Javascript – deel 06

Deze les behandelt DOM-tree nodes en gaat dieper in op event handling.

Permanente evaluatie

In de volgende les kan je docent je oplossingen van de opdrachten opvragen en laten meetellen voor je permanente evaluatie.

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.

DOM-tree elementen opvragen

We zagen eerder al 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.

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 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 mogelijkheden voor CSS-selectoren dan we in de CSS-cursus gezien hebben. Als je querySelectorAll gebruikt is het wel handig om deze ook te kennen.

Opdracht: bekijk het voorbeeld “demo-events.zip”

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 ook een **getElementsByName** method :

```
document.getElementsByName(name)
```

De 'name' parameter is een String en de method retourneert een verzameling 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 begonnen we steeds elementen te zoeken op documentniveau, we gebruikten hiervoor de get-methods van het document object.

Soms is het echter nodig/handig/performanter 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 opzoekmethods!

Bijvoorbeeld, om alle foutmeldingen (bv. spans met class="errorMessage") uit een form met id="frmContact" te verwijderen :

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

In dit geval kon het ook eenvoudiger met querySelectorAll

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

maar soms is het wel handig om op basis van een specifiek DOM-tree element verder te kunnen zoeken.

DOM-tree elementen toevoegen of verwijderen met .innerHTML

In een vorige les zagen we dat 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];
firstList.innerHTML="<li>item 1</li><li>item 2</li><li>item 3</li>";
```

Werken met .innerHTML heeft echter een paar nadelen :

1. de browser moet veel werk verrichten om de aanpassing door te voeren
    - de HTML tekst ontleden en analyseren om te weten wat er precies moet gebeuren
    - de DOM-tree bewerken : de nodes voor de bestaande kinderen verwijderen, node objecten voor de nieuwe kinderen aanmaken en deze aan de DOM-tree toevoegen
    - de pagina layout opnieuw berekenen en hertekenen
2. we krijgen geen verwijzing naar de nieuwe DOM-tree elementen
    - als we er programmatorisch nog iets mee willen doen (bv. een eventlistener koppelen), moeten we ze opvissen uit de DOM-tree, wat niet altijd even eenvoudig als ze eerdere generiek zijn.
3. we kunnen enkel alle kinderen vervangen

Bij het derde puntje 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'
```

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 elders in ons programma verwijzingen hadden bijgehouden naar de originele DOM-tree objecten zullen deze verwijzingen plots niet meer kloppen. Idem met gekoppelde event listeners.

Merk op dat inlassingen middenin trouwens zeer complex kunnen zijn, we moeten zelf code schrijven om het juiste inlaspunt te bepalen. We verkrijgen immers de .innerHTML tekst en onze code moet de HTML-code dan zelf ontleden.

Voor eenvoudige gevallen kunnen we ons behelpen met de .innerHTML property, de code die we hiervoor moeten schrijven is relatief eenvoudig. Voor complexere situaties, kunnen we gebruik maken van geavanceerdere properties en methods die elk DOM-tree element ter beschikking stelt. Deze bieden ons krachtigere mogelijkheden, maar onze code zal complexer zijn.

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

DOM-tree node vs. DOM-tree element

Als we het over de bestanddelen van de DOM-tree hebben, moeten we eigenlijk over **DOM-tree nodes** spreken en niet over DOM-tree elementen.

De meeste nodes stellen weliswaar elementen (zoals <ul> en <li>) voor, maar de DOM-tree bevat ook nodes voor andere zaken zoals bv. de tekstuele inhoud van een element en de commentaar stukken in het HTML document.

Voorheen hebben we het onderscheid niet gemaakt omdat we steeds met nodes werkten die elementen voorstelden. De namen van de methods die we hiervoor gebruikten gaven dit ook al aan : get**Element**ById, get**Elements**ByClassName, etc.

In wat volgt is het echter belangrijk dat je je van het verschil bewust bent.

DOM-tree nodes

Elke node n uit de DOM-tree heeft een aantal properties die te maken hebben met haar relatie tot andere nodes in de DOM-tree :

- n**.parentNode**
  - geeft de parent van n
  - geeft null indien er geen is, bv. omdat n (nog) niet in DOM-tree zit of een document node is

- n**.childNodes**
  - geeft een soort lijst object dat alle children van n bevat
  - dit lijst object is geen array, maar je kan wel .length en [idx] erop toepassen

- n**.firstChild** / n**.lastChild**
  - geeft het eerste/laatste child van n
  - geeft null indien er geen is

- n**.nextSibling** / n**.previousSibling**
  - geeft de volgende/vorige sibling van n
  - geeft null indien er geen is

Vaak zijn we enkel geïnteresseerd in de DOM-tree nodes die elementen voorstellen. Bovenstaande methods geven ons echter alle soorten nodes. We moeten dan ietwat omslachtige code schrijven om enkel de element nodes eruit te pikken en de overige soorten te negeren.

Een node heeft ook enkele methods die dit werk voor ons doen :

- n**.children**
- n**.firstElementChild** / n**.lastElementChild**
- n**.nextElementSibling** / n**.previousElementSibling**

Let wel op, sommige browsers onderteunen deze pas sinds kort (op een correcte manier).

De DOM-tree kan veel verschillende soorten nodes bevatten, bijvoorbeeld

- element nodes (voor html elementen zoals <h1>, <a>, <p>, etc.)
- text nodes (voor de teksten die in en tussen de elementen staan)
- comment nodes (voor html commentaren tussen <!-- en -->
- ...

Elke node heeft een aantal properties die ons vertellen wat voor soort node het is :

- n**.nodeName**
  - geeft het soort element als string indien e een element node is, bv. "img", "h1", etc.
  - geeft "#text" indien n een text node is

- n**.nodeType**
  - geeft een getal naargelang welke soort node n is
    - bv. 1 indien n een element node is
    - bv. 3 indien n een text node is
  - zie https://developer.mozilla.org/en-US/docs/Web/API/Node/nodeType

- n**.nodeValue** (enkel voor text nodes!)
  - geeft de tekst van de node als string, indien n een text node is
  - geeft null indien n een element node is

De text nodes waarvan hierboven sprake is, zijn extra nodes in de DOM tree die de teksten bevatten die tussen de element tags staan.

Voorbeeld

Het HTML fragment

```
<p>
    hallo
    <span>dit is een</span>
    tekst
</p>
```

ziet er in de DOM-tree als volgt uit

```
P ELEMENT NODE
    TEXT NODE  met nodeValue="hallo"                    (*)
    SPAN ELEMENT NODE
        TEXT NODE met nodeValue="dit is een"
    TEXT NODE met nodeValue="tekst"                     (*)
```

(*) in de nodeValue strings zit ook nog whitespace en newlines, maar da's moeilijk om hier te tonen.

Je ziet dus dat element nodes nooit zelf 'hun' tekst bevatten, deze wordt altijd in een text node verpakt!

Voor eenvoudige element nodes die enkel tekst bevatten en geen kinderen hebben, is het natuurlijk veel handiger om de **.textContent** property te gebruiken dan text nodes te manipuleren.

Opdracht

Stop het HTML fragment uit bovenstaand voorbeeld in een geldig (!) HTML bestand en geef de paragraaf een id="abc".

Voorzie de pagina van een stukje javascript code die de element node voor de paragraaf met id="abc" opzoekt.

Open deze pagina in Chrome en open vervolgens de Chrome Developer Tools.

Plaats een breakpoint net na de code die de bewuste paragraaf opvist en herlaad de pagina.

Bekijk in de debugger de boomstructuur door de parentNode/childNodes links te volgen.

Ga na wat de precieze inhoud is van de TEXT nodes.

Bekijk nu het document object (dit is ook een node trouwens). Je kunt dit bv. in het Watch venster van de debugger doen. Volg weerom de parentNode/childNodes. Welke soorten nodes kom je tegen?

Werken met DOM-tree nodes

In deze sectie wordt uitgelegd hoe we rechtstreeks met node objecten kunnen werken (dit is dus het krachtigere alternatief voor .innerHTML dat eerder werd aangehaald).

Nodes aanmaken

De eigenlijke node objecten aanmaken gebeurt via het document object :

- document**.createElement(s)**
  - retourneert een nieuwe element node
  - de 's' parameter is een string die aangeeft om wat voor element het gaat
    - bv. "img", "h1", etc.

- document**.createTextNode(t)**
  - retourneert een nieuwe text node
  - de 't' parameter is een string met de tekst voor de node
  - (soms is .textContent echter een handiger alternatief)

Bijvoorbeeld

```
let elementNode = document.createElement("a");
let textNode = document.createTextNode("Hello world!");
```

Element nodes en hun attributen

De attributen van een element node e kun je opvragen en aanpassen met de volgende methods

- e**.getAttribute(a)**
  - retourneert de waarde van attribuut a van node e
  - de 'a' parameter is een string met de naam van het attribuut
    - bv. "href", "src", etc.
  - bv. e.getAttribute("href")

- e**.setAttribute(a, v)**
  - stelt de waarde van attribuut a in op v, voor node e
  - de 'a' parameter is een string met de naam van het attribuut
  - de 'v' parameter is een string met de waarde van het attribuut

- e**.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 nml. 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");
```

Custom attributen

Het is toegelaten om zelfverzonnen attributen aan een element toe te voegen, op voorwaarde dat hun naam met "**data-**" begint.

Dit is handig om extra data aan specifieke elementen uit de DOM-tree op te hangen.

Bijvoorbeeld:

Stel dat je een applicatie moet schrijven om personen en vergaderingen te beheren.

De gegevens van een persoon zullen in je programma netjes in een javascript object worden bijgehouden, net als de gegevens over een vergadering (hoe dit eruit ziet komt in een volgende les aan bod). Dus per persoon 1 object, alsook per vergadering 1 object.

Ergens in je UI zal een aantal van die personen gevisualiseerd worden, bv. de lijst van genodigden voor een specifieke vergadering. Veronderstel dat we per persoon een <li> voorzien in die lijst.

Stel nu dat we deelnemers uit de vergadering kunnen verwijderen door erop te klikken. De code van de click eventlistener van het <li> element moeten kunnen achterhalen wat het bijbehorende persoon object is, zodat we dit uit het vergaderingsobject kunnen verwijderen. Het is dus niet voldoende om enkel iets uit de UI te verwijderen.

Een mogelijke oplossing hiervoor is een zelfgekozen attribuut "data-persoon" aan elk <li> element toe te voegen. In de click eventlistener kun je dan de waarde van dit attribuut opvragen en zo het persoon object terugvinden.

Wat de waarde van dit "data-persoon" attribuut is moet je zelf bedenken. Indien elk persoon object een uniek id heeft zou je deze id kunnen gebruiken. Dan noem je het attribuut beter ook "data-persoonId".

Merk op dat het wellicht geen goed idee is om hiervoor het id-attribuut van de <li> elementen te gebruiken.

- indien er meerdere vergaderingen op de pagina staan kunnen er nml. meerdere <li> items voor die ene persoon voorkomen en zitten we met dubbele waarden voor het id-attribuut.
- een vergaderingsobject zal wellicht ook een id hebben en als we daar ook het id-attribuut voor gebruiken kunnen we ook conflicten krijgen tussen id's van personen en id's van vergaderingen.

Nodes toevoegen en verwijderen

Nodes kunnen aan de DOM-tree worden toegevoegd of verwijderd via hun parent node p.

- p**.appendChild(c)**
  - voegt node c toe aan de parent node p

- p**.removeChild(c)**
  - verwijdert node c als child van parent node p

Bijvoorbeeld

```
let gallery = document.getElementsByClassName ("gallery")[0];
...
let image = document.createElement("img");
image.setAttribute("src", "images/logo.png");
gallery.appendChild(image);                    // image toevoegen
...
let someImage=gallery.children[3];             // vierde image (willekeurig)
gallery.removeChild(someImage);                // image verwijderen
```

Opdracht

Zie extra document: “extra_info_nodes.pdf”

Online beeldmateriaal: https://www.linkedin.com/learning-login/share?forceAccount=true&redirect=https%3A%2F%2Fwww.linkedin.com%2Flearning%2Fcollections%2F6645597977589137408%3Ftrk%3Dshare_collection_url&account=76814530

Event handling

We hebben al verschillende malen gebruik gemaakt van eventlisteners en bijna elke keer registreerden we per element een unieke eventlistener, bv. per button 1 aparte callback functie.

> *Een uitzondering hierop waren de eventlisteners van de sliders in de colorpicker opdracht. Daar registreerden we dezelfde callback functie bij elk van de drie sliders. Deze callback diende om de kleur van de swatch in te stellen en moest hiervoor sowieso de waarde van elke van de drie slider opvragen. Het gedrag was dus identiek voor elke slider, vandaar dat we dezelfde functie konden gebruiken.*

Wanneer bv. twee elementen totaal ander klikgedrag vertonen, dan is het logisch om per element een eigen eventlistener functie te voorzien.

Maar wat als het gedrag grotendeels gelijkloopt en enkel in een klein detail verschilt?

Wat heel vaak gebeurt is dat een pagina een verzameling elementen bevat die eigenlijk allemaal op dezelfde manier reageren op een event, maar waarbij het gedrag afgestemd is op het eigenlijke element waar het event ontstond.

> *Veronderstel een pagina met een aantal images die stuksgewijs verdwijnen als we erop klikken. De code in de click eventlistener zal voor elk element hetzelfde doen : het geklikte element verwijderen uit de DOM-tree. Hoe kunnen we achterhalen wat het geklikte element precies is?*

Welnu, elke event listener functie heeft eigenlijk een parameter die het event voorstelt dat zich voordeed. In de voorbeelden tot nu toe lieten we die echter steeds achterwege.

Deze parameter wijst naar een event object met een aantal interessante properties :

- 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.

Dit heeft te maken met de manier waarop de browser bepaalt 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 eventlisteners van de ancestors ook de kans om te reageren op het event.

Opdracht: bekijk zeker de demo: “demo event bubbling.zip”

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 eventlistener bij het object waar het event zich zal voordoen.

Bv. we registreren de click event listener netjes bij de button waarop geklikt kan worden.

Wanneer elementen programmatorisch worden toegevoegd, kan het interessant zijn om voor een andere aanpak te kiezen : de event listener wordt 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 <img> kinderen) waarin de afbeeldingen een bepaald klikgedrag vertonen. We kunnen dan de click eventlistener aan het <section> element koppelen i.p.v. aan elk <img> 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 i.g.v. een listview met links in JQuery Mobile
- een eventlistener op ancestor niveau laat ons toe om het gedrag op een prominente plaats vast te leggen i.p.v. een eerder obscure functie die de elementen toevoegt
  - bv. in de window load handler en niet in een AJAX callback functie
- we kunnen gedrag definieren 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.

Event listeners en this

Merk op dat je heel vaak code zult tegenkomen waarin **this** gebruikt wordt om het element te achterhalen wiens eventlistener aan het uitvoeren is.

Dit is in de meeste omstandigheden ook correct maar kan bij complexere code wel eens een onverwacht resultaat opleveren. In de code die we in deze lessen gebruiken is er echter geen verschil tussen beide.

De redenen waarom 'this' soms iets anders oplevert is zeer technisch en wordt hier niet behandeld. Weet wel dat als je **event.currentTarget** gebruikt, je altijd het juiste resultaat krijgt.

Zeer belangrijke opmerking 'this' werkt niet in een arrow functions -> bekijk zeker het voorbeeld "this-keyword.zip"

Opdracht : Colorpicker uitbreiding

Bekijk eerst het korte filmpje dat het eindresultaat demonstreert.

Maak een nieuw project dat een kopie is van de voorbeeldoplossing van de colorpicker opdracht.

Voeg rechts aan de colorpicker component een 'Save' knop toe.

Een klik op de 'Save' knop voegt onderaan een kopie van de swatch toe en een bijbehorende delete knop (met 'X' als opschrift). De swatch is dit keer rechthoekig (niet afgerond) en de delete knop staat netjes in de rechterbovenhoek van deze gekleurde rechthoek.

Een klik op de toegevoegde swatch, stelt de colorpicker component in op de bewaarde kleur.

Een klik op de bijbehorende delete knop verwijdert de bewaarde swatch weer.