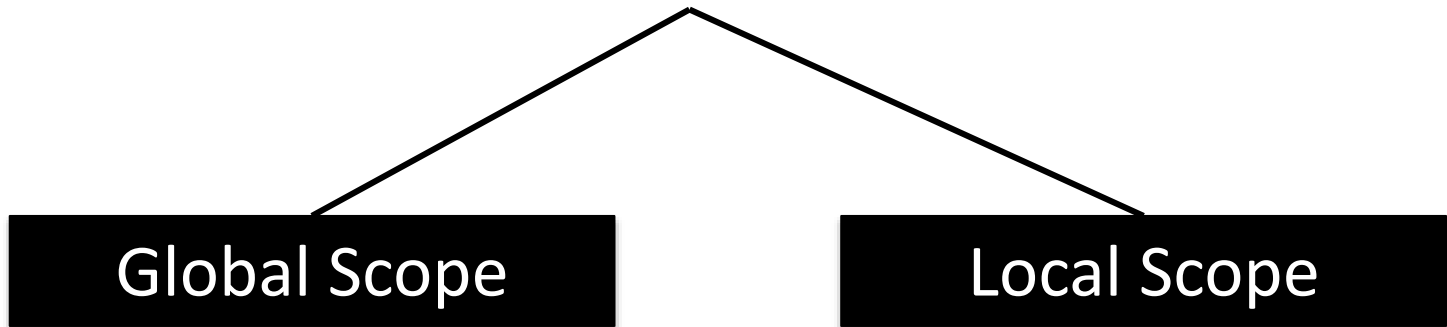


JavaScript

Variable scopes

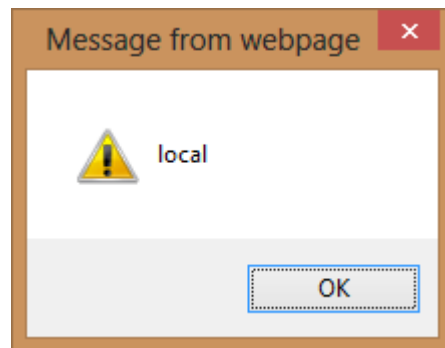
SCOPE: gedeelte van het programma waarin een variabele gedefinieerd en bruikbaar is



```
var globalScope = "global";  
function checkScope() {  
    var localScope = "local";  
}
```

In tegenstelling tot andere programmeertalen
kan de scope overgenomen worden!!

```
var scope = "global";  
function checkScope() {  
    var scope = "local";  
    return scope;  
}  
  
alert(checkScope());
```



LET OP: wat is de output van volgende code:

```
var scope = "global";  
function checkScope() {  
    localscope = "local";  
}  
  
checkScope();  
alert(localscope);
```

Functions

Functions

- Functie is een blok code dat 1 maal gedefinieerd is maar meermaals kan herbruikt worden
- In JavaScript is een functie meer dan dat
- Het goed onder de knie hebben van functies is een basis voor alles wat er nog komt
 - OO
 - Frameworks
 - ...

keyword

name

#parameters

```
function demoFunction(param1, param2)
{
    body
}
```

```
var foo = demoFunction();
```

Function always
returns something

If no retvalue is defined,
undefined will be returned

A Function can be used as statement or expression

statement

```
function demoFunction(param1, param2)
{

}
```

expression

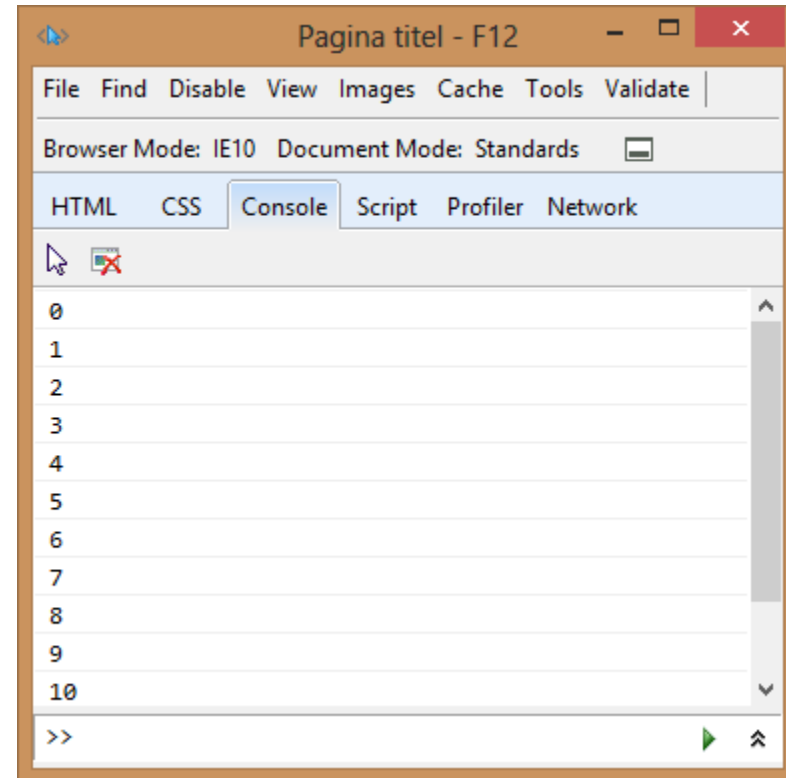
Natuurlijk niet
verplicht

```
var demo = function demoFunction(param1, param2)
{

}
```

Be carefull. What's the output??

```
function question() {  
    for (var i = 0 ; i < 10 ; i++)  
    {  
        console.log(i);  
    }  
    console.log(i);  
}
```



Hoisting

- Declaratie statements in een functie worden altijd bovenaan geplaatst
- Dit wordt uitgevoerd:

```
function question() {  
    var i;  
    for (i = 0 ; i < 10 ; i++) {  
        console.log(i);  
    }  
    console.log(i);  
}
```

Nested functions

- Functies kunnen in JavaScript ook in elkaar genest worden.
- Kan zeer interessant worden wanneer we met de scope van een variabele spelen
- De geneste functie is private en enkel beschikbaar vanuit de outerfunctie
 - Dit noemt men closures
 - Wordt zeer veel gebruikt bij JavaScript patterns

```
var c = pythagoras(12, 14);  
console.log(c);
```

340

```
console.log(power(12));  
console.log(pythagoras.power());
```

'power' is undefined

'Object' doesn't support
property of method 'power'

Closures

- De inner functie kan alleen maar opgeroepen worden door statements in de outer functie
- De inner functie kan de argumenten en parameters van de outer functie gebruiken
- De outer functie kan de argumenten en variabelen niet van de inner functie gebruiken

Behoud van variabelen

```
function outerFunction(x) {  
  function innerFunction(y) {  
    return x + y;  
  }  
  
  return innerFunction;  
}
```

X blijft behouden in de
inner functie

```
insideFunction = outerFunction(5);  
result = insideFunction(12);
```

Een closure behoudt alle referenties
die aangemaakt worden

```
console.log(result);
```

17

```
result1 = outerFunction(12)(45);
```

```
console.log(result1);
```

57

Function arguments and parameters

- Net zoals variabelen wordt er geen type bepaald bij het aanmaken van een parameter
- JavaScript checked niet of het aantal parameters die je doorgeeft welk klopt met het aantal die verwacht worden
 - Je kan wel de lengte van het aantal argumenten opvragen

```
function demo(a, b, c) {  
    if (arguments.length !== 3) {  
        throw new Error("verkeerd aantal argumenten");  
    }  
}
```

Self invoking function

Self invoking function

- Functie die zichzelf oproept
- Wordt veelal gebruikt bij het module pattern
 - Later meer daarover

```
(function selfInvoking() {  
  
}())
```

Objecten

JavaScripts meest fundamentele datatype

Definitie: Een object is een verzameling van niet geordende key/value paren

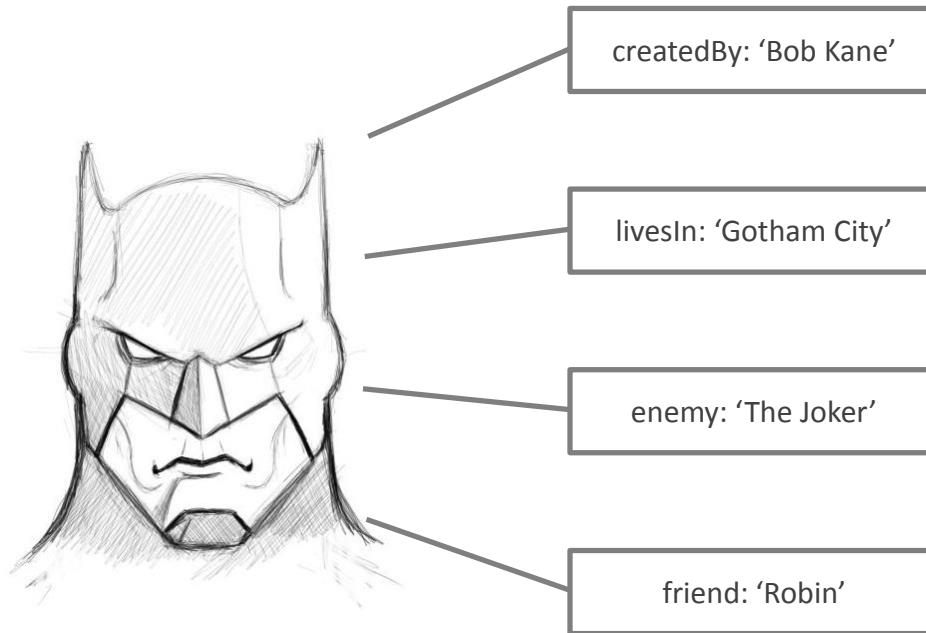
Begin met een
{

Key als Property
Type: 'string'

Value
Kan elke JS Type zijn

property: 'value';

Eindig met een
}




<http://www.drawing-factory.com/image-files/how-to-draw-batman-11.jpg>

Property

Value

```
var batman = {  
  createdBy: {  
    firstname: 'Bob',  
    lastname: 'Kane',  
    fullname: function () {  
      return this.firstname + ' ' + this.lastname  
    }  
  },  
  createdIn: 1939,  
  livesIn: 'Gotham City',  
  enemy: 'The Joker',  
  friends: ['Robin', 'Catwoman']  
};
```

A diagram with two black boxes at the top labeled 'Property' and 'Value'. A line from the 'Property' box points to the 'createdBy' property in the JavaScript code. A line from the 'Value' box points to the 'firstname' value ('Bob') within the 'createdBy' object.

Objecten aanmaken

Er zijn verschillende manieren om een object aan te maken

Via Object Literals

Via een zelfgemaakte constructor functie die je dan oproept aan
adhv het new keyword

Via `Object.create()`;

Object Literals

```
var emptyObject = {};  
var batman = {  
  createdBy: {  
    firstname: 'Bob',  
    lastname: 'Kane',  
    fullname: function () {  
      return this.firstname + ' ' + this.lastname  
    }  
  },  
  createdIn: 1939,  
  livesIn: 'Gotham City',  
  enemy: 'The Joker',  
  friends: ['Robin', 'Catwoman']  
};
```

Via new keyword

- New maak en initialiseert een nieuw object.
- New moet gevolgd worden door een functie
 - Deze wordt de constructor functie genoemd

```
var o = new Object(); // Maakt een nieuw object aan: hetzelfde als {}.  
var a = new Array(); // Maakt een lege array aan: zelfde als [].  
var d = new Date(); // Maakt een date object aan
```

Object.create()

- Vanaf ES5
- Static function

```
var JamesBond = {  
  orderCocktail: function () {  
    console.log("Martini, shaken not stirred");  
  }  
};
```

Definieer het prototype.
inheritance

```
var sean = Object.create(JamesBond,  
  { 'id': '007' },  
  { 'car': 'Aston Martin' }  
)
```

Definieer extra eigenschappen

Eigenschappen of Properties

- Object is een mutable datatype
 - Er kunnen properties toegevoegd of verwijderd worden
 - Maakt een object heel dynamisch
 - Veel frameworks en libraries zijn hier op gebaseerd
- Een Object bezit zijn eigen properties, maar ook deze van het bovenliggend type: prototype

Eigenschappen oproepen of invullen

Via dot notatie of via indexer

```
// eigenschappen oproepen  
console.log(batman.livesIn);  
console.log(batman["livesIn"]);
```

```
// eigenschappen invullen  
batman.livesIn = "Gent";  
batman["livesIn"] = "Gent";
```


Eigenschappen overlopen van een object

```
var s = '';  
for (var prop in batman) {  
    s += prop + ': ' + batman[prop] + '\n';  
}
```

Bestaat een bepaalde eigenschap

```
if ("enemy" in batman) {  
    alert("find the damn bastard");  
}
```

OF

```
batman.hasOwnProperty("enemy");
```

Getters en Setters

- Tot nu toe spraken we over data properties
 - Je kan er iets insteken en iets uit halen
 - Je kan wel niet controleren voor je een van de 2 uitvoert
- Oplossing: getter en setter function

```
var batman = {  
  getCreatedIn: function () { return this.createdIn; },  
  setCreatedIn: function (value) {  
    if (isNaN(value) || value > new Date().getFullYear()) {  
      throw "This is not a valid date";  
    }  
    this.createdIn = value;  
  }  
};
```

Getters en Setters sinds ES5

```
get CreatedIn() { return this.createdIn; },
set CreatedIn(value) {
    if (isNaN(value) || value > new Date().getFullYear())
    {
        throw "This is not a valid date";
    }
    this.createdIn = value;
}
```

Eigenschappen toevoegen

- Je kan op een eenvoudige manier eigenschappen toevoegen door de .notatie of via de indexer

```
batman.isInjured = true;
```

```
/* OF */
```

```
batman["isInjured"] = true;
```

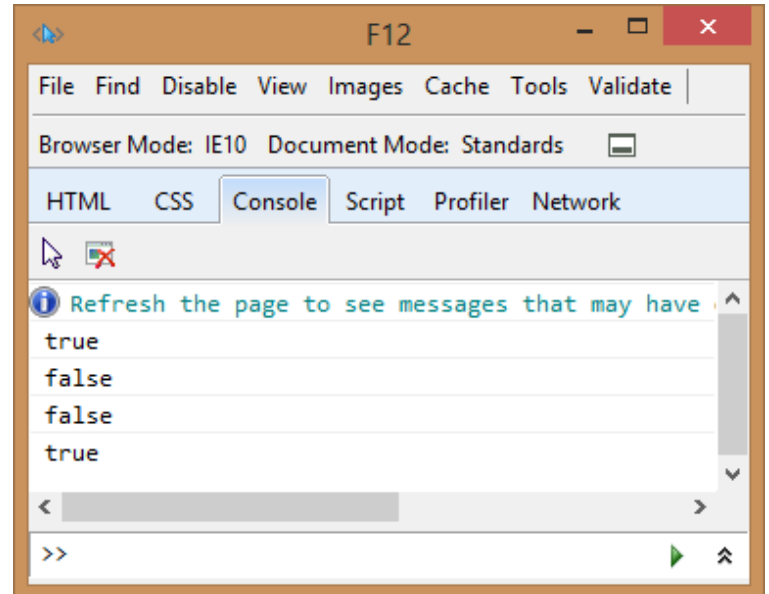
```
if ("isInjured" in batman) { // --> TRUE  
    alert("Heal batman");  
}
```

Eigenschappen toevoegen die defineProperty

- defineProperty vanaf ES5
- Laat u toe om mee te geven hoe je de eigenschap wil behandelen eens aangemaakt
 - **Configurable**: kan je het object opnieuw instellen
 - **Enumerable**: kan je een for in gebruiken
 - **Value**: de waarde
 - **Writable**: kan de waarde overschreven worden

Noemt men de
propertydescriptors

```
var descr =  
Object.getOwnPropertyDescriptor(  
    batman,  
    "isInjured");  
  
console.log(descr.writable);  
console.log(descr.enumerable);  
console.log(descr.configurable);  
console.log(descr.value);
```



Eigenschappen verwijderen

- Het delete statement verwijdert een property uit een object, maar niet de waarde in het geheugen

```
delete batman.isInjured;
```


Classes

Easy topic: JavaScript ondersteunt geen classes

An orange callout box with a black outline, featuring a tab-like shape on its top-left side. It contains the text 'Wel vanaf ES6' in white.

Wel vanaf ES6

Arrays

Array

- Is een Object
- Geordende verzameling van values
 - Elke value wordt ook wel een *element* genoemd
 - Elk element staat op een bepaalde index
- JavaScript arrays zijn untyped
 - Elk element kan dus van een ander type zijn
- Zero-based

Array aanmaken

```
var emptyArray = [];  
var priemGetallen = [2, 3, 7, 11];  
var undefineds = [1, , 6];
```



Undefined

Hoeveel elementen zitten er in volgende array



```
var question = [ , , ];  
console.log(question.length);
```

Trailing comma

2



New Array();




- Een andere manier om een array te maken is via new Array();

```
var a = new Array(5, 4, 3, 2, 1, "blabla, blabla");
```


Patterns

Design Pattern: Herbruikbare oplossing om een typisch probleem op te lossen.

Gekende design patterns

- Constructor pattern 
- Module pattern 
- Singleton pattern 
- Observer pattern
- Mediator pattern
- Flyweight pattern
- Prototype pattern
- Command pattern
- Facade pattern
- Factory pattern
- Mixin
- Decorator pattern

Constructor Pattern

Constructor pattern

Probleem

- In JavaScript is alles een object
- Er zijn ook verschillende manieren om een object te maken

```
var newObject = {};  
// or  
var newObject = Object.create(null);  
// or  
var newObject = new Object();
```

Constructor pattern

Oplossing

- Constructor functie
- Functie die opgeroepen wordt wanneer je een object aanmaakt
- *this* verwijst naar het object dat aangemaakt is

```
function JamesBond(name, firstname, age) {  
    this.name = name;  
    this.firstName = firstname;  
    this.age = age;  
  
    this.toString = function () {  
        return this.name + ", " + this.firstName + " " + this.name;  
    }  
}  
  
var sean = new JamesBond("Connery", "Sean", 70);
```

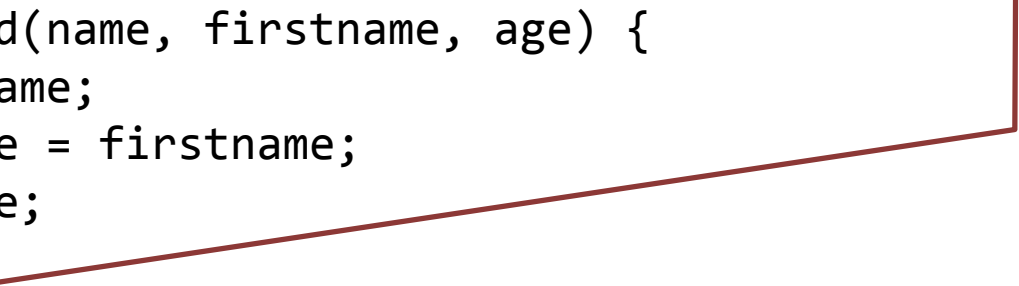
Deze constructor functie heeft een aantal minpunten



Alles we inheritance willen gebruiken dan is dit niet de ideale manier

toString wordt elke keer opnieuw aangemaakt

```
function JamesBond(name, firstname, age) {  
  this.name = name;  
  this.firstName = firstname;  
  this.age = age;  
  
  this.toString = function () {  
    return this.name + ", " + this.firstName + " " + this.name;  
  }  
}  
  
var sean = new JamesBond("Connery", "Sean", 70);
```



Prototype!!

Prototype

- Elk object heeft een 2^e object geassocieerd genaamd het prototype
- Het eerste object erft alle eigenschappen van het prototype object
 - Overerving
 - Elke object gemaakt met `new Object()` erft van `Object.prototype`
 - Elke object gemaakt met `new Array()` erft van `Array.prototype`

Constructor Pattern

Oplossing

```
function JamesBond(name, firstname, age) {  
    this.name = name;  
    this.firstName = firstname;  
    this.age = age;  
}  
  
JamesBond.prototype.toString = function () {  
    return this.name + ", " + this.firstName + " " + this.name;  
}
```

Module Pattern

Module pattern

- Modules zijn stukken code die ervoor zorgen dat je project overzichtelijk blijven
- Is vooral gebaseerd op objecten
 - Ongeordend lijst van key/value pairs

```
var comics = {  
  numberOfComics: 34,  
  listComics: function () {  
    // ...  
  }  
};
```

Module Pattern

Probleem

- JavaScript heeft geen private of public accessors
 - Alles is publiek
- Het Module pattern probeert dit te simuleren
 - Met closures die we eerder bespraken

```
var testModule = (function () {  
    var counter = 0;  
  
    return {  
        incrementCounter: function () { return counter++; },  
        resetCounter: function () {  
            alert("resetting " + counter);  
            counter = 0;  
        }  
    }  
})();
```

testModule.

- constructor constructor() (+ 1 overload(s))
- hasOwnProperty
- incrementCounter
- isPrototypeOf
- propertyIsEnumerable
- resetCounter
- toLocaleString
- toString
- valueOf

Mooi voorbeeld van het module pattern is
het gebruik van namespaces

```
var MI6 = (function () {  
  
    var privateInfo;  
  
    privateMethod= function(info){  
        console.log(info);  
    };  
  
    return {  
        // public  
        publicName: "MI6",  
        publicMethod: function (info) { }  
    };  
  
})();
```

MI6.

- ⊗ constructor
- ⊗ hasOwnProperty
- ⊗ isPrototypeOf
- ⊗ propertyIsEnumerable
- ⊗ **publicMethod**
- 🔗 publicName
- ⊗ toLocaleString
- ⊗ toString
- ⊗ valueOf

publicMethod(info)

Be careful with memory leaks

Garbage collection: Gebruikt om het geheugen 'op te kuisen' wanneer een variabele niet meer gebruikt wordt.

Lifecycle object

- Wanneer een object gemaakt wordt dan wordt de benodigde plaats in het geheugen aangemaakt
- Van nu af aan is het de GC die bepaalt wanneer deze variabele afgebroken wordt.
 - Kijkt of een object nog reachable is
- 2 mogelijke manieren
 - Reference counting
 - Mark-and-sweep

Reference counting

- Voor elk aangemaakt object wordt bijgehouden hoeveel referenties er naar dat object bestaan
- Wanneer er geen enkele referentie naar dat object bestaat wordt deze afgebroken

Reference counting

```
var person = { name: 'kevin', lastname: 'derudder' };  
// #references naar kevin: 1
```

```
var person2 = person;  
// #references naar kevin: 2
```

```
var person = { name: 'johan', lastname: 'vannieuwenhuyse' };  
// #references naar kevin: 1
```

```
person2 = null;  
// #references naar kevin: 0 --> GC kan zijn werk doen
```

Gevaar

Wat met volgende gevallen??

```
function foo() {  
  var person = { name: 'kevin'};  
  var person2 = { name: 'johan' };  
  
  person.collega = person2; // person refereert naar person2  
  person2.collega = person; // person2 refereert naar person  
  
  return "tsjaka";  
}  
  
foo();
```

Circular reference

Dit komt redelijk veel voor!

```
var div = document.createElement('div');  
  
div.onclick = function () {  
    handleClick();  
};
```

Mark-and-sweep

- Bestaat uit 2 fases
 - Mark-fase
 - Sweep-fase
- Mark
 - De GC overloopt alle variabelen en houdt de waarden bij
 - De GC overloopt alle variabelen en vergelijkt de referenties met de vorige waarden
- Sweep
 - Variabelen naarwaar geen referentie bestaat wordt afgebroken

Best Practices

- Gebruik enkel variabelen die echt nodig zijn
 - Variabelen die toegewezen zijn aan de Global Scope worden niet afgebroken
- Vermijd circular references
- Vermijd het gebruik van objecten in timers
- Vermijd het doorgeven van objecten aan `console.log()`
 - De console heeft de info nodig, zal daarom het object in leven houden

Best practices

- Vermijd het delete keyword
 - Delete haalt een property uit een object maar het object blijft wel bestaan

```
var person = {  
  name: 'kevin',  
  lastname: 'derudder'  
};
```

```
console.log(person.lastname); // derudder  
delete person.lastname;  
console.log(person.lastname); // undefined
```