

EScript (0.7.2 Egon)

A Short Introduction

Benjamin Eikel, Claudius Jähn

Version: March, 2015



Overview

- 1 Introduction
- 2 Data Types and Operators
- 3 Calling functions
- 4 Local variables
- 5 Arrays and Maps
- 6 Control Structures
- 7 Functions
- 8 Objects and Types
- 9 Std library
- 10 Example

Overview

- 1 Introduction
- 2 Data Types and Operators
- 3 Calling functions
- 4 Local variables
- 5 Arrays and Maps
- 6 Control Structures
- 7 Functions
- 8 Objects and Types
- 9 Std library
- 10 Example

What is EScript?



EScript . . .

- is an object-oriented scripting language.
- is compiled and executed by a virtual machine.
- has a similar syntax to C.
- was developed to use C++ objects from scripts easily.

What is EScript?



EScript ...

- is released under a free software license (MIT).
- is available from <https://github.com/EScript>.
- has a command-line interpreter.
- can be built using CMake.
- can be used internally by other C++ projects (e.g. by PADrend <http://PADrend.de>).
- stands for HasE-Script.

First Example

- A simple script:

```
outln( "Hello World!" ); // Outputs: Hello World!
```

First Example

- A simple script:

```
outln( "Hello World!" ); // Outputs: Hello World!
```

- Calls the global function `outln` with the string "Hello World!" as parameter value.
- The statements ends with a semicolon.
- Comments begin with `//` or are enclosed with `/* */`.

Overview

- 1 Introduction
- 2 Data Types and Operators
- 3 Calling functions
- 4 Local variables
- 5 Arrays and Maps
- 6 Control Structures
- 7 Functions
- 8 Objects and Types
- 9 Std library
- 10 Example

Simple Types (call-by-value)

Number

42

27.4

0x1a

-2.7e+10

0b101010

String

"an"

'example string'

Bool

true

false

Void (empty or no value)

void

Operators

Some operators

```
outln( 2+40 ); // Output: 42
outln( 2*21 ); // Output: 42
outln( "4" + "2" ); // Output: 42
outln( "foo"+"bar" ); // Output: foobar
outln( "wup " * (6/2) ); // Output: wup wup wup
outln( 1>2 ); // Output: false
outln( !true ); // Output: false
outln( true & true ); // Output: true
outln( false || true ); // Output: true
outln( "foo" == "bar" ); // Output: false
outln( "foo" != "bar" ); // Output: true
```

Type Conversion

Only **false** and **void** convert to **false**

```
println(false || false); // Output: false
println(false || void); // Output: false
println(false || 0); // Output: true
println(false || ""); // Output: true
```

Type Conversion

Only **false** and **void** convert to **false**

```
outln(false || false); // Output: false
outln(false || void); // Output: false
outln(false || 0); // Output: true
outln(false || ""); // Output: true
```

Conversion to number (left operand is a number)

```
outln( 12 + "3" ); // Output: 15
outln( 10 * "10" ); // Output: 100
outln( 10 == "10" ); // Output: true
outln( 10 == "10.0" ); // Output: true
```

Type Conversion

Only **false** and **void** convert to **false**

```
outln( false || false ); // Output: false
outln( false || void ); // Output: false
outln( false || 0 ); // Output: true
outln( false || "" ); // Output: true
```

Conversion to number (left operand is a number)

```
outln( 12 + "3" ); // Output: 15
outln( 10 * "10" ); // Output: 100
outln( 10 == "10" ); // Output: true
outln( 10 == "10.0" ); // Output: true
```

Conversion to string (left operand is a string)

```
outln( "12" + 3 ); // Output: 123
outln( "10" == 10 ); // Output: true
outln( "10.0" == 10 ); // Output: false
```

Equality checks

Check equality with conversion ==

Check equality without conversion ===

```
outln( 10 == "10" ); // Output: true
outln( 10 === "10" ); // Output: false
outln( 10 === 10 ); // Output: true
outln( true == "foo" ); // Output: true
outln( true === "true" ); // Output: false
outln( "true" == true ); // Output: true
outln( "true" === true ); // Output: false
```

Special type: Identifier

- Variable and attribute names have a special data type: Identifier.
- Identifier objects are immutable (can not be changed).
- Identifiers are often used as value for constants.
- Identifiers are created using the dollar sign: `$exampleIdentifier`
- `$foo == "foo" // false`
- `"foo" == $foo // true`

Overview

- 1 Introduction
- 2 Data Types and Operators
- 3 Calling functions
- 4 Local variables
- 5 Arrays and Maps
- 6 Control Structures
- 7 Functions
- 8 Objects and Types
- 9 Std library
- 10 Example

Calling functions

Calling functions with different origins:

```
// call global function 'load':  
load( "someScript.escript" );  
  
// call function 'saveTextFile' in namespace 'IO':  
IO.saveTextFile( "foo.txt" , "bar" );  
  
// call method 'sqrt' of object 9.0:  
out( (9.0).sqrt() ); // Output: 3
```

Overview

- 1 Introduction
- 2 Data Types and Operators
- 3 Calling functions
- 4 Local variables**
- 5 Arrays and Maps
- 6 Control Structures
- 7 Functions
- 8 Objects and Types
- 9 Std library
- 10 Example

Declaring Variables

Declare a variable with **var**:

```
// "foo" is an empty variable (containing void).  
var foo;  
  
// The variable "xPos" contains a number  
var xPos = 500 - 80 / 2;  
  
// The variable "message" will be of type String  
var message = "Please click the button";  
  
// Dynamically change the type to Number  
message = 5;
```

Overview

- 1 Introduction
- 2 Data Types and Operators
- 3 Calling functions
- 4 Local variables
- 5 Arrays and Maps**
- 6 Control Structures
- 7 Functions
- 8 Objects and Types
- 9 Std library
- 10 Example

Built-in collection types

Array

```
var numbers = [3, 23, 7, 3, 100, 1, 35];  
var colors = ["red", "green", "blue"];  
outln( numbers[4] ); // Outputs: 100  
outln( numbers.count() ); // Outputs: 7  
outln( numbers.empty() ); // Outputs: false
```

Map

```
var fruits = {  
    "lemon" : "yellow",  
    "cherry" : "red"  
};  
fruits["apple"] = "green";
```

Overview

- 1 Introduction
- 2 Data Types and Operators
- 3 Calling functions
- 4 Local variables
- 5 Arrays and Maps
- 6 Control Structures**
- 7 Functions
- 8 Objects and Types
- 9 Std library
- 10 Example

Conditionals (1)

Conditional execution with **if/else**.

```
var result = someFunction();  
if(result) {  
    out("Success");  
} else {  
    out("Failure");  
}  
  
var num = calculateSomething();  
if(num < 0)  
    out("Too small");  
else if(num >= 0 && num <= 100)  
    out("Range okay");  
else  
    out("Too large");
```

Conditionals (2)

? (conditional operator)

```
var num = calculateSomething();  
var positive = (num > 0) ? true : false;
```


Loops (1)

Looping with **while**:

```
var numbers = [ 4,5,29,32 ];  
while (!numbers.empty()) {  
    var n = tasks.back();  
    n.popBack();  
    out( n, " " );  
}  
// Outputs: 32 29 5 4
```

Loops (2)

Looping with **for**:

```
var sum = 0;
for(var i = 0; i < 100; ++i) {
    sum += i;
}
outln("Sum of numbers: ", sum);
```

Loops (3)

Iterate over a container: **foreach**.

```
var chars = ['a', 'c', 'k', 'b', 'd', 'x', 'j'];  
foreach(chars as var index, var currentChar) {  
    if(currentChar == 'x') {  
        println("Character 'x' found at index ",  
                index);  
        break;  
    }  
}
```

- Output: Character 'x' found at index 5
- The index variable is optional:

```
foreach( collection as var value) println(value);
```

Exception handling

Catch and handle an exception: `try/catch`.

```
try{
    outln( 42/0 );
} catch(e){
    outln( e );
}
```

- Output: Division by zero...
- For throwing an exception, use `Runtime.exception('message');`

Overview

- 1 Introduction
- 2 Data Types and Operators
- 3 Calling functions
- 4 Local variables
- 5 Arrays and Maps
- 6 Control Structures
- 7 Functions**
- 8 Objects and Types
- 9 Std library
- 10 Example

Declaring simple functions

- Declare functions with **fn**
- Functions have no names, but they can be stored in a variable:

```
var square = fn(num) {  
    return num * num;  
};  
var a = square(5);  
outln( a ); // Outputs: 25
```

- Parameters can be restricted with type checks:

```
var square = fn(Number num) {  
    return num * num;  
};  
square(4); // ok  
square( 'foo' ); // runtime error
```

Parameters

- Parameters can be restricted with type checks:

```
var square = fn(Number num) {  
    return num * num;  
};  
square(4); // ok  
square( 'foo' ); // runtime error
```

- Parameters can have default values:

```
var add = fn(a, b=1) {  
    return a+b;  
};  
outln( add( 10, 2 ) ); // Outputs: 12  
outln( add( 10 ) ); // Outputs: 11
```


Multi parameters

- Multi parameters accept arbitrary many values and store them in an array:

```
var sum = fn( numbers... ) {  
    var sum = 0;  
    foreach( numbers as var n)  
        sum += n;  
    return sum;  
};  
outln( sum( 10,100,1000,4 ) ); // Outputs 1114
```

Bind parameter values

- Set the first parameters to fixed values: **Array** => **fn**(...).
- Bound function object behaves like normal function.

```
var myFun = fn(a,b,c) {  
    out( 'a:', a, ' b:', b, ' c:' c );  
};  
myFun( 1, 2, 3); // Output: a:1 b:2 c:3  
  
var myBoundFun = [ 100, 200 ] => myFun;  
myBoundFun( 300 ); // Output: a:100 b:200 c:300
```

Bind calling object

- Create a combination of a function and an object: `object->fun`
- Bound function object behaves like normal function.
- When called, the bound object is the function's **this**-object.

```
var myObject = [100,200];  
var myBoundFun = myObject -> Array.max;  
  
outln( myBoundFun() ); // Output: 200
```

Variable and parameter scopes

- The scope of local variables (**var**) is the tightest enclosing block, but *excluding* functions defined in the block.
- The scope of a parameter is the enclosing block of the function, but *excluding* functions defined in the function.
- Local variables and parameters are allocated for every call of the containing function (on a stack).

Variable and parameter scopes

- The scope of local variables (**var**) is the tightest enclosing block, but *excluding* functions defined in the block.
- The scope of a parameter is the enclosing block of the function, but *excluding* functions defined in the function.
- Local variables and parameters are allocated for every call of the containing function (on a stack).
- The scope of static variables (**static**) is the tightest enclosing block, *including* functions defined in the block.
- Static variables are allocated once for all calls of the containing function.

Variable and parameter scopes

- The scope of local variables (**var**) is the tightest enclosing block, but *excluding* functions defined in the block.
- The scope of a parameter is the enclosing block of the function, but *excluding* functions defined in the function.
- Local variables and parameters are allocated for every call of the containing function (on a stack).
- The scope of static variables (**static**) is the tightest enclosing block, *including* functions defined in the block.
- Static variables are allocated once for all calls of the containing function.

```
static factorial = fn( Number n ) {  
    return (n == 0) ? 1 : factorial (n - 1) * n;  
};  
out( factorial( 5 ) ); // Output: 120
```

Overview

- 1 Introduction
- 2 Data Types and Operators
- 3 Calling functions
- 4 Local variables
- 5 Arrays and Maps
- 6 Control Structures
- 7 Functions
- 8 Objects and Types**
- 9 Std library
- 10 Example

Extendable object

Extendable objects: **ExtObject**.

```
var car = new ExtObject;  
car.color := "red"; // := creates a new member  
car.speed := 190;  
car.outputDesc := fn() {  
    out("This is a ", this.color, " car ");  
    out("with top speed ", this.speed, ".\n");  
};  
  
car.speed = 185;  
car.outputDesc();
```

Output: This is a red car with top speed 185.

Inheritance

Types and inheritance:

```
var Shape = new Type;
Shape.color := "white";

// New type that is derived from Shape
var Polygon = new Type(Shape);
Polygon.numVertices := 3;

// New type that is derived from Shape
var Circle = new Type(Shape);
Circle.radius := 0;

var circle = new Circle;
circle.color = "red";
circle.radius = 5;
```

Member attribute properties

Example

```
var Polygon = new Type;
Polygon.vertices @(private, init) := Array;
Polygon.shapeType @(const) := "Polygon";

Polygon.getNumVertices ::= fn() {
    return this.vertices.count();
};

var polygon = new Polygon;
polygon.getNumVertices();
```

Overview

- 1 Introduction
- 2 Data Types and Operators
- 3 Calling functions
- 4 Local variables
- 5 Arrays and Maps
- 6 Control Structures
- 7 Functions
- 8 Objects and Types
- 9 Std library**
- 10 Example

Std library

- Set of helper functions and types.
- Implements the module concept.
- Example: MultiProcedure

```
var myFun = new Std.MultiProcedure;

myFun += fn(arr){    arr += 'Hallo'; };
myFun += fn(arr){
    arr += ' Welt';
    return $REMOVE;
};
myFun += fn(arr){    arr += ' ';    };

var arr = [];
myFun( arr );
outln(arr.implode() ); // Outputs: Hallo Welt!

var arr2 = [];
myFun( arr2 );
outln(arr2.implode() ); // Outputs: Hallo!
```

Overview

- 1 Introduction
- 2 Data Types and Operators
- 3 Calling functions
- 4 Local variables
- 5 Arrays and Maps
- 6 Control Structures
- 7 Functions
- 8 Objects and Types
- 9 Std library
- 10 Example**

Example

```
var Player = new Type;
Player.x @(private) := 0;
Player.y @(private) := 0;
Player.move ::= fn(Number dx, Number dy) {
    this.x += dx;
    this.y += dy;
};
Player.printPos ::= fn() {
    outln("Position: (", this.x, ", ", this.y, ")");
};

var playerA = new Player;
playerA.move(5, 7);
playerA.printPos(); // Output: Position: (5, 7)
```

Further Documentation

You can find additional documentation in
`EScript/docs/Introduction.html`.