JavaScript

as a dynamic, functional language

Wouter Groeneveld

Contents

Today we will talk about JS as a **language**.
We will not talk about JS as a way to manipulate the **DOM**.

How JS is used :-(

```
function oops() {
  doStuff();
  i = 10;
  copyPasta(this.i, 20);
}

function copypasta(i, j) {
  alert(new Date(i).getTime());
  return j;
}

<a onclick="oei();">lala</a>
```

Javascript is just like java!

How JS can be used used :-)

```
$(document).ready(function() {
  $("#link").click(MyStuff.oei);
});

var MyStuff = (function() {
  function privateStuff() { ... }
  return {
    oei: function() {
      ...
    }
  };
})();
<a id="link">lala</a>
```

Javascript is *completely different* from java!

Primitives & objects

Variable declaration

```
var variabele = 5;
var hello = "hello";

var 1 = “one”;      // syntax error
var _ = “omgh4x”; // ok
```

Primitives

1. number
2. string
3. boolean

immutable & case sensitive!

Special values

undefined

```
var a;
a === undefined;
```

null

```
var a = null;
a === null;
```

NaN

```
isNaN(parseInt("granny")) === true
```

Infinity

```
1 / 0 === Infinity
```

"typeof" keyword

returns strings:

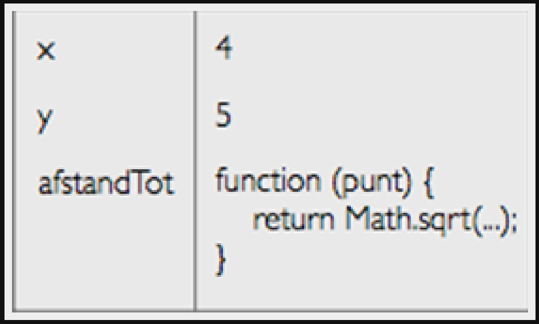
- object
- function
- string
- boolean
- number
- undefined

string utils

```
str.split
str.indexOf
str.replace(regex)
str.toLowerCase
//...
```

See https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/String.

Objects: a map of key/values



x	4
y	5

afstandTot	function (punt) {
	    return Math.sqrt(...);
	}

value can be a number, string, object, function

ways to create objects

literal

```
var location = {
  x: 3,
  y: 4,
  distanceTo: function () {
    var dx = this.x + this.y;
    return Math.sqrt(dx);
  }
}
```

new Object()

```
var location = new Object();
location.x = 3;
location.y = 4;
location.distanceTo = function () { … };
```

ways to create objects

Literal object syntax

`key : value`, and not `key = value1;`

```
// this:
var obj = {
  key1: value1,
  key2: function () {..},
  key3: value3
};

// not this:
var obj = {
  key1 = value1;
  function key2(){..};
  key3 = value3;
};
```

- read:

```
console.log(location.x);
console.log(location[‘x’];
```

- iterate:

```
for (var key in location)
  console.log(location[key
}
```

- add:

```
location.z = 1;
location[‘z’] = 1;
```

- modify:

```
location.x = 43;
location[‘x’] = 43;
```

- delete:

```
delete location.z;
delete location[‘zumba’];
```

Functions

function

```
function aFunction(name) {
  console.log("Hello, " + name);
}
```

function literals

```
var aFunction = function(name) {
  console.log("Hello, " + name);
}
```

Both definitions are equivalent!

function arguments

```
function wow() {
  console.log(arguments[1]);
}

wow("jos", "lowie"); // prints "lowie"
```

creating an object using a function

```
function createPoint(x,y) {
  return {
    x: x,
    y: y,
    distanceTo: function (otherPoint) {
      return Math.sqrt(...);
    }
  };
}
```

Arrays

```
var arr = ["a", "b", "c"];
```

iterating

```
x.forEach(function(i) {
   console.log(i);
});

for(var i = 0; i < ...)
```

Array utility functions

```
arr.length
arr.push, pop
arr.splice, slice
arr.shift, unshift
arr.sort
arr.filter, arr.map
// ...
```

See https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array.

An array is just an object

```
var objArr = {
  0: “dog”,
  1: “cat”
};

var arr = [ “dog”, “cat” ];
for(a in arr){console.log(a)};
```

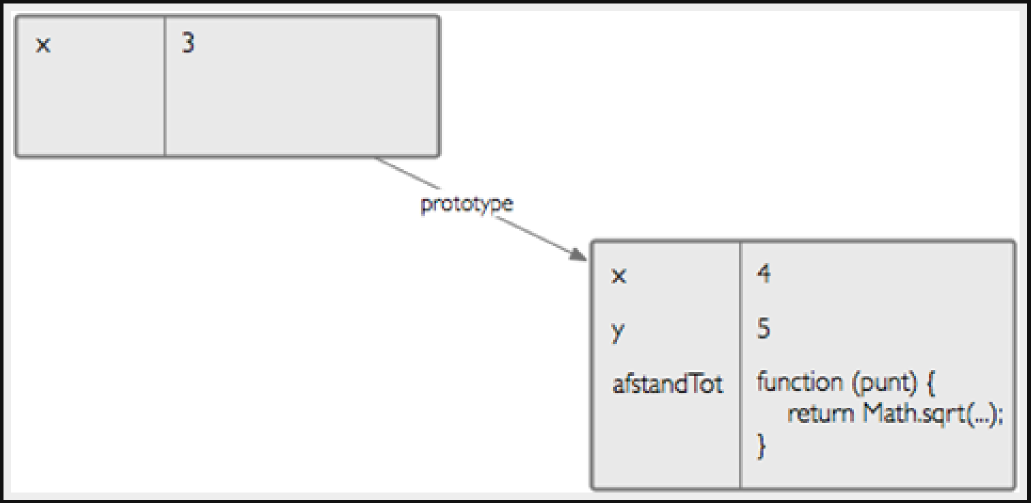
What does the code above do?

LAB 1

Jasmine

Prototypal inheritance

Prototypes



Object refers to another object (it's *prototype*).

Looking for a property

First in own object, then further down the chain.

Changing a property

Always in own object. (modify, delete)

How is this different from class inheritance? (Java, C#)

- classes don’t typically change at runtime
- difference between inheriting methods and fields
- difference between instance of class and class

Object.create()

```
var point = {
  distanceTo: function(otherPoint) {
    return Math.sqrt(...);
  }
};

var point1 = Object.create(point);
point1.x = 3;
point1.y = 4;
var point2 = Object.create(point);
```

Prototype = object, so can be changed

```
var proto = {
  wow: “wow man!”
};

var obj = Object.create(proto);
console.log(obj.wow);

proto.wow = “mind is blown”;
console.log(obj.wow);
```

Changing internal objects

```
Array.prototype.addFirstTwo = function () {
  return this[0] + this[1];
}
[1,2].addFirstTwo();
```

Be careful with this!

LAB 2

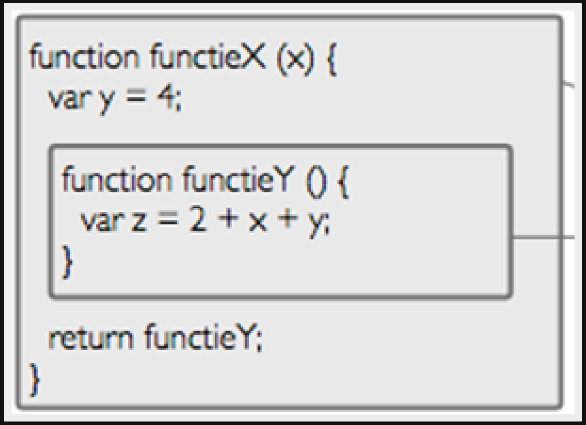
Jasmine

Closures

Closures - definition

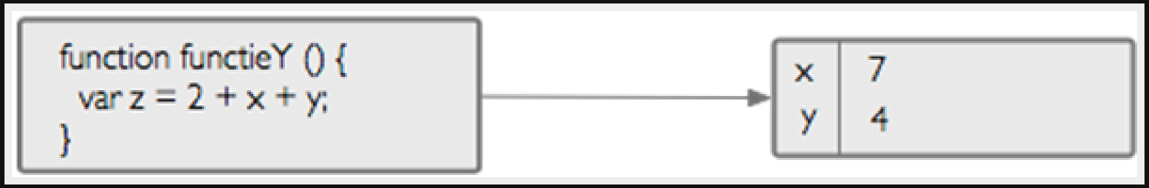
"a function that retains the environment in which it is created"

Function inside a function



```
function functieX (x) {
  var y = 4;

  function functieY () {
    var z = 2 + x + y;
  }

  return functieY;
}
```

Has access to variables from declared scope



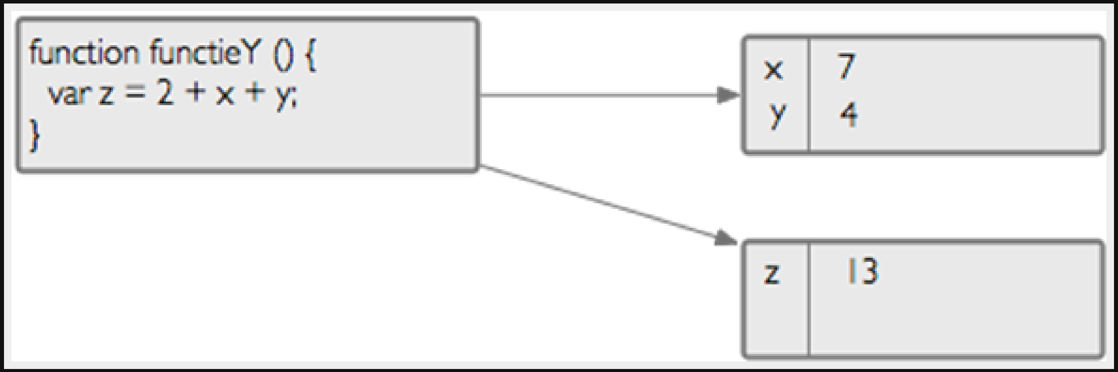
```
function functieY () {
  var z = 2 + x + y;
}
```

x	7
y	4

`functieX(7)()` - What happens?

Closures - definition

"a function that retains the environment in which it is created"

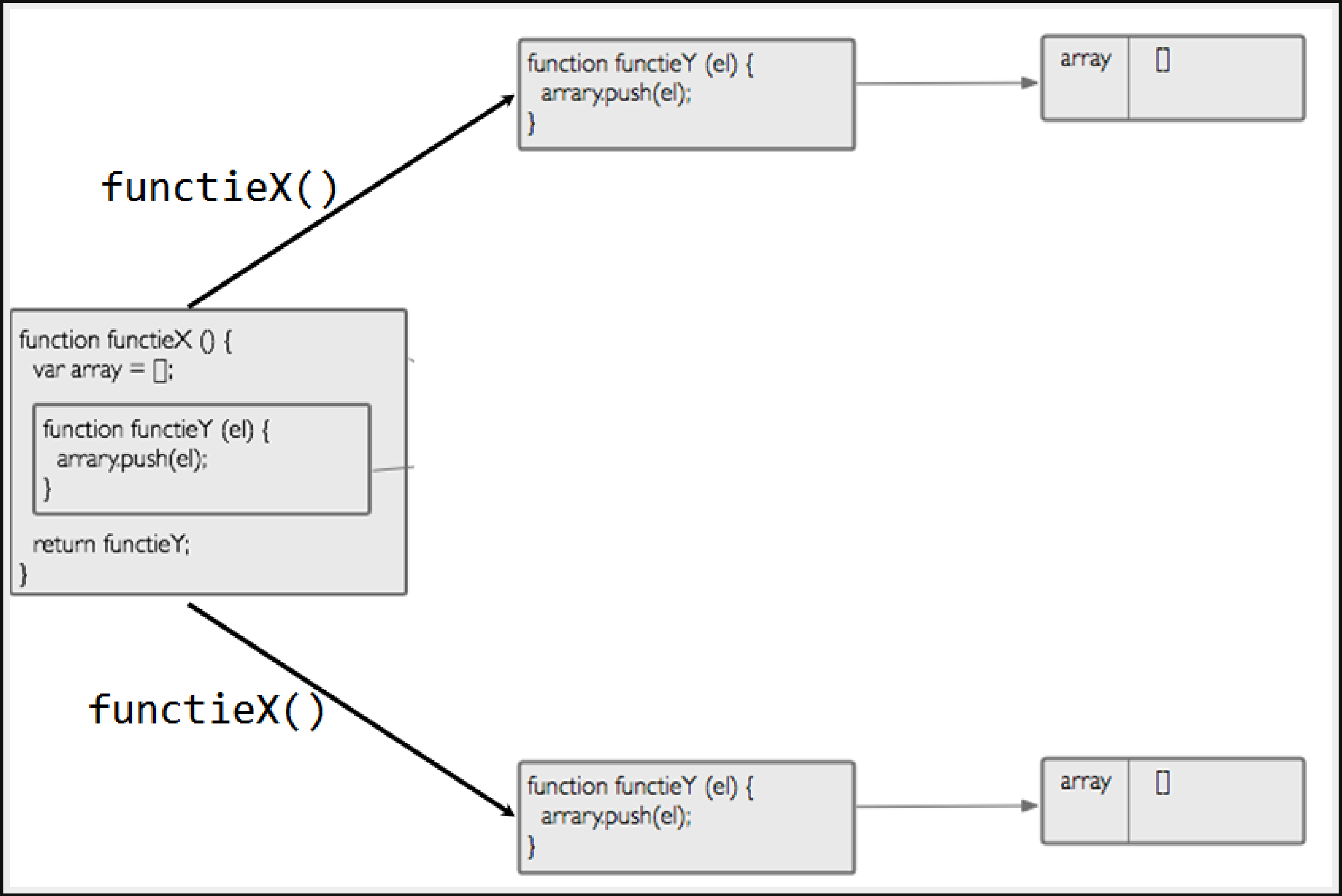


```
function functieY () {
  var z = 2 + x + y;
}
```

when functieY is called, a **new environment** is created for the local variables and parameters.

when looking for a variable, it will first look in this new environment, next it will look in the environment it has retained.

Closures - definition



```
function functieX () {
  var array = [];
  function functieY (el) {
    arrary.push(el);
  }
  return functieY;
}
```

```
function functieY (el) {
  arrary.push(el);
}
```

```
function functieY (el) {
  arrary.push(el);
}
```

every time you call a function, a **new** environment is created.

LAB 3

Jasmine

Scope

*So... for how long is an **environment** used?*

or differently put:

*For how long is the **scope** of a variable valid?*

Javascript does not have *block level* but **function level scope**!

Block level scope (Java, C#)

```
public void Test() {
   if (1 == 1) {
     var x = 4;
   }
   Debug.WriteLine(x); //error
}
```

function level scope (Javascript)

```
function test() {
  if (1 === 1) {
    var x = 4;
  }
  console.log(x); // prints 4
```

Hoisting

Local variables are automatically "pulled up":

```
var a = 3;
function f() {
  console.log(a);
  var a = 5;
}
```

becomes:

```
var a = 3;
function f() {
  var a;
  console.log(a); //prints undefined not 3
  a = 5;
}
```

Hoisting - best practice

define variables @ start of function

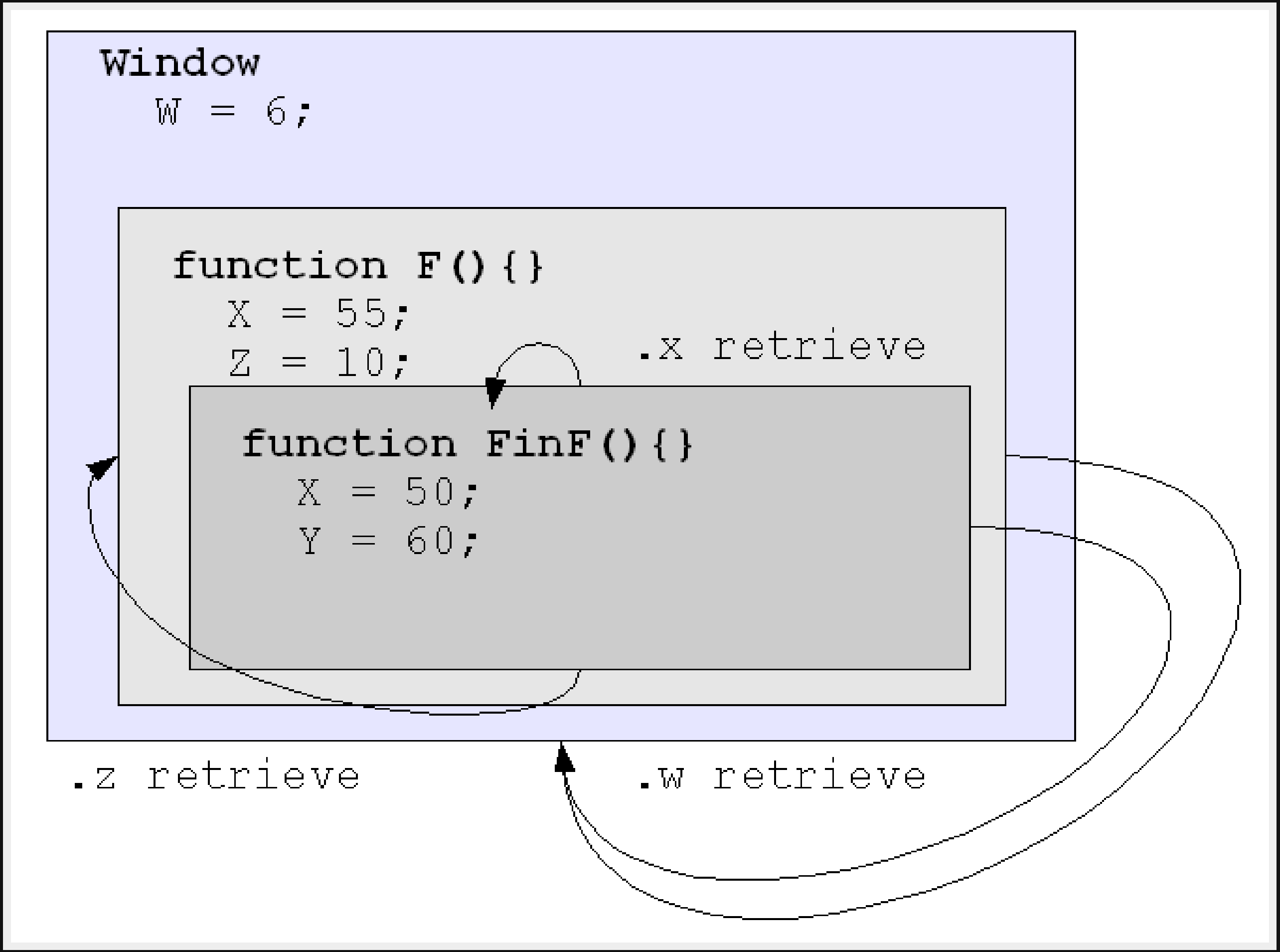
```
function a() {
  var a, b, c, ...;
  ...
  a = 5;
}
```

This makes it clearer what's going on.
JSLint will give a warning if you don't do this.

Scope chaining

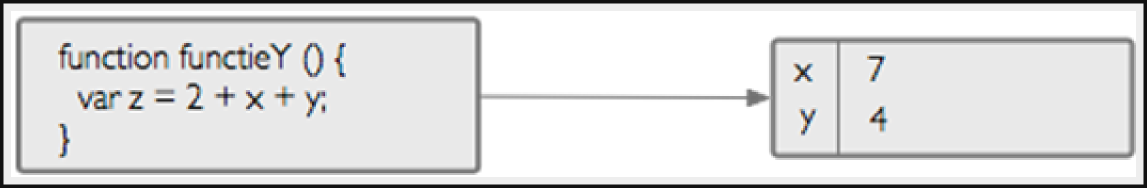
Toplevel scope = `window`!

```
w = 6; // same as ‘window.w = 6;’
         // same as ‘var w = 6;’

function F() {
  var X = 55;
  var Z = 10;

 function FinF(){
    var X = 50;
    var Y = 60;
 }
}
```


Window
W = 6;
function F(){}
X = 55;
Z = 10;
.x retrieve
function FinF(){}
X = 50;
Y = 60;
.z retrieve
.w retrieve

Encapsulation

Remember closures?


```
function functieY () {
  var z = 2 + x + y;
}
```

an environment is "private".
We can use this to encapsulate variables.

"private" variables for an object

```
function createPerson() {
  var name = 'Jos';
  return {
    getName: function() {
      return name;
    },
    setName: function(x) {
      name = x;
    }
  };
}
```

`name` is inaccessible outside the function scope:

```
var persoon = createPersoon();
persoon.getName(); // === “Jos”
persoon.name // === undefined
```

I heard you like closures...

... So I put a closure in a closure so you can wrap while you wrap.

```
function moduleCreator() {
  function createPerson() {
    var name = 'Jos';
    return {
      getName: function() {…}
    };
  }

  // createDog is not exposed
  function createDog() {…}

  return {
    createPerson: createPerson
  }
}
var MOD = moduleCreator();
var person = MOD.createPerson();
```

module pattern

```
(function(){})();
```

When to use a module?

- Avoid pollution of global namespace
- Duplication
- Plugin, framework, ...
- Reuse (component-based)

When not to use a module

- Code that is used only once on a single page
- Not a lot of code, not a lot of pollution

Keep it SIMPLE!

```javascript
function emulateBlockLevelScope() {
  var a = 5;
  if(a === 5) {
    (function () {
      var b = 555;
    })();
  }
  console.log(b);
}
```

Context in function

```
function standaloneDistanceTo(otherPoint) {
  return this.x + this.y;
}
var point1 = {
  x: 1,
  y: 1,
  distanceTo: standaloneDistanceTo
}
var point2 = {
  x: 2,
  y: 2,
  distanceTo: standaloneDistanceTo
}
```

What does `this` refer to?

this is determined when calling function

```
point1.distanceTo(point2);                    // this = point1
point2.distanceTo(point1);                    // this = point2
standaloneDistanceTo(point1);     // this = window (global obj in JS)
```

this can be explicitly passed when calling a function

```
standaloneDistanceTo.apply(point1, [point2]);      // this = point1
standaloneDistanceTo.call(point1, point2);// this = point1
```

Using this in callbacks

```
var person = {
  name: "jos",
  shout: function() {
    alert(“hey, ” + this.name);
  }
}

setTimeout(person.shout, 1000);
```

Why doesn't this work? - look at it from the point of view of `setTimeout()`:

```
function setTimeout(myCallback,millis) {
  //wait for millis
  myCallback();
}
```

this will be bound to window object because myCallback is called as a standalone function.

How do we fix this?

Use a closure

```
var person = {
  name: "jos",
  shout: function() {
    alert(“hey,” + this.name);
  }
}

setTimeout(function() {
  person.shout()
}, 1000);
```

Use bind method on function

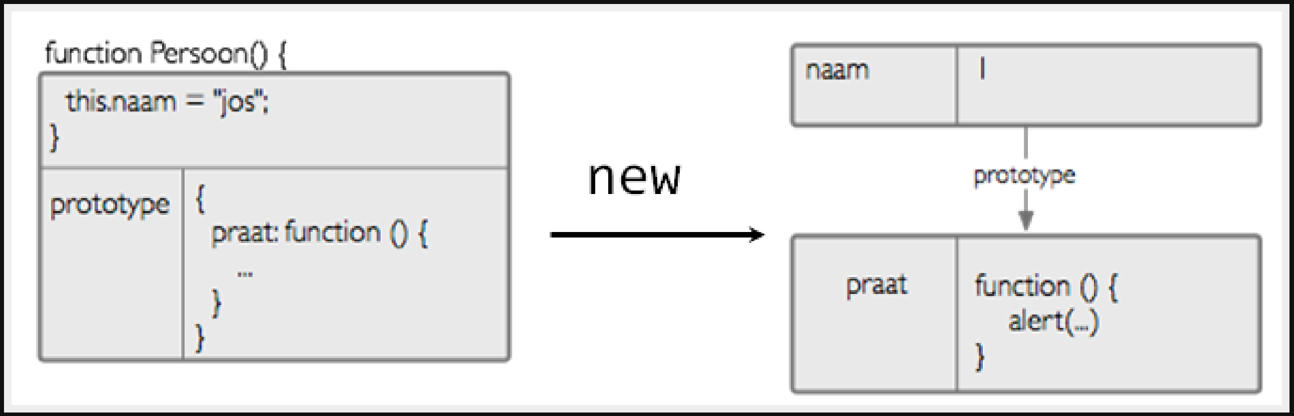
```
var person = {
  name: "jos",
  shout: function() {
    alert(“hey, ” + this.name);
  }
}

setTimeout(person.shout.bind(person), 1000);
```

LAB 4

Jasmine

Using the new operator

new operator: constructor functions



```
var jos = new Person();
```

is the same as

```
var jos = Object.create(Person.prototype);
Persoon.call(jos);
```

new operator: constructor functions

```
function Person() {
  this.name = “jos”;
  // no return statement
}
Person.prototype.talk = function () {...}

var jos = new Person();
jos.name === “jos";
jos.talk();
```

instanceof

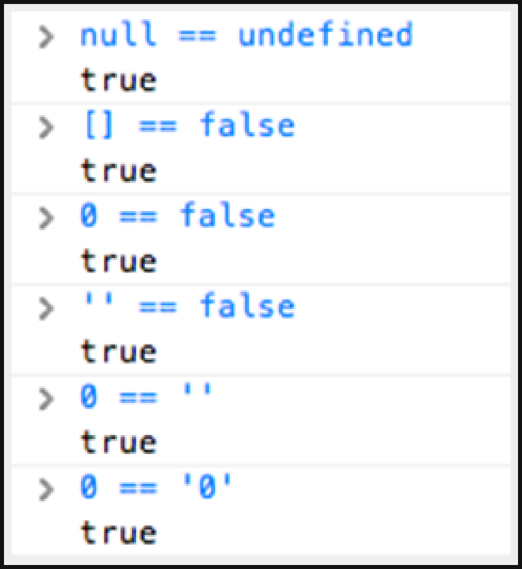
```
var jos = new Person();

jos instanceof Person === true;

// is same as:
Person.prototype.isPrototypeOf(jos)
```

Best practice: use `Object.isPrototypeOf()`! Why? It might look like class-based inheritance but it's still prototypal inheritance!.

misc: equality and identity

Why do we use === instead of == ?

The **Equality operator**(==) tries to cast both sides to the same type, resulting in "falsy/truthy" confusing outcomes.



```
> null == undefined
  true
> [] == false
  true
> 0 == false
  true
> '' == false
  true
> 0 == ''
  true
> 0 == '0'
  true
```

misc: ECMAScript?

ECMA = European Computer Manufacturers Association = Standardisation

Modern browsers implement EcmaScript 5+ = Javascript 8+

For IE<9 use ES5 shim: https://github.com/kriskowal/es5-shim.

More

http://brainbaking.com/wiki/code/javascript/home