

COMSOL SCRIPT™

USER'S GUIDE Update

VERSION 1.0 a



How to contact COMSOL:

Benelux

COMSOL BV
Röntgenlaan 19
2719 DX Zoetermeer
The Netherlands
Phone: +31 (0) 79 363 4230
Fax: +31 (0) 79 361 4212
info@femlab.nl
www.femlab.nl

Denmark

COMSOL A/S
Diplomvej 376
2800 Kgs. Lyngby
Phone: +45 88 70 82 00
Fax: +45 88 70 80 90
info@comsol.dk
www.comsol.dk

Finland

COMSOL OY
Lauttasaarentie 52
FIN-00200 Helsinki
Phone: +358 9 2510 400
Fax: +358 9 2510 4010
info@comsol.fi
www.comsol.fi

France

COMSOL France
19 rue des bergers
F-38000 Grenoble
Phone: +33 (0)4 76 46 49 01
Fax: +33 (0)4 76 46 07 42
info@comsol.fr
www.comsol.fr

Germany

FEMLAB GmbH
Berliner Str. 4
D-37073 Göttingen
Phone: +49-551-99721-0
Fax: +49-551-99721-29
info@femlab.de
www.femlab.de

Italy

COMSOL S.r.l.
Contrada Santa Croce, 22
25125 Brescia
info.it@comsol.com

Norway

COMSOL AS
Verftsgata 4
NO-7485 Trondheim
Phone: +47 73 84 24 00
Fax: +47 73 84 24 01
info@comsol.no
www.comsol.no

Sweden

COMSOL AB
Tegnérsgatan 23
SE-111 40 Stockholm
Phone: +46 8 412 95 00
Fax: +46 8 412 95 10
info@comsol.se
www.comsol.se

Switzerland

FEMLAB GmbH
Technoparkstrasse 1
CH-8005 Zürich
Phone: +41 (0)44 445 2140
Fax: +41 (0)44 445 2141
info@femlab.ch
www.femlab.ch

United Kingdom

COMSOL Ltd.
Studio G8 Shepherds Building
Rockley Road
London W14 0DA
Phone: +44-(0)-20 7348 9000
Fax: +44-(0)-20 7348 9020
info.uk@comsol.com
www.uk.comsol.com

United States

COMSOL, Inc.
1 New England Executive Park
Suite 350
Burlington, MA 01803
Phone: +1-781-273-3322
Fax: +1-781-273-6603

COMSOL, Inc.
1100 Glendon Avenue, 17th Floor
Los Angeles, CA 90024
Phone: +1-310-689-7250
Fax: +1-310-689-7527

COMSOL, Inc.
744 Cowper Street
Palo Alto, CA 94301
Tel: +1-650-324-9935
Fax: +1-650-324-9936

info@comsol.com
www.comsol.com

For a complete list of international
representatives, visit
www.comsol.com/contact

Company home page

www.comsol.com

COMSOL user forums

www.comsol.com/support/forums

COMSOL Script User's Guide Update

© COPYRIGHT 1994–2005 by COMSOL AB. All rights reserved

Patent pending

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from COMSOL AB.

COMSOL, COMSOL Multiphysics, COMSOL Script, and COMSOL Reaction Engineering Lab are trademarks of COMSOL AB.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Version: December 2005 COMSOL 3.2a

User-Defined Classes

Using COMSOL Script it is possible to define new data types, classes, and to create objects that are instances of these classes. A class is an aggregate of fields and methods.

The class model used is inspired by Java's class system.

This document describes this new functionality in COMSOL Script 1.0a

Introductory Example: Rectangle

The file `Rectangle.cs1` defines the class `Rectangle`:

```
class Rectangle
%A class for rectangles.

public x0 x1 % x0<x1
public y0 y1 % y0<y1

function Rectangle(varargin)
%RECTANGLE(X0, Y0, X1, Y1) creates a rectangle with vertices
%in (x0,y0) and (x1,y1).
switch nargin
case 4
    in = varargin;
    [x0 x1] = deal(min(in{1}, in{3}), max(in{1}, in{3}));
    [y0 y1] = deal(min(in{2}, in{4}), max(in{2}, in{4}));
case 1
    r = varargin{1};
    [x0 x1] = [r.x0 r.x1];
    [y0 y1] = [r.y0 r.y1];
otherwise
    error('Usage: Rectangle(x0, y0, x1, y1)');
end

public function out = area
%AREA returns the area of the rectangle.
out = (x1-x0)*(y1-y0);

public static function out = overlap(r1, r2)
%OVERLAP(R1, R2) returns the area of the overlap between the
%two rectangles R1 and R2.
out = max(min(r1.x1, r2.x1)-max(r1.x0, r2.x0), 0)*...
    max(min(r1.y1, r2.y1)-max(r1.y0, r2.y0), 0);
```

This defines a `Rectangle` *class* with the four *fields* `x0`, `y0`, `x1`, and `y1`, a *constructor*, and the two *methods* `area` and `overlap`. To create a `Rectangle` *object*, use the same syntax as for a function call:

```
r = Rectangle(1, 3, 2, 9)
r =
Rectangle object
x0: [1]
x1: [2]
y0: [3]
y1: [9]
```

```
r.area
```

```
ans =  
6
```

The call `Rectangle(1, 3, 2, 9)` returns a rectangle with vertices in (1,3) and (2,9). The variable `r` is of the type `Rectangle`; it is an *instance* of the `Rectangle` class. There can only be one definition of a class, but there can be many instances of it. By default, the fields of an object are displayed in the same way as if it were a structure.

The call `r.area` invokes the `area` method of the object `r`. The syntax for invoking a method without arguments is the same as for accessing a field of a structure.

The class contains three methods: the constructor `Rectangle`, the accessor `area`, and the static function `overlap`. The method declarations look like declarations of local functions in a function M-file, but unlike local functions, they can be accessed outside the class.

`overlap` is an example of a static method: it is invoked when `overlap` is invoked with arguments where at least one is a `Rectangle`:

```
r = Rectangle(1, 5, 6, 8);  
s = Rectangle(3, 2, 5, 7);  
overlap(r, s)  
  
ans =  
4
```

The `overlap` method is not visible if none of the arguments is a `Rectangle`: for example, `overlap(1,2)` gives an error.

The Structure of the Class File

A class is defined by a `.class`-file on the path that has the following contents:

- Class header
- Precedence declarations
- Field declarations
- Method declarations

Only the class header is mandatory. Below you find brief descriptions of the contents of each part of the file. The next section contains a more detailed description.

Class Header

The class header declares the name of the class, its copy semantics, and its superclass, if any. The most basic form is a value class with no superclass:

```
class Rectangle
```

The name of the class, here `Rectangle`, must coincide with the filename, here `Rectangle.class`. That the class is a value class means that a copy is made whenever an instance of the class is used as argument to a function, or is used in an assignment. All other data types except for Java objects have value semantics. The opposite is a reference class, declared with the `reference` modifier:

```
reference class Rectangle
```

Reference semantics means that a true copy of the object is never made, only references to the same object. This is the semantics that Java objects have.

The superclass of a class is declared using the `extends` keyword:

```
class Rectangle extends Shape
```

The superclass must be defined by a `.class`-file on the path.

Precedence Declarations

This optional section contains the classes that have higher or lower precedence than the class being declared. It contains zero or more lines of the form

```
superiorto <classname>
```

or

`inferiorto <classname>`

Field Declarations

This optional section contains zero or more field declarations of the forms

```
<access modifiers> <name> = <expression>
```

or

```
<access modifiers> <name1> [<name2> ...]
```

The fields declared are like the fields of a structure, with the difference that a class always contains the same fields. The access modifiers control where the field is visible. You must specify one of `public`, `protected`, and `private`, and you can optionally use `static` and `transient`.

The first syntax allows for initial values to be supplied:

```
public version = 1;
```

This declares the field `version` with the default value `1`. This means that the `version` field is assigned to `1` when an instance of the class is created. Fields without initial values are assigned to `[]`.

Method Declarations

This optional section contains zero or more method declarations of the form

```
[<access modifiers>] <function declaration>
```

You can specify one of the access modifiers `public`, `protected`, and `private`, as well as `static`. The method is considered `public` unless you specify `protected` or `private`.

Except for the optional access modifiers, the syntax of a function declaration is the same as for a local function declaration in an M-file.

Access Modifiers

You can assign access modifiers to the fields and methods of a class. There are three types of modifiers:

- Visibility modifiers: `public`, `protected`, and `private`. They specify where the member is visible: `public` means that the member is visible everywhere, `protected` that it is visible in the class and classes inheriting from it, and `private` that it is visible only in the class where it was declared.
- Static modifier: `static`. A static member is one that is associated with the class, not with an instance of the class. The opposite is an instance member (the default).
- Transient modifier: `transient`. COMSOL Script does not save a field marked as transient when you run the command `save`. You can use this to avoid saving temporary data that is easy to recompute.

Each member field must have a visibility modifier specified. For methods this is not necessary; if omitted, the default visibility is `public`.

A class where all fields are `public` behaves a lot like a structure: It is possible to read and assign values to all fields using the `var.field` syntax. Restricting the visibility can be useful for hiding class internals that are inconsequential for the user of the class.

Member Fields

A class can be viewed as a structure with a predefined set of fields, the instance (non-static) fields. Static fields have semantics similar to `global` or `persistent` variables.

Instance Fields

Fields not declared as static are instance fields: They belong to an instance of the class, like fields in a structure. The fields of the `Rectangle` class are examples of instance fields:

```
class Rectangle

public x0 x1 % x0<x1
public y0 y1 % y0<y1
```

Outside of the class, you can use these fields like the fields of a structure:

```
r = Rectangle(2, 3, 6, 7);
r.x1

ans =
6
```

This is only possible for `public` fields; you cannot access `protected` and `private` fields this way. It is also possible to modify public fields like fields of a structure:

```
r = Rectangle(2, 3, 6, 7);
r.y1 = 10

r =
Rectangle object
x0: [2]
x1: [6]
y0: [3]
y1: [10]
```

When a non-static method of a class is run, you can access its instance fields like local variables:

```
public function out = area
%AREA returns the area of the rectangle.
out = (x1-x0)*(y1-y0);
```

The values of `x0`, `x1`, `y0`, and `y1` are taken from the instance fields, and any assignments to them would result in the instance fields being changed.

Static Fields

Fields declared `static` belong to the class, not any instance of it. As an example, consider the class `MathConst`:

```
class MathConst

% The golden ratio
public static golden = (sqrt(5)+1)/2;
% Perfect numbers less than 10^11
public static perfect = [6 28 496 8128 33550336 8589869056];
```

The two fields `golden` and `perfect` are static. Their values are set using initializers, (see the next section for an explanation of this). To access them, access the class like a structure:

```
MathConst.golden

ans =
1.6180

MathConst.perfect(2:3)

ans =
28 496
```

It is also possible to change their values:

```
MathConst.golden = -17
```

Initialization of Fields

The `MathConst` class illustrates initialization:

```
public static golden = (sqrt(5)+1)/2;
```

The above means that when the class is loaded, the static field `golden` is assigned the expression in the right-hand side. For instance fields, initializers provide initial values when the object is created. As an example, consider the point class:

```
class Point

public x = 2;
public y = 5;
```

The default placement of the point is at coordinates (2, 5):

```
p = Point
    p =
    Point object
      x: [2]
      y: [5]
```

If a field is not given an initializer, it is assigned to []. More specifically: When an object is created, all instance fields are initialized to []. Then initializers, if any, are evaluated in the order the fields are declared. This means that

```
public x;
public y = {x 5};
public z = length(y);
```

is equivalent to

```
public x = [];
public y = {[] 5};
public z = 2;
```

Member Methods

Constructor

The constructor is a function with the same name as the class and is called when an instance of the class is created. The purpose of the constructor is to set up a valid object, possibly using input arguments. It cannot be static and must not return anything. As an example, consider the constructor of the `Rectangle` class:

```
function Rectangle(varargin)
%RECTANGLE(X0, Y0, X1, X1) creates a rectangle with vertices
%in (x0,y0) and (x1,y1).
switch nargin
case 4
    in = varargin;
    [x0 x1] = deal(min(in{1}, in{3}), max(in{1}, in{3}));
    [y0 y1] = deal(min(in{2}, in{4}), max(in{2}, in{4}));
    ...
```

The constructor is called whenever a `Rectangle` is created: When

```
r = Rectangle(1, 3, 2, 9)
```

is executed, an empty `Rectangle` object is created. At this point, the four instance fields `x0`, `x1`, `y0`, and `y1` have been initialized with `[]`. Then the constructor is invoked for the new object. The constructor is an instance method and can therefore modify the instance fields: The assignments to, for example, `x0` modify the field `x0` of the instance. The methods of a class differ from local functions in this respect: A local function `Rectangle` with the above contents would assign values to `x0` in its own workspace, but when leaving the function, these values would be lost.

A class does not need a constructor: If there is no constructor, the fields are assigned default values if provided, otherwise `[]`.

Instance Methods

An instance method is a method that is not declared static. It operates on an instance of a class and has access to the instance fields. The `area` method in the `Rectangle` class is an example of an instance method:

```
public function out = area
%AREA returns the area of the rectangle.
out = (x1-x0)*(y1-y0);
```

An instance method can read or write the instance fields like local variables, just like the constructor. `area` uses the coordinates of the rectangle to compute its area.

Static Methods

A static method has access to the static fields of the class but not to any instance fields as it cannot be run for any instance of the class. The `overlap` method of the `Rectangle` class is static:

```
public static function out = overlap(r1, r2)
%OVERLAP(R1, R2) returns the area of the overlap between the
%two rectangles R1 and R2.
out = max(min(r1.x1, r2.x1)-max(r1.x0, r2.x0), 0)*...
      max(min(r1.y1, r2.y1)-max(r1.y0, r2.y0), 0);
```

There are two way to invoke a static method:

- Calling it like a function with one or more objects as arguments:

```
r = Rectangle(1, 5, 6, 8);
s = Rectangle(3, 2, 5, 7);
overlap(r, s)
```

```
ans =
4
```

- Specifying it explicitly:

```
Rectangle.overlap(r, s)
```

```
ans =
4
```

The first syntax works because at least one of the arguments to `overlap` is an object, here of the class `Rectangle`, and the class `Rectangle` has a visible static method called `overlap`.

Inheritance

Sometimes a class is a specialization or extension of another class. You can specify this relation in the class header using the `extends` keyword:

```
class Rectangle extends Shape
```

A rectangle is a specialization of the shape concept, and it therefore makes sense to let the class `Rectangle` inherit from the class `Shape`, which then becomes the *superclass* of `Rectangle`, the *derived class*. The members and methods of the superclass are visible in the derived class. Suppose that the `Shape` and `Rectangle` classes contain the following fields:

```
class Shape
public name
...

class Rectangle extends Shape
public x0 x1
public y0 y1
...
```

The `Rectangle` class contains five fields: The four fields declared in `Rectangle`, and the field declared in the superclass, `Shape`. For a user of the class, there is no distinction between the two types of fields:

```
r = Rectangle(1, 3, 2, 9);
r.name = 'COMSOL';
r.name

ans =
    COMSOL
```

Reference and Value Classes

Differences

When you make an assignment `a = b`, COMSOL Script normally assigns a copy of `b` to `a`. This is the case for all data types except for Java objects and instances of reference classes: For these data types, only a reference is copied. Consider the class `RefClass` defined as follows:

```
reference class RefClass
public x = 5;
```

The software only copies a reference when an assignment is made:

```
r = RefClass;
s = r;
r.x = 10;
s.x

ans =
10
```

Consider on the other hand the class `ValueClass`, defined as follows:

```
class ValueClass
public x = 5;
```

For a value class, the assignment makes a true copy:

```
v = ValueClass;
s = v;
v.x = 10;
s.x

ans =
5
```

The same rules apply when passing arguments to functions: For an instance of a value class, a copy of the value is passed; for an instance of a reference class, a reference to the value is passed.

Choosing Class Type

The main factor determining the class type is how you view the class:

- If you think of the class as a structure augmented with methods, declare it as a value class.
- If you think of the class as a lightweight Java class, declare it as a reference class.

A reference class cannot inherit from a value class, or vice versa. Thus the choice of class type for a base class affects the entire class hierarchy.

If you define several different classes, try to use the same class type for all of them. Mixing class types can easily lead to bugs as the difference in semantics is subtle.

Built-In Functions

Functions That You Can Only Use for Objects

THIS

Inside an instance method, `this` returns the instance for which the method is run.

SUPER

When a constructor is run in a derived class, you can use `super` to run the constructor of the superclass. Suppose `Rectangle` inherits from `Shape`:

```
class Rectangle extends Shape
...
function Rectangle(varargin)
  super(varargin);
...
```

The call `super(varargin)` runs the constructor of the `Shape` class.

It is also possible to use `super` as a structure to access or modify visible fields or methods from the superclass: `super.a = 17`; sets the field `a` in the superclass to 17. This can be useful if a member in the superclass has been shadowed by a member with the same name in the derived class.

CLONE

The `clone` function creates a true copy of an object. For a value object, this has no effect, but for a reference object it is necessary in order to copy the contents, not only a reference to the object. The class `RefClass` was above defined as

```
reference class RefClass
  public x = 5;
```

Use the `clone` function to create a true copy of a `RefClass` object:

```
r = RefClass;
s = clone(r);
r.x = 10;
s.x

ans =
5
```

Without `clone`, only a reference would be copied.

Functions with Special Semantics for Objects

CLEAR CLASSES

`clear classes` removes all variables, just like `clear` does, but it also tries to remove all class definitions. This is necessary if the definition of a class changes: Unless you perform `clear classes`, COMSOL Script uses the old class definition even if the class file on disk changes.

`clear classes` can only remove the definitions of classes of which there are no instances. This means that sometimes not all classes are removed: If `clear classes` is called from a function and there still are instances of some class in the root workspace, then that class cannot be cleared. The same thing can happen if you store objects in global or persistent variables.

HELP

`help` for a class displays the comment block following the class header as well as the comment blocks following each public non-static method:

```
help Rectangle
```

```
A class for rectangles.
```

```
RECTANGLE(X0, Y0, X1, Y1) creates a rectangle with vertices  
in (x0,y0) and (x1,y1).
```

```
AREA returns the area of the rectangle.
```

```
OVERLAP(R1, R2) returns the area of the overlap between the  
two rectangles R1 and R2.
```

You can also retrieve the help text for a method:

```
help Rectangle.overlap
```

```
OVERLAP(R1, R2) returns the area of the overlap between the  
two rectangles R1 and R2.
```

FIELDNAMES

For an object, `fieldnames` returns the names of the instance fields that are visible from the workspace where `fieldnames` is called:

```
r = Rectangle(2, 3, 6, 7);
```

```
fieldnames(r)

ans =
    'x0'
    'x1'
    'y0'
    'y1'
```

All the fields of the `Rectangle` class are `public`, and `fieldnames` therefore returns them. `private` and `protected` members are only returned by `fieldnames` when invoked from a method of the class.

METHODS

`methods` displays the methods declared by a class:

```
methods('Rectangle')

class Rectangle
    public function Rectangle
    public function area
```

By default, `methods` only displays public non-static methods. It is possible to select methods to display based on access modifiers:

```
methods('Rectangle', 'static')

class Rectangle
    public static function overlap
```

The second argument to `methods` is a string or cell array of strings that lists all access modifiers to include.

When requesting output, `methods` returns a cell array:

```
c = methods('Rectangle', 'static')

c =
    {'overlap'}
```

STRUCT

When invoked for an object, `struct` returns a structure containing the fields of the object that are visible in the workspace where `struct` is called:

```
r = Rectangle(1, 3, 2, 9);
struct(r)

ans =
    x0: [1]
    x1: [2]
```

```
y0: [3]  
y1: [9]
```

Overloading

Overloading Operators

You can overload all unary and binary arithmetic, relational, and logical operators, as well as the concatenation operators [,] and [;]. To overload an operator, create a public static function that defines the semantics of the overloaded function. As an example, consider a class `Rational` that represents rational numbers:

```
class Rational
%RATIONAL is a class for exact representation of a rational number
%as a ratio between integers.

private a % Numerator
private b % Denominator, always > 0

...

static function out = plus(r1, r2)
%PLUS Sum.
% OUT = PLUS(R1, R2) returns the sum of R1 and R2.
r1 = Rational(r1);
r2 = Rational(r2);
out = Rational(r1.a*r2.b+r1.b*r2.a, r1.b*r2.b);

...
```

The static function `plus` provides an overload for the `+` operator:

```
Rational(1,3)+Rational(1/4) % 1/3+1/4 = 7/12
7 / 12
Rational(pi)+1 % uses 355/113 as approximation of pi
468/113
```

A demonstration example in this release includes the complete definition of the `Rational` class. Run `help Rational` or type `Rational` to see its contents.

Each operator has a corresponding function that the software invokes when at least one of the operands is an object. This is the function that the class must provide for it to define an overloaded operator. The example above includes overloading of the `+`

operator by defining the `plus` member method. Table 1-1 contains the complete map between operators and functions:

TABLE 1-1: OPERATORS AND THEIR CORRESPONDING FUNCTIONS

OPERATOR	FUNCTION
+ (unary, +a)	<code>uplus</code>
+ (binary, a+b)	<code>plus</code>
- (unary, -a)	<code>uminus</code>
- (binary, a-b)	<code>minus</code>
*	<code>mtimes</code>
<code>.*</code>	<code>times</code>
/	<code>mrdivide</code>
<code>./</code>	<code>rdivide</code>
<code>\</code>	<code>mldivide</code>
<code>.\</code>	<code>ldivide</code>
<code>^</code>	<code>mpower</code>
<code>.^</code>	<code>power</code>
<code>==</code>	<code>eq</code>
<code>~=</code>	<code>ne</code>
<code>>=</code>	<code>ge</code>
<code>></code>	<code>gt</code>
<code><=</code>	<code>lt</code>
<code><</code>	<code>le</code>
<code>&</code>	<code>and</code>
<code> </code>	<code>or</code>
<code>~</code>	<code>not</code>
<code>[, ,]</code>	<code>horzcat</code>
<code>[; ;]</code>	<code>vertcat</code>
<code>:</code>	<code>colon</code>
<code>'</code>	<code>ctranspose</code>
<code>.'</code>	<code>transpose</code>

FIELD READ/WRITE

If a class has an instance method called `fieldread`, then COMSOL Script calls that function when it reads a field of an object using the `var.field` syntax. The `fieldread` method must take one input argument and return one output:

```
function out = fieldread(field)
```

The `field` argument is the string that follows the `.` (dot) in `var.field`. By overloading `fieldread` it is possible to let a class behave as if it has fields that it does not have. You can use this overloading for making the representation of the data independent of the interface to the class.

Similarly, if a class has an instance function called `fieldwrite`, then that function is called when a field of an object is written using the `var.field = val` syntax. It must take two input arguments and not return anything:

```
function fieldwrite(field, val)
```

The `field` parameter is the string that followed the `.` in `var.field` and the `val` parameter is the value to which the field was assigned. You can use `fieldwrite` when the fields of the class have dependencies: Changing the value of one field can force the recalculation of others.

ARRAY READ/WRITE

An object can only be indexed using parentheses if it has an instance method called `arrayread`: If `c` is an object, then `c(args)` is interpreted as `c.arrayread(args)`, where `arrayread` is invoked for the arguments (one or more) and is expected to return one value:

```
function out = arrayread(args)
```

Overloading `arrayread` can be useful for classes where the parentheses denote evaluation, for instance in a class representing a mathematical function of one or more variables.

Similarly, array assignment using the syntax `c(args) = val` is interpreted as `c.arraywrite(args, val)`.

CELL ARRAY READ/WRITE

It is only possible to index an object using curly brackets if it has an instance method called `cellread`: If `c` is an object, then `c{args}` is interpreted as `c.cellread(args)`

where `cellread` is invoked for the arguments (one or more) and is expected to return one value:

```
function out = cellread(args)
```

Similarly, cell array assignment using the syntax `c{args} = val` is interpreted as `c.cellwrite(args, val)`.

Overloading Save and Load

When an object is saved to file using `save`, the default behavior is to save all non-transient instance fields. This can be over-ridden by a class providing a `writeobject` method. This method must have the interface

```
public function out = writeobject
```

The output from `writeobject` is written to file when the object is saved.

If the class has an overloaded `writeobject` method it usually also has to provide an overloaded `readobject` method: This method is called when a object is loaded from file using `load`. It must have the interface

```
public function readobject(data)
```

When the software loads an object, it first creates an empty instance of its class. Then the `readobject` method is invoked, and the data argument is the output from `writeobject` when the object was saved.

It is possible to overload `readobject` but not `writeobject`. In this case, the input to `readobject` is the default representation of the object: As a cell array with one column for each level of the class hierarchy. Consider the `Rectangle` class inheriting from `Shape`:

```
class Shape
public name
...

class Rectangle extends Shape
public x0 x1
public y0 y1
```

A `Rectangle` object with vertices in (2, 3) and (5, 7) and the name 'MyRect' would be saved as the cell array

```
{ 'Shape' 'Rectangle'
  struct('name', 'MyRect') struct('x0',2,'x1',5,'y0',3,'y1',7)}
```

This would be the argument to an overloaded `readobject` in `Rectangle` if no overloaded `writeobject` was present. There is one column for each level of the class hierarchy. The first row contains the name class names, and the second row contains a structure with one field for each non-transient field on that level of the hierarchy.

Overloading Display

The default way to display an object is to display its public fields like the fields of a structure:

```
r = Rectangle(1, 3, 2, 9)

r =
Rectangle object
  x0: [1]
  x1: [2]
  y0: [3]
  y1: [9]
```

If the class has a public non-static function called `display`, the software invokes it when it displays an object of the class. You can extend the `Rectangle` class with such a method:

```
public function display
  sprintf('Rectangle with corners (%.2f,%.2f) and (%.2f,%.2f).', ...
         x0, y0, x1, y1)
```

This results in the following output:

```
r = Rectangle(1, 3, 2, 9)

ans =
Rectangle with corners (1.00, 3.00) and (2.00, 9.00).
```

Precedence

Precedence Between Functions and Methods

Methods in different classes can have the same name, and method names can also coincide with names of functions, both built-in functions and M-files. COMSOL Script resolves a function invocation of the form `func(arg1, ...)` by considering possible interpretations in the following order:

- 1 As an array index expression, if there is a variable called `func` in the workspace.
- 2 As the invocation of the static method `func` of the class of any object in the argument list.
- 3 As a call to the built-in function `func`.
- 4 As a call to the method `func` in the class where execution currently takes place.
- 5 As a call to the user-defined function `func`.
- 6 Interpretations 3–5 tried using case-insensitive name matching.

The `Rectangle` class contains a static method called `overlap`. Suppose that there also is an M-file, `overlap.m`. By rule 2 above,

```
overlap(Rectangle(1,2,3,4), Rectangle(5,6,7,8))
```

invokes the `overlap` method of the `Rectangle` class, but

```
overlap(5, 7)
```

instead calls `overlap.m`.

Precedence Between Methods from Different Classes

When a function call contains several object arguments, the software normally considers classes in the order they appear in the argument list: Suppose that the classes `Rectangle` and `Circle` both have a static `overlap` method. Then

```
overlap(Rectangle(2, 3, 4, 5), Circle(1, 2, 3))
```

is interpreted as

```
Rectangle.overlap(Rectangle(2, 3, 4, 5), Circle(1, 2, 3))
```

The `overlap` method of the `Circle` class would only be run if there were no `overlap` method in `Rectangle`.

In the example above, the two classes `Rectangle` and `Circle` have equal precedence. You can specify the precedence between classes using `superiorto` and `inferiorto` in the class file. The classes listed in a `superiorto` declaration have lower priority than the current class, those listed in an `inferiorto` declaration have higher priority.

The `Circle` class can be modified as follows:

```
class Circle
    superiorto Rectangle
```

With this change, the methods of the `Circle` class are always given priority over methods in the `Rectangle` class when the argument list contains `Circle` and `Rectangle` objects. Then

```
overlap(Rectangle(2, 3, 4, 5), Circle(1, 2, 3))
```

is interpreted as

```
Circle.overlap(Rectangle(2, 3, 4, 5), Circle(1, 2, 3))
```

It is possible to combine several `superiorto` and `inferiorto` declarations in the same class:

```
class Circle
    superiorto Rectangle Square
    inferiorto Hexagon
```

Note: Excessive use of `superiorto` and `inferiorto` makes the program flow hard to follow.

Using Classes as Packages

You can use classes to bundle groups of functions together: A class without instance fields where all methods are `public` and `static` can serve as a container for a set of related functions. As an example, consider the following class:

```
class MyMaths

    static function y = sinc(x)
    %SINC(X) returns SIN(X)/X if X is nonzero, otherwise 1.
    y = ones(size(x));
    ix = find(x~=0);
    y(ix) = sin(x(ix))./x(ix);

    static function y = smhs(x, scale)
    %SMHS(x) approximates the step function Y=(X>0) by smoothing the
    %transition within the interval -SCALE < X < SCALE.
    x=x./scale;
    y=(x>-1 & x<1).*(0.5+x.*(0.75-0.25*x.*x))+(x>=1);
```

You can view `MyMaths` as a package that contains the two functions `sinc` and `smhs`:

```
MyMaths.sinc(0:3)

ans =
     1     0.8415     0.4546     0.0470
```

By creating a class containing a group of related functions, preferably small, it becomes easier to maintain the functions, as only one file is ever changed. Sharing code between functions also becomes easier.