

# JavaScript.

## Fundamentals

Chrome File Edit View History Bookmarks Window Help 95% Fri 10:22 Diarmuid O' Connor

Application Architect x Authentication with A x Enterprise Web Devel x Top Github Language x Diarmuid O'Connor x

adambard.com/blog/top-github-languages-2014/

| Language     | Rank |      |      | # New Repos Created |        |        |
|--------------|------|------|------|---------------------|--------|--------|
|              | 2014 | 2013 | 2012 | 2014                | 2013   | 2012   |
| JavaScript   | 1    | 1    | 2    | 383185              | 320534 | 277875 |
| Java         | 2    | 3    | 3    | 283354              | 185530 | 240992 |
| Ruby         | 3    | 2    | 1    | 259268              | 228145 | 310281 |
| C            | 4    | 7    | 4    | 178891              | 79223  | 203992 |
| CSS          | 5    | 12   | 25   | 175573              | 18869  | 3791   |
| PHP          | 6    | 4    | 6    | 175476              | 139591 | 157185 |
| Python       | 7    | 5    | 5    | 151669              | 126027 | 165655 |
| C++          | 8    | 6    | 7    | 78878               | 104499 | 88615  |
| Objective-C  | 9    | 8    | 11   | 60579               | 40072  | 36539  |
| C#           | 10   | 10   | 10   | 59472               | 34992  | 39486  |
| Shell        | 11   | 9    | 8    | 48388               | 35204  | 68720  |
| R            | 12   | 23   | 26   | 25229               | 3790   | 3216   |
| CoffeeScript | 13   | 13   | 14   | 22340               | 14760  | 15085  |
| Go           | 14   | 15   | 12   | 16631               | 8267   | 18452  |
| Perl         | 15   | 11   | 9    | 13888               | 19418  | 46607  |

store.zip lt20065534.rar ziyi jiang.rar WenkaiGao finial project.rar Show All x

# Background.

- **Javascript is Ubiquitous.**
  - ... also on the server-side (node.js), embedded (Duktape)
- **Written at Mosaic by Brendan Eich (early 1990s) under the name Mocha and later LiveScript, ECMAScript, and Jscript.**
- **Influenced heavily by Java, Self and Scheme.**
- **Douglas Crockford - JavaScript - Volume 1: The Early Years**
- **Currently Trademarked by Oracle.**
- **ECMA standard 261**
  - **ECMAScript.**

# JavaScript Data Types.

- **Language data types:**
  1. **Primitives: number, string, boolean, null, undefined.**
  2. **Everything else is an object (even functions).**
- **JS is a dynamically typed language.**

# Primitive types.

- Suppose this code is in a file, called *primitives.js*

```
/*  
| Primitive data types in JS  
*/  
var foo1 = 5    // var means variable  
var foo2 = 'Hello'  
var foo3 = true // not 'true'. foo3 is a boolean variable  
var foo4 = null // null is a keyword, just like var  
console.log( foo1 + ' ' + foo2 + ' ' + foo3 + ' ' + foo4)  
foo1 = 3    // change foo1 to be 3. No need for var keyword.  
foo2 = 10   // JS is dynamically typed. Great, but don't misuse!!  
var foo5  
console.log (foo5)
```

- Thanks to the node.js platform, I can execute this code from the command line – no browser needed.

```
$  
$ node primitives.js  
5 Hello true null  
undefined  
$
```

# Primitive types (The syntax).

```
var foo = 20
```

- **var** – keyword to indicate we are declaring something – a primitive number variable in this case.
- **Identifier** – ‘foo’ is an identifier or name for this thing.
  - Lots of rules about valid format for identifiers (no spaces, don’t start with numeric character, etc etc)
- **Operator** – +, =, \* (multiply), -, [ ] (subscript) etc
  - Some rules about where they can appear in a statement.

# Objects.

- **The object - fundamental structure for representing complex data.**
- **A unit of composition for data ( or STATE).**
- **Objects are a set of key-value pairs defining properties.**
  - **Keys (property names) are identifiers and must be unique**
  - **Values can be any type, including other objects (nesting).**
- **Literal syntax for defining an object:**
  - `{ <key1> : <value1>, <key2> : <value2>, ... }`
  - **Example:**  
`var me = { first_name: "Diarmuid", last_name: "O'Connor" }`

# Objects.

- **Two notations for accessing the value of a property:**
  1. **Dot notation e.g** `me.first_name`
  2. **Subscript notation e.g.** `me['first_name']` (Note quotes)
- **Same notations for changing a property value.**  
`me.first_name = 'Jeremiah'`  
`me['last_name'] = 'O Conchubhair'`
- **Subscript notation allows the subscript be a variable reference.**  
`var foo = 'last_name'`  
`me[foo] = .....`



# Objects.

```
1  var me = {
2      name : "Diarmuid O'Connor",    // use " " when string contains '
3      address : '1 Main Street',
4      age : 21,
5      bank_balance : 20.2,    //millions
6      male : true    // no comma for the last property
7  }
8
9  console.log (me.name + ' lives at ' + me['address'])
10 // Can also use a variable in subscript notation
11 var prop = 'bank_balance'
12 var balance = me[prop]
13 console.log('Balance = ' + balance)
14 // Changing a property value
15 me.address = '2 Main Street'
16 console.log (me.name + ' lives at ' + me['address'])
17
```

# Objects are dynamic.

- Properties can be added and removed at any time – JS is dynamic.

```
1  var me = {
2      name : "Diarmuid O'Connor",
3      address : '1 Main Street',
4      age : 21,
5      bank_balance : 20.2,    //millions
6      male : true
7  }
8  // New property
9  me.employer = 'WIT'
10 console.log (me.name + ' works for ' + me.employer)
11 // Remove property
12 delete me.age
13 console.log (me.age)    // undefined
14
```

# Nested objects.

- A property value may be an object structure.

```
1  var me = {
2    name : {
3      first : 'Diarmuid',
4      last  : "O'Connor"
5    },
6    address : '1 Main Street',
7    age : 21,
8    bank_balance : {
9      amount : 20.2,
10     type : 'D',
11     bank : 'AIB'
12   },
13   male : true
14 }
15
16 console.log (me.name.first + ' banks with ' + me['bank_balance']['bank'])
```

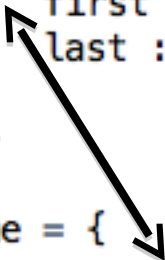
\$ node nested\_objects.js  
Diarmuid banks with AIB

- Nesting can be to any depth.

# Object property.

- A property value can be a variable reference.

```
1  var my_name = {
2      first : 'Diarmuid',
3      last  : "O'Connor"
4
5  }
6
7  var me = {
8      name : my_name,
9      address : '1 Main Street',
10     age : 21,
11     bank_balance : {
12         amount : 20.2,
13         type : 'D',
14         bank : 'AIB'
15     },
16     male : true
17 }
18
19 console.log (me.name.first) // Diarmuid
```



# Object keys

- Internally JS stores keys as strings.
- Hence the subscript notation – `me['age']`

# Array data structure.

- Dfn: Arrays are an ordered list of values.
  - An object's properties are not ordered.
- Literal syntax: [ <value1>,<value2>,... ]
- In JS, the array values may be of mixed type.
  - Although mixed types may reflect bad design.
- Use an index number with the subscript notation to access individual elements:

```
1  var nums = [12,22,5,18]
2  var first = nums[0]    // not nums['0']
3  var second = nums[1]
4  console.log(second)   // 22
5  var stuff = [ 12,
6                'web',
7                {a : 1, b : 2},
8                null
9            ]
10 console.log(stuff[1])  // 'web'
11 console.log(stuff[2].b) // 2
```

# Array data structure.

- In JS, arrays are really just 'special' objects:
  - The indexes are not numbers, but properties - the index number is converted into a string:  
    nums['2'] same as nums[2]
  - Special length property, e.g. var len = nums.length // 4
  - Some utility methods for manipulating elements e.g push, pop, shift, unshift, join etc
    - push/pop – add/remove at the tail.
    - shift/unshift – add/remove at the head.

```
// Manipulating arrays
nums.push(10)
console.log(nums)
var element = nums.pop() // 10
console.log(nums)
element = nums.shift() //
console.log(nums)
nums.unshift(3)
console.log(nums)
```

```
[ 12, 22, 5, 18, 10 ]
[ 12, 22, 5, 18 ]
[ 22, 5, 18 ]
[ 3, 22, 5, 18 ]
$
```

# Nested collections.

- **Arrays and objects can be nested.**
- **Ex.:**
  - **Array of array values.**  
array\_outer[3][2]
  - **Array of objects**  
array\_outer[2].propertyX.
  - **Object property with an array value.**  
objectY.propertyX[2]
  - .....



# JS - Behavior structures

# Looping/iteration constructs

```
1 var nums = [12,22,5,18]
2
3 ▼ for (var i = 0 ; i < nums.length ; i++ ) {
4     nums[i] += 1
5     // other lines of code
6 }
7 console.log(nums)
8 var j = 0
9 ▼ while (j < nums.length ) {
10     console.log(nums[j])
11     j++
12 }
13
14 ▼ var me = {
15     name : "Diarmuid O'Connor",
16     address : '1 Main Street',
17     age : 21,
18     bank_balance : 20.2,
19     male : true
20 }
21 // for-in form especially for object iteration
22 for (var prop in me) {
23     console.log(prop + ' = ' + me[prop])
24 }
```

A more elegant form later.

```
$ node loop_construct.js
[ 13, 23, 6, 19 ]
13
23
6
19
name = Diarmuid O'Connor
address = 1 Main Street
age = 21
bank_balance = 20.2
male = true
$
```

# JavaScript functions.

- **Fundamental unit of composition for logic ( or BEHAVIOUR).**
- **Basic syntax:** `function <func_name>( <parameters> ) { <body of code> }`
  - **Some functions don't need parameters.**
- **A function's body is executed by calling/invoking it with arguments -** `<func_name>( <argumentss>)`

```
1  function sayHello(person) {
2      if (person.male == true) {
3          console.log('Hello Mr. ' + person.name.last )
4      } else {
5          console.log('Hello Mrs. ' + person.name.last )
6      }
7  }
8
9  var me = {
10     name : {
11         first : 'Diarmuid',
12         last  : "O'Connor"
13     },
14     male : true
15 }
16
17 // Calling/invoking a function
18 sayHello(me)      // Hello Mr. O'Connor|
```

# Functions - Variable scopes.

- **Every function creates a new variable scope.**
  - Variables declared inside the function are not accessible outside it.
  - All variables defined within the function are “hoisted” to the start of the function, as if all the var statements were written first.
    - You can use a variable inside a function before declaring it.
- **Global scope – default scope for everything declared outside a function’s scope.**
  - Variables in global scope are accessible inside functions.

# Functions - Variable scopes.

```
22 var foo1 = 2           // Global scope
23 function variableScopes() {
24     var foo2 = 12
25     foo3 = foo2 + foo1
26     console.log('foo3 = ' + foo3)
27     var foo3         // Declared; not initialized
28 }
29 variableScopes()
30 console.log(foo2)    // ERROR !!!!
31
```

```
foo3 = 14
```

```
node.js:134
```

```
    throw e; // process.nextTick error, or 'error' event on first tick
```

```
    ^
```

```
ReferenceError: foo2 is not defined
```

```
    at Object.<anonymous> (/Users/diarmuidoconnor/Notes/Common2/JavaScript/fundamentalsJS/functions.js:30:1)
```

```
    at Module._compile (module.js:402:26)
```

```
    at Object..js (module.js:408:10)
```

```
    at Module.load (module.js:334:31)
```

```
    at Function._load (module.js:293:12)
```

```
    at Array.<anonymous> (module.js:421:10)
```

```
    at EventEmitter._tickCallback (node.js:126:26)
```

```
$
```

Stack trace

# JavaScript functions.

- **Can be created using:**
  1. **A declaration (previous examples).**
  2. **An expression.**
  3. **A method (of a custom object).**
  4. **An anonymous unit.**
- **Can be called/invoked as:**
  1. **A function (previous examples).**
  2. **A method.**
  3. **A constructor.**

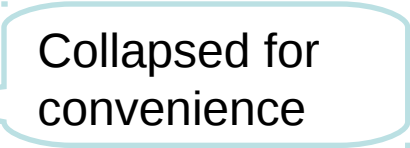
# Function Declarations

- Define a function using the syntax:

```
function name( ... ) { ... }
```

- Function definitions are “hoisted” to the top of the current scope.
  - You can use a function before it is defined – like function-scoped variables.

```
1  var me = { ... }  
8  }  
9  
10 sayHello(me)    // Hello Mr. O'Connor  
11 // .....|  
12 function sayHello(person) { ... }  
18 }  
19
```



- Can also define functions inside other functions – same scoping rules as variables.

# Function Expressions

- **Defines a function using the syntax:**  
`var name = function( ... ) { ... }`
- **Unlike function declarations, there is no “hoisting”.**
  - **You can’t use the function before it is defined, because the variable referencing the function has no value, yet.**
- **Useful for dynamically created functions.**
- **Called in the same way as function declarations:**  
`name( argument1, argument2, ... )`



# Function Expressions

```
10 var me = {
11     name : {
12         first : 'Diarmuid',
13         last : "O'Connor"
14     },
15     male : true
16 }
17
18 var addMiddleName = function(person, middle_name) {
19     if (person.name.middle == undefined) {
20         person.name.middle = middle_name
21     } else {
22         person.name.middle += ' ' + middle_name
23     }
24 }
25
26 addMiddleName(me, 'Stephen')
27 console.log(me.name)
28
```

```
{ first: 'Diarmuid',
  last: 'O'Connor',
  middle: 'Stephen' }
```

# Function Returns

- Typically, functions perform some logic AND return a result.

```
45  var my_worth = {
46      current : [ { amount : 20.2, bank : 'AIB'},
47                  { amount : 5.1, bank : 'BoI' } ],
48      deposit : [{ amount : 20.2, bank : 'Ulster'}],
49      investment : [] // Empty array
50  }
51  var computeTotal = function (accounts) {
52      var total = 0.0
53      for (var type in accounts) {
54          for (i = 0 ; i < accounts[type].length ; i++) {
55              total += accounts[type][i].amount
56          }
57      }
58      return total
59  }
60  console.log(computeTotal(my_worth)) // 45.5
```

- [A function without a return statement will return 'undefined']



# Methods

Use 'this' to reference the enclosing object

```
63 var person = {
64   name : {
68   finances : {
69     current : [ { amount : 10.2, bank : 'AIB'},
70                { amount : 5.1, bank : 'BoI' } ],
71     deposit : [{ amount : 10.2, bank : 'Ulster'}],
72     investment : []
73   },
74   computeTotal : function () {
75     var total = 0.0
76     for (var type in this.finances) {
77       for (i = 0 ; i < this.finances[type].length ; i++) {
78         total += this.finances[type][i].amount
79       }
80     }
81     return total
82   },
83   addMiddleName : function(middle_name) {
84     if (this.name.middle == undefined) {
85       this.name.middle = middle_name
86     } else {
87       this.name.middle += ' ' + middle_name
88     }
89     return this.name
90   }
91 }
92 console.log('Full worth = ' + person.computeTotal())
93 var full_name = person.addMiddleName('Paul')
94 console.log(person.name)
95 console.log(full_name)
```

```
Full worth = 25.5
{ first: 'Joe', last: 'Bloggs', middle: 'Paul' }
{ first: 'Joe', last: 'Bloggs', middle: 'Paul' }
```

# Methods.

- **Syntax comparison:**

- **Function:**

- computeTotal(person)    addMiddleName(person,'Paul')

- **Method:**

- person.computeTotal()    person.addMiddleName(me,'Paul')

- **The special 'this' variable.**

- **Always references the enclosing object.**

- **Used by methods to access properties of the enclosing object.**

```
98  var obj1 = {
99      name : 'Waterford',
100     print : function() {console.log(this.name)}
101     }
102  var obj2 = {
103     name : 'Joe Bloggs',
104     print : function() {console.log(this.name)}
105     }
106  obj1.print()    // Waterford
107  obj2.print()    // Joe Bloggs
108
```

# Anonymous functions.

- You can define a function without giving it a name:

```
function( ... ) { .... }
```

- Mainly used for “**callbacks**” - when a function/method needs another function as an argument, which it calls.
  - EX. The `setTimeout` system function.

```
110 setTimeout(function() {console.log('After 1000 miliseconds')}, 1000)
111 console.log('Immediately')
```

```
Immediately
After 1000 miliseconds
```

- [Note: Any type of function (declaration, expression, method) can be used as a callback, not just anonymous functions.]

# Anonymous functions.

- A more elegant way of processing an array.
  - Objective: Display every number  $> 20$  from the array.

```
var nums = [12,22,5,28]
nums.forEach(function(entry) {
  if (entry > 20) {
    console.log(entry)
  }
})
```

- The anonymous function is called by `forEach()`, once for each entry in the array. The function's parameter (`entry`) will be set to the current array entry being processed.

```
var products = [ {name: 'Product 1', price: 110},
                  {name: 'Product 2', price: 90 },
                  {name: 'Product 3', price: 120 } ]
products.forEach(function(product) {
  product.price = product.price - product.price * 0.1
})
products.forEach(function(e) {console.log (e)})
```

# Constructors.

- **The object literal syntax is not efficient for creating multiple objects of a common 'type'.**
  - **Efficiency = Amount of source code.**

```
var customer1 = { name 'Joe Bloggs',  
  address : '1 Main St',  
  finances : { . . . . . },  
  computeTotal : function () { . . . . . },  
  adjustFinance : function (change) { . . . }  
}  
var customer2 = { name 'Pat Smith',  
  address : '2 High St',  
  finances : { . . . . . },  
  computeTotal : function () { . . . . . },  
  adjustFinance : function (change) { . . . }  
}  
var customer3 = . . . . .
```

Constructors solve  
this problem



# Constructors.

- **Constructor** - Function for creating (constructing) an object of a *custom type*.
  - Custom type examples: Customer, Product, Order, Student, Module, Lecture.
    - Idea borrowed from class-based languages, e.g. Java.
    - No classes in Javascript.
- **Convention: Capitalize function name to distinguish it from ordinary functions.**

```
function Foo(. . . ) { ... }
```
- **Constructor call must be preceded by the `new` operator.**

```
var a_foo = new Foo( . . . )
```

# Constructors.

- **What happens when a constructor is called?**
  1. **A new (empty) object is created, ie. { } .**
  2. **The this variable is set to the new object.**
  3. **The function is executed.**
  4. **The default return value is the object referenced by this.**

```
function Customer (name_in,address_in,finances_in) {  
    this.name = name_in  
    this.address = address_in  
    this.finances = finances_in  
    this.computeTotal = function () { . . . . }  
    this.changeFinannce = function (change) { . . . . }  
}  
var customer1 = new Customer ('Joe Bloggs','I Main St.', { . . . } )  
var customer1 = new Customer ('Pat Smith','2 High St.', { . . . } )  
console.log(customer1.name)    // Joe Bloggs  
var total = customer1.computeTotal()
```