custom data validation fails. For this purpose, you can use the event object and its associated methods which are accessible from methods that are called from events. The event object has the following methods:

```
event.getOldValue()
event.getNewValue()
```

which return the old and new value, respectively, of the source associated with the form object. The methods return an object called Value which is similar to the DataSource object returned when you call:

```
app.declaration(<declarationName>)
```

A DataSource object has the methods getBoolean, getBooleanArray, getBooleanMatrix, getDouble, and so on to allow accessing the value stored in the DataSource object no matter what type it has. It also has the corresponding set methods to allow setting a value. For more information, see "Data Source" on page 109.

The Value object is a read-only version of the DataSource object with only the get methods.

Access to Forms and Form Objects in Event Methods

For methods that are called from events, you can access the properties formTag and formObjectTag for the form object that triggered the event. This way, you can access the form object from the event method using the syntax:

```
app.form(<formTag>).formObject(<formObjectTag>)
```

To make it more convenient to access the user control that triggered the event there is also a method

event.getUserControl()

that returns the form object, menu item, or ribbon item that triggered the event. The event object is additionally available for methods called from command sequences in, for example, menu items and ribbon items. The interface IPropertyAccess which has methods for getting and setting values on all user interface controls such as form objects, menu items and so on can then be returned by the getUserControl method.

DATA VALIDATION FROM DIALOGS

A form that is shown as a dialog can function in two different ways. The first way, which is the default option, works on a copy of the data and stores changes when the user clicks **OK** in the dialog. The second way is enabled by changing the **Store changes** value of the form, used as a dialog, from **On request** to **Immediately**, as shown in the figure below.



> Grid Layout for Contained Form Objects

Changes performed in the dialog will then be stored to the associated source variable or parameter immediately and it will work like a regular form in the main application window.

To accomplish storing of the values when the dialog is set to work on a copy, corresponding to the first option above, and store the values on request, the **Settings** window for a button object has a checkbox **Store changes** in the section **Dialog Actions**, as shown in the figure below. When this checkbox is selected, the values in the dialog are stored from the copy used by the dialog to the actual values after the command sequence specified for the button has been run. Part of this command sequence can include a method that performs validation of the data that the user has entered in the dialog. The figure below shows an example of this scenario, with a method validation, for an **OK** button used in a dialog.



However, this validation approach will not work. This is due to the fact that when the dialog is set to **Store changes** on request, the values have not yet been stored when the validation method runs since dialog actions such as **Store changes** are performed after the command sequence for the button has been run. This means that the validation code cannot access the new values to perform a validation on them.

To remedy this, there is an access method that can be used together with form objects such that the value currently entered in them can be used for data validation before they have been stored in their associated source. The syntax for calling this access method is:

```
app.form(<formTag>).formObject(<formObjectTag>).getValue();
```

This call will return a Value object, the same type of object described earlier and used for calls to event.getOldValue() and event.getNewValue(). With this type

of access to the current value in the form object, a data validation can be performed. The following form object types support the getValue method.

- Input field
- Toggle Button
- Checkbox
- Combo box
- File import
- Array input
- Radio button
- Selection input
- Text
- List box
- Table
- Slider
- Knob
- Gauge

If the user has clicked **OK** in a dialog and the data validation fails, you typically want to show an error dialog and then leave the dialog open to allow the user to correct the input fields that failed validation. This not possible to achieve using a direct approach since if the **OK** button is set to have **Store changes** as a **Dialog Action** the storing of the data will always be performed after the command sequence, defined in the **Settings** window of the **OK** button, has been run.

To remedy this, there is a method for programmatically store changes:

```
storeChanges(String name)
```

which will store the changes for a given form, used as a dialog.

Note that you can call:

```
closeDialog(<dialogTag>)
```

to close a specific dialog.

When using the technique described above, you can clear the **Store changes** and **Close dialog** checkboxes, in the **Dialog Actions** section for the **OK** button and instead call storeChanges and closeDialog as part of the flow in the validation method if the validation passes.

Having a way to programmatically store changes also helps with the case where the dialog contains a button that also performs something when closing the dialog, for example, a **Compute** button. A validation may then be followed by, for example, a compute method. In order to get this compute method to work on the new values entered in the dialog, storeChanges can be called after the data validation has been performed but before the compute method is called.

Example of Data Validation in Forms and Dialogs

This example illustrates the use of data validation in a form and a dialog. This app is very simple and does not fulfill any other function than to demonstrate important aspects of data validation. The app consists of a form with one input field that expects the user to type the string open dialog, as shown in the figure below



If the user types a different string, then an alert window with a message Unknown command is shown:

•	Untitled.mph — 🗆	×
	Command: dialog	
	Alert × Abo	ut
-	Unknown command.	
	ОК	

and the string value for the **Command** is reset to the default Type open dialog.

If the correct string open dialog is entered, then a dialog opens, as shown in the figures below.

•	Untitled.mph — 🗆 🗙	Untitled.mph - 🗆 🗙
	Command: open dialog About	Command: open dialog
		Number x (1-10): 3 Number y (y=x): 3 OK 0

The dialog expects an integer between 1 and 10, in the first input field. In addition, it expects the value in the second input field to be identical to the value in first input field. If all these criteria is fulfilled, no error message is shown and the app starts from the beginning showing the string Type open dialog in the main form. If the user enters different values, as shown in the figure below:

•	Untitled.mph —		×
	Command: open dialog		
	E Form 2		×
	Number x (1-10):	2	
	Number y (y=x):	3	
		01/	

then an error message is shown with the message The value of y must be equal to x.



When the user now closes the Error message dialog, the user gets a new opportunity to enter matching values in the dialog.

The data validation functionality in this demonstration app is implemented using an On data change event for the **Command** input field in the main form form1, as shown in the figure below.



The code in method3 shows the user of event.getNewValue and event.getOldValue, as shown below:

```
String value = event.getNewValue().getString();
String oldValue = event.getOldValue().getString();
if (value.toLowerCase().equals("open dialog")) {
    dialog("/form2");
} else {
    alert("Unknown command.");
    command = oldValue;
}
```

The dialog form2 has Store changes set to On request, as shown in the figure below.

Setti Form	ngs		▼□ ‡×
Name:	form2		Ē
Title:	Form 2		
lcon:	Defau	lt	• + 🖻
Shov	v in Mod	lel Builder	
 Size 			
Initial siz	e: Au	utomatic	•
✓ Mar	gins		
Horizont	al: 20		
Vertical:	20		
✓ Dial	og Sett	ings	
Store ch	anges:	On request	•
 Resiz Verti 	able cally scro	ollable	

In the dialog, the value of the variable x is validated in the **On data Change** event of the input field with label **Number x (1-10)** and also when clicking the **OK** button. The y value is only validated when clicking the **OK** button. The data validations are implemented using the form object access method getValue. The **OK** button makes use of the storeChanges method. The figure below shows the method run for the **On data change** event for the input field with label **Number x (I-I0)**.

Preview form1 × form2 ×	₹ method2 ×	□ ₹	Settings ×	Editor Tools ×	× 0 #
	<pre>1 int valueX = app.form("form2").formObject("inputfield1").getValue().getInt(); 2 = if (valueX < 1 valueX > 10) {</pre>		Input Field		
Number x (1-10): 0	3 error("The value of x must be greater than or equal to 1 and less than or equal to 10.);	Name: inp	utfield1	
Number y (y=x): 0	4 [}		✓ Editable		
ОК			Tooltip:		
4					01.00
			 Source 		⊕ + 8
			✓	arations	
			> abc S	tring	
			✓ 123 In	nteger m	
			1	E X	
				-,	
			📑 Use as So	ource 🖅 Edit Node	
			Selected sour	ce:	
			123 Integ	er=x	
			Initial value:	From data source	•
			Value:	0	
			> Data Va	lidation	
			> Number	r Format	
			> Position	and Size	
			> Appeara	ance	
			✓ Events		
			On data char	ige: method2	• • •
			On focus gai	ned: None	• h+•

The corresponding code in method2 is as follows:

```
int valueX =
app.form("form2").formObject("inputfield1").getValue().getInt();
if (valueX < 1 || valueX > 10) {
    error("The value of x must be greater than or equal to 1 and less than or
equal to 10.");
}
```

The **OK** button calls method1 for the **On data Change** event, as shown in the figures below.

Q Preview form2 x form1 × □ ⇒	Settings	→ □ # ×
•	Button	
Number x (1-10): 0	Name:	button1
Number y (y=x): 0	Text:	ОК
	lcon:	None +
	Size:	Small 👻
	Style:	Raised 🔹
	Tooltip:	
	Keyboard shortcut:	
	 Choose Comm 	nands to Run
	 > ← Forms > GUI Comma > ≡ Declarations > ← ← Methods ↓ ← method ↓ ← method 	inds 5 1 2
	🖅 Edit Node 🕨 F	Run 🕅 Plot 🦻 Set Value 🗌 Show
	Show as Dialog	↑ Import File Enable Disable
	** Command	Icon Arguments
	method1	4
	1 + 🖶 🐙 🖽	 ✓
	 Dialog Actions 	5
	Close dialog	
	Store changes	
<pre>Preview</pre>	<pre>ect("inputfield1' ect("inputfield2' to x.");</pre>	<pre>').getValue().getInt(); ').getValue().getInt();</pre>

The corresponding code in method1 is as follows:

```
int valueX =
app.form("form2").formObject("inputfield1").getValue().getInt();
```

```
int valueY =
app.form("form2").formObject("inputfield2").getValue().getInt();
method2();
if (valueX != valueY) {
    error("The value of y must be equal to x.");
} else {
    storeChanges("/form2");
    closeDialog("/form2");
    command = "Type open dialog";
}
```

Using Selections in Add-ins

In order to use selections in the Model Builder from an add-in, you leave the source settings empty when using a selection input form object at the time the add-in is created. Then you use a method to create an explicit selection in the current model and link it to the selection input object of the add-in.

The figure below shows a simple add-in **Boundary Selections** used to demonstrate this functionality. It contains a form with a selection input object and a button. When the add-in is in focus, the user can click on one or more boundaries in the

graphics window to create the selection. Clicking the button triggers an Alert with a dialog where the selected boundaries are listed.



The figure below shows the form of the add-in as it appears when in use in a model:



When the add-in **Settings** form is added to a model, an **On load** event is triggered that runs a method createSelection. The **Settings** form is shown in the figure below:

Setti Form	ngs 👻 🗖 🖡 🗙
Name:	form1
Title:	Boundary Selection
lcon:	Default 🔹 🕂 🕞
🖌 Sho	v in Model Builder
~ Size	
Initial si:	e: Automatic 👻
> Ma	gins
> Dial	og Settings
> Grid	Layout for Contained Form Objects
> App	earance
~ Eve	nts
On load	createSelection 🔹 🗄 + 🔹
On close	: None Ē1 + ▼

The code for the method createSelection is listed below:

```
// Create explicit selection in the model.
if (model.selection().index(selectionTag) < 0) {
    SelectionFeature selection = model.selection().create(selectionTag,
"Explicit");
    selection.geom(2);
    selectioninput1.set("source", selection);
}
// Activate the selection whenever the settings form is selected
selectioninput1.set("active", true);
```

The variable selectionTag stores a unique identifier for the Model Builder **Explicit** selection feature. This string needs to be different enough not to accidentally

collide with the user's selection features, used for other purposes, in the Model Builder. The figure below shows the declaration of this String variable.

Settings String		- □ ‡ ×
List of Variables		
Name	Initial value	Description
selectionTag	selection_addin_prefix_sel1	A unique tag for this add-in
↑ 1 == = -	✓ □ III	

In addition, a shortcut selectioninput1 is used for the selection input object, as shown in the figure below.

Settings Shortcuts		✓ □ # ×
List of Shortcuts		
* Name	Target	Description
selectioninput1	form1/selectioninput1	Shortcut to Selection input
↑↓≒™		

The figure below shows the **Settings** window for the **Selection Input** form object selectioninput1. Note the empty selection of the **Source** and **Graphics to Use When Active**. These settings are set by the method createSelection.

form1 × Q Preview	□ ₹	Settings Selection Input
Display Selection Boundaries		Selection Input Name: selectioninput

form1 X Preview		Settings Button Name: Text: Icon: Size: Style: Tooltip: Keyboard shortcut: Choose Comm Choose Comm Choose Comm Choose Comm Choose Comm Comma C	button1
		✓ ✓	election election Run I Import File Impo

The figure below shows the **Settings** window for the **Button** form object button1.

When this button is clicked, a method displaySelection is run. The code for this method is shown below.

```
SelectionFeature selection = model.selection(selectionTag);
alert("Selection boundaries are: "+concat(", ",
toString(selection.entities())));
```

For your own add-in, you can replace this code with any number of actions that accepts an explicit selection as an input. For example, you can add the following lines of code to the end of the method displaySelection in order to add a variable a, local to this explicit selection, having the value 5.

```
model.component("comp1").variable().create("var1");
model.component("comp1").variable("var1").selection().geom("geom1", 2);
model.component("comp1").variable("var1").set("a", "5");
model.component("comp1").variable("var1").selection().named(selectionTag);
```

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771

The relevant files for this example are:

- selection_input_add_in_source.mph
- busbar_with_selection_add-in_demo.mph

For more information on creating and using add-ins, see the *Introduction to Application Builder*.

Using Built-In Methods from an External Java Library

When developing an external Java® library to be used in the Application Builder, it is possible to call the built-in methods from the external library. In order to do so, the com.comsol.api JAR-file needs to be added to the project build path, and the classes that call the methods need to extend the ApplicationLanguageBase class. The following steps explain how to create a simple example JAR library when using the Eclipse® integrated development environment:

- Start Eclipse®.
- 2 Create a new Java® project. Enter JavaLibDemo as the project name and click Next.
- **3** Go to the **Libraries** tab and click **Add External JARs**. Add the JAR-file com.comsol.api_1.0.0.jar from the plugins directory under the COMSOL Multiphysics installation directory; for example

C:\Program Files\COMSOL\COMSOL63\Multiphysics\plugins

- 4 Click Finish.
- 5 Right-click the src folder in your Eclipse project and select New > Package. Enter demo as the package name and click Finish.
- 6 Right-click the demo package in your **Eclipse** project and select **New** > **Class**. Enter Hello as the class name.
- 7 Click on Superclass > Browse and select ApplicationLanguageBase. Click OK and Finish.
- 8 In the editor window for the Hello class, add the following method and save the file:

```
public static void hello() {
   alert("Hello!");
}
```

- 9 Right-click the JavaLibDemo project and select Export > Java > JAR file. Select the export destination JAR-file and click Finish.
- **10** Start COMSOL Multiphysics and create a **Blank Model**.

- II In the Application Builder, under the application tree, right-click **Libraries** and select **External Java Library**. In the **Settings** window, click **Browse** and select the JAR-file previously exported from Eclipse[®].
- ¹²In the Application Builder, add a form, button, and method for the button.
- I3 In the Method Editor for the method, add the following code: Hello.hello();
- ¹⁴Click **Test Application** and click the button in the application to verify that the alert method is invoked.

Measuring the Java Heap Space Memory

Software components that are based on Java®, such as certain parts of COMSOL Multiphysics, are predefined to use only a limited amount of memory. This limit, the Java® heap space, is specified during startup. Note that the Java® heap space only affects certain parts of the software and not, for example, meshing or solvers. By default, COMSOL Multiphysics allocates 2 GB of Java® heap space memory. See the *Reference Manual* for information about increasing the available heap space memory.

In a method, you can measure the amount of heap space memory currently in use. For example, while debugging, you can add the following code:

```
Runtime runtime = java.lang.Runtime.getRuntime();
runtime.gc();
debugLog("Used memory (MB): "+(runtime.totalMemory()-runtime.freeMemory())/
(1024*1024));
```

The first line adds a request to run the Java® garbage collector.

In order to monitor the memory usage in an application, you can replace ${\tt debugLog}$ with message.

Time-Limited and Hardware-Locked Applications

By writing a few lines of code you can make your application expire after a set date and lock the application to specific hardware.

The example MPH file used in this section is available in the Application Gallery at https://www.comsol.com/model/time-limited-and-hardware-locked-application-70151.

PASSWORD PROTECTION

The settings of an application can in principle be read from the file system by a user, including method code. By making your application password protected for editing, the method code will no longer be readable. This setting is available from the root node in either the model tree or the application tree, as shown in the figure below.

 Protection 		Protect Edit with Password	>
Editing not protected	Set Password	S rotett zan min assiona	-
Running not protected	Set Password	No current password	
 Used Products COMSOL Multiphysics 		Retype new password: Note: Lost passwords cannot be recove	recovered
 Unit System 		OK Cano	el

Before implementing a time limit or hardware lock, as described below, make sure your application is password protected. Password protection for running the application is not required for this purpose.

TIME-LIMITED APPLICATION

To have an application expire after a specific date, create a method as follows:

```
java.text.SimpleDateFormat f = new java.text.SimpleDateFormat("yyyy-MM-dd");
//java.text.SimpleDateFormat f = new java.text.SimpleDateFormat("MM/dd/yyyy");
ok = false;
try {
  java.util.Date d = f.parse(timeoutDate);
  long currentTime = timeStamp();
  long timeoutTime = d.getTime()+24*60*60*1000; // To allow running until the end
        of the day
  if (currentTime < timeoutTime) {</pre>
    ok = true;
  }
}
catch (java.text.ParseException e) {
  debugLog("Failed to parse timeout date "+timeoutDate);
  debugLog(e.getMessage());
}
```

In this method, you need to decide on a date format. Two format examples are shown and you can uncomment the line corresponding to the format you would like to use. For more details on available formats, see the Java® documentation for

SimpleDateFormat. This method has one string input argument, timoutDate, and one Boolean output argument, ok, as shown below.



The expiration date is defined as a string variable, trial_date, in **Declarations** > **String**, as shown below.

Settings String		- □ # X
List of Variables		
Name	Initial value	Description
solution_state	nosolution	Solution state
trial_date	'2018-11-9'	Trial expiration date
↑ ↓ ≔ ≔	- 🖳 🔣	

HARDWARE-LOCKED APPLICATION

To lock an application to the MAC address of a specific network card on a computer, create a method as follows:

```
ok = false;
try {
  java.util.List < java.net.NetworkInterface > nis =
  java.util.Collections.list(java.net.NetworkInterface.getNetworkInterfaces());
  for (java.net.NetworkInterface ni : nis) {
    StringBuilder macString = new StringBuilder();
    byte[] macBytes = ni.getHardwareAddress();
    if (macBytes != null && macBytes.length > 0) {
      for (byte b : macBytes) {
         if (macString.length() > 0) {
            macString.append(":");
            }
```

```
macString.append(String.format("%02x", b));
}
if (contains(allowedAddresses, macString.toString())) {
    ok = true;
    break;
    }
}
}
catch (java.net.SocketException e) {}
```

In order to check the MAC address when running an application, you need to enable **Allow access to network sockets** under **Security** in **Preferences**. However, for a compiled application, no security changes are needed.

This method has one array 1D string input argument, allowedAddresses, and one Boolean output argument, ok, as shown below.



The MAC address is defined as a string array mac_addresses in **Declarations** > **Array ID String**, as shown below.

Settings Array 1D String				- □ # ×
List of Variables				
* Name	Initial values	New element value	Description	
mac_addresses	{'34:17:eb:d2:5e:14'}		MAC address (needs to be lowercase)	

Note that you can provide a list of MAC addresses to allow use on a computer with multiple network cards or multiple computers.

CHECKING FOR ALLOWED DATE AND HARDWARE

To check for both the MAC address and the date, create a method check_allowed_to_run as follows:

```
if (!check_mac_address(mac_addresses)) {
```

```
alert("You are not allowed to run this application on this computer.",
    "COMSOL");
exit();
}
if (!check_date(trial_date)) {
    alert("The trial for this application has expired "+trial_date, "COMSOL");
    exit();
}
```

The figure below shows this method in the Method Editor.



You can call this type of method at startup of the application, for example, as an On load event for the main form of the application. In the Tuning Fork example application, available in the Application Library of COMSOL Multiphysics, there is a method p_init_application that is run as an On load event for the main form. In this case, the method p_init_application can be edited as follows:

```
check_allowed_to_run();
if (model.sol("sol1").isEmpty()) {
   solution_state = "nosolution";
}
else {
   solution_state = "solutionexists";
}
zoomExtents("graphics1");
```

Notice the call to the method check_allowed_to_run in the first line. The figure below shows this method in the Method Editor.

The method p_init_application is then called as an **On load** event. This is specified in the **Settings** window of the main form, as shown in the figure below.



Get and Set of 3D Camera Parameters

The following method code demonstrates how to programmatically retrieve and log the complete set of camera settings for the **View** with tag view1 in the **Component** with tag comp1. It captures parameters for: zoom angle, position, target, up vector, rotation point, view offset, manual grid flag, and axis spacings. This information can be used in order to at a later point restore the exact camera state in your application.

```
// Cache the camera feature
ViewFeature view1Camera = model.component("comp1").view("view1").camera();
// Retrieve zoom angle
zoomAngle = view1Camera.getDouble("zoomanglefull");
// Retrieve position vector
position = new double[3];
for (int i = 0; i < 3; i++) {
    position[i] = view1Camera.getDouble("position", i);
}
// Retrieve target vector
target = new double[3];
for (int i = 0; i < 3; i++) {
    target[i] = view1Camera.getDouble("target", i);
```

```
}
// Retrieve up vector
up = new double[3];
for (int i = 0; i < 3; i++) {
 up[i] = view1Camera.getDouble("up", i);
}
// Retrieve rotationPoint vector
rotationPoint = new double[3];
for (int i = 0; i < 3; i++) {
 rotationPoint[i] = view1Camera.getDouble("rotationpoint", i);
}
// Retrieve viewOffset
viewOffset = new double[2];
for (int i = 0; i < 2; i++) {
 viewOffset[i] = view1Camera.getDouble("viewoffset", i);
}
// Retrieve manualGrid flag
manualGrid = view1Camera.getBoolean("manualgrid");
// Retrieve axis spacings
xSpacing = view1Camera.getDouble("xspacing");
ySpacing = view1Camera.getDouble("yspacing");
zSpacing = view1Camera.getDouble("zspacing");
```

In this example, the variables are stored globally under the **Declaration** node, shown in the figures below.

Settings Double		* □ # X	Settings Array 1D Doub	ble		Ŧ
List of Variabl	es		List of Variable	25		
* Name	Initial value	Description	Name	Initial values	New element vi	Description
zoomAngle	10.039775848388672	Zoom angle	position	{0.0524999983	0.0	Position vector
xSpacing	0.0	X grid spacing	target	{0.0524999983	0.0	Target vector
ySpacing	0.0	Y grid spacing	up	{0,1,4.3711388	0.0	Up vector
zSpacing	0.0	Z grid spacing	rotationPoint	{0.0524999983	0.0	Rotation point
			viewOffset	{-0.040149394	0.0	View offset
Settings Boolean		~ □ # ×				
List of Variabl	es					
** Name	Initial value	Description				
manualGrid	false	Grid on/off				
* 1 :::* ::						

The following method code shows how to restore the camera settings.

```
// Cache the camera feature
ViewFeature view1Camera = model.component("comp1").view("view1").camera();
// Restore zoom angle
view1Camera.set("zoomanglefull", Double.toString(zoomAngle));
// Restore position vector
for (int i = 0; i < position.length; i++) {</pre>
  view1Camera.setIndex("position", Double.toString(position[i]), i);
}
// Restore target vector
for (int i = 0; i < target.length; i++) {</pre>
  view1Camera.setIndex("target", Double.toString(target[i]), i);
}
// Restore up vector
for (int i = 0; i < up.length; i++) {
  view1Camera.setIndex("up", Double.toString(up[i]), i);
}
// Restore rotation point vector
for (int i = 0; i < rotationPoint.length; i++) {</pre>
  view1Camera.setIndex("rotationpoint", Double.toString(rotationPoint[i]), i);
}
// Restore view offset
for (int i = 0; i < viewOffset.length; i++) {</pre>
  view1Camera.setIndex("viewoffset", Double.toString(viewOffset[i]), i);
}
// Restore manual grid flag
view1Camera.set("manualgrid", manualGrid);
// Restore axis spacings
view1Camera.set("xspacing", Double.toString(xSpacing));
view1Camera.set("yspacing", Double.toString(ySpacing));
view1Camera.set("zspacing", Double.toString(zSpacing));
selectNode(model.component("comp1").view("view1"));
```

In the last line of code, the call to **selectNode** ensures that the view is updated automatically.

The figure below shows these two methods, named StoreCamera and RestoreCamera respectively, as **Method Call** nodes in the Model Builder.



These two methods can be added to any 3D model in order to store and restore camera settings.

This example is part of a collection available for download:

https://www.comsol.com/model/application-programming-guide-examples-140771 The relevant file for this example is:

• saving_camera_settings.mph

316 |

ID array 14, 44, 164 2D array 14, 44, 164

A accessing 284 add-in 27, 176, 300 selections 300 Add-in Libraries 27 alert 1.38, 145 anisotropic diffusion coefficient 47 Annotation Data plot type 189 AppEventHandlerList methods 116 Application Builder 76 Application Builder Reference Manual 1.52 application example tubular reactor 136 application methods 26 application object 7, 37, 76, 118 app variable 78 classes 78 application tree 76 array 14 methods 164 array ID object 109 array 2D object 109 array input object 87 Arrow Data plot type 187, 188 assignments 9 auto complete 21 automatic solver sequence 62 auxiliary sweep 255 average computing 258, 259 axisymmetric property 40

backslash 12 backspace 12 basic data type 41 big endian 214 binary file processing 214 reading 212 writing 211 Blank Model 41 boolean data type 8, 41 Boolean variable 8 conversion 162 bound 288 boundary condition 55 built-in method library 118 button object 88 **c** C libraries external 151 card stack object 88

в

carriage return 12 catch 20 C-code linking 151 char data type 8 character data type 8 character stream 206 characters special 205 Chatbot window 32 checkbox object 89 choice list 79, 111, 142

methods 111 object 109 classes application object 78 code completion 21 code generation 21 collection methods 173 color 81 of user interface component 80 combo box object 89 Compact History 71 Compile Equations node 60 computation time 161 last 160 Compute 60 COMSOL Desktop 67 COMSOL Help Desk 73 COMSOL Multiphysics 7, 37 COMSOL Multiphysics Programming Reference Manual 280 confirm 11, 139, 145 contour plot 63 contour plot data 208 control flow statements 18 conversion between data types 9 methods 162 coordinate information 222, 223 Copy as Code to Clipboard 41 creating feature node 52, 56 model object 39, 67 CSV-file 190, 197 curve interpolation 208 parameterization 215 cut point dataset 47, 120 data display object 90

Data Source class 79, 109 data types primitive 8 data validation 288 dialog 291, 294 new value 290 old value 290 Data Viewer window 29, 250, 254 dataset 64, 179 DataSource object 290 date and time methods 159 debug methods 151 Debug Log window 151 declaration 288 type 74 Declaration methods 116 Declarations 14, 16, 77 deformation plot 47 degrees of freedom 280 Dependent Variables node 60 Derived Values 241 description 20 parameter 20, 48 variable 20 Developer tab 178 dialog 137, 138 data validation 291, 294 Dialog Actions 292 diffusion coefficient anisotropic 47 dimension spatial 40 disable form object 80, 85, 86, 142 Display Name for choice list 142 double 9 data type 8, 41

318

D

variable conversion 162 double guotation mark 11 E edge parameterization 215 Editor Tools window 49 eigenfrequency data 273 Electric Currents 66 element order 280 size 46, 53 elementary math functions 17 email class 1.3.3 methods 1.3.3 preferences 135 email attachment export 133 report 133 table 133 embedded model 67 enable form object 78, 80, 85, 86, 142 endian 214 equation object 91 error dialog 20 Error message 289 eval 289 Evaluation Group 241, 244, 259 events 113, 291 example code 118, 178 Excel® file 65, 127, 190, 195, 198 exception handling 20 exit 149 application 150 expiration date for application 307 explicit selection 300 export email attachment 133

external C libraries 151 external Java Ilibrary 306 face parameterization 217 feature node creating 52, 56 removing 53, 56 file methods 123, 124 name 127 open 130 reading 190 writing to 190 File Declaration 193 file import 193 file import object 91 file open system method 130 file scheme syntax 124, 193 Filter 288 finite element accessing 280 system matrix 283 vector 283 floating point number 8 for loop 19, 86, 117 form class 79, 84 declarations 17 list methods 116 form collection 92 form feed 12 form method 27 form object 91 class 79, 85 list methods 116 types 87 FormList methods 116

F

319

FormObjectList methods 116 Fully Coupled node 62 G general properties 80 generating code 21 Geometry node 52 geometry object 52, 53 names 223 get 41 global method 27 global parameter 48 global quantity computing 260 graphics object 93, 138 view 143, 146 grid dataset 178 GUI command methods 149 GUI related methods 137 H hardware lock 307 heap space 307 Heat Transfer in Solids 54, 66 higher-order element nodes 280 history model 71 HTMI report 146 hyperlink object 94 I. if-else statement 18 image object 94 import file 193 information card stack object 95 information node 161 inherit color 81 input field 288 input field object 96

integer data type 8, 41 variable conversion 162 integral computing 260 interpolation curve 208, 214, 215 interpolation function 214 Introduction to Application Builder 7, 21, 25, 26, 76, 112, 124, 161, 190 Introduction to COMSOL Multiphysics 25 37 isosurface 179 item class 79 list methods 116 menu 108 object 108 toolbar 108 ItemList methods 116 iterative solver 60 Iterator class and method 56 jagged arrays 14 lava character stream 206 classes for read/write 205 Documentation, model object class structure 73 external library 306 heap space 307 math library 17 programming language 7, 8, 118 syntax 9 unary and binary operators 9 Java Shell window 28, 45, 49, 270 K keyboard shortcut Ctrl+Space 21 knob object 103

legend 46, 48 L

Т

length of parametric sweep 268 license check out 151 method 120 lighting 185 Line Data plot type 182 line object 98 list box object 98 literals 9 little endian 214 load vector 287 loading model 67, 119 local method 27 locking application to hardware 307 log object 99 looplevel 65, 249, 270, 273, 274 Lower bound 288 MAC address 307 main application class 78, 82 main user interface component classes 79 Main Window class 79 83 node 79 material link 47 tag 112 Materials node 57 math functions 17 maximum computing 260 maximum value 64 measuring geometric and mass properties 235 menu item 108 mesh information and statistics 279

mesh element size 46, 53 Mesh node 53, 119, 120 message log object 99, 141 message method 145 method 7.118 form 27 get 41 global 27 local 27 name 27 Method class 116 Method Editor 118 using 7, 21 method name 28 Microsoft® Word® format 146 minimum computing 260 model 67 loading 67, 119 saving 67, 119 Model Builder 37 model component 40 model data access 25 model history 71, 181 turning off 181 Model lava-file 71 model methods 26 Model M-file 71 model object 7, 37, 56, 76, 118 class structure 73 tag 37 model tree 37 node 55, 56 model utility methods 68, 118 Model Wizard 41, 67 models, working with multiple 67 MPH file 67, 118, 150 multiphysics 66 Multiphysics node 66

multiple models 67

Ν

name form 76, 78 form object 76, 78 in application object 78 method 27, 28 scoping 38 shortcut 16,76 user interface component 76, 78 Name, of a feature 56 network card 307 new value data validation 114, 290 newline 11, 12 node finite element 280 nonlinear solver 62 normal surface 185, 186 number of parameters 267 number of time steps 267 numerical Derived Values 65 numerical data 240 numerical results 240 Numerical validation 288 o old value data validation 114, 290 On data change event 114, 289 operating system methods 130 operators 50 Java 9 model object 50 OS commands 130 parameter 20, 47, 48, 65 Ρ method 19, 26 real and imaginary part 50

parameterization edge 215 face 217 parametric solution 65 parametric sweep 218, 244, 246, 248 Parametric Sweep, nested feature nodes 261, 264 parsing text file 205 path, getting 196 physics interface 55, 59 play sound 131 plot group 48, 145 mesh element nodes 120 point trajectories 120 surface 51, 63 table surface 65 useGraphics 138 Plot Group node 63 plot type Annotation Data 189 Arrow Data 187, 188 Line Data 182 Point Data 180, 181, 283 Surface Data 183, 184, 185 Tube Data 187 Point Data plot type 180, 181, 283 point trajectories plot 120 precedence, of operators 9, 50 primitive data types 8 printing graphics 149 programming examples 118, 178 Programming Reference Manual 37, 62, 118, 120, 226, 280 progress 153 dialog 154, 158 methods 153

progress bar object 99, 155, 157, 158 properties general 80 property and property values 41 R radio button object 100 ragged arrays 14, 45 ragged sweep 264 reading binary file 2/2 CSV-file 191 Excel® file 195 file 190 matrix file 198 Microsoft Excel® Workbook file 195 spreadsheet 191 text file 205 text file to string 201 real and imaginary part of parameter 50 Record Code 23, 61, 243 Record Method 243 recursion 275 recursive loop 275 removing feature node 53, 56 report 136 email attachment 133 HTMI 146 Microsoft® Word® format 146 request 140, 145 reset model history 71 Results node 63 results table object 100, 142 RGB color 81 S save application 150

save as 149, 150 saving

model 67.119 scalar object 109 scene light 150 sectionwise format 208 210 sectionwise format 208 selection 229 explicit 226, 300 finalized geometry 232 geometry 226, 230 selection input object 101, 300 selections 223 add-in 300 set 41 setIndex 41 shortcuts 16 76 Shortcuts node 16 single quotation mark 12 sleep 161 slider object 102 SMTP 135 solution data 178 data structure 60 parameteric 65 Solution node 60 Solver Configurations node 59 solver sequence 59 spacer object 104 spatial dimension 40 special character 12 Java 67 special characters 205 sphere 179 spreadsheet 191 format 124, 198, 199, 207, 219 Stationary Solver node 60

Stationary study step 59 status bar 153 Store changes, in dialog 291 stream character 206 String data type 10, 41 methods 172 string variable 65 conversion 163 methods 172 strings comparing 11 concatenating 11 Study node 58 subform object 91 subset of output times 270 surface parameterization 217 Surface Data plot type 183, 184, 185 surface normal 185, 186 surface plot 51, 63 Sweep type 246, 248 system matrix accessing 284 system methods 130 OS commands 130 system vector 284 tab 12

```
table 65
email attachment 133
object 105, 141
Table node 65
table surface plot 65
tag 78
model object 37
physics interface 55
Tag, of a feature 56
tag, unique 181
```

temporary folder location 198 text file 190 reading 205 writing 205 text label object 106 text object 105 time 159 time-limited application 307 title 1.38 toggle button object 106 toolbar item 108 object 107 Tooltip 289 transient (time-dependent) simulations 266 transparency 147, 150 transparent color 81 try and catch 20 Tube Data plot type 187 type declaration 74 Type, of a feature 56 U unique tag 181 unit 49 object 107 Unicode 239 Unit List 79 unit set methods 111 object 109 unit sphere 179 Unit System 49 Upper bound 288 username 130 V Value input field 288

Value object 290

т
variable 26 description 20 name completion 22 video object 107 view graphics 143, 146 visualization 178 w web page object 108 while loop 19 with statement 19 writing CSV-file 197 Excel file 198 matrix file 199 string to text file 204 text file 205 to binary file 211

to file 190, 198 Z zoom extents 145, 150