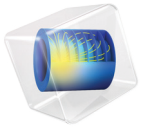


COMSOL Multiphysics

Application Programming Guide

5.2a

COMSOL

Application Programming Guide

Version: COMSOL 5.2a

Contact Information

Visit the Contact COMSOL page at www.comsol.com/contact to submit general inquiries, contact Technical Support, 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 at the COMSOL Access page at www.comsol.com/support/case. Other useful links include:

- Support Center: www.comsol.com/support
- Product Download: www.comsol.com/product-download
- Product Updates: www.comsol.com/support/updates
- COMSOL Blog: www.comsol.com/blogs
- Discussion Forum: www.comsol.com/community
- Events: www.comsol.com/events
- COMSOL Video Gallery: www.comsol.com/video
- Support Knowledge Base: www.comsol.com/support/knowledgebase

Part number: CM020012

Contents

Introduction

This book is a guide to writing code for COMSOL® 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.

Data Types

Primitive Data Types

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

DATA TYPE	DESCRIPTION	EXAMPLE
`byte`	Integer between -127 and 128	byte b=33;
`char`	Unicode character; integer between 0 and 65535 (0 and 2^{16}-1)	char c='a'; char c=97;
`short`	Integer between -32768 and 32767 (-2^{15}-1 and 2^{15}-1)	short s=-1025;
`int`	Integer between 2^{31} and 2^{31}-1	int i=15;
`long`	Integer between 2^{63} and 2^{63}-1	long l=15;
`float`	32-bit floating point number	float f =4.67f;
`double`	64-bit floating point number	double d=4.67;
`boolean`	Boolean with values `false` or `true`	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[5] 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
11	`= += -= *= /= %= >>= <<= &= ^= |=`	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
```

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 non-rectangular 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.

1D 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

For copying arrays, the following code:

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

```
    }
  }
```

is not equivalent to the line:

```
input_array=init_input_array;
```

since the last line will only copy by reference.

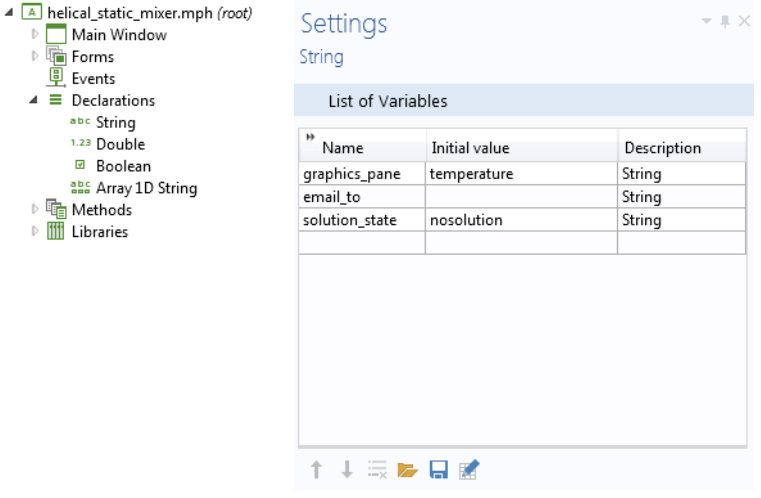
Instead, you can use the `copy` method as follows:

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

which allocates a new array and then copies the values.

The Declarations Node

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



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 51.

Built-in Elementary Math Functions

Elementary math function 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`. 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.");
}
```

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.");
```

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
}
```

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 38.

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 a `with` statement that is used to make Application Builder code more compact and easier to read. 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 31 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®.

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`.

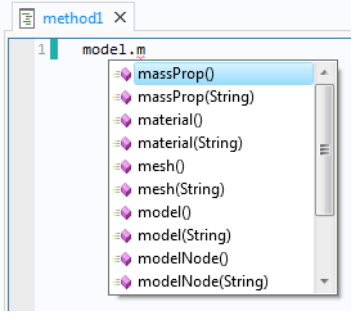
Important Programming Tools

The Method editor includes several tools for automatically generating code. These tools include code completion, **Record Code**, **Convert to New Method**, **Editor Tools**, **Language Elements**, and **Copy as Code to Clipboard**, 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 auto-generate blocks of code.

Ctrl+Space for Code Completion

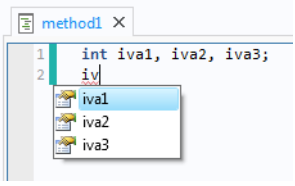
While typing code in the Method editor, the Application Builder can provide suggestions for code completions. The list of possible completions are shown in a separate completion list that opens while typing. In some situations, detailed information appears in a separate window when an entry is selected in the list. Code completion can always be requested with the keyboard shortcut Ctrl+Space. 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.

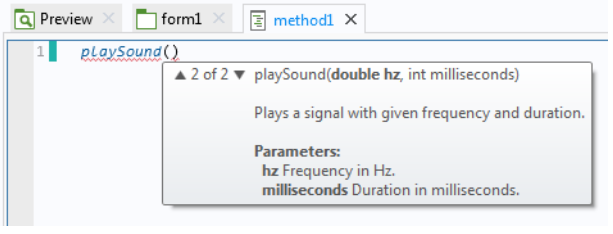
If the list is long, you can filter by typing the first few characters of the completion you are looking for.

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.

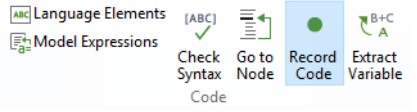


The Method editor also supports code completion for properties, including listing the properties that are available for a given model object feature node, and providing a list of allowed values that are available for a given property.

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.

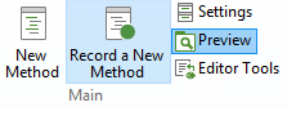
Recording Code

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.

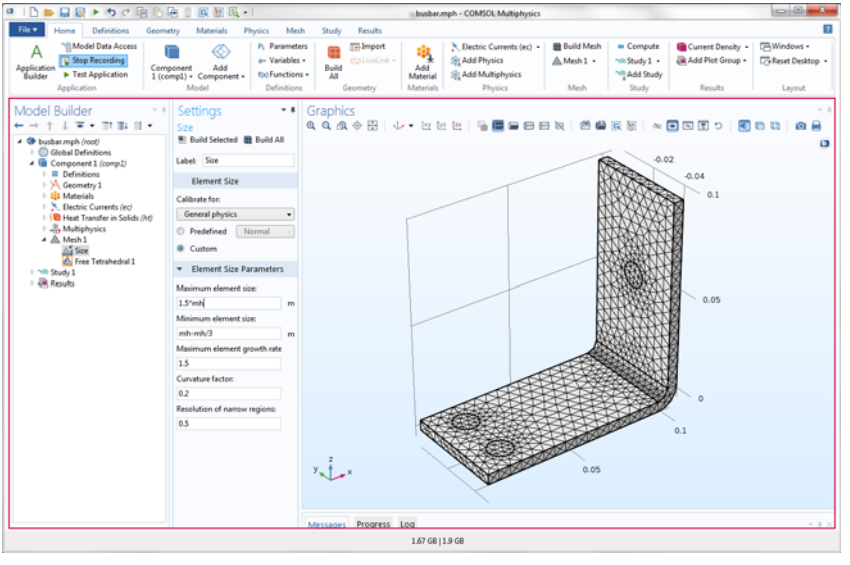


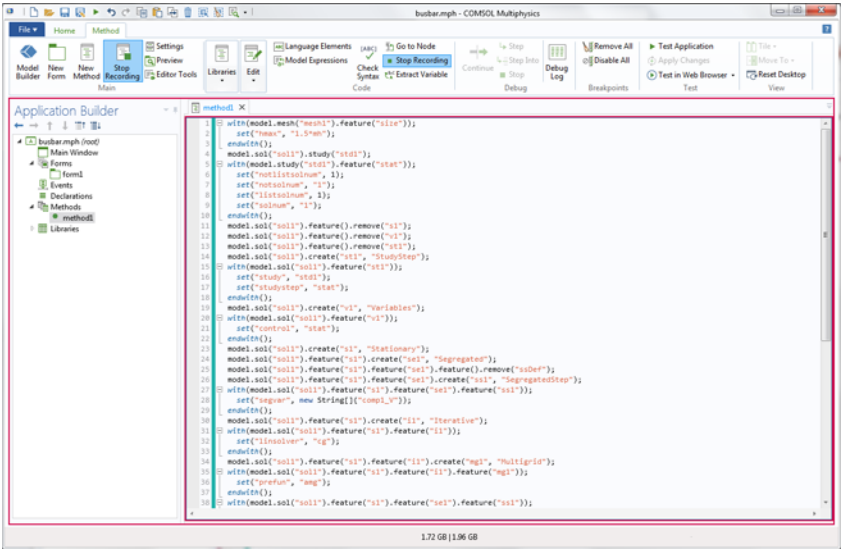
Certain operations in the application tree can also be recorded, for example, code that changes the color of a text label in a running application may be generated.

To record a new method, click the **Record a New Method** button in the **Main** section of the Method editor ribbon.



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

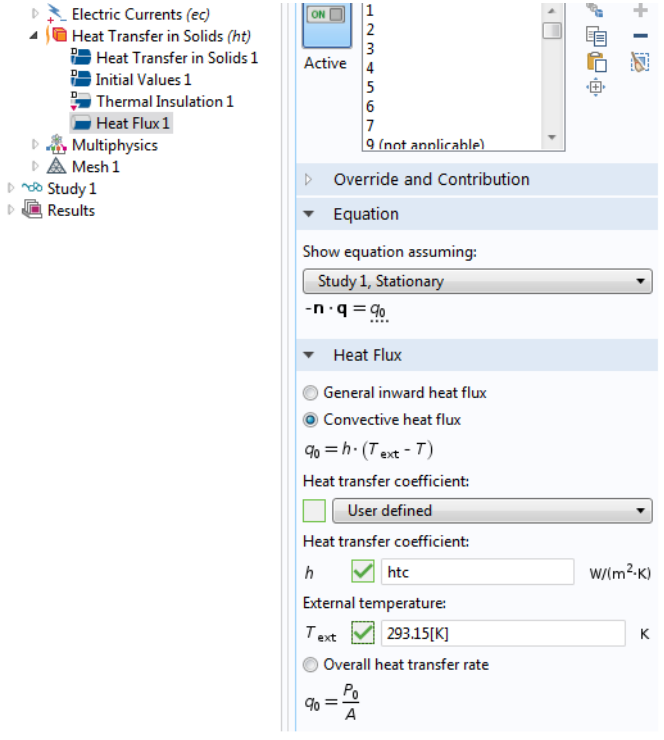




To stop recording code, click one of the **Stop Recording** buttons in the ribbon of either the Model Builder or the Application Builder.

By using **Model Data Access**, you can set the values of the **Heat transfer coefficient** and the **External temperature** properties of the busbar tutorial model used in the

books *Introduction to COMSOL Multiphysics* and *Introduction to Application Builder*.



To generate similar code using **Record Code** (**Model 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, click **Record a New 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:

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

In this case, the autogenerated code contains a `with()` statement in order to make the code more compact. For more information on the use of `with()`, see "The With Statement" on page 15.

To generate code corresponding to changes to the application object, use **Record Code** or **Record a New 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**:

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

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 **Model 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 **Model Data Access**, see the *Introduction to Application Builder*.

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.

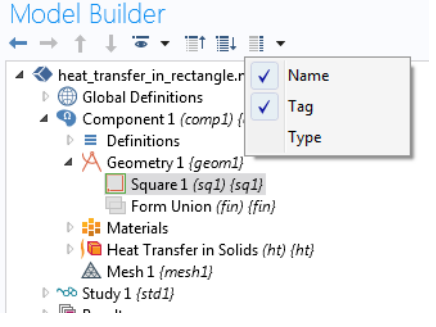
The model object methods are structured in a tree-like way, similar to the nodes in the model tree. The top-level methods just return references that support further methods. At a certain level the methods perform actions, such as adding data to the model object, performing computations, or returning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

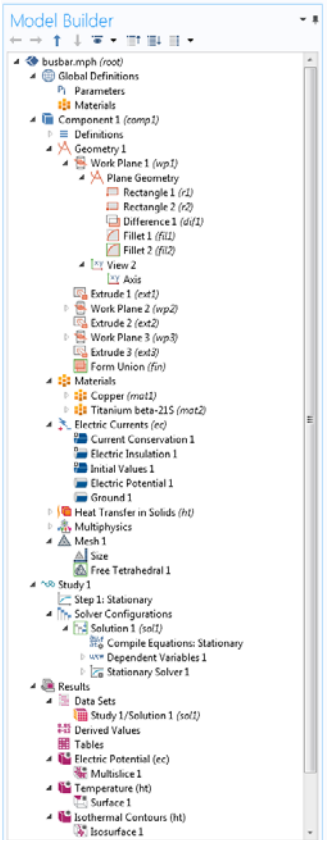
The contents of the application tree in the Application Builder are accessed through the application object, which is an important part of the model object. You can write code using the Method editor to alter, for example, button text, icons, colors, and fonts in the user interface of a running application.

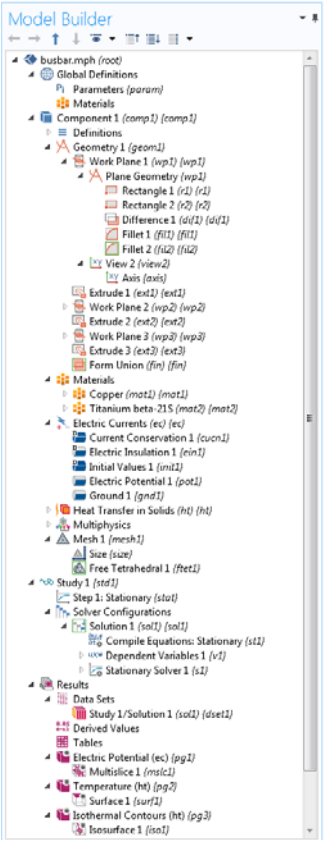
Model Object Tags

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.





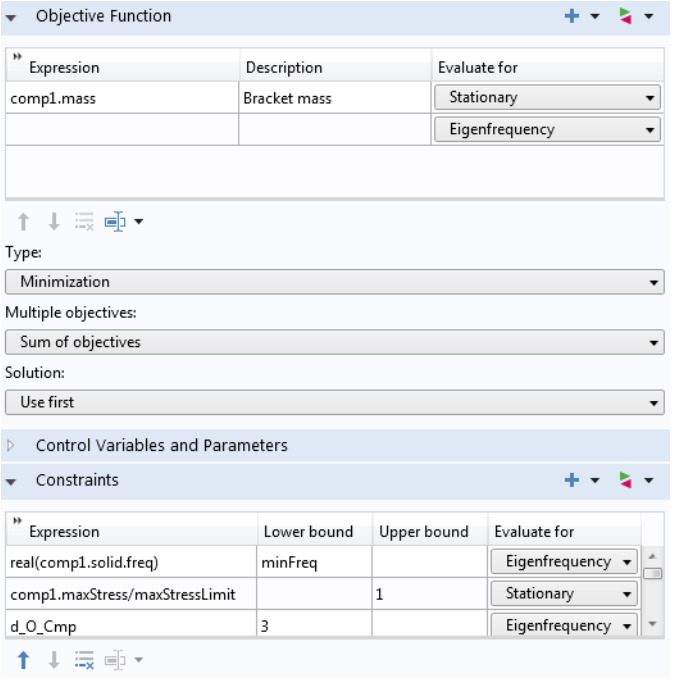
In code, the tags are referenced using double quotes. For example, in the following line

```
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`, `freq`, and

`maxStress` are referenced by scope names: `comp1.mass`, `comp1.solid.freq`, and `comp1.maxStress`.



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. You may also create an embedded model with a call to the `createModel` method:

```
Model model = createModel("Model1");
```

The model tag `Model1` is automatically created and may instead be `Model2`, `Model3`, and so on, to ensure a unique model tag (this depends on which model tags were used previously). When using the Model Wizard, the creation of the model tag is automatically handled.

When writing methods in the Method editor you can directly access the model object `model` without first calling `createModel`.

If you want to create additional model objects in the same application, then you need to call `createModel` or load a model object from file. For more information on working with several model objects, see the section "Working with Model Objects" on page 48.

Creating Model Components and Model Object Nodes

A model contains one or more model components. You create a model component as follows:

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

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.

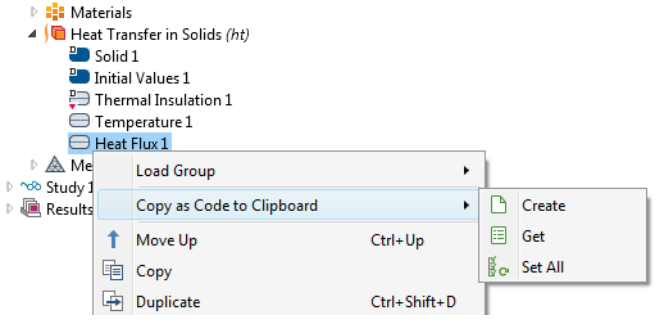
Using the Model Wizard, if you first create a **Blank Model** and then add a component using the Model Builder, you will be prompted to choose the space dimension of the component. This operation will, in addition to creating a component, also create a geometry and mesh node. For example, selecting a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

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

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**.



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 29 below 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	`getInt(String name)`
Integer array	`getIntArray(String name)`
Integer matrix	`getIntMatrix(String name)`

TYPE	SYNTAX
Double	`getDouble(String name)`
Double array	`getDoubleArray(String name)`
Double matrix	`getDoubleMatrix(String name)`
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	`set(String name,String val1)`
String array	`set(String name,new String[]{"val1","val2"})`
String matrix	`set(String name,new String[][]{{"1","2"},{"3","4"}})`
Integer	`set(String name,17)`
Integer array	`set(String name,new int[]{1,2})`
Integer matrix	`set(String name,new int[][]{{1,2},{3,4}})`
Double	`set(String name,1.3)`
Double array	`set(String name,new double[]{1.3,2.3})`
Double matrix	`set(String name,new double[][]{{1.3,2.3},{3.3,4.3}})`
Boolean	`set(String name,true)`
Boolean array	`set(String name,new boolean[]{true,false})`
Boolean matrix	`set(String name,new boolean[][]{{true, false},{false, false}})`

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.

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"` 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,{"1","2","3"},0);
setIndex(name,{"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"}});
```

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:

```
String[] properties();
```

returns the names of all available properties,

```
boolean hasProperty(String name);
```

returns true if the feature has the named property,

```
String[] getAllowedPropertyValues(String name);
```

returns the allowed values for named properties, if it is a finite set.

Example Code

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

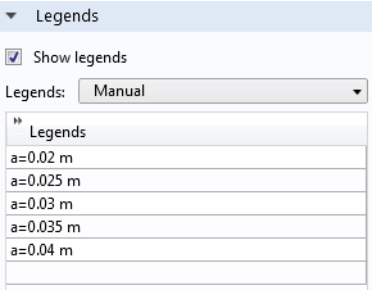
```
if (model.mesh("mesh1").feature("size").getDouble("hauto")<=3) {
  exp_time = "Solution times may be more than 10 minutes for finer element
  sizes.";
}
```

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 line of 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**.



The following line of code sets the **Data Set** `dset1` for the **Plot Group** `pg1`:

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

The following lines of 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 code below sets the global parameter `L` to a fixed value.

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

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

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

The code below sets the coordinates of a cut point data set `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 code below sets the components of a deformation plot.

```
with(model.result("pg7").feature("surf1").feature("def"));
  setIndex("expr", withstru, 0);
  setIndex("expr", withstrv, 1);
  setIndex("expr", withstrw, 2);
endwith();
```

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");
```

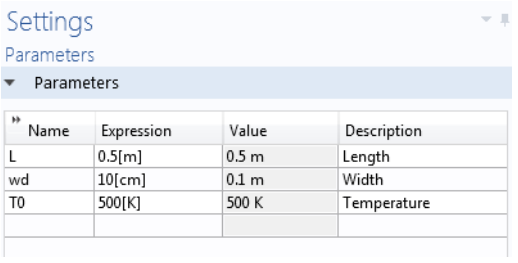
There is an alternative syntax using three input arguments:

```
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**:



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");
```

The evaluation is in this case 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 `Length` with `2` and write the result to the parameter `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 parameters, variables, material properties, and boundary conditions, as well as in expressions used in results for postprocessing and visualization.

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

PRECEDENCE LEVEL	SYMBOL	DESCRIPTION
8	== !=	comparisons: equal, not equal
9	&&	logical and
10	||	logical or
11	,	element separator in lists

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();
```

Geometry

Once the **Geometry** node is created (see "Creating Model Components and Model Object Nodes" on page 25) 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");
```

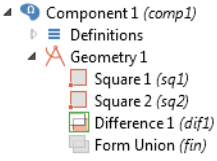
Take the set difference between the first and second square:

```
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);
}
```

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.

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:

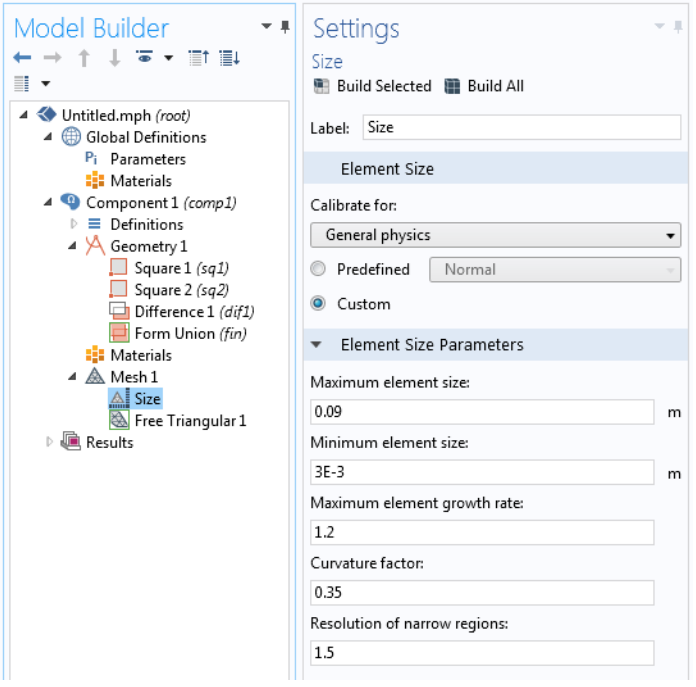
```
model.mesh("mesh1").run();
```

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});
```

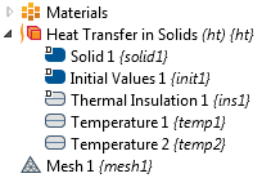
To set the temperature on the boundary to a fixed value of 400 K, use:

```
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("T0", "(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;
java.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++) {
    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.

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");
```

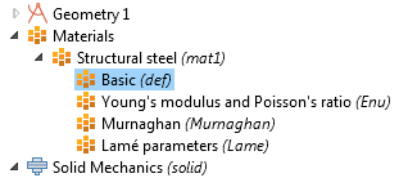
You can give the material a name, for example, `Aluminum`, as follows:

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

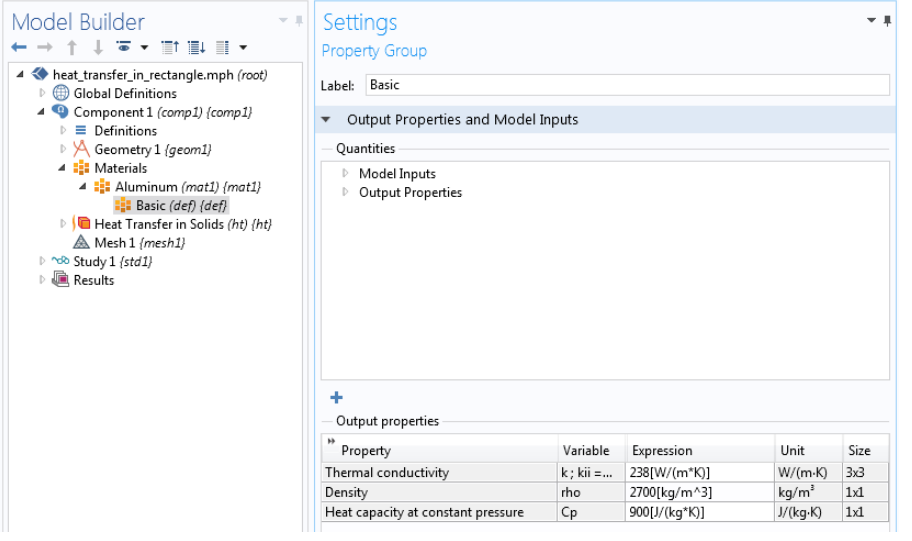
The following lines of code shows how to create a basic material property group for heat transfer:

```
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.

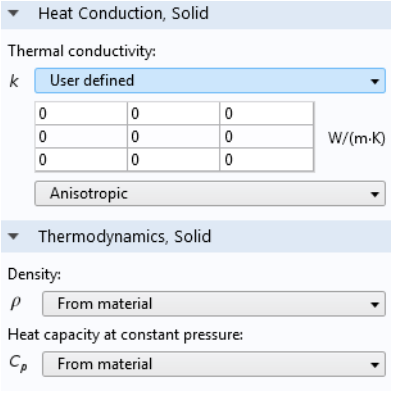


The resulting model tree and **Material** node settings are shown in the figure below.



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.



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, some of the low-level settings in `model.sol` are automatically generated when using **Record Code**.

The example below shows a somewhat more elaborate case of programing the study that would be applicable for the stationary heat transfer example shown earlier. The instructions below more closely resemble the output autogenerated by using the **Record Code** option.

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`:

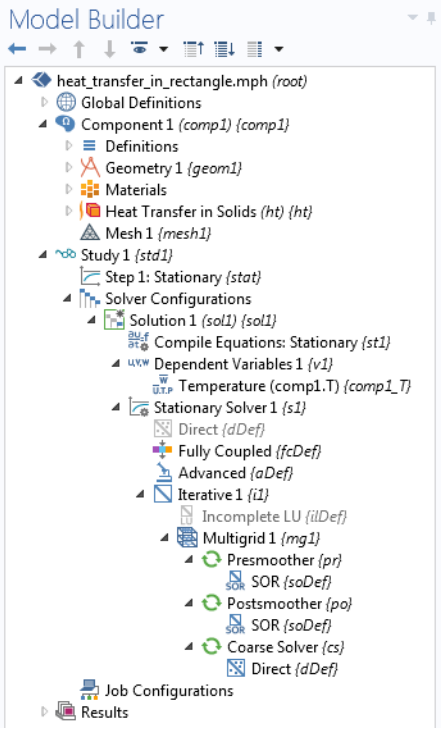
```
model.sol("sol1").attach("std1");
```

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.



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");
with(model.sol("sol1").feature("s1").feature("fc1"));
  set("dtech", "auto"); //      The Nonlinear method (Newton solver)
  set("initstep", "0.01"); //   Initial damping factor
  set("minstep", "1.0E-6"); //  Minimum damping factor
  set("rstep", "10"); //        Restriction for step-sized update
  set("useminsteprecovery", "auto"); // Use recovery damping factor
  set("minsteprecovery", "0.75"); //    Recovery damping factor
  set("ntermauto", "tol"); //   Termination technique
  set("maxiter", "50"); //      Maximum number of iterations
  set("ntolfact", "1"); //      Tolerance factor
  set("termonres", "auto"); //  Termination criterion
  set("reserrfact", "1000"); // Residual factor
endwith();
```

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 `solution_state`. The following code sets the state depending on the output from the `isEmpty` method:

```
if (model.sol("sol1").isEmpty()) {
  solution_state = "nosolution";
}
```

```
else {
  solution_state = "solutionexists";
}
```

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

Results

The **Results** node contains nodes for **Data Sets**, **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 data set `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 intra-element interpolation scheme, and the Data Set referring to the parent of the **Surface** plot node, which is the `pg1` node:

```
model.result("pg1").feature().create("surf1", "Surface");
model.result("pg1").feature("surf1").label("Surface");
with(model.result("pg1").feature("surf1"));
  set("colortable", "ThermalLight");
  set("smooth", "internal");
  set("data", "parent");
endwith();
```

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

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

```
  set("data", "parent");
endwith();
```

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

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

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

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 following example 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");
with(model.result("pg1").feature("tbls1"));
  set("table", "tbl1");
endwith();
with(model.result("pg1").feature("tbls1"));
  set("dataformat", "cells");
endwith();
model.result("pg1").feature("tbls1").create("hght1", "TableHeight");
with(model.result("pg1").feature("tbls1").feature("hght1"));
  set("view", "view3");
endwith();
with(model.view("view3").camera());
  set("viewscaletype", "manual");
  set("xscale", "1");
  set("yscale", "1");
  set("zscale", "1");
endwith();
// The following line is needed to update the plot
model.result("pg1").run();
```

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 doesn't 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.

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

```
    set("EMHeat_physics", "ec");
    set("Heat_physics", "ht");
  endwith();
```

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.

Working with Model Objects

When using the Model Builder in the COMSOL Desktop, 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 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 model = 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("My_Model_1", "C:\\COMSOL_Work\\my_model.mph");
```

The model tag `My_Model_1` is chosen arbitrarily by you and is assigned to the model upon load. You just need to ensure that the model tag doesn't collide with

the tag of the embedded model, or with any other tag associated with a loaded model. Note the double-backslash syntax in the file name. Backslash (\) 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 since it is used to retrieve the model from other methods. Once loaded, the model `extmodel` exists in the work space of the current COMSOL Multiphysics or COMSOL Server session. Note that an MPH file can only contain one model object, so there is no ambiguity on which model you refer to 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.

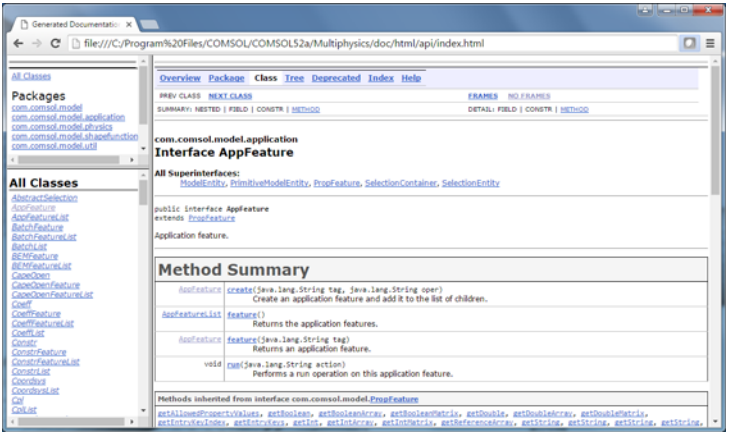
For a list of model utility methods, see “Model Utility Methods” on page 82.

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 Help Desk*. For a Windows installation in the default location, the document is located at:

```
C:/Program Files/COMSOL/COMSOL52a/Multiphysics/doc/html/comsol/helpdesk.html
```

The figure below shows the document as displayed in a browser.



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.

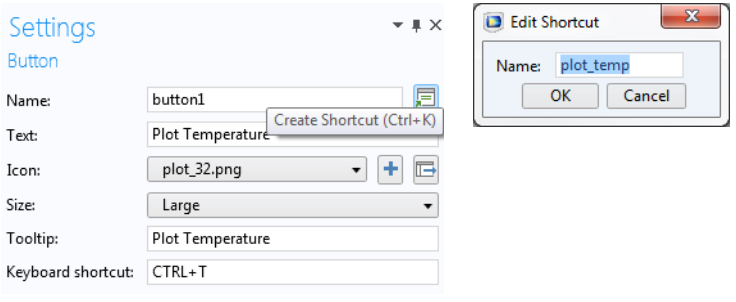
You can write code using the Method editor to directly access and change the features presented in a running application, including button text, icons, colors, and fonts.

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 run time 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*.

Shortcuts

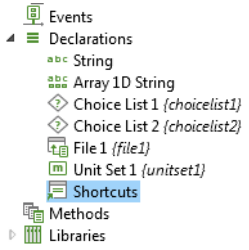
Form objects and other user interface components are referenced in methods by using a certain syntax. For example, using the default naming scheme `form3/button5` refers to a button with name `button5` in `form3` and `form2/graphics3` refers to a graphics object with name `graphics3` in `form2`. You 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.

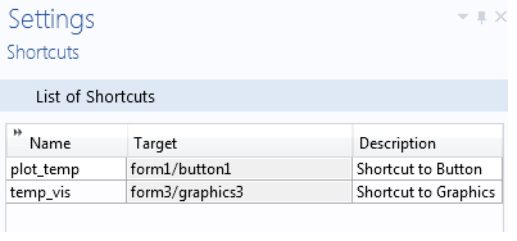


To create or edit a shortcut, you can also use the keyboard shortcut Ctrl+K.

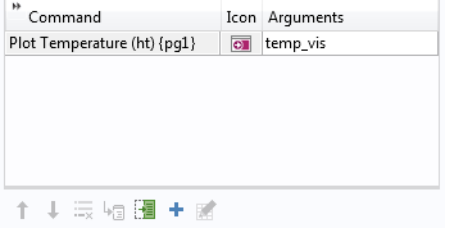
All shortcuts that you create are made available in a **Shortcuts** node under **Declarations** in the application tree.



In the **Settings** window for **Shortcuts** shown below, two shortcuts `plot_temp` and `temp_vis` were created for a button and a graphics object, respectively.



The shortcuts can be referenced in other form objects or in code in the Method editor. The example below shows a shortcut `temp_vis` 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.

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, etc. 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`. For more information, see "Data Source: Choice List and Unit Set" on page 78.

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.

Each class has a set of associated methods that are used to edit the corresponding user interface component at run time. 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 Color

The `get` and `set` methods described in the section "Get and Set Methods for Accessing Properties" on page 26 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 user interface component:

NAME	SYNTAX	DESCRIPTION
`setColor`	`setColor(String prop, int r, int g, int b)`	Set a color property using red, green, and blue values.
`getColor`	`int[] getColor(String prop)`	Get the value of a color property as an array of 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
`visible`	`true | false`	`true`	If the value is `true`, the corresponding object is visible in the user interface.
`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.
`background`	`String`	`default`	The background color for the corresponding user interface element.
`foreground`	`String`	`default`	The foreground color for the corresponding user interface element.
`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.
`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.

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)");
```

The Main Application Methods

The main application class `AppModel` has the following methods:

NAME	SYNTAX	DESCRIPTION
`mainWindow`	`MainWindow mainWindow()`	Returns the `MainWindow` object.
`form`	`FormList form()`	Returns the list of forms.
`form`	`Form form(String name)`	Returns the form with the specified name.
`declaration`	`DataSource declaration(String name)`	Returns the declaration object (`ChoiceList` or `UnitSet`) with the specified name.

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");
```

Main Window

The `MainWindow` class has the following methods:

NAME	SYNTAX	DESCRIPTION
`menuBar`	`ItemList menuBar()`	Returns the list of items in the menu bar.
`menuBar`	`Item menuBar(String name)`	Returns the menu bar item with the specified name.
`toolBar`	`ItemList toolBar()`	Returns the list of items in the toolbar.
`toolBar`	`Item toolBar(String name)`	Returns the toolbar item with the specified name.
`ribbon`	`ItemList ribbon()`	Returns the list of items in the ribbon.
`ribbon`	`Item ribbon(String name)`	Returns the ribbon item with the specified name.
`fileMenu`	`ItemList fileMenu()`	Returns the list of items in the file menu.
`fileMenu`	`Item fileMenu(String name)`	Returns the file menu item with the specified name.

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.

The `MainWindow` class has the following properties:

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`showfilename`	`true | false`	`true`	If true the filename is included in the window titlebar title.
`title`	`String`	`My application`	The text to display in the window titlebar.

Example Code

```
app.mainWindow().set("showfilename", false);
```

Form

The `Form` class has the following methods:

NAME	SYNTAX	DESCRIPTION
`getName`	`String getName()`	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.
`formObject`	`FormObjectList formObject()`	Returns the list of form objects.
`formObject`	`FormObject formObject(String name)`	Returns the form object 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 Images>Libraries node in the application tree.
`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);
```

Form Object

The `FormObject` class has the following methods:

NAME	SYNTAX	DESCRIPTION
`getName`	`String getName()`	Returns the name of this form object.
`getParentForm`	`Form getParentForm()`	Returns the parent form that contains this form object.
`getType`	`String getType()`	Returns the form object type name, as defined in the following sections.
`form`	`FormList form()`	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.

Most form objects have one or more of the properties listed in "General Properties" on page 55. 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"`.

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`	`Line`	`Text`
`DataDisplay`	`ListBox`	`TextLabel`
`Equation`	`Log`	`ToggleButton`
`FileImport`	`MessageLog`	`Toolbar`
`Form`	`ProgressBar`	`Unit`
`FormCollection`	`RadioButton`	`Video`
`Graphics`	`ResultsTable`	`WebPage`

Array Input

Property	Value	Default	Description
background enabled font fontbold fontitalic fontsize foreground visible			See "General Properties" on page 55.

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 55.

`icon`	`String`	`""`	The button icon. Valid values are images defined in "Images > Libraries" 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 55.

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("butto
n1").set("enabled", false);
```

Check Box

PROPERTY	VALUE	DEFAULT	DESCRIPTION

`background`
`enabled`
`font`
`fontbold`
`fontitalic`
`fontsize`
`fontunderline`
`foreground`
`visible`
			See "General Properties" on page 55.

`text`	`String`	Generated automatically	The check box label text.
`tooltip`	`String`	`""`	The check box tooltip text.

Example Code

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

Combo Box

PROPERTY	VALUE	DEFAULT	DESCRIPTION

`enabled`
`font`
`fontbold`
`fontitalic`
`fontsize`
`foreground`
`visible`
			See "General Properties" on page 55.

Example Code

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

DATA DISPLAY

PROPERTY	VALUE	DEFAULT	DESCRIPTION

`background`
`enabled`
`font`
`fontbold`
`fontitalic`
`fontsize`
`foreground`
`visible`
			See "General Properties" on page 55.

`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 55.

Example Code

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

File Import

PROPERTY	VALUE	DEFAULT	DESCRIPTION

`background`
`enabled`
`font`
`fontbold`
`fontitalic`
`fontsize`
`foreground`
`visible`
			See "General Properties" on page 55.

`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 55.

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

Example Code

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

GRAPHICS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`enabled` `visible`			See "General Properties" on page 55.
`source`	`ModelEntity`		Defines the type of model entity (Plot Group, Geometry, Mesh, Explicit Selection or Player Animation) 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");
```

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("graphi
cs1").set("source", model.mesh("mesh1"));
```

HYPERLINK

PROPERTY	VALUE	DEFAULT	DESCRIPTION

`background`
`enabled`
`font`
`fontbold`
`fontitalic`
`fontsize`
`visible`
			See "General Properties" on page 55.

`text`	`String`	Generated automatically	The text to display on the `HyperLink` form object.
`url`	`String`	`""`	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 55.
`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 tree.

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 55.

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 55.

`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 check box 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 check box 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 check box 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 check box 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`	`""`	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

`enabed`
`font`
`fontbold`
`fontitalic`
`fontsize`
`foreground`
`visible`
			See "General Properties" on page 55.

`text`	`String`	`""`	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.

Example Code

```
app.form("form1").formObject("line1").set("text", "divider text");
```

LIST BOX

PROPERTY	VALUE	DEFAULT	DESCRIPTION

`enabled`
`font`
`fontbold`
`fontitalic`
`fontsize`
`foreground`
`visible`
			See "General Properties" on page 55.

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 fontsize foreground visible			See "General Properties" on page 55.

Example Code

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

MESSAGE LOG

PROPERTY	VALUE	DEFAULT	DESCRIPTION
background enabled font fontbold fontitalic fontsize foreground visible			See "General Properties" on page 55.

Example Code

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

PROGRESS BAR

PROPERTY	VALUE	DEFAULT	DESCRIPTION
enabled visible			See "General Properties" on page 55.

To create and update progress information see "Progress Methods" on page 110.

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 55.

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 55.

To change the contents of the results table use the method `useResultsTable` or `evaluateToResultsTable`. See also "GUI-Related Methods" on page 97.

Example Code

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

SELECTION INPUT

PROPERTY	VALUE	DEFAULT	DESCRIPTION

`background`
`enabled`
`font`
`fontbold`
`fontitalic`
`fontsize`
`foreground`
`visible`
			See "General Properties" on page 55.

`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 55.
`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.
`type`	`real | integer`	`real`	If `type` has the value `integer`, then the slider value is restricted to integer values. If `type` has the value `real`, the slider value can be a noninteger value.

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);
```

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 55.

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);
```

TEXT

PROPERTY	VALUE	DEFAULT	DESCRIPTION

`background`
`enabled`
`font`
`fontbold`
`fontitalic`
`fontsize`
`foreground`
`visible`
			See "General Properties" on page 55.

`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.

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 55.

`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 55.

`icon`	`String`	`""`	The button icon. Valid values are images defined in "Images>Libraries" 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 55.

Example Code

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

UNIT

PROPERTY	VALUE	DEFAULT	DESCRIPTION

`background`
`enabled`
`font`
`fontbold`
`fontitalic`
`fontsize`
`foreground`
`visible`
			See "General Properties" on page 55.

Example Code

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

VIDEO

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`visible`			See "General Properties" on page 55.

Example Code

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

Web Page

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`visible`			See "General Properties" on page 55.
`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	`http://www.comsol.com`	The URL to display.

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();
```

Item

`Item` objects represent items, toggle items, and submenus in the menu bar, toolbar, ribbon and file menu. The following methods are available:

NAME	SYNTAX	DESCRIPTION
`item`	`ItemList item()`	Returns the list of subitems.
`item`	`Item item(String name)`	Returns the subitem with the specified name.
`getParentItem`	`Item getParentItem()`	Returns the parent item, or null for a top-level item.

The `Item` class contains the following properties:

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 **Images > Libraries 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.

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.

Example Code

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

Data Source: Choice List and Unit Set

A `DataSource` object is either a `ChoiceList` or a `UnitSet`.

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 run time.

NAME	SYNTAX	DESCRIPTION
`setListRow`	`setListRow(String value, String displayName, int row)`	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`	`setList(String[] values, String[] displayNames)`	Sets all of the values and display names, replacing the contents of the choice list or unit set.
`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.
`getValue`	`String getValue(int row)`	Returns the value for the given row (0-based).
`getDisplayName`	`String getDisplayName(int row)`	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`	`String[] getValues()`	Returns all values as an array.
`getDisplayNames`	`String[] getDisplayNames()`	Returns all display names as an array.
`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.
`removeListRow`	`removeListRow(int row)`	Removes the given row (0-based) from the list.

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
`set`	`set(String value)`	Switch unit for the unit set.
`getString`	`String getString()`	Returns the currently selected value for the unit set.
`getString`	`String getString(String unitList)`	Returns the selected unit for the given unit list.

Form, Form Object, and Item List Methods

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

NAME	SYNTAX	PURPOSE
`names`	`String[] names()`	Returns an array of names for all objects in the list.
`size`	`int size()`	Returns the number of objects in the list.
`index`	`int index(String name)`	Returns the 0-based position of the object with a given name in the list.

`get`	`Form get(String name)`
	`FormObject get(String name)`
	`Item get(String name)`
		Returns the object with a given name.

`get`	`Form get(int index)`
	`FormObject get(int index)`
	`Item get(int index)`
		Returns the object at a certain index.

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:

```
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 51.

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 creating, loading, and saving model objects. The model object is stored on the MPH file format.

NAME	SYNTAX	DESCRIPTION
`createModel`	`Model createModel(String tag)`	Creates a new model with the given tag.
`removeModel`	`removeModel(String tag)`	Removes a model. The embedded model cannot be removed.
`modelTags`	`String[] modelTags()`	Returns an array of model tags for all loaded models, including the embedded model.
`uniqueModeltag`	`String uniqueModeltag(String prefix)`	Returns a model tag that is not in use.
`getModel`	`Model getModel(String tag)`	Returns the model with a specified tag.
`loadModel`	`Model loadModel(String tag, String filename)`	Loads a model with a specified tag from a file.
`loadProtectedModel`	`Model loadProtectedModel(String tag, String filename, String password)`	Loads a password protected model with a specified tag from a file.

NAME	SYNTAX	DESCRIPTION
`loadRecoveryModel`	`Model loadRecoveryModel(String tag, String foldername)`	Loads a model from a recovery directory/folder structure.
`saveModel`	`saveModel(Model model, String filename)`	Saves a model to a file. The filename can be a file scheme path or (if allowed by security settings) a server file path.

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 data set in combination with a 3D Point Trajectories plot.)

```
Model extmodel = loadModel("model", "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.*

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 107.

NAME	SYNTAX	DESCRIPTION
`readFile`	`String readFile(String name)`	Returns the contents in the given file `name` 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 file `name`.

NAME	SYNTAX	DESCRIPTION
`openBinaryFileStreamReader`	`CsBinaryReader openBinaryFileStreamReader(String 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.
`readStringMatrixFromFile`	`String[][] readStringMatrixFromFile(String name)`	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.
`readCSVFile`	`String[][] readCSVFile(String name)`	Reads a file with comma-separated values (CSV file) into a string matrix. Expects file to use the RFC 4180 format for CSV.
`writeFile`	`writeFile(String name, String contents)`	Writes the given string `contents` to the given file `name`.
`writeFile`	`writeFile(String name, String contents, boolean append)`	Writes the given string `contents` to the given file `name`. If `append` is true, then the contents are appended instead of overwritten.
`writeFile`	`writeFile(String name, double[][] data)`	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`	`writeFile(String name, String[][] data)`	Writes the array `data` to the given file. The spreadsheet format is used, which means it can be read by `readStringMatrixFromFile`.

NAME	SYNTAX	DESCRIPTION
`writeFile`	`writeFile(String name, String[][] data, boolean append)`	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.
`openBinaryFileStreamWriter`	`CsBinaryWriter openBinaryFileStreamWriter(String name)`	Returns a `CsBinaryWriter` that can be used to write to the given file byte-by-byte.
`openBinaryFileStreamWriter`	`CsBinaryWriter openBinaryFileStreamWriter(String 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`	`writeCSVFile(String name, String[][] data)`	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`	`writeCSVFile(String name, double[][] data)`	Writes the given double array `data` to a CSV file. The RFC 4180 format is used for the CSV.
`writeCSVFile`	`writeCSVFile(String name, double[][] data, boolean append)`	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.

NAME	SYNTAX	DESCRIPTION
`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.
`copyFile`	`copyFile(String sourceFile, String destFile)`	Copies a file on the server. Both the source and target names can use file scheme paths.
`importFile`	`importFile(String name)`	Displays a file browser dialog box and uploads the selected file to the file declaration with the given `name`. After this, the uploaded file can be accessed with `upload:///<name>`.
`importFile`	`importFile(ModelEntity entity, String name)`	Displays a file browser dialog box 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`.
`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.

NAME	SYNTAX	DESCRIPTION
`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.
`readExcelFile`	`String[][] readExcelFile(String name)`	Reads the first sheet of an Excel file, starting from the first cell, into a String[][].
`readExcelFile`	`String[][] readExcelFile(String name, String sheet, String cell)`	Reads the specified sheet of an Excel file, starting from the specified cell, into a String[][].
`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.

NAME	SYNTAX	DESCRIPTION
`getClientFileName`	`String getClientFileName(String name)`	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.
`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). 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.

Example Code

This line of code copies the uploaded file `file1` to the `temp` folder with new file name `file2.mphbin`.

```
copyFile("upload:///file1", "temp:///file2.mphbin");
```

This line of code deletes the file `file2.mphbin` from the `temp` folder.

```
deleteFile("temp:///file2.mphbin");
```

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`	`String executeOSCommand(String command, int timeoutSec, String params...)`	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 "File Methods".
`getUser`	`String username = getUser()`	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.
`openURL`	`openURL(String url)`	Opens a URL in the default browser on the client.

NAME	SYNTAX	DESCRIPTION
`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");
```

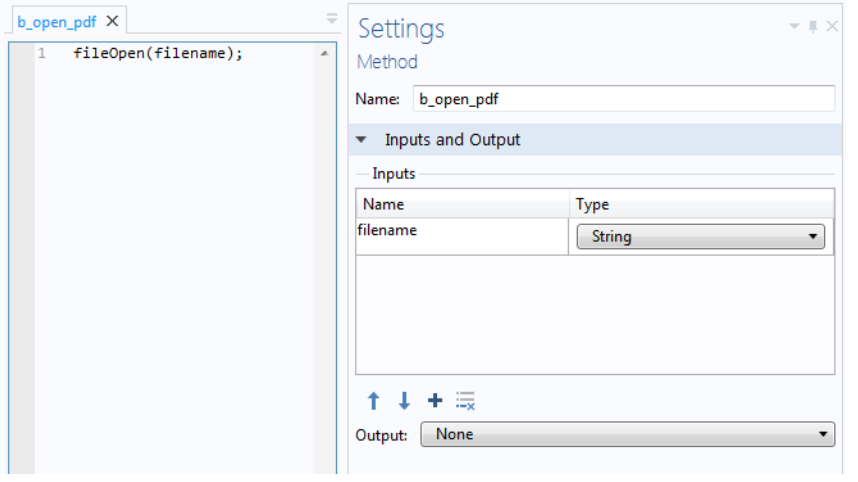
In the command sequence of a form object, this is equivalent to selecting a sound node under **Libraries** and clicking **Run**.

The line of code below opens a PDF file embedded in the application and stored in the File library.

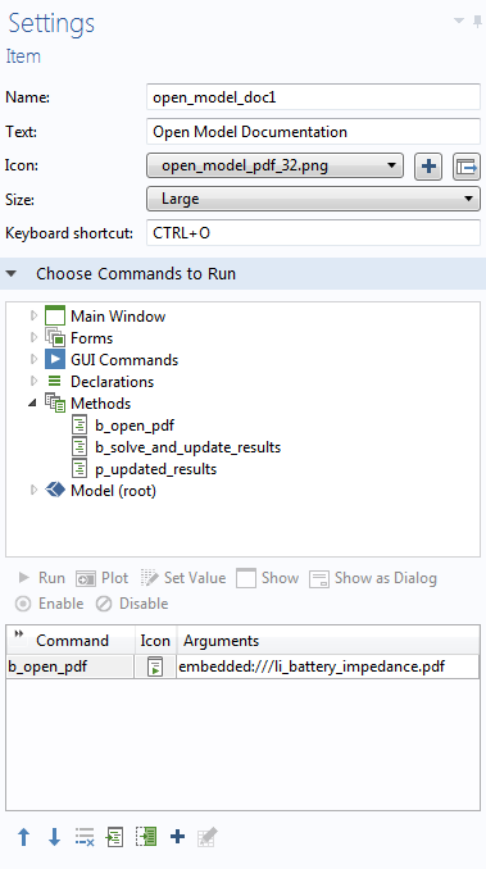
```
fileOpen("embedded:///tubular_reactor.pdf");
```

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 an alternative technique, you can call a method in a command sequence with an input argument, as shown in the example below. The figure below shows a method `b_open_pdf` that opens a file with `filename` as an input argument.



The figure below shows the corresponding command sequence for a ribbon menu item.



Note that the same functionality is available from a command sequence by selecting the editor tree node **GUI Commands**>**File Commands**>**Open File**.

This line of code opens the COMSOL home page in the default browser:

```
openURL("http://www.comsol.com");
```

Email Methods

Email methods are used for sending emails from an application, typically with attachments containing results from a simulation.

NAME	SYNTAX	DESCRIPTION
`emailFromAddress`	`String emailFromAddress()`	Returns the email from address from the COMSOL Server or preferences setting.
`sendEmail`	`sendEmail(String subject, String bodyText)`	Sends an email to the default recipient(s) with the specified subject and body text.
`sendEmail`	`sendEmail(String subject, String bodyText, ModelEntity... modelEntity)`	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.setServer`	`mail.setServer(String host, int port)`	Sets the email (SMTP) server host and port to use for this email message.

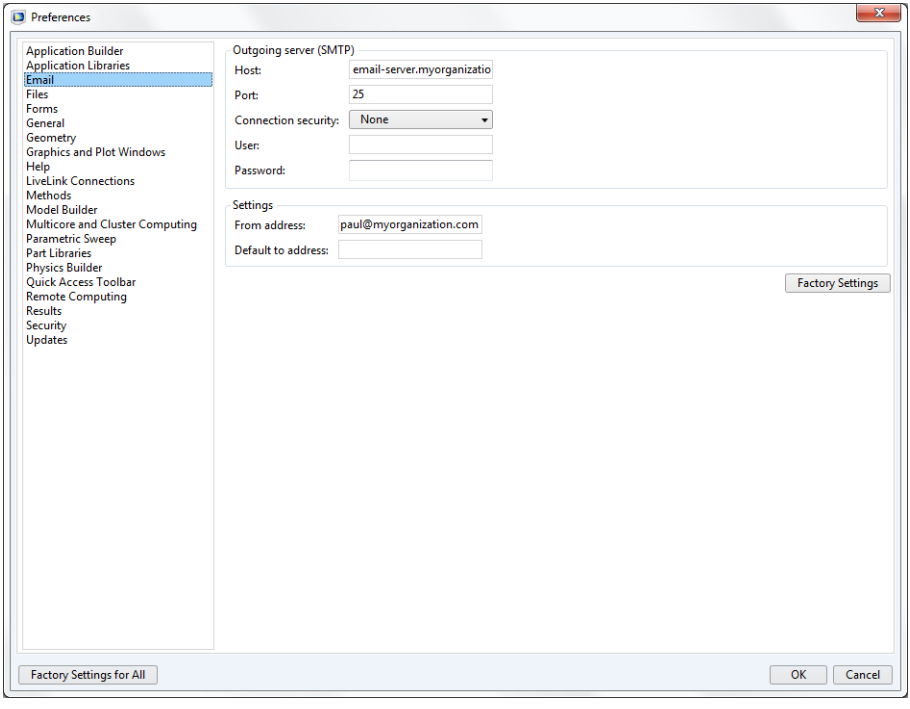
NAME	SYNTAX	DESCRIPTION
`EmailMessage.setUser`	`mail.setUser(String name, String password)`	Sets the username and password to use for email (SMTP) server authentication. This method must be called after the `setServer` method.
`EmailMessage.setSecurity`	`mail.setSecurity(String security)`	Sets the connection security type for email (SMTP) server communication. Valid values are `'none'`, `'starttls'` and `'tls'`. This method must be called after the `setServer` method.
`EmailMessage.setFrom`	`mail.setFrom(String fromAddress)`	Sets the from address.
`EmailMessage.setTo`	`mail.setTo(String... to)`	Sets the to addresses.
`EmailMessage.setCc`	`mail.setCc(String... cc)`	Sets the cc addresses.
`EmailMessage.setBcc`	`mail.setBcc(String... bcc)`	Sets the bcc addresses.
`EmailMessage.setSubject`	`mail.setSubject(String subject)`	Sets the email subject line. Note that newline characters are not allowed.
`EmailMessage.setBodyText`	`mail.setBodyText(String body)`	Sets the email body as plain text. An email can contain both a text and an HTML body.
`EmailMessage.setBodyHtml`	`mail.setBodyHtml(String body)`	Sets the email body as HTML text. An email can contain both a text and an HTML body.
`EmailMessage.attachFile`	`mail.attachFile(String filename)`	Adds an attachment from a file. The attachment MIME type is determined by the file name extension.
`EmailMessage.attachFile`	`mail.attachFile(String filename, String mimeType)`	Adds an attachment from a file with the specified MIME type.
`EmailMessage.attachFromModel`	`mail.attachFromModel(ModelEntity modelEntity)`	Adds an attachment created from a report, export, or table feature in the model.
`EmailMessage.attachText`	`mail.attachText(String text, String mimeSubType)`	Adds a text attachment with a specified sub-MIME type, such as plain or HTML.

NAME	SYNTAX	DESCRIPTION
`EmailMessage.attachBinary`	`mail.attachBinary(byte[] binary, String mimeType)`	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 preferences for an outgoing email (SMTP) server, open the **Email** page of the **Preferences** dialog box, as shown in the figure below.



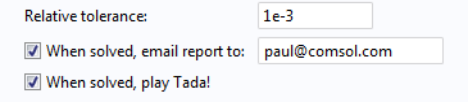
COMSOL Server provides a similar set of email preferences.

Example Code

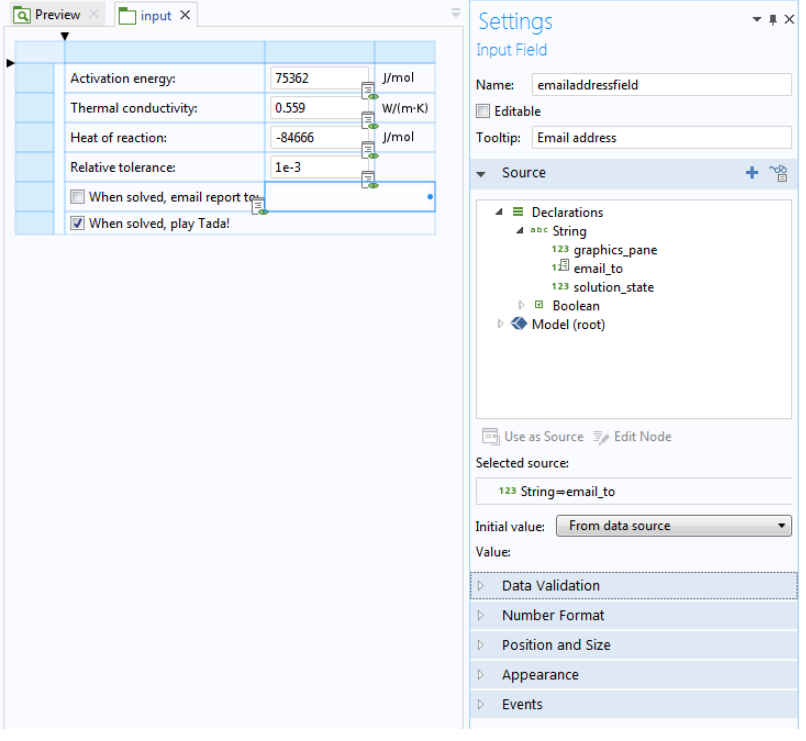
The following code sends an email and attaches a report:

```
EmailMessage mail = new EmailMessage();
mail.setTo(email_to);
mail.setSubject("Tubular Reactor Simulation");
mail.setBodyText("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.



The figure below shows the corresponding form object and **Settings** window.



The following code is similar but also configures the email server settings.

```
EmailMessage mail = new EmailMessage();
mail.setServer("smtp.myemail.com", 587);
mail.setUser("user@myemail.com", "password");
mail.setSecurity("starttls");
mail.setFrom("user@myemail.com");
mail.setTo("otheruser@somedomain.com");
mail.setSubject("Tubular reaction simulation");
mail.setBodyText("The computation has finished");
mail.send();
```

GUI-Related Methods

The graphical user interface (GUI) related methods are used for displaying dialog boxes 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>()`	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.
`useGraphics`	`useGraphics(ModelEntity entity, String name)`	Plots the given entity (Plot Group, Geometry, Mesh, Explicit Selection or Player Animation) in the graphics form object given by the name or name path in the second argument.
`useForm`	`useForm(String name)`	Shows the form with the given name in the current main window. Equivalent to the `use` method of a `Form` object; see below.
`closeDialog`	`closeDialog(String name)`	Closes the form, shown as a dialog box, with the given name.

NAME	SYNTAX	DESCRIPTION
`dialog`	`dialog(String name)`	Shows the form with the given name as a dialog box. 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.
`confirm`	`String confirm(String text)`	Stops execution and displays a confirmation dialog box 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 box 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 box 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.

NAME	SYNTAX	DESCRIPTION
`confirm`	`String confirm(String text, String title, String yes, String no, String cancel)`	Stops execution and displays a confirmation dialog box 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 box with the given message.
`request`	`String request(String text)`	Stops execution and displays a dialog box 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 box 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 box 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 box. 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.
`evaluateToResultsTable`	`evaluateToResultsTable(NumericalFeature entity, String name, boolean clear)`	Evaluates the given entity, a Derived Value, in the table object given by the name or name path 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(NumericalFeature entity)`	Evaluates the given entity, a Derived Value, and returns the non-parameter 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.
`evaluateToIntegerArray2D`	`int[][] evaluateToIntegerArray2D(NumericalFeature entity)`	Evaluates the given entity, a Derived Value, and returns the non-parameter 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.

NAME	SYNTAX	DESCRIPTION
`evaluateToStringArray2D`	`String[][] evaluateToStringArray2D(NumericalFeature entity)`	Evaluates the given entity, a Derived Value, and returns the non-parameter 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)`	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`	`setFormObjectEnabled(String name, boolean enabled)`	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.
`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.

NAME	SYNTAX	DESCRIPTION
`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`	`setFileMenuItemEnabled(String name, boolean enabled)`	Sets the enable state for the file menu item specified by the name or name path (from file menu) in the first argument.
`setRibbonItemEnabled`	`setRibbonItemEnabled(String name, boolean enabled)`	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)`	Applies a view to the graphics contents given by the name or name path in the second argument.
`resetView`	`resetView(String name)`	Resets the view to its initial state in the graphics contents given by the name or name path in the second argument.
`getView`	`View getView(String name)`	Returns the view currently used by the graphics contents given by the name or name path in the second argument.

NAME	SYNTAX	DESCRIPTION
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.
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.

Alerts and Messages

The methods `alert`, `confirm`, and `request` display a dialog box 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 box:

```
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

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");
```

The code below displays a request dialog box that lets the user type in a file name for an HTML report. If the user has typed a file name, then a report is generated.

```
String answerh = request("Enter file name","File Name", "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();
}
```

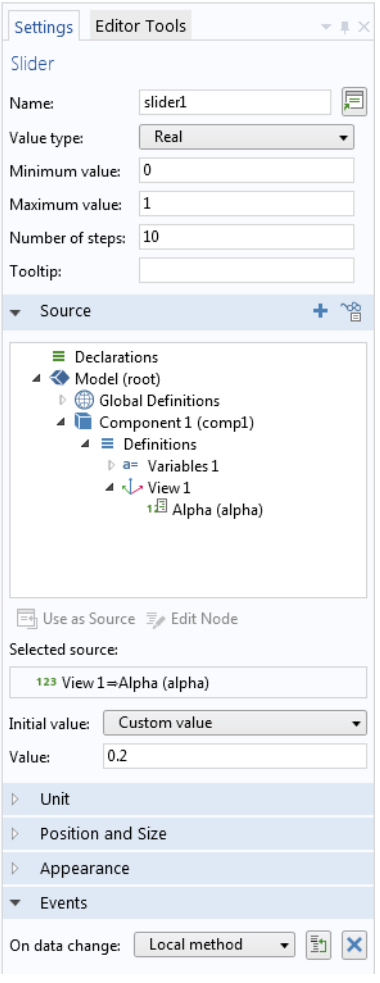
The code below is similar to the code above, but in this case the report is saved in Microsoft® Word® format (`.docx`).

```
String answerw = request("Enter file name","File Name", "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 **Model 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**.

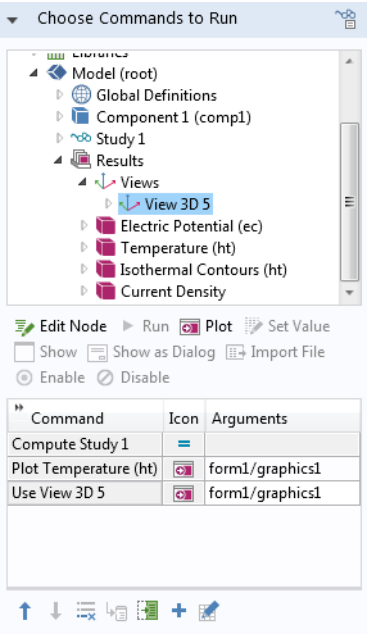


In this case you need to create a method for updating the view that is called to handle an event from the slider or form object. In the example above, the slider uses a **Local method** defined in the **Events** section. This method contains one line of code that updates the view:

```
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.



This line of code sets the URL source of the form object webpage1 to the COMSOL web page:

```
setWebPageSource("/form1/webpage1", "http://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).

GUI Command Methods

The GUI command methods correspond to the **GUI Commands** node in the editor tree. The editor tree is 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.
`exit`	`exit()`	Exits the application.
`fileOpen`	`fileOpen(String name)`	Opens a file with the associated program on the client.
`fileSaveAs`	`fileSaveAs(String file)`	Downloads a file to the client. See also the section "File Methods".
`printGraphics`	`printGraphics(String graphicsName)`	Prints the given graphics object.
`saveApplication`	`saveApplication()`	Saves the application.
`saveApplicationAs`	`saveApplicationAs()`	Saves the application under a different name. (Or as an MPH file.)
`scenelight`	`sceneLight(String graphicsName)`	Toggles scene light in the given graphics object.
`transparency`	`transparency(String graphicsName)`	Toggles transparency in the given graphics object.
`zoomExtents`	`zoomExtents(String graphicsName)`	Makes the entire model visible within the extent of the given graphics object.

Example Code

The line of code below saves a document to the `user` folder (as specified in the Preferences).

```
fileSaveAs("user:///mixer.docx");
```

The following code changes the camera position 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");
}
```

Debug Method

The debug method is used to display variable contents in the **Debug Log** window.

NAME	SYNTAX	DESCRIPTION
`debugLog`	`debugLog(arg)`	Prints the value of `arg` to the Debug Log window. The input argument `arg` can be a scalar, 1D 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]);
```

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.

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. (Only 32 bits are returned on a 32-bit platform.)
`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. (Only 32 bits are returned on a 32-bit platform.)
`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

Progress methods are used to create and update progress information in the Status bar, in a progress form object, and in a dialog box.

NAME	SYNTAX	DESCRIPTION
`setProgressInterval`	`setProgressInterval(String message, int intervalStart, int intervalEnd)`	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 `intervalEnd` must be between `intervalStart` and `100.` Calling this method implicitly resets any manual progress previously set by calls to `setProgress()`.
`setProgress`	`setProgress(int value, String message)`	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`	`setProgress(int value)`	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.
`showIndeterminateProgress`	`showIndeterminateProgress(String message)`	Shows a progress dialog box with an indeterminate progress bar, given `message` and a cancel button.
`showIndeterminateProgress`	`showIndeterminateProgress(String message, boolean cancelButton)`	Shows a progress dialog box with an indeterminate progress bar, given `message` and an optional cancel button.
`showProgress`	`showProgress()`	Shows a progress dialog box with a cancel button. No model progress is included.
`showProgress`	`showProgress(boolean modelProgress)`	Shows a progress dialog box with a cancel button and an optional model progress.
`showProgress`	`showProgress(boolean modelProgress, boolean addSecondLevel)`	Shows a progress dialog box 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 box 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 box.

NAME	SYNTAX	DESCRIPTION
`startProgress`	`startProgress(String name)`	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`	`setProgressBar(String name, int workDone, String message)`	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`	`setProgressBar(String name, int workDone)`	Same as above, but does not update the progress message.

Example Code

```
showProgress(true, true, true);
/* Opens a progress dialog box with cancel button showing two levels of
progress. The values shown in progress dialog box 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 0-20). */

// 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>
/* 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 box. 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 box, 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 box 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 COMSOL model 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 Box

A progress dialog box 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** check box 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 box. 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`.

Date and Time Methods

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`	`String currentDate()`	Returns the current date as a string (formatted according to the server's defaults) for the current date.
`currentTime`	`String currentTime()`	Returns the current time as a string (not including date, and formatted according to the server defaults).
`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`	`sleep(long timeInMs)`	Sleep for the specified number of milliseconds.
`timeStamp`	`long timeStamp()`	Current time in milliseconds since midnight, January 1, 1970 UTC.

NAME	SYNTAX	DESCRIPTION
`getExpectedComputationTime`	`model.setExpectedComputationTime(String format)`	Returns a string describing the approximate computation time of the application. The string can be altered by the method `setExpectedComputationTime`.
`setLastComputationTime`	`model.setLastComputationTime(long time)`	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).
`getLastComputationTime`	`String model.getLastComputationTime (String format)`	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("ms")`.

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*.

License Methods

The license method checks the license number for the current session. This can be used, for example, to limit the use of an application to one or a few license numbers.

NAME	SYNTAX	DESCRIPTION
`getLicenseNumber`	`String license = getLicenseNumber()`	Returns a string with the license number for the current session.

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'` returns 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`	`double[] toDouble(String... strs)`	Converts all the strings in the given arrray to doubles and returns a double array.
`toDouble`	`double[][] toDouble(String[]... strs)`	Converts all the strings in the given matrix to doubles and returns a double matrix.
`toInt`	`int toInt(String str)`	Converts the given string to an integer.
`toInt`	`int[] toInt(String... strs)`	Converts all the strings in the given arrray to integers and returns an integer array.
`toInt`	`int[][] toInt(String[]... strs)`	Converts all the strings in the given matrix to integers and returns an integer matrix.
`toString`	`String toString(int value)`	Converts the given integer to a string.
`toString`	`String toString(double value)`	Converts the given double to a string.

NAME	SYNTAX	DESCRIPTION
`toString`	`String toString(boolean value)`	Converts the given Boolean to a string.
`toString`	`String toString(double value, int decimals)`	Converts the given double to a string with the given number of decimals.
`toString`	`String toString(double value, int decimals,boolean remove)`	Converts the given double to a string with the given number of decimals with trailing zeros removed if the Boolean `remove` is `true`. For example, `10.0000001` with number of decimals 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`	`String[] toString(double[] darray)`	Converts all the doubles in the given array to strings and returns a string array.
`toString`	`String[][] toString(double[][] dmatrix)`	Converts all the doubles in the given matrix to strings and returns a string matrix.
`toString`	`String[] toString(int[] iarray)`	Converts all the integers in the given array to strings and returns a string array.
`toString`	`String[][] toString(int[][] imatrix)`	Converts all the integers in the given matrix to strings and returns a string matrix.
`toString`	`String[] toString(boolean[] barray)`	Converts all the Booleans in the given array to strings and returns a string array.
`toString`	`String[][] toString(boolean[][] bmatrix)`	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`	`String[] getColumn(String[][] matrix, int column)`	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`	`double[] getColumn(double[][] matrix, int column)`	Returns a `double[]` for a specified column in the matrix.
`getColumn`	`int[] getColumn(int[][] matrix, int column)`	Returns an int[] for a specified column in the matrix.
`getColumn`	`boolean[] getColumn(boolean[][] matrix, int column)`	Returns a boolean[] for a specified column in the matrix.
`getSubMatrix`	`String[][] getSubMatrix(String[][] matrix, int startCol, int endCol, int startRow, int endRow)`	Returns a rectangular submatrix of the input matrix spanning columns from startCol to endCol, and rows from startRow to endRow.
`getSubMatrix`	`double[][] getSubMatrix(double[][] matrix, int startCol,int endCol, int startRow, int endRow)`	Returns a rectangular submatrix of the input matrix spanning columns from startCol to endCol, and rows from startRow to endRow.
`getSubMatrix`	`int[][] getSubMatrix(int[][] matrix, int startCol, int endCol, int startRow, int endRow)`	Returns a rectangular submatrix of the input matrix spanning columns from startCol to endCol, and rows from startRow to endRow.
`getSubMatrix`	`boolean[][] getSubMatrix(boolean[][] matrix, int startCol, int endCol, int startRow, int endRow)`	Returns a rectangular submatrix of the input matrix spanning columns from startCol to endCol, and rows from startRow to endRow.
`insert`	`String[] insert(String[] array, String value, int index)`	Inserts an element at position index in an array and returns the expanded array.

NAME	SYNTAX	DESCRIPTION
insert	`double[] insert(double[] array, double value, int index)`	Inserts an element at position index in an array and returns the expanded array.
insert	`int[] insert(int[] array, int value, int index)`	Inserts an element at position index in an array and returns the expanded array.
insert	`boolean[] insert(boolean[] array, boolean value, int index)`	Inserts an element at position index in an array and returns the expanded array.
insert	`String[] insert(String[] array, String[] value, int[] index)`	Inserts elements in an array at positions given by the index array and returns the expanded array.
insert	`double[] insert(double[] array, double[] value, int[] index)`	Inserts elements in an array at positions given by the index array and returns the expanded array.
insert	`int[] insert(int[] array, int[] value, int[] index)`	Inserts elements in an array at positions given by the index array and returns the expanded array.
insert	`boolean[] insert(boolean[] array, boolean[] value, int[] index)`	Inserts elements in an array at positions given by the index array and returns the expanded array.
append	`String[] append(String[] array, String value)`	Adds an element to the end of an array and returns the expanded array.
append	`double[] append(double[] array, double value)`	Adds an element to the end of an array and returns the expanded array.
append	`int[] append(int[] array, int value)`	Adds an element to the end of an array and returns the expanded array.
append	`boolean[] append(boolean[] array, boolean value)`	Adds an element to the end of an array and returns the expanded array.
append	`String[] append(String[] array, String[] value)`	Adds elements to the end of an array and returns the expanded array.
append	`double[] append(double[] array, double[] value)`	Adds elements to the end of an array and returns the expanded array.

NAME	SYNTAX	DESCRIPTION
append	`int[] append(int[] array, int[] value)`	Adds elements to the end of an array and returns the expanded array.
append	`boolean[] append(boolean[] array, boolean[] value)`	Adds elements to the end of an array and returns the expanded array.
remove	`String[] remove(String[] array, int index)`	Removes an element from an array and returns the shortened array.
remove	`double[] remove(double[] array, int index)`	Removes an element from an array and returns the shortened array.
remove	`int[] remove(int[] array, int index)`	Removes an element from an array and returns the shortened array.
remove	`boolean[] remove(boolean[] array, int index)`	Removes an element from an array and returns the shortened array.
remove	`String[] remove(String[] array, int[] index)`	Removes elements from an array and returns the shortened array.
remove	`double[] remove(double[] array, int[] index)`	Removes elements from an array and returns the shortened array.
remove	`int[] remove(int[] array, int[] index)`	Removes elements from an array and returns the shortened array.
remove	`boolean[] remove(boolean[] array, int[] index)`	Removes elements from an array and returns the shortened array.
insertRow	`String[][] insertRow(String[][] matrix, String[] value, int rowIndex)`	Inserts a row into a rectangular 2D array and returns the expanded array.
insertRow	`double[][] insertRow(double[][] matrix, double[] value, int rowIndex)`	Inserts a row into a rectangular 2D array and returns the expanded array.
insertRow	`int[][] insertRow(int[][] matrix, int[] value, int rowIndex)`	Inserts a row into a rectangular 2D array and returns the expanded array.
insertRow	`boolean[][] insertRow(boolean[][] matrix, boolean[] value, int rowIndex)`	Inserts a row into a rectangular 2D array and returns the expanded array.

NAME	SYNTAX	DESCRIPTION
insertRow	`String[][] insertRow(String[][] matrix, String[][] value, int[] rowIndex)`	Adds rows to a rectangular 2D array and returns the expanded array.
insertRow	`double[][] insertRow(double[][] matrix, double[][] value, int[] rowIndex)`	Adds rows to a rectangular 2D array and returns the expanded array.
insertRow	`int[][] insertRow(int[][] matrix, int[][] value, int[] rowIndex)`	Adds rows to a rectangular 2D array and returns the expanded array.
insertRow	`boolean[][] insertRow(boolean[][] matrix, boolean[][] value, int[] rowIndex)`	Adds rows to a rectangular 2D array and returns the expanded array.
appendRow	`String[][] appendRow(String[][] matrix, String[] value)`	Adds a row to the end of a rectangular 2D array and returns the expanded array.
appendRow	`double[][] appendRow(double[][] matrix, double[] value)`	Adds a row to the end of a rectangular 2D array and returns the expanded array.
appendRow	`int[][] appendRow(int[][] matrix, int[] value)`	Adds a row to the end of a rectangular 2D array and returns the expanded array.
appendRow	`boolean[][] appendRow(boolean[][] matrix, boolean[] value)`	Adds a row to the end of a rectangular 2D array and returns the expanded array.
appendRow	`String[][] appendRow(String[][] matrix, String[][] value)`	Adds rows to the end of a rectangular 2D array and returns the expanded array.
appendRow	`double[][] appendRow(double[][] matrix, double[][] value)`	Adds rows to the end of a rectangular 2D array and returns the expanded array.
appendRow	`int[][] appendRow(int[][] matrix, int[][] value)`	Adds rows to the end of a rectangular 2D array and returns the expanded array.
appendRow	`boolean[][] appendRow(boolean[][] matrix, boolean[][] value)`	Adds rows to the end of a rectangular 2D array and returns the expanded array.
removeRow	`String[][] removeRow(String[][] matrix, int rowIndex)`	Removes a row from a 2D array and returns the smaller array.

NAME	SYNTAX	DESCRIPTION
removeRow	`double[][] removeRow(double[][] matrix, int rowIndex)`	Removes a row from a 2D array and returns the smaller array.
removeRow	`int[][] removeRow(int[][] matrix, int rowIndex)`	Removes a row from a 2D array and returns the smaller array.
removeRow	`boolean[][] removeRow(boolean[][] matrix, int rowIndex)`	Removes a row from a 2D array and returns the smaller array.
removeRow	`String[][] removeRow(String[][] matrix, int[] rowIndex)`	Removes rows from a 2D array and returns the reduced array.
removeRow	`double[][] removeRow(double[][] matrix, int[] rowIndex)`	Removes rows from a 2D array and returns the reduced array.
removeRow	`int[][] removeRow(int[][] matrix, int[] rowIndex)`	Removes rows from a 2D array and returns the reduced array.
removeRow	`boolean[][] removeRow(boolean[][] matrix, int[] rowIndex)`	Removes rows from a 2D array and returns the reduced array.
insertColumn	`String[][] insertColumn(String[][] matrix, String[] value, int columnIndex)`	Adds a column into a rectangular 2D array and returns the expanded array.
insertColumn	`double[][] insertColumn(double[][] matrix, double[] value, int columnIndex)`	Adds a column into a rectangular 2D array and returns the expanded array.
insertColumn	`int[][] insertColumn(int[][] matrix, int[] value, int columnIndex)`	Adds a column into a rectangular 2D array and returns the expanded array.
insertColumn	`boolean[][] insertColumn(boolean[][] matrix, boolean[] value, int columnIndex)`	Adds a column into a rectangular 2D array and returns the expanded array.
insertColumn	`String[][] insertColumn(String[][] matrix, String[][] value, int[] columnIndex)`	Adds columns to a rectangular 2D array and returns the expanded array.
insertColumn	`double[][] insertColumn(double[][] matrix, double[][] value, int[] columnIndex)`	Adds columns to a rectangular 2D array and returns the expanded array.
insertColumn	`int[][] insertColumn(int[][] matrix, int[][] value, int[] columnIndex)`	Adds columns to a rectangular 2D array and returns the expanded array.
insertColumn	`boolean[][] insertColumn(boolean[][] matrix, boolean[][] value, int[] columnIndex)`	Adds columns to a rectangular 2D array and returns the expanded array.

NAME	SYNTAX	DESCRIPTION
appendColumn	`String[][] appendColumn(String[][] matrix, String[] value)`	Adds a column at the end of a rectangular 2D array and returns the expanded array.
appendColumn	`double[][] appendColumn(double[][] matrix, double[] value)`	Adds a column at the end of a rectangular 2D array and returns the expanded array.
appendColumn	`int[][] appendColumn(int[][] matrix, int[] value)`	Adds a column at the end of a rectangular 2D array and returns the expanded array.
appendColumn	`boolean[][] appendColumn(boolean[][] matrix, boolean[] value)`	Adds a column at the end of a rectangular 2D array and returns the expanded array.
appendColumn	`String[][] appendColumn(String[][] matrix, String[][] value)`	Adds columns to the end of a rectangular 2D array and returns the expanded array.
appendColumn	`double[][] appendColumn(double[][] matrix, double[][] value)`	Adds columns to the end of a rectangular 2D array and returns the expanded array.
appendColumn	`int[][] appendColumn(int[][] matrix, int[][] value)`	Adds columns to the end of a rectangular 2D array and returns the expanded array.
appendColumn	`boolean[][] appendColumn(boolean[][] matrix, boolean[][] value)`	Adds columns to the end of a rectangular 2D array and returns the expanded array.
removeColumn	`String[][] removeColumn(String[][] matrix, int columnIndex)`	Removes a column from a rectangular 2D array and returns the smaller array.
removeColumn	`double[][] removeColumn(double[][] matrix, int columnIndex)`	Removes a column from a rectangular 2D array and returns the smaller array.
removeColumn	`int[][] removeColumn(int[][] matrix, int columnIndex)`	Removes a column from a rectangular 2D array and returns the smaller array.
removeColumn	`boolean[][] removeColumn(boolean[][] matrix, int columnIndex)`	Removes a column from a rectangular 2D array and returns the smaller array.
removeColumn	`String[][] removeColumn(String[][] matrix, int[] columnIndex)`	Removes columns from a rectangular 2D array and returns the reduced array.

NAME	SYNTAX	DESCRIPTION
`removeColumn`	`double[][] removeColumn(double[][] matrix, int[] columnIndex)`	Removes columns from a rectangular 2D array and returns the reduced array.
`removeColumn`	`int[][] removeColumn(int[][] matrix, int[] columnIndex)`	Removes columns from a rectangular 2D array and returns the reduced array.
`removeColumn`	`boolean[][] removeColumn(boolean[][] matrix, int[] columnIndex)`	Removes columns from a rectangular 2D array and returns the reduced array.
`matrixSize`	`int[] matrixSize(String[][] matrix)`	Returns the number of rows and columns of a matrix as an integer array of length 2.
`matrixSize`	`int[] matrixSize(double[][] matrix)`	Returns the number of rows and columns of a matrix as an integer array of length 2.
`matrixSize`	`int[] matrixSize(int[][] matrix)`	Returns the number of rows and columns of a matrix as an integer array of length 2.
`matrixSize`	`int[] matrixSize(boolean[][] matrix)`	Returns the number of rows and columns of a matrix as an integer array of length 2.

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`	`String[] concat(String colSepar, String rowSepar, String[]... matr)`	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.

NAME	SYNTAX	DESCRIPTION
`contains`	`boolean contains(String[] strs, String str)`	Returns true if the given string array `strs` contains the given string `str`.
`find`	`int[] find(String[] strs, String str)`	Returns an array with the indices to all occurrences of `str` in `strs`.
`findIn`	`int findIn(String[] strs, String str)`	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`	`int length(String str)`	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`	`String[] split(String str)`	Returns an array of strings by splitting the given string at spaces.
`split`	`String[] split(String str, String separator)`	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`	`String[] unique(String[] strs)`	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
`copy`	`String[] copy(String... toCopy)`	Returns a copy of the given array of strings, which can also be specified as a varargs of strings.
`copy`	`String[][] copy(String[]... toCopy)`	Returns a copy of the given string matrix, which can also be specified as a varargs of rows (string arrays).
`copy`	`double[] copy(double... toCopy)`	Returns a copy of the given array of doubles, which can also be specified as a varargs of doubles.
`copy`	`double[][] copy(double[]... toCopy)`	Returns a copy of the given double matrix, which can also be specified as a varargs of rows (double arrays).
`copy`	`int[] copy(int... toCopy)`	Returns a copy of the given array of integers, which can also be specified as a varargs of integers.
`copy`	`int[][] copy(int[]... toCopy)`	Returns a copy of the given integer matrix, which can also be specified as a varargs of rows (integer arrays).
`copy`	`boolean[] copy(boolean... toCopy)`	Returns a copy of the given array of booleans, which can also be specified as a varargs of booleans.
`copy`	`boolean[][] copy(boolean[]... toCopy)`	Returns a copy of the given boolean matrix, which can also be specified as a vararags of rows (boolean arrays).
`equals`	`boolean equals(String[] str1, String[] str2)`	Returns true if all strings in the given array are equal and they have the same number of elements.

NAME	SYNTAX	DESCRIPTION
`equals`	`boolean equals(String[][] matr1, String[][] matr2)`	Returns true if all strings in the given matrix 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 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`	`boolean equals(double[] dbls1, double[] dbls2, double relErrorTolerance)`	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`	`boolean equals(double[] dbls1, double[] dbls2)`	Same as above, but uses a default `relErrorTolerance` of `0.0001`.
`equals`	`boolean equals(double[][] dbls1, double[][] dbls2, double relErrorTolerance)`	Compares the relative errors (~ 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`	`boolean equals(double[][] dbls1, double[][] dbls2)`	Same as above, but uses a default `relErrorTolerance` of `0.0001`.
`sort`	`sort(String[] strs)`	Sorts the given array of strings. NOTE: The array is sorted in place.

NAME	SYNTAX	DESCRIPTION
`sort`	`sort(int[] ints)`	Sorts the given array of integers. NOTE: The array is sorted in place.
`sort`	`sort(double[] doubles)`	Sorts the given array of doubles. NOTE: The array is sorted in place.
`merge`	`merge(String[]... toMerge)`	Returns an array of strings with all strings merged from the given arrays.
`merge`	`merge(int[]... toMerge)`	Returns an array of integers with all integers merged from the two given arrays.
`merge`	`merge(double[]... toMerge)`	Returns an array of doubles with all doubles merged from the two given arrays.

Index