



JS basics

# Inhoud - Javascript

---

Variabelen gebruiken:

Soorten en naamgeving

Functies

Controlestructuren

Debugging in JS

DOM

Wat is het?

Hoe te gebruiken?

# Variabelen

# Variabelen in JS: werking

Kan **vanelles** bevatten:

int, string, functie, object, array, ...

Hoeven niet te weten wat er in zal komen  
**(loosely typed)**

Je kunt om het even wat zetten in de variable tijdens de uitvoering.

*Bv.: Eerst bevat een variabele een string, dan kan je hier later ook een integer in zetten.*



# Variabelen in JS: soorten

Er zijn verschillende manieren om een variabele aan te maken:

1. var
2. const
3. let

Met deze keywords initialiseren we een variabele (eenmalig).

Straks bij het bekijken van functies komen we hierop terug !



# Variabelen in JS: naamgeving

---

We hanteren camelCase om variabelen een naam te geven.

```
let web_developer;      *
let web-developer;     **
let WebDeveloper;
let webDeveloper;
```

- \* Een '\_' is voor SQL syntax.
- \*\* Een '-' kan als **min-bewerking** gezien worden, dus dat willen we vermijden.  
'-' wordt ook al gebruikt in **HTML data-attributen** en **CSS properties**.

# Variabelen in JS: naamgeving

---

Begin een variabele niet met een:

- **Getal**
- **\$ teken** (Veel frameworks gebruiken dit al).

# Variabelen in JS: naamgeving

---

Schrijf een string met enkele quote '.

```
let question = "How you doing 😊?";      *  
let question = 'How you doing 😊?';      *  
let question = 'How y\all doing 😊?';    *
```

De **dubbele quote** is al voor **HTML-attributen**.

**Escape** een enkele quote met backslash \.

\* <https://www.youtube.com/watch?v=dc2Z7CL4Cv0>

# Functies()

Actie!

# Wat is een functie?

---

**Een stuk code dat precies 1 iets doet.**

Kan je op verschillende manieren aanroepen en opbouwen.

Rechts staat 2 keer dezelfde functie:

Beide zijn correct.

Bij de laatste staat de naam duidelijk vooraan (grotere projecten).

Ps: een functie krijgt parameters mee tussen de ronde haakjes.



```
function calculateSurface(a, b) {  
    return a * b;  
}
```



```
const calculateSurface =  
function(a, b) {  
    return a * b;  
}
```

# Wat is een functie?

---

**Een stuk code wat precies 1 iets doet.**



```
function calculateSurface(a, b) {  
    return a * b;  
}
```



```
const calculateSurface =  
function(a, b) {  
    return a * b;  
}
```

Om side-effects te vermijden, krijgt een functie een omgeving om te werken.

Dat is de functie-scope.

Daardoor kan je binnen een functie berekenen, data verwerken etc. zonder bang te zijn iets anders aan te passen.

Dit komt neer op de curly brackets: { ... }:

# Throwback

---

Er zijn verschillende soorten variabelen:

1. var
2. const
3. let

*Straks bij het bekijken van functies komen we hierop terug !*

👉 Nu dus

```
var userInput = 12;  
// Een variabele met een integer: 12.  
  
const userInput = 12;  
// Ook een variabele met een integer: 12.  
  
let userInput = 12;  
// Je raadt het nooit...
```

# Variabelen: overschrijven: let & var vs. const

---

Q: Wat zijn nu **de verschillen**?

A: Het toestaan van:

1. **het overschrijven** van een variabele.
2. **een variabele** in een (andere) **function-scope**.

```
var userInput = 12;  
// Een variabele met een integer: 12.  
  
const userInput = 12;  
// Ook een variabele met een integer: 12.  
  
let userInput = 12;  
// Je raadt het nooit...
```

We bekijken die 2 verschillen:

*Want in deze code kan je dat niet zien.*

# Variabelen: overschrijven: let & var vs. const

---

Bekijk volgende code.

We gaan dit nu runnen (bv. In Chrome).

Er wordt een var en een let geïnitialiseerd en gedeclareerd met een string-waarde.

Vervolgens overschrijven we de values.

We printen de waardes af in de console.

We doen hetzelfde met een const waarde:

```
let bestTrack = "Smart Tech";
var theFuture = "Native";
const bestTrack = "Web App Developer";

bestTrack = "Web App Developer";
theFuture = "The web";
bestTrack = "Smart Tech";
```

✖ ▶ Uncaught SyntaxError: Identifier 'bestTrack' has already been declared  
at <anonymous>:1:1

# Variabelen: een (andere) function-scope

---

Bekijk volgende code.

Let op de { } die een scope (afscherming) maken.

We gaan dit nu runnen.

We doen hetzelfde met een let ipv. een var.

We gaan dit nu runnen.

En een const is idem aan een let qua scope!

```
{  
  var scoped = "Available!";  
}  
console.log(scoped);  
  
{  
  let scoped = "Let me be.";  
}  
console.log(scoped);  
→ Uncaught ReferenceError: scoped is not  
defined
```

# Conclusie: variabelen

---

We schrijven volgens camelCase.

We kiezen const en let boven var:

Het is altijd beter om de scope af te schermen.

Als iets niet meer kan of mag aanpassen: const.

# Conclusie: functies

We schrijven functies die één klein stukje functionaliteit doen.

We zetten een functie in een const variabele.

```
const calculateSurface = function(a, b) {  
    return a * b;  
};
```

# Controlestructuren

Halt .

# Controlestructuren

---

De basis van elke programmeertaal is aan de hand van controlestructuren. Een paar voorbeelden:

Dit is een standaard statement.

```
let age = 36;  
  
if (age < 30) {  
    console.log('Je bent jong!');  
} else if (age < 45) {  
    console.log('Je bent ouder!');  
} else {  
    console.log('Je bent oud!');  
}
```

# Controlestructuren

---

Soms weten we niet zeker of een variable al iets van inhoud heeft.

We kunnen dan in een keer kijken of een variabele niet per ongeluk:

- leeg is (lege string),
- null is
- of undefined is.

```
if (someObject) {  
    console.log('Something:',  
    someObject);  
}
```

# Controlestructuren

---

Als we meerdere mogelijkheden hebben die zouden kunnen, gebruiken we een switch statement.

```
let color = 'green';

switch (color) {
  case 'blue':
    console.log('The color is sea.');
    break;
  case 'green':
    console.log('The color is grass.');
    break;
  case 'red':
    console.log('The color is blood.');
    break;
  default:
    console.log('Color not valid... 😢');
}
```

# Debugging in JS

Wat gebeurt er?

# Debugging: moeilijk of niet?

---

JS debugging wordt soms als moeilijk ervaren.

Vaak typfouten: wordt niet aangeduid.

Goede formattering van code helpt fouten vermijden.

# Debugging: console

---

Je kan een variabele bekijken in de console.

Er zijn verschillende manieren om dit te doen. We gaan dit even testen.

Let niet op de undefined meldingen.

Dit kan je geregeld in je code plaatsen om in de console te kijken of iets niet (goed) werkt.

```
//Type in Google Chrome
let test = "Hallo!";

//Probeer de volgende dingen:
console.log(test);
console.log({ test });
console.table(["zeg eens", test]);
console.error(test);
console.warn(test);
console.info(test);
```

# Debugging: chrome

---

Mogelijkheid om breakpoints te zetten.

**Demo**

# DOM

Heel erg slim.

# DOM = Document Object Model

---

Alle elementen die je in een **HTML** pagina hebt.

**HTML != DOM Tree**

Een browser maakt van de HTML (tekst) de DOM tree (objecten).

De **DOM** is **een weerspiegeling** van je HTML document:

De browser voorziet API's om er wat mee te doen, zodat je dit kan **inlezen en manipuleren**.

Let op: de **DOM moet eerst ingeladen zijn**.

Een paar **demo's in JS**

# DOM API's via JS: ophalen van een element

---

We hebben de volgende HTML in de body:

```
<h1>Hello buddy 🙌</h1>
<p class="description">Welcome here.</p>
<button id="surprise">Surprise Me</button>
```

We halen met JS een element op aan de hand van een element-naam, **class** en ID.

```
// Element selector (de tag-naam)
let h1 = document.querySelector('h1');

// Class selector (de class-naam -> .)
let class = document.querySelector('.description');

// ID selector (de ID)
let surprise = document.querySelector('#surprise');
surprise = document.getElementById('surprise');
```

# DOM API's via JS: ophalen van elementen

---

We hebben de volgende HTML in de body:

- We halen met JS elementen op aan de hand van een element-naam, classes.
- Dit kan niet voor ID's; een ID moet uniek zijn.
- Dit geeft je een Array terug (leeg, 1 element of meerdere).
  - NodeList; [element, element]

```
<p class="description">Welcome here!</p>
<p class="description">More info,</p>
<p class="description">in the next p.</p>
```

```
let pTags = document.querySelectorAll('p');
let descriptions =
document.querySelectorAll('.description');
```

# DOM API's via JS: ophalen van elementen en weergeven

---

Navigeer naar de website [mct.be](http://mct.be)

Typ deze code in de console.

Met een for-loop overlopen we de elementen.

Er zijn nog andere loops mogelijk:

for-of, for-in, forEach

Map

while

...

```
let pTags = document.querySelectorAll('p');
for (let i = 0; i < pTags.length; i++) {
    console.log(pTags[i]);
}
```

# DOM API's via JS:wegschrifven van elementen

---

Als je (HTML) tekst in de DOM zet, wordt dit een element.

Let op het gebruik van ' en " in de opbouw van een string (attributen).

Met backticks ` kan je een variabele zetten in een string; dat met \${ ... }.

Je kan via JS ook een DOM element maken.

```
<!-- Here comes the HTML -->
<div id="holder"></div>

let message = 'Hello!';
let holder = document.querySelector('#id');

holder.innerHTML = '<p class="test">Hello!</p>';
holder.innerHTML = `<p class="test">${message}</p>`;

const newP = document.createElement('image');
```

# DOM API's via JS: erg langzaam

---

De **DOM** op een webpagina is **traag om aan te passen** (dus niet om in te lezen).

Als we **DOM toevoegen of aanpassen** proberen we dit **in een keer te doen!**

*Een kleine DOM is beter:*

*Laadt sneller.*

*Minder ‘werk’ om iets op te zoeken met een selector.*

# Herhaling

---

Welke variabelen gaan wij gebruiken?

Wat is kenmerkend aan deze 2 variabelen?

Hoe schrijven we een functie netjes?

Wat is de beste manier om een heel aantal conditionele opties te overlopen?

Wat zijn de beste manieren om te debuggen in Chrome?

Waarvoor staat DOM?