

Model Functions

Model functions let a script inspect and change the page model: create components and blocks, remove them, read bindings, copy models to and from strings, and reorder components. For the basics of the page model (@, @parent, @stage and so on), see [Page Model](#).

Most functions take a *model object* such as `@parent.@shape1` as the first parameter. If you pass something that is not a model object, the function does nothing and returns `null`.

Function	Purpose
modelLoad(map)	Create a model object from a map
modelSave(model)	Save a model object to a map
modelToString(modelOrMap)	Save a model or map to a string
modelFromString(string)	Parse a model string into a map
modelGetType(model)	Type name of a model object
modelsComponent(model)	Is the object a component (not a block)
modelGetFields(model)	Names of the fields that are set or bound
modelSetValue(model, field, value)	Set a field value
modelGetBinding(model, field)	Binding path of a field
modelHasBindingTarget(model, field)	Is a field bound to something
modelFindPath(target, base, [field])	Relative path between two objects
modelFindParent(model, type, ...)	Find an ancestor by type or values
modelIterateChildren(model, callback, [runOnSelf])	Walk all children
modelAddComponent(parent, model, name)	Add a component to a container
modelAddObject(parent, model, name)	Add a block or object to a parent
modelRemoveComponent(model, [recycle])	Remove a component
modelRemoveObject(model, [recycle], [clearBinding])	Remove a block or object
modelDestroy(model)	Remove and destroy an object
modelsDestroyed(model)	Has the object been destroyed
modelGetRepeaterChildren(repeater)	Components created by a repeater
modelMoveForward / modelMoveBackward / modelMoveToFront / modelMoveToBack	Change stacking order
modelAddArray(array, value, [index])	Add to a model array
modelRemoveArray(array, index)	Remove from a model array
dgGetViewValue / dgSetViewValue	Read and write view-only values
paramGetEnum / paramSetEnum	Read and change symbol parameter enums

Saving and loading models

modelLoad

```
modelLoad(map)
modelLoad(map, "page")
```

Creates a new model object from a map in the page file format. The map must have a `@type` key. Other keys set field values. The new object is not part of the page yet; add it with [modelAddComponent](#) or [modelAddObject](#).

With the second parameter "page", the map is loaded as a whole page instead. That page is disabled and is destroyed automatically shortly afterward, so use it only to read values from it right away.

Returns the new model object, or `null` if the parameter is not a map.

```
var block = modelLoad({"@type": "script", "script": "return 6 * 7;"});
return modelGetType(block);
```

Output: script

For editor components (textEditor, enumEditor, boolEditor, numberSliderEditor, percentSliderEditor, colorEditor, fillEditor, gradientListEditor), target and field get default values if you leave them out.

modelSave

```
modelSave(model)
```

Returns the model object as a map, in the same format that `modelLoad` accepts. Use it to copy a component, or to inspect its settings.

```
var block = modelLoad({"@type": "script", "script": "return 1;"});
var data = modelSave(block);
return data["@type"] + ": " + data.script;
```

Output: script: return 1;

modelToString

`modelToString(modelOrMap)`

Saves a model object, or a map, as a JSON string. Keys keep their original order.

```
return modelToString({"b": 2, "a": 1});
```

Output: {"b":2,"a":1}

modelFromString

`modelFromString(string)`

Parses a string made by `modelToString` (or the content of a .dg5 file) into a map. Pass the result to `modelLoad` to create a model object.

```
var m = modelFromString('{"@type": "script", "script": "return 2;"}');  
return m["@type"];
```

Output: script

Example: duplicate a component

This example copies `@parent.@shape1` and adds the copy to the same container. If `shape1` is taken, a new name is chosen.

```
var src = @parent.@shape1;  
var copy = modelLoad(modelSave(src));  
modelAddComponent(src.@parent, copy, "shape1");
```

Inspecting models

modelGetType

`modelGetType(model)`

Returns the type name of a model object, for example `script` for a Script block or `shape` for a shape. Returns null if the parameter is not a model object.

```
return modelGetType(@);
```

Output: script

modelIsComponent

`modelIsComponent(model)`

Returns `true` if the object is a component (something you can place on the stage) and `false` for dataflow blocks and other objects.

```
return modelIsComponent(@);
```

Output: false

modelGetFields

`modelGetFields(model)`

Returns a list of field names that have a value set or a binding. For blocks with custom parameters (such as a Script block), the custom parameters are always included. Fields still at their default value are not listed.

```
var block = modelLoad({"@type": "script", "script": "return 1;"});  
return modelGetFields(block);
```

Output: [script]

modelSetValue

`modelSetValue(model, field, value)`

Sets a field. This is the same as `model.field = value`, but useful when the field name is in a variable. Lists and maps are copied before they are set. Returns `null`.

```
var block = modelLoad({"@type": "script"});  
modelSetValue(block, "script", "return 3;");  
return block.script;
```

Output: return 3;

modelGetBinding

`modelGetBinding(model, field)`

Returns the binding path of a field as a string, for example `@parent.@table1.data`. Returns `null` if the field is not bound.

```
var path = modelGetBinding(@parent.@shape1, "fill");
```

modelHasBindingTarget

`modelHasBindingTarget(model, field)`

Returns `true` if some other field is bound to this field (this field is the source of a binding), otherwise `false`.

```
if (!modelHasBindingTarget(@parent.@table1, "data")) {  
    print("nothing uses the table");  
}
```

modelFindPath

`modelFindPath(target, base, [field])`

Returns the path from `base` to `target` as a string, in the form you would use in a binding or script, for example `@parent.@group1.@shape1`. If `field` is given, it is appended. Returns `null` if no path can be found.

```
var p = modelFindPath(@parent.@group1.@shape1, @, "width");  
// p is "@parent.@group1.@shape1.width"
```

modelFindParent

`modelFindParent(model, type, [values], [checkSelf], [crossStage])`

Walks up the model from `model` and returns the first ancestor that matches, or `null`.

- `type` - type name to match, or `null` to match any type.
- `values` - optional map of field values that must all match, for example `{"name": "main"}`.
- `checkSelf` - if `true`, `model` itself is checked first. Default `false`.

- `crossStage` - if `true`, the search continues out of symbols into the page that uses them (it follows `@parent` instead of the direct parent). Default `false`.

You must give a type or values.

```
var group = modelFindParent(@, "group");
var tagged = modelFindParent(@, null, {"tooltip": "panel"});
```

modelIterateChildren

`modelIterateChildren(model, callback, [runOnSelf])`

Calls `callback(child)` for the children of `model`. The callback must return `true` to continue into that child's own children; any other return value skips them. If `runOnSelf` is `true`, the callback is also called for `model` itself first (and returning anything but `true` stops the whole walk). Default `false`.

```
var count = 0;
modelIterateChildren(@stage, function(child) {
    if (modelGetType(child) == "shape") {
        count++;
    }
    return true;
});
return count;
```

Adding and removing models

modelAddComponent

`modelAddComponent(parent, model, name)`

Adds a component to a container (the stage, a group, ...) under the given name, and shows it. If the name is taken, a new name is chosen. Returns the added component, or `null`. All three parameters are required.

```
var shape = modelLoad({"@type": "shape", "left": 10, "top": 10, "width": 100,
    "height": 50});
modelAddComponent(@stage, shape, "box");
```

modelAddObject

`modelAddObject(parent, model, name)`

Adds an object (for example a dataflow block) to parent as a named child, without adding it to the parent's component list. If the name is taken, a new name is chosen. Returns the added object, or null.

```
var block = modelLoad({"@type": "script", "script": "return 1;"});  
modelAddObject(@parent, block, "helper");
```

modelRemoveComponent

`modelRemoveComponent(model, [recycle])`

Removes a component from its container. By default the component is destroyed. Pass false as the second parameter to keep it, so you can add it somewhere else. Returns the removed component.

```
var box = modelRemoveComponent(@stage.@box, false);  
modelAddComponent(@stage.@group1, box, "box");
```

modelRemoveObject

`modelRemoveObject(model, [recycle], [clearBinding])`

Removes a named child (such as a block) from its parent.

- `recycle` - destroy the object. Default true.
- `clearBinding` - also remove bindings from other objects to this one. Default false.

Returns the removed object.

```
modelRemoveObject(@parent.@helper);
```

modelDestroy

`modelDestroy(model)`

Destroys an object. If it is a component in a container, it is also removed from the container. Calling it on an object that is already destroyed does nothing. Returns null.

modelIsDestroyed

modelIsDestroyed(model)

Returns true if the object has been destroyed. Use it to check a model object you kept in a variable before you use it.

```
var block = modelLoad({"@type": "script"});  
print(modelIsDestroyed(block));  
modelDestroy(block);  
return modelIsDestroyed(block);
```

Print: false, Output: true

modelGetRepeaterChildren

modelGetRepeaterChildren(repeater)

Returns a list of the components that a repeater currently shows. Returns null for other objects.

```
var items = modelGetRepeaterChildren(@stage.@repeater1);  
for (var i = 0; i < items.length; i++) {  
    items[i].alpha = 0.5;  
}
```

modelMoveForward, modelMoveBackward, modelMoveToFront, modelMoveToBack

modelMoveForward(list)
modelMoveBackward(list)
modelMoveToFront(list)
modelMoveToBack(list)

Change the stacking order of components in their container, like the Arrange menu. The parameter is always a **list** of components that share the same container. If any item is not a model object or is destroyed, nothing happens.

```
modelMoveToFront([@stage.@box]);
```

modelAddArray

```
modelAddArray(array, value, [index])
```

Adds a value to a model array (such as a container's component list). Without `index` the value is added at the end. Returns the index of the new item.

modelRemoveArray

```
modelRemoveArray(array, index)
```

Removes the item at `index` from a model array. Returns `null`.

View values

```
dgGetViewValue(component, name)  
dgSetViewValue(component, name, value)
```

Some components expose values that live only in the displayed view, not in the model. These functions read and write them. They return `null` when the component has no such view value or is not displayed.

Symbol parameter enums

```
paramGetEnum(model, name)  
paramSetEnum(model, name, enums)
```

`paramGetEnum` returns the choices of the enum parameter name of a symbol as a comma-separated string, or `null`. `model` is the object that owns the parameters, for example `@parent` in the symbol's root dataflow.

`paramSetEnum` replaces the choices. `enums` is a comma-separated string. Any custom enum labels are removed.

```
paramSetEnum(@parent, "mode", "day,week,month");  
return paramGetEnum(@parent, "mode");
```

From:

<http://wiki.dglogik.com/> - **DGLogik**

Permanent link:

http://wiki.dglogik.com/dglux5_wiki:dgscript:model_functions

Last update: **2026/10/22 17:28**

