

Javascript – deel 06

Deze les behandelt

- elementen uit de DOM-tree opvragen op basis van een CSS-selector
- de werking en nadelen van `.innerHTML`
- content toevoegen met `.insertAdjacentHTML()`
- attributen opvragen en instellen
- event bubbling

Verslag

In dit deel van de cursus staan verschillende vragen die je moet beantwoorden en opdrachten om iets te maken of uit te proberen. Het is belangrijk dat je alle opdrachten zorgvuldig uitvoert!

Documenteer je werk in een verslag document 'javascript deel 06' waarin je

- voor elke uitprobeer opdracht een entry maakt met screenshots ter staving van wat je deed
- je antwoorden op de gestelde vragen neerschrijft

Oplossingen van grotere opdrachten (met veel code) bewaar je aparte folders in een Webstorm project.

Elementen uit de DOM-tree opvragen (bis)

We zagen in 'Javascript deel 03' hoe we bepaalde elementen uit de DOM-tree kunnen te pakken krijgen, bv. om hun properties te wijzigen. We deden dit met enkele get-methods van het **document** object :

```
document.getElementById(id)
document.getElementsByClassName(className)
document.getElementsByTagName(tagName)
```

Het document object voorziet echter nog een paar andere handige methods om dit te doen.

Twee zeer belangrijke methods zijn **querySelector** en **querySelectorAll**

```
document.querySelector(selector)
document.querySelectorAll(selector)
```

De 'selector' parameter is een string die een CSS-selector bevat, bv. `#playfield` of `.important>img`

```
document.querySelector("#playfield")
document.querySelectorAll(".important>img")
```

De method **querySelector** retourneert een verwijzing naar **het eerste element** dat door de CSS-selector geselecteerd wordt (of retourneert de null waarde indien er geen is).

De method **querySelectorAll** geeft een verzameling terug met verwijzingen naar **alle elementen** die door de selector geselecteerd worden. Deze verzameling is een NodeList, je kunt er op dezelfde manier mee omgaan als met het resultaat van bv. `getElementsByClassName`.

Er bestaan trouwens nog meer soorten CSS-selectoren dan deze die in de CSS-lessen aan bod kwamen.

Het is beslist de moeite om wat tijd in te stoppen in [al de verschillende soorten CSS-selectoren](#) : hoe handiger je hiermee bent, hoe eleganter je CSS-regels en je Javascript code kunnen worden.

Met name

- id selector
- class selector
- descendant combinator
- child combinator
- attribute selector
- [pseudo-class selector](#) (zoals `:checked` en `:nth-of-type`)

Je kunt bijvoorbeeld elementen selecteren op basis van hun attribuutwaarden met []. Zo zal de selector `a[href^="http:"]` enkel hyperlinks selecteren wiens href attribuut begint met "http:", d.w.z. alle links naar externe sites dus. Meer info over attribuut selectoren vind je op

https://developer.mozilla.org/en/docs/Web/CSS/Attribute_selectors

Er bestaat trouwens ook een **getElementsByName** method :

```
document.getElementsByName(naam)
```

De 'naam' parameter is een string en de method retourneert een lijst met de DOM-tree elementen die een 'name' attribuut hebben met de gegeven naam. Dit kan wel eens handig zijn als je met forms werkt maar helaas werkt deze method niet op alle browsers op exact dezelfde manier, zie

<https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementsByName>

Als betrouwbaarder alternatief kun je `querySelectorAll("[name='xyz']")` gebruiken 😊

Bijvoorbeeld, om een inputveld met `name="firstname"` in een form met `id="frmContact"` te selecteren :

```
document.querySelector("#frmContact [name='firstName']")
```

(let op de spatie in die selector!)

In al het voorgaande zochten we steeds elementen op documentniveau, d.w.z. eender waar in de DOM-tree ofte 'eender waar in de pagina'. De zoektocht begon steeds bij de root van de DOM-tree, nl. het `<html>` element.

Soms is het echter nodig/handig/permanter om elementen te zoeken, die kinderen zijn van een element dat we reeds eerder te pakken kregen. Gelukkig ondersteunt elk DOM-tree element zelf ook de eerder geziene opzoekmethoden!

Bijvoorbeeld, om alle foutmeldingen (i.e. de tekst in spans met `class="errorMessage"`) uit een form met `id="frmContact"` te verwijderen :

```
let frmContact=document.getElementById("frmContact");
...
let errorMessages=frmContact.getElementsByClassName("errorMessage");
for (let i=0;i<errorMessages.length;i++) {
    errorMessages[i].textContent="";    // maak de span leeg zodat de foutmelding verdwijnt
}
```

Je ziet dat de zoektocht begint bij het **frmContact** object en niet bij document zoals gewoonlijk.

In dit specifieke voorbeeld kon het weliswaar iets eenvoudiger met `document.querySelectorAll` en de descendant selector `#frmContact .errorMessage` (let op de zeer belangrijke spatie in de selector!)

```
let errorMessages=document.querySelectorAll("#frmContact .errorMessage");
for (let i=0;i<errorMessages.length;i++) {
    errorMessages[i].textContent="";
}
```

Soms is het echter handiger en/of efficiënter om te beginnen bij een DOM-tree element dat je al hebt, bijvoorbeeld het element dat je kreeg via `event.target`.

Opdracht 01

Unzip de file 'begin situatie opdracht 01.zip' in een Webstorm project (of folder).

Je mag de inhoud van het HTML-bestand niet wijzigen in deze opdrachtenreeks!

Vul de Javascript code verder aan voor de volgende functionaliteit :

- Een klik op een ingrediënt maakt het rood

Je zult misschien in de verleiding komen om op elk ingrediënt een class "ingredient" toe te voegen? Dat zou een goeie oplossing zijn, maar hier is het expliciet de bedoeling om dit niet te doen (vandaar dat je het HTML-bestand niet mag wijzigen). Gebruik `querySelectorAll` met een descendant-selector om de kinderen te bekomen van het `` element dat de ingrediënten bevat.

Test nu eerst je oplossing uit en ga na dat een klik op een ingrediënt, een rode tekst oplevert.

Breid nu je oplossing uit met

- Een klik op de 'Voeg toe' knop, voegt een ingrediënt toe
- Een klik op de 'Wis alles' knop, verwijdert alle ingrediënten

Ik ga ervan uit dat je `.innerHTML+=` gebruikt hebt om een `` toe te voegen, iets anders hebben we immers tot nu toe nog niet gezien. Beantwoord de volgende vragen :

- nadat je een ingrediënt hebt toegevoegd, kun je dan een oorspronkelijk ingrediënt nog steeds rood maken? Waarom niet?
- kun je een nieuw ingrediënt ook rood maken door erop te klikken? Waarom niet?

Voor- en nadelen van .innerHTML

Eerder zagen we hoe we nieuwe elementen aan de DOM-tree kunnen toevoegen, door de `.innerHTML` van de toekomstige parent aan te passen.

Bijvoorbeeld, een lijst opvullen met nieuwe items deden we met

```
let firstList = document.getElementsByTagName("ul")[0];           // de toekomstige parent

firstList.innerHTML = "<li>item 1</li><li>item 2</li><li>item 3</li>";
```

Realiseer je dat de browser relatief veel werk moet verrichten als we de waarde van de `.innerHTML` property van een element opvragen :

- in de DOM-tree moeten alle descendants van het element opgezocht worden
- van elke descendant moet een HTML-tekst voorstelling gemaakt worden (i.e. een *serialisatie*)

Zo'n serialisatie bevat enkel de tags, teksten en attributen van de descendant elementen, andere informatie uit de DOM-tree zoals gekoppelde event listeners zal er niet in voorkomen!

De waarde van de `.innerHTML` property van een element wijzigen is ook complex, de browser moet :

- alle kinderen van het element weggooien (!!!)
- de nieuwe HTML-string ontleden (i.e. parsen) om de elementen en hun attributen te vinden
- DOM-tree nodes maken voor al deze elementen
- deze nieuwe nodes toevoegen als descendant nodes in de DOM-tree
- de pagina layout en element styling opnieuw berekenen en hertekenen

Het grote voordeel van `.innerHTML` is de eenvoudige code : het is heel makkelijk (1 opdracht!) om iets in de DOM-tree te krijgen.

De nadelen van `.innerHTML` zijn

1. de browser moet veel werk verrichten om de aanpassing door te voeren (zie hierboven)
2. we kunnen enkel alle kinderen vervangen, inlassingen of toevoegingen zijn niet mogelijk
3. we krijgen geen verwijzing naar de nieuwe DOM-tree nodes

Bij puntje 2 denk je misschien dat het wel meevalt, we kunnen immers makkelijk vooraan of achteraan toevoegen met code als

```
firstList.innerHTML += "<li>nieuw item</li>";           // achteraan 'toevoegen'
firstList.innerHTML = "<li>nieuw item</li>" + firstList.innerHTML; // vooraan 'toevoegen'
```

Maar van 'toevoegen' is eigenlijk geen sprake : telkens zullen alle kinderen in de DOM-tree vervangen worden. Bestaande kinderen worden weliswaar vervangen door exacte kopies, maar het resulteert wel degelijk in andere objecten in de DOM-tree. Indien we dus in een variabele een verwijzing hadden naar de originele DOM-tree nodes, dan zullen deze verwijzingen plots niet meer kloppen. Hetzelfde geldt voor de gekoppelde event listeners : hun element zit niet meer in de DOM-tree en daardoor zal de browser ze niet meer oproepen.

Merk op dat inlassingen middenin zo goed als onmogelijk zijn, niemand wil code schrijven om de serialisatie te analyseren om het juiste inlaspunt te vinden!

Om een een nieuw child element toe te voegen kun je beter **.insertAdjacentHTML** gebruiken (zie verderop) i.p.v. `.innerHTML += "..."` te schrijven.

Puntje 3 is nogal belangrijk : vaak willen we met sommige van de nieuw toegevoegde nodes nog iets doen in ons programma zoals bv. een event listener koppelen. Als we geen verwijzing naar de nieuwe elementen bekomen, moeten we ze na het toevoegen meteen opzoeken in de DOM-tree wat vaak nogal omslachtig is.

Enkele bladzijden terug zagen we dat de 'DOM-tree opvraag methoden' ook op element-niveau kunnen gebruikt worden (en dus niet enkel op document-niveau). Dit kan het 'meteen opzoeken in de DOM-tree' iets makkelijker te maken (nl. via het element wiens `.innerHTML` we aanpasten), maar het blijft onhandig.

In een volgende les zullen we zien hoe we zelf expliciet DOM-tree nodes kunnen aanmaken en toevoegen aan de DOM-tree. Dit biedt een oplossing voor alle bovenstaande problemen, maar leidt onvermijdelijk tot ingewikkeldere code.

Je zult je dus in elke situatie opnieuw moeten afvragen, welke aanpak je best kiest.

DOM-Element method `insertAdjacentHTML()`

Alle DOM-tree elementen ondersteunen een method [insertAdjacentHTML](#) waarmee makkelijk content kan toegevoegd worden op welbepaalde plaatsen.

- `element.insertAdjacentHTML(pos, html)`
 - analyseert de html tekst en voegt de DOM-elementen toe relatief t.o.v. het element
 - de 'html' parameter is een string met HTML-tekst
 - de 'pos' parameter geeft aan waar de nieuwe elementen moeten toegevoegd worden
 - `beforebegin`, `afterbegin`, `beforeend` of `afterend`
- indien het element een `` is met drie `` kinderen, is de positie als volgt :

```
beforebegin
<ul>
  afterbegin
  <li>...</li>
  <li>...</li>
  <li>...</li>
  beforeend
</ul>
afterend
```

Voorbeeld :

```
let firstList = document.getElementsByTagName("ul")[0];
firstList.insertAdjacentHTML("beforeend", "<li>een nieuw item</li>");
```

deze code gebruikt positie "beforeend" om het nieuwe item achteraan de lijst toe te voegen (i.e. als nieuwe laatste `` kind)

Zoals in nadeel puntje 2 uit de vorige sectie werd aangehaald :

Gebruik liever `.insertAdjacentHTML()` in plaats van `.innerHTML`

Merk op `.insertAdjacentHTML()` het probleem van puntje 3 niet oplost!

In een volgende les zullen we zien hoe we dit probleem kunnen oplossen door

- zelf expliciet de nodige DOM-tree node objecten aan te maken en ze toe te voegen in de boom, dit is de krachtigste manier maar tegelijk ook de meest omslachtige qua code.
- de lijst van alle child elements op te vragen en daaruit bv. de eerste/laatste te nemen
- op element-niveau met `.querySelector()` via een goedgekozen CSS-selector, het net toegevoegde child element op te vragen.

Opdracht 02

Op het einde van opdracht 01 konden we na het toevoegen van een nieuw ingrediënt, de 'oude' ingrediënten niet meer rood kleuren, zie vraag (a).

We zouden dit kunnen oplossen door in de click event listener van `#btnAdd`, na het toevoegen van een nieuw `` element, steeds alle `` elementen opnieuw koppelen aan de 'maakBelangrijk' click event listener.

Dat is echter een prutsoplossing die zelden werkt in grotere programma's, omdat je vaak niet weet welke extra event listeners bepaalde libraries her en der in de DOM-tree hebben geregistreerd.

Los het probleem uit vraag a) op door een nieuw ingrediënt steeds toe te voegen met `.insertAdjacentHTML()` i.p.v. met `.innerHTML+=`

Los vervolgens het probleem uit vraag b) op, door het laatste child element van `#lstIngredients` te pakken te krijgen (want dat is het `` element dat net werd toegevoegd) koppel daar de click event listener functie met naam 'maakBelangrijk' aan.

Met hetgeen we tot nu toe gezien hebben, zou je dit kunnen doen door

- ofwel een slimme CSS-selector (bijvoorbeeld `:last-of-type`) te gebruiken
- ofwel in `#lstIngredients` verder te zoeken met `getElementsByTagName("li")` en het laatste element uit die verzameling te nemen
 - het ganse document doorzoeken met `document.getElementsByTagName("li")` en dan het laatste nemen zal trouwens niet werken, dat is net de reden waarom er een `` met instructies onderaan de HTML staat 😊

Attributen van DOM-elementen manipuleren

De attributen van een HTML-element kun je opvragen en aanpassen met de volgende methods

- **element.getAttribute(a)**
 - retourneert de waarde van attribuut 'a' van het element
 - de 'a' parameter is een string met de naam van het attribuut
 - bv. "href", "src", etc.
 - bv. element.getAttribute("href")
- **element.setAttribute(a, v)**
 - stelt de waarde van attribuut 'a' in op waarde 'v', voor dit element
 - de 'a' parameter is een string met de naam van het attribuut
 - bv. "href", "src", etc.
 - de 'v' parameter is een string met de waarde van het attribuut
 - bv. element.setAttribute("href", "http://www.example.com");
- **element.hasAttribute(a)**
 - geeft aan of het element het 'a' attribuut al dan niet heeft
 - te gebruiken indien je niet op voorhand weet of het element het attribuut heeft
 - bij sommige browser(versies) geeft getAttribute bij afwezigheid namelijk een lege String terug ipv de null waarde

Bijvoorbeeld

```
let firstImage = document.getElementsByTagName("img")[0];  
let oldSource = firstImage.getAttribute("src");  
firstImage.setAttribute("src", "images/logo.png");
```

Opdracht 03

Schrijf een webpagina met een inputveld, een 'Wijzig' button en een afbeelding (een `` element).

Initieel wordt deze afbeelding getoond (dit is hardgecodeerd in de HTML) :

https://upload.wikimedia.org/wikipedia/en/0/02/Homer_Simpson_2006.png

en ziet de pagina er als volgt uit :



De beginwaarde in het inputveld is gebaseerd op het 'src' attribuut van het `` element in de HTML-code. Indien we de HTML-code zouden aanpassen naar een andere 'src' url, dan zou ook de begintekst in het `<input>` element anders zijn. Je zult dit dus in je setup functie moeten regelen.

Door in het inputveld een afbeeldings-url te typen en op 'Wijzig' te klikken, kan de gebruiker de afbeelding op de pagina veranderen.

Let op, we leggen in deze opdracht een extra eis op : je Javascript code mag enkel het 'src' attribuut van het bestaande `` element aanpassen, de afbeelding mag niet vervangen worden door een ander `` element.

Event bubbling

Tot nu toe hebben we steeds de event listener gekoppeld aan het element waar het event zich zal voordoen. Bijvoorbeeld, de code voor een button click, koppelden we aan de button zelf.

In eenvoudige programma's lukt dit doorgaans, maar in complexere situaties is dit niet altijd mogelijk of soms zelfs niet wenselijk.

Bijvoorbeeld, als een library een pak geneste HTML-elementen toevoegt en de gebruiker klikt ergens op, dan ontstaat het click event ergens diep in dat geneste kluwen. Het is niet eenvoudig om precies dat ene (diep geneste) element te pakken te krijgen om er onze event listener aan te koppelen.

Gelukkig biedt **event bubbling** hiervoor een oplossing!

Telkens er een event optreedt in een element, worden de gekoppelde event listeners van dat element opgeroepen ***en daarna deze van de ancestors van dat element!***

Demonstratie event bubbling

Unzip de file 'demo event bubbling.zip' in een Webstorm project of folder.


Bekijk de HTML-file en **teken voor jezelf de DOM-tree van dit HTML document**.

Er is een `` element, met drie `` elementen die elk een `<a>` bevatten. In totaal dus 1 `` element, 3 ``'s en 3 `<a>`'s.

Dezelfde click event listener functie 'klik' wordt geregistreerd bij elk van deze 7 elementen. Deze functie toont op de console wat voor soort elementen 'event.target' en 'event.currentTarget' zijn.

Neem je DOM-tree schema erbij en voer de volgende opdrachten uit

- Klik op het blauwe gebied en kijk welke output er komt op de console.
- Klik op een roze gebied en kijk welke output er komt op de console.
- Klik op een grijs gebied en kijk welke output er komt op de console.

Wis telkens de console zodat je goed ziet welke output je klik opleverde! Je kunt de console wissen door op het  icoontje te klikken linksbovenaan in de console.

Kun je de output verklaren?

Als je op een hyperlink klikt, wordt blijkbaar niet alleen diens event listener opgeroepen maar ook die van alle ancestors! (einde demonstratie)

Eerder zagen we al dat elke event listener functie eigenlijk een parameter heeft die het event voorstelt dat zich voordeed. Deze parameter wijst naar een event object met een aantal interessante properties waarvan we 'target' al kennen :

- `event.target`
 - dit is het DOM-tree element waar het event zich voordeed, bv. waarop geklikt werd
- `event.currentTarget`
 - dit is het DOM-tree element wiens event listener we aan het uitvoeren zijn.

De `.currentTarget` property heeft te maken met de manier waarop de browser een event afhandelt, nml. welke event listeners moeten reageren op een event.

Wanneer we bv. op een element klikken, ontstaat het click event op dit element maar "borrelt" eveneens omhoog doorheen de DOM-tree. Tijdens deze "bubble" fase krijgen event listeners van de ancestors ook de kans om te reageren op het event!

Een zeer leerrijke beschrijving vind je op

http://www.quirksmode.org/js/events_order.html

Voorbeeld :

```
element.addEventListener("click", klik);
...
const klik = (event) => {
  console.log("u klikte op een "+event.target.name+" element");
  console.log("de listener is geregistreerd bij een "+event.currentTarget.name+" element");
}
```

Heel vaak zijn `.target` en `.currentTarget` per definitie gelijk in ons programma : we registreren doorgaans de event listener bij precies dát object waar het event zich zal voordoen.

Bv. tot nu toe registreerden we de click event listener netjes bij de button waarop geklikt kan worden.

Als onze Javascript code rechtstreeks (of indirect via een library) wat complexere HTML toevoegingen doet, kan het interessant zijn om voor een andere aanpak te kiezen : de event listener wordt dan geregistreerd bij een ancestor van de elementen waar het event zich zal voordoen.

Bv. stel dat we dynamisch een gallerij met afbeeldingen opbouwen (een <section> met kinderen) waarin de afbeeldingen een bepaald klikgedrag vertonen. We kunnen dan de click eventlistener aan het <section> element koppelen i.p.v. aan elk element afzonderlijk.

De redenen om voor deze aanpak te kiezen hebben te maken met het structureren van code in complexere programma's, bijvoorbeeld

- de code die de elementen toevoegt, is vaak niet de juiste plaats is om het gedrag van die elementen vast te leggen
 - bv. pagina navigatie via een lijst met links (cfr. listview in JQuery Mobile)
- een eventlistener op ancestor niveau laat ons toe om het gedrag op een prominente plaats vast te leggen i.p.v. in een eerder obscure functie die de elementen toevoegt
 - bv. in de window load listener en niet in een anonieme AJAX callback functie
- we kunnen gedrag definiëren ongeacht hoe de kinderen worden toegevoegd
 - bv. handig om snel iets te proberen met enkele hardgecodeerde kinderen in de HTML

Een event object heeft o.a. ook nog twee algemene methods die de event afhandeling beïnvloeden

- event.stopPropagation()
 - stop de event bubbling fase (i.e. de listeners van verdere ancestors worden niet verwittigd)
- event.preventDefault()
 - stop het standaard gedrag voor dit event, bv. om te beletten dat de browser naar een andere pagina navigeert als iemand op een link klikt.

Je kan trouwens ook gewoon false retourneren uit je event listener, dat heeft hetzelfde effect als stopPropagation() en preventDefault() tegelijk.

Dan zou je je de vraag kunnen stellen, moet je in je event listener 'event.target' of 'event.currentTarget' gebruiken? Daar is geen simpel antwoord op te geven, het hangt er werkelijk van af wat je probeert te bereiken.

Merk op dat je in een click event listener, met

```
if (event.target === event.currentTarget) {
    // rechtstreeks geklikt op het element met de gekoppelde event listener
} else {
    // geklikt op een descendant van het element met de gekoppelde event listener
}
```

kunt achterhalen of er rechtstreeks op het element werd geklikt dan wel op een van z'n descendants!

Opdracht 04

Wijzig de oplossing van opdracht 02 zodat de 'maakBelangrijk' click event listener gekoppeld wordt aan het bovenste element (d.w.z. #IstIngredients) en niet aan de individuele elementen van de ingrediënten.

Vereenvoudigt dit het ganse programma? Op welke manier?

Merk ook op dat deze ene koppeling ervoor zorgt dat zowel ingrediënten uit de HTML code alsook deze die via Javascript werden toegevoegd, rood kunnen kleuren!

Event listeners en this

Wij gebruiken event.currentTarget om het element te achterhalen wiens event listener aan het uitvoeren is.

Merk op dat je heel vaak code zult tegenkomen waarin daarvoor **this** gebruikt wordt.

Dit is in de meeste omstandigheden ook correct maar kan bij complexere code wel eens een onverwacht resultaat opleveren.

De redenen waarom 'this' soms iets anders oplevert is zeer technisch en wordt hier niet behandeld.

Weet wel dat als je event.currentTarget gebruikt i.p.v. 'this', je altijd het juiste resultaat krijgt.

In het soort code die we in deze lessen gebruiken is er echter geen verschil tussen beide.

Terzijde : Bij arrow functies is er zelfs geen this!