

# Web Application Development

---

Produced  
by

David Drohan ([ddrohan@wit.ie](mailto:ddrohan@wit.ie))

Department of Computing & Mathematics  
Waterford Institute of Technology

<http://www.wit.ie>



Waterford Institute of Technology  
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE



# JavaScript

---

JAVASCRIPT FUNDAMENTALS



# Agenda

---

- ❑ Background & Introduction
- ❑ Data Types, Objects & Arrays
- ❑ Looping & Iteration
- ❑ Functions, Methods & Constructors

---

# What is JavaScript?

# JavaScript is...

---

- ❑ a lightweight interpreted or JIT-compiled programming language with first-class functions. (functions treated like variables)
- ❑ While it is most well-known as the scripting language for Web pages, many non-browser environments also use it, such as Node.js, Apache CouchDB and Adobe Acrobat.
- ❑ JavaScript is a prototype-based, multi-paradigm, dynamic language, supporting object oriented, imperative, and declarative (e.g. functional programming) styles.

# The JavaScript Engine

---

- ❑ At present, JavaScript can execute not only in the browser, but also on the server, or actually on any device where there exists a special program called [the JavaScript engine](#).
- ❑ The browser has an embedded engine, sometimes it's also called a “JavaScript virtual machine”.
- ❑ Different engines have different “codenames”, for example:
  - ❑ [V8](#) – in Chrome and Opera.
  - ❑ [SpiderMonkey](#) – in Firefox.
  - ❑ ...There are other codenames like “Trident”, “Chakra” for different versions of IE, “ChakraCore” for Microsoft Edge, “Nitro” and “SquirrelFish” for Safari etc.

# How Engines Work

---

- Engines are complicated. But the basics are easy.
  1. The engine (embedded if it's a browser) reads (“parses”) the script.
  2. Then it converts (“compiles”) the script to the machine language.
  3. And then the machine code runs, pretty fast.
  
- The engine applies optimizations on every stage of the process. It even watches the compiled script as it runs, analyzes the data that flows through it and applies optimizations to the machine code based on that knowledge. Basically, scripts are pretty fast.

# JavaScript is *NOT* Java

---

- ❑ JavaScript was written was created in 10 days in May 1995 by Brendan Eich.
- ❑ JavaScript was originally called Mocha and was renamed to LiveScript before being renamed again to JavaScript.
- ❑ The standard for JavaScript is ECMAScript. As of 2012, all modern browsers fully support ECMAScript 5.1. Older browsers support at least ECMAScript 3. On June 17, 2015, ECMA International published the sixth major version of ECMAScript, which is officially called ECMAScript 2015, and was initially referred to as ECMAScript 6 or ES6. Since then, ECMAScript standards are on yearly release cycles.
- ❑ The latest draft version is currently ECMAScript 2018.



---

# Why JavaScript?

(or maybe not?!)

# Reason #1

---

- ❑ JavaScript is very easy to learn. No setup is required; it's built right into the web browser! Just start writing code and see the results immediately in your browser.
- ❑ BUT - I pity would-be software engineers who are too intimidated and lethargic to follow simple installation instructions for languages like Python, Ruby, Smalltalk, Go and Elixir. They lack the kind of initiative, curiosity and determination that is so necessary to becoming an IT professional.
- ❑ Once you overcome *the huge, overbearing obstacle* of installing one of these languages, you'll find that the syntax is wonderfully easy and inviting. They'll make JavaScript look like the awkward stepchild that it is in the world of engineering.

# Reason #2

---

- ❑ JavaScript is used *everywhere*...web browser (Angular, React), server-side (Node), mobile, desktop, games, Internet of Things, robotics, virtual reality, etc.
- ❑ **BUT** - JavaScript dominates in web development because you have no choice—it's the native language of the web browser. *Outside of the web*, is worse, for example, in the mobile space, JavaScript has no hope of displacing Java (for Android) and Objective-C/Swift (for iOS).
- ❑ On the desktop, cannot compete with C++ and Java. In games, C++ and C# rule. Internet of Things? We're looking at C, Python, Java, even Perl! Robotics means C/C++, Python, Java and Smalltalk. And so on.
- ❑ Do you see a pattern? In every case, solid programming languages win the day. JavaScript is objectively a very flawed language. That it has managed to be shoehorned into these different areas is a testament to the JavaScript community's creativity, but we have to face the fact that, in practice, JavaScript is not gaining traction. **JavaScript is used for web development – Period.**

# Reason #3

---

- ❑ Node is super popular. Proof: there are over 30,000 NPM packages available!
- ❑ **BUT** - How many of these 30,000+ packages have significant usage? Probably relatively few. The fact that so many NPM packages exist shows the “gold rush” mentality that most contributors have. They hope to cash in on the JavaScript hype, or at least make a name for themselves in the open source community. Open source is another good example. Open source has countless thousands of contributions, most of which never see the light of day.

# Reason #4

---

- ❑ There are lots of high-paying jobs for JavaScript developers. What a great way to start your IT career!
- ❑ **BUT** - It's true, there are lots and lots of JavaScript job postings. And all of them are related to front-end web development or Node. You won't find any postings for JavaScript programming in mobile, desktop, Internet of Things, games, robotics, virtual reality, etc. *If all you want to do is write web apps, then JavaScript may be your ticket.* But it's hardly the path to a healthy career in IT, where there are so many other exciting developments going on, such as Artificial Intelligence, robotics, Big Data, Internet of Things, cloud computing, scientific modelling and simulation, Virtual Reality, etc., which use grown-up languages like Java, Python, C/C++, Go.

# Reason #5

---

- JavaScript is an incredibly *expressive and powerful* language.
- BUT - Is JavaScript any more expressive than, say, Clojure, Scheme, Erlang/Elixir, Haskell, C#, or even Smalltalk? Smalltalk has first-class functions and closures, which means that it can do anything JavaScript can do.

So there you have it: the five top reasons to use JavaScript.

Good luck with it. You'll need it.



Richard Kenneth Eng [Follow](#)

Mr. Smalltalk: <https://medium.com/p/domo-arigato-mr-smalltalk-aa84e245beb9>

May 28, 2016 · 3 min read

---

# Some Facts & Figures (Early 2018)

## Tiobe Programming Index

---

For decades, Tiobe (the software quality company) has generated an [index of the most popular programming languages](#). They update this list monthly, pulling in data from hundreds of sources around the world. For more on how the Tiobe Index is calculated, see [here](#).

---

## Indeed.com

As the world's largest job search engine, [Indeed](#) represents a good measurement of the most in-demand programming jobs. We looked at the number of job openings for the top 50 programming languages on the Tiobe Index.

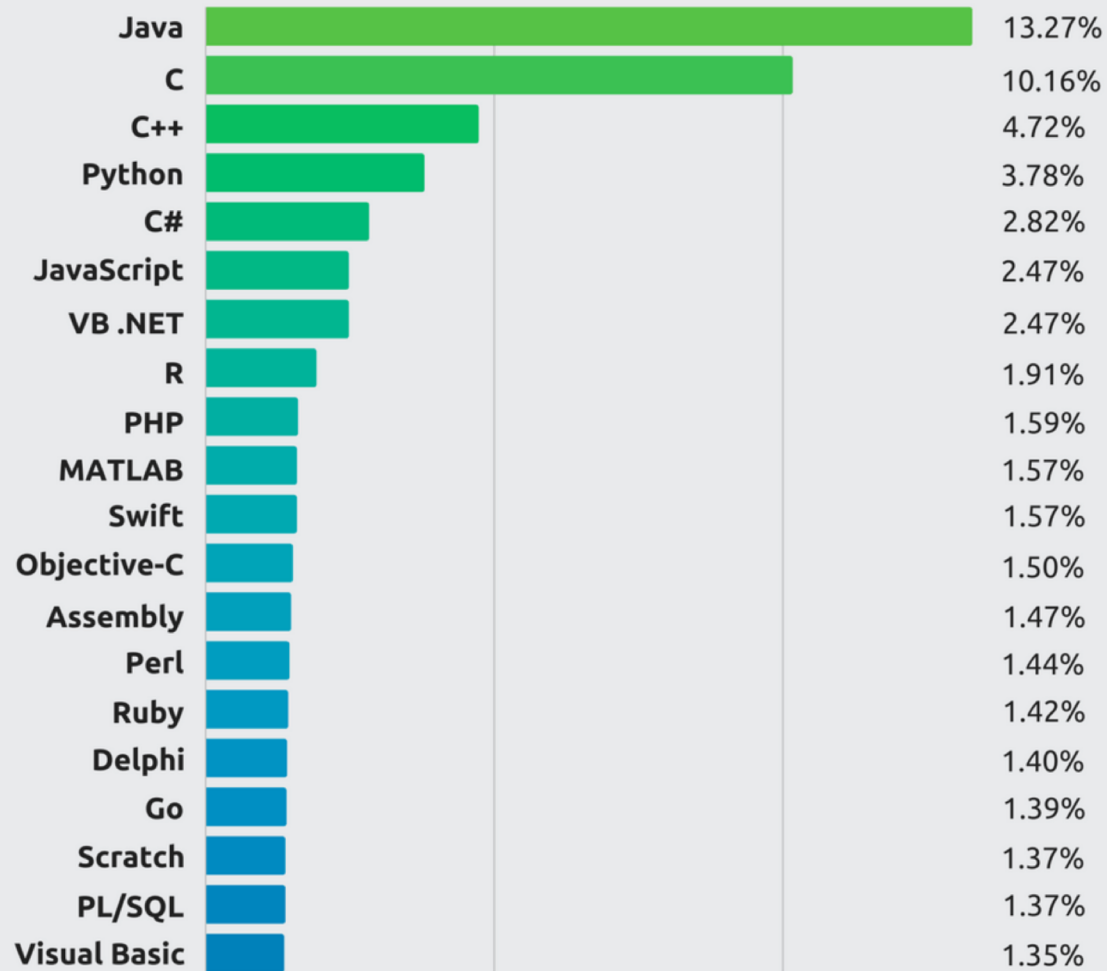
## GitHub

GitHub is one of the largest code repositories in use today. Every year they create a [Year in Review](#) report, sharing statistics about their programmer community. We cite their metric "Pull Requests Opened" as another indicator of language popularity.



# Top Programming Languages

Tiobe Index - December 2017

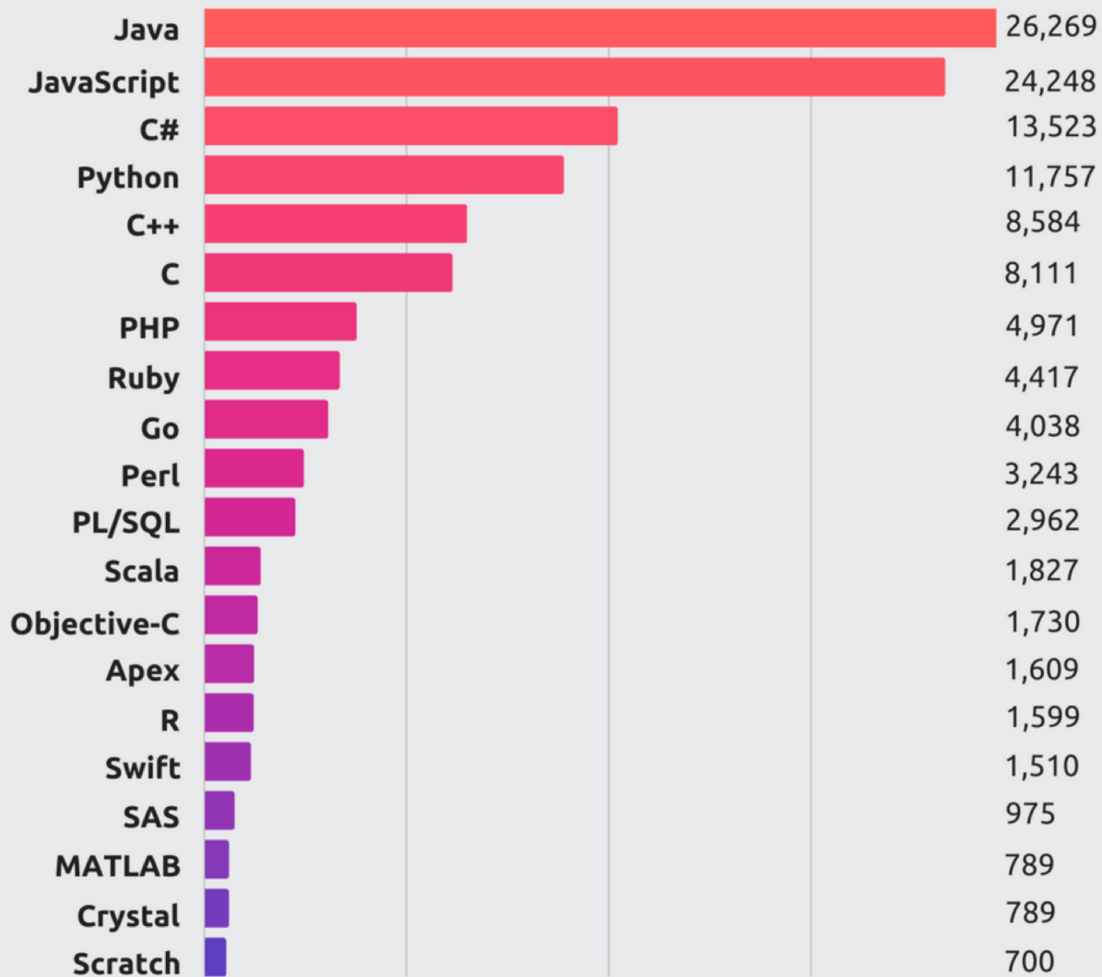


The graph above shows the top 20 most popular programming languages as of December 2017. The Tiobe Index works like market share; the percentage is the amount of “market share” a language holds. All of the languages combined total 100%.

Tiobe factors in variables like the number of professional developers worldwide, training courses, and third-party vendors. Most of this information comes from analyzing search engine results.

# Most In-Demand Languages

Indeed Job Openings - Dec. 2017

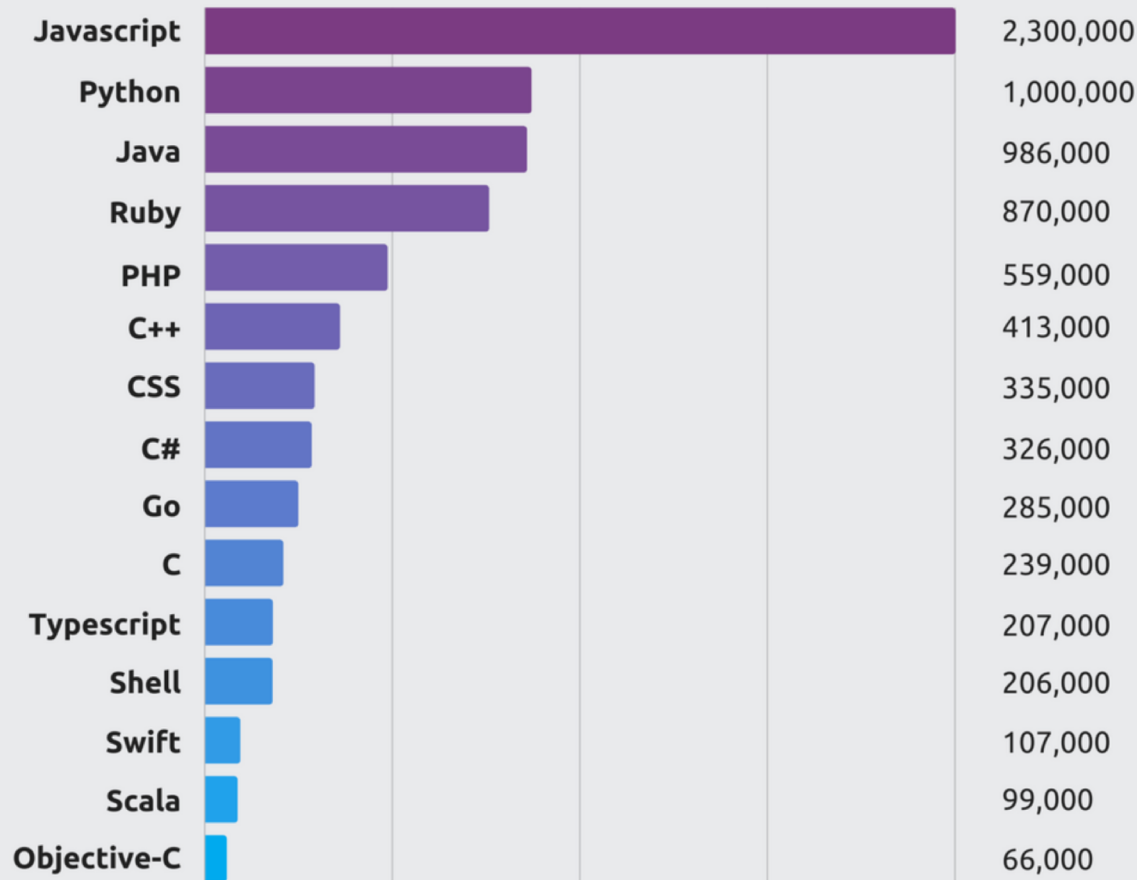


This graph charts the most in-demand jobs according to [Indeed.com](https://www.indeed.com). We took the 50 most popular languages from the Tiobe Index and searched for "(Language name) Developer". We put the name of the language in quotes to make our results more accurate.

It makes sense that the most popular languages have the most job openings. However, it's interesting to see less popular languages, like Apex (Salesforce native language), in the top 20. This might indicate an upward trend in popularity for these languages.

# Most Pull Requests 2017

GitHub



The [Github Year in Review](#) gives us the top 15 pull requests from their community. Pull requests are an indicator of the amount of code being written.

According to [OSS-Watch](#), "A pull request is a method of submitting contributions to an open development project." Javascript has a massive footprint on GitHub, with more than twice the number of pull requests than the second-leading language.

In 2017, Python overtook Java as the second most popular language by pull requests.

---

# The Basics

# Objects & Primitives

---

- ❑ Everything is either a primitive or an object.
- ❑ Objects are *ALWAYS* passed by reference
- ❑ Primitives are *ALWAYS* passed by value
- ❑ Objects in JavaScript are mutable keyed collections/dictionaries.
- ❑ JavaScript uses *prototypes* for inheritance. There is no such thing as a *class* in JavaScript. (allows you to add properties 'on the fly', amongst other things )

<https://docs.microsoft.com/en-us/scripting/javascript/advanced/prototypes-and-prototype-inheritance>

# Primitive Types

---

- ❑ JavaScript has six primitive types:
  - ❑ Boolean
  - ❑ Null
  - ❑ Undefined (yes, this is a type)
  - ❑ Number (can be a number between  $-(2^{53} - 1)$  and  $2^{53} - 1$ , NaN, `-Infinity`, or `Infinity`).
  - ❑ String (single or double quotes declares a string literal)
  - ❑ Symbol (new in ECMAScript 6)

# Variables

---

- ❑ JavaScript is an **untyped** language.
- ❑ Variables are declared using the `var` keyword (or most recently `let` or `const`)

## Examples:

- `var name;` - creates variable `name` of type `undefined`.
- `var name = 'Bloggs';` - string literal
- `var age = 18;` - declaring a number literal
- `var hasFriends = false;` - declaring a boolean
- `var significantOther = null;`

# Control Statements

---

```
1  // if statement syntax is identical to C++/Java
2  if (condition) {
3  } else if (condition) {
4  } else {
5  }
6
7  // ternary syntax is just like C++/Java
8  var a = condition ? val_if_true : val_if_false;
9
10 for (initializer; condition; incrementor) {
11     // for loop syntax is identical
12 }
13
14 for (var prop in obj) {
15     obj[prop].doThing(); // prop is the key
16                         // could be a number or a string
17 }
```



# Objects: Basics

---

- ❑ 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:
  - ❑ Example:

```
var me = { first_name: 'Dave', last_name: 'Drohan' }
```

# Objects: Basics

---

- ❑ 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 = 'Joe'`  
`me[ 'last_name' ] = 'Bloggs'`
  
- ❑ Subscript notation allows the subscript be a variable reference.  
`var foo = 'last_name'`  
`me[foo] = .....`

# Objects: Inheritance & the Prototype Chain

---

- ❑ Every JavaScript object is linked to a *prototype*. If a member is not found in an object (i.e. if `obj.foobar == undefined`) then the prototype is searched. It defines a sort of “default” set of values for the object.
- ❑ “Empty” objects start with `Object.prototype` defined as their prototype.
- ❑ You can set the prototype of an object to another object (or to `undefined`) by calling `myObj.prototype = otherObj;`
- ❑ Since the prototype of an object is just another object, it too can have a prototype. Hence the *prototype chain*. When you access a property of an object, the whole prototype chain is searched for it.
- ❑ The prototype relationship is a dynamic relationship. If a property is added to the prototype, it is automatically visible to all objects based on that prototype.

# Objects: Syntax

---

```
1  var myObj = { // this is an object literal
2      a: 3,
3      'b': 'JavaScript',
4      'is-awesome?': true,
5      doSomething: function () {
6          console.log(this.a); // 3
7          console.log(a); // error
8      }, // trailing commas are allowed
9  };
10 myObj.doSomething();
11 console.log(myObj.b, myObj['is-awesome?']);
```

## Output:

```
1  3
2  error: a is undefined
3  JavaScript true
```

# Objects are dynamic

---

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


```
var me = { first_name: 'Joe', last_name: 'Bloggs' }  
me.age = 21
```

# Objects: property

---

- A property value can be a variable reference.

```
var my_name = { first: 'Joe', last: 'Bloggs' }  
  
var me = { name: my_name,  
          age: 21,  
          address: '1 Main Street'  
        }  
  
console.log(me.name.first) // Joe
```



# Objects: Arrays

---

□ JavaScript arrays are basically vectors (and are also objects).

```
1 var arr = [1, 'a', {}, [], true];
2 arr[0] = 'not a number';
3 arr.push('this is basically a vector'); (also, pop, shift, unshift, join etc)
4 console.log(arr);
```

## Output:

```
1 [ 'not a number', 'a', {}, [], true, 'this is basically a vector' ]
```

*Note that the elements of an array do not have to be the same type.*

# Array Data Structure

---

- ❑ Definition: 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:



# Functions

---

- ❑ Functions are just objects with two special properties: a context (scope) and the function code.
- ❑ Functions can be defined anywhere where an object can be defined and can be stored in variables.
- ❑ Functions can access all arguments passed to a function via the arguments variable.
- ❑ Functions can access the callee of a function (`callee.func()`) via the `this` variable.
- ❑ Functions can also have named parameters.
- ❑ Functions always return a value. If no return is explicitly specified, the function will return `undefined`.

# Functions

---

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

# Functions: Callback

---

- Since JavaScript functions are objects, they can be passed just like other objects.

```
1  function doStuff(callback) {
2      // do a bunch of processing
3      var x = 3;
4      console.log('in doStuff');
5      callback(x);
6  }
7
8  doStuff(function(x) {
9      console.log(x * 3);
10 });
```

## Output:

```
1  in doStuff
2  9
```

# Functions: New

---

- ❑ JavaScript functions **can** be invoked with the `new` keyword, mimicking traditional class-based languages:

```
1     function Thing(val) {  
2         this.v = val;  
3     }  
4  
5     var t = new Thing(12);  
6     console.log(t.v); // prints 12
```

# 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.

# 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)  
11 // .....|  
12 function sayHello(person) { ... }  
18 }  
19
```

Collapsed for convenience

- ❑ 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 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’]



# Functions: (fat) arrow notation =>

- ❑ 'Syntactic sugar' (and ES5+ support) allows us to use the arrow notation to replace the `function` keyword.

```
1  function doStuff(callback) {  
2      // do a bunch of processing  
3      var x = 3;  
4      console.log('in doStuff now');  
5      callback(x);  
6  }  
7  
8  doStuff(x => console.log(x * 3));
```

## Output:

```
1  in doStuff now  
2  9
```

# Methods

---

- ❑ A property value of an object can be a function, termed a method.
- ❑ The same form of function definition as function expressions.
- ❑ Syntax: 

```
var obj = { .....  
          methodX : function(.....) { ..... },  
          ..... }
```
- ❑ Calling method syntax: `obj.methodX(.....)`
- ❑ Methods of an object can be redefined or added at any time.
  - ❑ JS is dynamic!!
- ❑ Methods must be defined before use.
- ❑ [ *A bit on Application design* – The dominant design methodology encourages encapsulating state (data) and behavior (methods) into units called classes. In JS, most custom objects are a mix of state and methods, where the latter manipulates the state. ]

# Methods

```
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)
96 }
```

Use 'this' to reference the enclosing object

## Output:

- 1 Full worth = 25.5
- 2 { first: 'Joe', last: 'Bloggs', middle: 'Paul' }
- 3 { 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.
  - ❑ Example - The setTimeout system function.

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

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

## Output:

```
1  Immediately  
2  After 1000 milliseconds
```

# 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.

Constructors solve  
this problem

```
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.

---

- ❑ 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()
```

# Scope

---

- ❑ There are two scopes in JavaScript: global and function.
- ❑ Variables declared outside of a function are automatically in the global scope.
- ❑ Variables declared within a function *without* the `var` keyword are also in the global scope.

```
1  var a = 2;
2  (function () {
3      b = 3
4      var c = 5;
5  }) (); // this creates and invokes the function immediately
6
7  console.log(a); // logs 2
8  console.log(b); // logs 3
9  console.log(c); // error since c is undefined in global scope
```

# Private Variables

---

□ You can simulate private variables like so:

```
1  var Dog = function(name) {
2      var gender = 'male';
3      this.name = name;
4      this.isBoy = function () {
5          return gender == 'male';
6      };
7  };
8
9  var myDog = new Dog('Sebastian');
10 console.log(myDog.gender); // logs undefined
11 console.log(myDog.name); // logs 'Sebastian'
12 console.log(myDog.isBoy()); // logs true
```

# Pitfalls: Variable Hoisting

---

- ❑ Variables are *hoisted* to the top of the function they are declared in. Thus, the following is entirely valid.

```
1  function scopeEx() {  
2      b = 5;  
3      console.log(b); // logs 5  
4      var b = 3  
5      console.log(b); // logs 3  
6  }
```

- ❑ This is confusing. Just declare your variables before you use them (and that's what `let` and `const` do now!)

# Pitfalls: Truthy, Falsy and `==` vs `===`

---

- ❑ JavaScript has the notion of being *truthy* and *falsey*.
- ❑ The following values are always falsy: `false`, `0`, `""`, `null`, `undefined`, `NaN`.
- ❑ Do not expect all falsy values to be equal to each other (`false == null` is `false`).
- ❑ JavaScript has two equality operators:
  - ❑ `==` compares without checking variable type. This will cast then compare.
  - ❑ `===` compares and checks variable type.

# DOM Manipulation

---

The *Document Object Model* is an API used by JavaScript to interact with the elements of an HTML document.

jQuery is great for simple DOM manipulation.

```
1 <div id="cool">Cool</div>
2 <div class="myCls">jQuery Demo</div>

1 var coolDiv = document.getElementById('cool'); // pure JS
2 coolDiv.style.background = 'blue';
3
4 var coolDiv = $('#cool'); // jQuery
5 coolDiv.css('background-color', 'blue');
```

jQuery does a ton of other useful things as well, but that's what the docs are for.

# Canvas Manipulation

---

- Many JS games use lots of HTML and CSS to draw the game, with some JS and DOM/JQuery-stuff for the logic. However, you can also draw the game directly using a Canvas. All you need then is a few lines of HTML and the rest can happen in your script. You can even create 3D stuff with WebGL or a 3rd party library like Three.js.

```
1 var c = document.getElementById("myCanvas");
2 var ctx = c.getContext("2d");
3 ctx.moveTo(0,0);
4 ctx.lineTo(200,100);
5 ctx.stroke();
```

# Additional Resources

---

A lot of this presentation was based off of *JavaScript: The Good Parts* by Douglas Crockford. This is an essential read for anyone interested in learning JavaScript for anything more than writing a few simple scripts.

MDN is the best resource for JavaScript documentation (<https://developer.mozilla.org/en-US/>).

**JSHint** (<http://jshint.com/about/>) is a tool which checks JavaScript syntax and helps prevent bugs in your code. JSHint has plugins for most IDEs and text editors. Here's a SO article on the Vim plugin:

<http://stackoverflow.com/questions/473478/vim-jslint/5893447>



# References

---

- ❑ <https://javascript.info>
- ❑ <https://gist.github.com/not-an-aardvark/cb9dbfba750e9a28cb78447491a1d079>
- ❑ <https://medium.com/javascript-non-grata/the-five-top-reasons-to-use-javascript-bd0c0917cf49>
- ❑ <https://stackify.com/popular-programming-languages-2018/>

Sumner Evans and Sam Sartor

November 10, 2016

---

# Questions?