

**HO
GENT**

The background of the slide features a photograph of a modern building with a facade made of large, light-colored concrete panels. Some panels are recessed, creating a textured, three-dimensional effect. A woman with long brown hair, wearing a brown jacket, blue jeans, and a red backpack, is walking from left to right in the foreground. She is looking back over her shoulder towards the camera. There are some green plants in the lower left corner.

Webapplicaties II

Hoofdstuk 02 – Objecten en functies

Inhoud

- Objecten
- Functies
- Objecten en functies (methodes)
- Events en event handlers
- Closures
- Zelfstudie: functies in depth

02 Objecten en functies

Objecten

Objecten

JavaScript is designed on a simple object-based paradigm. An object is a **collection of properties**, and a property is an **association between a name (or key) and a value**.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects

Object

- een **object** is een verzameling van **properties**, die het object beschrijven
- een **property**
 - heeft een **naam** en een **waarde**.
 - de **naam, of de key**, van een property is een **string**
 - deze is case sensitive
 - de **waarde** bevat een **primitief datatype of een object**
- meer op https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects

Objects

```
{  
  key : value,  
  ... ,  
  key : value  
}
```

Object

- aanmaken van objecten via **object initializer syntax** (aka literal notation)

- Groepeer properties tussen `{}`
- Scheid properties met `,`
- Scheid naam en waarde van property met `:`

```
// een lege object literal genaamd emptyObject  
const emptyObject = {};
```

```
// een object literal genaamd myAvatar  
// met 4 properties  
const myAvatar = {  
  name: 'Bob',  
  points: 20,  
  gender: 'male',  
  hair: { color: 'black', cut: 'punk' }  
};
```


Object

- een waarde van een property uitlezen

- arraynotatie: objectnaam[naamProperty]

```
const name = myAvatar['name'];
```

- puntnotatie: objectnaam.naamProperty

```
const points = myAvatar.points;
```

- conventie: gebruik waar mogelijk de puntnotatie
 - de puntnotatie is makkelijker te lezen;
 - de arraynotatie heeft als voordeel dat je binnen de haken een variabele kunt gebruiken; als de property naam een spatie of ander niet-identifieer teken bevat dien je ook de array notatie te gebruiken.

- uitlezen van onbestaande eigenschap retourneert undefined

- object chaining

```
const haircut = myAvatar.hair.cut;
```

Object

- een **property toevoegen** aan een object
 - met de arraynotatie

```
myAvatar['age'] = 18;
```

- met de puntnotatie

```
myAvatar.age = 20;
```

Object

- je kan **steeds properties toevoegen aan een object**
- voorbeeld: myAvatar aangemaakt op twee verschillende manieren

```
const myAvatar = {  
  name: 'Bob',  
  points: 20,  
  gender: 'male'  
}
```

```
const myAvatar = {};  
  
myAvatar.name = 'Bob';  
myAvatar.points = 20;  
myAvatar.gender = 'male';
```

Object

- de **waarde van een property** van een object **wijzigen**
 - met de arraynotatie

```
myAvatar['points'] = 50;
```

- met de puntnotatie

```
myAvatar.points = 50;
```

Object

- een **property** van een object **verwijderen**

- met de arraynotatie

```
delete myAvatar['gender'];
```

- met de puntnotatie

```
delete myAvatar.gender;
```

- na de delete heeft myAvatar niet langer een property met de naam gender...

Object

- overlopen van alle properties via **for .. in ..**

```
const myAvatar= {  
  name:'Bob',  
  points:20,  
  gender:'male',  
  hair: {color: 'black', cut: 'punk'}  
};  
console.log(`Hi, I am ${myAvatar.name}`);  
for (let key in myAvatar) {  
  console.log(`-- ${key} : ${myAvatar[key]}`);  
}
```

Merk op: via de array notatie kunnen we aan de waarden van de properties adhv de **variabele** key

```
Hi, I'm Bob  
name : Bob  
points : 20  
gender : male  
hair : [object Object]
```

- alle keys van properties in een **array** stoppen via **Object.keys(o)**

```
const keys = Object.keys(myAvatar);
```

```
> keys  
◀ ▶ (4) ["name", "points", "gender", "hair"]
```

Object

- **object destructuring**

- dit is een krachtige manier om de waarden van properties vast te pakken in variabelen

```
const myAvatar = {  
  name: 'Bob',  
  points: 20,  
  gender: 'male',  
  hair: { color: 'black', cut: 'punk' }  
};
```

```
const name = myAvatar.name;  
const points = myAvatar.points;  
const sex = myAvatar.gender;
```

klassieke manier

```
const { name, points, gender: sex } = myAvatar;
```

*zelfde resultaat via **object destructuring***

merk op dat je de variabele namen niet moet specificeren indien ze overeenkomen met de namen van de properties, bv. name en points

Object

```
const myAvatar = {  
  name: 'Bob',  
  points: 20,  
  gender: 'male',  
  hair: { color: 'black', cut: 'punk' }  
};
```

- object destructuring (vervolg)
 - indien de declaratie van de variabelen en de destructuring gescheiden zijn moet je ronde haakjes gebruiken

```
let name, points;  
({ name, points } = myAvatar);
```

- indien een property niet bestaat krijg je de waarde **undefined**

```
const { gender, weight } = myAvatar;
```

```
> gender  
< "male"  
> weight  
< undefined
```

- je kan **default waarden** voorzien, deze worden gebruikt indien een waarde undefined is

```
const { gender = 'female', weight = '63' } = myAvatar;
```

```
> gender  
< "male"  
> weight  
< "63"
```


Object

- object destructuring (vervolg)
 - je kan genest werken

```
const { name, hair: { color } } = myAvatar;
```

```
> name  
< "Bob"  
> color  
< "black"
```

- je kan object en array destructuring combineren

```
const myAvatar = {  
  name: 'Bob',  
  points: 20,  
  gender: 'male',  
  hair: { color: 'black', cut: 'punk' },  
  features: ['beard', 'sunglasses', 'smile']  
};
```

```
const { name, hair: { color }, features: [first, , third] } = myAvatar;
```

```
> first  
< "beard"  
> third  
< "smile"
```

```
const myAvatar = {  
  name: 'Bob',  
  points: 20,  
  gender: 'male',  
  hair: { color: 'black', cut: 'punk' }  
};
```

Object

- Open de folder 02thVoorbeelden
- Open index.html en zet de script objects.js uit commentaar
- Open bestand objects.js in de folder js
- Maak gebruik van de debugger en de console om de uitvoer te bekijken. Experimenteer zelf met het gegeven script.

Object

- samengevat
 - in JavaScript kan een object gezien worden als een verzameling van properties
 - via de object literal syntax kunnen we een object aanmaken en eventueel enkele properties initialiseren
 - we kunnen steeds properties toevoegen en/of verwijderen.
 - de waarde van een property kan van eender welk type zijn, dus ook wederom een object, dit laat toe dat we complexe datastructuren bouwen
 - properties worden geïdentificeerd aan de hand van een key, dit is een string (of een Symbol, zie later)
 - in een volgend hoofdstuk gaan we zien hoe we met klassen en objecten als instanties van klassen kunnen werken

Oefening

- Oefening 1: [clone 02exObjectsAndFunctions](#)
 - Open 02exObjectsAndFunctions\exercise1\js\objects.js
 - Los op volgens de commentaar
 - Open 02exObjectsAndFunctions\exercise1\ index.html via Live Server om het script in actie te zien...

02 Objecten en functies

Functies

Functies

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure—a set of statements that performs a task or calculates a value. To use a function, you must define it somewhere in the scope from which you wish to call it.

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions>

Functies

- de **functie declaratie**

- **function** keyword
- **naam** van de functie
- **lijst van parameters** voor de functie
 - tussen ronde haakjes, gescheiden door komma's
- **Javascript statements**
 - de function body, tussen accolades

```
function functionname (par1, par2, ... , parn) {  
    some code goes here..  
}
```

- voorbeeld

```
function sayHi(name) {  
    return `Hi, my name is ${name}`;  
}
```

Functies

- het **aanroepen van een functie** zorgt dat de statements van de functie, eventueel gebruik makend van de parameters, worden uitgevoerd
 - gebruik de naam van de functie gevolgd door de argumenten tussen ronde haakjes
- voorbeeld

```
function sayHi(name) {  
  return `Hi, my name is ${name}`;  
}
```

```
console.log(sayHi('Ann'));
```

Hi, my name is Ann

Functies

- een **return statement** stopt verdere uitvoering van een de function body en retourneert een waarde naar de aanroeper van de functie
 - **return** kan overal geplaatst worden in de function body
 - **return** kan meerdere malen in de function body voorkomen
 - indien het einde van de function body bereikt wordt **zonder return statement**, dan retourneert de functie **undefined**

```
function sayHi(name) {  
  // John never says Hi...  
  if (name !== 'John') return `Hi, my name is ${name}`  
;  
}
```

```
const message1 = sayHi('Ann');  
const message2 = sayHi('John');
```

```
> message1  
◀ "Hi, my name is Ann"  
> message2  
◀ undefined
```

Functies

- veranderingen in de function body aan parameters van een **primitief type** zijn niet zichtbaar in de scope van de aanroeper
- veranderingen aan de properties van een **object type** parameter zijn wel zichtbaar in de scope van de aanroeper

```
const myAvatar = {  
  name: 'Bob',  
  points: 20,  
  gender: 'male',  
  hair: { color: 'black', cut: 'punk' }  
};  
  
function dyeHair/avatar, color) {  
  avatar.hair.color = color;  
  color = 'pink';  
}  
const newColor = 'Red';  
console.log('Before dyeing hair:');  
console.log(`newColor = ${newColor}`);  
console.log(`myAvatar.hair.color = ${myAvatar.hair.color}`);  
dyeHair(myAvatar, newColor);  
console.log('After dyeing hair:');  
console.log(`newColor = ${newColor}`);  
console.log(`myAvatar.hair.color = ${myAvatar.hair.color}`);
```

Before dyeing hair:
newColor = Red
myAvatar.hair.color = black
After dyeing hair:
newColor = Red
myAvatar.hair.color = Red

Functies

- wanneer geen argumenten aangeleverd worden voor een of meerdere parameters dan zullen deze de waarde **undefined** aannemen

```
function dyeHair(avatar, color) {  
  avatar.hair.color = color;  
}  
  
dyeHair(myAvatar);  
console.log(`myAvatar.hair.color = ${myAvatar.hair.color}`);
```

myAvatar.hair.color = undefined

- je kan **default waarden** voor parameters voorzien, deze worden dan gebruikt voor parameters waarvoor geen argumenten aangeleverd worden

```
function dyeHair(avatar, color = 'green') {  
  avatar.hair.color = color;  
}  
  
dyeHair(myAvatar);  
console.log(`myAvatar.hair.color = ${myAvatar.hair.color}`);
```

myAvatar.hair.color = green

Functies

- rest parameter
 - via de rest parameter syntax kan je een onbeperkt aantal parameters voorstellen als een array
 - je doet dit door de laatste parameter in de parameterlijst te laten voorafgaan door ...

```
function f(a, b, ...otherArgs) {  
  for (let value of otherArgs) {  
    console.log(value);  
  }  
}
```

```
f(1, 2, 3, 'Four', 5);
```

3
Four
5

*otherArgs is een **Array** object*

Functies

- rest parameter vervolg
 - je kan gebruik maken van array destructuring in de parameterlijst

```
function f(a, b, ...[c, d, e, f])  
{  
  console.log(a);  
  console.log(b);  
  console.log(c);  
  console.log(d);  
  console.log(e);  
  console.log(f);  
}
```

```
f(1, 2, 3, 'Four', 5);
```

1
2
3
Four
5
undefined

Functies

- functie declaraties worden **gehoist**
 - door de hoisting tijdens de compilatiefase wordt de functie declaratie als het ware naar boven verplaatst
 - de functie kan **overall** aangeroepen worden, ook vóór de declaratie van de functie

```
dyeHair(myAvatar, 'yellow');
```

```
function dyeHair(avatar, color = 'green') {  
  avatar.hair.color = color;  
}
```

```
console.log(`myAvatar.hair.color = ${myAvatar.hair.color}`);
```

```
myAvatar.hair.color = yellow
```

Functies

- Javascript **functies zijn** first class **objects**
 - javascript is een functionele taal
 - een functie is een object
 - het kan worden toegekend aan een variabele
 - het kan de waarde zijn van een property van een object
 - het kan doorgegeven worden als argument naar een functies
 - het kan een returnwaarde zijn van een functie
 - het kan aangemaakt worden 'at run-time'.

Functies

- via een **functie expressie** kunnen we **een functie toekennen aan een variabele**
 - de functie zelf **kan anoniem** zijn, in dat geval heeft ze geen naam

```
const dyeHair = function (avatar, color = 'green') {  
  avatar.hair.color = color;  
}  
  
dyeHair(myAvatar, 'yellow');  
  
console.log(`myAvatar.hair.color = ${myAvatar.hair.color}`);
```

*dyeHair is een **variabele**, de waarde van deze variabele is een anonieme functie door de variabele te gebruiken in combinatie met ronde haakjes kan de functie aangeroepen worden*

```
> dyeHair  
◀ f (avatar, color = 'green') {  
  avatar.hair.color = color;  
}
```


Functies

- merk op dat de hoisting nu volgens de regels van variabelen plaatsvindt
 - mogelijk is er een temporal dead zone (zie h1)

```
dyeHair(myAvatar, 'yellow');
```

✖ ▶ Uncaught ReferenceError: dyeHair is not defined

```
const dyeHair = function(avatar, color = 'green') {  
  avatar.hair.color = color;  
};
```

```
console.log(`myAvatar.hair.color = ${myAvatar.hair.color}`);
```

*door de hoisting tijdens de compilatiefase wordt de declaratie van de const dyeHair naar boven geplaatst
maar dyeHair kan niet gebruikt worden zolang de toekenning niet werd gepasseerd*

functie declaratie: de functie wordt gehoist naar boven en kan steeds overal gebruikt worden

functie expressie: de variable die de naar de functie verwijst wordt gehoist

Functies

- voorbeeld: een functie doorgeven als parameter

```
const isPass = function(mark) {  
  return mark >= 10;  
};  
  
const giveFeedback = function(mark) {  
  return mark < 10 ? 'Disastruous' : mark < 12 ? 'Mediocre' :  
  'Very good';  
};  
  
const convertMark = function(converter, number) {  
  return converter(number);  
};  
  
console.log(convertMark(isPass, 11));  
console.log(convertMark(giveFeedback, 11));
```

true

Mediocre

*de eerste parameter van convertMark krijgt bij aanroep een functie aangeleverd
in de body van convertMark wordt de aangeleverde functie aangeroepen*

Functies

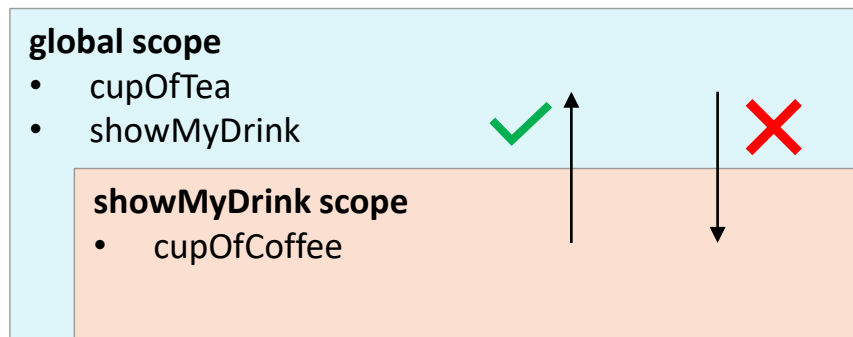
- Scope van variabelen
 - variabelen, en de parameters, gedeclareerd in een functie, zijn enkel binnen de functie toegankelijk
 - een functie creëert een **local scope**
 - de local scope bevat **lokale variabelen**
 - **scope chaining**: naast lokale variabelen heeft een functie ook toegang tot variabelen zichtbaar binnen de scope waarin de functie werd gedeclareerd
 - indien de functie in de global scope werd gedeclareerd heeft de functie toegang tot alle variabelen uit de global scope
 - indien een functie genest is binnen een andere functie heeft deze functie toegang tot zijn lokale variabelen, de variabelen die gedeclareerd werden in de parent functie, en alle variabelen waar de parent functie ook toegang tot heeft
 - indien in een scope twee variabelen met dezelfde naam toegankelijk zijn, dan wordt de meest lokale variabele gebruikt

Functies

- Scope van variabelen – voorbeeld 1

```
let cupOfTea = 'Camomille';
const showMyDrink = function() {
  const cupOfCoffee = 'Cappuccino';
  cupOfTea = 'Peppermint';
  console.log(`I like ${cupOfCoffee} and ${cupOfTea}`);
};
console.log(`I like ${cupOfTea}`);
showMyDrink();
console.log(`I like ${cupOfTea}`);
```

I like Camomille
I like Cappuccino and Peppermint
I like Peppermint



cupOfTea is geen onderdeel van de function scope showMyDrink, wanneer in de functie cupOfTea gebruikt wordt, dan zal dit de variabele uit de global scope zijn

vanuit de global scope is het onmogelijk om de variabele cupOfCoffee te gebruiken

Functies

- Scope van variabelen – voorbeeld 2

```
let cupOfTea = 'Camomille';
const showMyDrink = function() {
  const cupOfCoffee = 'Cappuccino';
  const cupOfTea = 'Peppermint';
  console.log(`I like ${cupOfCoffee} and ${cupOfTea}`);
};
console.log(`I like ${cupOfTea}`);
showMyDrink();
console.log(`I like ${cupOfTea}`);
```

```
I like Camomille
I like Cappuccino and Peppermint
I like Camomille
```

global scope

- cupOfTea
- showMyDrink

showMyDrink scope

- cupOfCoffee
- cupOfTea

cupOfTea is nu onderdeel van de function scope showMyDrink, wanneer in de functie cupOfTea gebruikt wordt, dan zal dit de lokale variabele uit de function scope zijn, we zeggen dat deze cupOfTea, de globale cupOfTea overschaduw

Functies

- Scope van variabelen – voorbeeld 3

```
let cupOfTea = 'Camomille';
const showMyDrink = function(cupOfTea) {
  const cupOfCoffee = 'Cappuccino';
  cupOfTea = 'Peppermint';
  console.log(`I like ${cupOfCoffee} and ${cupOfTea}`);
};
console.log(`I like ${cupOfTea}`);
showMyDrink();
console.log(`I like ${cupOfTea}`);
```

I like Camomille
I like Cappuccino and Peppermint
I like Camomille

global scope

- cupOfTea
- showMyDrink

showMyDrink scope

- cupOfCoffee
- cupOfTea

cupOfTea is een parameter van de function showMyDrink, je kan het beschouwen als een lokale variabele van showMyDrink

Functies

- Scope van variabelen – a taste of all...

```
let globalX = 'global X';
let cupOfTea = 'Camomille';
let outerFunction = function(param1, param2) {
  let outerX = 'outer X';
  let cupOfTea = 'Fennel';
  let cupOfCoffee = "Cappuccino";
  let innerFunction = function(param1, param3) {
    let innerX = 'inner X';
    let cupOfCoffee = "Espresso";
  };
};
```

je maakt steeds gebruik van de meest lokale variabele, bv. `cupOfCoffee` in `innerFunction`
wanneer in een scope een variabele wordt gebruikt die daar niet gedeclareerd is, dan wordt er in de omringende scope gekeken: bv. `cupOfTea` gebruiken in `innerFunction`,
dit proces herhaalt zich tot je de global scope bereikt, bv. `globalX` gebruiken in `innerFunction`

global scope

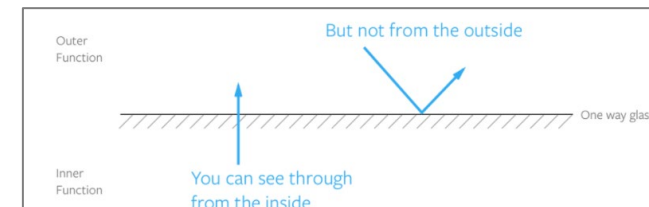
- `globalX`
- `cupOfTea`
- `outerFunction`

outerFunction scope

- `param1`
- `param2`
- `outerX`
- `cupOfTea` // hides `cupOfTea` from global scope
- `cupOfCoffee`
- `innerFunction`

innerFunction scope

- `param1` // hides `param1` of `outerFunction`
- `param3`
- `innerX`
- `cupOfCoffee`



Functies

- Let op voor temporal dead zones

```
let cupOfTea = 'Camomille';  
const showMyDrink = function() {  
  const cupOfCoffee = 'Cappuccino';  
  console.log(`I like ${cupOfCoffee} and ${cupOfTea}`);  
  const cupOfTea = 'Peppermint';  
};  
showMyDrink(cupOfTea);
```

✖ ▶ Uncaught ReferenceError: cupOfTea is not defined



Declareer locale variabelen steeds bovenaan de functie!

Functies

- **arrow functions**

- een **compactere syntax** om functie expressies te schrijven
- zijn altijd **anoniem**

```
const isPass = function(mark) {  
    return mark >= 10;  
};
```



```
const isPass = mark => mark >= 10;
```

*zelfde functie met arrow
notatie*

```
console.log(isPass(10));
```

de functie aanroep blijft dezelfde

Functies

- arrow functions
 - voorbeeld: doorgeven van een functie aan een andere functie

```
const convertMark = function(converter, number) {  
  return converter(number);  
};  
  
console.log(convertMark(mark => mark >= 10, 50));
```

Functies

- arrow functions – algemene syntax

`(par1, par2, ..., parn) => {statements}`

- als er geen parameters zijn, gebruik ()
- als er maar 1 parameter is, () mag je weglaten
- als de code enkel de return van een expressie is, {} en return weglaten

```
const function1 = (a, b, c) => {  
  a = a + 1;  
  return a + b + c;  
};  
  
const function2 = (a, b, c) => a + 1 + b + c;  
  
const function3 = () => Math.random() * 20;
```

Functies

- aandachtspunten
 - via een functie expressie ken je een waarde toe aan een variabele
 - de waarde van de variabele is de functie (~functie object)
 - wanneer je de variabele gebruikt zonder () refereer je naar de waarde van de variabele, i.e. de functie, en roep je de functie niet aan!

```
const isPass1 = function pass(mark) {  
  return mark >= 10;  
};  
  
const isPass2 = mark => mark >= 10;  
  
console.log(isPass1);  
console.log(`isPass1(16): ${isPass1(16)}`);  
  
console.log(isPass2);  
console.log(`isPass2(16): ${isPass2(16)}`);
```

```
f pass(mark) {  
  return mark >= 10;  
}  
isPass1(16): true  
mark => mark >= 10  
isPass2(16): true
```

Functies

- aandachtspunten vervolg
 - je kan via toekenning de functie toekennen aan een ander variabele

```
let isPass = mark => mark >= 10;
const isBiggerThanNine = isPass;
console.log(`isPass: ${isPass}`);
console.log(`isBiggerThanNine: ${isBiggerThanNine}`);
console.log(isPass === isBiggerThanNine);
console.log(`isPass(20): ${isPass(20)}`);
console.log(`isBiggerThanNine(20): ${isBiggerThanNine(20)}`);
isPass = mark => mark >= 50;
console.log(`isPass: ${isPass}`);
console.log(`isBiggerThanNine: ${isBiggerThanNine}`);
console.log(isPass === isBiggerThanNine);
console.log(`isPass(20): ${isPass(20)}`);
console.log(`isBiggerThanNine(20): ${isBiggerThanNine(20)}`);
```

```
isPass: mark => mark >= 10
isBiggerThanNine: mark => mark >= 10
true
isPass(20): true
isBiggerThanNine(20): true
isPass: mark => mark >= 50
isBiggerThanNine: mark => mark >= 10
false
isPass(20): false
isBiggerThanNine(20): true
```

Functies

- Zie functions.js in 02thVoorbeelden
- Tip
 - maak gebruik van <ctrl><k> - <ctrl><u> om een geselecteerd blok uit commentaar te halen
 - maak gebruik van <ctrl><k> - <ctrl><c> om een geselecteerd blok in commentaar te zetten
- **Oefening**
 - zie functions.js in 02exObjectsAndFunctions.js – exercise1

02 Objecten en functies

Objecten en functies

Objecten en functies

a **method** is a function which is
a property of an object.

□ **Note:** In JavaScript functions themselves are objects, so, in that context, a method is actually an object reference to a function.



Objecten en functies

- een **methode** van een object is eenvoudigweg een **property met als waarde een functie**.
- een methode van een object kan worden aangeroepen met de punt- of de array-notatie

```
const myAvatar = {  
  name: 'Bob',  
  points: 20,  
  gender: 'male',  
  hair: { color: 'black', cut: 'punk' },  
  sayHi: function() {  
    const title = this.gender === 'male' ? 'Sir' : 'Miss';  
    return `Hi, I am ${title} ${this.name}`;  
  }  
};  
console.log(myAvatar.sayHi());
```

Hi, I am Sir Bob

Objecten en functies

- aandachtspunten
 - in de praktijk wordt er meestal gesproken over ‘properties en methodes’, alsof een methode iets anders is dan een property, nu weet jij wel beter...
 - **this**
 - binnen de body van een functie is de **this** variabele beschikbaar. **The this keyword, passed to all functions, is a reference to the object that contains the function.**
 - als de function gedefinieerd is in de **global scope** verwijst this naar het global object (in de browser is dat **window**)
 - als this gebruikt wordt in een **methode van een object** verkrijg je een verwijzing naar het parent object waar de methode (property) toe hoort.
 - **arrow functions** beschikken niet over hun ‘eigen’ this, en dit maakt ze **ongeschikt om als methodes gebruikt te worden!**

Objecten en functies

- **voorbeeld:** een methode die geen gebruik maakt van **this**

```
const myAvatar = {  
  name: 'Bob',  
  points: 20,  
  gender: 'male',  
  hair: { color: 'black', cut: 'punk' },  
  sayHi: function() {  
    const title = gender === 'male' ? 'Sir' : 'Miss';  
    return `Hi, I am ${title} ${this.name}`;  
  }  
};  
console.log(myAvatar.sayHi());
```



✖ ▶ Uncaught ReferenceError: gender is not defined



Gebruik steeds **this** om binnen een object naar één van zijn properties of methods te verwijzen!

Objecten en functies

- **voorbeeld:** een methode gedefinieerd als een arrow functie

```
const myAvatar = {  
  name: 'Bob',  
  points: 20,  
  gender: 'male',  
  hair: { color: 'black', cut: 'punk' },  
  sayHi: () => {  
    const title = this.gender === 'male' ? 'Sir' : 'Miss';  
    return `Hi, I am ${title} ${this.name}`;  
  }  
};  
console.log(myAvatar.sayHi());
```

Hi, I am Miss



Gebruik geen arrow functies voor methodes!

Objecten en functies

- voor- en nadelen een **object literal**
 - **eenvoudig** instantiatie en gebruik van een object
 - geen mogelijkheid om meerdere instanties van eenzelfde **type** te maken
 - geen afscherming van private en publieke properties/functies. Alle code in object is beschikbaar voor de buitenwereld. Ze kunnen worden uitgelezen EN aangepast. Toch bestaat er een conventie om de keynamen van lokale variabelen en methodes binnen een object met een _ te laten beginnen

Objecten en functies

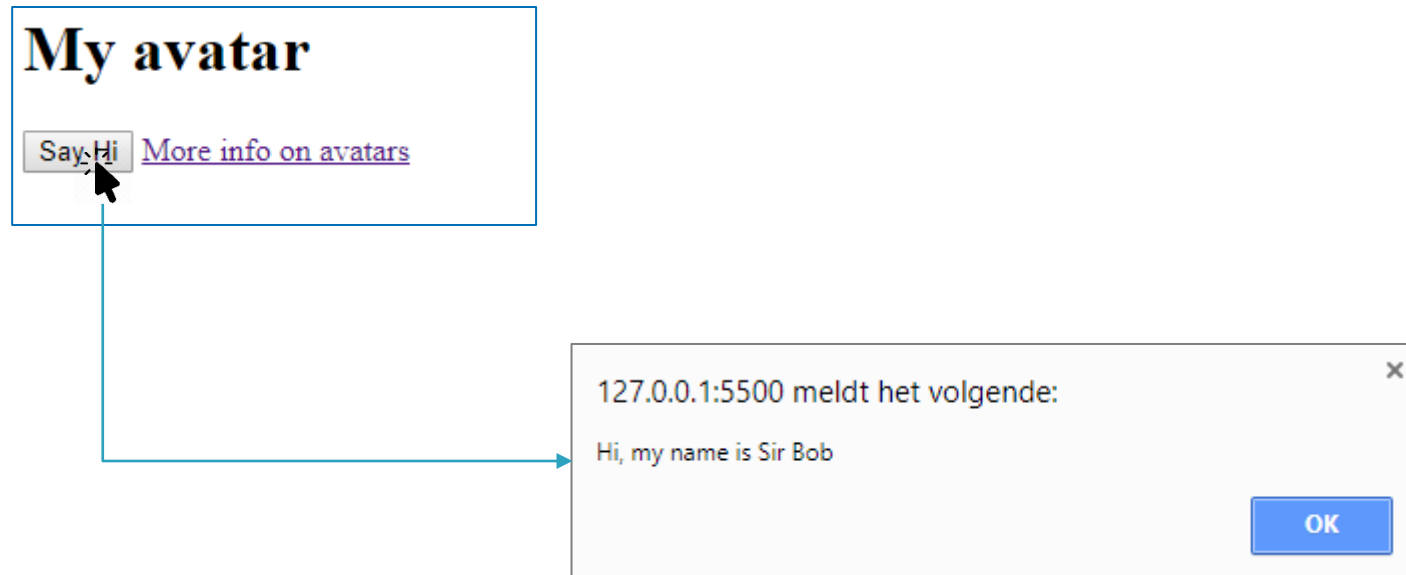
- voorbeelden
 - zie `objectsAndFunctions.js` in `02thVoorbeelden`
- **oefening**
 - zie `objectsAndFunctions.js` in `02exObjectsAndFunctions.js` – `exercise1`

02 Objecten en functies

Events

Events

- Laat ons wat leven in onze pagina's blazen: event afhandeling!



Events


- een gebruiker interageert met een webpagina via een **user interface**
- een handeling op een HTML element zal resulteren in het afvuren van één of meerdere **events**
- wanneer een event wordt afgevuurd, dan wordt de bijhorende **event handler** aangeroepen.
 - een event handler is een **functie**
 - wij kunnen voor de verschillende events, event handlers definiëren en de pagina laten reageren naar onze wensen

Events

- als je javascript wordt uitgevoerd **in de browser** heb je toegang tot een **Window object**

```
console.log(this);
```

een simpel one-line script...



```
▼ Window {postMessage: f, blur: f, focus: f, close: f, frames: Window, ...} ⓘ  
  ▶ alert: f alert()  
  ▶ applicationCache: ApplicationCache {status: 0, onchecking: null, onerror: null, onond  
  ▶ atob: f atob()  
  ▶ blur: f ()  
  ▶ btoa: f btoa()  
  ▶ caches: CacheStorage {}  
  ▶ cancelAnimationFrame: f cancelAnimationFrame()  
  ▶ cancelIdleCallback: f cancelIdleCallback()  
  ▶ captureEvents: f captureEvents()  
  ▶ chrome: {loadTimes: f, csi: f}  
  ▶ clearInterval: f clearInterval()  
  ▶ clearTimeout: f clearTimeout()  
  ▶ clientInformation: Navigator {vendorSub: "", productSub: "20030107", vendor: "Google  
  ▶ close: f ()  
    closed: false  
  ▶ confirm: f confirm()  
  ▶ createImageBitmap: f createImageBitmap()
```

- dit is een object representatie van het venster waarin je webpagina is geopend
- het Window object omvat de **global scope**.

Events

- het **document object is een property** van het window object.
- dit document object heeft properties die **object representaties van de HTML pagina** bevatten
 - DOM zie verder hfdstk

elk **HTML element** van een pagina komt overeen met een **javascript object**

de **attributen** van de HTML elementen zijn **properties** van dit object

Events

- voorbeeld
 - gebruik de methode **getElementById** om een object op te halen die overeenkomt met de html tag met dat id

```
const sayHiButton = document.getElementById('sayHi');
```

HTML met een input-element met id = 'sayHi'

```
<body>
<h1>My avatar</h1>
<input type="button" id="sayHi" value="Say Hi" />
...
<script src="js/events.js"></script>
</body>
```

een javascript object sayHiButton

```
▼ sayHiButton: input#sayHi
  accept: ""
  accessKey: ""
  align: ""
  alt: ""
  assignedSlot: null
  ► attributes: NamedNodeMap {0: type, 1: id, ...}
  autocapitalize: "none"
  autocomplete: ""
  autofocus: false
  baseURI: "http://127.0.0.1:5500/"
  checked: false
  childElementCount: 0
  ► childNodes: NodeList []
```

Events

- voorbeeld vervolg
 - wanneer we in het javascript object een **property wijzigen**, dan wordt het overeenkomstig HTML element aangepast

```
sayHiButton.value = 'Yebo!';
```

My avatar

Say Hi [More info on avatars](#)

My avatar

Yebo! [More info on avatars](#)

Events

- in de DOM worden tal van **events** getriggered
 - user generated, bv. keyboard/mouse events
 - API generated, bv. animation finished running
- een **event** heeft
 - **een naam**
 - bv. click, change, load, mouseover, keyup, focus,...
 - **een target**
 - het element waarop het event van toepassing is, bv de button met id sayHi

Events

- als een object moet reageren op een bepaalde event, voorzie je een **eventhandler** of **callback functie**

A callback is a piece of executable code that is **passed as an argument to other code, which is expected to call back (*execute*) the argument at some convenient time**

— [Wikipedia](#)

- wanneer een event getriggered wordt, zal de bijhorende event handler uitgevoerd worden
- je kent hiervoor de event handler toe aan de juiste methode (d.i. een property) van het object
 - de naam van deze property is '**on**' gevolgd door de **naam van het event**
 - bv. onclick, onload, onmouseenter, ondblclick

*merk op:
geen camel casing!*

Events

- voorbeeld

```
const myAvatar = {  
  name: 'Bob',  
  points: 20,  
  gender: 'male',  
  hair: { color: 'black', cut: 'punk' },  
  sayHi: function() {  
    const title = this.gender === 'male' ? 'Sir' : 'Miss';  
    return `Hi, I am ${title} ${this.name}`;  
  }  
};  
  
const sayHiButton = document.getElementById('sayHi');  
  
sayHiButton.onclick = function() {  
  alert(myAvatar.sayHi());  
};
```

*wanneer op de button met id 'sayHi' wordt geklikt, wordt het **click event** getriggert, de functie die werd toegekend aan de **onclick property** zal dan worden uitgevoerd*

Events

- scheiding HTML en javascript
 - je zou de onclick event handler ook in de HTML code kunnen definieren

```
<input type="button" id="sayHi" value="Say Hi"  
onclick="alert(myAvatar.sayHi());"/>
```



*we willen de opmaak (HTML) en het
gedrag (Javascript) gescheiden houden!*



Event afhandeling wordt beschreven in de scripts
die in aparte bestandjes zitten

Events

- we moeten er voor zorgen dat we de eventhandlers instellen op een moment wanneer het volledige document is geladen
 - bv. je zal geen onclick event handler kunnen instellen voor een button als die button nog niet beschikbaar is
- we kunnen dit doen door te reageren op het **load event** van **window**

The **load** event fires at the end of the document loading process. At this point, all of the objects in the document are in the DOM, and all the images, scripts, links and sub-frames have finished loading.

```
window.onload = function() {  
    // hier kunnen we de eventhandlers instellen  
};
```

Events

- voorbeeld

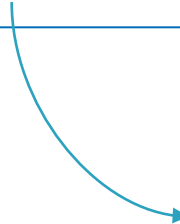
```
const myAvatar = {  
  name: 'Bob',  
  points: 20,  
  gender: 'male',  
  hair: { color: 'black', cut: 'punk' },  
  sayHi: function() {  
    const title = this.gender === 'male' ? 'Sir' : 'Miss';  
    return `Hi, I am ${title} ${this.name}`;  
  }  
};  
  
const sayHiClicked = function() {  
  alert(myAvatar.sayHi());  
};  
  
const init = function() {  
  const sayHiButton = document.getElementById('sayHi');  
  sayHiButton.onclick = sayHiClicked;  
};  
  
window.onload = init;
```

in dit voorbeeld worden de functies als variabelen (constanten) gedefinieerd, en worden de variabelen gebruikt om de event handlers in te stellen

Events

- **this** keyword
 - het **this keyword binnen een event handler** verwijst naar het object die het event heeft getriggered

```
const sayHiClicked = function() {  
  alert(myAvatar.sayHi());  
  alert(`this: ${this} & this.id: ${this.id}`);  
};
```



127.0.0.1:5500 meldt het volgende

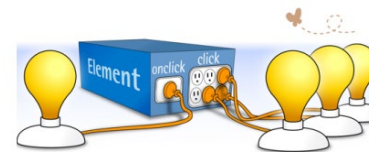
this: [object HTMLInputElement] & this.id: sayHi

OK

Events

- je kan ook gebruik maken van de methode **addEventListener** voor het instellen van een eventhandler.
 - op deze manier kunnen meerdere event handlers ingesteld worden voor één en hetzelfde event

```
const sayHiClicked = function() {  
  alert(myAvatar.sayHi());  
};  
  
const beClever = function() {  
  alert(`I know you clicked on button that says ${this.value}`);  
};  
  
const init = function() {  
  const sayHiButton = document.getElementById('sayHi');  
  sayHiButton.addEventListener('click', sayHiClicked);  
  sayHiButton.addEventListener('click', beClever);  
};  
  
window.onload = init;
```



Events

- voorbeelden
 - zie events.js in 02thVoorbeelden
- **oefening**
 - zie events.js in 02exObjectsAndFunctions.js – exercise1

02 Objecten en functies

Closures

Functions in depth: Closures

- “A **closure** is a function, whose return value depends on the value of one or more variables declared outside this function. **The function defined in the closure 'remembers' the environment in which it was created**”
- als we een functie retourneren die een waarde van een variabele gebruikt die buiten deze functie gedefinieerd werd, dan wordt de waarde van de variabele tussen functieaanroepen door opgeslagen. De ‘inner functie’ die gebruik maakt van deze variabele creëert een closure.

Closures

- closures worden gecreëerd via **inner (geneste) functions**

```
const makeFunc = function() {  
  const message = 'Sir Bob is still around ;-);  
  const displayName = function() {  
    console.log(message);  
  };  
  return displayName;  
};  
  
const aDisplayFunction = makeFunc();  
  
console.log(aDisplayFunction());
```

- **displayName (de inner function)** is een functie binnen de functie **makeFunc (de outer function)**
- displayName maakt, zoals de scope regels toelaten, gebruik van **message, een lokale variabele** van de outer function makeFunc
- de **outer function makeFunc retourneert de inner function** displayName
- deze returnwaarde, functie, kunnen we vastnemen in een variabele en **aanroepen**
- via **de closure werd de variabele message en zijn waarde bijgehouden**, het is deel van de functie die makeFunc retourneerde
- de functie reageert zoals verwacht

Closures

- De closure kan je bekijken in Chrome developer tools

The screenshot shows the Chrome Developer Tools interface. On the left, the source code editor displays a JavaScript function `makeFunc` that returns a closure `displayName`. The closure has access to the `message` variable from the `makeFunc` function's scope. The line `console.log(message);` inside the closure is highlighted with a blue breakpoint. On the right, the 'Paused on breakpoint' panel is open, showing the current execution context. The 'Call Stack' shows the `displayName` function call. The 'Scope' panel shows the local scope with `this: Window` and the closure scope with `message: 'Sir Bob is still around ;-).'`. The 'Breakpoints' panel shows a breakpoint set at `closures.js:4` on the line `console.log(message);`.

```
const makeFunc = function() {  
  const message = 'Sir Bob is still around ;-);'  
  const displayName = function() {  
    console.log(message);  
  };  
  return displayName;  
};  
  
const aDisplayFunction = makeFunc();  
  
console.log(aDisplayFunction());  
|
```

Paused on breakpoint

- ▶ Watch
- ▼ Call Stack
 - ▶ displayName
 - (anonymous)
- ▼ Scope
 - ▼ Local
 - ▶ this: Window
 - ▼ Closure (makeFunc)
 - message: "Sir Bob is still around ;-)"
 - ▶ Script
 - ▶ Global
- ▼ Breakpoints
 - ☒ closures.js:4
console.log(message);

Closures

- ook parameters van de outer functie kunnen deel uitmaken van een closure

```
const makeFunc = function(title) {  
  const message = `${title} Bob is still around ;-)`;  
  const displayName = function() {  
    console.log(message);  
  };  
  return displayName;  
};  
const misterDisplayFunction = makeFunc('Mister');  
const sirDisplayFunction = makeFunc('Sir');  
console.log(misterDisplayFunction());  
console.log(sirDisplayFunction());
```

Mister Bob is still around ;-)
undefined
Sir Bob is still around ;-)
undefined

de makeFunc is nu als het ware een function factory, in dit voorbeeld worden twee verschillende functies aangemaakt en gebruikt...



Exceptions

- throw een exception igv ongebruikelijke fout
 - throw: gooit een exception object
- opvangen van exception try ... catch ... finally

```
try {  
  throw {  
    name: 'SomethingWentWrongError',  
    message: 'Something went wrong. You should fix it'  
  };  
} catch (e) {  
  // handle the exception here  
  console.log(`Error ${e.name}: ${e.message}`);  
} finally {  
  // this is executed even if an exception occurs  
}
```

Error SomethingWentWrongError: Something went wrong. You should fix it

**HO
GENT**