JavaScript

as a

dynamic, functional

language

by:

Jan Sabbe &
Wouter Groeneveld

1. Context
2. Primitives & objects
3. Prototypal inheritance
4. Functions & closures
5. Scopes & contexts

Today we will talk about JavaScript as a **language**.

We will not talk about JavaScript as a way to manipulate the **DOM**

How JavaScript is used ☹

JavaScript is just like Java!

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

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

</script>
<a onclick=“oei();”>lala</a>
```

How JavaScript ***can*** be used ☺

```
<script>
$(document).ready(function() {
  $(“#link”).click(MyStuff.oei);
});

var MyStuff = (function() {
  function privateStuff() { ... }
  return {
    oei: function() {
      ...
    }
  };
})();
</script>
<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

```
number
string
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:

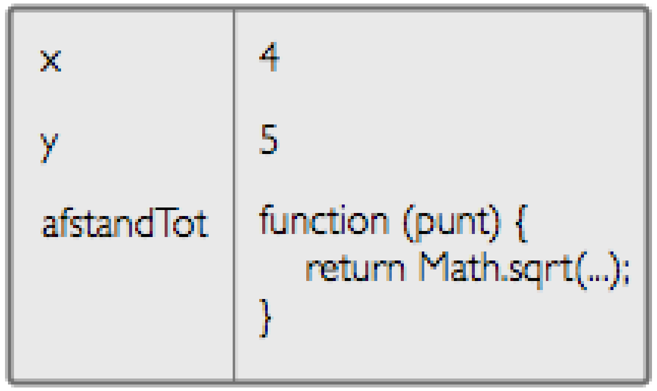
```
“object”
“function”
“string”
“boolean”
“number”
“undefined”
```

String utility functions

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

See API:
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*** – common rookie mistake

do 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’];
```

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(...);
    }
  };
}
```

Array

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

Iterate

```
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 API:
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” ];
```

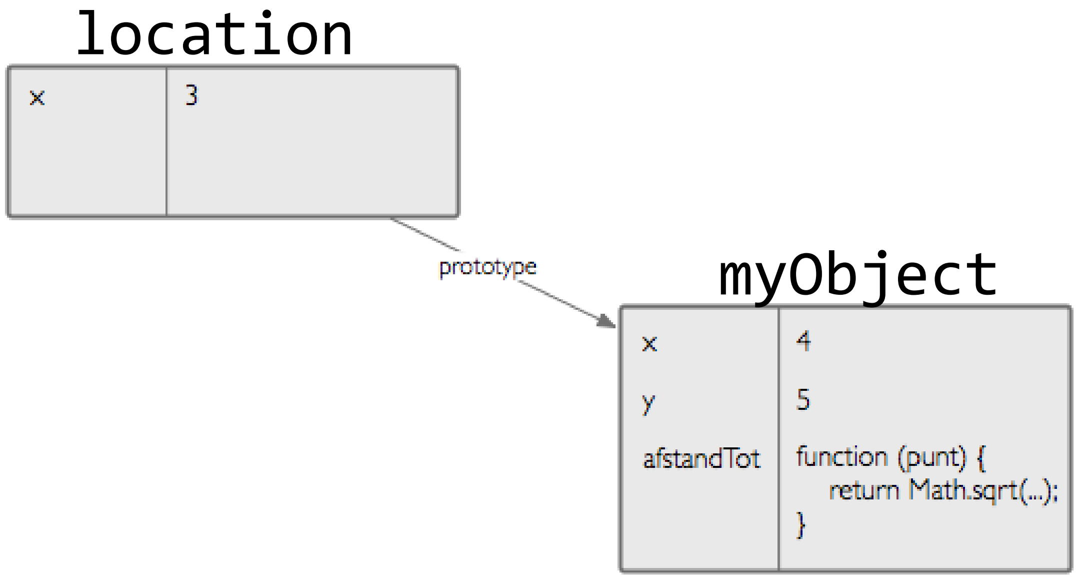
Question: what does the following do?

```
for(a in arr){console.log(a)};
```

Lab 1

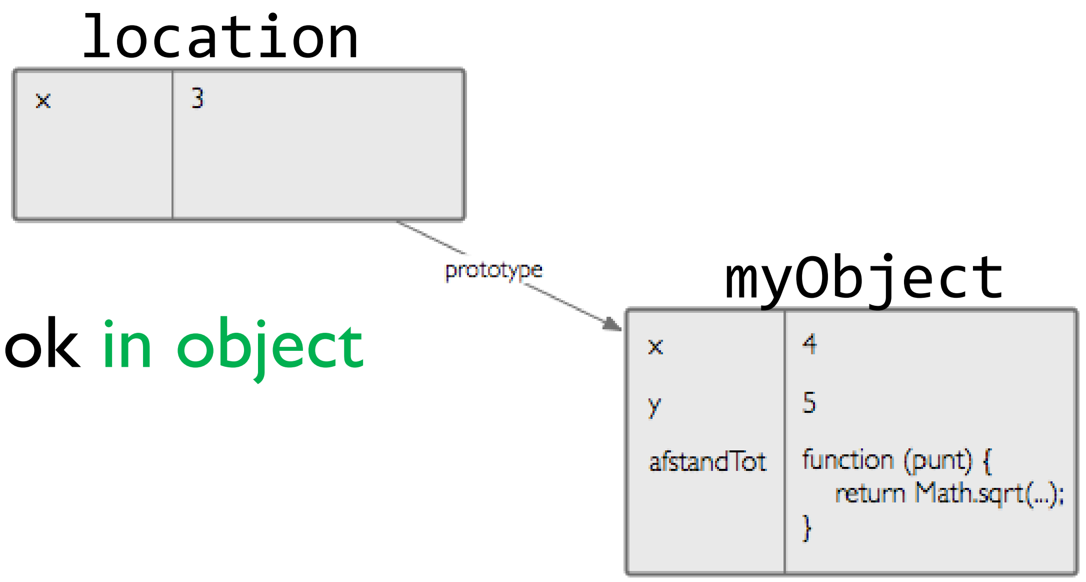
Prototypal inheritance

prototypes



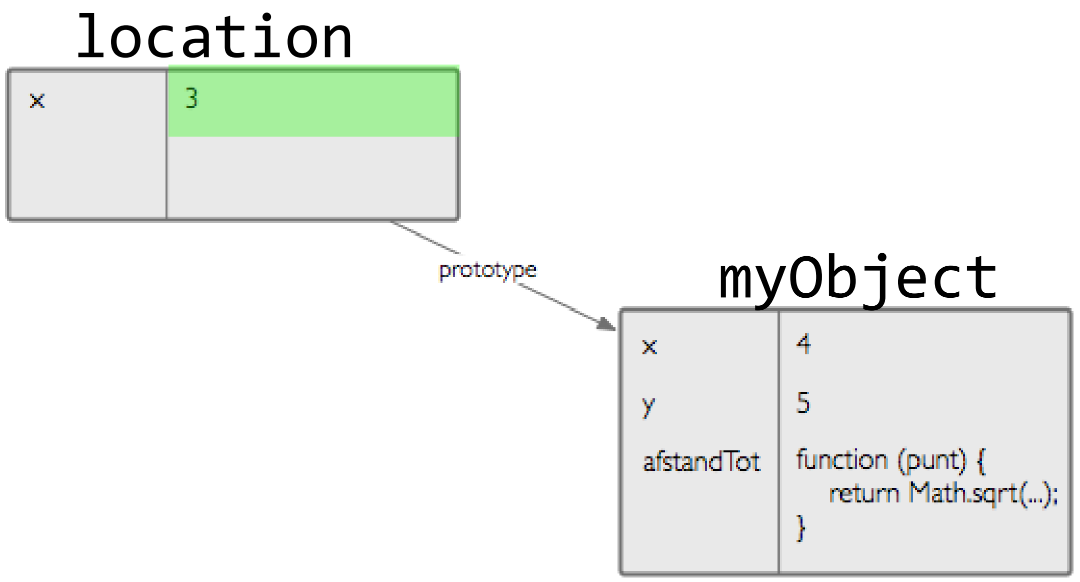
object refers to another object (prototype)

looking for a property



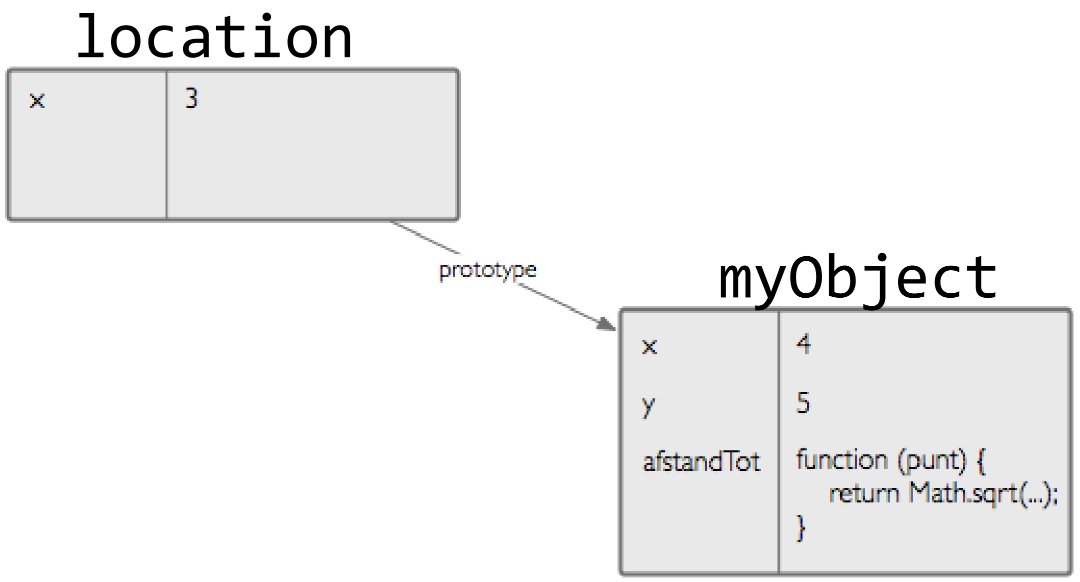
first look in object

looking for a property



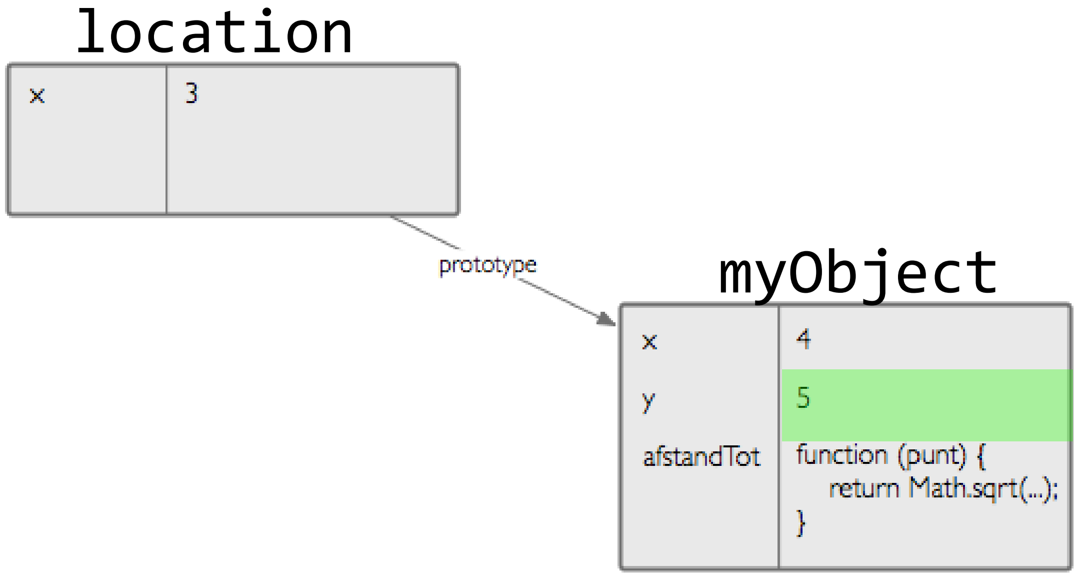
```
location.x
```

looking for a property



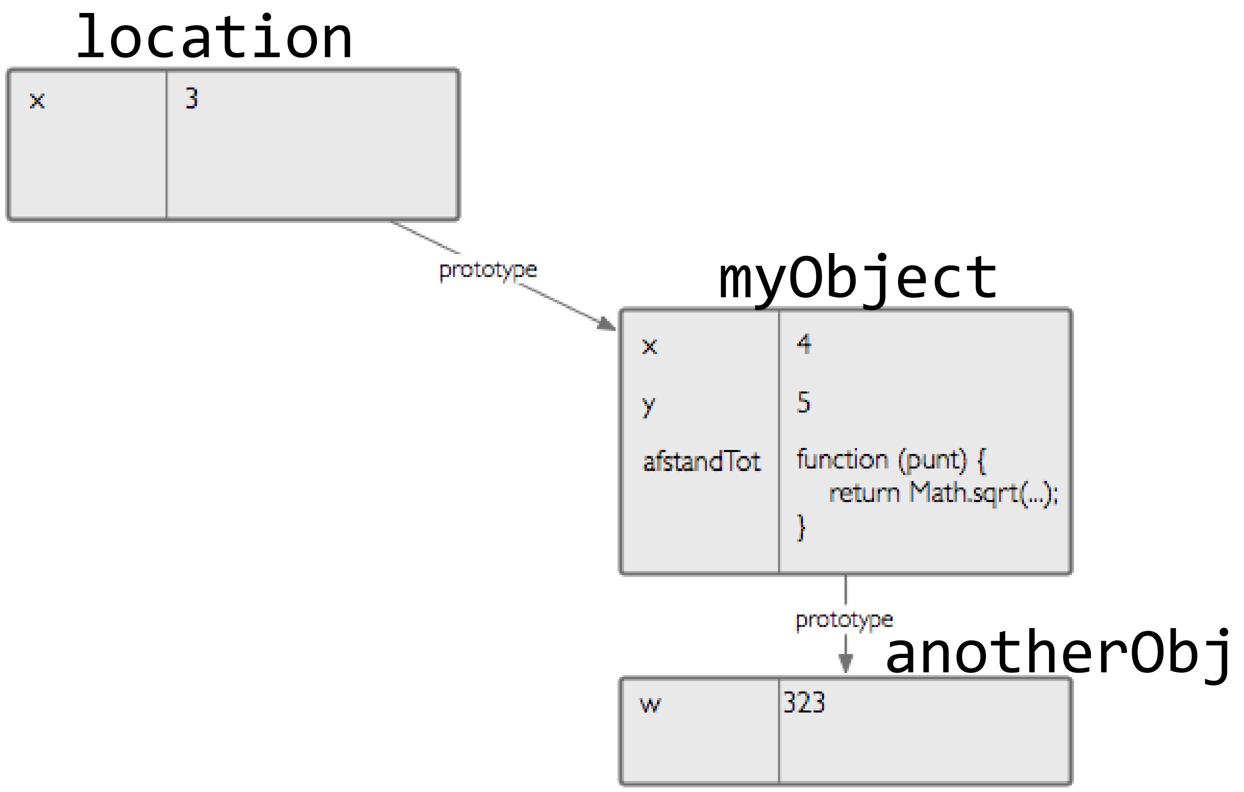
if nothing found,
look in prototype

looking for a property



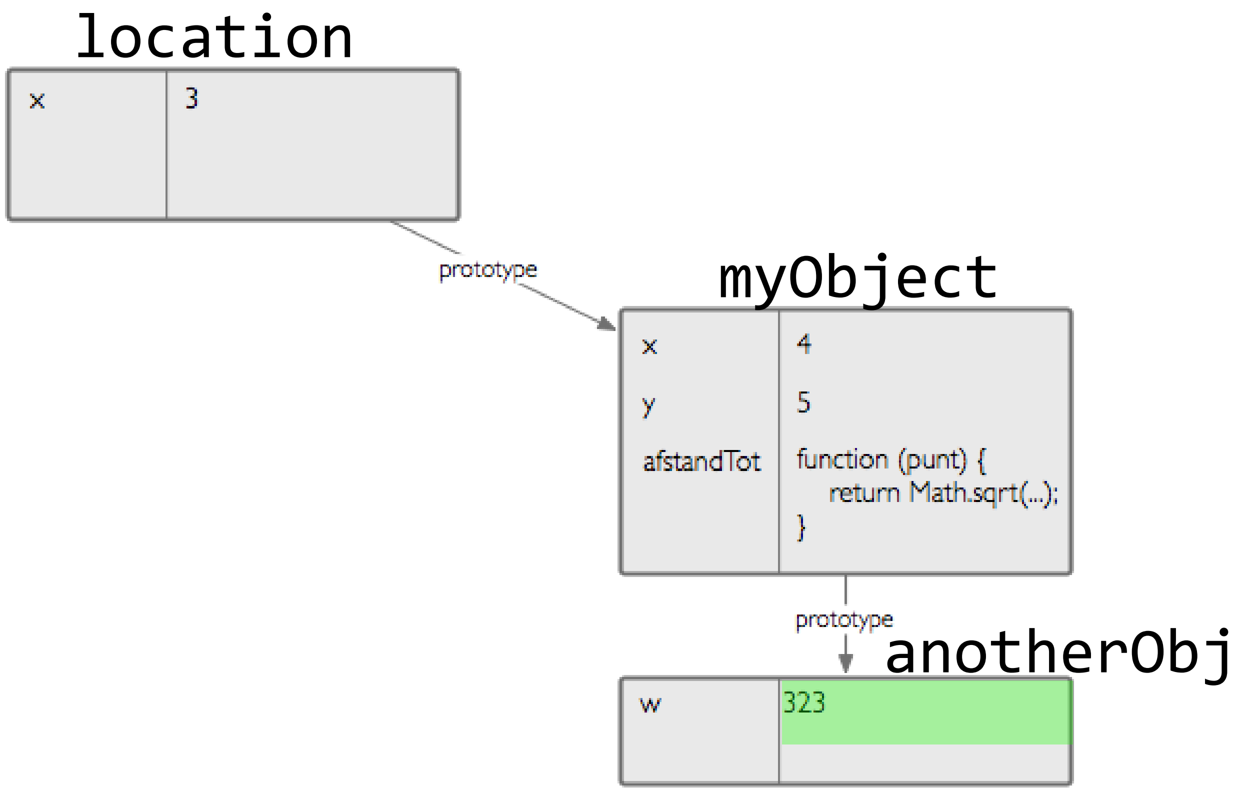
```
location.y
```

looking for a property



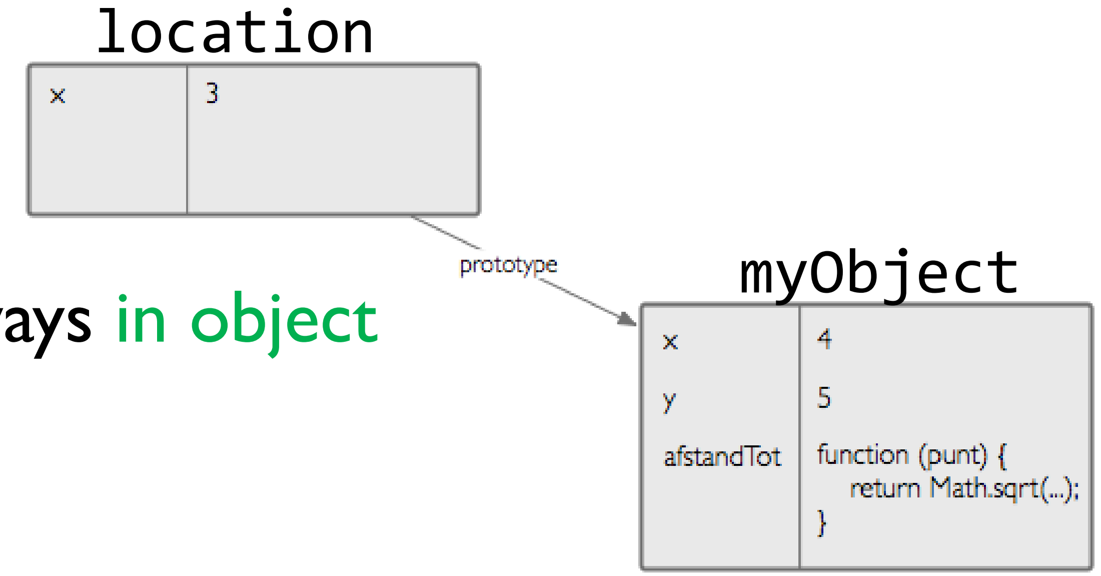
if nothing found look further in prototype chain

looking for a property



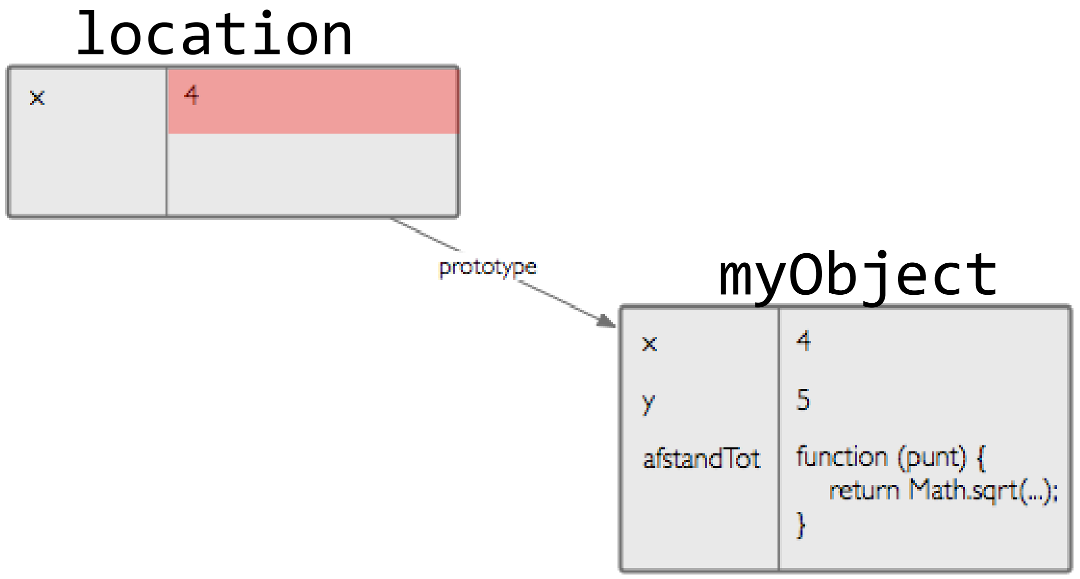
```
location.w
```

modifications



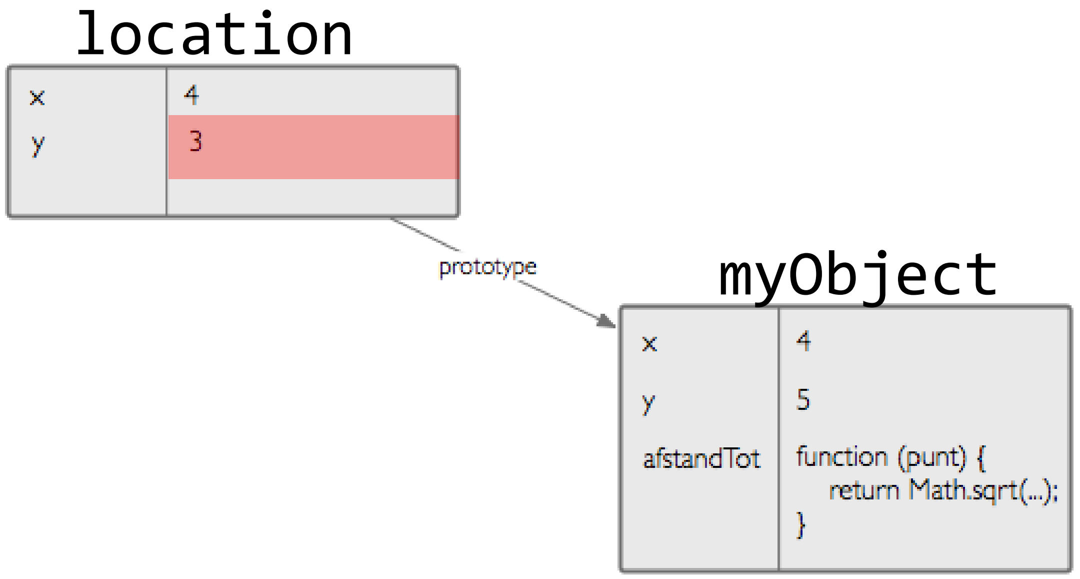
always in object

modifications



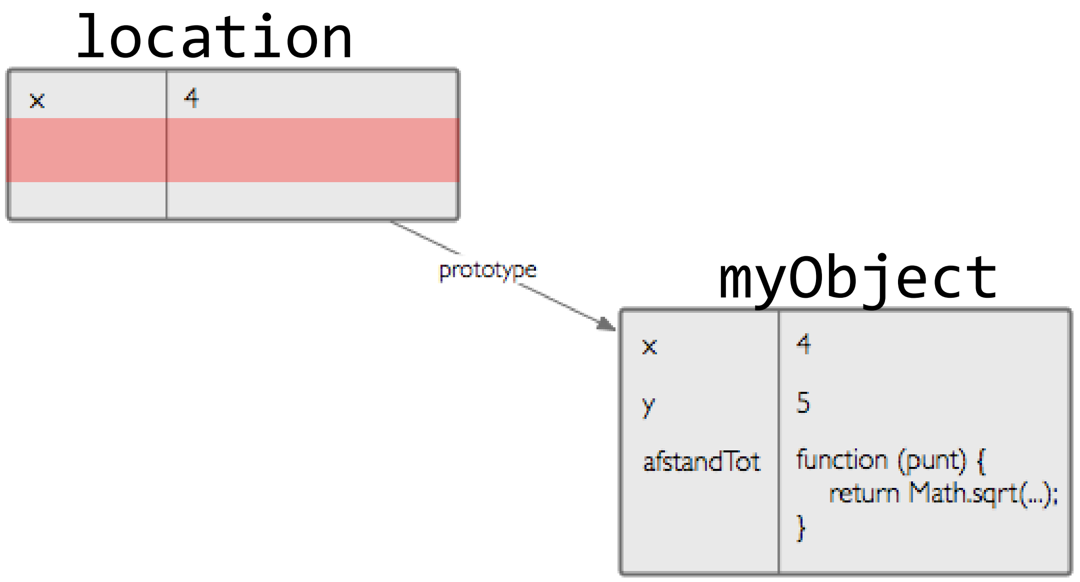
```
location.x = 4;
```

add



```
location.y = 3;
```

delete



```
delete location.y
```

How is this different from class inheritance? (like in Java)

- 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 is an 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);
```

prototype is an object – so can be changed

changing internal objects

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

Changing internal objects

advantages?

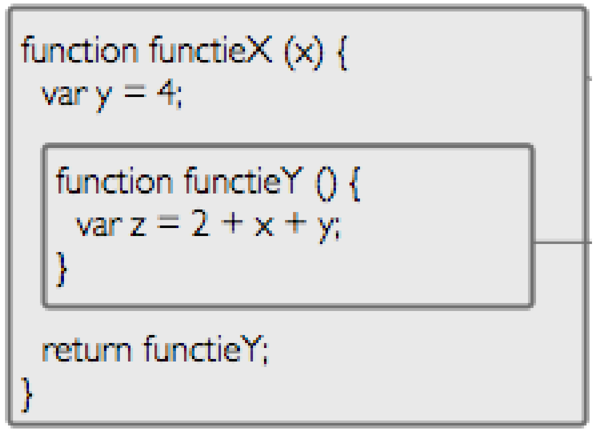
dangers?

when to use?

Lab 2

Closures

closure:
“*a function that retains the environment in which it is created*"

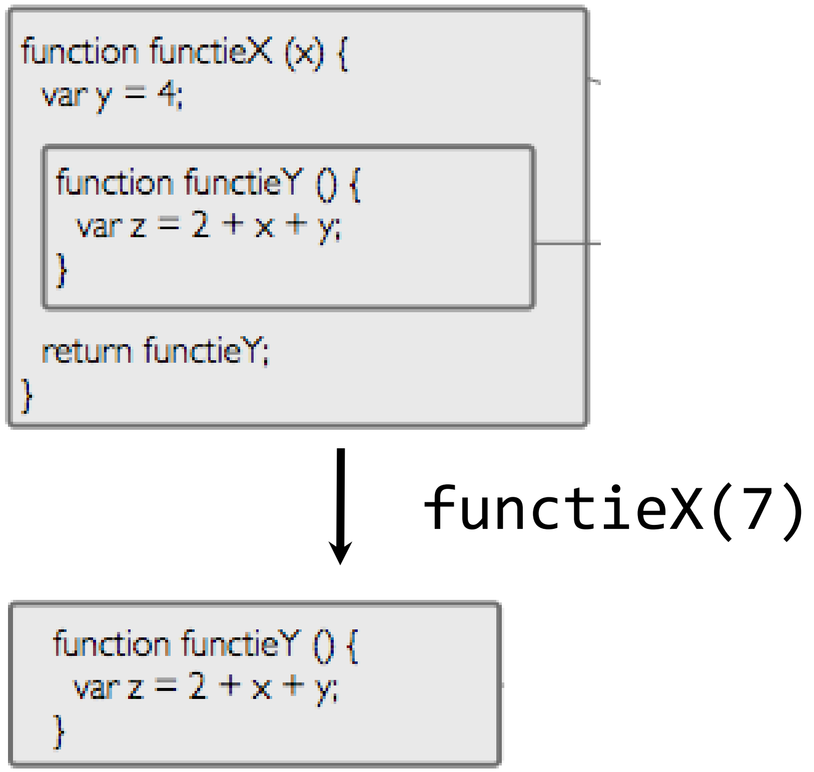


```
function functieX (x) {
  var y = 4;

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

  return functieY;
}
```

we define a function inside a function and return it

closure:

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

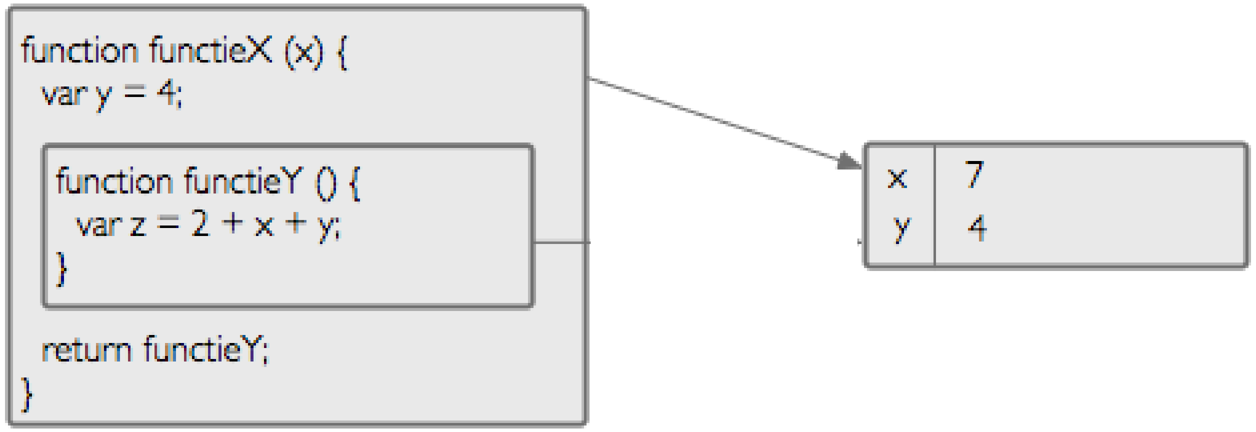


```
function functieX (x) {
  var y = 4;

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

  return functieY;
}
```

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

when `functieX` is called, `functieY` will be created

closure:
“*a function that retains the environment in which it is created*"

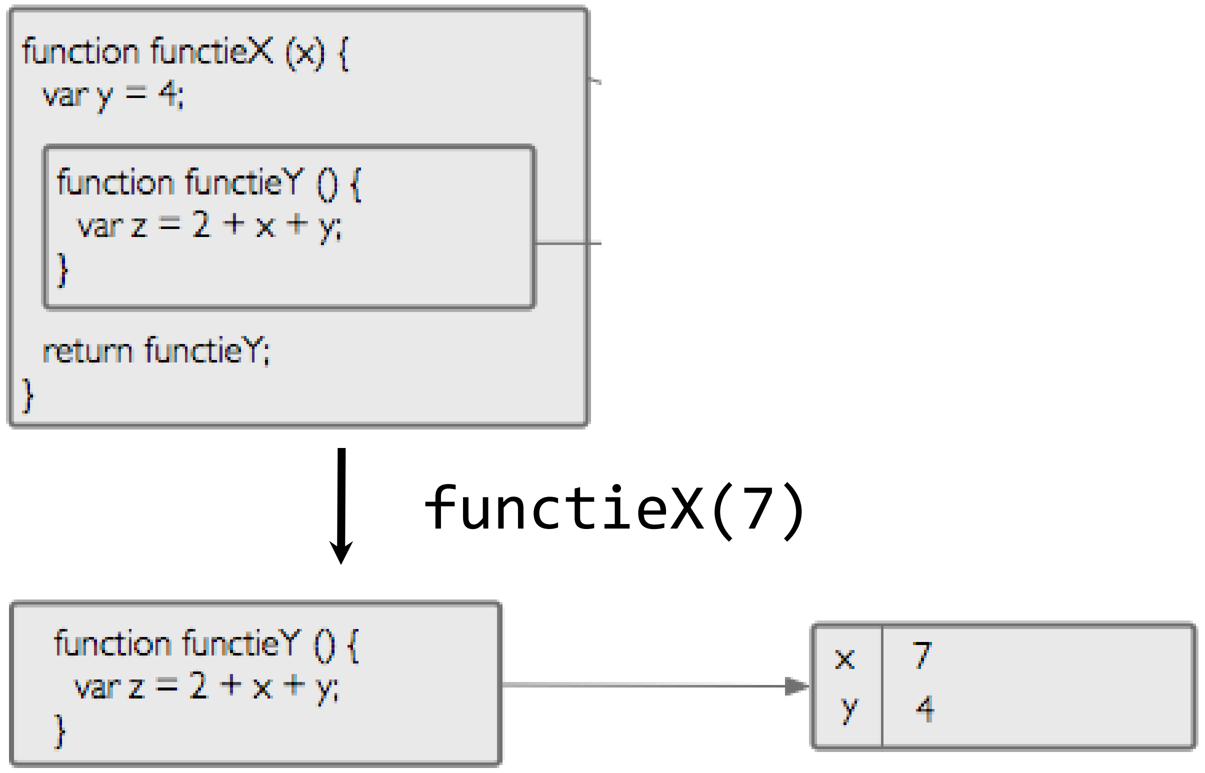


```
function functieX (x) {
  var y = 4;

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

  return functieY;
}
```

`functieX(7)`

when a function is called, an environment is created which contains local variables and parameters

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



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

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

  return functieY;
}
```

functieX(7)

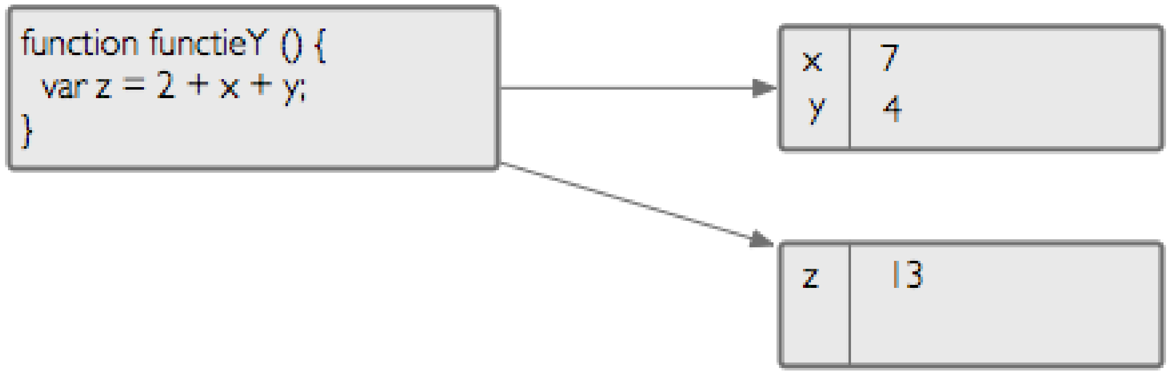
```
function functieY () {
  var z = 2 + x + y;
}
```

x	7
y	4

when `functieY` is created and returned, it retains the environment in which it is created

closure:
“*a function that retains the environment in which it is created*"

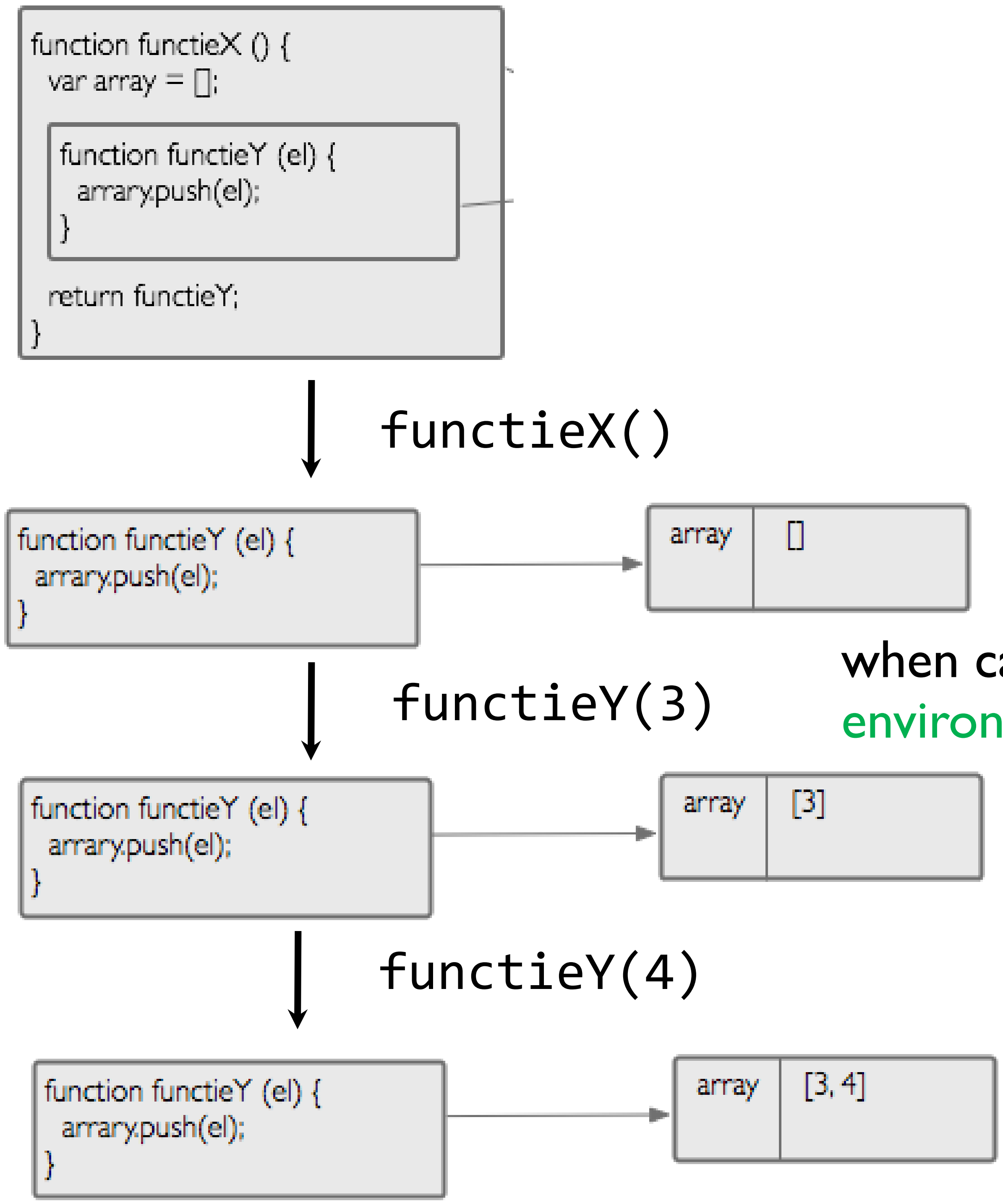
`functieY()`



```
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



```
function functieX () {
  var array = [];

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

  return functieY;
}
```

```
functieX()
```

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

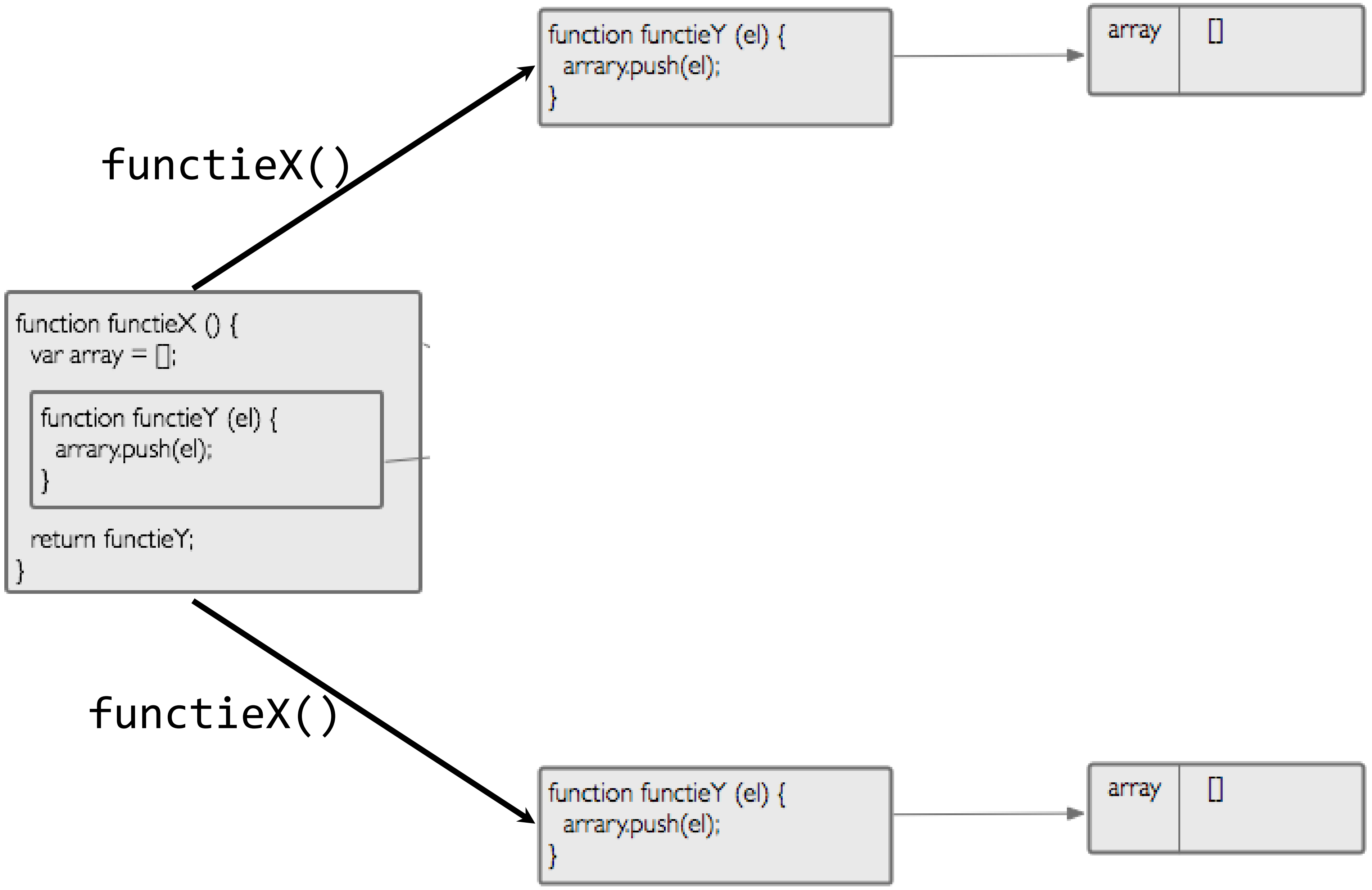
```
functieY(3)
```

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

```
functieY(4)
```

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

when calling a function, the retained environment can be changed every time you call a function, a new environment is created



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

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

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

example

```
var firstFunctionY = functieX();
firstFunction(3);
firstFunction(4); // array is [3, 4]

var secondFunctionY = functieX();
secondFunctionY(10); // array is [10]
```

Lab 3

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 scope

... it **does** have function level scope

Java – block level scope

```
public void test() {
   if (1 == 1) {
     int x = 4;
   }
   System.out.println(x); //error
}
```

JavaScript – function level scope

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

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

hoisting: local variables automatically pulled up

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

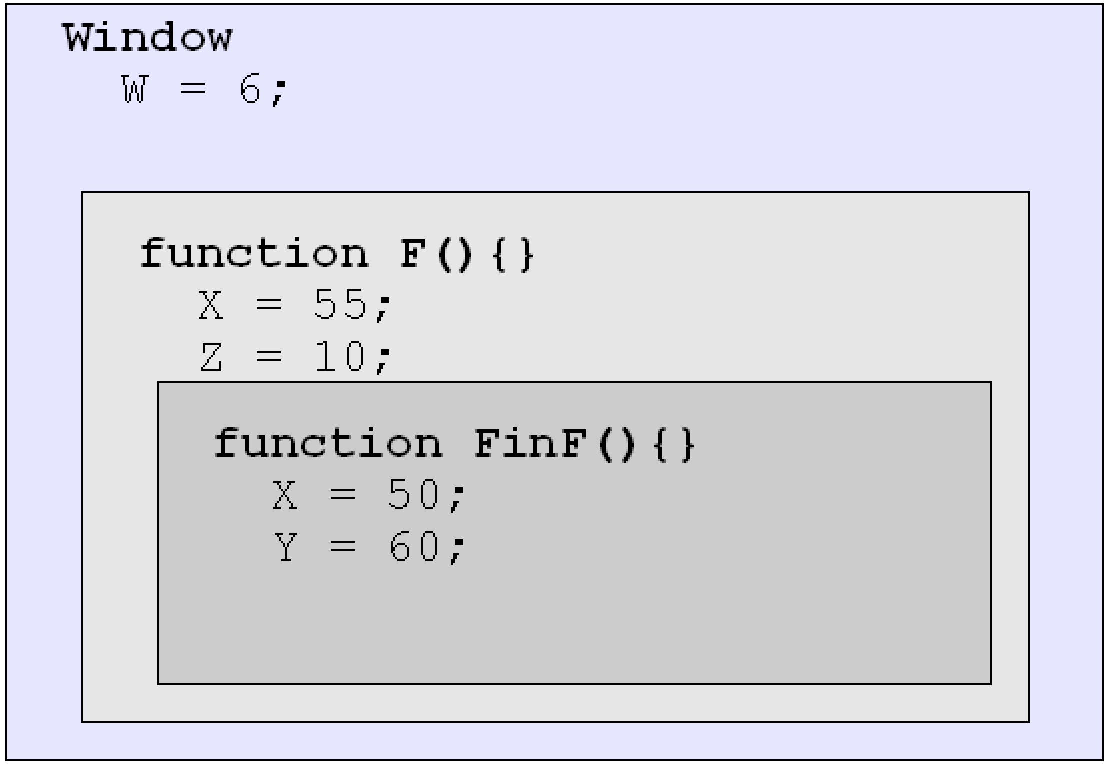
best practice: define variables at start of function:

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

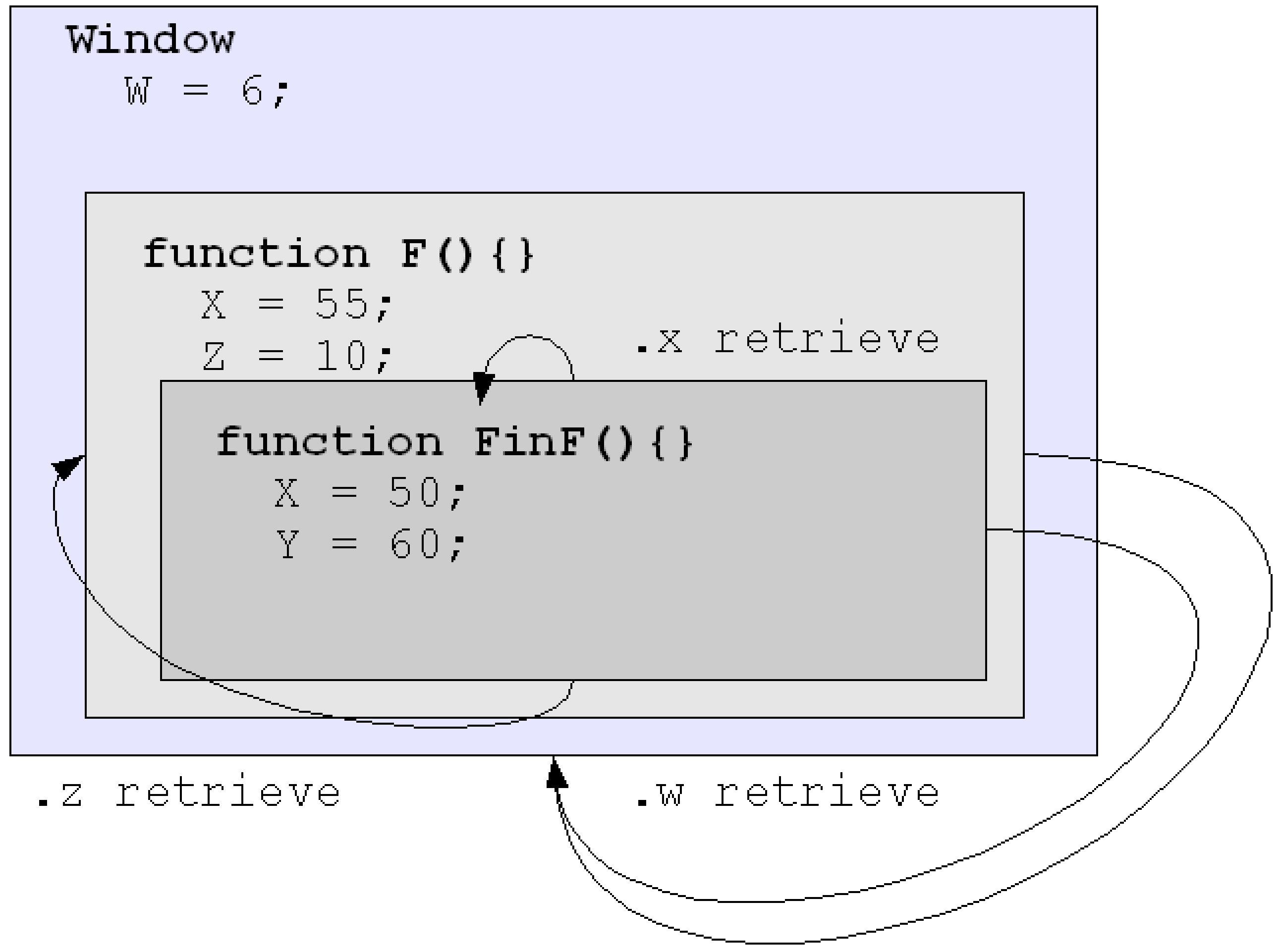
This makes it clearer what's going on.
JSLint will give a warning if you don't do this.

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;
 }
}
```

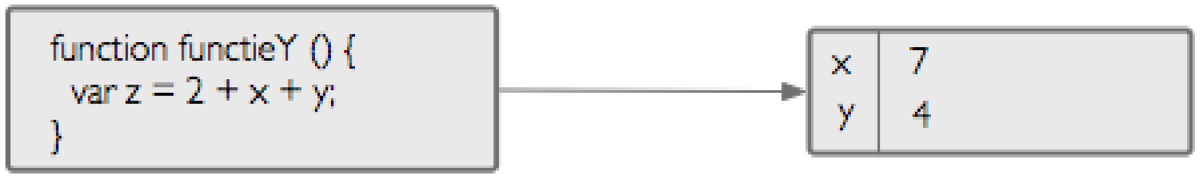


Scope chaining



Encapsulation

Remember closures?



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

x	7
y	4

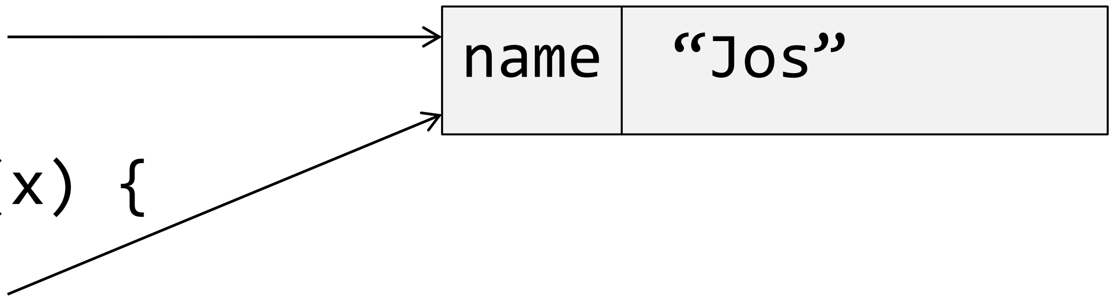
an environment is “private”

we can use this to encapsulate variables

Private variables for an object

the environment will contain the “private” fields

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



Private variables for an object

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

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

I heard you like closures
… so I put a closure in a closure

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

  function createDog() {…}

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

createDog is not exposed

Avoid creating multiple modules that are the same

```
var MOD1 = moduleCreator();
var MOD2 = moduleCreator();
```

Define and immediately evaluate a function

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

Module pattern:

```
var MOD = (function(){
  function createPerson(name) {
    return {
      getName: function() {
        return name;
      }
    };
  }
  function createDog() {}

  return {
    createPersoon: createPersoon
  };
})();
```

When do I create a **module**?

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

When should I not create a **module**?

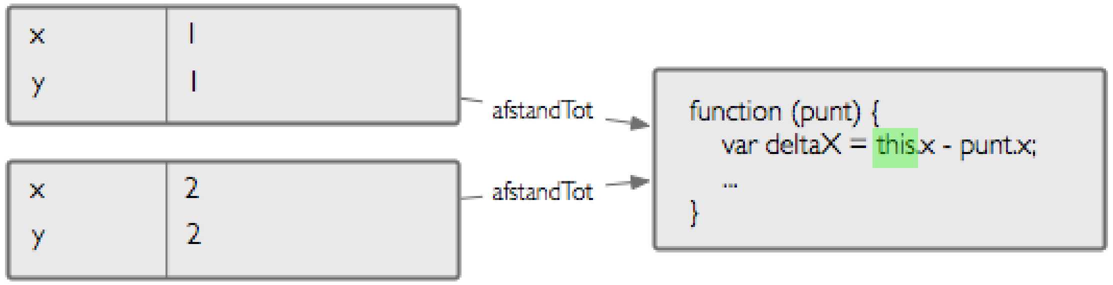
- Code that is used only once on a single page
- Not a lot of code, not a lot of pollution

Keep it simple!

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

Context in functions

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



```
function (punt) {
   var deltaX = this.x - punt.x;
   ...
}
```

what does *this* refer to?

this **is determined when calling function**

```
point1.distanceTo(point2)
```

this = point1

```
point2.distanceTo(point1)
```

this = point2

```
standaloneDistanceTo(punt1)
```

this = window (global object in JavaScript)

this **can be explicitly passed when calling a function**

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

this = punt1

using *this* in callbacks

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

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

why doesn’t it 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?

change setTimeout?

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

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

use a closure:

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

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

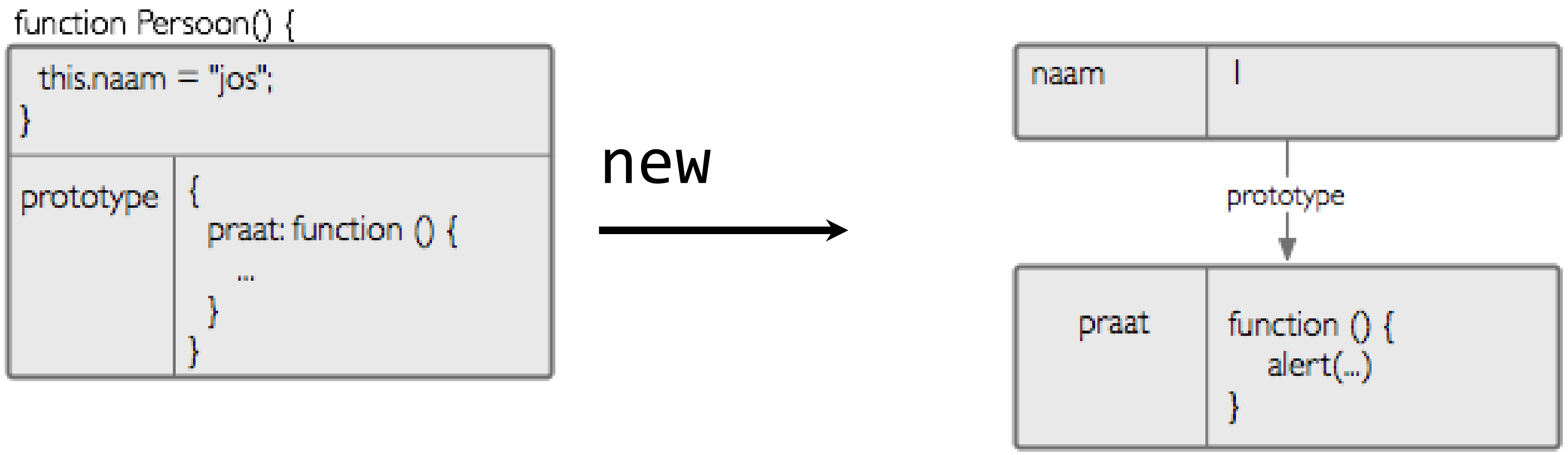
use bind method on Function:

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

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

Lab 4

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

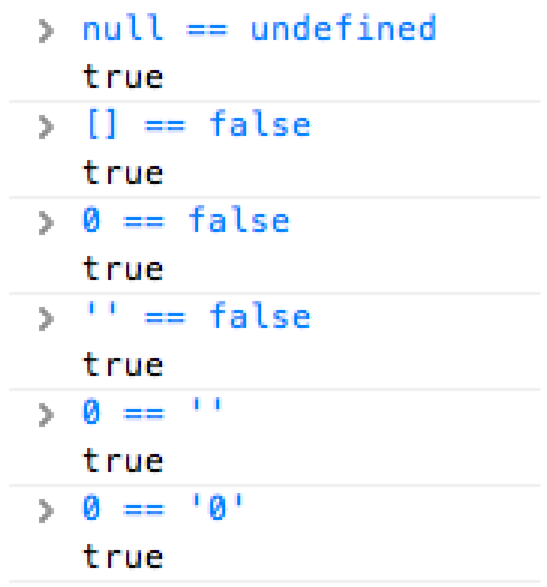
```
var jos = new Person();

jos instanceof Person === true;

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

Warning: it might look like class-based inheritance,
but it is still prototypal inheritance

That's why we prefer `Object.create` over `new`.
It makes is clear that JavaScript uses prototypal inheritance

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

== (equality operator) and != tries to cast both sides to the same type



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

confusing!

=== (identity operator) and !== doesn’t try to cast to the same type.

ECMAScript ?

ECMA

= European Computer Manufacturers Association

= standarisation

Modern browsers implement EcmaScript 5 = Javascript

For IE<9 use ES5 shim:

https://github.com/kriskowal/es5-shim

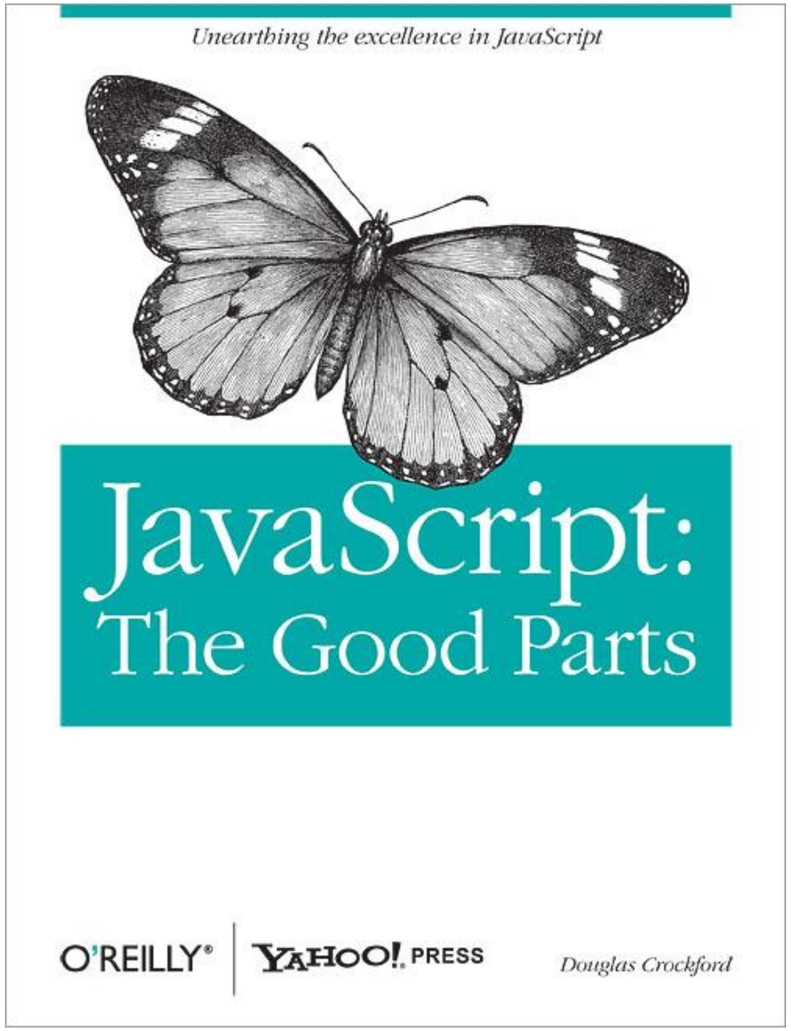
Conclusion

How JS **was** used!!

```
<script>
function oei() {
  doStuff();
  i = 10;
  copyPasta(this.i, 20);
}

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

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

room for improvement?

I want **MORE!!**


Unearthing the excellence in JavaScript
JavaScript: The Good Parts
O'REILLY®
YAHOO! PRESS
Douglas Crockford

http://www.jefklak.be/wiki/code/javascript/home