

Language Basics

DGLux5 script (DG Script) looks like JavaScript, but it is a separate, smaller language with its own interpreter. Most simple JavaScript code runs unchanged. This page describes the syntax that DG Script supports and the places where it behaves differently from JavaScript.

Topics on this page:

- [Script basics and the result of a script](#)
- [Statements](#)
- [Variables and scope](#)
- [Functions, closures, and new](#)
- [Types and typeof](#)
- [Operators](#)
- [What is not supported](#)
- [Differences from JavaScript \(summary\)](#)

See also: [Page model](#), [Debugging](#).

Script Basics

You write a script in the **script** property of a [Script](#) block, or as an expression in a table block such as [Filter](#) or [Column Mapping](#).

The value of a script is:

- the value of the first return statement that runs, or
- if no return runs, the value of the last statement that ran. An expression statement such as `x * 2`; produces its value; `var` declarations, `if` statements, and loops produce nothing (null).

The Script block puts this value in its **output** property.

```
var x = 10;  
x * 2;    // output is 20
```

Every run starts fresh: variables do not keep their values between runs. To keep a value, store it in a property, for example `@.count` (see [Page model](#)).

Comments and Semicolons

- `//` comment runs to the end of the line.
- `/*` comment `*/` can span several lines.
- A semicolon or a line break ends a statement.

Important

- After `return`, `break`, and `continue`, always write a semicolon. Unlike JavaScript, a line break after `return` does not end the statement: the next line becomes the return value, and `return` directly before `}` is a syntax error.

Literals

Kind	Examples	Notes
Number	42, 3.14, .5, 1e3, 1.5e-3, 0x1F	
String	'single', "double"	See Strings for escapes. Template literals (backquotes) are not supported.
Boolean	true, false	
Null	null	There is no undefined value. See Types .
List (array)	[1, 2, 3]	See Lists and maps .
Object (map)	{a: 1, "b c": 2}	Keys can be names or strings. Quote keys that are numbers ({ "1": "a" }); an unquoted number key is silently dropped) or reserved words such as <code>default</code> or <code>new</code> . No trailing comma.
Regular expression	/ab+c/gi	See Strings .
Function	function (x) { return x * 2; }	See Functions .

Names

Names (identifiers) can contain the letters a-z and A-Z, digits, `_`, `$`, and `@`. They cannot start with a digit. Other characters, including accented letters, are not allowed in names (they are fine inside strings).

Names that start with `@` refer to the page model, for example `@` and `@parent`. Names that start with `$` are used for special context variables such as `$params` and `$thisRow`. See [Page model](#).

The following words are reserved and cannot be used as variable names, function names, or after a dot (`obj.default` is an error; write `obj["default"]`):

```
as break case catch class const continue default delete do else extends false
finally for function if import in is namespace new null package private public
return super switch this throw true try typeof use var void while abstract
debugger enum export goto implements interface native protected synchronized
throws transient volatile instanceof with
```

Statements

DG Script supports these statements:

Statement	Example
Block	{ a = 1; b = 2; }
var	var a = 1, b;
Expression	a = b + 1;
if/else if/else	if (a > 1) { ... } else if (a < 0) { ... } else { ... }
for	for (var i = 0; i < 10; i++) { ... }
for ... in	for (var key in obj) { ... }
while	while (a < 10) { a++; }
do ... while	do { a++; } while (a < 10);
break, continue	Optionally with a label: break outer;
Labeled statement	outer: for (...) { ... }
switch	See switch below.
return	return a;
function declaration	function add(a, b) { return a + b; }

The conditional operator `cond ? a : b` is also supported.

for ... in

`for (var k in x)` loops over:

- the keys of an object (map), in insertion order;
- the **indexes** (0, 1, 2, ...) of a list.

```
var obj = {a: 1, b: 2, c: 3};
var keys = [];
var sum = 0;
for (var k in obj) {
    keys.push(k);
    sum += obj[k];
}
return keys.join("") + sum;    // "abc6"
```

```
var list = ["x", "y"];
var out = "";
for (var i in list) {
```

```
    out = out + i;
}
return out;    // "01" - the indexes, not the items
```

Labels

Use a label to break out of, or continue, an outer loop.

```
var found = "";
outer: for (var i = 0; i < 3; i++) {
    for (var j = 0; j < 3; j++) {
        if (i * j == 2) {
            found = String(i) + "," + j;
            break outer;
        }
    }
}
return found;    // "1,2"
```

switch

case values are compared with == (loose equality). switch works differently from JavaScript:

- **There is no fall-through.** After a matching case runs without break, the following case clauses are tested again; each runs only if it also matches.
- **default runs whenever it is reached**, even if an earlier case matched. Only break prevents it from running.

To get the usual behavior, end every case with break; and put default last.

```
var out = "";
switch (2) {
    case 1: out += "one";
    case 2: out += "two";
    case 3: out += "three";
    default: out += "default";
}
return out;    // "twodefault" (JavaScript would give "twothreedefault")
```

Important

Inside the braces of a switch, write only case and default clauses. Any other code directly inside the braces is a syntax error.

Variables and Scope

- You do not have to declare variables. `x = 5;` creates `x`.
- `var` inside a function creates a variable that is local to that function. A `var` anywhere in the function makes the name local in the whole function.
- There is no block scope. A variable declared inside `{ }` of an `if` or loop is visible in the whole function (or script).
- Inside a function, assigning to a name that is not declared with `var` (or as a parameter) in that function or an enclosing function sets a script-level variable.
- A variable that has never been set reads as `null`. Reading it is not an error.
- Script-level variables shadow built-in functions such as `print` or `Math` only while they are not `null`.

```
function setTotal() {  
    total = 5;    // no "var": sets the script-level variable  
}  
setTotal();  
return total;    // 5
```

Tip

Give variables a starting value before you use `+=`, `-=`, `++`, or `--`. These operators do nothing useful on `null`:
`s += "a"` leaves `s` as `null` if `s` was never set.

```
var s;  
s += "a";  
return s == null;    // true
```

Functions

You can declare named functions, assign function expressions to variables, put functions in objects, and pass functions as arguments.

```
function add(a, b) {  
    return a + b;  
}  
var square = function (x) { return x * x; };  
return add(square(3), 1);    // 10
```

Rules:

- **No hoisting.** A function declaration is created when the script reaches it. Declare a function before the line that calls it. Calling it earlier fails with "invalid call to ...".
- Missing arguments are null. Extra arguments are ignored. There is no arguments object.
- A function without a return returns null.
- Recursion works.

```
function f(a, b) {  
    return b == null;  
}  
return f(1);    // true
```

Closures

A function keeps access to the variables of the function that created it.

```
function makeCounter() {  
    var count = 0;  
    return function () {  
        count++;  
        return count;  
    };  
}  
var next = makeCounter();  
next();  
next();  
return next();    // 3
```

this

When you call a function as a member of an object, obj.method(), this is that object.

```
var counter = {  
    n: 0,  
    inc: function () {  
        this.n = this.n + 1;  
        return this.n;  
    }  
};  
counter.inc();  
return counter.inc();    // 2
```

At the top level of a script, this is null. To refer to the block that runs the script, use @.

new

`new` works with the built-in `DateTime` class (see [Date and time](#)) and with your own functions. With your own function, `new` creates an empty object, runs the function with `this` set to that object, and returns the object. Prototypes and classes are not supported.

```
function Point(x, y) {  
  this.x = x;  
  this.y = y;  
}  
var p = new Point(1, 2);  
return p.x + p.y;    // 3
```

Types and typeof

DG Script has these kinds of values: number, string, boolean, null, list, object (map), function, `DateTime`, regular expression, XML node, promise, and page model objects (components and blocks).

There is only one "empty" value: `null`. `undefined` is not a keyword; it is just a variable name that has never been set, so it reads as `null`. That is why old code such as `if (s == undefined)` still works.

`typeof x` returns:

Value	typeof
Number (including NaN)	number
String	string
Boolean	boolean
List	list (not object)
Page model object	its type, such as <code>script</code> or <code>symbol</code>
Anything else, including null, objects, and functions	object

```
var t = [typeof 1, typeof "a", typeof true, typeof [1], typeof {a: 1}, typeof  
null];  
return t.join(",");    // "number,string,boolean,list,object,object"
```

True and False Values

In `if`, loops, `!`, `&&`, `||`, and `?` `:`, these values count as false: `false`, `null`, `0`, `NaN`, and the empty string `""`. Everything else is true, including `"0"`, `"false"`, empty lists, and empty objects.

```
return [!!0, !!"", !!"0", !!null, !![], !!{}].join(",");
// "false,false,true,false,true,true"
```

&& and || return one of their operands, as in JavaScript. For example, name || "unknown" returns "unknown" when name is null or empty.

Operators

Arithmetic

Operator	Description	Example	Result
+	Add or concatenate (see rules below)	5 + 5	10
-	Subtract	10 - 5	5
*	Multiply	2 * 2	4
/	Divide	9 / 2	4.5
%	Remainder; the result has the sign of the left operand	-7 % 3	-1
-x	Negate a number. Returns null if x is not a number.	-(2 + 3)	-5
++, --	Increment, decrement (prefix or postfix). Only change numbers.	i++	

-, *, /, and % convert strings and booleans to numbers: "10" - 3 is 7 and 2 * "4" is 8. A string that is not a number becomes NaN.

Unary + does nothing: +"5" is still the string "5". Use Number("5") to convert. See [Numbers and math](#).

Rules for +

The **left** operand decides what + does:

Left operand	Result	Example	Result
Number	Numeric addition. The right operand is converted to a number.	1 + "2"	3 (JavaScript gives "12")
String	Concatenation. The right operand is converted to a string; null becomes "".	"1" + 2	"12"
Anything else (null, boolean, list, ...)	null	true + 1	null

This matters when you build text that starts with a number: i + ", " + j gives NaN. Start with a string instead: "" + i + ", " + j or String(i) + ", " + j.

```
var r = [1 + "2", "1" + 2, 1 + "a", "a" + null];
```

```
return r.join("|"); // "3|12|NaN|a"
```

Comparison

Operator	Description	Example	Result
<, >, <=, >=	Compares strings alphabetically when both are strings, otherwise compares as numbers	"abc" < "abd"	true
==	Loose equality: a string equals a number if it parses to that number, and "true"/"false" equal the booleans	"5" == 5	true
!=	Opposite of ==	"1" != 1	false
===	Strict equality: no conversion	5 === "5"	false
!==	Opposite of ===	"1" !== "1"	false
in	key in object tests whether an object has a key; index in list tests whether a list index exists	"a" in {a: 1}	true

== differs from JavaScript in some cases: 1 == true and null == 0 are false. Lists and objects are equal only if they are the same object.

```
var r = ["5" == 5, "5" === 5, true == "true", 1 == true, null == 0, "abc" < "abd"];
return r.join(","); // "true,false,true,false,false,true"
```

```
return ["1" !== "1", "1" !== 1].join(","); // "false,true"
```

```
return ["a" in {a: 1}, 1 in [5, 6], 2 in [5, 6]].join(","); // "true,true,false"
```

Logical

Operator	Example	Result
&& (and)	true && false	false
(or)	true false	true
! (not)	!true	false

&& and || stop evaluating as soon as the result is known.

Bitwise

Operands are converted to integers.

Operator	Description	Example	Result
&	And	5 & 3	1
	Or	5 3	7
^	Xor	5 ^ 3	6

Operator	Description	Example	Result
<<	Shift left	1 << 4	16
>>	Shift right. Negative numbers round toward zero: -7 >> 1 is -3 (JavaScript gives -4).	7 >> 1	3
>>>	Unsigned shift right	-1 >>> 28	15

There is no ~ (bitwise not) operator.

```
var r = [9 / 2, 10 % 3, -7 % 3, 2 * "4", "10" - 3, 7 >> 1, -7 >> 1, 5 & 3, 5 | 3, 5 ^ 3, 1 << 4];
return r.join(","); // "4.5,1,-1,8,7,3,-3,1,7,6,16"
```

Assignment

Operator	Description
=	Assign
+=, -=, *=, /=, %=	Compound arithmetic assignment
<<=, >>=, &=, =, ^=	Compound bitwise assignment (>>>= is not supported)
~=	DGLux-only: set a page model property so that it is saved, or create a binding. Only works on page model properties. See Page model .

Other Operators

- `delete obj.key` removes a key from an object. On a page model property, it sets the property to null.
- `void expr` evaluates `expr` and returns null.
- `typeof x` — see [Types](#).
- `a, b` (comma) evaluates both and returns `b`.

```
var o = {a: 1, b: 2};
delete o.a;
return JSON.stringify(o); // {"b":2}
```

What Is Not Supported

The following JavaScript features cause a syntax error or do not work:

- `try`, `catch`, `finally`, `throw`. An error stops the whole script (see [Errors](#)). Check values before you use them instead.
- `let` and `const`. Use `var`.
- Arrow functions (`x => x * 2`). Use function (`(x) { return x * 2; }`).

- Classes, extends, super, prototypes, instanceof.
 - Template literals (``Hello ${name}``). Use `+`.
 - The undefined value (it reads as null), the arguments object, with, eval.
 - Spread (`...`), destructuring, default parameter values, getters and setters.
 - `~` (bitwise not) and `>>>=`.
 - The JavaScript Date object. Use `DateTime` (see [Date and time](#)).
 - Indexing a string with brackets: `"abc"[1]` returns null. Use `"abc".charAt(1)`.
 - Custom properties on lists: `list.name = "x"` is silently ignored.
-

Differences from JavaScript (Summary)

Read this list if you already know JavaScript:

- `+` with a number on the left adds numerically: `1 + "2"` is 3. See [Rules for +](#).
 - `switch` has no fall-through, and `default` runs whenever it is reached. See [switch](#).
 - Function declarations are not hoisted.
 - `typeof` returns `list` for lists and the type name for page model objects.
 - There is no undefined; unset values are null.
 - `-7 >> 1` is -3; unary `+` does not convert to a number; unary `-` on a non-number gives null.
 - A line break after `return` does not end the statement.
 - Reading an index past the end of a list is an error ("can not access ..."), not undefined. Reading a missing key of an object returns null.
 - Months in `DateTime` start at 1 (January = 1), unlike JavaScript `Date`.
 - Statements that do not exist in JavaScript: `~`= (see [Page model](#)) and the `@` names.
-

[Previous: DGLux5 Script](#)

[Next: Page Model](#)

From:

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

Permanent link:

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

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

