

**HO
GENT**



Webapplicaties II

Hoofdstuk 03 – OOP in Javascript

Inhoud

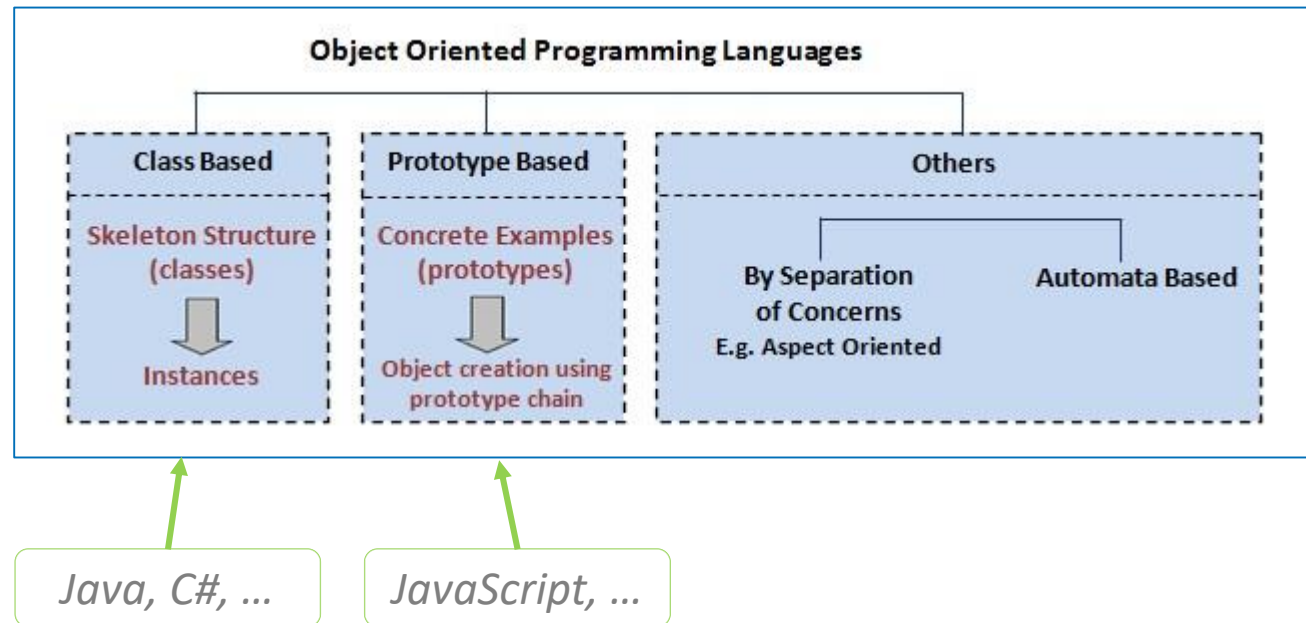
- Object oriented programming in JavaScript
 - Object orientation in JavaScript
 - Class
 - constructor method & new keyword
 - get/set methods
 - other methods
 - static methods
 - inheritance
 - Prototypes
 - concept
 - constructor function & new keyword
 - prototype chain
 - JavaScript's built-in objects

03 OOP in Javascript

Inleiding

OO & JavaScript

JavaScript is an **object-based language** based on **prototypes**, rather than being class-based.



OO & JavaScript

- JavaScript has a **prototype-based, object-oriented programming model**.
 - it creates objects using other objects as blueprints and to implement inheritance it manipulates what's called a prototype chain.
- Although the prototype pattern is a valid way to implement object orientation it can be **confusing for newer JavaScript developers or developers used to the classical pattern**.
- So **in ES6 we have an alternative syntax**, one that closer matches the classical object orientated pattern as is seen in other languages.



Under the hood the new syntax still uses the prototype pattern with constructor functions and the prototype-chain. However, it provides a more common and convenient syntax with less boilerplate code.



**HO
GENT**

03 OOP in Javascript

Class declarations

Class declaration

The **class declaration** creates a new class with a given name using **prototype-based inheritance**.



```
class ClassName {  
  
    constructor (...) {...}  
  
    method1(...) {...}  
  
    get someProp() {...}  
    set someProp(value) {...}  
  
    static staticMethod(...) {...}  
}
```


Class declaration

- Voorbeeld: Blog

een Blog heeft een 'creator'
en een lijst van 'blog entries'

Ruby's blog



08/14/2008
Got the new cube I ordered.
It's a real pearl

08/15/2008
Solved the new cube but of course now
I'm bored and shopping for a new one

08/16/2008
Managed to get a headache toiling over the new
cube. Gotta nap.

08/21/2008
Found a 7x7x7 cube for sale online. Yikes!

een BlogEntry heeft
een 'date' en
een 'body'

Class declaration

- Voorbeeld: BlogEntry – een klasse voor een blog entry

Ruby's blog

08/14/2008
Got the new cube I ordered.
It's a real pearl

08/15/2008
Solved the new cube but of course now
I'm bored and shopping for a new one

08/16/2008
Managed to get a headache toiling over the new
cube. Gotta nap.

08/21/2008
Found a 7x7x7 cube for sale online. Yikes!

```
class BlogEntry {  
  constructor(body, date) {  
    this.body = body;  
    this.date = date;  
  }  
}
```

een basis klasse declaratie voor BlogEntry

03 OOP in Javascript

Class declarations

new & constructors

Class declaration

- **new** – instanties maken van een klasse

```
class BlogEntry {  
  constructor(body, date) {  
    this.body = body;  
    this.date = date;  
  }  
}
```

```
const aBlogEntry = new BlogEntry(  
  'Managed to get a headache toiling over the new cube. Gotta nap.',  
  new Date(2008, 7, 16)  
);
```

er wordt een nieuwe instantie van BlogEntry gemaakt, deze bevat o.a. een date object dat hier ook wordt geïnstantieerd...

```
console.log(`aBlogEntry is of type ${typeof aBlogEntry}`);  
console.log(aBlogEntry);
```

aBlogEntry is of type object

► BlogEntry {body: "Managed to get a headache toiling over the new cube. Gotta nap.", date: Sat Aug 16 2008 00:00:00 GMT+0200 (Romance (zomertijd))}

Class declaration

- de **constructor**
 - gebruik keyword **constructor** met eventuele parameters
 - gebruik **this** in de constructor body om **properties** te definiëren en eventueel te initialiseren
- **default constructor**
 - indien in een klasse niet expliciet een constructor wordt gedefinieerd dan heeft de klasse impliciet een parameterloze constructor

```
class NoCtorClass {}  
const myObject = new NoCtorClass();
```

- er is **geen constructor overloading** mogelijk!
 - je mag hoogstens 1 constructor definiëren

Class declaration

- de properties gedefinieerd in de constructor zijn **publiek toegankelijk**

```
aBlogEntry.body = 'Nothing to say about cubes today!';  
aBlogEntry.date = new Date();  
console.log(`body: ${aBlogEntry.body}`);  
console.log(`date: ${aBlogEntry.date}`);
```

```
body: Nothing to say about cubes today!  
date: Thu Feb 15 2018 22:31:06 GMT+0100 (Romance (standaardtijd))
```

- er zijn mogelijkheden om effectief private properties te maken maar deze vallen buiten de scope van deze cursus...
 - zie bv. http://exploringjs.com/es6/ch_classes.html#sec_private-data-for-classes

Class declaration

- **try it yourself**
 - zie blog1.js in 03thOopInJavaScript

03 OOP in Javascript

Class declarations

properties: getters & setters

Class declaration

- **conventie** i.v.m. private properties: **_**
 - de naam van een property die niet publiek zou mogen gebruikt worden laten we voorafgaan door een **underscore**
 - **leg voldoende discipline aan de dag om dergelijke properties niet in client code te gebruiken!**

```
class BlogEntry {  
  constructor(body, date) {  
    this._body = body;  
    this._date = date;  
  }  
}
```

```
const aBlogEntry = new BlogEntry('...', new Date());  
aBlogEntry._body = 'I can but I should not be doing this :(';
```



- voorzie in de klasse nu een **publieke interface**: publieke methodes, getters en setters die gebruik maken van de private properties

Class declaration

- **getter**

The **get** syntax binds an object property to a function that will be called when that property is looked up.

- gebruik keyword **get** gevolgd door de naam van de property
- **geen parameters**
- merk op: de get syntax associeert een functie met een property maar je roept die functie niet expliciet op
 - de get-functie wordt uitgevoerd als de waarde van de property gelezen moet worden

```
class BlogEntry {  
  constructor(body) {  
    this._body = body;  
    this._date = new Date();  
  }  
  
  get body() { return this._body; }  
  get date() { return this._date; }  
}
```

```
const aBlogEntry = new BlogEntry('...');  
console.log(`body: ${aBlogEntry.body}`);  
console.log(`date: ${aBlogEntry.date}`);
```

de property body wordt opgevraagd -> de get-functie gekoppeld aan body wordt uitgevoerd

Class declaration

- **setters**

The **set** syntax binds an object property to a function to be called when there is an attempt to set that property.

- gebruik keyword **set** gevolgd door de naam van de property
- **exact 1 parameter**: de waarde die we wensen toe te kennen aan de property
- merk op: de set-functie wordt uitgevoerd wanneer we proberen een waarde toe te kennen aan de property
 - ook hier dus geen expliciet aanroep van de set-functie

```
class BlogEntry {  
  constructor(body) {  
    this.body = body;  
    this._date = new Date();  
  }  
  
  get body() { return this._body; }  
  set body(value) { this._body = value; }  
  get date() { return this._date; }  
}
```

```
const aBlogEntry = new BlogEntry('...');  
aBlogEntry.body = 'This is the way to go :)';  
console.log(`body: ${aBlogEntry.body}`);  
console.log(`date: ${aBlogEntry.date}`);
```

Wanneer we een waarde willen toekennen aan de body-property wordt de set-functie, gekoppeld aan body, uitgevoerd; de parameter value krijgt de waarde die we willen toekennen aan de property

Class declaration

- voorbeeld: class BlogEntry samengevat

```
class BlogEntry {  
  constructor(body) {  
    this.body = body;  
    this._date = new Date();  
  }  
  
  get body() { return this._body; }  
  set body(value) { this._body = value; }  
  get date() { return this._date; }  
}
```

*_body en _date zijn private en zullen we enkel binnen de klasse gebruiken, niet in de client code.
merk op: in de constructor gebruiken we this.body: de setter voor body wordt aangeroepen!*

body en date zijn publiek en gebruiken we in de client code, date heeft geen setter



```
const aBlogEntry = new BlogEntry('...');  
aBlogEntry.body = 'This is the way to go :);'  
console.log(`body: ${aBlogEntry.body}`);  
console.log(`date: ${aBlogEntry.date}`);
```

de setter wordt uitgevoerd

de getter wordt uitgevoerd

de getter wordt uitgevoerd

```
body: This is the way to go :)  
date: Thu Feb 15 2018 23:05:03 GMT+0100 (Romance (standaardtijd))
```

Class declaration

```
3
4 class BlogEntry {
5   constructor(body) {
6     this.body = body;
7     this._date = new Date();
8   }
9
10  get body() {
11    return this._body;
12  }
13  set body(value) { value = "This is the way to go :)"
14    this._body = value;
15  }
16  get date() {
17    return this._date;
18  }
19 }
20
21 const aBlogEntry = new BlogEntry('...');
22 aBlogEntry.body = 'This is the way to go :>';
23 console.log(`body: ${aBlogEntry.body}`);
24 console.log(`date: ${aBlogEntry.date}`);
```

Paused on breakpoint

- Watch
- Call Stack
 - set body blog.js:14
 - (anonymous) blog.js:22
- Scope
 - Local
 - this: BlogEntry
 - value: "This is the way to go :)"
 - Script
 - Global Window
- Breakpoints
 - blog.js:11 return this._body;

als je breakpoints plaatst in de get/set methodes kan je goed volgen wat er gebeurt...

Class declaration

- **getter vs. parameterloze methode**

- de get-syntax laat toe om aan een berekende waarde van een object te geraken zonder een methode aan te roepen

twee manieren om hetzelfde te bewerkstelligen, let op de verschillen in de klasse definitie, en op de verschillen in gebruik...

```
class BlogEntry {  
  constructor(body) {  
    this.body = body;  
    this._date = new Date();  
  }  
  get nrOfWords() {  
    return this._body.split(' ').length;  
  }  
  getNrOfWords() {  
    return this._body.split(' ').length;  
  }  
  // rest omitted for brevity  
}
```

```
const myBlogEntry = new BlogEntry('Een twee drie vier vijf');  
console.log(myBlogEntry.nrOfWords);  
console.log(myBlogEntry.getNrOfWords());
```

5

5

Class declaration

- enkele opmerkingen
 - ook in **object literals** (cf. H01) kan je werken met get/set methodes

```
const myAvatar = {  
  _name: 'Bob',  
  get name() {return _name;},  
  set name(value) {_name = value;}  
};  
myAvatar.name = 'Ann';
```

- in een class kan je een property niet definiëren als key/value pair zoals in object literals, enkel methodes, getters en setters...
- het werken met get/set methodes biedt tal van **voordelen**
 - encapsulatie van het gedrag die hoort bij manipulatie van een property
 - bv. validatie
 - verbergen van de interne representatie van een property
 - de representatie naar buiten toe kan anders zijn
 - zorgen dat de interface van je klasse geïsoleerd is tegen veranderingen
 - de implementatie van de klasse kan wijzigen zonder dat de client code aangepast hoeft te worden
 - je kan debuggen op veranderingen in de property

03 OOP in Javascript

Class declarations

methodes

Class declaration

- **methodes**
 - zijn **functie-properties**
 - naam, paramaters, return waarde zie H02: Objects and functions

Class declaration

- voorbeeld: class BlogEntry uitgebreid met **methode**

```
class BlogEntry {  
  constructor(body) {  
    this.body = body;  
    this._date = new Date();  
  }  
  get body() { return this._body; }  
  set body(value) { this._body = value; }  
  get date() { return this._date; }  
  contains(searchText) {  
    return searchText  
      ? this._body.toUpperCase().includes(searchText.toUpperCase())  
      : false;  
  }  
}
```

```
let searchText = '';  
do {  
  searchText = prompt('Enter search term (leave blank to stop searching)');  
  if (searchText)  
    alert(  
      `aBlogEntry ${  
        aBlogEntry.contains(searchText) ? 'contains' : 'does not contain'  
      } ${searchText}.`  
    );  
} while (searchText);
```

Class declaration

- **static methods**

- worden aangeroepen zonder de klasse te instantiëren
 - kunnen **niet** worden aangeroepen via instanties van de klasse
- voorbeeld

*declaratie: gebruik
keyword **static***

```
class BlogEntry {  
  constructor(body) {  
    this.body = body;  
    this._date = new Date();  
  }  
  static createDummy() {  
    return new this('Nothing much to say today...');  
  }  
  // rest of class omitted for brevity...  
}
```

*aanroep: gebruik
de naam van de*

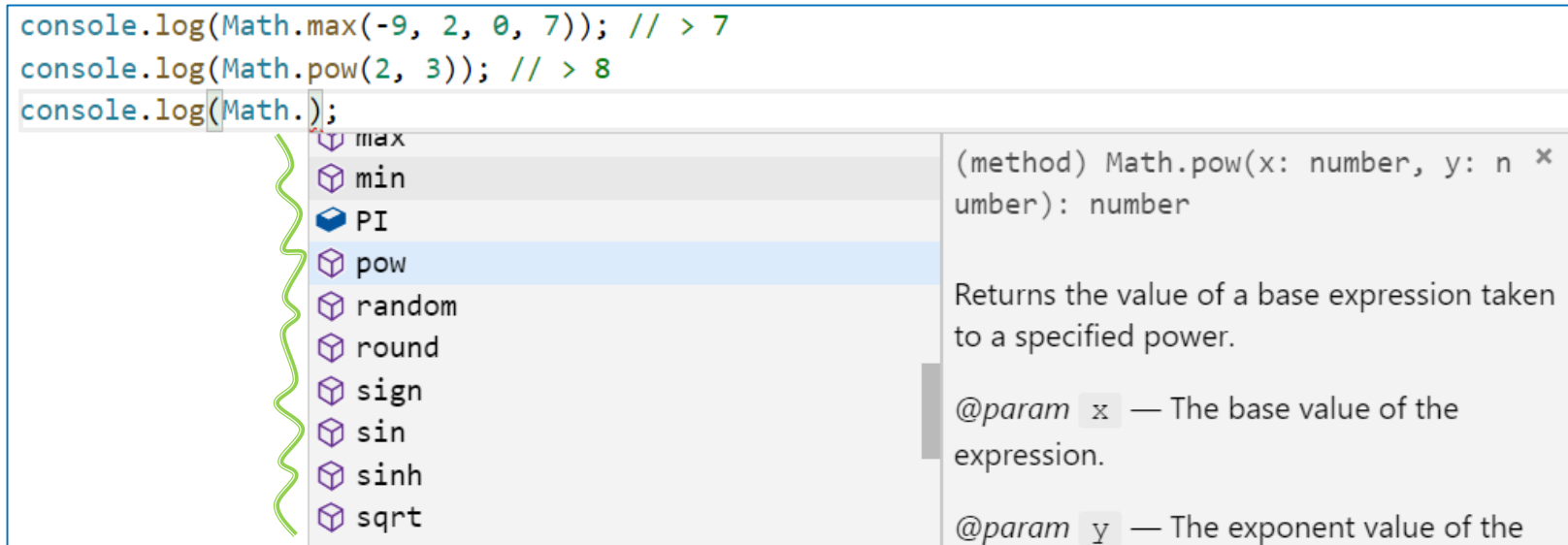
```
const dummyBlogEntry = BlogEntry.createDummy();  
console.log(dummyBlogEntry.body);
```

Nothing much to say today...

Class declaration

- static methods
 - enkele voorgedefinieerde objecten bevatten heel wat interessante static methods
 - voorbeeld: Math bevat enkel static properties/methods

```
console.log(Math.max(-9, 2, 0, 7)); // > 7
console.log(Math.pow(2, 3)); // > 8
console.log(Math.);
```



max	(method) Math.max(x: number, y: number, ...): number
min	(method) Math.min(x: number, y: number, ...): number
PI	(constant) Math.PI: number
pow	(method) Math.pow(x: number, y: number): number Returns the value of a base expression taken to a specified power. @param x — The base value of the expression. @param y — The exponent value of the
random	(method) Math.random(): number
round	(method) Math.round(x: number): number
sign	(method) Math.sign(x: number): number
sin	(method) Math.sin(x: number): number
sinh	(method) Math.sinh(x: number): number
sqrt	(method) Math.sqrt(x: number): number

Class declaration

- **try it yourself**

- zie blog2.js in 03thOopInJavaScript
- vergeet niet: pas eerst index.html aan

```
<script src="js/blog2.js"></script>
```



03 OOP in Javascript

Class declarations

overriving

Class declaration

- **overerving**

- gebruik **extends** om een subklasse van een klasse te maken
 - een subklasse heeft maar 1 superklasse (single inheritance)
- je kan methodes uit de superklasse **overriden**
 - gebruik **super. method(...)** om te refereren naar een methode van de superklasse
- gebruik **super(...)** in de **constructor** om gebruik maken van de constructor van de superklasse
 - deze aanroep **moet** gebeuren en moet bovendien gebeuren **vóór je this gebruikt**
 - indien je geen constructor maakt in de subklasse dan krijgt de subklasse een **default constructor**, deze ziet er als volgt uit:

*de default constructor in
een subklasse*

```
class B extends A {  
  constructor(...args) { super(...args); }  
}
```

Class declaration

- voorbeeld: TaggedBlogEntry extends BlogEntry

```
class TaggedBlogEntry extends BlogEntry {  
  constructor(body, tags) {  
    super(body);  
    this._tags = tags;  
  }  
  get tags() {  
    return this._tags;  
  }  
  addTag(tag) {  
    this._tags.push(tag);  
  }  
  removeTag(tag) {  
    const index = this._tags.indexOf(tag);  
    if (index !== -1) {  
      this._tags.splice(index, 1);  
    }  
  }  
  contains(searchText) {  
    return super.contains(searchText) || this.tags.includes(searchText);  
  }  
}
```

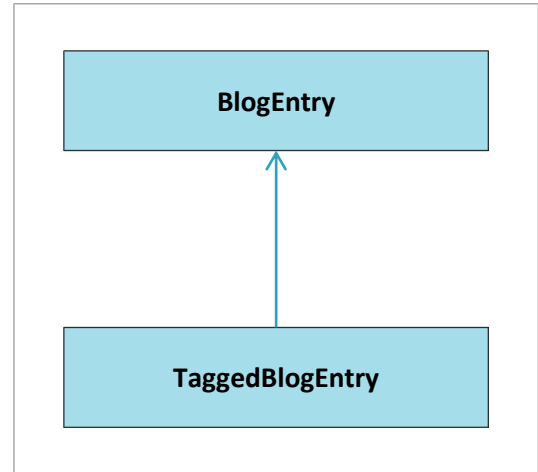
*een TaggedBlogEntry heeft een lijst van tags...
de constructor van de superklasse wordt aangeroepen vóór we this gebruiken*

een getter voor de extra property, er is geen setter voorzien

je kan een tag toevoegen aan de lijst van tags

je kan een tag verwijderen uit de lijst van tags

*de methode contains wordt overschreven, ook de tags worden doorzocht,
de methode contains uit de superklasse wordt aangeroepen...*



Class declaration

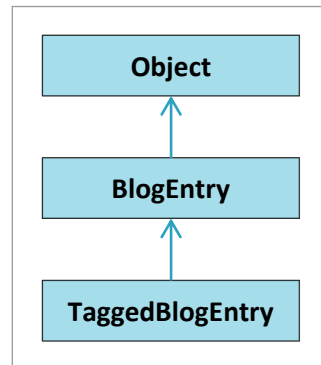
- voorbeeld: gebruik van de subklasse

```
const aTaggedBlogEntry = new TaggedBlogEntry(  
  'Today I want to write about my chickens.',  
  ['pets', 'animals']  
);  
console.log(aTaggedBlogEntry.body); // Today I want to write about my chickens  
aTaggedBlogEntry.addTag('rooster');  
console.log(aTaggedBlogEntry.tags); // (3) ["pets", "animals", "rooster"]  
console.log(aTaggedBlogEntry.contains('pets')); // true  
console.log(aTaggedBlogEntry.contains('write')); // true  
console.log(aTaggedBlogEntry.contains('cube')); // false  
aTaggedBlogEntry.removeTag('animals');  
console.log(aTaggedBlogEntry.tags); // (2) ["pets", "rooster"]
```

in de subklasse maken we gebruik van methodes uit de superklasse en methodes uit de subklasse zelf

Class declaration

- opmerkingen
 - class declarations worden **niet gehoist**
 - je moet de klasse declareren alvorens je ze kan gebruiken!
 - herinner je: function declarations worden wel gehoist...
 - indien een klasse niet expliciet erft van een andere klasse, dan erft de klasse van Object
 - de methodes gedefinieerd in Object kunnen dus steeds gebruikt worden



```
console.log(myBlog.hasOwnProperty('_authors'));  
// true
```

The `hasOwnProperty()` method defined in the Object prototype returns a boolean indicating whether the object has the specified property as own (not inherited) property.

Class declaration

- opmerkingen
 - je kan de built-in javascript klassen ook extenden
 - voorbeeld: BlogDate extends Date

```
class BlogDate extends Date {  
  toBlogFormat() {  
    const options = {  
      weekday: 'long',  
      year: 'numeric',  
      month: 'long',  
      day: 'numeric'  
    };  
    return this.toLocaleDateString('en-NL', options);  
  }  
}
```

```
const blogDate = new BlogDate();  
console.log(`Today's date: ${blogDate.toBlogFormat()}`);
```

Today's date: Friday, 16 February 2018

Class declaration

- voorbeeld: BlogEntry revisited
 - use extended Date, override toString() from Object

```
class BlogDate extends Date {  
  toBlogFormat() {  
    const options = {  
      weekday: 'long',  
      year: 'numeric',  
      month: 'long',  
      day: 'numeric'  
    };  
    return this.toLocaleDateString('en-NL', options);  
  }  
}
```

```
class TaggedBlogEntry extends BlogEntry {  
  //nothing changed, omitted for brevity  
}
```

```
const anEntry = new TaggedBlogEntry(  
  'Today I want to talk about my purple rabbit...',  
  ['pets', 'rabbit']  
);  
console.log(anEntry.toString());  
alert(anEntry); // automatically calls toString()
```

```
class BlogEntry {  
  constructor(body) {  
    this.body = body;  
    this._date = new BlogDate();  
  }  
  toString() {  
    return `${this.date.toBlogFormat()}\n  
    ---\n  
    ${this.body}`;  
  }  
  // rest of class omitted for brevity...  
}
```

Friday, 16 February 2018

Today I want to talk about my purple rabbit...

127.0.0.1:5500 meldt het volgende
Friday, 16 February 2018

Today I want to talk about my purple rabbit...

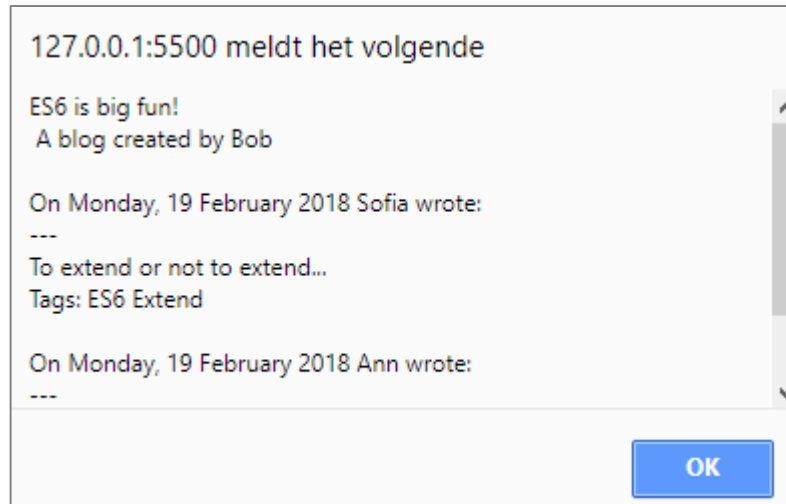
OK

Class declaration

- **try it yourself**

- zie blog3.js in 03thOopInJavaScript
- vergeet niet: pas eerst index.html aan

```
<script src="js/blog3.js"></script>
```



03 OOP in Javascript

Prototypes

Prototypes

- terug naar het begin



Under the hood the new syntax still uses the prototype pattern with constructor functions and the prototype-chain. However, it provides a more common and convenient syntax with less boilerplate code.

```
▼ BlogEntry: class BlogEntry
  arguments: (...)
  caller: (...)
  length: 1
  name: "BlogEntry"
  ▶ prototype: {constructor: f, contains: f, toString: f}
  ▶ proto : f ()
  [[FunctionLocation]]: blogFull3.js:2
  ▶ [[Scopes]]: Scopes[2]
```



Let's have a look at what is happening under the hood!

OO & JavaScript

- ▶ JavaScript has a **prototype-based, object-oriented programming model**.
 - It creates objects using other objects as blueprints and to inherit from them. Inheritance it manipulates what's called a prototype chain.
- ▶ Although the prototype-based model is closer to the object-oriented model, it is not as common and convenient as the object-oriented model.

Under the hood the new syntax still uses the prototype pattern with constructor functions and the prototype-chain. However, it provides a more common and convenient syntax with less boilerplate code.

Prototypes

- in JavaScript is **alles een object**, er bestaan geen klassen zoals we die kennen uit Java
- via **prototype objecten** kunnen objecten toestand en gedrag delen
 - een object heeft zijn **eigen properties en methodes**
 - een object heeft **een property __proto__**
 - **__proto__** is het **prototype object** via dewelke het object **gedrag en toestand erft**

Prototypes

- de constructor & new
 - een JavaScript functie
 - via **this keyword** kunnen properties en methodes gedefinieerd worden

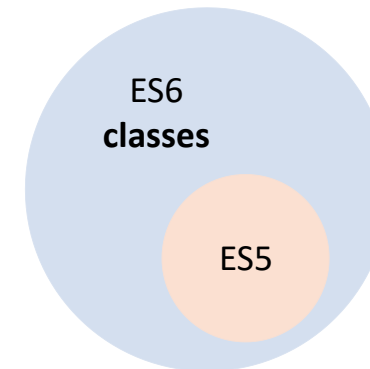
conventie: naam
constructor functie
begint met
hoofdletter

```
function BlogEntry(body, date) {  
  this.body = body;  
  this.date = new Date();  
}
```

ES5

```
class BlogEntry {  
  constructor(body, date) {  
    this.body = body;  
    this.date = date;  
  }  
}
```

ES6
classes



Prototypes

- de constructor & new
 - wanneer je een functie aanroep laat voorafgaan door het new keyword creëer je een object

```
function BlogEntry(body, date) {  
  this.body = body;  
  this.date = new Date();  
}  
  
const myBlogEntry = new BlogEntry('Prototypes rock!');
```

```
▼ myBlogEntry: BlogEntry  
  body: "Prototypes rock!"  
  ▶ date: Sun Feb 18 2018 17:1  
  ▶ __proto__: Object
```

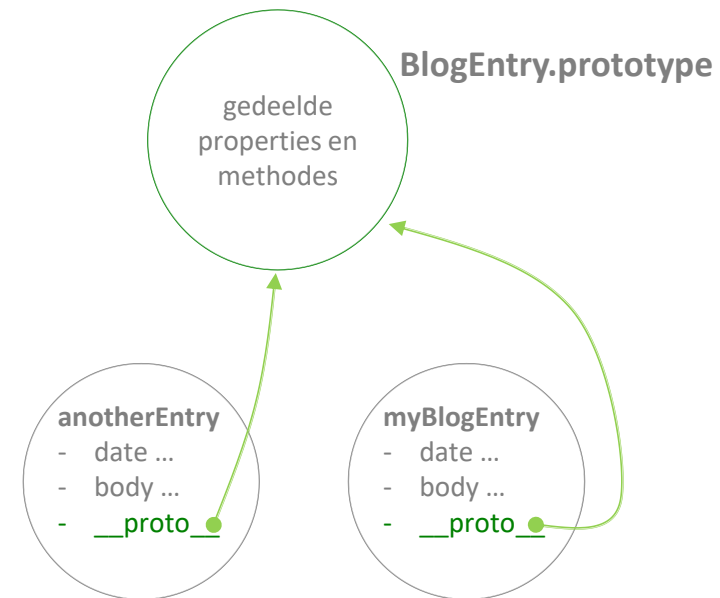
naast de properties body en date heeft het object myBlogEntry ook een property genaamd `__proto__`, dit wijst naar het prototype object via dewelke toestand en gedrag kan geërfd worden

Prototypes

- `__proto__`

```
const myBlogEntry = new BlogEntry('Prototypes rock!');  
const anotherEntry = new BlogEntry('JavaScript rules!');
```

```
▼ myBlogEntry: BlogEntry  
  body: "Prototypes rock!"  
  ▶ date: Sun Feb 18 2018 17:30:38 GMT+0100 (Romance (standaarttijd)) {}  
  ▼ __proto__:  
    ▼ constructor: f BlogEntry(body, date)  
      arguments: null  
      caller: null  
      length: 2  
      name: "BlogEntry"  
      ▶ prototype: {constructor: f}  
      ▶ __proto__: f ()  
      [[FunctionLocation]]: prototypes1.js:1  
      ▶ [[Scopes]]: Scopes[2]  
      ▶ __proto__: Object  
  
  > console.log(myBlogEntry.__proto__ === anotherEntry.__proto__);  
  true  
  
  > console.log(myBlogEntry.__proto__ === BlogEntry.prototype);  
  true
```



*de twee instanties van `BlogEntry` wijzen via hun `__proto__` property naar eenzelfde **prototype-object***

*dit prototype-object wijst naar de prototype property van `BlogEntry`, dat is de **constructor** die werd gebruikt om de instanties aan te maken...*

Prototypes

- het prototype object is ook maar een object...
- ... en heeft dus op zijn beurt ook een prototype...
- zo ontstaat er een ketting van prototype objecten:
prototype chain
- wanneer naar een property van een object wordt gerefereerd
 - dan wordt eerst gezien of die property bestaat bij het object zelf,
 - indien niet gevonden dan wordt er gezocht bij zijn prototype,
 - dit proces herhaalt zich recursief, tot het oer-object wordt bereikt: Object
 - Object.prototype is de laatste schakel in de ketting

Prototypes

- alle properties/methodes die je toevoegt aan de **prototype property van een constructor functie** worden gedeeld door alle instanties die gecreëerd werden via die constructor functie
- voorbeeld 1

```
const myBlogEntry = new BlogEntry('Prototypes rock!');  
const anotherEntry = new BlogEntry('JavaScript rules!');  
BlogEntry.prototype.language = 'EN';  
  
console.log(myBlogEntry.language); // EN  
console.log(anotherEntry.language); // EN
```

*myBlogEntry heeft zelf **geen** property genaamd language
de prototype chain wordt gevolgd, bij zijn prototype, dat is BlogEntry.prototype, wordt de property wel gevonden...*

alle nieuwe instanties van BlogEntry hebben dus, via hun prototype, een language die ingesteld staat op 'EN'

Prototypes

- voorbeeld vervolg

```
const myBlogEntry = new BlogEntry('Prototypes rock!');
const anotherEntry = new BlogEntry('JavaScript rules!');
BlogEntry.prototype.language = 'EN';

console.log(myBlogEntry.language); // EN
console.log(anotherEntry.language); // EN
myBlogEntry.language = 'NL';
console.log(myBlogEntry.language); // NL
console.log(anotherEntry.language); // EN
```

*In H01 hebben we gezien dat je aan een object steeds properties kan toevoegen;
myBlogEntry heeft nu een property language, anotherBlogEntry heeft die property niet!
Bekijk het resultaat...*

```
> myBlogEntry.hasOwnProperty('language')
< true
> anotherEntry.hasOwnProperty('language')
< false
```

*hasOwnProperty is gedefinieerd in het prototype van
Object, het bepaalt of een object al dan niet een bepaalde
property zelf heeft (niet via een prototype)*

Prototypes

- voorbeeld vervolg
 - gedrag toevoegen via prototype

```
const myBlogEntry = new BlogEntry('Prototypes rock!');  
const anotherEntry = new BlogEntry('JavaScript rules!');  
BlogEntry.prototype.language = 'EN';  
BlogEntry.prototype.contains = function(searchText) {  
    return this.body.toUpperCase().indexOf(searchText.toUpperCase()) !== -1;  
};
```

```
> console.log(myBlogEntry.contains('rules'));  
false
```

```
> console.log(myBlogEntry.contains('rock'));  
true
```

Prototypes

- JavaScript komt met heel wat standaard, built-in objecten...

Fundamental objects

These are the fundamental, basic objects upon which all other objects are based. This includes objects that represent general objects, functions, and errors.

- `Object`
- `Function`
- `Boolean`
- `Symbol`
- `Error`
- `EvalError`
- `InternalError`
- `RangeError`
- `ReferenceError`
- `SyntaxError`
- `TypeError`
- `URIError`

Numbers and dates

These are the base objects representing numbers, dates, and mathematical calculations.

- `Number`
- `Math`
- `Date`

Text processing

These objects represent strings and support manipulating them.

- `String`
- `RegExp`

voor een volledige lijst: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

Prototypes

- vervolg...

Indexed collections

These objects represent collections of data which are ordered by an index value. This includes (typed) arrays and array-like constructs.

- `Array`
- `Int8Array`
- `Uint8Array`
- `Uint8ClampedArray`
- `Int16Array`
- `Uint16Array`
- `Int32Array`
- `Uint32Array`
- `Float32Array`
- `Float64Array`



Keyed collections

These objects represent collections which use keys; these contain elements which are iterable in the order of insertion.


- `Map`
- `Set`
- `WeakMap`
- `WeakSet`

Structured data

These objects represent and interact with structured data buffers and data coded using JavaScript Object Notation (JSON).

- `ArrayBuffer`
- `SharedArrayBuffer` 
- `Atomics` 
- `DataView`
- `JSON`

Control abstraction objects

- `Promise`
- `Generator`
- `GeneratorFunction`
-  `AsyncFunction`

Reflection

- `Reflect`
- `Proxy`

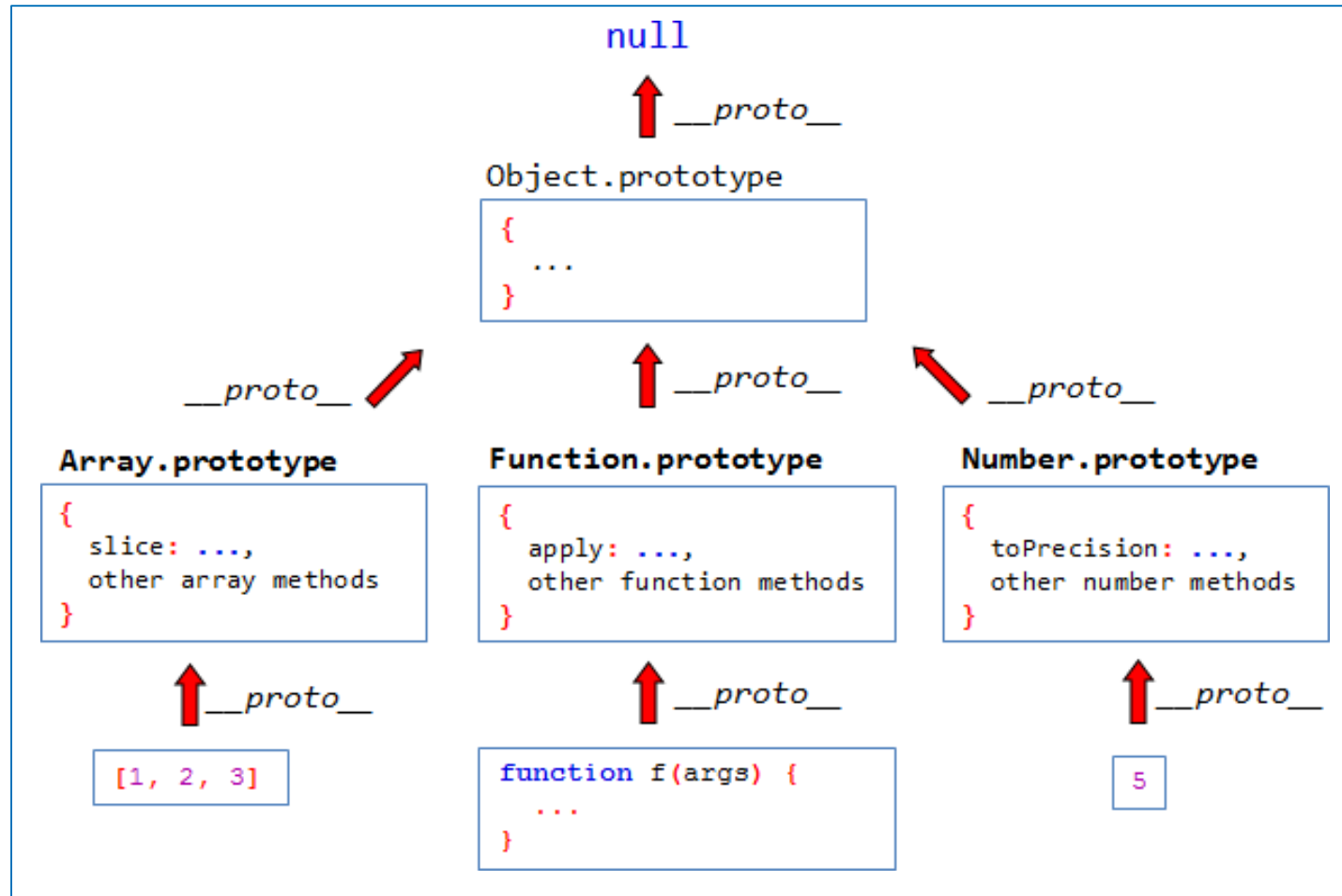
Internationalization

Additions to the ECMAScript core for language-sensitive functionalities.

- `Intl`
- `Intl.Collator`

Prototypes

- built-in objects & prototype chain:



Prototypes

- voorbeeld: **Object.prototype**
 - de laatste link in de prototype chain
 - deze methodes kunnen op elk object aangeroepen worden, tenzij ze overriden worden...

```
▶ constructor: f Object()  
▶ hasOwnProperty: f hasOwnProperty()  
▶ isPrototypeOf: f isPrototypeOf()  
▶ propertyIsEnumerable: f propertyIsEnumerable()  
▶ toLocaleString: f toLocaleString()  
▶ toString: f toString()  
▶ valueOf: f valueOf()
```

Prototypes

- nu je de basis van prototypes kent kan je de uitleg op MDN beter begrijpen...
- wat kan je doen met **Arrays? Array.prototype!**

```
▼ Properties  
Array.length  
Array.prototype
```

```
▼ Methods  
Array.from()  
Array.isArray()  
Array.observe()  
Array.of()  
Array.prototype.concat()  
Array.prototype.copyWithin()  
Array.prototype.entries()  
Array.prototype.every()  
Array.prototype.fill()  
Array.prototype.filter()  
Array.prototype.find()  
Array.prototype.findIndex()  
Array.prototype.flatten()  
Array.prototype.forEach()  
Array.prototype.includes()  
Array.prototype.indexOf()
```

Zie https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

```
Array.prototype.slice()  
Array.prototype.some()  
Array.prototype.sort()  
Array.prototype.splice()  
Array.prototype.toLocaleString()  
Array.prototype.toSource()  
Array.prototype.toString()  
Array.prototype.unshift()  
Array.prototype.values()
```

```
Array.prototype.join()  
Array.prototype.keys()  
Array.prototype.lastIndexOf()  
Array.prototype.map()  
Array.prototype.pop()  
Array.prototype.push()  
Array.prototype.reduce()  
Array.prototype.reduceRight()  
Array.prototype.reverse()  
Array.prototype.shift()
```

Prototypes

Comparison of class-based (Java) and prototype-based (JavaScript) object systems	
Class-based (Java)	Prototype-based (JavaScript)
Class and instance are distinct entities.	All objects can inherit from another object.
Define a class with a class definition; instantiate a class with constructor methods.	Define and create a set of objects with constructor functions.
Create a single object with the <code>new</code> operator.	Same.
Construct an object hierarchy by using class definitions to define subclasses of existing classes.	Construct an object hierarchy by assigning an object as the prototype associated with a constructor function.
Inherit properties by following the class chain.	Inherit properties by following the prototype chain.
Class definition specifies <i>all</i> properties of all instances of a class. Cannot add properties dynamically at run time.	Constructor function or prototype specifies an <i>initial set</i> of properties. Can add or remove properties dynamically to individual objects or to the entire set of objects.

Excerpt from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model

**HO
GENT**