

The background of the slide features a photograph of a modern building with a facade of large, light-colored concrete panels. Some panels are recessed, creating a textured, three-dimensional effect. A person with long brown hair, wearing a brown jacket, blue jeans, and a red backpack, is walking from left to right in the lower half of the image. Green plants are visible in the bottom left corner.

Web Development II

Hoofdstuk 04 – OOP in Javascript

Inhoud

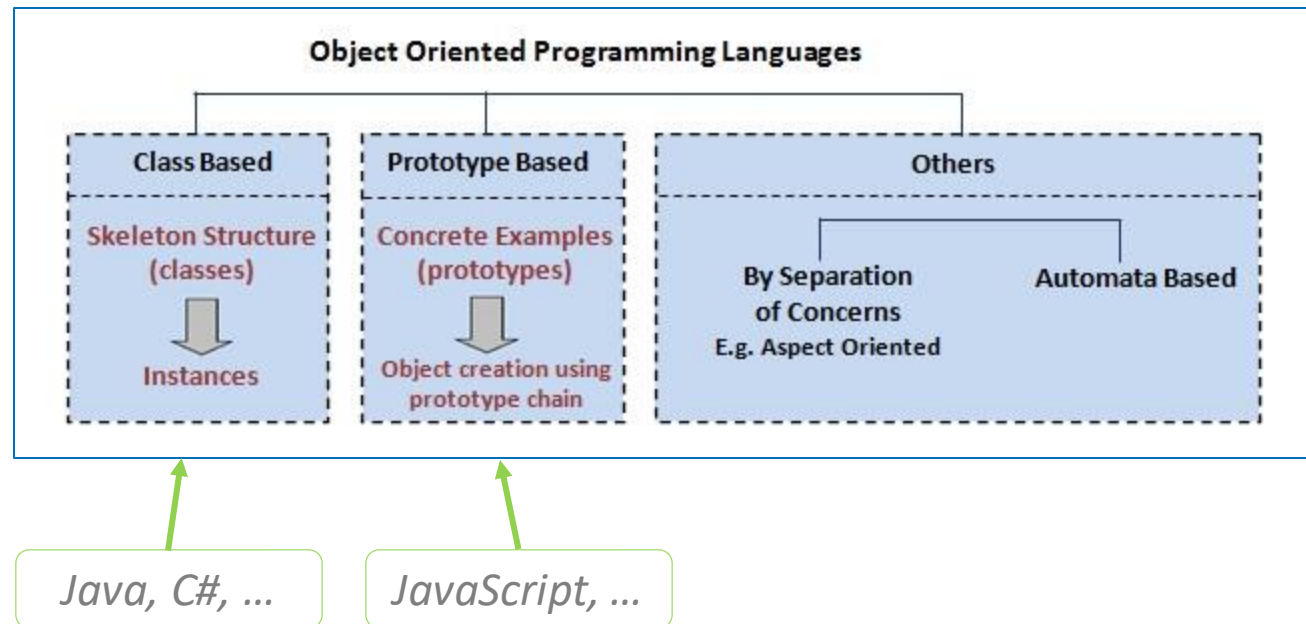
- Object oriented programming in JavaScript
 - Object orientation in JavaScript
 - Class
 - constructor method & new keyword
 - get/set accessors
 - other methods
 - static members
 - inheritance
 - Prototypes, a little background
 - concept
 - constructor function & new keyword
 - prototype chain
 -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

04 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**

04 OOP in Javascript

Class declarations

Class declaration

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

```
class ClassName {  
  
    (static) fields  
  
    constructor (...) {...}  
  
    (static) methods(...) {...}  
  
    get someProp() {...}  
    set someProp(value) {...}  
  
}
```

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 {  
  body;   
  date = new Date();  
  
  constructor(body) {  
    this.body = body;  
  }  
}
```

fields gedeclareerd zonder
initialisatie krijgen de waarde
'undefined'

fields kunnen ook in de constructor
geïnitieerd worden; gebruik
steeds het **this** keyword!

een basis klasse declaratie voor BlogEntry

Class declaration

Fields zijn standaard **publiek toegankelijk**.

Maak gebruik van **hash-names** om een field **private toegankelijkheid** te geven.

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

BlogEntry met de publieke fields body en date

```
class BlogEntry {  
  #body;  
  #date = new Date();  
  
  constructor(body) {  
    this.#body = body;  
  }  
}
```

*BlogEntry met de private fields #body en #date;
hash-names starten met #*



*in JavaScript gebruik je **geen access modifier** maar is het de **naamgeving** die de toegankelijkheid bepaalt!*

Class declaration

Je hoeft **public fields** niet **expliciet** te declareren in een class.
Je moet **private fields** **expliciet** declareren in een class.

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



BlogEntry met expliciete declaratie van public fields

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



BlogEntry zonder expliciete declaratie van public fields

```
class BlogEntry {  
    #body;  
    #date = new Date();  
  
    constructor(body) {  
        this.#body = body;  
    }  
}
```



BlogEntry met expliciete declaratie van private fields

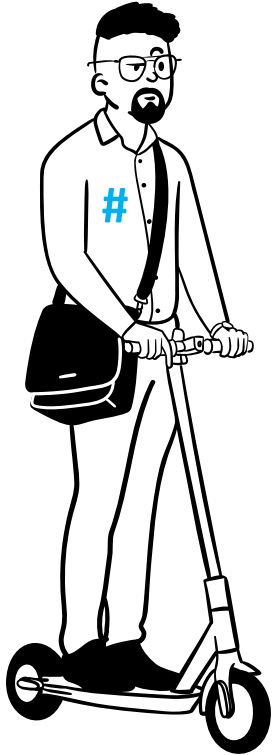
```
class BlogEntry {  
    constructor(body) {  
        this.#body = body;  
        this.#date = new Date();  
    }  
}
```



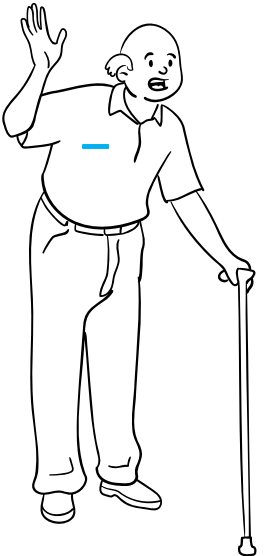
BlogEntry zonder expliciete declaratie van private fields

Class declaration

is the new _



- **private class members** werden relatief recent geïntroduceerd in JavaScript [ES2022]
 - vroeger kon je enkel gebruik maken van public members
- in heel wat JavaScript code kom je members tegen met een naam die begint met een **underscore**
 - vb. `_body`, `_entries`, `_creator`, ...
 - dit werd vroeger gebruikt om aan te geven dat een field bedoeld is om private te zijn
 - respecteer deze algemeen aanvaarde afspraak als je dit in code tegenkomt en behandel dergelijke fields als private



Class declaration

- de notie van een property in (prototype based) Javascript (zie H03) valt niet samen met de notie van een property in een class
 - bv. private fields, met get/set, die wij in OOP properties noemen ga je niet zomaar terugvinden als properties van een prototype object
 - we zullen verderop in dit hoofdstuk de notie van een property gebruiken zoals we die kennen uit OOP
 - ook tijdens het debuggen wordt je geconfronteerd met het feit dat JavaScript prototype based is en dat classes/objecten vertaald worden naar prototypes

04 OOP in Javascript

Class declarations

new & constructors

Instantiatie

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

```
class BlogEntry {  
  #date = new Date();  
  #body;  
  
  constructor(body) {  
    this.#body = body;  
  }  
}
```

```
const aBlogEntry = new BlogEntry('I am a newly created instance of BlogEntry');
```

er wordt een nieuwe instantie van BlogEntry gemaakt

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

aBlogEntry is of type object

► Object { #date: Date Wed Feb 23 2022 11:16:13 GMT+0100 (Midden-Europese standaardtijd), #body: "I am a newly created instance of BlogEntry" }

Instantiatie

- de **constructor**
 - gebruik keyword **constructor** met eventuele parameters
 - gebruik steeds **this** om naar fields te verwijzen
 - **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

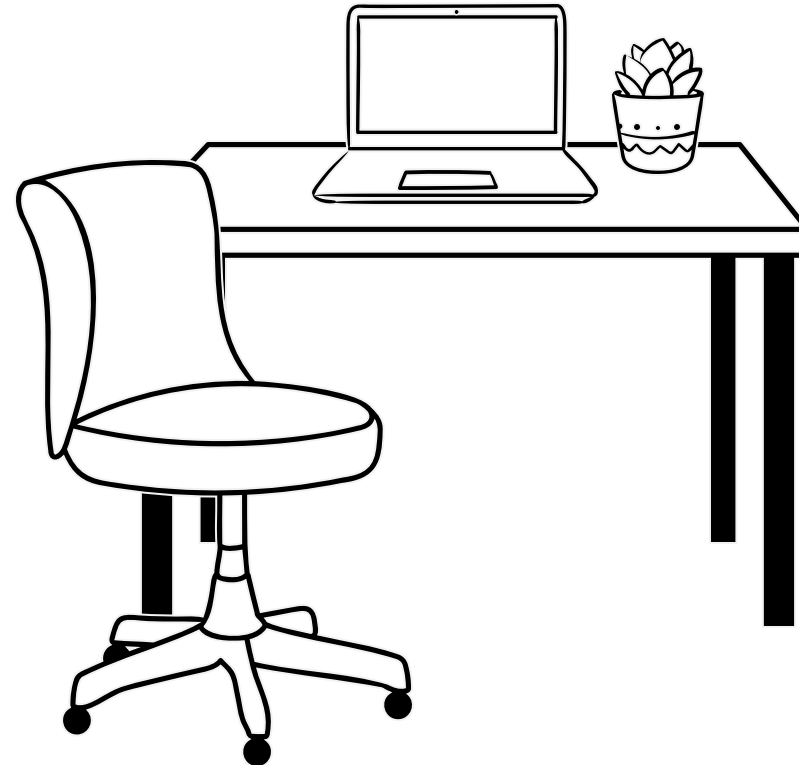
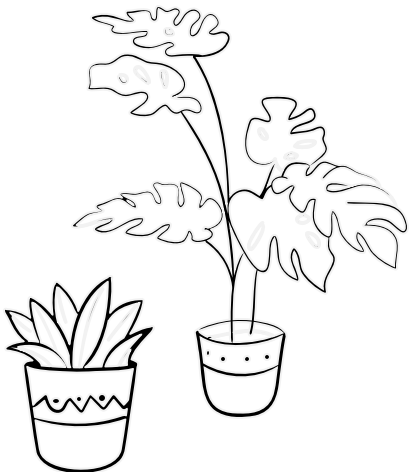
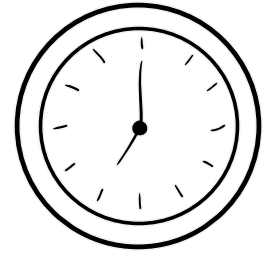


# Class declaration

- **try it yourself**

- zie blog01.js in 04thOOP\_In\_JavaScript
- zorg dat het juiste script geladen wordt

```
<script src="js/blog01.js"></script>
```



# 04 OOP in Javascript

Class declarations

properties: getters & setters

# Getters & Setters

- in een klasse kan je twee speciale soorten properties gebruiken: **accessor properties**
  - **get accessor**
    - associeert een property met een **parameterloze functie**
    - de functie wordt uitgevoerd telkens de property wordt **uitgelezen**
  - **set accessor**
    - associeert een property met een **functie die 1 parameter heeft**
    - de functie wordt uitgevoerd telkens de property wordt **gewijzigd**

# Getters & Setters

- **get accessor property**

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

```
get property_name () { body }
```

- belangrijk:
  - je gebruikt de get accessor als een gewone property, je gebruikt dus **geen haakjes wanneer je de property wil raadplegen**
  - de functie wordt automatisch uitgevoerd telkens de waarde van de property wordt geraadpleegd

# Getters & Setters

- voorbeeld get

```
class BlogEntry {
 #date = new Date();
 #body;

 constructor(body) {this.#body = body;}

 // KLASSIEK: via publieke getters gecontroleerde toegang geven tot private properties
 get body() {return this.#body;}

 get date() {return this.#date;}

 // Via een getter kan een berekende waarde verkregen worden zonder een methode-aanroep
 get outDated() {return new Date().getFullYear() - this.date.getFullYear() >= 5;}
}
```

```
const aBlogEntry = new BlogEntry('I am learning about get
accessors');
console.log(aBlogEntry.body);
console.log(aBlogEntry.date);
console.log(aBlogEntry.outDated);
```

I am learning about get accessors

► Date Wed Feb 23 2022 22:11:40 GMT+0100 (Midden-Europese standaardtijd)

false

# Getters & Setters

- **set accessor property**

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

```
set property_name (value) { body }
```

- belangrijk:
  - je gebruikt de set accessor als een gewone property, je gebruikt dus **geen haakjes wanneer je aan de property een waarde wil toekennen**
  - de functie wordt automatisch uitgevoerd telkens de waarde van de property wordt gewijzigd

# Getters & Setters

- voorbeeld set

```
class BlogEntry {
 #date = new Date();
 #body;

 // MERK OP: vanuit de constructor wordt de setter aangeroepen
 constructor(body) { this.body = body; }

 get body() { return this.#body; }

 set body(value) { this.#body = value || 'This entry is work in progress'; }
}
```

```
const aBlogEntry = new BlogEntry();
console.log(aBlogEntry.body);
aBlogEntry.body = "Let's dance";
console.log(aBlogEntry.body);
aBlogEntry.body = '';
console.log(aBlogEntry.body);
```

This entry is work in progress

Let's dance

This entry is work in progress

# Getters & Setters

- de get/set accessor properties kan je **private** maken door ze een **hash-name** te geven
- let op: je kan geen twee properties met eenzelfde naam hebben in een klasse

```
class BlogEntry {
 #date = new Date();
 #entryBody;

 constructor(body) { this.#body = body; }

 get body() { return this.#entryBody; }

 set #body(value) { this.#entryBody = value || 'This entry is work in progress'; }
}
```

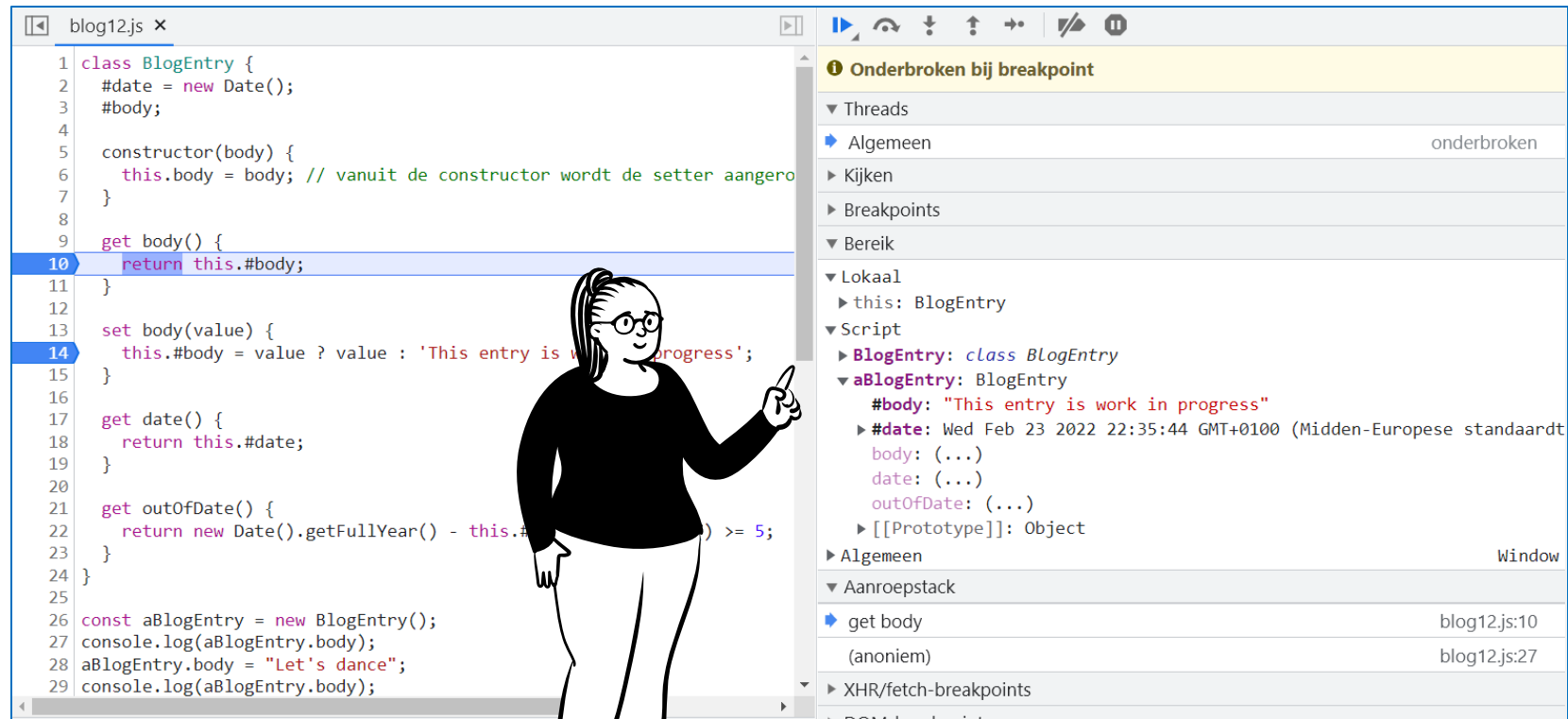
```
let aBlogEntry = new BlogEntry();
console.log(aBlogEntry.body);
aBlogEntry = new BlogEntry('Still dancing!');
console.log(aBlogEntry.body);
```

```
This entry is work in progress
Still dancing!
```



# Getters & Setters

- Tip: maak regelmatig gebruik van de debugger om bv. breakpointsgewijs of stapsgewijs door je script te wandelen; bekijk de aanroepstack en de variabelen in het Script-bereik



# Getters & Setters

- get/set accessors kan je ook gebruiken in **object literals** (cf. H03)

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

- in object literals kan je **geen private properties** definiëren

# Getters & 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

# Getters & Setters

- In tegenstelling tot Java hoeft je in JavaScript geen getters en setters aan te maken als je ze (nog) niet nodig hebt.
- Getters en setters hebben namelijk dezelfde naam heeft als het publieke field dat ze vervangen.
- De interface van je klasse blijft dus behouden en de client code blijft werken.

# Getters & Setters

```
// v1. Public field
class Student {
 name;
}

// Client code
const alice = new Student();
alice.name = "Alice";
console.log(alice.name);
```

```
// v2. Private field met getter en setter
class Student {
 #name;

 get name() { return this.#name; }
 set name(value) { this.#name = value; }
}

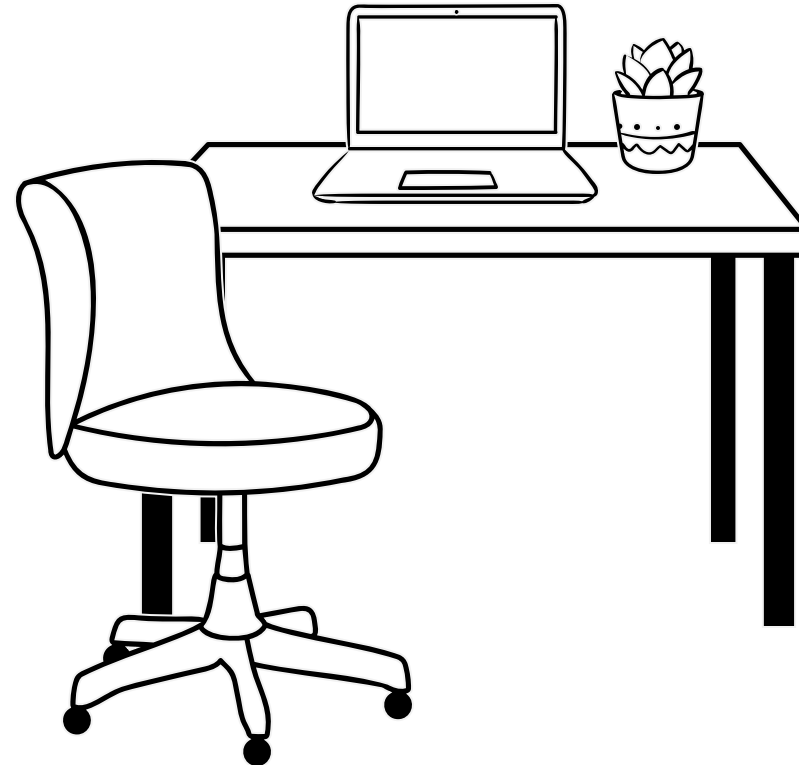
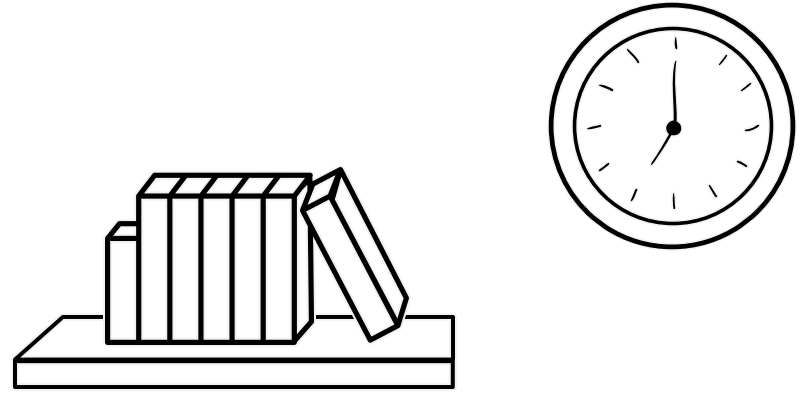
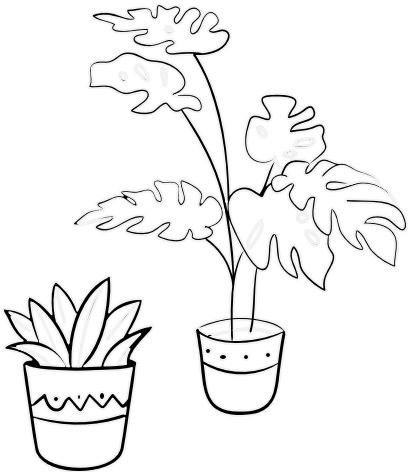
// Client code
const alice = new Student();
alice.name = "Alice";
console.log(alice.name);
```

# Getters & Setters

- **try it yourself**

- zie blog02.js in 04thOOP\_In\_JavaScript
- vergeet niet index.html aan te passen

```
<script src="js/blog02.js"></script>
```



**HO  
GENT**

# 04 OOP in Javascript

Class declarations

methodes

# Methodes

- **methode declaratie**

```
method_name (parameters) { method_body }
```

- gebruik in de method body steeds **this** om te verwijzen naar fields
- naam, parameters, return waarde zie H02: Functies – Arrays - Modules



# Methodes

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

```
class BlogEntry {
 #date = new Date();
 #entryBody;
 constructor(body) {
 this.#body = body;
 }

 contains(searchText) {
 return searchText
 ? this.body.toUpperCase().includes(searchText.toUpperCase())
 : false;
 }
}
```

```
const aBlogEntry = new BlogEntry('This is a brand new blog entry');
let word = 'hello';
```

```
console.log(`aBlogEntry ${aBlogEntry.contains(word)} ? 'contains' : 'does not contain'} the word "${word}".`);
word = 'brand';
console.log(`aBlogEntry ${aBlogEntry.contains(word)} ? 'contains' : 'does not contain'} the word "${word}".`);
```

aBlogEntry does not contain the word "hello".

aBlogEntry contains the word "brand".

# Methodes

- gebruik hash-names voor private methodes
- voorbeeld: class BlogEntry met een **private methode**

```
class BlogEntry {
 #date = new Date();
 #body;
 constructor(body) {
 this.#body = this.#checkBody(body);
 }

 #checkBody(body) {
 return body || 'Work in progress...';
 }
}
```

```
let aBlogEntry = new BlogEntry('This is a brand new blog entry');
console.log(aBlogEntry);
aBlogEntry = new BlogEntry();
console.log(aBlogEntry);
```

```
► Object { #date: Date Wed Feb 23 2022 20:09:14 GMT+0100 (Midden-Europese standaardtijd), #body: "This is a brand new blog entry" }
```

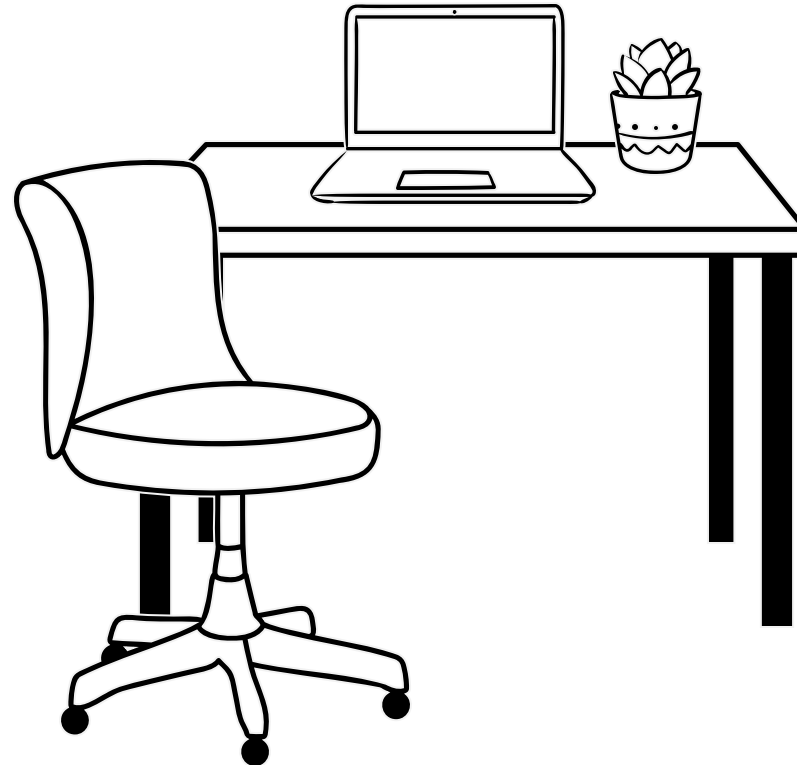
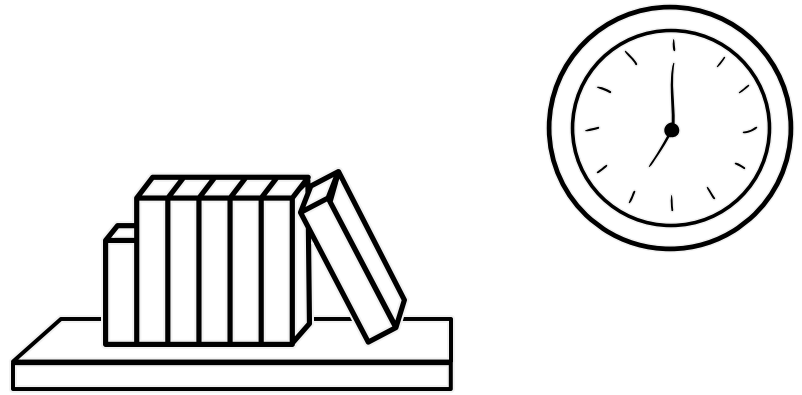
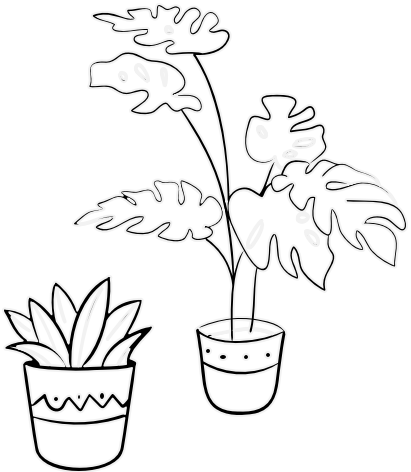
```
► Object { #date: Date Wed Feb 23 2022 20:09:14 GMT+0100 (Midden-Europese standaardtijd), #body: "Work in progress..." }
```

# Methodes

- **try it yourself**

- zie blog03.js in 04thOOP\_In\_JavaScript
- vergeet niet: pas eerst index.html aan

```
<script src="js/blog03.js"></script>
```



# 04 OOP in Javascript

Class declarations

static class members

# Static class members

- **static fields/properties**

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

```
class BlogEntry {
 static #wordsInShortBody = 5;
 #date = new Date();
 #entryBody;

 static get wordsInShortBody() {
 return BlogEntry.#wordsInShortBody;
 }

 get shortBody() {
 return (
 this.body.split(' ').slice(0, BlogEntry.wordsInShortBody).join(''
) + '...'
);

 // parts omitted...
 }
}
```

declaratie: gebruik  
keyword *static*

een static get  
accessor

aanroep: gebruik  
de naam van de  
klasse

```
const aBlogEntry = new BlogEntry('This is the story of my olympic gold medal.');
```

```
console.log(BlogEntry.wordsInShortBody);
console.log(aBlogEntry.shortBody);
```

5

This is the story of...

# Static class members

- **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 {
 static #wordsInShortBody = 5;
 #date = new Date();
 #body;

 // parts omitted...

 static createDummy() { return new this('Nothing much to say today...'); }
}
```

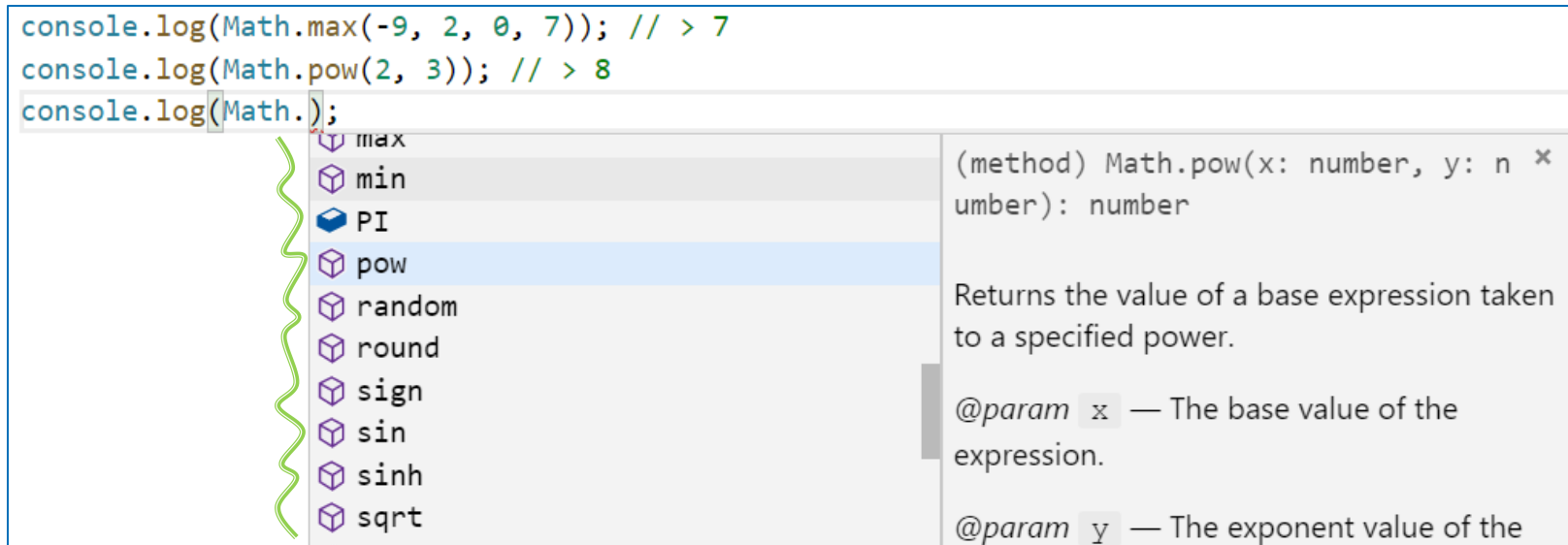
*aanroep: gebruik  
de naam van de  
klasse*

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

Nothing much to say today...

# Static class members

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



# Static class members

- MDN reference: instance vs static methods
  - instance methods zijn gedefinieerd op “prototype”
  - `vb. const a = [1, 2, 3]; a.push(100);`

## **Array.prototype.push()**

The `push()` method adds one or more elements to the end of an array and returns the new length of the array.

- static methods zijn niet gedefinieerd op “prototype”
- `vb. const arrayWithLetters = Array.from('JavaScript');`

## **Array.from()**

The `Array.from()` static method creates a new, shallow-copied `Array` instance from an array-like or iterable object.

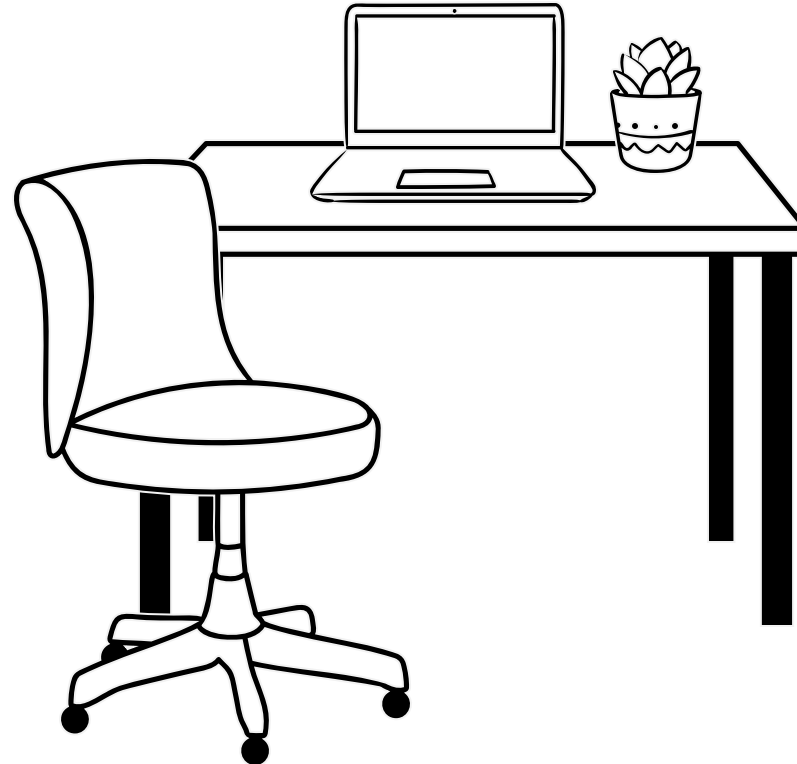
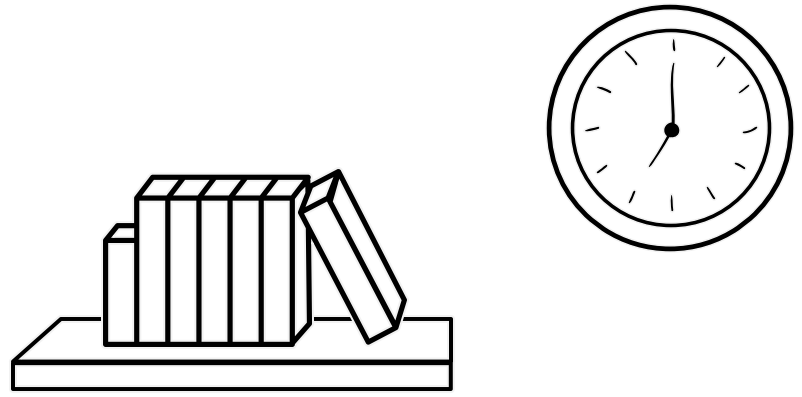
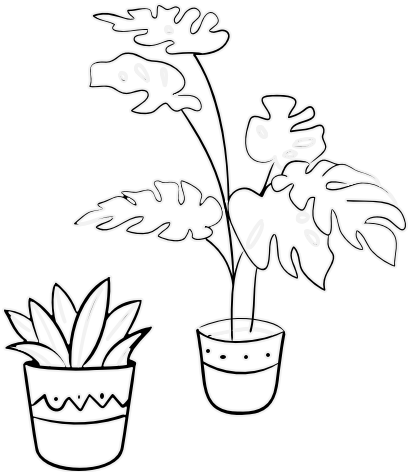


# Methodes

- **try it yourself**

- zie blog04.js in 04thOOP\_In\_JavaScript
- vergeet niet: pas eerst index.html aan

```
<script src="js/blog04.js"></script>
```



# 04 OOP in Javascript

Class declarations

overriving

# Overerving

- JavaScript kent **single inheritance**
  - een subklasse heeft maar 1 superklasse
- gebruik keyword **extends** om een subklasse van een klasse te maken
- een subklasse heeft geen toegang tot private fields van de superklasse
- gebruik de **instanceof** operator om na te gaan of een object een instantie van een bepaalde klasse is

# Overerving

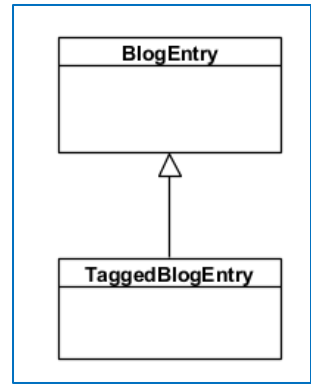
- je kan methodes uit de superklasse **overriden**
  - gebruik **super.method(parameters)** om expliciet te refereren naar een methode van de superklasse
- gebruik **super(parameters)** in de **constructor** om expliciet de constructor van de superklasse aan te roepen
  - deze aanroep **moet** gebeuren 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); }
}
```

# Overerving

- voorbeeld: TaggedBlogEntry extends BlogEntry



```
class TaggedBlogEntry extends BlogEntry {
 #tags;

 constructor(body, author, ...tags) {
 super(body, author);
 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 is een BlogEntry met 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...*

# Overerving

- voorbeeld: gebruik van de subklasse

```
const myTaggedEntry = new TaggedBlogEntry(
 'A day in the life of a UNICEF Goodwill Ambassador',
 'Nafi Thiam',
 'United Nations', 'UNICEF'
);
console.log(myTaggedEntry.body); // A day in the life of a UNICEF Goodwill Ambassador
myTaggedEntry.addTag('children');
console.log(myTaggedEntry.tags); // ["United Nations", "UNICEF", "children"]
console.log(myTaggedEntry.contains('life')); // true
console.log(myTaggedEntry.contains('children')); // true
myTaggedEntry.removeTag('United Nations');
console.log(myTaggedEntry.tags); // ["UNICEF", "children"]
console.log(myTaggedEntry instanceof TaggedBlogEntry); // true
console.log(myTaggedEntry instanceof BlogEntry); // true
console.log(myTaggedEntry instanceof Object); // true
```

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

# Overerving

- voorbeeld: polymorfisme

```
let entries = [];
entries.push(new BlogEntry('Simple entry', 'Nafi Thiam'));
entries.push(
 new TaggedBlogEntry('Tagged entry', 'Matthias Casse', 'judo', 'olympics')
);
for (const entry of entries) {
 console.log(entry.toString());
}
```

```
On Friday, 4 March 2022 Nafi Thiam wrote:

Simple entry
```

```
On Friday, 4 March 2022 Matthias Casse wrote:

Tagged entry
Tags: judo, olympics
```

# Overerving

- **Object** staat bovenaan de hiërarchie van overerving
- elke klasse die niet expliciet erft van een andere klasse erft van Object
  - in H03 maakte je kennis met enkele **static methods** van Object

```
Object.keys()
Object.values()
Object.entries()
```
  - als je klassiek OOP-stijl, met klassen en instanties van klassen, programmeert, dan pikken deze methodes enkel **publieke non-static properties** van je object op



# Overerving

- **toString()** en **toLocaleString()** zijn interessante instance methods van object die dikwijls overridden worden

```
const number1 = 123456.789;
console.log(number1.toString()); // "123456.789"
console.log(number1.toLocaleString('nl-BE')); // "123.456,789"
console.log(number1.toLocaleString('en-UK')); // "123,456.789"
```

*number1 is van een primitief type number en wordt ge-autoboxed naar een Number*

```
class BlogEntry {
 // parts omitted
 toString() {
 return `On ${this.#toBlogFormat(this.#date)} ${this.author} wrote:\n---\n${this.body}`;
 }
}
```

```
class TaggedBlogEntry {
 // parts omitted
 toString() {
 return `${super.toString()}\nTags: ${this.tags.join(', ')} `;
 }
}
```

# Overerving

- opmerkingen
  - class declarations worden **niet gehoist**
    - je moet de klasse declareren alvorens je ze kan gebruiken!
    - herinner je: function declarations worden wel gehoist...
  - net zoals function expressions kan je ook gebruik maken van **class expressions**
    - meestal wordt dan een anonieme klasse gebruikt

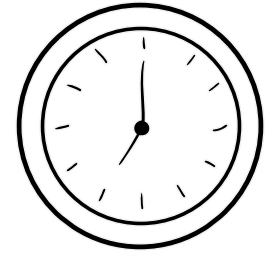
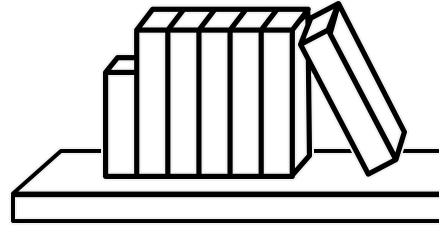
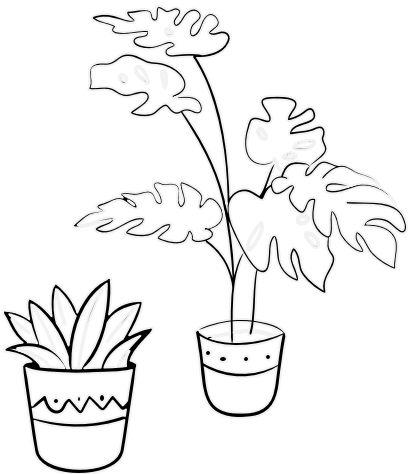
```
const BlogEntry = class {
 // body of class omitted
}
```

# Overerving

- **try it yourself**

- zie blog05.js in 04thOOP\_In\_JavaScript
- vergeet niet: pas eerst index.html aan

```
<script src="js/blog05.js"></script>
```



# 04 OOP in Javascript

Modules

# Modules

- je kan klasse definities in modules opnemen

```
export default class BlogEntry {
 // body of class omitted
}
```

BlogEntry.js

```
import BlogEntry from "./BlogEntry.js";

export default class TaggedBlogEntry extends BlogEntry
{
 // body of class omitted
}
```

TaggedBlogEntry.js

```
import Blog from "./Blog.js";
import GroupBlog from "./GroupBlog.js";
const ourGroupBlog = new GroupBlog('Nafi Thiam', 'Matthias Casse', 'Bashir Abdi');
const woutsBlog = new Blog('Wout Van Aert');
// rest omitted
```

index.js

```
import BlogEntry from './BlogEntry.js';
import TaggedBlogEntry from './TaggedBlogEntry.js';
```

```
export default class Blog {
 // body of class omitted
}
```

Blog.js

```
import Blog from "./Blog.js";
export default class GroupBlog extends Blog {
 // body of class omitted
}
```

GroupBlog.js

# 04 OOP in Javascript

Prototypes  
a little background

# 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 not strictly object-oriented, it is often referred to as such by developers.

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

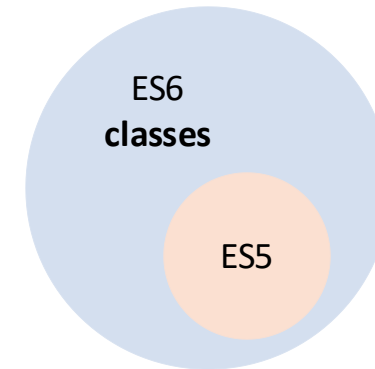
conventie: naam  
constructor functie  
begint met  
hoofdletter

```
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 = date;
}

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

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

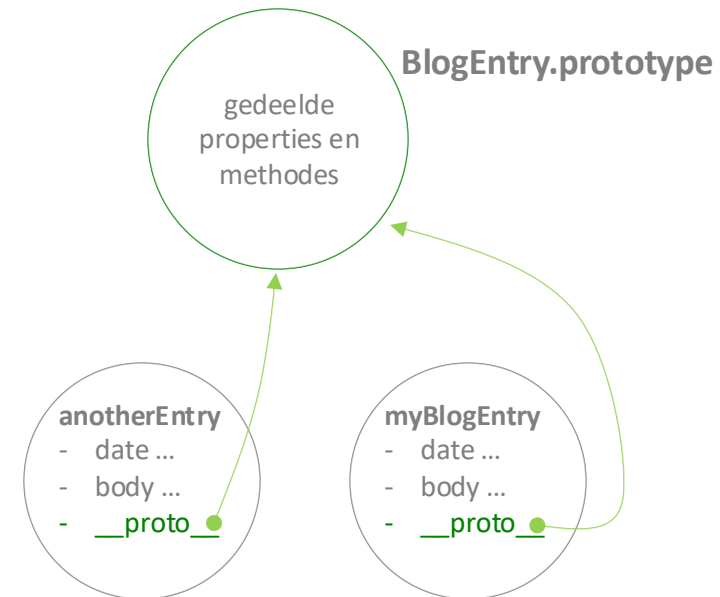
naast de properties body en date heeft het object myBlogEntry ook een property genaamd `__proto__`, dit wijst naar het prototype object via dewelke toestand en gedrag kan geërfd worden

# 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 H03 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](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`

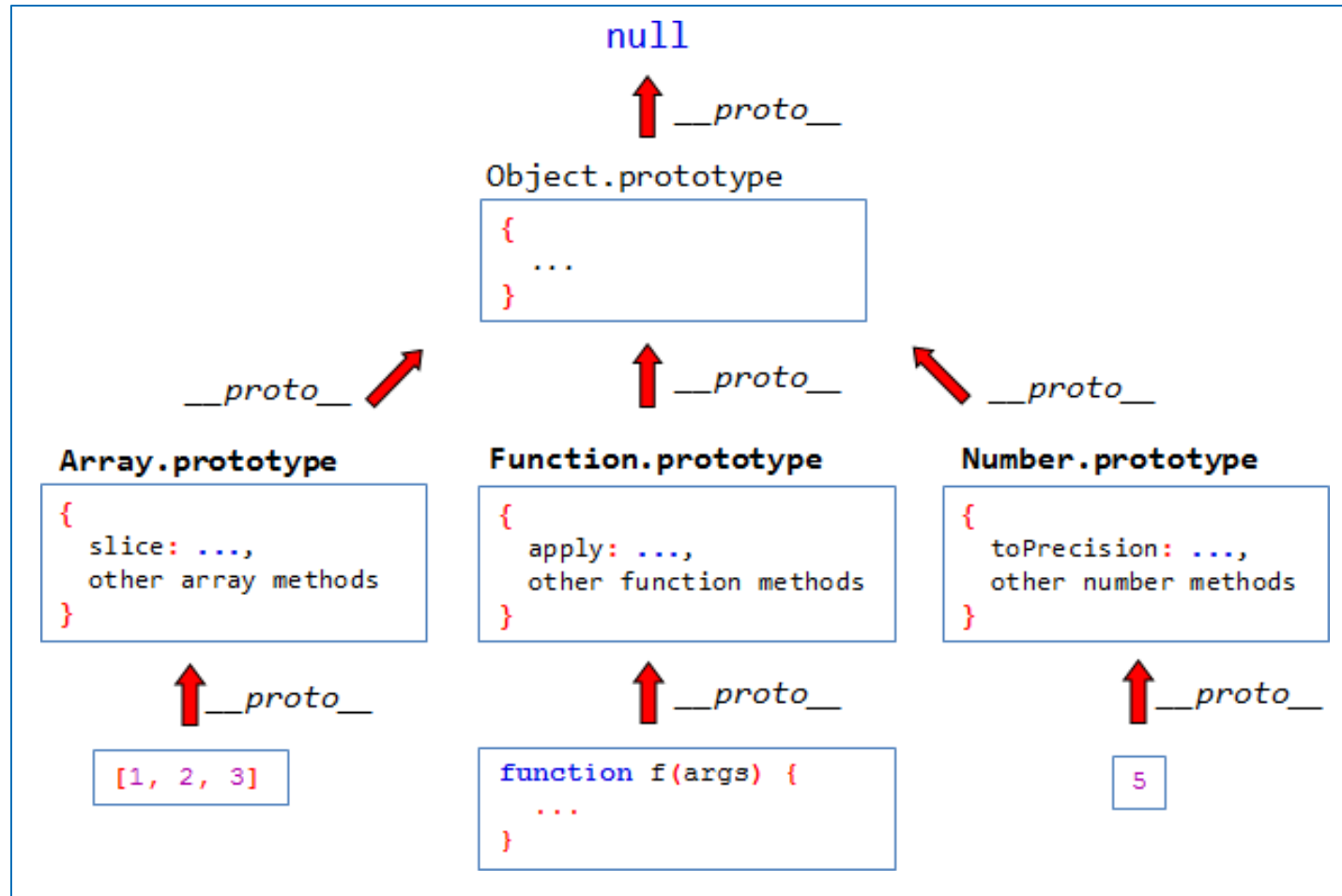
## Internationalization

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()
```

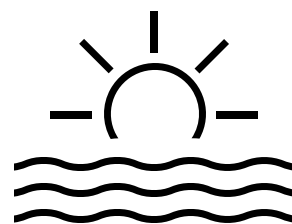
Zie [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

```
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 |                                                                                                                              |                                                                                                                                                                                  |
|----------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Category                                                                         | Class-based (Java)                                                                                                           | Prototype-based (JavaScript)                                                                                                                                                     |
| Class vs. Instance                                                               | Class and instance are distinct entities.                                                                                    | All objects can inherit from another object.                                                                                                                                     |
| Definition                                                                       | Define a class with a class definition; instantiate a class with constructor methods.                                        | Define and create a set of objects with constructor functions.                                                                                                                   |
| Creation of new object                                                           | Create a single object with the <code>new</code> operator.                                                                   | Same.                                                                                                                                                                            |
| Construction of object hierarchy                                                 | 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.                                                                    |
| Inheritance model                                                                | Inherit properties by following the class chain.                                                                             | Inherit properties by following the prototype chain.                                                                                                                             |
| Extension of properties                                                          | 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. |



**HO  
GENT**