# JAVASCRIPT

Theorie

Inhoudsopgave

# 1 Inleiding

*Client-side Scripting* is tegenwoordig even belangrijk als server-side scripting.

Met de komst van mobiele toestellen wordt interactie van de gebruiker met zijn toestel hoe langer hoe diverser en belangrijker.

Een degelijke kennis van Javascript is een *must* voor elke ontwikkelaar.

# 2 De cursus

## *2.1 Over deze handleiding*

De cursus "*Javascript* " bestaat uit twee grote delen: een basisgedeelte genaamd "*Programming Fundamentals*" en een "*Advanced*" gedeelte.
Het eerste deel is bedoeld om een zeer degelijke basis te ontwikkelen van client-side scripting met Javascript.
Het tweede deel is enkel bedoeld voor Javascript gebruikers die verder willen gaan en een echt gevorderde kennis willen ontwikkelen.

Het hangt van je traject af welke van de twee delen je moet doen.

Beide delen bestaan uit drie cursussen:

Zo bestaan er dus:

1. *"Javascript Programming Fundamentals Projecten"*
2. *"Javascript Programming Fundamentals Taken"*
3. *"Javascript Projecten Advanced"*
4. *"Javascript Taken Advanced "*
5. *"Javascript Theorie"*

Het Theorie boek geldt als naslagwerk voor beide gedeelten.

Een **eindwerk/test** wacht je aan het einde van elk deel, vraag je coach erom.

## *2.2 Hoe ga je tewerk?*

Dit is het **theoretische deel** voor de handleidingen " *Javascript Programming Fundamentals* " en "*Javascript Advanced*".

In de respectieve projectboeken wordt uitgelegd hoe je best tewerk gaat.

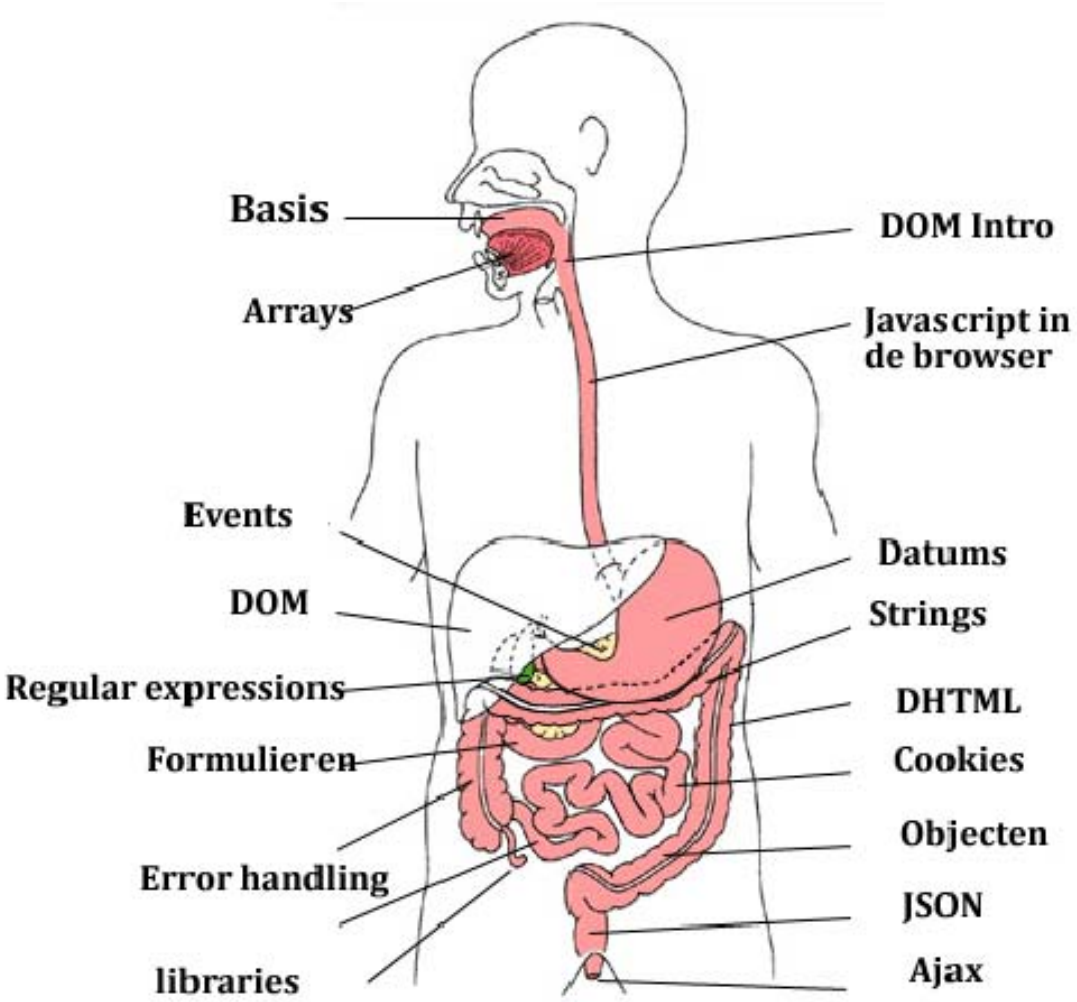
## *2.3 Voorkennis*

Je hebt nu de cursus HTML achter de rug en bent dus vertrouwd met de meest courante HTML elementen, je kent CSS. Je weet dus hoe een html document in elkaar zit, hoe je hyperlinks, lijsten, tabellen, formulieren, etc. correct moet maken. Je weet hoe je een element en een volledig document moet stylen. Daar komen we niet op terug.

## *2.4 De topics*

Via het projectenboek proberen we je op *'het juiste pad '* te leiden doorheen de Javascript *jungle*. Want een jungle is het wel: er zijn veel onderwerpen, de een al interessanter dan de ander en er is ook de vraag wat je eerst moet begrijpen vooraleer je verder kunt gaan, *de kip of het ei?*

Het volgende overzicht toont het juiste traject:




U begrijpt het, laat u glijden en drink af en toe een slokje water. Heb je opmerkingen over de 'flow' of vind je dat een ander/nieuw topic moet opgenomen worden, laat het ons weten via de cursusfeedback.

# 3 Javascript introductie

## 3.1 Een beetje geschiedenis

De taal werd rond 1995 gecreëerd door **Brendan Eich** van Netscape en verscheen eerst in de Netscape Navigator 2.0 browser.

De eerste versie werd *LiveScript* genoemd later veranderd in *Javascript* met als resultaat globale verwarring over naam, eigenaar en mogelijkheden.

*Javascript* heeft niks te maken met *Java*, het heeft een syntax die lijkt op *C*.

Het jaar nadien bracht Microsoft een perfecte kloon op de markt (gemaakt via reverse-engineering) en noemde die *JScript* zodat Sun geen eigendom kon claimen (zoals ook J++ een kloon van Java is). *Javascript* en *JScript* zijn dus exact dezelfde taal met een andere benaming.

Om Microsoft te verhinderen de taal op eigen houtje verder te ontwikkelen en er voordeel uit te halen, vroeg Sun een standaard aan voor de taal bij de ECMA (European Computer Manufacturing Association).

*Javascript* heeft zijn naam tegen:

- Het prefix "*Java-"* suggereert dat Javascript iets te maken heeft met *Java*: ·er zijn infeite zeer weinig overeenkomsten tussen de twee talen
- Het suffix "*–script"* heeft een minderwaardige klank en suggereert dat Javascript geen echte programmeertaal is:
  Javascript is een gesofisticeerde en volwaardige, functionele programmeertaal

Die reputatie kreeg het mee van bij zijn ontstaan door Netscape/Sun. Het is vandaag misschien de meest gebruikte taal ter wereld als we alle platformen in acht nemen waar het gebruikt wordt.

Javascript had vroeger een wat mindere reputatie, maar met de DOM specificatie en *Ajax* is het getij gekeerd! Het lelijke eendje blijkt een mooie zwaan: iedereen houdt nu weer van Javascript!

## 3.2 Javascript versies

Javascript is een cross-platform, obje-gebaseerde scriptingtaal.

De kerntaal versies worden aangeduid als ECMA versies. De voornaamste zijn:

- Javascript1.0 (ECMA-262), de eerste standaardversie (1997)
- Javascript1.5 (ECMA-262, 3ed), veel verbeteringen (1999)
- Javascript1.8.5 (ECMA-262, 5ed), (2010), native JSON support, generator expressions

JS 1.8.5 wordt ondersteund door IE9+, FF4, Chrome, O11.

Minstens even belangrijk als ondersteuning voor de Javascript taal, is de ondersteuning voor de verschillende DOM modules (zie verder).

## 3.3 bouwstenen van Javascript

Javascript is gebaseerd op een aantal sleutelpunten:

- *Load and go delivery*
- *Losse typering*
- *Objecten*

- *Prototypal inheritance*
- *Lambda functies*
- *Linkage d.m.v. global variabelen*

*Load and go delivery*

Andere talen worden geleverd als *executables*, of *class* bestanden, terwijl JS programma's pure tekst zijn. De reden hiervoor is het inbouwen in html pagina's die ook tekst zijn.

*Losse typering*

Variabelen moeten niet strikt gedeclareerd worden en zijn makkelijk om te zetten naar een ander type

*Objecten*

Objecten zijn volledig dynamisch en kunnen op eender welk moment aangepast worden

*Prototypal inheritance*

In tegenstelling tot *class inheritance* kunnen objecten eigenschappen en methodes direct erven van andere objecten. Er zijn geen klassen

*Lambda functions*

Het gebruik van anonieme functions die als waarde kunnen voorkomen in andere expressies

*Linkage d.m.v. global variabelen*

Om onderdelen van een programma met elkaar te laten communiceren, moet je globale variabelen gebruiken. Een niet zo gezond principe…

## 3.4 gezonde programmeerprincipes in javascript

Enkele gezonde programmeerprincipes in JS:

- ***unobtrusiveness***

  Net als met CSS scheiden we **Javascript van de html code**.
  Javascripts bevinden zich in een `script` tag, en nog beter in een extern bestand gekoppeld via een `link` element.
  In de HTML code zelf vinden we
  - **geen** inline eventhandlers zoals `onclick, onmouseover` of `onload`
  - **geen** `<a href: javascript`: …
  - **geen** `document.write`

  Je krabt nu al waarschijnlijk in je haar… *hoe doe je dan een knop werken???*
  In deze cursus leer je hoe je dat moet doen.

  Maar *unobtrusiveness* betekent nog meer, het betekent ook dat een webpagina moet blijven functioneren – beperkt weliswaar - als er geen

Javascript is. Een voorbeeldje: werkt *Google Search* nog als je JS afzet?

Het moet echter gezegd dat Javascript hoe langer hoe meer een prominente rol inneemt en websites er zwaar op steunen.

- ***object (feature) detection***

  Het grote probleem in client-side programmeren is niet Javascript zelf maar de verschillen in ondersteuning door de browsers. Vroeger maakte men **browser-detection** scripts: er werd getest welke browser gebruikt werd, en daaraan werd het script aangepast.

  Browsers groeien echter als onkruid: er duiken telkens nieuwe of nieuwe versies op (Google Chrome net verschenen!) , sommige gebruiken de *'engine'* van een andere browser, je weet nooit zeker welke versie van welke browser een gebruiker bezit.

  De sleutel tot het browser-onafhankelijk programmeren is niet browser-detection maar **object detection**: je test de ondersteuning van een **object** of de **property** of **method** ervan vooraleer je het gebruikt.

  Een veel voorkomende situatie:

```
if(document.getElementById) {
   var anker = document.getElementById('anker');
         if(anker){
                anker.appendChild(inhoud)
         }
  }
```

  Hier wordt *object detection* tweemaal toegepast:

  - eerst testen we of de **method** `getElementById` bestaat (=ondersteund wordt door de browser van de gebruiker), vooraleer ze toe te passen. Let op de afwezigheid van haakjes: we kijken of het `document` object een '*member*' `getElementById` heeft. Pas dan passen we de method toe: `getElementById('anker')`.
  - Daarna kijken we of de variabele anker geen `null` is: dat zou betekenen dat het element 'anker' niet gevonden is.

    De eerste test is een typisch voorbeeld van testen op DOM2 ondersteuning. Je mag er ook van op aan dat als een browser `getElementById` ondersteunt, hij ook zijn broertjes en zusjes zal ondersteunen, zoals `getElementsByTagName`.

    De tweede test wordt regelmatig gebruikt: vooral bij scripts die je op eendere welke pagina wil gebruiken, kan het goede verloop afhangen van de aanwezigheid van een bepaald element.

  Anders is het gesteld met het aanmaken van een `XMLHttpRequest` object in

een Ajax call. Daar weten we pertinent zeker dat IE verschilt van andere browsers:

```
if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
}
else if(window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
}
else {
        throw new Error("XMLHttpRequest niet ondersteund");
 }
```

Ook hier testen we voor de ondersteuning van de objecten, niet van de browser.

- ***graceful degradation***

  Uit het principe van *object-feature detection* volgt dat als het *feature* afwezig is, je een (eventueel minder) alternatief biedt of bij gebrek daaraan, de gebruiker op de hoogte brengt van het probleem. Wat niet aanvaardbaar is, is een 'crash' of het doodbloeden van het programma.

  Een webpagina moet ook **bruikbaar zijn zonder Javascript**, misschien met gereduceerde functies, maar bruikbaar. Fundamentele zaken zoals navigatie mogen daarom niet exclusief afhankelijk zijn van scripting.

  Een speciaal aandachtspunt is het gebruik van muis events: toegankelijheidsregels stellen dat een website bruikbaar moet zijn zonder muis!

Meer specifieke technieken en zelf *design patterns* vind je verder in het betreffende hoofdstuk.

## 3.5 De document tree

*Welke stappen neemt een browser die een HTML document toegestuurd krijgt – bv. van op een webserver – om dit op je scherm te zetten als een webpagina*?

1. de browser **interpreteert** de HTML code. De Engelse term daarvoor is *parsing*.
2. hij bouwt hiermee een ***document tree*** op in zijn intern geheugen: een soort boomstructuur van ***nodes***
3. hij past hierop de **css** style rules toe: de element nodes in de *document tree* krijgen een bepaalde opmaak
4. eventuele **Javascripts** kunnen de *document tree* nog wijzigen
5. vanuit de *document tree* wordt het **scherm** opgebouwd: *rendering*

*Het is dus erg belangrijk te beseffen dat de pagina die je ziet op je scherm niet de HTML code is maar het beeld van de document tree!*

De browser past het **Document Object Model** (DOM) toe om de ***document tree*** te bouwen. De **DOM** is een concept dat voortvloeit uit **XML**: het beschouwt een document als een collectie nodes van verschillend type die in relatie met elkaar staan. Je kan er methods op toe passen en properties van opvragen. Hieronder wordt het *Document Object Model* verder uitgediept.

Javascript (tenminste, de huidige versies) bevat de middelen (DOM methods, properties) om **rechtsreeks** en **op elk moment** de *document tree* te **manipuleren**.
Dat betekent dat de HTML code in de toegestuurde webpagina slechts de basis is en dat eenmaal ingelezen de browser die kan wijzigen naar believen via een javascript. Omdat dat gebeurt in de browser van de gebruiker (en niet in de webserver) noemen we dat *client-side scripting*.

*Javascript toegepast in de browser is een **client-side scriptingtaal***

*Kan je de structuur van een document tree zichtbaar maken?*

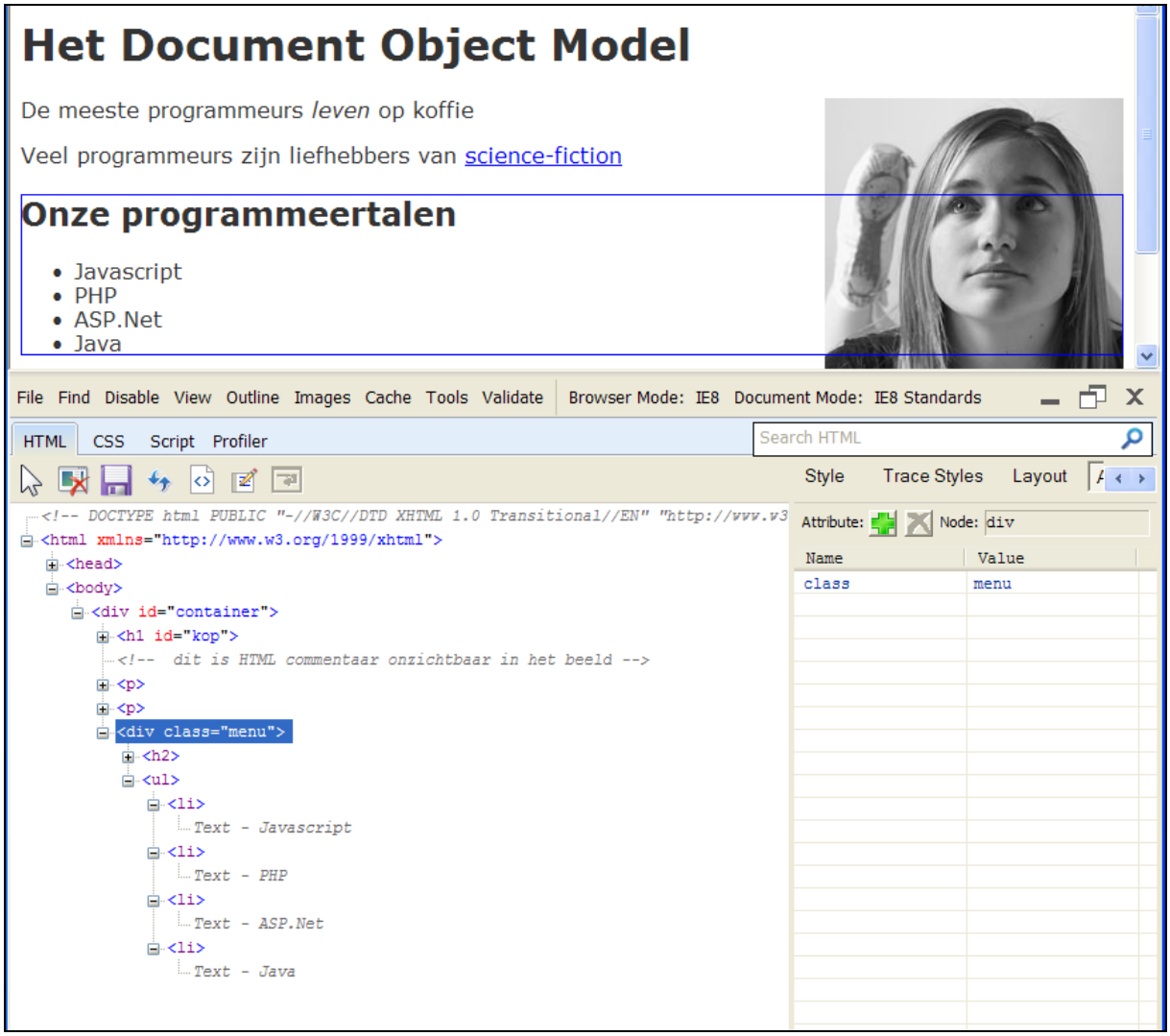
Ja, met een daarvoor geschikt programma, zoals één van de vele plugins van een browser, of via de 'developer tools' die sommige browsers ingebouwd hebben.
Het eerste project van de basiscursus toont je hoe je dat moet doen.

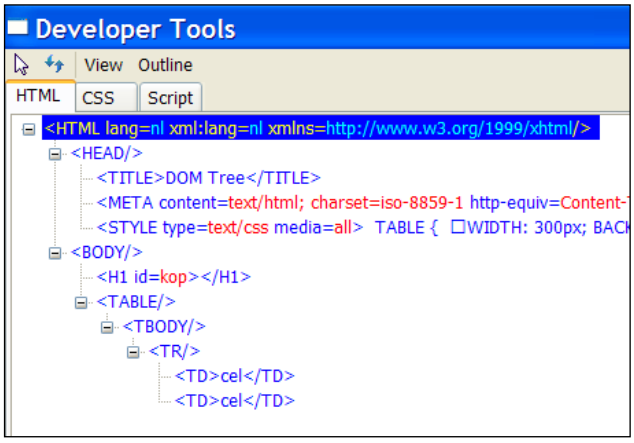
Een voorbeeld: bekijk de HTML code van het volgende document:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>JS basis project: het Document Object Model</title>
<style type="text/css" media="all">
body{
  font-family:Verdana, Geneva, sans-serif;
  color:#333;
  }
img{
  float:right;
  }
</style>
</head>
<body>
<div id="container">
```

```
  <h1 id="kop">Het Document Object Model</h1>
  <!-- dit is HTML commentaar onzichtbaar in het beeld-->
  <p><img src="../images/linda.jpg" alt="Linda, een front-end programmeur" />De
  meeste programmeurs <em>leven</em> op koffie</p>
  <p>Veel programmeurs zijn liefhebbers van <a
  href="http://en.wikipedia.org/wiki/Science-fiction">science-fiction</a></p>
  <div class="menu">
    <h2>Onze programmeertalen</h2>
    <ul>
      <li>Javascript</li>
      <li>PHP</li>
      <li>ASP.Net</li>
      <li>Java</li>
    </ul>
  </div>
</div>
</body>
</html>
```

Bekijken we nu de pagina in de browser samen met zijn *document tree* in de *Webdeveloper's tools* van IE8, dan zien we dit:




aparte nodes weergegeven

Bemerk:

- bovenaan zie je het normale beeld: een titel, twee alinea's, een foto en een lijst
- De *document tree* toont alle **nodes**: *elementNodes* en *textNodes*. Wat deze tool niet toont zijn de textNodes die een nieuw lijnkarakter bevatten. Andere tools tonen dat wel
- html **attributen** zoals `class=menu` worden niet als

## *3.6 Het Document Object Model*

Het **Document Object Model** is een **standaard** om objecten voor te stellen in HTML, XHTML en XML documenten. Het gaat uit van het XML concept van *nodes.*

Deze standaard kan je vinden op de website van het W3C.

Omdat die bij een eerste aanblik nogal overweldigend kan overkomen, kan je ook eerst hier verder lezen…

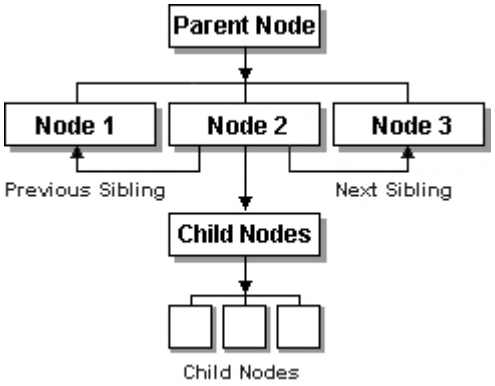
*Neem zeker ook het hoofdstuk "**een XML Primer**" door. Een aantal belangrijke concepten in DOM stammen uit XML.*

Er zijn verschillende **types** nodes. In een HTML pagina zijn van belang:

- **ElementNodes**: dat zijn de HTML elementen.
  Er zijn *lege* nodes (zoals een `img` element) en *elementNodes* met childNodes: dat kunnen andere *elementNodes* zijn of *textNodes*
- **textNodes**: de tekstuele inhoud van een *elementNode* of tekst tussen de elementen

**Nodes** staan in **relatie** met elkaar: er zijn verschillende soorten relaties:

- ***parent-child*** : met uitzondering van het `html` element heeft elke andere *elementNode* een *parentNode*, een ouder.
  Een *elementNode* kan ook `childNodes` hebben, kinderen. Zoals hierboven gezegd kunnen dat andere *elementNodes* zijn of *textNodes*
- ***sibling***: *siblings* zijn nodes met dezelfde parent (broers/zusters)




Een browser gaat in deze vogorde tewerk:

1. hij laadt het document en al zijn bronnen: de HTML, de stylesheets, externe scripts, alle images
2. *parsed* (interpreteert) het en construeert een DOM tree
3. bouwt het scherm op vanuit die DOM tree

Door middel van **methods** en **properties** kan **Javascript** de *document tree* lezen en veranderen, op eender welke moment, ook lang na het inlezen van de htmlpagina.

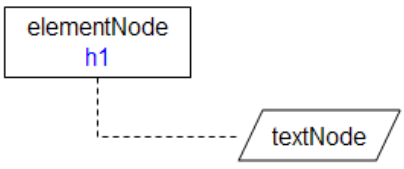
Hieronder geven we een kort overzicht van de **meest gebruikte** DOM methods en properties. Een volledig overzicht vind je verder in het hoofdstuk DOM.

| method | beschrijving |
|---|---|
| `document.getElementById(id)` | legt een referentie naar **1 element** d.m.v. zijn `id` attribuut |
| *`node`*`.getElementsByTagName(tag)` | legt een referentie naar een **collectie** elementen met dezelfde `tagName`. Let op de meervoudsvorm. Let ook op het object, een **elementNode**, dat hoeft de `document` node niet te zijn |
| *`node`*`.querySelectorAll(CSSselector)` | maakt ook een collection aan van nodes, maar d.m.v. een CSS selector. Dit geeft een veel grotere flexibiliteit om sets van elementen aan te maken. Enkel moderne browsers |
| `document.createElement(tag)` | maakt een **elementNode** aan |
| `document.createTextNode()` | maakt een **textNode** aan |
| *`node`*`.appendChild(`*`toeteVoegenNode`*`)` | voegt de *toeteVoegenNode* als **childNode** toe aan de *node* |
| *`node`*`.removeChild(`*`teVerwijderenNode`*`)` | verwijdert de *teVerwijderenNode* die een child is van *node* |
| **property** | **beschrijving** |
| *`collection`*`.length` | het **aantal** items in een collection |
| `childNodes` | de ***collection*** van alle *childNodes* van een node |
| `firstChild, lastChild` | de eerste en de laatste *child* van de `childNodes` |

Nu geven we enkele eenvoudige voorbeelden om het gebruik van deze methods en properties toe te lichten. In het projectboek wordt onderstaand voorbeeld volledig uitgewerkt.

Deze voorbeelden gebruiken het voorbeelddocument van hierboven.

*Ik wil de tekst van de titel (* `h1`*)– met JS wijzigen. Hoe ga ik tewerk?*



- Om de tekst van de titel te kunnen invullen moeten we eerst een **referentie** leggen naar die titel.
  We veronderstellen dat deze titel een `id="kop"` heeft:

```
var titel = document.getElementById('kop');
titel.innerHTML = "De document tree";
```

1) De `document` method `getElementById` refereert een element via zijn `id` attribuut en slaat die referentie op in de variabele *titel*

Merk op:

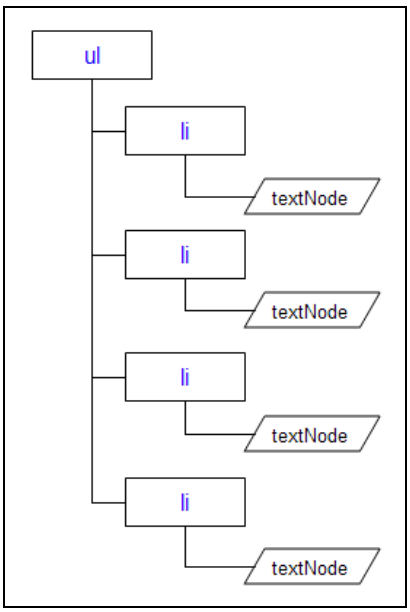
- let op de schrijfwijze: de methods en properties zijn **hoofdlettergevoelig**!
- ook het `id` is hoofdlettergevoelig!
- deze method kan enkel op `document` gebruikt worden en zoekt dus het gehele document af

2) via de method `innerHTML` overschrijven we de inhoud van de titelNode.

Merk op:

- de method `innerHTML` overschrijft de **volledige** inhoud tussen de begin- en eindtag van een element, moest je dit dus toepassen op het `body` element dan vervang je gehele inhoud van de pagina… De tekst van de titel is nu ingevuld.
- Je kan er dus meer dan tekst mee ingeven, ook een volledige HTML structuur
- is geen echte DOM method vermits hij geen rekening houdt met nodes

*Hoe wijzig ik de tekst van het tweede lijst-item?*




```
var alleBolletjes      = document.getElementsByTagName('li');
var aantalBolletjes    = alleBolletjes.length;
var tweedeItem         = alleBolletjes[1];
```

```
tweedeItem.removeChild(tweedeItem.childNodes[0]);
var tekst = document.createTextNode('C#');
tweedeItem.appendChild(tekst);
```

Omdat dit `ul` element geen `id` heeft gaan we anders tewerk.

- De method `getElementsByTagName` maakt een **collection**: de verzameling van alle elementen van het type `li` in het document.
- De `length` property vertelt ons hoeveel items er in deze collectie zijn.
- We refereren het tweede item door de index 1 te gebruiken: een index begint te tellen vanaf 0.
- We verwijderen eerste de aanwezige *textNode* met de method `removeChild` die we de eerste (en enige) *childNode* meegegeven.
- We maken een nieuwe *textNode* aan die we opvullen met tekst en die we invoegen in het `li` element.

In het tweede bolletje is de tekst nu gewijzigd.

*Hoe voeg ik een nieuw item toe achteraan de lijst?*

```
var lijst = document.getElementsByTagName('ul')[0];
var nieuwItem = document.createElement('li');
var tekst = document.createTextNode('Perl');
nieuwItem.appendChild(tekst);
lijst.appendChild(nieuwItem);
```

Bespreking:

- we refereren de lijst met `document.getElementsByTagName('ul')`: ook hier maakt `getElementsByTagName` een collection van **alle** `ul` elementen in de pagina, dus gebruiken we de array index `[0]` om aan de eerste `ul` aan te duiden
- nu maken we een nieuw `li` element aan met `document.createElement('li')`
- we maken ook een *textNode*
- die we daarna toevoegen aan het `li` element
- het `li` element wordt achteraan toegevoegd

De lijst heeft nu een nieuwe list-item met inhoud.

Veel meer over het Document Object Model vind je in het volledige hoofdstuk erover verderop in deze cursus.

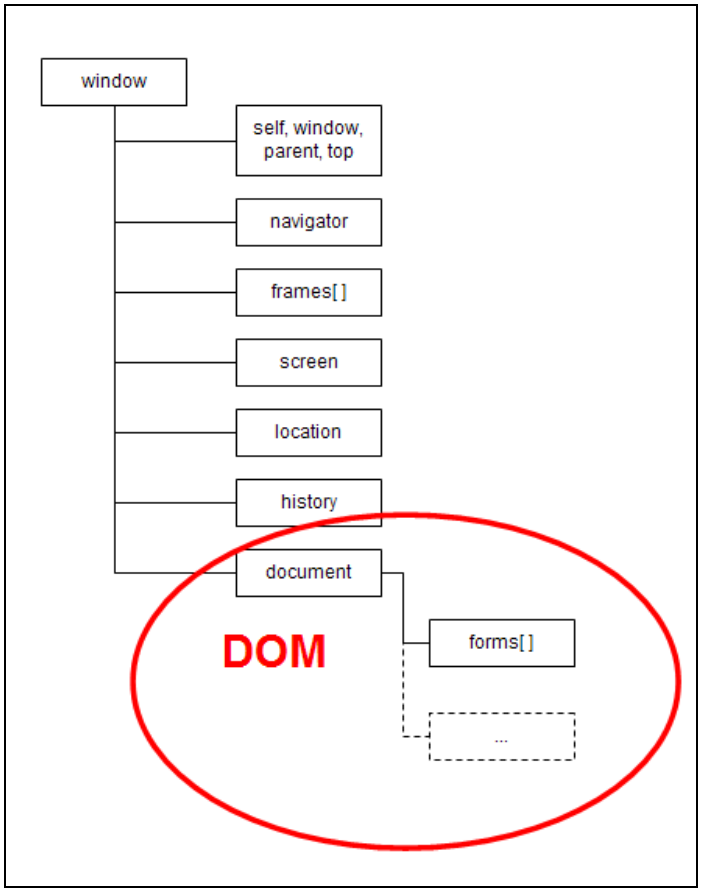
## 3.7 Het Browser Object Model

Het *browser object model* is een onofficiële term waarmee men de objecten bedoelt die verbonden zijn aan de **browser zelf**, niet aan het document

Met andere woorden, deze objecten die eigen zijn aan het browservenster, het `window` object. Zo is er bijvoorbeeld de `window.history` eigenschap, die bijhoudt welke url's het venster al bevat heeft.

Eén van de *children* van dit globale `window` is het `document`, het html-document geladen in dit venster.

Alles wat spruit uit `window.document` noemt men het *Document Object Model* of DOM.




Het `window` object is in Client-side Javascript het **globale object**, dus zijn alle andere objecten eigenschappen ervan. Zo is het `navigator` object een `property` van window:

```
window.navigator
```

Zo ook `window.history`, `window.location`, `window.screen` en `window.document`. Deze objecten hebben dan weer eigen properties en methods.

Het `window.document` heeft bijvoorbeeld een array van forms objecten, waarvan we de eerste form omschrijven als `window.document.forms[0].`

Als er geen verwarring bestaat, mag je de referentie naar het `window` object achterwege laten: `document.forms[0]`.

## *3.8 Javascript, BOM en DOM*

Je zult deze termen naast en door elkaar heen gebruikt zien worden.
Betekenen ze hetzelfde? We scheppen wat duidelijkheid:

- **Javascript**, de **taal** zelf: syntax, ingebouwde objecten en structuren, bijvoorbeeld het `Number` data type, het `Date` object, de `for` loop.

- het **Browser Object Model** (**BOM**), de objecten die te maken hebben met de **browser** als applicatie. Bv. het `window` object
- het **Document Object Model (DOM)** is de *document tree* van objecten van een **document**

Onderstaand voorbeeld demonstreert de vermenging van DOM, BOM en Javascript in een stukje code:

```
var anchorTags = document.getElementsByTagName("a");
for (var i = 0; i < anchorTags.length ; i++)
{
   alert("Href van dit a element is : " + anchorTags[i].href.toLowerCase() + "\n");
}
alert("uw browser is " + navigator.userAgent);
```

- Tekst in *schuinschrift* (rood) in dit codevoorbeeld behoort tot de **DOM**: het zijn methods of properties van de DOM.
  De *alert()* method is infeite een oude DOM0 method van het `window` object.
  De method *getElementsByTagName()* is een DOM2 method
- `navigator` (groen) is een browserobject dus behoort tot de **BOM**
- de rest (blauw) is **Javascript**. Bijvoorbeeld de method toLowerCase() zet een tekst om naar kleine letters

# 4 Core Javascript

Javascript als taal.
Wat hier volgt is de **naslag** van de syntax, structuren en de belangrijkste objecten.
Voor praktische toepassingen verwijzen we je naar het projectenboek.

## *4.1 Basisregels*

- JS is **Hoofdlettergevoelig**:
  de variabele *objTekstBox* is verschillend van *objtekstbox* en *OBJtekstbox*...
  de methode `Focus()` zal niet werken, `focus()` daarentegen wel
- JS negeert witruimte (spaties en tabs)
- lange statements kunnen afgebroken worden **na** een operator of een komma. Indenteer de vervolgregel:

```
var a = b +
        c;

var test = document.
        getElementById('test');
```

- elke statement afsluiten met een **puntkomma** wordt **sterk aangeraden**;
- **Commentaar** kan op twee manieren:
  - gebruik `//` voor commentaar op 1 lijn
  - gebruik `/* */` voor commentaar gespreid over meerdere lijnen
- **Escape sequences**: het backslash karakter (\) gevolgd door een karakter zorgt ervoor dat dat karakter steeds letterlijk genomen wordt en niet geïnterpreteerd door de parser. We noemen dit een **escape sequence** .
  Bijvoorbeeld: onderstaande stringvariabele bevat de aanhalingstekens letterlijk

```
var strTekst = "en toen zei ze \"oei, het is gevallen\"";
```

- Sommige speciale statements worden ook met een escape sequence weergegeven:

  \b backspace

  \r nieuwe regel

  \n nieuwe lijn

  \t tabulatie

- **Identifiers** zijn de *termen* die je mag gebruiken voor variabelen en functies:

```
var getal = 5
var $inhoud = document.getElementById('inhoud')
function maakBullet()
```

  ze mogen beginnen met een letter, een underscore _ , of een dollarteken $.
  ze mogen **niet** beginnen met een getal
- bij **conventie** beginnen alle variabelennamen, functienamen, parameters met een kleine letter, enkel Constructors krijgen een hoofdletter

- **Sleutelwoorden** mag je niet gebruiken als variabele of functienaam.
  Er zijn er heel wat: `break, do, this, final`, ...
  alle sleutelwoorden worden in kleine letters geschreven

## 4.2 Data types

### 4.2.1 Eenvoudige data types

JS heeft een beperkt aantal eenvoudige datatypes en enkele waarden:

- Number
- String
- Boolean
- *null*
- *undefined*

alle andere waarden zijn objecten

#### 4.2.1.1 Number

Het **Number** type is het enige getal-datatype in JS: alle getallen opgeslagen worden als 64-bit *floating-point* getallen (=*double* in C++). Er is geen apart *integer*type of dergelijke.

#### 4.2.1.2 NaN

*"**N**ot **a** **N**umber".*

Is het resultaat van een foutieve of *undefined* bewerking. Bijvoorbeeld als je een getal deelt door nul.
`NaN` kan niet vergeleken worden met iets anders, inclusief zichzelf, dus moet je gebruik maken van de speciale functie `isNaN()` om deze waarde te detecteren.
`NaN` is *giftig*: als `NaN` voorkomt in een bewerking, wordt het resultaat ook `NaN`:

```
NaN + 3 * 2 = NaN
```

#### 4.2.1.3 Het Math object

Om verdere berekeningen te doen met een getal kan je gebruik maken van de functies van het aparte `Math` object (onvolledig):

| functie | return waarde |
|---|---|
| `abs` | absolute waarde |
| `floor` | integer |
| `log` | logaritme |
| `max` | maximum |
| `pow` | macht |
| `random` | willekeurig getal |
| `round` | dichtstbijzijnde integer |
| `sin` | sinus |

| functie | return waarde |
|---|---|
| `sqrt` | vierkantswortel |

#### 4.2.1.4 String

Een **String** is een tekst van 0 of meer karakters, opgeslagen als 16-bit Unicode tekens.

Strings worden zowel met **enkele** als met **dubbele aanhalingstekens** geschreven:

```
"string" == 'string' //true
```

#### 4.2.1.5 Escape

Om ervoor te zorgen dat bepaalde karakters niet geïnterpreteerd worden gebruiken we de **backslash** (\):

```
's\'Avonds als het regent'
```

Ook om speciale tekens voor te stellen zoals een nieuwe regel karakter, gebruiken we een *escape sequence*. Enkele voorbeelden:

| functie | return waarde |
|---|---|
| `\n` | newline |
| `\t` | tab |
| `\\` | backslash |

#### 4.2.1.6 String methods

| method | return waarde |
|---|---|
| `charAt` | karakter op de indexpositie *n* |
| `concat` | combineert meerdere strings in één nieuwe string |
| `indexOf` | geeft de index van het *eerste* karakter van het *eerste voorkomen* van een *zoekString* in een andere string. |
| `lastIndexOf` | geeft de index van het *eerste* karakter van het *laatste voorkomen* van een *zoekString* in een andere string. |
| `match` | Zoekt de opgegeven regular expression en returnt een array: |
| `replace` | Zoekt de opgegeven regular expression en vervangt die door newString. |
| `search` | Zoekt de opgegeven regular expression en returnt de *positie* van de opgegeven regexp. |

| method | return waarde |
|---|---|
| `slice` | returnt een nieuwe string met het stuk beginnend vanaf de *startIndex* positie en tot - maar niet inclusief - een eventuele *stopIndex* positie. |
| `split` | splitst een string in een *array* van strings, daarbij gebruik makend van het *separator* karakter. |
| `substring` | returnt een gedeelte van de oorspronkelijke string, beginnend vanaf *startIndex* positie, tot - maar niet inclusief - de *stopIndex* positie. |
| `toLowerCase` | geeft de string in kleine letters |
| `toUpperCase` | geeft de string in hoofdletters |

#### 4.2.1.7 Boolean

Javascript kent de waarden `true` en `false`.

Er is ook de functie `Boolean()` die één van bovenstaande returnt.

```
alert (Boolean(document.getElementById)) // browser?
```

Een aantal waarden leveren altijd `false` op:

```
var waarde = Boolean(0) // getal nul
var waarde = Boolean('') // lege string
var waarde = Boolean(null) // null
var waarde = Boolean(undefined) // afwezige waarde
var waarde = Boolean(false) // de waarde false
var waarde = Boolean(NaN) // de waarde NaN
var waarde = Boolean('') // lege string
```

Alle andere waarden, inclusief objecten, leveren `true` op

#### 4.2.1.8 null

De speciale waarde `null` staat voor *geen object*. Als een variabele de waarde `null` heeft, weet je dat het een ongeldig of niet bestaand object bevat.

```
var element = document.getElementById('iets')
//returnt null als 'iets' niet bestaat
```

#### 4.2.1.9 undefined

`undefined` is de standaardwaarde voor variabelen die gedeclareerd worden, maar verder geen waarde toegekend krijgen.

```
var waarde
  alert(waarde) //undefined
```

Het is ook de waarde die je krijgt voor ontbrekende *members* (*properties, methods*) van objecten.

```
(document.getElementById) //returnt object
  (document.getJeanSmitsByTelephone) //undefined
```

### 4.2.2 Objecten

Objecten zijn in Javascript verzamelingen **naam-waarde** paren.

Je kan zelf objecten aanmaken of ze afleiden van bestaande Javascript objecten. Daarnaast zijn *Functies* ook objecten, *Arrays* zijn objecten, *regular expressions* zijn objecten. Ze stammen infeite allemaal af van het **Object** object.

Een **object** is een **verzameling eigenschappen (**properties ) die elk een **naam** en een **waarde** hebben, een soort associatief array dus. Er zijn geen beperkingen op de naamgeving van de eigenschap of aan de waarde die het kan bevatten: properties kunnen eender welke JS waarde bevatten inclusief andere objecten.
Object **methods** zijn properties die naar een **functie** verwijzen.

Je kan je eigen objecten aanmaken of je kunt gebruik maken van de ingebouwde objecten: `Error, Math, Date, Regular Expressions`, etc..

Javascript objecten zijn heel handig om een verzameling waarden, instellingen, op te slaan. Omdat ze andere objecten kunnen bevatten zijn ze ideaal om hiërarchische structuren - *trees* - weer te geven.

Hier bespreken we het eenvoudig aanmaken en gebruiken van een object.

Een object kan je aanmaken

- als **object literal** (letterlijk)
- afgeleid van een ander object, of de `prototype` van een andere object

De snelste manier om een object te creëren is als **object literal**:

```
var boek = {
        titel: "Javascript, een vdab handleiding",
        auteur:"Jean Smits"
}
```

De *object literal* syntax gebruikt **accolades** `{ }` om alle **eigenschappen** (literal : value) in te vatten en **komma's** als scheidingsteken.
Merk op dat het onnodig is een dergelijk object toe te wijzen aan een variabele:

```
{titel: "Javascript, een vdab handleiding", auteur :"Jean Smits"}
```

De andere manier om dit object aan te maken is met een **constructor**:

```
var boek = new Object();
boek.titel: "Javascript, een vdab handleiding";
```

```
boek.auteur:"Jean Smits";
```

Het eerste statement maakt een algemeen object aan met het `new` sleutelwoord. Eenmaal gemaakt, kan je het *properties* meegeven met de dot-notatie.
Deze manier wordt meestal gebruikt als je een object wil afleiden van een bestaand object, zoals een `Date()`:

```
var vandaag = new Date();
```

omdat je hiermee de specifieke eigenschappen van een datum erft.

#### 4.2.2.1 properties

Een **object** in JS heeft enkel **properties**. Maar als een property verwijst naar een functie, dan noemen het een **method**.
Properties en methods worden ook wel eens **members** van een object genoemd.

De **naam** van een property is steeds een `string`, de **waarde** van de eigenschap is **eender welk datatype**, inclusief een ander object of een functie.

Om een eigenschap van een object aan te spreken hebben we twee mogelijkheden:

- de **dot . notatie**: `object.property`
- de **sleutel notatie**: `object["property"]`

Ze zijn gelijkwaardig.

De sleutel notatie heeft het voordeel dat de property name als `string` meegegeven wordt en dat je die dus programmatorisch kunt berekenen at run-time, bijvoorbeeld in een lus, of via een variabele.

Het is voldoende een naam=waarde paar aan het object te koppelen om een nieuwe eigenschap aan te maken, een `var` keyword is onnodig:

```
var boek = new Object();
boek.titel = "Javascript, een vdab handleiding";
boek.auteurs = new Object();
boek.auteurs.auteur1 = "Jean Smits";
boek.auteurs.auteur2 = "Jan Vandorpe";
boek["auteurs"]["auteur3"] = "Jules Vernes";
boek.publicatieDatum = new Date(2008, 07, 30);
boek.aantalPaginas;
boek.commentaren = [“goed”,”slecht”,”wazegde?”];
boek.uitgever = undefined;
boek['editie'] = "hardcover";
```

Een eigenschap die geen waarde toegekend krijgt heeft de waarde `undefined`.

De letterlijke notatie is evenwaardig (en korter...):

```
var boek = {
  titel:"Javascript, een vdab handleiding",
  auteurs: {
          auteur1: "Jean Smits",
          auteur2: "Jan Vandorpe"
  },
  publicatieDatum: new Date(2008, 07, 30),
  aantalPaginas,
  commentaren : [“goed”,”slecht”,”wazegde?”],
  uitgever : undefined,
  editie: "hardcover"
}
```

Om alle eigenschappen van een object te doorlopen gebruiken we een `for in` loop waarbij we van de sleutel notatie gebruik maken:

```
for (var key in obj ){
          console.log( key + ": " + obj[key])
}
```

Opmerkingen:

- deze lus (*enumeratie*) zal `boek.aantalPaginas` niet tonen wegens `undefined`
- de `for` loop maakt gebruik van de `hasOwnProperty` property. Dit om te vermijden dat ook alle eigenschappen van een eventueel *parent object* (waarvan dit object afgeleid is) getoond worden.
  In ons voorbeeld speelt dat echter geen rol.

Een property kan verwijderd worden met de `delete` operator:

```
delete boek.uitgever;
```

Vanaf nu zal deze `undefined` opleveren.

#### 4.2.2.2 methods

Een **method** van een object is een **property** die verwijst naar een `function`.
Dat kan in de property zelf - als **anonieme functie** - of als verwijzing naar een benoemde functie:

```
function Planeet (naam, diameter, afstand, jaar, dag){
  //properties
  ...
  //methods
  this.volume = function(){
```

```
        //anonieme functie
        return (4 / 3) * Math.PI * Math.pow(this.diameter/2 , 3)
  }
  this.toon = toonPlaneet;
        //verwijzing naar benoemde fucntie
}


function toonPlaneet(){
  strPlaneet = "planeet: " + this.naam + "\n";
  strPlaneet += "diameter: " + this.diameter + "\n";
  strPlaneet += "afstand: " + this.afstand + "\n";
  strPlaneet += "jaar: " + this.jaar + "\n";
  strPlaneet += "dag: " + this.dag;
  strPlaneet += "leefbaar: " + this.leefbaar;
  return strPlaneet;
}
```

In dit voorbeeld maakt de method `volume` gebruik van een anonieme functie, terwijl de method `toon` verwijst naar een benoemde functie. Merk ook de verwijzing naar de externe functie op: **geen haakjes**, want de property `toon` **refereert** nu naar toonPlaneet() en bevat niet het resultaat van toonPlaneet().

De functies refereren naar het object zelf via het `this` sleutelwoord.

Om de method toe te passen gebruiken we dus de **naam van de property aangevuld met een paar haakjes**, toegepast vanuit het object:

```
alert(mars.volume())
alert(wereld.toon())
```

### 4.2.3 Arrays

Een Array is een tabelvariabele, ideaal om lijsten gegevens bij te houden.

*- gebruik een **array** als je de elementen wil opzoeken met een **getal** (index), bijvoorbeeld `namen[2]`*
*- gebruik een **object** als je iets wil opzoeken a.h.v. **woord** (key), zoals in `personen['naam']`*

Je kan een array aanmaken op twee manieren:

- als *array literal*

- met een *constructor*

Als je kan, gebruik de letterlijke manier:

```
var tabel = [2,"javascript",false,];
```

Het array wordt direct samengesteld uit waarden gescheiden door **komma's en vervat** in **vierkante haakjes**. Dit voorbeeld bevat 4 waarden waarvan de laatste een `undefined` is, omdat er nog een laatste komma blijft. NB: Als deze laatste komma een foutje is van uwentwege, dan moet je oppassen: sommige browsers (FF) zullen die waarde negeren, terwijl andere (IE) hem zullen beschouwen als een geldige waarde.

De andere wijze is een nieuw array afleiden van de functie `Array()` met het `new` sleutelwoord:

```
var tabel = new Array();
tabel[0] = 2;
tabel[1] = "javascript";
tabel[2] = false;
tabel[99] = undefined;
```

Eenmaal het array aangemaakt kan je waarden toekennen aan individuele elementen via hun index.
Het is ook mogelijk (maar niet nodig) het array te dimensioneren met het aantal elementen in de constructor.

```
var tabel = new Array(4);
```

Dit is onnodig want JS arrays zijn volledig dynamisch: je kunt er altijd een element aan toevoegen of uit wegnemen.

#### 4.2.3.1.1 array elementen

Het aantal elementen in een array wordt gegeven door de property `length`.

Om één enkel element te bereiken gebruiken we de index in *vierkante haakjes* [ ].
Het eerste element heeft index `0`, het laatste een index = `length-1`.

```
var tabel = new Array();
// toewijzing
tabel[0]  = 2;
tabel[1]  = "javascript";
tabel[2]  = false;
tabel[99] = undefined; // indexgetallen moeten geen continue reeks vormen

// aflezen:

tabel.length; // returnt 100
```

```
tabel[0]; // returnt 2
tabel[tabel.length-1]; // returnt undefined
```

Om alle elementen in een array te doorlopen wordt een `for` loop gebruikt, vanaf het 0 element tot array.`length` -1:

```
for (var i=0;i<tabel.length;i++){
        console.log(tabel[i]);
 }
```

### 4.2.4 Datatype conversie

#### 4.2.4.1 Omzetten van strings naar getallen

##### 4.2.4.1.1 `Number()`

De functie (en constructor) `Number()` voert een expliciete conversie uit:

```
var getal = Number('3.5') // returnt 3.5
var probleemgetal = Number('3.5 witte muizen') // returnt NaN
```

Bij enig probleem returnt de functie `NaN`.

`Number()` doet enkel base10 conversies. Spaties voor en na het getal worden geaccepteerd.

##### 4.2.4.1.2 `parseInt()`, `parseFloat()`

Deze twee functies zetten een string om naar een getal, `parseInt()` naar een geheel getal, `parseFloat()` zowel naar integers als naar floating point getallen.

Beide functies stoppen bij het eerste niet-numerieke teken dat ze tegenkomen.

```
var geheel = parseInt('3.5 witte muizen') // return 3
var float = parseFloat('3.5 witte muizen') // return 3.5
```

Er moet wel opgelet worden met strings die beginnen met een `0` of `0x`: dergelijke waarden worden als octaal of hexadecimaal geïnterpreteerd. Om dit te vermijden moet je het *radix* argument gebruiken:

```
var base8 = parseInt(08) // return 0
var base10 = parseInt(08, 10) // return 8
var hexwaarde = parseInt('0xFF', 16) // return 255
```

## *4.3 Expressies en Operatoren*

Een **expressie** is een JS statement dat **geëvalueerd wordt tot een resultaat**.

Een expressie die bestaat uit een letterlijke waarde is die waarde zelf, maar waarden kunnen ook vergeleken worden of er kan een bewerking mee gedaan worden.
Voorbeelden van expressies:

```
"javascript is OK"      // een letterlijke expressie
tekst.length    // het aantal karakters in een string
i++      // de variabele i verhogen met 1
i > j    // twee variabelen vergelijken
```

Expressies maken gebruik van **operatoren**:

## *4.4 Operators*

Een operator wijzigt, test of vergelijkt een waarde:

| Operator | voorbeeld | betekenis |
|---|---|---|
| **toewijzingsoperatoren** | | |
| = | x = y | normale toewijzing: x krijgt de waarde y |
| += | x += y | toevoeging: x wordt verhoogd met y<br>idem als x = x + y |
| -= | x -= y | verschil: x wordt verlaagd met y<br>idem als x = x - y |
| *= | x *= y | toevoeging: x wordt vermenigvuldig met y<br>idem als x = x * y |
| /= | x /= y | toevoeging: x wordt gedeeld door y<br>idem als x = x / y |
| **berekeningsoperatoren** | | |
| + | x = y + z | optelling: x krijgt de waarde y plus z<br><br>zie opmerkingen |
| - | x = y - z | verschil: x krijgt de waarde y min z |
| * | x = y * z | product: x krijgt de waarde y maal z |
| / | x = y / z | quotiënt: x krijgt de waarde y gedeeld door z |
| % | x = y % z | modulo: x krijgt als waarde de rest van de deling y / z |
| ++ | x++<br><br>++x | increment: x heeft nu de waarde x maar wordt nadien met 1 verhoogd<br>x wordt eerst met 1 verhoogd en heeft dus de nieuwe waarde |
| -- | x--<br><br>--x | decrement: x heeft nu de waarde x maar wordt nadien met 1 verlaagd<br>x wordt eerst met 1 verlaagd en heeft dus de nieuwe waarde |
| - | x = -x | negatie: x krijgt de negatieve waarde x |
| **vergelijkingsoperatoren** | | |
| == | 3 == "3" | `true` indien **gelijk**, niet noodzakelijk van |

| | | |
|---|---|---|
| | | hetzelfde datatype, anders `false`<br>Hier `true` |
| `===` | 3 === "3" | `true` indien **gelijk**, en van hetzelfde **datatype**, anders `false`<br>Hier `false` |
| `!=` | "3" != 3 | `true` indien **ongelijk**, een conversie van het datatype wordt geprobeerd, anders `false`<br>Hier `false` |
| `!==` | 3 !== "4" | `true` indien **ongelijk** en van een verschillend datatype, anders `false`<br>Hier `true` |
| `>` | x > y | `true` indien x **groter** is dan y, anders `false` |
| `<` | x < y | `true` indien x **kleiner** is dan y, anders `false` |
| `>=` | x >= y | `true` indien x **groter of gelijk** is dan y, anders `false` |
| `<=` | x <= y | `true` indien x **kleiner of gelijk** is aan y, anders `false` |
| **logische operatoren** | | |
| `&&` | expr1 && expr2 | logische **AND**: `true` als beide expressies `true` zijn |
| `\|\|` | expr1 \|\| expr2 | logische **OR**: `true` als één van beide expressies `true` is |
| `!` | !expr1 | logische **NOT**: `true` als expr1 `false` is.<br>keert de expressie om |
| **string operatoren** | | |
| `+` | "hello " + "world" | concatenatie:<br>twee string variabelen worden aan elkaar gelast.<br>zie opmerkingen |
| `+=` | x += "world" | korte schrijfwijze voor de concatenering<br>x = x + "world" |
| **andere** | | |
| `()` | alert() | `call` operator. Voert een functie uit |
| `in` | x in y | `in` operator:<br>gebruikt om het bestaan van eigenschappen te testen<br>`true` als x een *property* is van het *object* of *array* y |
| `instanceof` | x instanceof y | `instanceof` operator:<br>om te controleren of een objectvariabele een afgeleide instance is van een bepaald objecttype.<br>Zie objecten |
| `new` | x = new Object() | `new` operator:<br>om een nieuwe instance van een object |

| | | |
|---|---|---|
| | | variabele aan te maken |
| `typeof` | (typeof(x) =="string) | `typeof` operator:<br>returnt het type variabele. Mogelijke returnwaarden zijn "number", "string", "boolean", "object", "function", "undefined" |
| `void` | void window.open() | `void` operator:<br>wordt gebruikt om een returnwaarde op `undefined` te zetten, zodat de expressie niets teruggeeft.<br>Typisch bij een javascript statement in de `href` van een `a` element |
| `delete` | delete object.property | `delete` operator:<br>verwijdert een property van een object |
| `? :` | x = (expr) ? truewaarde : falsewaarde | `ternary` operator:<br>een verkorte versie van een `if` statement. Als de *expr* `true` evalueert krijgt de variabele x de *truewaarde* toegewezen, zoniet de *falsewaarde* |

Opmerkingen:

- de `+` operator kan zowel als optelling als **concatenator** gebruikt worden. Hij geeft steeds de voorkeur aan strings. Dat betekent dat als één variabele een string is, het resultaat ook een string zal zijn.
  Dit gebeurt typisch bij het optellen van twee getallen die ingevuld worden in een html-formulier: invulvelden returnen altijd een stringwaarde zodat

  `1 + 1` de stringwaarde `11` zal opleveren

  Om de rekenkundige waarde te krijgen moeten de waarden eerst in getallen omgezet worden (zie datatype conversie)
- Bitwise operators hebben we hier niet behandeld

### 4.4.1 eval()

`eval()` is een functie van het globale object (het `window` object) die een string die Javascript code bevat compileert, dus interpreteert en probeert uit te voeren

```
eval("3 * 4") // returnt 7


var x = 6;
eval("if(x>7) {alert('x groter dan 7')} else {z = 0 }") // z = 0


var o = eval("{naam:'jan'}") // geeft een object
```

***eval() is Evil!***

Probeer `eval()` zoveel mogelijk te vermijden: kwaadaardige code in tekstvorm kan erdoor uitgevoerd worden met de volledige permissies van de webgebruiker.

`eval()` werd vroeger regelmatig gebruikt om JSON objecten aan te maken, tegenwoordig gebruikt men daarvoor een gespecicialiseerde library.

## *4.5 Variabelen*

Een **variabele** is een naam die een waarde krijgt. Een variabele laat je programma toe gegevens tijdelijk op te slaan en te wijzigen.

De variabele krijgt een **waarde** of het **resultaat van een bewerking** toegewezen.

In JS is het ook mogelijk een functie aan een variabele toe te wijzen.

```
i = 0;
var totaal      = 23 * 7;
var flag        = true;
var voorlopig   = undefined;
var lijst       = ['jan','piet','pol'];
var verwijder   = function(){ this.parentNode.removeChild(this)}
```

### 4.5.1 Typing

De term ***typing*** slaat op het **datatype** van een variabele: is het een getal, een tekst, een boolean?

Javascript is een *'los getypeerde*' taal. Javascript kent datatypes maar ze worden niet streng toegepast.
In tegenstelling tot 'sterk getypeerde' talen zoals C++ of Java zijn een aantal dingen mogelijk:

- Een variabele **moet geen datatype** toegekend worden tijdens de **declaratie**, dus kan je er eender welke waarde aan toekennen
- **impliciete omzetting** van sommige datatypes in andere types gaat heel makkelijk (voor zover mogelijk)
- een variabele die een getal toegekend gekregen heeft, kan later zonder problemen een string toegekend krijgen

```
i = 10;
i = "tien"
i += 5 // geeft "tien5"
```

Als er twijfel bestaat over welk type een variabele is, gebruik de `typeof` operator:

```
alert(typeof i)
```

### 4.5.2 Naamconventies voor variabelen

Eerst en vooral willen we duidelijk stellen dat *identifiers* (variabelen en gereserveerde woorden) **hoofdlettergevoelig** zijn!

`getal ≠ Getal ≠ geTal ≠ GETAL`

Omdat Javascript **geen getypeerde variabelen** kent, en een variabele dus eender wat kan bevatten, op elk moment kan veranderen van inhoud, is het niet steeds duidelijk wat het nu eigenlijk *hoort* te bevatten...

Een veel voorkomend voorbeeld illustreert dit:

```
var leeftijd    = document.getElementById('leeftijd');
```

De var *leeftijd* laat uitschijnen dat deze een getal zal bevatten. Dat is zeker niet de waarheid want *leeftijd* is infeite een DOM element. Juister zou zijn

```
var veld        = document.getElementById('leeftijd');
var leeftijd    = veld.value
```

Beter, maar we zullen nog steeds niet zeker zijn want de gebruiker kan een tekst in het veld typen.

Om de duidelijkheid van onze scripts te verhogen, gebruiken we schrijfconventies.

Een aantal schrijfconventies zijn een "Must":

- variabelen beginnen steeds met een kleine letter: `leeftijd`, `nLeeftijd`
  - enkel constructorfuncties (gebruikt met het `new` keyword) worden met een hoofdletter geschreven, gewone functies niet: `function Inhoudstafel()`
- variabelen beginnen nooit met een getal
- voor lange variabele namen kies je één van deze manieren om woorden samen te stellen:
  - CamelCase: `evenRijen`
  - met een underscore: `even_rijen`

Een aantal conventies zijn geen verplichting, maar maken je scripts beter leesbaar:

- globale variabelen worden volledig in hoofdletters geschreven: `STARTDATUM`, `PI`
- gebruik **prefixes** om gekende datatypes aan te geven.

  Sommigen gebruiken 1 letter:

| | | |
|---|---|---|
| `n` | Number | `nLeeftijd` |
| `s` | String | `sInhoud` |
| `b` | Boolean | `bAntwoord` |
| `o` | Object | `oData` |
| `a` | Array | `aWinkels` |

| | | |
|---|---|---|
| `e` | DOM element | `eRij` |

Dit laat je ook toe om meer aan te geven:

`aaData` is een array van subarrays, terwijl `aoData` een array van objecten is.

Anderen gebruiken 3 letters:

| | | |
|---|---|---|
| `int` | Integer | `intLeeftijd` |
| `str` | String | `strInhoud` |
| `boo` | Boolean | `booAntwoord` |
| `obj` | Object | `objData` |
| `arr` | Array | `arrWinkels` |
| `elm` | DOM element | `elmRij` |

Speciale variabelen kan je natuurlijk altijd benoemen zoals je zelf wil, zolang je maar **duidelijk** overkomt. Bijvoorbeeld:

```
jsonPersoon = {
                "naam":"jan",
                "leeftijd":25
}
```

Het is hier inderdaad duidelijk dat *jsonPersoon* een JSON object is …

Javascript libraries zoals Prototype en jQuery maken een eigen "wrapper" voor elementen met een `$`-teken. Het is perfect geldig om in javascript een variabele te laten voorafgaan door een dollarteken om aan te geven dat dit geen gewoon DOM element is:

```
$diefs = $('div');
```

### 4.5.3 Declaratie

Om een variabele te gebruiken moet je hem **declareren** met het `var` sleutelwoord:

```
var i;
var nTotaal;
var sVoornaam;
```

je **kan** dit combineren met het toekennen van een waarde (**initialiseren**):

```
var i=0;
var nTotaal = i+55;
var sVoornaam = document.form0.voornaam.value;
```

als je een variabele gebruikt die niet gedeclareerd is, krijg je een **runtime error** en het JS programma stopt.

```
var nTotaal = j+55; // geeft een runtime error, want j is niet gedeclareerd
```

Als je een variabele geen waarde meegeeft bij de declaratie, dus niet initialiseert - zal zijn oorspronkelijke waarde `undefined` zijn.

```
var i;
var nTotaal = i+55; // geeft NaN want i is undefined, maar geen foutmelding
var t;
var sTekst = "tekst" + t       //geeft "tekstundefined", maar geen foutmelding
```

In sommige structuren worden interne variabelen gebruikt (vb. `for` loop), die moeten ook gedeclareerd worden:

```
for (var i=0; i < tabel.length; i++){ … }
```

### 4.5.4 Scope

De *scope* van een variabele is het **bereik binnen het programma** waarin de variabele geldt.

Een ***global variable*** is overal bereikbaar (binnen de *Namespace* = geheugenruimte van één webpagina) in tegenstelling tot een ***local variable*** die enkel bereikbaar is binnen de **functie** waarin hij gedeclareerd wordt.

Een **global** variabele wordt gedeclareerd **buiten** een functie, een **local binnen** de functie:

```
var GETAL = 2;
function scopeTest(){
  var tekst = "local";
  alert(GETAL);  //toont 2 want global
  alert(tekst);  //toont "local" want bereikbaar
}
alert(GETAL);    //toont 2
alert(tekst);    //geeft een runtime error want tekst is onbereikbaar
```

Helaas heeft JS ook enkele onzorgvuldige trekjes…
Zo is het dat een variabele die binnen een functie gedeclareerd wordt **zonder** het `var` sleutelwoord, steeds een global bereik heeft:

```
function scopeTest(){
  teller = 0;
  alert(teller); //toont 0
}
alert(teller);  // toont 0 want global scope
```

Dit kan leiden tot onverwachte resultaten, maak daarom steeds gebruik van het `var` sleutelwoord!

Javascript heeft ***function scope***, geen *block scope*: variabelen die gedeclareerd worden binnen een function statement zijn overal binnen de functie :

```
function scope(){
  var tekst = "this: " + this;

  (function (){
        var aantal=0;

        for(var i=0;i<6;i++){
              aantal += i;
        }
        tekst += "\naantal: " + aantal + "\n i is nu: " + i;
        alert(tekst);
  })()
  alert("aantal:" + aantal);//fout
}
```

De functie toont:

```
  this: [global Object]
  aantal: 15
  i: 6
```

Dit voorbeeld demonstreert dat de variabele *i* niet beperkt is tot de `for` loop maar binnen de anonieme functie beschikbaar is, net als *aantal*.

*aantal* is echter niet beschikbaar buiten de anonieme functie: dit veroorzaakt een foutmelding (zie Javascript console).

*tekst* is beschikbaar voor alle geneste functies binnen *scope()*.

## *4.6 Statements*

Een **statement** is een programmeerregel.
Een statement wordt afgesloten door

- Een nieuwe lijn
- Een puntkomma
- Een block-accolade

```
alert('joehoe')
window.close()
```

en

```
alert('joehoe'); window.close()
```

zijn 2 statements en identiek.

### 4.6.1 Block statements

Een groep statements kan verzameld worden in een *block statement* door het in accolades `{ }` te zetten:

```
{
  var p = Math.PI;
  alert(pp);
}
```

Dit *block* gedraagt zich als één statement (met twee sub-statements). Een block statement heeft geen puntkomma aan het einde nodig.

Programmastructuren zoals lussen en functies maken gebruik van *block statements*.
Merk op dat de literal (letterlijke) schrijfwijze van een object ook van accolades gebruik maakt.

## *4.7 Programmastructuren*

Maken gebruiken van block statements.

### 4.7.1 Conditionele structuren

#### 4.7.1.1 if

Het `if` statement **test** een expressie en voert die uit als hij `true` is

- de **expressie** moet worden ingesloten in **ronde haakjes ( )**, anders wordt hij niet geëvalueerd
- de uit te voeren code wordt liefst als een *block* geschreven

```
if (wachtwoord == "geheim") login = true; //mogelijk


if (username == "") { // beter
  username = "default";
  login = true;
}
```

Merk op  dat een enkelvoudig statement op dezelfde lijn zonder block kan geschreven worden. Toch beter een block gebruiken.

Indien de expressie `false` evalueert, gebeurt hier niets.

#### 4.7.1.2 if else

Het `if` statement kan uitgebreid worden met een `else` die uitgevoerd wordt als de expressie `false` evalueert:

```
if (Jaar % 4 == 0) {
```

```
 febdagen = 29
} else {
 febdagen = 28
}
```

`if..else` statements kunnen **genest** worden. Het is daarbij aan te raden consistent accolades te gebruiken.

```
if (geslacht == "v") {
  if(lengte>190){
          sport="hoogspringen";
  }
  else{
          sport="volleyball";
  }
}
else
{
  if(lengte>190){
          sport="basketball";
  }
  else{
          sport="boksen";
  }
}
```

`if..else` statements zijn ook specifiek bedoeld om keuzes te maken uit twee mogelijke waarden, indien er meer dan twee mogelijke antwoorden zijn gebruik je best een andere structuur.

Een eerste mogelijkheid is `else if` statements in een `if`.

### 4.7.2 else if

Een `else if` is een uitdrukking in een `if ...else`. Hij wordt onmiddellijk gevolgd door zijn eigen expressie.
De laatste optie in de `if` is dan een `else` die alles voor zijn rekening neemt dat niet hogerop past:

```
if(lengte>190){
          sport="hoogspringen";
  }
else if (lengte>180){
          sport="basketball";
  }
else if (lengte>170){
```

```
        sport="volleyball";
  }
else
{
  sport="worstelen";
}
```

Indien je het programma volgens meerdere waarden van dezelfde variabele wil laten vertakken, is het beter gebruik te maken van een `switch` statement.

### 4.7.3 switch

Het `switch` statement vertakt het programma aan de hand van de waarde van een expressie.
Een `case` statement vergelijkt de waarde van de expressie (berekening, variabele, ..).
Het `break` statement stopt de uitvoering van de `switch` als een bepaalde tak gekozen is. Noteer dat de `break` verplicht is, anders gaat hij door.
Een `default` statement neemt alles voor zijn rekening dat hogerop niet past.

```
switch (cityVal) {
  case "bru" :
        city = "Brussels";
        break;
  case "ams" :
        city = "Amsterdam";
        break;
  case "lon" :
        city = "London";
        break;
  case "nyk" :
        city = "New York";
        break;
  default :
        city = "unknown"
}
```

Opmerkingen over `switch`:

- ideaal voor meerdere waarden van dezelfde variabele
- is veel sneller in uitvoering dan meerdere `else if` uitdrukkingen of een geneste `if`
- de `case` statements kunnen een expressie zijn, zoals *n+1*, maar mogen geen vergelijkingsoperators bevatten zoals *> 4*

### 4.7.4 Lusstructuren

Lusstructuren of **iteraties** laten je toe een bepaald stuk code meerdere keren uit voeren.

### 4.7.5 while

een `while` loop voert een statement uit zolang aan een voorwaarde voldaan is. Deze structuur is ideaal als je vooraf niet weet hoeveel maal je de lus nodig zult hebben. Omdat de evaluatie **vóór** de lus gebeurt, wordt de lus mogelijks zelf niet eenmaal uitgevoerd.

De syntax:

```
while (condition) {
    statements
}
```

In het volgende voorbeeld wordt de loop uitgevoerd zolang n=3 niet bereikt is.

```
var n = 0
var x = 0
while(n<3) {
  n++;
  x += n;
}
```

In een ander voorbeeld worden alle items van een keuzelijst nagegaan om te weten te komen wie de gebruiker geselecteerd heeft:

```
function watGekozen(keuzeLijst) {
  var i=0;
  while (keuzeLijst.options[i].selected===false) {
         i++;
         }
  return keuzeLijst.options[i].value;
}


...
<form name="muziekForm">
<select name="muzikanten">
  <option>Will Tura</option>
  <option>Danna Winner</option>
  <option>Jo Valy</option>
  <option>Will Ferdy</option>
  <option>Eva de Roovere</option>
  <option>Freddy Mercury</option>
</select>
<input type="button" value="Wie geselecteerd?"
  onclick="alert('U koos '  + watGekozen(document.muziekForm.muzikanten))">
</form>
```

In deze structuur is de lus niet verantwoordelijk voor de teller i, daarom moet die expliciet verhoogd worden.

### 4.7.6 do...while

Een `do...while` loop voert een lus uit zolang aan een voorwaarde voldaan wordt, maar de evaluatie gebeurt **na** de lus.
Dit heeft voor gevolg dat de loop minstens eenmaal uitgevoerd wordt.
De syntax van de expressie is als volgt:

```
do  {
   statement
} while (condition)
```

Een dergelijke lus wordt minder gebruikt dan een `while` precies omdat de lus altijd eenmaal uitgevoerd wordt, maar dat kan zijn nut hebben:

```
var deLijst = [["Julie","vriend"],["Kurt","vijand"],["Leeloo","vriend"]];


function vriendOfVijand(arrArray){
  var i =0;
  do {
    i++
  } while (arrArray[i-1][1]!="vijand")
alert(arrArray[i-1][0] + " is de vijand");
}
```

In bovenstaand voorbeeld zoeken we een twee-dimensioneel array af naar een 'vijand'. Eenmaal gevonden stopt de lus. Toch moet minstens eenmaal de lus doorlopen worden.

### 4.7.7 for

Een `for` lus voert een statement een **vast aantal malen** uit.

De syntax van de structuur is:

```
for ([startExpressie]; [voorwaarde]; [wijziging])
{
  statements
}
```

- De *startExpressie* is de startwaarde, meestal een interne variabele die geïnitialiseerd wordt
- De *voorwaarde* is een expressie die geëvalueerd wordt: zolang die `true` is wordt de lus verder uitgevoerd, bij `false` gaat de uitvoering verder bij de code voorbij de lus
- De wijziging bepaalt met hoeveel de startwaarde bij elke lus verhoogt of verlaagt, meestal maakt men gebruik van de ++ of -- operators
- de drie onderdelen zijn gescheiden door puntkomma's, de gehele `for`-expressie zit in ronde haakjes

Een heel eenvoudig voorbeeld verduidelijkt:

```
for (var i=0; i<3; i++)
{
  alert("in de lus:" + i);
}
alert ("Na de lus:" + i);
```

De statement in deze lus zal 3 maal uitgevoerd worden, maar noteer dat de eindwaarde van de variabele i = 3.
Let er ook op de teller variabele met een `var` te declareren want anders wordt hij **global** en kan interfereren met een variabele met dezelfde naam in een andere functie!

In een ander voorbeeld telt een functie het aantal geselecteerde items in een multiple-selection list.
De `for` lus start vanaf nul tot het totale aantal opties (`selectObject.options.length`) in de lijst en controleert telkens of deze optie geselecteerd werd.

```
<script type="text/javascript">

function howMany(keuzeLijst) {
  var aantal=0;
        for (var i=0;i<keuzeLijst.options.length; i++) {
               if (keuzeLijst.options[i].selected===true) aantal++;
        }
  return aantal;
}
</script>

<form name="muziekForm">
<p>Kies enkele muziektypes en bevestig uw keuzes</p>
<select name="musicTypes" multiple="multiple">
  <option selected>R&B</option>
  <option>Jazz</option>
  <option>Blues</option>
  <option>New Age</option>
  <option>Classical</option>
  <option>Opera</option>
</select>
<input type="button" value="hoeveel geselecteerd?"
  onclick="alert('aantal geselecteerd: ' + howMany(document.muziekForm.musicTypes))">
</form>
```

De startExpressie, voorwaarde en wijziging uitdrukkingen kunnen echter veel complexer worden als je gebruikt maakt van de **komma operator** die je toelaat **meerdere voorwaarden** te gebruiken:

```
for (i=0,j=8; i<j; i+=2, j++){
  alert ("i: "+i+" j: "+j)
}
alert ("na de lus: i: "+i+" j: "+j)
```

### 4.7.8 for...in

Een `for...in` lus wordt specifiek gebruikt om doorheen **properties van een object** te lussen in een willekeurige volgorde.
Properties en methods van ingebouwde objecten, zoals het `Math` object, worden niet getoond, wel alle eventuele uitbreidingen gemaakt door de programmeur.

Ook arrays kunnen met dit statement doorlopen worden, maar dit wordt afgeraden (gebruik eerder een `for` lus) omdat naast de geïndexeerde array-elementen ook eventuele andere (user-defined) eigenschappen getoond zullen worden en omdat een numerieke (index) volgorde niet gegarandeerd is.

```
function toonProps(obj){
var strProps="";
for(var eigenschap in obj){
  strProps += eigenschap + ": " + obj[eigenschap] +"\n";
  }
return strProps;
}
```

Let er op dat de eigenschap hier als een key van een associatief array gebruikt wordt.

### 4.7.9 for each... in (JS1.6)

Is identiek aan de `for...in` lus maar itereert over de waarden van de eigenschappen i.p.v. de eigenschappen zelf.
Bovenstaand voorbeeld omgevormd voor een `for each...in` lus:

```
function toonProps2(obj){
        var strProps="";
        for each(eigenschapWaarde in obj){
                strProps += eigenschapWaarde +"\n";
        }
        return strProps;
  }
```

JS1.6 wordt niet door alle browsers ondersteund.

### 4.7.10 Andere structuren

### 4.7.11 break

`break` kan voorkomen in een lus, in een `switch` of in een `label` statement.

Het **beëindigt** telkens de structuur en transfereert controle naar het statement dat volgt op de afgebroken structuur.

```
for (var i = 0; i < arr.length; i++) {
  if (arr[i]== mijnWaarde) {
        break;
  }
}
alert(mijnWaarde + "staat op positie " + i+1);
```

### 4.7.12 continue

Een `continue` statement in een loop breekt de verdere uitvoering van de statements in de **huidige iteratie** af en gaat door met de **volgende iteratie**

```
var i = 0;
var n = 0;
while (i < 5) {
  i++;
  if (i == 3)    continue;
  n += i;
}
```

In dit voorbeeld wordt de waarde i = 3 niet opgeteld bij n. De waarden van n zijn dus 1,3,7,12

### 4.7.13 Labels

Een `label:` is een identifiër die **een positie** in de code identificeert **waarnaar kan verwezen** worden.
Een label wordt geschreven als eender welke geldige identifiër gevolgd door een dubbelpunt.
Een voorbeeld

```
checkiandj :
  while (i<4) {
        document.write(i + "<BR>");
        i+=1;
        checkj :
        while (j>4) {
              document.write(j + "<BR>");
              j-=1;
              if ((j%2)==0);
                    continue checkj;
              document.write(j + " is odd.<BR>");
        }
```

```
  document.write("i = " + i + "<br>");
  document.write("j = " + j + "<br>");
}
```

### 4.7.14 with

Met een `with` statement kan je de **properties of methods van een object** gemakkelijker bereiken: je hoeft het object er niet telkens voor te typen:

```
with (Math){
  opp = PI * pow(r,2);
  y = r * sin(theta);
  x = r * cos(theta);
}
```

is identiek aan

```
opp = Math.PI * Math.pow(r,2);
y = r * Math.sin(theta);
x = r * Math.cos(theta);
```

Let erop dat je de dotnotatie (puntje) niet hoeft te gebruiken in het `with` statement.

### 4.7.15 Structuren voor exception handling

Een *exception* is het voorkomen van een fouttoestand.
Het opvangen van die fout noemen we "to *catch* an exception".
Het zelf opwerpen van een fout noemen we "to *throw* an exception".

Javascript zelf werpt exceptions als er een ***runtime error*** voorkomt - dat is een fout die enkel te detecteren is bij uitvoering, maar je kunt dat zelf doen met een `throw` statement.
Exceptions kunnen afgehandeld worden met een `try...catch` statement.
Deze statements werden geïntroduceerd in JS1.5

### 4.7.16 throw

Een `throw` statement werpt een *door de gebruiker gedefinieerde exception* op. De exception kan eender welke waarde hebben: een string, number, boolean of een object. Heel dikwijls zal de waarde een `Error` object zijn of een afgeleide klasse.
Als een exception geworpen wordt, stopt het programma onmiddellijk en gaat naar een eventuele ***exception handler*** - een `try...catch` statement.

Voorbeelden vind je hieronder.

### 4.7.17 try...catch

`try...catch...finally` is een structuur om exceptions op te vangen en af te handelen. Het bestaat minstens uit twee delen:

- `try` is het block die de code bevat waarin mogelijk *exceptions* zullen optreden

- `catch` is een optioneel block die de code bevat die de fout zal behandelen of ze negeren
- het optionele `finally` block bevat statements die altijd uitgevoerd worden, ongeacht het optreden van een *exception* of niet

Een voorbeeld van de volledige structuur:

```
function exceptionHandler1(){
  try {
        nietBestaandeFunctie();
        alert('functie uitgevoerd');
  }
  catch(e){
        alert('een exception stak de kop op');
  }
  finally{
        alert('eind van de reis door exceptionLand');
  }
}
```

In bovenstaande functie wordt getracht een andere, niet bestaande functie op te roepen. In dit geval levert dat een fout op en wordt het `catch` gedeelte uitgevoerd. Het laatste deel wordt sowieso uitgevoerd voor statements volgend op het `try...catch...finally` block.
Je bemerkt een argument `e` in `catch(e):` de `e` stelt het Error object voor dat opgeworpen is. Via deze variabele kunnen we meer informatie over de fout te weten komen: we kunnen er zijn eigenschappen van opvragen. Je bent volledig vrij om een andere variabele naam (vb. exception) ervoor te gebruiken.
De structuur kan ook zonder het `finally` deel gebruikt worden:

```
function exceptionHandler1(){
  try {
  ...
  }
  catch(e){
  ...
  }
}
```

of zonder het `catch` deel:

```
function exceptionHandler1(){
  try {
  ...
  }
  finally{
  ...
  }
```

```
}
```

Wat is het nut hiervan? Er wordt een poging gedaan om de code in de `try` uit te voeren, lukt dat niet dan stopt het programma tenminste niet.
`try...catch` statements kunnen genest worden.
Er kan ook voor een bepaald type fout gecontroleerd worden:

```
  try {
        eval("a++b");
  }
  catch(e){
        if(e instanceof SyntaxError){
              alert (e.name + ": " + e.description)
        }
  }
}
```

Een ontwikkelaar kan er ook voor kiezen zijn eigen fouten op te werpen:

```
function deelGetallen(){
  try {
        var getal1 = document.getallen.getal1.value;
        var getal2 = document.getallen.getal2.value;

        if(getal1==''||getal2==''||isNaN(getal1)||isNaN(getal2)){
              throw new TypeError('er worden twee getallen verwacht');
        }
        else if(getal1==0||getal2==0){
              throw new RangeError('een deling met of door nul is niet
  toegelaten');
        }
        else {
              alert("quotient: "+getal1/getal2);
  }
  }
  catch(e){
        alert("Foutieve input: "+e.name+": "+e.message);
  }
}
```

Bovenstaand voorbeeld evalueert twee inputveldjes in een formulier en controleert of ze getallen bevatten en of één van die getallen een 0 is. In elke situatie wordt een `Error` object geworpen en opgevangen.

## *4.8 Functies*

Een functie is een groep statements die als blok opgeroepen wordt en dus meerdere malen uitgevoerd kan worden.

In tegenstelling met andere programmeertalen zijn functies ook waarden, data types. Daardoor kunnen ze ook opgeslagen worden in variabelen, arrays en in objecten. Zo kunnen functies toegewezen worden als eigenschap van een object, in dat geval noemen we de functie een *method*.

```
function helloWorld(){
        alert("hello world");
}
```

### 4.8.1 return

Het sleutelwoord `return` *kan* een functie beëindigen en daarmee eventueel een eindwaarde meegeven zodat de `function` een resultaat teruggeeft aan zijn caller. Dat is geen verplichting: indien een `function` geen `return` heeft, returnt die `undefined`. Een functie kan ook meerdere `return` statements bevatten, die afhangen van het programmaverloop.
Een statement dat *na* een `return` staat, wordt nooit uitgevoerd.

```
function kwadraat(getal){
  return getal*getal;
  getal = null //nooit uitgevoerd
}
```

In bovenstaand voorbeeld geeft de functie het kwadraat van het getal terug. De `alert()` na de `return` wordt nooit uitgevoerd en heeft dus geen zin.
De bedoeling van een returnwaarde is dat je die gebruikt:

```
var opp = Math.PI * kwadraat(3);
```

### 4.8.2 function

Een functie wordt aangemaakt met het sleutelwoord `function` gevolgd door de (optionele) **naam** van de functie, een paar **ronde haakjes** met eventuele **argumenten** en daarna de **statements** vervat in een **block**.

De naam van de functie moet uniek zijn binnen de Namespace waarin gewerkt wordt: zo kan je geen twee functies helloWorld() hebben, maar wel een helloWorld() en mijnApplicatie.helloWorld(). Herinner je ook de hoofdlettergevoeligheid van JS.

Overloading bestaat in JS niet: als je twee functies hebt met dezelfde naam zal slechts de laatste werken.

#### 4.8.2.1 argumenten

Functies kunnen (tot 255) **argumenten** (parameters) verwerken.

Met uitzondering van **objecten** worden argumenten steeds doorgegeven **by value**, m.a.w. de waarde van de variabele wordt doorgegeven, niet de variabele zelf. Als de functie de waarde van de parameter wijzigt, is de oorspronkelijke waarde ongewijzigd.

Objecten (`Object`, `Array`, `Function`) die als argument gebruikt worden, worden **by reference** doorgegeven, m.a.w. als de functie de eigenschappen van het object wijzigt, is deze wijziging blijvend.

```
function absoluutVerschil(getal1,getal2){
  if(getal1>getal2){
                return getal1-getal2;
  }
  else{
                return getal2-getal1;
  }
}
```

Als een functie opgeroepen wordt met teveel argumenten worden deze genegeerd.
Als een functie opgeroepen wordt met te weinig argumenten, zijn deze `undefined`

```
function toonZe(strEen, strTwee){
        var strConcat = strEen + strTwee;
        alert(strConcat);
}
toonZe("jan","piet","joris"); //janpiet
toonZe("jan"); //janundefined
```

#### 4.8.2.2 arguments[ ]

Het `arguments` object, een property van `function`, bevat een soort array van de **doorgegeven** argumenten. Het heeft een `arguments.length` property die het aantal argumenten telt.
De items in dit array, `arguments[0], arguments[1],…` zijn synoniemen voor de echte parameters.
Dat is interessant om bijvoorbeeld het aantal argumenten te laten variëren of om een aantal optionele argumenten te laten volgen op een aantal vaste argumenten.

```
function som(){
  var totaal=0;

  for(var i=0;i<arguments.length;i++){
        if(!isNaN(arguments[i])){
                totaal+=arguments[i];
        }
  }
  return totaal;
  }
...


alert('som is: '+ som(1,2,3,'zorro was here',5,undefined,6,7))
```

#### 4.8.2.3 Optionele argumenten

De beste manier om met **optionele** argumenten om te gaan is de zogenaamde **options hash**. Een optioneel argument is daar een property in een `object`. Het laat ons eveneens toe standaardwaarden te voorzien in het geval dat ze afwezig zijn. Een voorbeeld: de functie maakTekst() plaats een tekst in een inhoud element. Je kan optioneel opmaak bij specifiëren :

```
function maakTekst(tekst,opties){
  // tekst is een verplicht argument
  // alle andere argumenten zijn optioneel, ook hun volgorde is onbelangrijk
  // door ze in een object te verzamelen
  if (tekst){
        var inhoud = document.getElementById('inhoud');
        // de standaardwaarden
        var settings = {
              kleur: "blue",
              achtergrondKleur: "yellow",
              type:"Times New Roman",
              vet: "normal",
              grootte: "1em",
              schuin: "italic"
        }
        for (var j in opties||{}) {
              settings[j] = opties[j];
        }
        var pee = document.createElement('p');
        pee.appendChild(document.createTextNode(tekst));

        with (pee.style){
              color = settings.kleur;
              backgroundColor = settings.achtergrondKleur;
              fontFamily = settings.type;
              fontWeight = settings.vet;
              fontSize = (settings.grootte);
              fontStyle = settings.schuin;
        }
        inhoud.appendChild(pee);
  }
  else {throw new Error('tekst is een verplicht argument')}
}
```

De functie kan dan opgeroepen worden:

```
maakTekst('geen argumenten: alle default waarden');
maakTekst('kleur en grootte',{kleur:"red", grootte:"3em"});
maakTekst('achtergrondkleur, lettertype en grootte',{achtergrondKleur:"silver",
```

```
        type:"Monotype Corsiva", grootte:"2em"})
maakTekst(); //geeft een fout
```

Bespreking:
het verplichte argument tekst wordt gecontroleerd: als dit afwezig is werpen we een fout
alle optionele argumenten worden doorgegeven als properties van een object:

```
{kleur:"red", grootte:"3em"}
```

de functie zelf krijgt dat als het argument opties binnen
deze 'opties' worden verwerkt via een `for in` , de standaard manier om doorheen properties van een object te lussen:

```
for (var j in opties || {} ) {
            settings[j] = opties[j];
      }
```

de lus gaat doorheen alle *opties* properties en stelt daarmee het *settings* object in. Properties die overeenkomen worden overschreven, andere blijven op de standaardinstelling staan.

Merk de `|| {}` op : als opties volledig achterwege blijft, maken we een leeg object aan zodat de lus niet crasht

#### 4.8.2.4 Recursie

Een functie kan zichzelf oproepen: **recursie**.

```
function factorieel(getal){
  if(getal==0){
        return 1;
  }
  else{
        return getal * factorieel(getal-1)
  }
}
```

### 4.8.3 anonieme functies

Omdat functies waarden zijn, kan je ook zogenaamde **anonieme** of **lambda** functies maken. Een anonieme functie werkt net als een andere maar heeft geen naam:

```
var kwadraat = function(getal){
 return getal*getal;
}
```

De variabele kwadraat bevat nu eigenlijk de functie zelf, niet zijn resultaat.
Als je bv. een alert van kwadraat zou doen, krijg je de functie te zien, geen eindwaarde. Om die te krijgen moet je de functie, ergo de variabele, uitvoeren:

```
alert(kwadraat); // geeft function(getal){…
alert(kwadraat(2)); //geeft 4
```

Anonieme functies zijn erg nuttig bij de aanmaak van methods van objecten:

```
var MijnApp = new Object;
MijnApp.bar = function () { }
MijnApp.foo = function () { }
```

In het hoofdstuk "object georiënteerd programmeren" vind je hiervan toepassingen.

### 4.8.4 geneste functies

Functies kunnen in elkaar genest worden:

```
function cylinderVolume(hoogte, straal){
        function basisOpp(){
               return straal * straal * Math.PI;
        }
 return basisOpp() * hoogte;
}
```

Het belangrijk op te merken dat de 'binnenfunctie' toegang heeft tot de variabelen van de 'buitenfunctie'.

### 4.8.5 het sleutelwoord `this`

De tijd is gekomen om wat dieper in te gaan op de betekenis van `this`.
De exacte waarde van `this` hangt sterk af van de **context** waarin het gebruikt wordt:

- in een **functie** wijst `this` naar het globale object, in een browser is dat het `window` object

```
function kleurMij(obj){
  obj.style.color = 'red';
  this.style.color = 'red'; //fout
}
```

- in een **object property** of **method** wijst `this` naar het object zelf

```
var MijnApp = new Object();
MijnApp.naam = 'MagicCalc 1.0';
MijnApp.spreek = function(){
  alert(this.naam);
}
```

- in een **constructor** is `this` gebonden aan het nieuw te maken object

```
function Planeet(naam, diameter){
  this.naam = naam;
  this.diameter = diameter;
  this.spreek = function(){
         alert(this.naam):
  }
}
var aarde = new Planeet('aarde',6378);
aarde.spreek();
```

# 5 DOM

In de introductie werd je al verteld dat het *Document Object Model* een interface is voor HTML en XML documenten. Je browser gebruikt het om een ingeladen document te lezen en te tonen.

De meeste programmeertalen laten tegenwoordig toe dat je de DOM direct manipuleert, Javascript is daar wel de belangrijkste taal van.

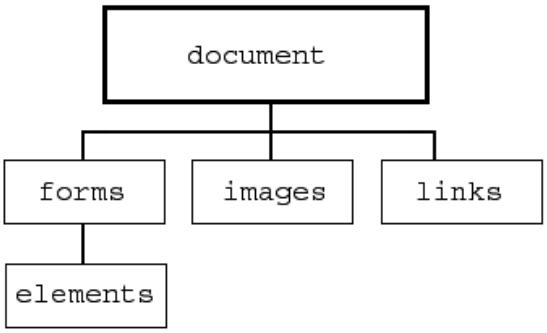
## *5.1 Standaarden*

Het DOM interface is geëvolueerd en nieuwe aanpassingen zullen zeker nog volgen. Een aantal stappen in die evolutie hebben een versienummer gekregen, die we kort belichten, omdat sommige oude methodes nog steeds werkzaam zijn.

Tegenwoordig spreken we van **specificaties** (standaarden) die je kunt raadplegen op de website van het W3C onder DOM specifications.

### 5.1.1 DOM Level 0

DOM0 is zeer beperkt. Het bevat het `window` object en zijn enige *child*, het `document` object. Enkel formulierelementen, images en links zijn bereikbaar




DOM0 kan nog steeds toegepast worden, met name de `forms` collection wordt gebruikt bij het verwerken van formulieren en ook de `images` en `links` collections zijn perfect bruikbaar.

Maar daar eindigt het dan ook mee. Met DOM0 is er bijvoorbeeld geen sprake van de inhoud van een `p` element te wijzigen na het inladen van het document.

### 5.1.2 DOM Level 2

DOM2 bracht een belangrijke vooruitgang (2008) door de toevoeging van veel methods en properties. Deze specificatie werd ook in verschillende modules opgebouwd.

Het laat ons toe om individuele elementen te bereiken, om hun CSS style te wijzigen en om de *document tree* dynamisch te wijzigen door elementen te verwijderen en toe te voegen. Ook XML documenten kunnen verwerkt worden.

### 5.1.3 DOM Level 3

DOM Level 3 mag je beschouwen als de **huidige standaard** (2012). De meest recente browsers ondersteunen grosso modo alle modules.

DOM3 is inderdaad modulair opgebouwd, m.a.w. er is niet één specificatie omdat die zo omvangrijk en en er steeds nieuwe modules bijkomen. Er zijn dus verschillende aparte modules die allemaal samen de specificatie vormen. Zo hebben we bijvoorbeeld:

- DOM Core (algemene node manipulatie),
- DOM HTML (werken met HTML elementen),
- DOM CSS (manipuleren van stylesheets),
- DOM Events (event handling),
- etc…

Je vindt deze modules terug op de website van het W3C, w3.org,
onder "*Web Design en Applications*" - "*Javascript Web API's*" – "*DOM*".

Directe links vind je ook achteraan deze cursus.

Er heeft momenteel een versnelde evolutie plaats die veel te maken heeft met HTML5, met nieuwe modules die al wel of niet geïmplementeerd worden door de browsers.

## *5.2 Interfaces*

De DOM beschouwt een "item" (bv. een `table` element) als iets waar verschillende **interfaces** van toepassing op zijn.

Wat bedoelen we met *interfaces*? als voorbeelden nemen we een *mens* en een *vogel*

> Een *mens* is terzelfdertijd een *vast voorwerp*, een *levend wezen* en een *zoogdier*.
> Daardoor kan je het ook *vastnemen*, *plant het zich voort* en kan het *zogen*.
> Een *vogel* is ook een *vast voorwerp*, een *levend wezen* maar is geen zoogdier en kan daardoor niet *zogen*, maar wel *vliegen*.
> Beide wezens implementeren de **interfaces** *vast voorwerp*, *levend wezen* maar verschillen in de derde .

Zo ook is een `table` element terzelfdertijd een `node`, een `HTMLElement`, en een `TableElement`. Dat betekent dat een `table` element terzelfdertijd de methods en properties van een `node`, van een `HTMLElement` en van een `TableElement` heeft:

- het is zowieso een DOM *node* en heeft daardoor bijvoorbeeld de beschikking over de eigenschap `parentNode` en kan de method `appendChild` uitvoeren
- het is een `HTMLElement` en heeft dus bijvoorbeeld een eigenschap `className` en de method `focus()` uitvoeren
- maar het implementeert ook de `HTMLTableElement` interface en krijgt van daaruit de methode `createTHead` toebedeeld.

Een ander element, `div` bijvoorbeeld, implementeert ook de `Node` en `HTMLElement` interfaces en zal over hun methods en properties beschikken, maar niet over de `HTMLTableElement` interface.

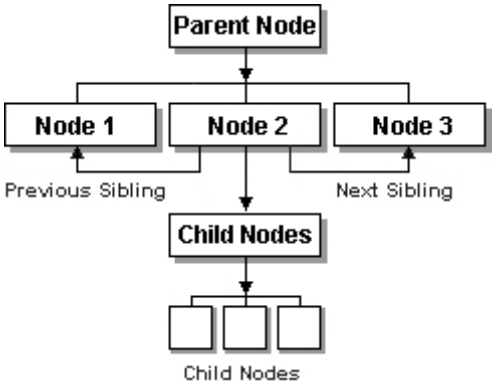
De **belangrijkste** interfaces zijn `Node` als algemene interface voor alle nodes, `Document` voor documenten, `DocumentFragment` voor documentFragments, `Attr` voor attributen, `Element` voor elementen, `Text` voor tekst.

Het is voor deze cursus niet essentieel te weten welke interfaces een `div` element allemaal implementeert, maar je begrijpt op die manier de reden waarom de property `styleSheets` enkel op een `document` node kan gebruikt worden en niet op een `div` element: een `div` heeft natuurlijk geen stylesheet collection, een document wel.

## *5.3 Nodes*

De DOM gaat uit van het concept van *nodes*: een document bestaat uit een reeks nodes van verschillende types die zich tot elkaar verhouden. Deze nodes worden opgebouwd tot een *DOM Tree*.

Nodes verhouden zich tot elkaar als een soort familie:




Een node heeft meestal een **Parent Node** en kan **Siblings** hebben, broertjes of zusjes, die dezelfde Parent Node hebben. Het kan zelf ook **Child Nodes** hebben.

De Node Interface voorziet properties en methods om deze familieleden te bereiken , te verwijderen of toe te voegen.

Welke eigenschappen en methods een node precies heeft hangt veel af van zijn `nodeType`. De `nodeType` property van die node geeft je een nummercode waaraan je hem kunt herkennen:

| type node | `nodeType` | beschrijving |
|---|---|---|
| `element` | 1 | een XML/HTML element: kan *childNodes* en *attribuutNodes* hebben |
| `attribute` | 2 | een *attribuutNode* is nooit een *childNode*, eerder een eigenschap van een `element` node, en maakt geen deel uit van de DOM Tree |
| `text` | 3 | de tekstuele inhoud van een element of attribute node.<br>Kan geen *child nodes* hebben |
| `CDATASection` | 4 | bevatten *escape* tekst voor karakters die anders geïnterpreteerd zouden worden door de browser. |

| type node | nodeType | beschrijving |
|---|---|---|
| | | Kan geen *child nodes* hebben |
| entityReference | 5 | een reference naar een Entity in een XML document |
| entity | 6 | is een Entity in een XML document |
| processing instruction | 7 | is een Processing Instruction in een XML document.<br>Kan geen *child nodes* hebben |
| comment | 8 | commentaar afgebakend met <!-- en --><br>Kan geen *child nodes* hebben |
| document | 9 | een volledig HTML of XML document: de `root` van dit doc |
| documentType | 10 | een `docType` attribuut van een document. Indien afwezig waarde `Null`.<br>Kan geen *child nodes* hebben |
| documentFragment | 11 | een deel van een document, een lichtgewicht document. Enkele gebruikt tijdens het dynamisch opbouwen van een 'stuk' DOM.<br>Moet niet noodzakelijk *well-formed* zijn. |
| notation | 12 | de opmaak van een unparsed Entity in een DTD.<br>Kan geen *child nodes* hebben |

**Elk van deze types erft de properties en methods van de node Interface.**

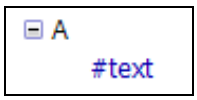
Naast bovenstaande node types zijn er nog enkele **datatypes** in de DOM waar je mee te maken kan krijgen:

| | |
|---|---|
| nodeList | Een `nodeList` is een array van `element` nodes.<br>Zo returnt de method `document.getElementsByTagName()` bijvoorbeeld een `nodeList`. Items in een `nodeList` kan je bereiken op twee manieren:<br>• `list.item(1)`<br>• `list[1]` |
| namedNodeMap | Een `namedNodeMap` is een *associatief array*, een array van nodes waar je de items kan bereiken via hun naam, zowel als via hun index.<br>De volgorde in een `namedNodeMap` is echter willekeurig en kan veranderen bij elke toevoeging of verwijdering, dus heeft de index weinig nut. |

Een eerste voorbeeldje voor meer duidelijkheid. Veronderstel een hyperlink, een `a` element:

```
<a href="http://www.vdab.be" id="lienk">vdab</a>
```

In de *document tree* ziet dat er zo uit:




Een `element` node waarin een `text` node zit. De attributen zijn niet voorgesteld in de tree. Als we hierop het volgende scriptje proberen krijgen we verschillende resultaten:

```
var lienk = document.getElementById('lienk');

alert(lienk.nodeValue); //geeft null
alert(lienk.firstChild.nodeValue) ; //geeft 'vdab'
alert(lienk.attributes['href'].nodeValue) ; //geeft 'http://www.vdab.be'

alert(lienk.nodeName) ; //geeft 'A'
alert(lienk.firstChild.nodeName);//geeft '#text'
alert(lienk.attributes['href'].nodeName); //geeft 'href'

alert(lienk.tagName) ; //geeft 'A'
alert(lienk.firstChild.tagName); //geeft undefined
alert(lienk.attributes['href'].tagName)      ; //geeft undefined
```

Omdat we hier respectievelijk een **element node**, een **text node** en een **attribuut node** aanspreken, moeten we niet verbaasd zijn dat sommige een bepaalde methode/property niet ondersteunen. Ze hebben het van hun Interfaces!

In de volgende sectie bespreken we kort de belangrijkste node types, daarna maken we een opsomming van de belangrijkste methods en properties, gevolgd door enkele voorbeelden voor hun gebruik.

*Onthoudt dat alle **elementen**, **documenten**, **documentFragmenten** etc.. steeds **nodes** zijn en dus beschikken over alle node properties en methods!*

## 5.4 text

Een `text` node, `nodeType` 3, is de **textuele inhoud** van een `element` of van een `attribute` node. Dus de tekst tussen begin en eindtag, of de tekst in een attribuut.
Een `EMPTY` element zoals een `<br>` heeft dus geen `textNode`.

Een text node kan zelf geen children hebben maar kan wel siblings hebben, andere text nodes die dezelfde parent hebben. Dit kan enkel gebeuren als wij een extra textnode toevoegen aan zijn parent. Door de method `normalize()` te gebruiken op die parent versmelten we die eventuele siblings met elkaar to één text node.

Een text node heeft geen eigen properties en slechts één minder belangrijke) method, `split()`.

## *5.5 document*

De `document` node stelt een XML of HTML document voor en implementeert methods en properties uit de DOM Core en HTMLDocument Interface.

De `document` node is een child is van het `window` object. Je moet echter niet steeds `window.document` schrijven, `document` is genoeg.
Het kan een onbepaald aantal `element` nodes bevatten.

Het `document` heeft heel wat properties en methods , je vind ze in de lijst verderop.

## *5.6 element*

Een `element` node is een HTML of XML element. Het implementeert methods en properties uit de DOM Core, DOM Elements en DOM Events Interface.

**Elk** `element` in een document heeft de volgende eigenschappen en methodes. Voor Events zie het relevante hoofdstuk.

## 5.7 De belangrijkste DOM Attributes

Om het wat overzichterlijker te maken groeperen we de attributen in een aantal rubrieken.

| Attribute (Property) | gebruik op een (Interface) | return type | returnwaarde | bespreking |
|---|---|---|---|---|
| **DOM Traversing** | | | | |
| `attributes` | `Node` | `NamedNodeMap` | verzameling van alle attributen. | de `attributes` collection kan de attributen van een element als **geïndexeerd array** voorstellen:<br>o `element.attributes[0].` We raden je af dit te gebruiken.<br>of met een **key**:<br>o `element.attributes['href']` is OK.<br><br>Probeer echter steeds de directe weg als property:<br>`element.href`<br>indien dit niet lukt probeer ook<br>`element.getAttributes('href')` |
| `childNodes` | `Node` | `NodeList` | alle *child nodes* van de node | met `childNodes` krijg je de onmiddellijke kinderen van een element, geen 'kleinkinderen':<br><br>`childNodes[0]` is de eerste `childNode`,<br>`childNodes[childNodes.length-1]` de laatste<br><br>hou er rekening mee dat het hier **alle** nodes betreft, niet enkel elementen, ook textNodes en comment nodes. |
| `firstChild` | `Node` | `Node \| null` | de eerste directe *child node* van de node | |

| **Attribute (Property)** | **gebruik op een** (Interface) | **return type** | **returnwaarde** | **bespreking** |
|---|---|---|---|---|
| `lastChild` | `Node` | `Node \| null` | de laatste directe *child node* van de node | |
| `nextSibling` | `Node` | `Node \| null` | de node die onmiddellijk volgt op de node en die dezelfde *parent* heeft | |
| `previousSibling` | `Node` | `Node \| null` | de node onmiddellijk vóór de huidige node en die dezelfde *parent* heeft | |
| `nodeName` | `Node` | `String` | de naam van de node, bijvoorbeeld `a` voor een hyperlink, maar `#text` voor een textNode, `#document` voor een documentNode, `#document-fragment` voor een documentFragmentNode, | Voor HTML elementen altijd in *lowercase*, voor XML zoals ze geschreven zijn.<br>Voor een element is `nodeName` identiek aan de `tagName` |
| `nodeType` | `Node` | `Number` | het getal dat een node type voorstelt.<br>Zo returnt deze eigenschap voor een element node steeds 1 | |
| `nodeValue` | `Node` | verschillend \| `null` | de `value` van de node. | Enkel van toepassing op `Attribuut`, `CDATA`, `Comment` en `Text` nodetypes .<br>Voor een element node is dat altijd `null`.<br>Voor een `Text` node geeft dit de tekst zelf. |
| `parentNode` | `Node` | `Node \| null` | de node die de onmiddellijke parent is van de huidige node of `null` indien er geen parent is | |
| `ownerDocument` | `Node` | `document \| null` | de `document` node waarin de node zich bevindt of `null` indien dat niet het geval is (tijdens de aanmaak bv.) | |

| Attribute (Property) | gebruik op een (Interface) | return type | returnwaarde | bespreking |
|---|---|---|---|---|
| **HTML Element** | | | | |
| `className` | `Element` | `String \| ""` | Leest of zet de `class` van een element | Hou er rekening mee dat je meerdere CSS classes kunt instellen voor een element |
| `id` | `Element` | `String \| ""` | Leest of zet het `id` attribuut van het element | is de directe manier om de inhoud van het attribuut id te lezen of in te stellen |
| `innerHTML` | `Element` | `String \| ""` | Leest of zet de inhoud (ook HTML-code) binnen de tag van het element | is geen standaard DOM method maar wordt door alle browsers ondersteund. Het laat je toe de volledige tekstuele inhoud van een element te lezen of in te stellen.<br>Zie ook de discussie innerHTML vs. standaard DOM methods in het projectenboek |
| `isContentEditable` | `Element` | `Boolean` | geeft aan of de inhoud van een element wijzigbaar is | alleen-lezen. Enkel in nieuwste browsers (HTML5) |
| `style` | `Element` | `CSSStyledeclaration` | het style object die de inline style attributes van het element bevat | Belangrijk: het gaat hier wel degelijk over de **inline styles**, niet over de stylerules van het stylesheet!<br>Inline styles hebben altijd de **overhand** op stylesheet rules |
| `tabIndex` | `Element` | `Number` | Leest of zet de positie van het element in de tabvolgorde | |
| `tagName` | `Element` | `String` | de naam van het element | zie ook `nodeName` |
| `name` | `Attr` | `String \| ""` | Leest of zet het `name` attribuut van het element | |
| `value` | `Attr` | `String \| ""` | leest of zet de waarde van een attribuut als String | vergelijkbaar met `element.getAttribute()` |

| Attribute (Property) | gebruik op een (Interface) | return type | returnwaarde | bespreking |
|---|---|---|---|---|
| `ownerElement` | `Attr` | `Element \| null` | geeft het Element waarin dit attribuut zit | |
| **document node** | | | | |
| `documentElement` | `document` | `Element` | geeft het root element van het document. Alleen-lezen | voor HTML is dat altijd `html` |
| `cookie` | `document` | `string` | leest en zet de cookies voor dit document | |
| `forms` | `document` | `Nodelist` | collection van alle `form` elementen in dit document | `forms[0]` is het eerste `form` element |
| `images` | `document` | `Nodelist` | collection van alle `image` elementen in dit document | `images[0]` is het eerste `image` element |
| `location` | `document` | `location` | Het URL van het document | lezen en schrijven |
| `refferer` | `document` | `String \| ""` | Het URL van de pagina die je naar deze pagina bracht | empty als je rechtsreeks kwam |
| `styleSheets` | `document` | `Nodelist` | collection van alle `stylesheet` elementen geassocieerd met dit document | zowel interne als externe stylesheets |
| `title` | `document` | `String \| ""` | leest of zet het `title` element in dit document | lezen en schrijven |

### 5.7.1 de belangrijkste DOM Methods

Een method voert een handeling uit, zoals het toevoegen van een *child* aan een element. Merk op dat het niet enkel de "zoek" methods zoals `getElementById` zijn die een returnwaarde geven, ook `removeChild` geeft iets terug…

Er zijn nog veel meer DOM methods, hier volgen de meest nuttige:

| Method | gebruik op een (Interface) | return waarde | actie | opmerkingen |
|---|---|---|---|---|
| **Elementen of Nodes zoeken** | | | | |
| `getElementById(id)` | `document` | `element` | returnt **één** element met de gegeven `id` of `null` indien die niet gevonden is. | enkel op een element node |
| `getElementsByTagName(name)` | `document, element` | `nodeList\|Null` | returnt een nodeList collection van alle elementen met deze `tagName` die child zijn van het `document` of van het startelement. | gebruik op het `document` of op een element node.<br>Gebruik `*` om alle child-elementen te krijgen |
| `getElementsByClassName(class)` | `document, element` | `nodeList\|Null` | returnt een nodeList van alle elementen met deze `class` die child zijn van het `document` of van het startelement. | **Nieuw!**<br>Gebruik op het `document` of op een element node |
| `querySelector(selectors)` | `document, element` | `element\|Null` | returnt het **eerste** element in de nodeList dat de selectie past. Returnt `Null` indien niets gevonden. | **Nieuw!**<br>De selectors zijn een string met CSS selectors. Zie Voorbeelden verderop |
| `querySelectorAll(selectors)` | `document, element` | `nodeList\|Null` | returnt de **volledige nodeList** dat de selectie past. Returnt `Null` indien niets gevonden | **Nieuw!**<br>De selectors zijn een string met CSS selectors. Zie Voorbeelden verderop |
| `hasChildNodes()` | `boolean` | `Boolean` | `true` als het element child nodes heeft | |
| **Nodes, elementen aanmaken** | | | | |
| `normalize()` | `Node` | | versmelt alle `text` nodes in deze node tot één `text node` | |

| Method | gebruik op een (Interface) | return waarde | actie | opmerkingen |
|---|---|---|---|---|
| `createTextNode(string)` | `document` | `text Node` | maakt en returnt een textNode met de meegegeven tekst | |
| `createElement(tagName)` | `document` | `element` | maakt en returnt een lege `element` node van de gegeven `tagName` | |
| `createDocumentFragment()` | `document` | `document-Fragment` | maakt en returnt een lege `documentFragment` node | |
| `cloneNode(deep)` | `Node` | de gekopieerde `Node` | kopieert een node.<br>Het argument `deep` is een `boolean` die bepaalt of alle child nodes ook mee gekopieerd moeten worden. | |
| | | | | |
| **Nodes, elementen toevoegen, verwijderen** | | | | |
| `appendChild(inTeVoegenNode)` | `Node` | de toegevoegde `Node` | voegt een node toe als **laatste** child aan het element | dus altijd achteraan |
| `insertBefore(`<br>`inTeVoegenNode,`<br>`voorDezeNode)` | `Node` | de ingevoegde `Node` | Voegt de `inTeVoegenNode` in als child van de huidige node juist voor het child `voorDezeNode`. | hiermee kan je dus een node eender waar invoegen |
| `removeChild(Node)` | `Node` | de gewiste `Node` | verwijdert de vermelde childNode uit het element | |
| `replaceChild(nieuweNode, oudeNode)` | `Node` | | vervangt de ene child node door de andere in het huidige element | |
| | | | | |
| **Attributen** | | | | |
| `getAttribute(name)` | `Node` | - | leest de waarde van het benoemde attribuut | |

| Method | gebruik op een (Interface) | return waarde | actie | opmerkingen |
|---|---|---|---|---|
| `setAttribute(name, value)` | `Node` | - | voegt een nieuw attribuut toe.<br>De waarde moet als een string gegeven worden.<br>Als er al een attribuut met deze naam aanwezig was, dan wordt zijn waarde overschreven met de nieuwe waarde. | |
| `createAttribute(name)` | `Node` | `attributeNode` | Maakt en returnt een attribuut Node | deze node moet nog ingevoegd worden in een element! |
| `removeAttribute(name)` | `geen` | | verwijdert dit attribuut. Als het attribuut een default waarde heeft, wordt het verwijderde attribuut onmiddellijk vervangen door het attribuut met de default waarde. | |
| **Event handlers** | | | | |
| `onclick, onsubmit, onkeypress, onchange, onmousemove, onmousedown,….` | | | event handlers die overeenkomen met de html "on-" attributen. De waarde van deze properties moet een functie zijn die zal uitgevoerd worden. | Dit zijn DOM0 eventhandlers. ze hebben het nadeel dat slechts één actie mogelijk is per event |
| `addEventListener(type, listener,useCapture)` | `Node, document, window, XHR` | geen | registreert een Event Handler aan een specifiek event type voor het object. | DOM2 manier. Veelzijdiger |
| `removeEventListener(type, listener,useCapture)` | | | verwijdert een Event Listener van het element | |
| **Andere** | | | | |

### 5.7.2 De nieuwe Selectors API

De twee nieuwe methods `querySelector` en `querySelectorAll` worden door alle moderne browsers ondersteund (2012). Ze zijn afkomstig uit de **selectors API**, http://www.w3.org/TR/selectors-api/.

Ze kunnen een aanzienlijke tijdsbesparing betekenen in het leggen van DOM referenties naar enkele of groepen elementen en de mogelijkheden zijn veel groter.

De syntax van beide methods is eenvoudig en kan zowel uitgevoerd worden op het `document` of vertrekkende vanaf een ander element.

```
document.querySelector(selectors) of element.querySelector(selectors)

document.querySelectorAll(selectors) of element.querySelectorAll(selectors)
```

`querySelector` returnt **één element:** het eerste in de collection die past.

`querySelectorAll` de volledige **collection** die past.

Indien geen enkel element de selector past returnen ze `null`.

De selector string is één of meerdere CSS selectors gescheiden door komma's, inclusief alle CSS3 selectors.

Enkele voorbeelden:

| Method | resultaat |
|---|---|
| `.querySelectorAll('div')` | alle `div` elementen in het document.<br>Identiek aan `document.getElementsByTagName('div')` |
| `.querySelector('div')` | het eerste `div` element in het document.<br>Identiek aan `document.getElementsByTagName('div')[0]` |
| `.querySelector('#speciaal')` | het element met de `id` "speciaal"<br>Identiek aan `document.getElementById('speciaal')` |
| `.querySelectorAll('.foo')` | alle elementen met de `class` "foo" .<br>Identiek aan `document.getElementsByClassName('foo')` |
| `.querySelectorAll('p.warning, p.error')` | alle `p` elementen die de `class`" warning" of "error" hebben. |
| `.querySelectorAll('ul.menu>li:first-child')` | alle eerste `li` elementen in `ul` elementen die de `class` "menu" hebben |

Deze methods maken gebruik van de CSS engine van de browsers en zijn daarmee het equivalent van de selectiemethodes in jQuery.

### 5.7.3 Enkele practische voorbeelden

Hieronder demonstreren we het gebruik van deze properties en methods in enkele veel voorkomende situaties.

Veronderstel de volgende HTML pagina
(document beschikbaar in basisbestanden : *DOMtests.html*) .

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
<title>DOM Attributen en methods uitgetest</title>
<style> … </style>
<script> … </script>
</head>
<body>
<div id="wrapper">
  <header>
    <hgroup>
      <h1>DOM Attributen en methods uitgetest</h1>
      <h2>Demos van de DOM in het theorieboek (editie 2012)</h2>
    </hgroup>
  </header>
  <article>
    <section id="huishouden">
      <h3>Boodschappenlijst</h3>
      <ul>
        <li>brood</li>
        <li>margarine</li>
        <li>eieren</li>
        <li>spek</li>
      </ul>
      <h3>Klusjeslijst</h3>
      <ul>
        <li>gordijnen strijken</li>
        <li>schilderen
          <ul>
            <li>keukenvenster</li>
            <li>garagedeur</li>
          </ul>
        </li>
        <li>mol vangen</li>
      </ul>
    </section>
```

```
    <section id="wishlist">
     <h3>Wishlist</h3>
      <table class="flashy">
        <thead>
          <tr>
            <th scope="col">Wat</th>
            <th scope="col">prijs</th>
            <th scope="col">foto</th>
            <th scope="col">wissen</th>
          </tr>
        </thead>
        <tbody>
          <tr>
            <td>iPad</td>
            <td>450</td>
            <td><img src="../images/ipad.jpg" width="80" height="80" alt="iPad"></td>
            <td><button class='wisknop'>wis</button></td>
          </tr>
          <tr>
            <td>boek</td>
            <td>34</td>
            <td><img src="../images/book.jpg" width="80" height="80" alt="book"></td>
            <td><button class='wisknop'>wis</button></td>
          </tr>
          <tr>
            <td>voetbal</td>
            <td>120</td>
            <td><img src="../images/football.jpg" width="80" height="80"
  alt="voetbal"></td>
            <td><button class='wisknop'>wis</button></td>
          </tr>
        </tbody>
      </table>
    </section>
  </article>
  <footer>Copyright vdab</footer>
</div>
</body>
</html>
```

dat er zo uit ziet:

# DOM Attributen en methods uitgetest

## Demos van de DOM in het theorieboek (editie 2012)

### Boodschappenlijst

- brood
- margarine
- eieren
- spek
- koffie

### Klusjeslijst

- gordijnen strijken
- schilderen
  - keukenvenster
  - garagedeur
- mol vangen

### Wishlist

| Wat | prijs | foto | wissen |
|---|---|---|---|
| iPad | 450 |  | wis |
| boek | 34 |  | wis |
| voetbal | 120 |  | wis |

**Opdracht 1:**

☞ **voeg een item "*koffie*" toe achteraan de boodschappenlijst**

Strategie:

- we moeten een referentie leggen naar de boodschappenlijst

  - probleem: de `ul` van de boodschappen heeft geen `id`, dus een rechtsreekse referentie leggen zal niet gaan…
    de `ul` bevindt zich wel in een `section` met een `id`

- we moeten een nieuw `li` element aanmaken en het dan toevoegen aan de lijst

Uitvoering:

```
var secBoodschappen          = document.getElementById('huishouden');
var lijstBoodschappen        = secBoodschappen.getElementsByTagName('ul')[0];
var nieuwItem                = document.createElement('li');
var tekst                    = document.createTextNode('koffie');
nieuwItem.appendChild(tekst);
lijstBoodschappen.appendChild(nieuwItem);
```

Bespreking:

- de method `document.getElementById()` wordt gebruikt om een referentie naar dat `section` element te leggen.
  De variabele *secBoodschappen* is nu een `elementNode`
- de method `.getElementsByTagName()` returnt een `nodeList` van alle `ul` elementen die zich in het `section` element bevinden.
  Merk op dat we deze method niet op `document` uitvoeren, maar rechtstreeks op *secBoodschappen*
- met `.getElementsByTagName()[0]` maken we een variabele *lijstBoodschappen* aan die refereert naar het **eerste** `ul` element in die nodeList
- met `document.createElement('li')` maken we een variabele *nieuwItem* aan die een leeg `li` element bevat
- met `document.createTextNode('koffie')` maken we een `textNode` aan in de var *tekst*
- we voegen de `textNode` toe aan het `li` element met `appendChild`
- we voegen het `li` element toe aan de `ul` lijst met `appendChild`

Een alternatieve en snellere manier om een referentie te leggen naar de boodschappenlijst krijg je met `querySelector`:

```
…
var lijstBoodschappen   = document.querySelector('#huishouden ul');
…
```

Hier leggen we de referentie naar de **eerste** ul van de `nodeList` van alle `ul's` die voorkomen in de `id` `#huishouden`.

**Opdracht 2:**
Deze opdracht demonstreert dat er veel wegen naar Rome leiden (en dat diegenen

die de kortste weg kennen er eerst zullen zijn…)

☞ **kleur de achtergrond van de even rijen van de tabel geel en de tekstkleur bruin, de achtergrond van de oneven rijen roze, de tekstkleur blauw**

Strategie:

- we leggen een referentie naar de even- en de oneven rijen van de tabel
  - deze twee referenties zijn *collections* (maps): het zijn arrays van meerdere elementen waardoorheen we zullen moeten lussen
- voor elk element in die collections stellen we de **inline style** eigenschap in

Uitvoering:

```
var evenRijen  = document.querySelectorAll('tbody tr:nth-child(odd)');
var onevenRijen       = document.querySelectorAll('tbody tr:nth-child(even)');
for (i=0;i<evenRijen.length;i++){
        //blauw op roze
        evenRijen[i].style.backgroundColor   = "#FFD88A";
        evenRijen[i].style.color                    = "#3369D7";
}
for (i=0;i<onevenRijen.length;i++){
        //bruin op geel
        onevenRijen[i].style.backgroundColor        = "#FF8AB0";
        onevenRijen[i].style.color                  = "#433025";
}
```

Bespreking:

- met de method `document.querySelectorAll` maken we een collection aan
- we gebruik de CSS selector `tbody tr:nth-child(n)` om de even en oneven `tr` elementen te selecteren
- we lussen met een for doorheen alle items van die collections en stellen de `style` eigenschappen `backgroundColor` en `color` in.

  Als je nu met een webtool de betreffende rijen controleert zul je het **inline** `style` attribuut opmerken.

De strategie om met inline styles te werken is effectief en heeft een belangrijk voordeel: inline styles hebben **altijd voorrang op stylesheetrules.** je hoeft je dus geen zorgen te maken over één of andere stylesheetrule die hier de overhand krijgt.

Het nadeel is natuurlijk dat je voor elke opmaak een lijntje bij moet schrijven: het wordt snel erg omvangrijk…

Een **alternatief** dat het aangehaalde probleem oplost is te werken met **CSS classes** die je voorziet in je stylesheet. In het interne stylesheet bovenaan de pagina merk je de volgende rules op:

```
table.flashy tr.oneven {background-color: #FFD88A; color:#3369D7}
table.flashy tr.even {background-color: #FF8AB0; color:#433025}
```

We wijzigen het script:

```
…
for (i=0;i<evenRijen.length;i++){
        //blauw op roze
        //evenRijen[i].style.backgroundColor              = "#FFD88A";
        //evenRijen[i].style.color                        = "#3369D7";
        evenRijen[i].setAttribute("class","even");
        }
for (i=0;i<onevenRijen.length;i++){
        //bruin op geel
        //onevenRijen[i].style.backgroundColor            = "#FF8AB0";
        //onevenRijen[i].style.color              = "#433025";
        onevenRijen[i].setAttribute("class","oneven");
        }
```

Bespreking:

- met de method `.setAttribute` geven we elk element in de collection een class attribuut met zijn respectieve waarde.
  Controleer dat met een webtool
- De browsers past onmiddellijk de bestaande rules in het stylesheet toe

Deze methode heeft het voordeel dat alle nieuwe opmaak in het stylesheet gezet kan worden en dat de scripting beperkt wordt toe het toekennen van de class. Een mogelijk nadeel is dat `setAttribute` de inhoud van het `class` attribuut volledig overschrijft: bestaande classes worden overschreven. Daar zou je via scripting een oplossing voor moeten zoeken.

Maar snelle (CSS) meisjes en jongens onder jullie zullen al opgemerkt hebben dat we hier **werk voor niets** aan het doen zijn…
Wie goed CSS kent beseft dat voor een statische opmaak van deze tabel (onmiddellijke toepassing toe bij de `load` van de pagina) **scripting onnodig is, CSS alleen is voldoende**:

- Zet het volledige script in commentaar
- Kopieer de 2 class selectors in het stylesheet, commentarieer de oorspronkelijke en pas de kopies aan:

```
/*
table.flashy tr.oneven {background-color: #FFD88A;color:#3369D7}
table.flashy tr.even {background-color: #FF8AB0;color:#433025}
*/
table.flashy tr:nth-child(even) {background-color: #FF8AB0; color:#433025}
table.flashy tr:nth-child(odd) {background-color: #FFD88A; color:#3369D7}
```

Bespreking:

- de selector `tr:nth-child(n)` past de style rules onmiddellijk en altijd toe
- scripting is niet nodig

Scripting zou wel nodig zijn als je **dynamisch** wil werken, bijvoorbeeld als je de rij wil oplichten als er op de knop geklikt wordt. Dan kan je één van de twee vorige strategiën gebruiken.

**Opdracht 3:**
Deze opdracht demonsteert eventhandlers.

☞ **een hyperlink "*wis*" verwijdert de betreffende rij uit de tabel**

Strategie:

- de hyperlinks "*wis*" moeten een event handler toegewezen krijgen
- deze handler is een functie "*verwijderRij*" die het echte werk voor zijn rekening neemt: de eigen rij verwijderen uit de DOM
  - er moet eerste een bevestiging gevraagd worden
  - daarna moet de rij waarin de knop zich bevindt, verwijderd worden

Uitvoering:

we zetten de structuur op

```
…
var wisLinks = document.querySelectorAll('a.wislink');
//eventhandler registratie
for (i=0;i<wisLinks.length;i++){
        wisLinks[i].addEventListener('click', verwijderRij, false);
        }

function verwijderRij(e){
/*
eventhandler: verwijdert de huidige produktrij
*/
e.preventDefault();
var antwoord = window.confirm("dit produkt verwijderen?");
if (antwoord === true){
        console.log('verwijder OK');
        }
}
```

Bespreking:

- met de method `document.querySelectorAll` maken we een collection van hyperlinks met de class "*wislink*" aan
- we registreren een eventhandler "*verwijderRij*" voor elke item in deze collection: dit doen we met de method `addEventHandler` waarin we het

soort event, de naam van de eventhandler functie (zonder haakjes) en het argument `useCapture` plaatsen

- de eventhandler *verwijderRij* is een functie en heeft een argument `e`, die het event (`click`) voorstelt:
  - de klik gebeurt op een hyperlink. Hyperlinks hebben een ingebouwde eventhandler voor `click`: ze brengen je namelijk naar een andere pagina. Die moeten we neutraliseren, daarom gebruiken we `e.preventDefault()`
  - we gebruiken `window.confirm` om een boolean (true/false) waqarde te krijgen op onze vraag
  - als deze variabele `true` is, zullen we de rij verwijderen

Nu diepen we het script uit, met name in deze laatste if structuur:

```
…
if (antwoord === true){
        var rij       = this.parentNode.parentNode;
        var tbodie    = rij.parentNode;
        tbodie.removeChild(rij);
  }
…
```

Bespreking:

- omdat dit een eventhandler is bevat het keyword `this` de hyperlink: hierop wordt geklikt.
- de rij (het `tr` element) is dus te bereiken door de `parentNode` van de `parentNode` van de link te nemen: zo zit deze tabel in elkaar
- een rij kan zichzelf niet wissen, enkel zijn *parent* kan dat, daarom maken we nog een referentie naar het `tbody` element, die de `parentNode` van de `tr` is
- met de method `.removeChild` verwijderen we de volledige rij uit de tabel

De strategie om vanuit het keyword `this` een element in de DOM tree te bereiken is meestal de goede als de DOM structuur niet te ingewikkeld is: hier staat de hyperlink in de rij, dus blijft de referentie eenvoudig.

Is de tabelstructuur te complex of staat de event target (de link) ergens anders, dan zal je moeten gebruik maken van `id`'s in de rij. Dat zal het script echter wat complexer maken.

### 5.7.4 Properties

Toegankelijk op de gebruikelijke manier, bijvoorbeeld `document.cookie` of `document.documentElement`. Er zijn heel wat DOM0 properties die met DOM2 en CSS minder interessant geworden zijn, maar nog perfect geldig: `document.anchors, document.bgColor, document.links`,…

| Property | return type | beschrijving |
|---|---|---|
| `body` | `element` | DOM0: het `body` element |
| `cookie` | `String` | DOM0: leest en zet de cookies geassocieerd met het huidige document |
| `documentElement` | `element` | Het **root** element van het `document`.<br>Voor een XHTML/HTML document is dat het `html` element, voor een XML document kan dat eender wat zijn uiteraard |
| `firstChild` | `Node \| null` | de eerste directe *child node* van het document.<br>niet noodzakelijk het rootelement, het kan ook een *Processing Instruction* zijn |
| `forms` | `Nodelist` | de lijst van alle `form` elementen in het document |
| `images` | `Nodelist` | de lijst van alle images in het document |
| `implementation` | `DOMImplementation` | returnt een `DOMImplementation` object geassocieerd met het document. |
| `location` | `location` | DOM0: Leest of zet de URL van het document. zie opmerkingen |
| `namespaceURI` | `String \| ""` | returns de XML namespace voor het document |
| `referrer` | `String \| ""` | geeft de URI van de pagina die naar deze pagina linkte. Leeg indien de gebruiker direct kwam |
| `styleSheets` | `Nodelist` | de lijst van alle `stylesheet` objecten geassocieerd met het document |
| `title` | `String \| ""` | Leest of zet de `title` van het document |
| `URL` | `String` | de URL van het document . . zie opmerkingen |

Bespreking van sommige van deze eigenschappen:

- `document.location` en `document.URL`: gebruik beter `window.location`
- `document.implementation` laat je toe te achterhalen of een DOM module ondersteund wordt door de browser. Dit object heeft daarvoor een method `hasFeature`:

  ```
  alert(document.implementation.hasFeature("Range","1.0");
  ```

- De `styleSheets` Nodelist (collection) bevat alle externe en interne stylesheets die gekoppeld zijn aan het document.
  zie het hoofdstuk DHTML

### 5.7.5 Methods

De belangrijkste methods van het `document` object:

| | |
|---|---|
| `close()` | DOM0: deze methods stopt het schrijven |

| | | |
|---|---|---|
| | | naar een document, geopend met document.open() |
| `createAttribute(name)` | `attributeNode` | maakt en returnt een attribuut node aan met de gegeven name |
| `createCDATASection(data)` | `CDATA` | maakt en returnt een CDATA section node aan met de meegegeven data |
| `createComment(data)` | `attributeNode` | maakt en returnt een comment node aan met de meegegeven data |
| `createDocumentFragment()` | `documentFragment` | maakt en returnt een lege `documentFragment` node |
| `createElement(tagName)` | `element` | maakt en returnt een lege `element` node van de gegeven tagName |
| `createEvent(type)` | `event` | maakt en returnt een event van het gegeven type. Het event moet dan geïnitialiseerd worden en dan doorgegeven met `element.dispacthEvent` |
| `createTextNode(data)` | `textNode` | maakt en returnt een Text node met de meegegeven data |
| `createRange()` | `range` | maakt en returnt een leeg `range` object |
| `createTreeWalker()` | `treeWalker` | maakt en returnt een leeg `TreeWalker` object |
| `getElementById(id)` | `element\| null` | geeft een referentie naar het `element` met de meegegegeven `id` |
| `getElementsByName(name)` | `Nodelist` | returnt een lijst van elementen met het gegeven `name` attribuut |
| `getElementsByTagName(tagName)` | `Nodelist` | returnt een lijst van elementen van het gegeven type |
| `open()` | | opent een document om naar te schrijven |
| `write()` | | schrijft tekst naar het document |

Bespreking:

- `document.open(),document.write(),` en `document.close()` laten je toe de inhoud van het huidige document te overschrijven.
  Niet verwarren met `window.open()` dat een nieuw (popup) venster opent.

  ```
  document.open()
  document.close()
  ```

  uitvoeren in een document verwijdert alle vroegere inhoud.

De method `document.write()` werd vroeger in JS zeer veel gebruikt om dynamisch inhoud te schrijven naar een pagina maar is sedert DOM2 volledig achterhaald

- de `create`… methods maken allerhande nodetypes of documentgedeelten aan.
- Het zijn vooral de `createElement()` en `createTexNode()` methods die voor ons van belang zijn.

  Om bijvoorbeeld een `br` element aan te maken doen we

  ```
  var nl = document.createElement('br');
  ```

  Waarna we deze node ergens aan zullen moeten koppelen met `element.appendChild` of `element.insertBefore`.
  Het `br` element is een EMPTY element dus heeft nooit *children*.

  Een `p` element daarentegen kan een tekstuele inhoud hebben die we kunnen aanmaken:

  ```
  var pee = document.createElement('p');
  var tee = document.createTextNode('wie lust een kopje thee?');
  var koffie = document.createTextNode('of liever een kopje
  koffie?');
  pee.appendChild(tee);
  pee.appendChild(koffie);
  ```

  zoals je merkt kunnen meerder text nodes aan een element toegevoegd worden, je zal het verschil niet merken.
  Toch is het beter ze nadien te normaliseren tot één text node:

  ```
  pee.normalize()
  ```

- de method `getElementById()` returnt één element met de gegeven `id` of `null` indien die niet gevonden is.
  IE7 returnt foutief ook een element met eenzelfde `name` attribuut als hij die eerst tegenkomt. De waarde van een `name` attribuut moet niet uniek zijn.
  In onderstaand voorbeeld returnt `getElementById('description')` in IE7 een verkeerd element:

  ```
  <meta name="description" value="mijn website" />
  …
  <textarea name="description" id=" description"></textarea>
  ```

- In de meeste gevallen is het raadzaam te controleren of er wel een element node gevonden is vooraleer verder te gaan met de code:

  ```
  var voornaam = document.getElementById('voornaam');
  if(voornaam) {
     …
  }
  ```

- de method `getElementsByTagName()` en `getElementsByName()`returnen een `Nodelist` (een **collection**) van nodes.
  Daarom moet je hun returnwaarde altijd **behandelen als een array**, zelf al zit er maar één element in:

```
var diefs =  = document.getElementsByTagName('div');
var eersteDief = diefs[0];
var laatsteDief = diefs[diefs.length-1];
```

- je kan ook `getElementsByTagName('*')` gebruiken om **alle** elementen te selecteren
- Merk ook op dat deze method ook toepasselijk is op een `element` node, niet enkel op de `document` node. zo kan je veel sneller een *child node* van een node zoeken:

```
var tabel = document.getElementById('mijnTabel');
var eersteRij = tabel.getElementsByTagName('tr')[0];
```

## 5.8 de DOM Tree manipuleren

Hier bespreken we een aantal veelgebruikte handelingen die je kunt uitvoeren met de DOM Tree. Onthoud ook:

- **attribuutnodes** worden nooit voorgesteld in de DOM Tree. Als je dus een functie schrijft die doorheen de nodes gaat, kom je attribuutnodes niet tegen. gebruik de specifieke DOM properties en methods voor attributen

### 5.8.1 doorheen de *Tree* wandelen

Om doorheen alle nodes van de DOM Tree te lussen gebruiken we best een recursieve functie:

```
function walkTree(node) {
   if (node == null) return;
   // doe hier iets met de node, bijvoorbeeld  alert(node.nodeName)
   for (var i = 0; i < node.childNodes.length; i++) {
      walkTree(node.childNodes[i]);
   }
}

var start = document.getElementById("start");
walkTree(start);
```

Deze functie start met een bepaalde node en gaat dan doorheen alle *childNodes* ervan. Voor elk van die *childNodes* roept de functie zichzelf nog eens op.

Merk op dat deze voorbeeldfunctie geen enkel onderscheid maakt tussen de types nodes: het is aan jou om te testen welk `nodeType` je hebt, vooraleer je er een property/method op los laat. Zo heeft het weinig zin de `tagName` van alle nodes op te vragen want enkel `element` nodes hebben deze eigenschap.

### 5.8.2 gezocht: voorouder

De vorige routine zoekt de *Tree* af naar beneden: alle *children*. Maar soms is het ook nodig doorheen de *Tree* te zoeken in de opwaartse richting: zoek de juiste *parent*!

Onderstaande functie returnt de eerste *parent* van een bepaald elementtype, bijvoorbeeld een `tr` element :

```
function zoekParent(n,strE){
/*
*returnt de eerstvolgende parentNode van een bepaald elementtype
*
*@param n        startNode van wie de parent gezocht wordt
*@param strE     string elementtype, vb "TR"; caps afhankelijk van html of xhtml
*@return Node/null
*/
while(n=n.parentNode){ //zolang dit lukt nemen we onmiddellijk de parentNode
  if(n.nodeName.toLowerCase()==strE.toLowerCase()){return n;}
```

```
}
return null;
}
```

Je kunt het makkelijk aanpassen naar je eigen noden, bijvoorbeeld een *parent* van een bepaalde CSS class.

### 5.8.3 een node aanmaken

De method die je gebruikt om een node aan te maken hangt af van het type node dat je wilt verkrijgen:

`document.createElement()` voor een element
`document.createTextNode()` voor een text node
`document.createComment()`,`document.createAttribute()`,
`document.createCDATASection()`, `document.createDocumentFragment()`, …

Details van deze laatste methods kan je nagaan in de W3C DOM reference. Wegens zelden gebruikt laten we ze hier achterwege.

Om een element aan te maken dat een tekstuele inhoud heeft maken we eerst de element node aan, maken daarna een text node aan, voegen die toe en voegen daarna de nieuwe elementnode toe in de DOM Tree:

```
var lijst = document.getElementById('lijst');
var nieuwItem = document.createElement('li');
var tekst = document.createTextNode('laatste item')
nieuwItem.appendChild(tekst);
lijst.appendChild(nieuwItem);
```

Een nieuwe node wordt ook aangemaakt met cloneNode() en importNode(): deze methods returnen copies van een andere node.

### 5.8.4 nodes kopiëren

De eenvoudigste manier om een node te kopiëren of te verplaatsen (kopiëren + verwijderen) is met de method `cloneNode()`. Zoals hierboven uitgelegd kan je met het argument *deep* bepalen of je alle children mee kopieert of niet:

```
var orig = document.getElementById('origineel');
var kopie = orig.cloneNode(true);
kopie.id = "kopie";
document.body.appendChild(kopie);
```

Als je een aantal nodes moet aanmaken kan `cloneNode()`wat kortere code opleveren t.o.v. `createElement()`. Veel tijdswinst in de uitvoering van het script zal je er niet mee maken. Hou er ook rekening mee dat je een eventuele `id` van de kloon ook moet aanpassen: een `id` moet uniek zijn in het document!

### 5.8.5 Een node invoegen

Om een node in te voegen ergens in de Tree moet je steeds een referentie *parent* hebben: in deze *parent* wordt ingevoegd. Veronderstel een lijst:

```
<ul id="lijst">
  <li>eerste item</li>
  <li>tweede item</li>
  <li>derde item</li>
</ul>
```

Ofwel gebeurt dat als laatste child met `appendChild():`

```
var lijst = document.getElementById('lijst');
var nieuwItem = document.createElement('li');
nieuwItem.appendChild(document.createTextNode('laatste item'));
lijst.appendChild(nieuwItem);
```

Ofwel kiezen we de plaats met `insertBefore()` zoals hier waar we de node invoegen al voorlaatste:

```
lijst.insertBefore(nieuwItem, lijst.lastChild);
```

### 5.8.6 nodes verwijderen

Een node kan enkel verwijderd worden via zijn *parent* met de method `removeChild()`. Vermits alle nodes een *parent* hebben, met uitzondering van de `document` node, kunnen we vanuit de node zelf zijn parent oproepen:

```
var start = document.getElementById("start");
start.parentNode.removeChild(start);
```

### 5.8.7 Attributen lezen en instellen

Er bestaan heel wat properties en methods die *beweren* attributen te verwerken. Een overzicht:

het array `attributes[]`
de methods `hasAttributes()` en `hasAttribute()` testen het bestaan van een attribute
de method `getAttribute()` leest een attribuutwaarde
de method `setAttribute()` zet een attribuutwaarde
de directe properties element.propertyname

*De verschillen tussen de browsers zijn hier het grootst, inconsistentie alomtegenwoordig …*

Veronderstel een `div` element:

```
<div id="p1" class="rood" style="border: 1px solid black;" align="left"
  onclick="alert('yes')" vdab="oostende">Dit is een textNode in p1</div>
```

De eerste (en waarschijnlijk beste) manier is een attribuut direct als property aanspreken:

```
var p1 = document.getElementById('p1');
alert(p1.id); // returnt 'p1'
alert(p1.class); // FF: returnt undefined  IE: error
alert(p1.className); // returnt 'rood'
alert(p1.style); // returnt [object]
alert(p1.style.border); // returnt '1px solid black '
alert(p1.align); // returnt 'left'
alert(p1.onclick); // returnt  function onclick(event){alert("yes");}
alert(p1.vdab); // FF returnt undefined IE: 'oostende'
```

Denk er aan dat sommige properties, zoals `style`, objecten returnen, maar ook dat is browserafhankelijk.

Een andere mogelijkheid is `getAttribute()` gebruiken: dat levert ook goede resultaten op hoewel je altijd goed moet opletten over het type dat je terugkrijgt:

```
alert(p1.getAttribute('id'); // returnt 'p1'
alert(p1.getAttribute('className'); // FF returnt null IE 'rood'
alert(p1.getAttribute('class'); // FF returnt 'rood' IE null
alert(p1.getAttribute('style'); // FF returnt '1px solid black ' IE [object]
alert(p1.getAttribute('style.border'); // returnt null
alert(p1.getAttribute('align'); // returnt 'left'
alert(p1.getAttribute('onclick'); // FF returnt alert("yes");}
                        // IE function onclick(event){alert("yes");}
alert(p1.getAttribute('vdab'); // returnt 'oostende'
```

Zoals je merkt zijn hier ook verschillen/gelijkenissen tussen de browsers verbazend.

Om een attribuut in te stellen gebruiken we opnieuw liefst de directe methode waar het attribuut als *property* aangesproken wordt:

```
var p1 = document.getElementById('p1');
with(p1){
  id="c4";
  className = "groen";
  style.border="none";
  style.font="normal small-caps 120% Courier";
  align="right";
  vdab="haasrode";
  onclick="alert(this.vdab)";
}
```

zullen alle attributen gewijzigd zijn met uitzondering van

- In FF: 'vdab' dat eerder ook al niet als een geldig attribuut erkend werd en ook niet gewijzigd zal zijn
- 'onclick': in FF niet en IE wel gewijzigd, maar niet werkend.

Met `setAttribute()` zijn de resultaten ook niet schitterend:

```
setAttribute('id','c4'); // OK
setAttribute('class','groen'); // niet IE
setAttribute('style','border:none'); niet IE
setAttribute('style','font: normal small-caps 120% Courier'); //niet IE
setAttribute('align','right'); //niet IE
setAttribute('vdab','haasrode'); // OK
setAttribute('onclick','alert(this.vdab)'); // niet IE
```

Enkele praktische tips om af te sluiten:

- neem je tijd en test goed uit
- gebruik `className` om de CSS class te lezen/schrijven
- probeer eerst een attribuut **direct** (als een property) te lezen/schrijven: `el.id, el.align, el.style`
- probeer dan `setAttribute()`: in dat geval gebruik je de echte naam van het html attribuut zoals 'class'
- let goed op het **type waarde** dat je terugkrijgt: soms is dat een `string` waarde, soms een `object`. Van objecten – zoals `style` – moet je de sub-properties lezen/schrijven als `string` waarden.
- Probeer een inline Event Handler - zoals een `onclick` of een `mouseover` – niet te wijzigen door het attribuut te schrijven!
  Dat kan enkel op een zekere manier met Event Registratie (zie Events)

## 5.8.8 de CSS class lezen en instellen

Elementen kunnen een `class` attribuut hebben die één of CSS classes bevat (tokens). Die waarden worden gescheiden door een spatie:

```
<div class="menuitem actief" id="mijnMenuItem"> menu item </div>
```

Als je een element test of deze een `class` heeft moet je er dus mee rekening houden dat die class niet noodzakelijk alleen staat.

### 5.8.8.1 className

Alle browsers ondersteunen de DOM property `className`. Deze leest of stelt de volledige inhoud van het `class` attribuut in.

```
var el = document.getElementById(mijnMenuItem);
alert(el.className); // returnt "menitem actief"
```

Als we simpelweg de property gebruiken om te schrijven, overschrijven we alle vorige inhoud:

```
var el = document.getElementById(mijnMenuItem);
el.className = "kop"
alert(el.className); // returnt "kop"
```

Om te testen of een CSS class aanwezig is  kunnen we ook niet gewoon vergelijken, we gebruiken best een *Regular Expression*:

```
var el = document.getElementById(mijnMenuItem);
var klasse = "actief";
var re = new RegExp('\\b' + klasse + '\\b');
var antwoord = re.test(el.className); //
if(antwoord===true) …
```

De Regex functie `test()` returnt een boolean die zegt of de *re* **voorkomt** in de string `className`.

Op dezelfde manier moeten we er ook rekening mee houden dat als we een class toekennen aan een element, dat er al classes aanwezig kunnen zijn. We mogen ze niet overschrijven:

```
var el = document.getElementById(mijnMenuItem);
var klasse = "actief";
el.className += " " + klasse;
```

Conclusie: `className` maakt het niet eenvoudig te testen of een class aanwezig is en om er eentje aan toe te voegen. Of…

#### 1.1.1.1 classList

Met de HTML5 API's kwam ook de nieuwe DOM property `classList`. Deze wordt slechts ondersteund door FF4+, Chrome8+, Safari5.1+, O11.5+, Android, maar niet in IE9, misschien IE10?

`classList` is een `object` met vijf methods: `add, remove, toggle, contains, item` en één property: `length`.

- je kan het gebruiken om gemakkelijk een CSS class toe voegen:

```
var el = document.getElementById(mijnMenuItem);
el.classList.add('rood'); // voegt de class rood toe
```

- om een CSS class te verwijderen (zonder de andere classes te wissen):

```
el.classList.remove('rood');
```

- om een CSS class te wisselen (aan/af):

```
el.classList.toggle('rood');
```

- om te testen of een class aanwezig is:

```
el.classList.contains('rood'); // return boolean
```

- om het aantal classes te kennen:

```
alert(el.classList.length);
```

- als je wil weten wat de tweede class is in de lijst van tokens, dan gebruik je:

```
alert(el.classList.item(1)); // return string
```

`ClassList` is een enorme verbetering t.ov. `className`, maar is door het gebrek aan ondersteuning nog niet echt bruikbaar.
Een JS libraries zoals jQuery gebruiken is momenteel nog aan te raden.

### 5.8.9 URL's lezen

Nogal wat attributen zoals `href, src, action`, etc.… in elementen als `a, img, script, link, form` , e.a. bevatten een URL-waarde.

- als je het attribuut direct leest zoals `el.src, el.href, el.action` krijg je steeds het absolute pad
- met `getAttribute()` krijg je de waarde die werkelijk ingevuld is in het attribuut

Veronderstel twee hyperlinks:

```
<a href="../index.html" id="lienk1" title="een relatieve hyperlink">index</a>
<a href="http://www.vdab.be" id="lienk2" title="een externe hyperlink">vdab</a>
```

dan verschillen de returnwaarden afhankelijk van de gebruikte methode:

```
lienk1.href; //returnt  http://localhost/javascript/index.html'
lienk1.getAttribute('href') ; //returnt '../index.html'

lienk2.href; //returnt 'http://www.vdab.be/'
lienk2.getAttribute('href') ; //returnt 'http://www.vdab.be'
```

Opmerking: in IE versies vóór IE8 is het pad steeds absoluut ongeacht de methode.

### 5.8.10 normaliseren

De methode `normalize()` versmelt meerdere *TextNodes* tot één, zodanig dat er geen *sibling* of *lege TextNodes* onder dezelfde *parent* bestaan.

```
Node.appendChild(tN1);
Node.appendChild(tN2);
```

```
Node.appendChild(tN3);
Node.normalize();
```

In bovenstaand voorbeeld worden meerdere textNodes aan een element toegevoegd. Hoewel je er in je browser niet veel van zal merken, zou dit element drie *childNodes* hebben zonder, en slechts één *childNode* met normalisering.

### 5.8.11 het *white-space* 'probleem'

Als je met de Mozilla DOM Inspector een pagina bekijkt zullen je de *textNodes tussen de elementen* opvallen: dit zijn de 'Enters', spaties, tabulaties etc. in de tekst, de zogenaamde *'whitespace'*.
*Gecko*-gebaseerde browsers (FireFox) maar ook Opera tonen deze 'whitespace' in de DOM Tree, Internet Explorer negeert ze.
Als je dus een script schrijft dat doorheen de Tree 'wandelt' hou er rekening mee dat niet alle *childNodes* `element`en zijn: er zitten gegarandeerd heel wat `textNodes` tussen. Een `firstChild` of een `nextSibling` kan dus een spatie of een 'Enter' returnen, simpelweg omdat je in de code een nieuwe lijn gezet hebt.

De oplossing is eenvoudig: voor functies die doorheen nodes 'wandelen', bouw steeds een `nodeType` test in die `textNodes` eruit filtert. Mozilla heeft zelf een script beschikbaar, nodomws.js, (zie referenties achteraan de cursus) die een aantal vervangende functies bevat (voor nextSibling, previousSibling, etc..) die nodes met witruimte negeren.
DOM Level3 zou een antwoord moeten bieden op dit probleem door een definitieve standaard voor te stellen.

### 5.8.12 DOM validatie

DOM2 is niet in staat een gewijzigde DOMtree te valideren tegen de DTD.
Met andere woorden, als je nieuwe elementen aanmaakt of andere verwijdert, moet je er *zelf* op letten dat de gewijzigde tree ook geldige HTML blijft!
Ook de online validators zoals het W3C valideren enkel de statische HTML, voor wat dynamisch aangemaakt is zijn ze blind.
DOM3 zal wel een Validate module bevatten.

# 6 Debuggen en foutafhandeling

Fouten in je programma vermijden is dikwijls een kwestie van discipline. Toch sluipen er altijd fouten in die je niet had voorzien. Debuggen is het proces van ze op te sporen. Daarnaast kan je ook fouten proberen op te vangen die je onmogelijk kan voorzien omdat ze afhangen van de browser van de gebruiker.

## *6.1 Soorten fouten*

Er zijn twee groepen fouten te onderscheiden:

**syntaxfouten**

omdat het script pas gecompileerd wordt tijdens uitvoering, lijkt het of ze *runtime* fouten zijn. Het script kan niet uitgevoerd worden en stopt.

**Exceptions**

dit zijn de echte *runtime* fouten die slechts ontdekt worden tijdens de uitvoering. Het zijn bijvoorbeeld logische fouten in het programma of fouten in arraybounds. Maar *exceptions* kunnen ook door de programmeur ingebouwd worden.

Javascripts worden door de browser gecompileerd tijdens het laden van de pagina. Daardoor kunnen syntaxfouten de verdere uitvoering van dit en andere scripts blokkeren.

### 6.1.1 Syntaxfouten

Heel veel fouten komen voort uit schrijffouten **tegen de regels** van de taal: de **syntax**.
Het moet gezegd: moesten we beschikken over een goede Javascripteditor, dan kon die ons op de vingers tikken. Pas recent wordt ondersteuning voor JS ingebouwd in enkele ontwikkelomgevingen (Eclipse, Visual studio, NetBeans), een dedicated Javascript editor bestaat echter nog steeds niet.

Syntaxfouten kunnen NIET opgevangen worden met een `try...catch` statement!

Veel voorkomende syntaxfouten:

- **Case**: JS is **hoofdlettergevoelig**: de method `getElementByID()` bestaat niet
- **Aanhalingstekens** die verkeerd afgesloten worden
- **foute structuur**: verkeerd afsluiten van block statements { }

Wat doe je er aan? Manueel verbeteren duurt te lang:

- laat je code controleren door *JSLint (zie referenties achteraan)*

### 6.1.2 Exceptions

Runtime fouten zijn opgeworpen **Exceptions**, net als syntaxfouten steken ze pas de kop op bij uitvoering.

Veel voorkomende runtime fouten:

- **toekenning i.p.v. vergelijking**: `if(getal=3)` moet zijn `if(getal==3)`
  In dit voorbeeld wordt de `var` niet vergeleken met de waarde 3, maar wordt die waarde er juist aan toegekend. Omdat dat geen probleem oplevert evalueert die actie tot `true`.
  Dit hoeft niet noodzakelijk een fout op te leveren, het zal echter wel onverwachte resultaten hebben.
- **variable scope**: verwarring tussen *globale* en *lokale* variabelen van scripts: declareer **alle variabelen** consequent met het `var` keyword, beperk het gebruik van globale variabelen
- **array bounds**: er wordt een array-element opgevraagd met een indexgetal dat hoger dan het laatste item is of lager dan het eerste.
- ***object expected*** : je kan de DOM enkel gebruiken nadat de pagina volledig geladen is: wacht met `document.getElementById` tot het `load` Event heeft plaatsgehad.
- ***objectname has no properties***: het object waarnaar je refereert kan niet gevonden worden

Wat doe je er aan?

## 6.2 Foutafhandeling

Het gros van de fouten kan vermeden worden door correcte code te schrijven, maar niet altijd.
Met een `try...catch` en een `throw` statement kunnen we niet te voorzien situaties opvangen en een eigen `Error` object werpen.
Bijvoorbeeld om een verkeerd aantal argumenten van een functie te onderscheppen:

```
function deelTellerDoorNoemer(getal1,getal2){


    (if arguments.length!=2){
         throw new Error('Twee getallen verwacht');
    }
    else {
         if(getal2==0){
         throw new RangeError('Deler mag niet 0 zijn');
        }
        else{
         return getal1/getal2;
        }
    }
}


try {
  var breuk =
  deelTellerDoorNoemer(parseInt(document.form1.teller.value),parseInt(document.form1.n
  oemer.value));
}
catch (oException){
```

```
  alert(oException.message)
}
```

In een ander voorbeeld wordt een AJAX HTTP request gemaakt. Hier merk je hoe voor IE de aanwezigheid van twee ActiveXObjecten uitgeprobeerd wordt. Dit is een situatie die je onmogelijk kan voorzien: welke browser heeft de gebruiker.
Hierop kunnen we enkel anticiperen:

```
function makeRequest(url) {
        var httpRequest;

        if (window.XMLHttpRequest) { // Mozilla, Safari, ...
            httpRequest = new XMLHttpRequest();
            if (httpRequest.overrideMimeType) {
                httpRequest.overrideMimeType('text/xml');
                // See note below about this line
            }
        }
        else if (window.ActiveXObject) { // IE
            try {
                httpRequest = new ActiveXObject("Msxml2.XMLHTTP");
                }
                catch (e) {
                           try {
                                httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
                               }
                             catch (e) {}
                          }
                                       }

        if (!httpRequest) {
            alert('Giving up :( Cannot create an XMLHTTP instance');
            return false;
        }
        httpRequest.onreadystatechange = function() { alertContents(httpRequest); };
        httpRequest.open('GET', url, true);
        httpRequest.send('');

    }
```

## 6.3 Debuggen

Debugging is het opsporen van runtime errors. Daar bieden de browsers tot nu toe zeer weinig steun.

### 6.3.1 DIY

Do-It-yourself is wat de meeste beginnende en de gehaaste JS-programmeurs toepassen om een fout te vinden bij gebrek aan een gesofisticeerde ontwikkelomgeving.

De meest directe methode van debuggen is het gebruik van de `alert()` functie om de waarde van een variabele of expressie te weten te komen. Enkele nieuwe debuggers zoals Firebug maken deze manier overbodig, maar het heeft het voordeel dat je snel iets kan te weten komen.
Je kan altijd de `alert()` statements commentariëren om ze later eventueel te reactiveren.

```
//alert(document.getElementById)
var getal1 = document.getElementById('getal1').value;
var getal2 = document.getElementById('getal2').value;
//alert(getal1 + "," + getal2 );
```

Een zeer vervelend nadeel van het gebruik van `alert()` statements verschijnt als je programma in een oneindige lus verzeilt raakt: de `alert()` 's blijven maar komen en je hebt geen enkele manier om het programma te stoppen.
Er is geen enkele toetsencombinatie die je daaruit kan helpen: er zal niets anders opzitten dan de browser te sluiten...

Een alternatief voor `alert()` statements is de Javascript console gebruiken en berichten via console.log sttements tonen.

```
var getal1 = document.getElementById('getal1').value;
var getal2 = document.getElementById('getal2').value;
console.log(getal1 + "," + getal2) ;
```

### 6.3.2 Tools

Je kan beter gebruiken maken van specifieke debuggers in elke browser. Die zijn:

- FireBug plug-in voor FireFox
- Developer Tools in Chrome (ingebouwd)
- Developer Tools in IE (ingebouwd)
- … andere browsers hebben elk hun eigen tool

#### 6.3.2.1 Chrome developer tools

Lees de documentatie en volg een tutorial op
https://developers.google.com/chrome-developer-tools

#### 6.3.2.2 Firebug

Lees de documentatie en volg een tutorial op
http://getfirebug.com/wiki/index.php

#### 6.3.2.3 IE Developer tools

Lees de documentatie en volg een tutorial op
http://msdn.microsoft.com/en-us/library/ie/gg589507(v=vs.85).aspx

# 7 Javascript objecten

Javascript bevat een aantal interne objecten, zoals `Date`, `Math`, `Error` die je dikwijls nodig hebt, meestal om variabelen van een bepaald type aan te maken of te bewerken. Daarnaast kan je ook zelf objecten aanmaken: daarover gaat dit hoofdstuk niet, hier vind je een overzicht van de 'ingebouwde' JS objecten met al hun properties en methods.

Voor uitleg over hoe je zelf objecten kunt aanmaken of object georiënteerd programmeren verwijzen we je naar het hoofdstuk 15 "Object georiënteerd programmeren in JS".

Op de *begeleidende website* vind je ook de meeste properties en methods gedemonstreerd in voorbeeldbestanden.

## *7.1 variabelen afgeleid van JS objecten*

In tegenstelling tot een *primitief datatype* is een object-variabele een meer complexe waarde. Zo is een datum infeite een datum-tijd object en is een JS Object een verzameling eigenschappen met waarden.

Om een dergelijke variabele aan te maken heb je in alle gevallen de mogelijkheid een **constructor** te gebruiken. Een constructor is een functie die ons toelaat een objectvariabele aan te maken.

Je gebruikt een constructor functie met het `new` sleutelwoord:

```
var auto = new Object();
var vandaag = new Date();
var lijst = new Array();
var tekst = new String();
var getal  = new Number();
var f = new Function();
var e = new Error();
var b = new Boolean();
```

Dit worden allemaal objectvariabelen: we gebruikten de constructor functie (altijd met een hoofdletter) en het `new` sleutelwoord.

Maar voor sommige types is het **helemaal niet nodig** een objectvariabele te maken: in het geval van een `string` of een `number` worden die objectvariabelen toch onmiddellijk omgezet in een primitief type.

En voor echte objectvariabelen is er (meestal) een veel kortere en snellere manier : de **letterlijke schrijfwijze** (literal):

```
var auto = { }; // een object
var lijst = [ ]; // een array
var f = function() { } // een functie
var b = true; // een boolean
```

*gebruik steeds een primitief datatype of gebruik de letterlijke notatie indien mogelijk.*

Dat gaat niet altijd, bijvoorbeeld een datumobject kan je alleen met een constructor aanmaken.

## 7.2 Het Array object

Een *array* is een tabelvariabele: een datatype waarin je **geïndexeerde** (genummerde) **waarden** kan opslaan. Elke waarde is een ***element*** van het array en de ***index*** is het nummer waarmee je de waarde ophaalt.

### 7.2.1 arrays in javascript

JS arrays verschillen nogal van arrays uit andere programmeertalen

- In andere programmeertalen bestaat ook een *associatief array*, dat is een tabel geïndexeerd op basis van sleutels (keys), woorden dus. In JS bestaat hetzelfde maar dan noemen we dat een *object*.

- `Array` erft van `Object`.
  de index wordt omgezet naar een string en deze gebruikt als naam om de waarde op te halen.
  - gebruik `Array` als je van een index (een getal dus) gebruik kan maken
  - gebruik `Object` als je arbitraire termen zoals in een *associatief array* (zoals de "*naam*" van een persoon )wil gebruiken
- het is *niet nodig een lengte* aan een array mee te geven: ze zijn volledig dynamisch qua lengte als gevolg van het vorige punt
- het is *onnodig om er een data type* aan mee te geven: ze kunnen elke waarde bevatten, zelf andere arrays en functies.

### 7.2.2 een array aanmaken

Je kan een array op aanmaken:

- met een *constructor*
- als *array literal* : een lijst van initiële waarden

**de constructor:**

```
var tabel = new Array();
tabel[0] = 2;
tabel[1] = "javascript";
tabel[2] = false;
tabel[99] = undefined;
```

Eenmaal het array aangemaakt met het `new` sleutelwoord, kan je waarden toekennen aan individuele elementen via hun index.
Het is ook mogelijk (maar niet noodzakelijk) het array te dimensioneren met het aantal elementen in de constructor.

```
var tabel = new Array(4);
```

**als array literal:**

Een directe manier is het array onmiddellijk waarden mee te geven: we schrijven ze in **vierkante haakjes** en gescheiden door **komma's**:

```
var tabel = [2,"javascript",false,];
```

omdat er nog een komma blijft hangen in dit voorbeeld is de laatste (weggelaten) waarde `undefined`.

### 7.2.3 array elementen

Het aantal elementen in een array wordt gegeven door de property `length`.
Een enkel element kan bereikt worden via zijn index, genoteerd in *vierkante haakjes* [ ].
Het eerste element heeft index `0`, het laatste een index = `length-1`.

```
var tabel = new Array();
// toewijzing
tabel[0] = 2;
tabel[1] = "javascript";
tabel[2] = false;
tabel[99] = undefined; // indexgetallen moeten geen continue reeks vormen

// aflezen:

tabel.length; // returnt 100
tabel[0]; // returnt 2
tabel[tabel.length-1]; // returnt undefined
```

### 7.2.4 iteratie

Om alle elementen in een array te doorlopen wordt traditioneel een `for` loop gebruikt, waarbij de `length` property gebruikt wordt:

```
for (var i=0;i<tabel.length;i++){
        alert(tabel[i]);
        }
```

Vanaf *JS1.6* kan ook de `forEach` method gebruikt worden, zie verder.

### 7.2.5 elementen toevoegen

Javascript arrays zijn **volledig dynamisch**: elementen kunnen op elk moment toegevoegd worden.
Dit kan door een indexgetal te gebruiken of via de methods array.`push()` en array.`unshift()` (zie verder):

Het is belangrijk te noteren dat de *range van indexgetallen niet continue moet zijn*: JS arrays kunnen **sparse** zijn. Er zal slechts geheugen gebruikt worden voor de werkelijk toegekende waarden.

```
tabel[0] = 2;
tabel[9] = undefined;

// een for loop zal 10 elementen tonen, alle undefined uitgezonderd het eerste
tabel[tabel.length] = 3;
tabel[tabel.length] = 4; // voegt steeds aan het einde toe
```

### 7.2.6 elementen verwijderen

Twee mogelijkheden: de `delete` operator verwijdert de inhoud van het array item maar laat een `undefined` achter.

Beter is de `splice()` method te gebruiken die effectief een element verwijdert:

```
myArray = ['a', 'b', 'c', 'd'];
delete myArray[1]; // geeft  ['a', undefined, 'c', 'd']

myArray.splice(1, 1); // geeft  ['a', 'c', 'd']
```

### 7.2.7 Multidimensionele arrays

Een multidimensioneel array kan je maken door een array als element in een ander array te plaatsen. Omdat JS elk datatype toelaat, zijn combinaties van types mogelijk.
Een voorbeeld van een drie-dimensionele matrix:

```
var matrix = [[[7,8],[2,6]],[[9,5],[1,2]]];
alert(matrix[0][1][0]); // geeft 2
```

Bovenstaande alert vraagt het eerste element van het tweede van het eerste element…

### 7.2.8 array methods

Methodes van het `Array` object kunnen in drie groepen opgedeeld worden:
mutator methods die het array wijzigen,
*accessor* methods die bepaalde elementen er uit halen en
*iteration* methods die het array doorlopen.

#### mutator methods

wijzigen het array op één of andere manier.

#### 7.2.8.1 push() en pop()

De `push()` method voegt één of meerdere waarden toe aan het ***einde*** *van het array* en returnt de nieuwe `length` van het array.
`pop()` verwijdert de ***laatste*** waarde en returnt die:

```
var tabel = new Array();
tabel[0] = 50;
var hoelang = tabel.push(100, 200); // voegt 100 en 200 toe en returnt 3
var gewist = tabel.pop(); // verwijdert de laatste waarde en returnt die
alert(gewist);
```

#### 7.2.8.2 unshift() en shift()

de `unshift()` method voegt één of meerdere waarden toe aan het ***begin*** *van het array* en returnt de nieuwe `length` van het array.
`shift()` verwijdert de ***eerste*** waarde en returnt die:

```
var tabel = new Array();
  tabel[0] = 50;
var hoelang = tabel.unshift(100, 200);
var gewist = tabel.shift() // verwijdert de waarde 100 en returnt die
```

#### 7.2.8.3 reverse()

de `reverse()` method keert de volgorde van de arrayelementen om.

```
var tabel = ['jan','piet','pol'];
  tabel.reverse() // volgorde is nu  'pol','piet','jan'
```

#### 7.2.8.4 sort()

de `sort()` method sorteert het array ter plekke.

- gebruikt zonder argument gebeurt een alfabetische sortering
- gebruikt met een *comparison functie* sorteert het volgens het resultaat van de functie

Een comparison function is een eigen functie die je maakt met de argumenten a,b die sorteert volgens de returnwaarde:

- als *a* vóór b moet sorteren, moet de returnwaarde negatief zijn
- als b vóór *a* moet sorteren, moet de returnwaarde positief zijn
- als er geen sortering moet plaatsvinden, moet de returnwaarde 0 zijn

Een voorbeeld van een omgekeerd-alfabetische sortering

```
tabel.sort(function(a,b) {
  if (a>b) return -1;
  if (b>a) return 1;
  return 0;
});
```

Let hier op het gebruik van de anonieme functie.

### 7.2.8.5 splice()

de `splice()` method kan zowel gebruikt worden om elementen uit een array te *wissen* als om er elementen aan *toe te voegen*, vervangen dus. Het verschil zit in het aantal argumenten dat je gebruikt:

Een eerste argument is de *beginpositie* van waaruit de toevoeging/verwijdering moet beginnen.
Het tweede *optionele* argument zegt *hoeveel* elementen toegevoegd/gewist moeten worden.
Wordt dit tweede argument weggelaten dan worden - voor wissen - alle elementen tot het einde van het array gewist.

De method returnt de gewiste elementen:

```
var tabel = ['jan','piet','pol','ilse','kurt','lut','carole'];
var verwijderd = tabel.splice(5)
// returnt ['lut','carole'], tabel behoudt de rest
var eersteTwee = tabel.splice(0,2)
// returnt ['jan','piet'], tabel behoudt ['pol','ilse','kurt']
```

Volgen op deze twee positie-argumenten nog andere argumenten, dan worden die **ingevoegd** in het array:

```
var tabel = ['jan','piet','pol','ilse','kurt','lut','carole'];

var toegevoegd = tabel.splice(1,0,'ann','bart')
// returnt niets, tabel krijgt twee nieuwe waarden toegevoegd
var vervangen = tabel.splice(1,2,'veerle','tim')
//  returnt ['ann', 'bart'], die vervangen zijn door 'veerle','tim'
```

### Accessor methods

wijzigen het array niet, maar returnen een deel van het array

### 7.2.8.6 concat()

De `concat()` method maakt een nieuw array van het oorspronkelijke en voegt daarbij de argumenten toe. Deze argumenten kunnen een lijst van waarden of arrays zijn. Is het argument een array dan wordt dit geplet, m.a.w. het wordt niet als array ingevoegd:

```
var tabel = [1,2,3];
var nTabel = tabel.concat(4,5,[6,7]); // geeft[1, 2, 3, 4, 5, 6, 7]
```

### 7.2.8.7 join()

De `join()` method zet alle waarden van het array om in strings en concateneert die in één string met de waarden gescheiden door komma's. Je kan een optioneel argument specificeren met een alternatieve separator:

```
var tabel = [1, false, undefined, ,'javascript'];
var s = tabel.join("; "); // geeft de string "1; false; ; ;javascript"
```

#### 7.2.8.8 slice()

De `slice()` method returnt een nieuw array als een **stuk** van het oorspronkelijke array met als argumenten de begin- en een eindpositie.
Let er wel op dat het element met index eindpositie NIET inbegrepen is:

```
var tabel = ['jan', 'piet', 'pol','ilse', 'kurt', 'lut', 'carole'];
var stuk = tabel.slice(0,4) // bevat ['jan', 'piet', 'pol','ilse']
```

#### 7.2.8.9 toString() en toLocaleString()

Net als elk ander object heeft ook een Array een `toString()` method die een komma-gescheiden lijst van tekstwaarden produceert.
Is identiek aan `join()` gebruikt zonder argumenten.
De `toLocaleString()` method verschilt ervan doordat het de lokale instellingen van het besturingssysteem voor scheidingsteken zal gebruiken:

```
var tabel = [1, 'piet', undefined, true];
alert(tabel.toString()) // returnt "1, piet,, true"
```

#### iteration methods

Een aantal methods kunnen een **functie als argument** gebruiken die een callback uitvoert op elk element van het array.
Deze callback functie gebruikt 3 argumenten: element, index, array waarvan je gebruik kunt maken

#### 7.2.8.10 filter()

Deze *JS1.6* method maakt een nieuw array met alle elementen van het oorspronkelijke array die door de filter raken:

```
var lijst = [1,3,8,9,5,4,0,9,11,2];
function isGrootGenoeg(element,index,array){
  return (element >= 6);
}
var nieuweLijst = lijst.filter(isGrootGenoeg);
```

#### 7.2.8.11 forEach()

De callback functie - hier een anonieme functie - in de `forEach` wordt voor elk arrayelement uitgevoerd. Het array-item wordt als argument in de functie meegegeven.

```
tabel.forEach(function(el){
        alert(el);
});
```

Het kan in dit geval zelf eenvoudiger:

```
tabel.forEach(alert);
```

Ook hier zal elk element van het array getoond worden

#### 7.2.8.12 map()

Net als `filter()` maakt `map()` een nieuw array aan de hand van de elementen van een oorspronkelijk array via een callback function.

```
var getallen = [1,4,9];
var wortels = getallen.map(Math.sqrt);
```

Terwijl `filter()`gebruikt wordt om een selectie te maken, en dus mogelijk niet evenveel elementen bevat als het oorspronkelijk array, gebruikt `map()` het array als basis voor een verwerking tot een nieuw array.

#### 7.2.8.13 every()

deze method is vergelijkbaar met `filter()` maar returnt een `boolean` voor de test in de callback functie. `every()` returnt een `false` *zodra* de callback functie een `false` genereert, anders returnt `every()` een `true`.

```
var lijst = [6,9,4,11,12];

function isGrootGenoeg(element,index,array){
  return (element >= 6);
}

var geslaagd = lijst.every(isGrootGenoeg);
```

Vergeleken met het voorbeeld van `filter()`, bevat de `var` geslaagd nu geen nieuw array maar een `false` waarde , want minstens één van de waarden van het array was kleiner dan 6.

#### 7.2.8.14 some()

Deze method is het spiegelbeeld van `every()`: aan de hand van een callback functie test `some()` elk element en returnt een `true` *zodra* de callback functie een `true` genereert, anders returnt `some()` een `false`.

```
var lijst = [6,9,4,11,12];

function isGrootGenoeg(element,index,array){
  return (element >= 6);
}

var geslaagd = lijst.some(isGrootGenoeg);
```

In tegenstelling tot het voorbeeld van `every()`, bevat geslaagd nu een `true` want minstens één waarde is voldoende groot.

## 7.3 Het Date object

Heb je nodig om met datums en tijden te kunnen werken.

### 7.3.1 constructor

Om in JS een datum (of tijd) te gebruiken moet je een `Date` object aanmaken. Dat doe je met een **constructor**.

Een `Date` object is altijd uitgedrukt in **milliseconden** (*ms*).

Die constructor kan op verschillende manieren aangemaakt worden afhankelijk van welk soort argument(en) hij meekrijgt:

- **zonder argument**

  geeft een datumtijd object met de **huidige datum en tijd** (volgens de systeemklok)

```
var nu = new Date(); // nu op vandaag, huidig moment
```

- met argument ***DateString***

  Een `string` waarde die een datum-tijd voorstelt. De datum-tijd is *lokaal*, niet UTC.
  De *DateString* waarde moet in een vorm staan die herkenbaar is voor de `parse` method:

```
var verjaardag = new Date("Dec 14, 1954");
var verjaardag = new Date("Dec 14, 1954 05:34:00");
```

- met argumenten ***jaargetal, maandgetal, daggetal[, uurgetal, minuutgetal, secondengetal, millisecondengetal]***

  Niet te verwarren met een *String* waarde: het zijn `integer` waarden die de onderdelen van de *lokale* datum/tijd aanduiden.

  Het *tijdsgedeelte* is optioneel bij een datum, maar kan helemaal alleen als argument ingegeven worden:

  - Het *jaargetal* is een viercijfer getal.
  - Het *maandgetal* is een getal van 0 tot 11, waarbij 0 januari voorstelt.
  - Het *daggetal* is een cijfer van 1 tot 31.
  - Het *uurgetal* is een cijfer van 0 tot 23.
  - Het *minuutgetal* is een cijfer van 0 tot 59.
    Het *secondengetal* is een cijfer van 0 tot 59.
  - Het *millisecondengetal* is een cijfer van 0 tot 999.

```
var verjaardag = new Date(1954,11,14); //identiek als hierboven
```

```
var verjaardag = new Date(1954,11,14,5,34,0,0); //identiek als hierboven
```

- argument ***milliseconden***

  Omdat de basiseenheid van een `Date` object *ms* is, kan je een tijdstip ook ingeven als een hoeveelheid *ms*: dat zijn dan het aantal *ms* die verstreken zijn sinds middernacht 1/1/1970.

  Eén dag is gelijk aan 86400000 ms.

```
var dayafter = new Date(86400000); // stelt middernacht 2 januari 1970 voor
```

### 7.3.2 properties

Het `Date` object heeft geen eigen (local) eigenschappen (uiteraard wel `prototype` en constructor geërfd van `Object`).

Alle informatie stel je in en kom je te weten via methods.

### 7.3.3 static methods

Het `Date` object heeft enkele **static methods**, dat zijn methods die je vanuit de class zelf - `Date` - oproept, niet vanuit een afgeleide variabele, een instance:

| method | beschrijving |
|---|---|
| `now()` | geeft het aantal milliseconden sinds 1/1/1970 middernacht |
| `parse()` | interpreteert een `string` die een datum voorstelt en returnt het aantal ms. Deze method wordt impliciet gebruikt bij de constructor met een datum string |
| `UTC()` | geeft het aantal ms in de langste vorm van de constructor (in integer waarden) voor een UTC tijd. Kan je gebruiken om een UTC tijdstip aan te maken |

### 7.3.4 methods

Bij de methods van een `Date` object kan je twee grote groepen onderscheiden: de 'set'-ters en de 'get'-ters: zo worden datum-tijd ingesteld of afgelezen.

Dikwijls - maar niet steeds - is er een *lokale tijd* variant en een *UTC* variant.
UTC (Coordinated Universal Time) komt overeen met het vroegere GMT, Greenwich Mean Time.
Daarnaast zijn er ook enkele methods die converteren naar een string.

| method | beschrijving |
|---|---|
| `get[UTC]Date()` | geeft de dag van de maand (getal), lokaal of in UTC |
| `get[UTC]Day()` | geeft het weekdaggetal (zondag=0), lokaal of in UTC |

| method | beschrijving |
|---|---|
| `get[UTC]Month()` | geeft numerische waarde van de maand in een jaar (januari=0), lokaal of in UTC |
| `get[UTC]FullYear()` | geeft het jaartal in 4-digit vorm, lokaal of in UTC |
| `get[UTC]Hours()` | geeft het uur in 24-uur schema, lokaal of in UTC |
| `get[UTC]Minutes()` | geeft de minuten van het uur, lokaal of in UTC |
| `get[UTC]Seconds()` | geeft de seconden van de minuut, lokaal of in UTC |
| `get[UTC]Milliseconds()` | geeft de milliseconden van de seconde, lokaal of in UTC |
| `getTime()` | geeft de *ms* waarde van de *lokale* datum-tijd |
| `getTimezoneOffset()` | geeft het verschil (*offset*) in *minuten* tussen de lokale datum-tijd en de UTC variant. Afhankelijk van zomertijd. |
| `set[UTC]Date(daggetal)` | stelt de dag van de maand in (1-31), volgens lokale of UTC datum-tijd |
| `set[UTC]Month(maandgetal [,daggetal])` | stelt de maand van het jaar in (0-11), volgens lokale of UTC datum-tijd. Optioneel ook het daggetal(JS1.3) |
| `set[UTC]FullYear(jaargetal [,maandgetal,[daggetal]])` | stelt het jaartal in, volgens lokale of UTC datum-tijd. Optioneel ook het maand- en het daggetal (JS1.3) |
| `set[UTC]Hours(uurgetal [,minuutgetal[,secondengetal[, millisecondengetal]]])` | stelt het uur in (0-23), volgens lokale of UTC datum-tijd. Optioneel ook het minutengetal, het secondengetal en het millisecondegetal |
| `set[UTC]Minutes(minuutgetal [,secondengetal[, millisecondengetal]])` | stelt de minuten in (0-59), volgens lokale of UTC datum-tijd. Optioneel ook het secondengetal en het millisecondengetal |
| `set[UTC]Seconds(secondengetal[, millisecondengetal])` | stelt de seconden in, (0-59), volgens lokale of UTC datum-tijd. |

| method | beschrijving |
|---|---|
| | Optioneel ook het millisecondengetal |
| `set[UTC]Milliseconds(millisecondegetal)` | stelt de *ms* in (0-999), volgens lokale of UTC datum-tijd |
| `setTime(milliseconden)` | stelt de tijdswaarde in met een waarde in *ms*. Deze waarde kan uiteraard ook bekomen worden door een andere methode |
| `toLocaleString()` | Zet een datum object om in een string in het lokaal datum-tijd formaat |
| `toLocaleDateString()` | Geeft het datum gedeelte van een datum object als string in het lokaal formaat |
| `toLocaleTimeString()` | Geeft het tijdsgedeelte van een datum object als string in het lokaal formaat |
| `toUTCString()` | Zet een datum object om in een string in het UTC datum-tijd formaat |
| `toString()` | Zet een datum object om in een string in het UTC datum-tijd formaat. Bevat ook de offset info van de tijdszone |
| `toSource()` | Geeft de sourcecode van een datum object als string |
| `valueOf()` | Geeft de primitieve waarde van een datum object in milliseconden. identiek aan `getTime()` |

## *7.4 Het String object*

IN JS zijn strings een primitief datatype. Toch is er ook een `String` object dat een aantal methods ter beschikking stelt om teksten te manipuleren.

### 7.4.1 gebruik

Een `string` variabele kan aangemaakt worden als **primitief datatype** of als **object**.

- Als we het `new` sleutelwoord gebruiken maken we een nieuw `String` **object** aan
- zonder `new` sleutelwoord passen we eigenlijk de **functie** `String()` toe: die converteert de inhoud naar een primitieve `string`

Veel belang heeft het verschil niet want Javascript converteert primitieve strings automatisch naar objecten, dus kan je elke eigenschap of methode ook op een primitieve string toepassen:

```
var tekst = "tekst";    //primitieve string
var oTekst = new String("tekst"); // String object
var sTekst = String("tekst"); //primitieve string
tekst.length; // geeft 5
oTekst.length; // geeft 5
sTekst.length; // geeft 5
```

### 7.4.2 properties

Er is maar één specifieke eigenschap van `String`:

| property | returnt |
|---|---|
| `length` | het aantal karakters in de string |

### 7.4.3 algemene methods

Een aantal algemene methods. Let op de camelCase schrijfwijze.

| method | returnt |
|---|---|
| `charAt(n)` | het karakter op de indexpositie n. De index van het eerste karakter is 0. De index van het laatste karakter is length-1. |
| `charCodeAt(n)` | De Unicode waarde van het karakter op de indexpositie n. De index van het eerste karakter is 0. De index van het laatste karakter is length-1. |
| `concat()` | combineert meerdere strings in één nieuwe string |
| `indexOf(zoekString, [startIndex])` | geeft de index van het eerste karakter van het eerste voorkomen van een zoekString in een andere string.<br>Case-sensitive!<br>Indien de zoekstring niet gevonden wordt, returnt -1.<br>Een optionele startIndex positie laat je toe te zoeken vanaf een bepaalde positie, dus niet vanaf het begin. |
| `lastIndexOf(zoekString [,startIndex])` | geeft de index van het eerste karakter van het laatste voorkomen van een zoekString in een andere string.<br>Case-sensitive!<br>Indien de zoekstring niet gevonden wordt, returnt -1.<br>Ook hier kan je een optionele startIndex positie gebruiken. |

| method | returnt |
| --- | --- |
| `match(regexp)` | Zoekt de opgegeven regular expression en returnt een array: indien de regexp de g flag niet bevat, dan wordt een array met enkel de eerste overeenkomst gereturnt,<br>indien de g flag wel gezet is, returnt de method een array van alle gevonden overeenkomsten.<br>Als een gewone string meegegeven wordt, wordt deze in een regular expression omgezet.<br>Indien niets gevonden wordt, returnt de method null.<br>Voor meer details zie Werken met regular expressions |
| `replace(regexp, newString)` | Zoekt de opgegeven regular expression en vervangt die door newString.<br>Indien niets gevonden wordt, returnt de method null.<br>Voor meer details zie Werken met regular expressions |
| `search(regexp)` | Zoekt de opgegeven regular expression en returnt de positie van de opgegeven regexp.<br>Als een gewone string meegegeven wordt, wordt deze in een regular expression omgezet.<br>Indien niets gevonden wordt, returnt de method -1.<br>search() negeert global search met de g flag.<br>Zie ook Werken met regular expressions |
| `slice(startIndex [,stopIndex])` | returnt een nieuwe string met het stuk beginnend vanaf de startIndex positie en tot - maar niet inclusief - een eventuele stopIndex positie.<br>Als er geen stopIndex gespecificeerd is, wordt tot het einde gerekend.<br>Als de stopIndex negatief is, wordt die geplaatst terugtellend vanaf het einde.<br><br>slice()verschilt van substr() door het gebruik van twee indexposities, terwijl substr() een beginpositie en een length gebruikt.<br>slice()is identiek aan substring().<br>Niet verwarren met de Array.slice() method. |
| `split([separator][,limit])` | splitst een string in een array van strings, daarbij gebruik makend van het separator karakter.<br>Indien dit karakter niet opgegeven werd, returnt de method de string op zijn geheel.<br>Indien een limit opgegeven wordt, wordt het array beperkt tot zoveel elementen.<br>split() is erg nuttig voor de verwerking van door-tekens-gescheiden tekstbestandjes, zoals |

| method | returnt |
|---|---|
| | cookies |
| `substr(startIndex [,length])` | returnt een gedeelte van de oorspronkelijke string, beginnend vanaf startIndex positie, met een lengte van length karakters.<br>Indien length niet opgegeven is, dan tot het einde.<br>De oorspronkelijke string blijft ongewijzigd. |
| `substring(startIndex [,stopIndex])` | returnt een gedeelte van de oorspronkelijke string, beginnend vanaf startIndex positie, tot - maar niet inclusief - de stopIndex positie.<br>Indien stopIndexniet opgegeven is, dan tot het einde.<br>De oorspronkelijke string blijft ongewijzigd.<br>substring()is identiek aan slice(). |
| `toLowerCase()` | geeft de string in kleine letters |
| `toSource()` | |
| `toUpperCase()` | geeft de string in Hoofdletters |
| `toString()` | returnt de primitieve string waarde |
| `valueOf()` | returnt de primitieve string waarde |

### 7.4.4 static method

Het String object heeft één static method (methods die je vanuit de class zelf oproept, niet vanuit een instance).

| method | beschrijving |
|---|---|
| `fromCharCode()` | maakt een nieuwe string aan gebruik makend van de karaktercode(s) die je als argument meegeeft |

### 7.4.5 HTML wrapper methods

Er bestaan een aantal string-HTML methods die een string in de relevante HTML-tags teruggeven. **Het gebruik hiervan raden we volstrekt af**, gebruik DOM in de plaats.

## *7.5 Het Math object*

Het `Math` object is een ingebouwd object, een Javascript base class. Het wordt gebruikt voor zijn specifieke wiskundige constanten en methods.

Het overzicht hier is onvolledig: we lichten er enkel deze methods uit die we voor deze cursus nuttig achten. Wil je toch weten hoe je een *boogtangens* kunt krijgen, kijk dan in de *Javascript Core reference*.

De voorbeeldpagina op de *ondersteunende website* demonstreert de belangrijkste constanten en methods.

### 7.5.1 Gebruik

Het is onmogelijk een constructor toe te passen om een 'nieuw ' `Math` object aan te maken. Alle constanten of methods worden op het `Math` object zelf gebruikt:

```
var opp = Math.PI * straal * straal;
opp = Math.floor(opp);


var Wiskunde = new Math(); // onmogelijk
```

### 7.5.2 properties

Een property van `Math` geeft een **constante** waarde. Let er op dat je ze hoofdlettergevoelig zijn (zoals alles in JS) :

```
var opp = Math.PI * straal * straal;
```

| property | beschrijving |
|---|---|
| `E` | wiskundige constante *e* |
| `PI` | getal *π*, verhouding van de cirkelomtrek tot de diameter |

### 7.5.3 static methods

Er zijn enkel static methods van `Math` : je kan ze enkel toepassen op de base class `Math` zelf. Alle methods worden steeds in kleine letters geschreven.

| method | beschrijving |
|---|---|
| `Math.abs()` | De absolute waarde van een getal |
| `Math.ceil()` | Rond een getal af naar boven tot de eerste integer |
| `Math.floor()` | Rond een getal af naar beneden tot de eerste integer |
| `Math.round()` | Rond een getal af ofwel naar boven ofwel naar beneden tot de eerste integer |
| `Math.max()` | Vind de hoogste waarde in een lijst van argumenten. Indien er een niet-numerieke waarde tussen zit, returnt `NaN`. |
| `Math.min()` | Vind de laagste waarde in een lijst van argumenten. Indien er een niet-numerieke waarde tussen zit, returnt `NaN` |
| `Math.pow(x,y)` | Berekent x tot de macht y |

| method | beschrijving |
|---|---|
| `Math.sqrt(x)` | Berekent de vierkantswortel van x. Indien een negatief getal gebruikt wordt, returnt `NaN` |
| `Math.random()` | Genereert een willekeurig getal tussen 0 en 1.<br>Om een willekeurig getal tussen twee limieten te krijgen, gebruik<br>`(Math.random() * bereik_getal) + start_getal` |
| `Math.sin()` | Berekent de sinus van een hoek gegeven in radialen |
| `Math.cos()` | Berekent de cosinus van een hoek gegeven in radialen |

Enkele voorbeeldjes:

```
var getal = Math.abs(-Math.PI);       //returnt de positieve waarde van PI
var vier = Math.ceil(Math.PI); //returnt 4
var drie = Math.floor(Math.PI);       //returnt 3
Math.round(99.5);       //returnt 100
Math.round(-99.5);      //returnt -99
Math.max(-1.99,0,2,8,7);       //returnt 8
Math.max('yes',0,2,8,7);       //returnt NaN
Math.min(-1.99,0,2,8,7);       //returnt -1.99
Math.pow(-2,3); // returnt -8
Math.pow(2,-3); // returnt 0.125
var wortel = Math.sqrt(4);     //returnt 2
var willekeurig = Math.random();      //decimaal ts 0 en 1
Math.round(Math.random() * 5) + 10    // integer tussen 10 en 15
var rechteHoek = Math.sin(Math.PI/2); //returnt 1
var rechteHoek = Math.cos(Math.PI/2);
//returnt 6.12...e-17, een benadering van 0;
```

## *7.6 Het Regular Expression object*

Het `RegExp` object is een object dat een **patroon** van een *regular expression* bevat. Het heeft eigenschappen en methodes om tekstgedeelten **waarop het patroon past** te vinden in een tekst.

### 7.6.1 aanmaken

Een regular expression kan in een **letterlijk formaat** gebruikt worden, of via een **constructor**.

De syntax die je moet gebruiken voor het letterlijk patroon is:

```
/pattern/flags
```

Het letterlijk patroon zit dus vervat tussen **twee slashes**: `/pattern/`

Een aantal `String` functies gebruiken dit letterlijk formaat.

Met een constructor doen we het zo:

```
var zoek = new RegExp("pattern"[,"flags"])
```

De argumenten van deze constructor zijn:

- `pattern:` het tekstpatroon waarmee gezocht, vervangen wordt
- `flags`: bepalen het gedrag van de regexp, ze zijn **optioneel**.
  Mogelijke flags zijn:

| | |
|---|---|
| `g` | global match |
| `i` | ignore case |
| `m` | match over multiple lines |

Meer uitleg vind je hieronder.

### 7.6.2 Speciale karakters in een regex

Een regex maakt intensief gebruik van speciale karakters om patronen te maken. Dergelijke karakters worden aangeduid als een **ge-escaped karakter**. Escapen betekent een **backslash** ervoor zetten: het karakter dat onmiddellijk volgt wordt letterlijk genomen en nooit geïnterpreteerd.

```
var zoek = /\i/;     zoekt het karakter i
var zoek = /\//;     zoekt een slash
```

In regular expressions zijn een aantal karakters of sequenties van karakters die een **speciale betekenis** hebben voor het patroon. **Hoofdlettergevoelig!**

In de voorbeelden wordt het tekstgedeelte dat *past* aangeduid in een kleur en met streepjes-onderlijning

| karakter | werking |
|---|---|
| `\` | neemt volgende karakter letterlijk, met uitzondering van de speciale sequences hieronder |
| `^` | past het begin van de input.<br>`/^E/` past niet op "een Ezel", maar wel op de eerste E van "Een Ezel". Bij een multiline flag past ook op het begin van een nieuwe lijn |
| `$` | past het einde van de input.<br>`/$s/` past enkel op de laatste s van "lies is thuis". Bij een multiline flag past ook op het einde van elke lijn |
| `*` | het voorgaande karakter kan geen of meerdere keren voorkomen.<br>`/wa*/` past in "www.waarwaswilly.net" op "waa", "wa" en alle andere w' s in de tekst |
| `+` | het voorgaande karakter kan één of meerdere keren voorkomen.<br>`/wa+/` past in "www.waarwaswilly.net" enkel op "waa" en "wa" |
| `?` | het voorgaande karakter kan geen of één keer voorkomen. |

| karakter | werking |
|---|---|
| | `/n\s?d/` past in "honden in de kennel " op de opeenvolging van n en d, zowel met of zonder spatie ertussen |
| `.` | Het decimale punt past elk enkel karakter met uitzondering van het newline karakter<br>`/w.s/` past enkel "was" in "www.waarwaswilly.net"<br>Het kan ook geplaatst worden na *, +, ? of {} en zorgt er dan voor dat deze slechts het minimum aantal matches returnen i.p.v. het maximum |
| `x\|y` | past zowel x als y<br>`/hond\|kat/` past allebei in "een hond met kattekwaad" |
| `{n}` | past exact n voorkomens van het voorgaande karakter. n is een integer.<br>`/w{3}/` past enkel "www" in "www.waarwaswilly.net" |
| `{n,m}` | past n (minimum) tot m (maximum) voorkomens van het voorgaande karakter.<br>`/a{2,3}/` past enkel "aa" in "www.waarwaswilly.net" |
| `[xyz]` | een karakterset. Elk van deze karakters past. Je kan ook een bereik definieren door een koppelteken te gebruiken<br>`/k[aeou]n/i` past meerdere malen in "'kennismaking met Konan de barbaar' kunnen schrijven" |
| `[^xyz]` | een complementaire karakterset. Alle karakters passen, uitgezonderd deze.<br>/k[^eu]n/i past "'kennismaking met Konan de barbaar' kunnen schrijven" |
| `(x\|y)` | Ronde haakjes (parentheses) hebben verschillende betekenissen in regx. Eén fucntie is het groeperen van expressies in een enkele subexpressie. Zo past /(22\|33)+\|05/ de reeks 05 of één of meerdere voorkomes van ofwel de reeks 22 of 33<br>Maar ronde haakjes onthouden ook de subexpressie in de algemene expressie zodat je die verder kun gebruiken. Zie verder bij backreferences<br>`/hoofdstuk (\d+)\.\d*/` past en onthoudt de eerste 4 in "Hoofdstuk 4.4 " |
| `(?=x)` | Een lookahead: enkel een match indien het subexpressie patroon onmiddellijk volgt. Maakt geen deel uit van de match<br>`/Firefox(?=2)/` past enkel "Firefox2.0.0.11" |
| `(?!x)` | Een negatieve lookahead: enkel een match indien het subexpressie patroon NIET onmiddellijk volgt<br>`/Firefox(?!2)/` past enkel "Firefox1" en niet "Firefox2.0.0.11" |
| `\b` | past een woordrand, zoals een spatie of een nieuwelijn.<br>`/\bn\w/` past enkel "neen aan nieuwe nonnennamen" |
| `\B` | past het patroon NIET aan een woordrand.<br>`/\Bn\w/` past enkel "neen aan nieuwe nonnennamen" |
| `[\b]` | past een backspace karakter |
| `\d` | past een getal(digit). Identiek aan `[0-9]`.<br>`/\d+\.\d+\.\d+\.\d+/` past "FireFox2.0.0.11 is de laatste release " |
| `\D` | past een NIET-getal. Identiek aan `[^0-9]`. |
| `\w` | past elk alfanumeriek karakter, inclusief de underscore.<br>Identiek aan `[a-zA-Z0-9_]` |
| `\W` | past elk NIET-alfanumeriek karakter. Identiek aan `[^a-zA-Z0-9_]` |

| karakter | werking |
|---|---|
| \f | past een formfeed. |
| \n | past een linefeed. |
| \r | past een nieuweregel karakter |
| \s | past een enkel witruimte karakter, zoals een spatie, een tab, form feed of line feed |
| \S | past een enkel NIET-witruimte karakter |
| \t | past een tab |

### 7.6.3 Javascript methods die regex gebruiken

Regular expressions kunnen gebruikt worden in een aantal `Regex` en `String` methods. Ze verschillen in mogelijkheden en in de waarde die ze returnen.
We overlopen ze van eenvoudig naar meer complex:

#### 7.6.3.1 RegExp.test()

`RegExp.test()` is de eenvoudigste method. Returnt `true/false` bij een match:

```
var re = /^.{8,16}$/gi;
var ww = document.frmEen.wachtwoord.value;
if (!re.test(ww)) {alert('ww tussen de 8 en 16 karakters')}
```

Als je de match zelf ook nodig hebt gebruik dan de volgende method

#### 7.6.3.2 String.match()

`String.match()` verschilt van de vorige method in dat hier de regex getest wordt op een `String` object en dat de match gereturnt wordt of `null`:

```
var re = /^.{8,16}$/gi;
var ww = document.frmEen.wachtwoord.value;
var pastHet = ww.match(re);
if (pastHet==null)) {alert('ww is niet tussen de 8 en 16 karakters')}
else {alert(pastHet + ' is OK')}
```

#### 7.6.3.3 String.search()

`String.search()` is een method die test voor een match in de `String`. Returnt de **index** van de match binnen de string of `-1` indien niet gevonden:

```
var re = /javascript/gi;
  var pos = vbTekst2.search(re);
  if (pos!=-1) {
         alert(re + ' gevonden op positie ' + pos + ' in vbtekst2');
         }
  else {
         alert(re + ' niet gevonden');
```

```
}
```

### 7.6.3.4 String.split()

`String.split()` Is een erg nuttige method die een string in stukjes hakt volgens een **scheidingsteken**. Elk stukje wordt een item in een **resultaatArray**. Het scheidingteken wordt niet bewaard.
De syntax van deze method is:

```
var resultaatArray = String.split([separator][, limit])
```

Het optionele argument separator is dus een karakter waarop gesplitst wordt, bijvoorbeeld een punt-komma. separator kan echter ook een RegeExp zijn zodat een meer complexe splitsing mogelijk is. Indien er geen separator is, wordt de oorspronkelijke string in het eerste element van het array gestopt.
Het optionele argument limit stelt een **bovengrens** aan het aantal items dat in het resukltaatArray gestopt wordt.
Deze method wordt courant gebruikt om delimited strings op te splitsen, bijvoorbeeld in **cookies**, of in **querystrings** of uitgelezen tekstbestanden.

Een voorbeeld zonder regex:

```
var queryString = "voornaam=Jean&familienaam=Smits&adres=schoolplein
  8&post=8600&gemeente=Diksmuide";
var arrVelden = queryString.split('&'); //array
for (i=0;i<arrVelden.length;i++){
  strResultaat+=arrVelden[i]+"\n";
}
alert(strResultaat);
```

Een voorbeeld met een regex die html code splitst op eender welke html-tag:

```
var re=/<\/?[a-z0-9]* ?\/?>/i;
var arrVelden = vbTekst2.split(re);
for (i=0;i<arrVelden.length;i++){
  strResultaat+=arrVelden[i]+"\n";
}
alert(strResultaat)
```

### 7.6.3.5 String.replace()

De `String.replace()` method geeft een nieuwe string (wijzigt de oorspronkelijke string niet) waarin een vervanging gebeurt is. Dat kan via een gewone tekstvariabele:

```
var origTekst = "Internet Explorer is beter dan FireFox, zei de internet EXPLORER
  fan";
var vervangTekst = "Opera";
var nieuweTekst = origTekst.replace("internet explorer", vervangTekst, "gi");
```

In dit voorbeeld is de inhoud van nieuweTekst "Opera is beter dan FireFox, zei de Opera fan". De flag `i` werkt zorgt voor het **negeren van de case** van de match, de flag `g` doet een **global search** zodat beide matches vervangen worden.

Maar door regex te gebruiken heb je veel meer mogelijkheden. Eerst exact hetzelfde voorbeeld als regex:

```
var origTekst = "Internet Explorer is beter dan FireFox, zei de internet EXPLORER
  fan";
var re = /internet explorer/gi;
var vervangTekst = "Opera";
var nieuweTekst = origTekst.replace(re, vervangTekst);
```

Merk op dat de flags hier in de regex zitten.

Door subexpressies te gebruiken kunnen we bijvoorbeeld een plaatsverwisseling doen:

```
var origTekst = "Internet Explorer is beter dan FireFox, zei de internet EXPLORER
  fan";
var re = /(internet explorer)(.*)(firefox)/gi;
var nieuweTekst = origTekst.replace(re, '$3 $2 $1');
```

geeft als output "Firefox is beter dan Internet Explorer, zei de internet EXPLORER fan".
'En plus' kan de `replace()` method ook een **geneste (lambda) functie** gebruiken als vervangargument, die de output dynamisch berekent. Deze functie kan ook gebruik maken van substrings in de regex. In dit voorbeeld worden alle woorden gekapitaliseerd.

```
var origTekst = "Internet Explorer is beter dan FireFox, zei de internet EXPLORER
  fan";
var re = /\b\w+\b/g;
var nieuweTekst = origTekst.replace(re, function(woord){
         return woord.substring(0,1).toUpperCase() + woord.substring(1);
         });
var re = /^(?:http:\/\/)(.*?)\/(.*)/i;
nadien = vbTekst4.replace(re, function ($0,$1,$2){
                return "het gedeelte "  + $2 + " volgt op het domein  "+ $1 + "";
         })
```

Hierboven worden het domein en het pad van een url uit elkaar gehaald en herschreven.

### 7.6.3.6 RegExp.exec()

De `RegExp.exec()` method past een patroon op een string. Het returnt een array of `null`. Geeft hetzelfde soort resultaat als `String.match().`

```
var re=/Javascript/gi;
var tekst="<p>Deze voorbeelden gebruiken <b>JAVASCRIPT</b>:<br />
  Als u deze tekst kunt lezen hebt u <b>javascript</b> gedesactiveerd in uw browser.
  <br /> Activeer <b>Javascript</b> om de <em>voorbeelden</em> te doen werken</p>";
```

```
while ((m= re.exec(tekst))!=null) {
  alert("gevonden pos " + m.index + ": " + m[0] + "\nvolgende pos: " + re.lastIndex +
  "\n\n");
}
```

In bovenstaand voorbeeld wordt het woord 'Javascript' 3 maal gevonden. Zolang de match geen `null` returnt zoekt `exec` het patroon en returnt een array

### 7.6.4 Subexpressies en backreferences

Door gebruik te maken van **ronde haakjes** (x), creëer je een **subexpressie** die **onthouden** wordt, zodat je die later opnieuw kan gebruiken.
Dit kan heel handig zijn als je naar iets op zoek bent dat voorkomt binnen een groter patroon. Het grote patroon interesseert je echter niet, wel de subexpressie.
Bijvoorbeeld in het regex patroon
`/hoofdstuk (\d+)\.\d*/`
wordt de eerste groep getallen vóór een decimaal punt onthouden. Dit patroon past dus bijvoorbeeld 'hoofdstuk 12.3', waar de 12 onthouden wordt. Het patroon past bijvoorbeeld niet op 'lees hoofdstuk 3 en 4'.
Een onthouden subexpressie kan gerefereerd worden als een array item, tellend van 1: `[1][2]...[n]`.
Noteer dat `[0]` de **volledige expressie** bevat. Afhankelijk van de `RegExp` of `String` method, kan je hier op één of andere manier van gebruik maken. Dit noemt men een backreference.

Om in dezelfde regex terug te referen naar de subexpressie gebruikt men een **backslash** gevolgd door een **index getal**.
Wordt de subexpressie gebruikt in een method dan wordt meestal een `$1, $1, $2` gebruikt.

In de regex
`/hoofdstuk (\d+)\.(\d*)/`
refereert \2 of $2 naar de tweede subexpressie, dus het getal na de decimale punt.
Omdat subexpressies genest kunnen worden, telt de positie van het linker-haakje als index:
`/hoofdstuk ((\d+)\.(\d*))/`
`\1` of `$1` zou hier refereren naar de subexpressie die beide getallen vóór en na de decimale punt onthoudt.
Zo ook in onderstaand patroon verwijst de `\1` terug naar de eerste haakjes: in zijn geheel wordt hier de open en sluittag van een xml/html tag gevangen.
`<(.*)>(.*?)<\/\1>`
Het gebeurt dat je de haakjes enkel wil gebruiken om te groeperen en niet om te onthouden: plaats dan `?:` aan het begin van de groep zoals in
`/(?:\d{2}|\d{3})/`
Hier past het patroon ofwel 2 of 3 getallen, maar het wordt niet onthouden.

## *7.7 Het Error object*

Het `Error` object is een globaal object.
Error objecten worden gebruikt met een `throw` statement en verwerkt in een `try...catch` block.

### 7.7.1 constructor

Een nieuw `Error` object maak je met een constructor die een foutbericht en een verwijzing naar een bestand kan bevatten. De syntax is:

```
new Error([message[,filename[,lineNumber]]])
```

Het argument message is een string met een foutboodschap. De argumenten filename en lineNumber verwijzen naar het bestand en het lijnnummer dat de fout veroorzaakte.
Als `Error()` opgeroepen wordt als een functie, gedraagt hij zich net als met een `new` sleutelwoord.
Een voorbeeld:

```
new Error();
new Error('Woeps! u trapt op mijn tenen');
```

Als message gespecificeerd is gebruikt JS het gespecificeerde bericht, anders toont hij een standaard foutbericht

### 7.7.2 specifieke fouttypes

JS werpt nooit een zuiver base `Error` object maar meestal één van de afgeleide objecten:

| type | beschrijving |
|---|---|
| `SyntaxError` | een syntaxfout binnen een `eval()` functie, een `Function()` of een `RegExp()` constructor.<br>Andere syntaxfouten kunnen niet door een `try...catch` gevangen worden |
| `EvalError` | illegale call naar een `eval()` functie |
| `RangeError` | als een getalwaarde het verwachte getallenbereik te buiten gaat. Bijvoorbeeld als een array-item opgeroepen wordt met een negatieve indexwaarde |
| `ReferenceError` | foutieve referentie naar een variabele die niet bestaat |
| `TypeError` | fout in verwacht variabele type. Meestal geworpen als de variabelewaarde null of undefined is of als een functie meer argumenten heeft dan verwacht |
| `URIError` | fout in de `encodeURI()` of `decodeURI()` functie |

Om een specifiek fouttype te werpen:

```
new SyntaxError();
new RangeError('getalwaarde moet tussen 0-10 zijn');
```

In je eigen code kan je zonder problemen een niet-gespecificeerd `Error` object werpen.

### 7.7.3 properties

Het `Error` object heeft - naast de geërfde eigenschappen van `Object` - de volgende eigenschappen:

| property | beschrijving |
|---|---|
| `name` | een string die het type fout aanduidt |
| `message` | het foutbericht |
| `description` | idem als message (enkel IE) |
| `number` | Error number (enkel IE) |
| `fileName` | pad van bestand waarin fout optrad (enkel Mozilla) |
| `stack` | Stack trace (enkel Mozilla) |

### 7.7.4 methods

Dit object erft natuurlijk de `toString()` method die je echter niet moet gebruiken om het foutbericht te tonen. Gebruik daarvoor de `name` en de `message` eigenschap.

## *7.8 Het Object object*

Het `Object` object is het fundament **waarop alle andere objecten in JS gebaseerd zijn**. Alle JS objecten **erven** de eigenschappen en de methods van `Object`. Een `Date` object heeft bijvoorbeeld ook een eigenschap `constructor`.

Waarom bespreken we het dan niet als eerste? omdat je in de praktijk veel eerder een `Date` of een `String` object nodig zult hebben dan dit algemene object. De concepten hier uitgelegd leiden ons verder naar *Object georiënteerd programmeren*.

### 7.8.1 aanmaken

Een nieuw algemeen `Object` maken doe je ofwel
met een constructor
als **literal**: letterlijke, directe notatie

Met een **constructor** is de syntax :

```
var mijnObject = new Object([waarde])
```

Het **optionele** argument *waarde* kan eender welke waarde zijn: het wordt in een objectvariabele gestopt. Dit object zal van het type zijn dat correspondeert met de *waarde*. Zo zal `new Object("tekst")` een `String` object aanmaken.

Is er geen *waarde* , of is die `null` of `undefined`, dan wordt een lege algemeen objectvariabele aangemaakt.

Als **literal** gebruik je een notatie met **accolades** waarin je direct de eigenschappen en hun waarde instelt:

```
var mijnObject = {property1:waarde1,property2:waarde2, …}
```

### 7.8.2 properties

Het `Object` object heeft slechts twee eigenschappen:

| property | beschrijving |
|---|---|
| `constructor` | een **referentie** naar de Object functie die het object aanmaakte |
| `prototype` | prototype object |

### 7.8.3 constructor property

De `constructor` eigenschap geeft een referentie - geen string - naar de functie waarmee een object aangemaakt werd. Merk op dat de `typeof` operator je vertelt wat het *type* van een variabele is (bv. 'boolean' of 'object'), terwijl `constructor` een object is.

Veronderstel een variabele *tabel* aangemaakt als `array`:

```
var lijst = [1,2,3];
alert(lijst.constructor);
```

dan geeft dit een *referentie* terug naar de `Array` functie en krijg je een output zoals:

```
function Array(){
  [native code]
}
```

Een goede test voor een `Array` object is dan:

```
function isArray(iets){
if (typeof iets == "object"){
  if (iets.constructor == Array){return true;}
}
else return false;
}
```

### 7.8.4 prototype property

Elk object kan een `prototype` eigenschap - een *object* - hebben. Bij de aanmaak van het object is deze eigenschap ofwel leeg ofwel `null` (afhankelijk van de JS versie en de manier van aanmaak).

als je een objectvariabele aanmaakt, dan is het altijd afgeleid van een ander object, soms van `Object` zelf. Tijdens zijn constructie erft het nieuwe object alle eigenschappen en methods die in het ***prototype* van zijn parent** vindt. Een *prototype* van een object is dus het **sjabloon** voor elk nieuw object dat ervan afgeleid wordt.

In het hoofdstuk *Object georiënteerd programmeren* leggen we het principe van **prototype inheritance** in meer detail uit.

Niet enkel je eigen objectvariabelen hebben een *prototype*, maar ook de ingebouwde objecten zoals `Date`, `Array`, `String` en uiteraard ook `Object`. Straks zien we hoe je daarmee het gedrag van deze objecten kunt wijzigen.

### 7.8.5 methods

`Object` heeft een aantal (niet erg nuttige) methods:

| method | beschrijving |
|---|---|
| `hasOwnProperty("prop")` | `Boolean`. Heeft het object een **eigen** eigenschap?<br>dus niet overgeërfd via zijn *prototype*, maar wel gedefinieerd in zijn constructor of later toegevoegd |
| `isPrototypeOf(obj)` | `Boolean`. Is dit object **zelf** een *prototype object* van een ander object? |
| `propertyIsEnumerable("prop")` | `Boolean`. Bestaat de vermelde eigenschap van dit object en zo ja, kan die eigenschap gelijst worden in een `for` loop?<br>(met uitzondering van properties geërfd van het prototype)<br>Zo returnt deze method voor eigenschappen die een array of een object zijn een `false`. |
| `toString()` en `toLocaleString()` | geeft een tekstuele voorstelling van het object |
| `valueOf()` | returns de primitieve waarde van het object. Voor instances van `Object` zelf, returnt dit opnieuw het object |

Alle objecten in JS erven deze methods en properties, ze kunnen echter overschreven (overridden) worden.

# 8 Storage

Over http cookies en Web Storage

## *8.1 the stateless web*

Het web is stateless.

De meeste applicaties behouden *state* (*preserve state, stateful*): ze onthouden je instellingen, ze onthouden wat de waarden zijn tussen twee formulieren, denk bijvoorbeeld aan Office programma's.
Het internet is dat niet: een webserver behandelt elk verzoek om een pagina alsof het een unieke transactie is. De server heeft per definitie geen enkel idee wat je voordien gedaan hebt of waar je daarna naartoe gaat en vergeet onmiddellijk wat en naar wie hij iets verstuurde.

Toch is het nuttig voor de websurfer én voor de achterliggende webapplicatie om op één of andere manier gegevens te bewaren of over te dragen naar een volgende pagina of naar het volgende bezoek.

Daar zijn een aantal oplossingen voor:

- via GET en POST formuliervariabelen
- via SESSION variabelen van de server
- met Web Storage
- via HTTP cookies

**Web storage** en **Cookies** zijn in principe twee methodes waarbij eenvoudige *naam=waarde* paren in tekstvorm gegevens opslaan op de *client* (PC van de gebruiker).
Bij een volgende bezoek aan de website kan de server kijken of er een data opgeslagen zijn en zo ja, die lezen en interpreteren, bijvoorbeeld, je taalinstellingen.

Data opslaan in **cookies** op de *client* is de klassieke manier van werken. Cookies hebben echter enkele nadelen en daarom is een nieuwe manier ontwikkeld die toegepast wordt in HTML5: **Web Storage**.
Hieronder worden beide methodes uitgelegd.

## *8.2 Session en persistent cookies*

Op vlak van leeftijd komen cookies in twee vormen:

- **session cookies**: zolang de session duurt
- **persistent cookies**: worden opgeslagen op de HD van de gebruiker

Een session cookie is een cookie dat niet opgeslagen wordt, maar in het geheugen blijft tot de sessie beëindigd is. Een **sessie** duurt zolang de browser open is. Van pagina naar pagina surfen beëindigt de session niet, de browser sluiten wel. Een session cookie maak je door de `expires` datum achterwege te laten.

Een persistent cookie is een cookie dat opgeslagen wordt in de **cookie file** (van de browser). Dat doe je door er een `expires` datum aan toe te voegen. Die datum ligt in de **toekomst**, typisch berekend vanuit de huidige datum + een tijdspanne. De datum moet in UTC vorm zijn, dit doe je op een `Date` object met de functie `toUTCString()` (vervangt `toGMTString()`).
Door het cookie te overschrijven met een datum in het verleden, verwijder je een persistent cookie.

## *8.3 document.cookie*

Het browser object `document.cookie` bevat de **volledige set cookies** voor het ingeladen document. Het is via dit object dat je een cookie kan lezen, wijzigen of een nieuwe aanmaken.
Naast de naam en de waarde, heeft elk cookie de volgende **'parameters'**

### 8.3.1 attributen van `document.cookie`

Leeftijd, scope en beveiliging kunnen ingesteld worden met deze 'parameters'.
Merk op dat dit geen echte JS properties zijn die je kan opvragen als `document.cookie.expires`. Jammer, maar neen, je moet ze aflezen uit de cookie string, zie verder.

| parameter | beschrijving |
|---|---|
| | Een datum in UTC formaat. |
| `expires` | Indien niet ingesteld blijft het cookie tijdelijk.<br>Indien ingesteld, is dit de houdbaarsheidsdatum van het cookie. Na deze datum wordt het cookie door de applicatie (browser) automatisch gewist. |
| | Een relatief pad. |
| `path` | Indien niet ingesteld, blijft de toegang tot het cookie beperkt tot de huidige pagina en deze in dezelfde map of submappen ervan.<br>Indien ingesteld, hebben pagina's in het path ook toegang tot het cookie |
| | Een domein. |
| `domain` | Indien niet ingesteld, blijft de toegang tot het cookie beperkt tot de Hostname van de huidige webserver.<br>Indien wel ingesteld, hebben ook pagina's in het gespecificeerde domein toegang tot het cookie |
| | boolean. |
| `secure` | Indien false (of niet ingesteld) worden cookies onbeveiligd doorgestuurd over het web.<br>Indien true (of ingesteld als secure), wordt het enkel over HTTPS of een andere beveiligd protocol verstuurd. |

### 8.3.2 levensduur

Als een cookie gezet wordt zonder het attribuut `expires`, is het een session cookie en verdwijnt na de sessie:

```
document.cookie = "versie=2.0"
```

Als een cookie gezet wordt met een `expires` datum in het **verleden**, is het niet persistent en verdwijnt. Zo kan je een persistent cookie overschrijven om die te vernietigen.

```
var vandaag = new Date();
var vorigJaar= new Date(vandaag);
vorigJaar = vorigJaar.setFullYear(vandaag.getFullYear - 1);
document.cookie = "versie=2.0;expires=" + vorigJaar.toUTCString()
```

Om een cookie persistent te maken moet het een `expires` datum in de **toekomst** krijgen:

```
var vandaag = new Date();
var volgendJaar= new Date(vandaag);
volgendJaar = volgendJaar.setFullYear(vandaag.getFullYear + 1);
document.cookie = "versie=2.0;expires=" + volgendJaar.toUTCString()
```

dit cookie zal een jaar bewaard en gelezen worden.

### 8.3.3 path scope

In bijna alle gevallen wil je enkel dat het cookie leesbaar is voor de pagina die gebruikt en andere pagina's in dezelfde map of submappen ervan. In dat geval is het attribuut `path` overbodig.
Gebruik `path` enkel als je het cookie wil laten lezen door pagina's in andere mappen van je website.
Zo wordt het volgende cookie bijvoorbeeld beschikbaar vanaf de root van de site ondanks het feite dat het zelf in een submap zit:

```
document.cookie = "user=Jan;path=/"
```

### 8.3.4 domain scope

Op dezelfde manier zal `domain` enkel nodig zijn als je wil dat het cookie vanuit een ander domein ook kan gelezen worden, bijvoorbeeld bij subdomeinen:

```
document.cookie = "versie=2.0; domain=.archimedes.org"
```

### 8.3.5 secure

Als het attribuut `secure` gebruikt wordt, kan het cookie enkel over een secure protocol verstuurd worden:

```
document.cookie = "login=james;secure"
```

## 8.4 Cookies bekijken

Om zonder scripting de cookie informatie te lezen, kan je zelf op zoek gaan op je PC naar de **cookie file(s).** Dat zijn de tekstbestanden met de cookies.
De plaats waar die staan is afhankelijk van de applicatie (en kan via path ingesteld worden):

- *Internet Explorer* slaat cookies op als aparte bestandjes, op Win XP in `c:\documents and settings\%user%\cookies.`
  Een dergelijk bestandje heeft een bestandsnaam als %USER%@DOMAIN.TXT
- *FireFox* bewaart zijn cookies in 1 enkel bestand, COOKIES.TXT, dat te vinden is op `C:\Documents and Settings\%user%\Application Data\Mozilla\Firefox\Profiles\%ProfielName%`
- ander webapplicaties zoals *Flash* bewaren ook cookies, maar die zijn niet altijd in tekstvorm en makkelijk terug te vinden.
  Voor Flash heb je bijvoorbeeld de *Settings Manager*, een webpagina op de Adobe site die je je Flash cookies toont

## 8.5 Cookies schrijven en lezen

Het lezen van cookies vergt een voorkennis van de `String` functies `indexOf()` en `substring()`: trek die even na.

### 8.5.1 syntax

Om een cookie te schrijven gebruik je de syntax:

```
"cookienaam=cookiedata[;expires=time;path=path;domain=domainname;secure]"
```

Dat toont aan dat, naast de gegevens zelf, de vier attributen optioneel zijn en telkens gescheiden door een **punt-komma**.

### 8.5.2 aanmaken

Om in Javascript een cookie aan te maken schrijf je een `string` naar het `document.cookie` object:

```
document.cookie = "voornaam=jan;expires="+jaarLater.toUTCString()+";path=/leden";
```

Deze volledige string terug lezen kan niet: enkel het naam=waarde paar is leesbaar.

Dikwijls maakt men een **functie** om een cookie te plaatsen:

```
function setCookie(naam,waarde,dagen){
  var verval = "";
  if(dagen){
          var vandaag = new Date();
          var vervalDatum = new Date(vandaag.getTime()+dagen*24*60*60*1000);
          verval = vervalDatum.toUTCString();
  }
document.cookie = naam + "=" + waarde + ";expires=" + verval;
}
```

1. als het argument *dagen* ingevuld is, maken we een datum object op *vandaag+ dagen*
2. wordt het cookie ingesteld door te schrijven naar het cookie object

De optionele parameters `domain`, `path` en `secure` laten we hier achterwege.

### 8.5.3 lezen

Als je een cookie wil gebruiken, wil je dus één of meer welbepaalde waarden te weten komen: diegene die je webpagina er zelf in het verleden in geplaatst heeft.
We moeten dus op zoek naar het betreffende naam=waarde paar in het cookie bestand.
Dat betekent de volledige inhoud analyseren tot je vindt wat je nodig hebt.
Typisch gaat men daarvoor een functie schrijven die op zoek gaat naar de naam:

```
function getCookie(naam){

(1)      var zoek = naam + "=";
(2)      if (document.cookie.length>0){
(3)              var begin = document.cookie.indexOf(zoek);
(4)              if (begin!=-1){
                         begin += zoek.length;
(5)                      var einde = document.cookie.indexOf(";", begin);
(6)                      if (einde==-1){
                                 einde = document.cookie.length;
                         }
(7)                      return document.cookie.substring(begin, einde);
         }
  }
}
```

We gebruiken deze functie bijvoorbeeld om het cookie 'voornaam' te zoeken:

1. de variabele *'zoek'* is een `string` bestaande uit de naam + een gelijkheidsteken
2. als er een cookiefile is (voor deze pagina, domain en user)
3. bepaal de beginpositie van *'zoek'* in de cookiefile : er zitten meestal meerdere cookies in het bestand, het cookie dat je zoekt staat niet noodzakelijk vooraan
4. als beginpositie gevonden (`indexOf` heeft de waarde -1 als die niet gevonden is), verplaats dan die beginpositie opnieuw tot juist na het gelijkheidsteken (dus eerdere 'begin'-positie plus de lengte van de zoekstring)
5. de variabele *'einde'* is de positie van de eerste punt-comma na de beginpositie
6. als de puntcomma niet gevonden wordt (laatste of enige cookie), stel einde in op de lengte van de cookiefile-string
7. de `String` method `substring()` knipt dan de waarde van het gevraagde cookie eruit en returnt die

Een laatste handige functie is clearCookie() die een cookie verwijdert door hem te overschrijven met een datum in het verleden:

```
function clearCookie(naam){
/*
verwijdert een cookie
naam: cookienaam
*/
  setCookie(naam,"",-1);
}
```

Op het net vind je talloze varianten van de functies setCookie en getCookie, deze hier voldoen echter voor de meeste toepassingen.

## 8.6 DOM storage

***Web Storage*** is een algemene term voor nieuwe methodes waarmee men veel meer gegevens op de client PC kan opslaan. deze methodes omvatten bv. SQL opslag van gegevens, een standaard die momenteel verworpen wordt.

***DOM Storage*** is één van die technieken: het bevat `window.sessionStorage` en `window.localStorage` die ondersteund worden door de recente versies van alle browsers: je moet al teruggaan naar IE7 om geen ondersteunign te hebben. Simpelweg kunnen we stellen dat deze *DOM Storage* cookies zeer snel zullen verdringen.
Het wordt *DOM Storage* genoemd omdat de gegevens opgeslagen worden in de DOM tree.

Voordelen t.o.v. Cookies:

| Cookies | Web storage |
|---|---|
| **Traag**: worden elke keer meegestuurd met elk HTTP Request, nodig of niet. | worden helemaal niet meegestuurd. Enkel toegankelijk via client-side scripting |
| **Onveilig**: geen encryptie mogelijk | encryptie mogelijk |
| Kleine capaciteit: max 4Kb | zeer grote capaciteit |
| **Onveilig**: data leakage tussen vensters<br><br>geen event | geen data leakage<br><br>`storage` **event** vuurt bij wijziging |

### 8.6.1 Session Storage en Local Storage

Webstorage bevat twee objecten:

- `sessionStorage` **:** gegevens bewaard voor de duur van de sessie
- `localStorage` **:** gegevens bewaard voor langere duur

Beiden zijn properties van het `window` object, infeite moet je dus schrijven `window.localStorage`, maar je mag dat afkorten naar `localStorage`.

Een `sessionStorage` waarde is er *per-pagina-per-venster*. De waarde blijft dus **niet behouden** bij het afsluiten van een venster of van de browser. `sessionStorage` is bedoeld om verschillende instanties van dezelfde applicatie toe te laten parallel te lopen zonder zich met elkaar te bemoeien, iets dat met cookies problemen geeft. `sessionStorage` overleeft wel een crash van de browser!

Een `localStorage` waarde is op *per-domein* basis. De waarde blijft **behouden** bij het afsluiten van een venster of van de browser. Een localStorage waarde heeft **geen verlooptijd**.

Beide objecten hebben dezelfde **methods**:

| method | beschrijving |
|---|---|
| `setItem('key', 'value')` | stelt een 'key' in met de waarde 'value' |
| `getItem('key')` | leest de waarde van 'key' |
| `removeItem('key')` | verwijdert de key helemaal |
| `clear()` | wist **alle** waarden |

Beide objecten hebben één **property**:

| property | beschrijving |
|---|---|
| `length` | het aantal 'keys' in de storage area |

In plaats van de methods `setItem()` of `getItem()` te gebruiken mag je ook de waarde aanspreken als een **property** van de storage, dus

```
var saldo = localStorage['saldo'];
var saldo = localStorage.saldo;
var saldo = localStorage.getItem('saldo');
```

zijn equivalent en leveren alledrie de waarde van de *key* 'saldo' op.

Voor het ogenblik laat DOM Storage enkel `String`s toe als data type (alhoewel het de bedoeling was ook objecten te gebruiken). Via een kleine omweg kunnen we natuurlijk ook objecten opslaan: we zetten ze om naar een JSON string.

### 8.6.2 Capaciteit

De HTML5 standaard raadt de browsers aan 5MB ruimte te voorzien en de meeste volgen dit, enkel IE voorziet het dubbel.

Meestal is deze optie niet (Chrome,Safari,IE) of moeilijk (FF, Opera) configureerbaar.

### 8.6.3 Storage event

Bij elke **wijziging** van de storage wordt een `storage` event afgevuurd. Het is bealngrijk te weten dat er ook een echte wijziging moet gebeuren: als dezelfde

waarde teruggegplaatst wordt of er gebeurt een `clear()` terwijl er geen items zijn, zal het event niet afvuren.

Momenteel ondersteunen nog niet alle browsers alle properties van dit event.

# 9 Formulieren

Formulieren zijn de basis van elke interactieve website. Via hun formulier*controls* worden gegevens doorgegeven aan de webserver. Om die reden is *een perfecte kennis van deze html-elementen en hun attributen* noodzakelijk om er client-side mee te kunnen scripten.

We gebruiken de term **formuliercontrol** om die html-elementen aan te duiden die te maken hebben met *gegevensuitwisseling*, dus bijvoorbeeld een `submit`-knop, een *hidden field*, een *invulveld*, een *keuzerondje*, etc. Elementen zoals een `fieldset`, `legend` en `label` horen ook in een `form` thuis, maar hebben geen directe impact op de gegevensuitwisseling.

Is je kennis van deze elementen onvoldoende, herhaal ze dan eerst en hou een html-naslagwerk bij de hand.

## *9.1 Het form en de formulier elementen*

Hier vind je een naslag van de relevante DOM eigenschappen, methods en events voor de HTML elementen `form` en zijn control elementen. Meestal weerspiegelen de *DOM properties* de specifieke *HTML attributen*, maar er zijn er ook enkele die enkel via DOM te lezen/schrijven zijn en dus geen HTML attribuut hebben.

Voor de Events hebben we de algemeen toepasbare *Event Handlers*, zoals `onclick, onmouseout`, etc.. achterwege gelaten en enkel de control-specifieke beschreven.

### 9.1.1 form

De voornaamste attributes, methods en events (DOM2) van een `form` element (naast de gebruikelijke Core attributen):

| attribute/method | html attribuut | beschrijving |
|---|---|---|
| **attributes** | | |
| `action` | action | `string`. De *form handler*: een server-side script of pagina |
| `enctype` | enctype | `string`. Het *content-type* van de submitgegevens, standaardwaarde "application/x-www-form-urlencoded". Andere waarde is "multipart/form-data" enkele te gebruiken voor een input element type file |
| `elements` | | *collection*. De collectie van alle form controls in het form |
| `length` | | *long*. Het aantal form controls in het formulier |
| `method` | method | `string`. HTTP method: get \| post |

| | | |
|---|---|---|
| `name` | name | `string`, readonly . De gegeven naam. |
| **methods** | | |
| `reset()` | | zet alle standaardwaarden van de controls terug. Doet hetzelfde als de Reset knop. |
| `submit()` | | Submits het form. Doet hetzelfde als de Submit knop. |
| **events** | | |
| `submit` | onsubmit | de formgegevens worden doorgestuurd |
| `reset` | onreset | de formgegevens worden terug standaard ingesteld |

### 9.1.2 input

Het `type` attribuut van een `input` element bepaalt de vorm van dit control: invulveld, keuzerondje, vinkje, submit-knop, etc...
Door de verschillen zijn sommige properties, methods of events niet toepasselijk of verschillen ze in werking.

Even herhalen:

- `input type="text"` is een invulveldje
- `input type="password"` is een invulveld voor een wachtwoord
- `input type="checkbox"` is een selectievakje
- `input type="radio"` is een keuzerondje
- `input type="submit"` is een submitknop
- `input type="reset"` is een resetknop
- `input type="image"` is een grafische submitknop
- `input type="hidden"` is een onzichtbare control
- `input type="file"` is een control dat toelaat een bestand te selecteren

Als een item in de tabel aangeduid is met *, betekent dat dat er onderscheid gemaakt wordt tussen de types.

| attribute/method | html attribuut | beschrijving |
|---|---|---|
| **attribute** | | |
| `accessKey` | accessKey | `string`. Een enkel karakter om met het toetsenbord toegang te geven tot de control. |
| `alt` | alt | `string`. Alternate tekst (tooltip). |
| `checked(*)` | | `boolean`. Enkel voor type *`checkbox`* en *`radio`*. Is de control aangevinkt? |
| `defaultChecked(*)` | checked | `boolean`. Enkel voor type `checkbox` en `radio`. Stelt de aanwezigheid van het HTML attribuut `checked` voor. Wijzigt niet door verandering van `checked` |

| | | |
|---|---|---|
| `defaultValue(*)` | value | `boolean`. Enkel voor type `text`, `file` en `password`.<br>Reflecteert de inhoud van een eventueel HTML attribuut `value`. Wijzigt niet door verandering van `value` |
| `disabled` | disabled | `boolean`. Bepaalt of de control aanklikbaar is voor de gebruiker. |
| `form` | | `node`, readonly. Het parent `form` element zelf |
| `maxLength(*)` | maxlength | `long`. Enkel voor type `text` en `password`.<br>Het maximaal aantal toegelaten karakters. |
| `name` | name | `string`. De gegeven naam. |
| `readOnly(*)` | readonly | `boolean`. Enkel voor type `text` en `password`.<br>Maakt het control alleen-lezen. |
| `size(*)` | size | `cdata`. Verschillend voor elk type:<br>• `text`, `password` : breedte van de control in aantal karakters<br>• `checkbox`, `radio`, `submit`, `reset`, `file`, `image`: breedte van de control in pixels<br>• `hidden`: niet relevant |
| `tabIndex` | tabindex | `long`. De plaats van het element in de 'tabbing order'. |
| `type` | | `string`, readonly. Zie hierboven |
| `value(*)` | | `string`. Verschillend voor elk type:<br>• `text`, `password`, `file` : de huidg ingevulde gegevens<br>• `hidden`,`checkbox`, `radio`, `submit`, `reset`, `image`: de waarde van het HTML attribuut *value* |
| **methods** | | |
| `blur()` | | verwijdert de *focus* van het element |
| `focus()` | | plaatst de *focus* op het element |
| **events** | | |
| `focus` | onfocus | het element krijgt de *focus* |
| `blur` | onblur | het element verliest de *focus* |
| `change` | onchange | de `value` van het element is gewijzigd |

### 9.1.3 select

Het `select` element laat de gebruiker kiezen tussen een aantal `option` elementen.
De toepasselijke DOM2 properties, methods en events zijn:

| attribute/method | html attribuut | beschrijving |
|---|---|---|
| **attributes** | | |
| `disabled` | disabled | `boolean`. Bepaalt of de control aanklikbaar is voor de gebruiker. |
| `form` | | formelement, readonly. Het parent form element zelf |
| `length` | | long. Het aantal option elementen in de select. |
| `multiple` | multiple | `boolean`. Of meerdere option elementen kunnen geselecteerd worden. |
| `name` | name | `string`, readonly. De gegeven naam. |
| `options` | | collection, readonly. De collectie option elementen |
| `selectedIndex` | | long. Het indexgetal van de geselecteerde `option` beginnende vanaf 0.<br>Als er niets geselecteerd is, is de waarde -1.<br>Als meerdere waarden geselecteerd zijn, heeft dit attribuut de waarde van de eerste geselecteerde `option`. |
| `size` | | long. Het aantal zichtbare rijen |
| `tabIndex` | tabindex | long. De plaats van het element in de tabvolgorde. |
| `type` | | `string`, readonly. Geeft het *type* control aan: "select-multiple" voor een `multiple=true` of "select-one" als `multiple=false` |
| `value` | | `string`. De `value` van de huidig geselecteerde `option`. Als `multiple=true` dan de waarde van de eerst geselecteerd `option`. |
| **methods** | | |
| `add(element, before)` | | Voegt een `option` element toe aan de select.<br>De parameter *element* is een geldig `option` HTMLElement dat toegevoegd wordt.<br>De parameter *before* is ofwel `null`, dan wordt er achteraan toegevoegd, ofwel een geldig `option` HTMLElement uit de aanwezige lijst, waarvoor het nieuwe element toegevoegd wordt. |
| `blur()` | | verwijdert de focus van het element |

| | | |
|---|---|---|
| `focus()` | | plaatst de focus op het element |
| `remove(index)` | | verwijdert een `option` element met indexNummer *index* uit de `select`. |
| **events** | | |
| `focus` | onfocus | het element krijgt de focus |
| `blur` | onblur | het element verliest de focus |
| `change` | onchange | de value van het element is gewijzigd |

### 9.1.4 option

Het `option` element is één van de keuzes in een `select` element.
De toepasselijke DOM2 properties zijn:

| attribute/method | html attribuut | beschrijving |
|---|---|---|
| **attributes** | | |
| `defaultSelected` | | `boolean`. Een nieuw DOM2 attribuut dat reflecteert of deze `option` geselecteerd is per default |
| `disabled` | disabled | `boolean`. Bepaalt of de control aanklikbaar is voor de gebruiker. |
| `form` | | formelement, readonly. Het parent `form` element zelf |
| `index` | | long, readonly. de index van dit element in de lijst van zijn parent `select`. Start met 0. |
| `label` | label | `string`. Een vervangende tekst voor de inhoud van het `option` element |
| `selected` | selected | `boolean`. Reflecteert of het `option` element momenteel geselecteerd is. Wijzigt de waarde van het HTML attribuut niet. |
| `text` | | `string`, readonly. De tekst binnen het `option` element |
| `value` | value | `string`. De `value` van de `option`. |

### 9.1.5 textarea

Het `textarea` element is multi-line invulveld. Een verschil met een invulveld is dat de waarde tussen de begin- en eindtag van het element staat, niet in een `value` attribuut.
De toepasselijke DOM2 properties zijn:

| attribute/method | html attribuut | beschrijving |
|---|---|---|

| attribute | | |
|---|---|---|
| accessKey | accessKey | string. Een enkel karakter om met het toetsenbord toegang te geven tot de control. |
| cols | cols | long. Breedte in aantal karakters. |
| defaultValue | | string. De standaardwaarde van het element |
| disabled | disabled | boolean. Bepaalt of de control aanklikbaar is voor de gebruiker. |
| form | | formelement, readonly. Het parent form element zelf |
| name | name | string, readonly. De gegeven naam. |
| readOnly | readonly | boolean. Maakt het control *alleen-lezen*. |
| rows | rows | long. Aantal lijnen. |
| tabIndex | tabindex | long. De plaats van het element in de tabvolgorde. |
| type | | string, readonly. Het type control: "textarea" |
| value | | string. De huidig ingevulde waarde. Als je deze property wijzigt, verandert de inhoud van de doorgestuurde waarde, maar niet de zichtbare tekst in het element. |
| **method** | | |
| blur() | | verwijdert de focus van het element |
| focus() | | plaatst de focus op het element |
| select() | | selecteert de inhoud van de textarea |
| **event** | | |
| focus | onfocus | het element krijgt de focus |
| blur | onblur | het element verliest de focus |
| change | onchange | de value van het element is gewijzigd |
| select | onselect | een gedeelte tekst werd geselecteerd |

### 9.1.6 button

Het button element is een verbeterde versie van een input type=submit element. Eén verschil is dat het naast een value ook een inhoud kan hebben , zoals een image. Het type attribuut bepaalt zijn gedrag als je het aanklikt:

- geplaatst **buiten een form** , heeft type geen invloed: het wordt gebruikt om een script te starten
- geplaatst **binnen een form**, reageert een button afhankelijk van het type attribuut ongeacht of er een script aan gekoppeld is of niet:

- geen `type` attribuut: het `submit` event voor het `form` wordt getriggert!
- `type=submit`: het `submit` event voor het form wordt getriggert
- `type=reset`: het `reset` event voor het form wordt getriggert
- `type=button`: geen form events worden getriggert, enkel het script wordt uitgevoerd

De toepasselijke DOM2 properties zijn:

| **attribute/method** | **html attribuut** | **beschrijving** |
|---|---|---|
| **attribute** | | |
| `accessKey` | accessKey | `string`. Een enkel karakter om met het toetsenbord toegang te geven tot de control. |
| `disabled` | disabled | `boolean`. Bepaalt of de control aanklikbaar is voor de gebruiker. |
| `form` | | formelement, readonly. Het parent `form` element zelf |
| `name` | name | `string`, readonly. De gegeven naam. |
| `tabIndex` | tabindex | long. De plaats van het element in de tabvolgorde. |
| `type` | type | string, readonly: keuze uit button\|reset\|submit. submit is default |
| `value` | value | `string`. De waarde van het `value` attribuut. Als je deze property wijzigt, verandert de inhoud van de doorgestuurde waarde, maar niet de zichtbare tekst in het element. |

## *9.2 het formulier en zijn controls aanspreken*

Om gegevens door te sturen moeten controls (bv een `input`-element) als child-elementen **in** een `form` element zitten. Ze maken dan deel uit van het array van form-elementen. Het is het `form` dat de ingevulde gegevens zal doorsturen via zijn `action` en `method` attributen.
Een form control dat zich buiten een `form` bevindt kan enkel nut hebben voor een Javascript, bijvoorbeeld een `button`-element dat een script opstart.

Formulieren bestaan al sedert de eerste html-versie en daarom kan je zowel de klassieke **DOM Level0** als de moderne **DOM Level2** methodes gebruiken.

### 9.2.1 Een form refereren

Er kunnen meerdere formulieren in dezelfde pagina zitten, daarom wordt een `form` best benoemd/geïdentificeerd. Dat gebeurt traditioneel (**DOM0**) door middel van een `name` attribuut, maar kan ook met een `id` attribuut(**DOM2**).
Referenties via de **forms collection** kan ook.

Veronderstel:

```
<form name="frmEen"  id="frmEen" action="" method="">...</form>
<form name="frmTwee"  id="frmTwee" action="" method="">...</form>
<form name="frmDrie"  id="frmDrie" action="" method="">...</form>
```

Dan kan het tweede formulier in de pagina aangesproken worden op volgende manieren:

- `document.forms[1]`
- `document.frmTwee`
- `document.getElementById('frmTwee')`

We kunnen dus stellen dat zowel het `name` als het `id` attribuut van een `form` element er enkel zijn om het element makkelijk te kunnen identificeren, maar niet essentieel zijn.

### 9.2.2 Een control refereren

Uit het vorige volgt dat er minstens drie manieren zijn om een control te refereren: DOM0 en DOM2. Veronderstel:

```
<form name="frmTwee"  id="frmTwee" action="" method="">
  <input type="text" name="voornaam" id="voornaam" />
  <input type="text" name="familienaam" id="familienaam" />
</form>
```

Dan kunnen we het control 'voornaam' aanspreken met:

- `document.forms[1].elements[0]`
- `document.forms[1].elements['voornaam']`
- `document.forms[1].voornaam`
- `document.frmTwee.voornaam`
- `document.getElementById('voornaam')`

Al deze methodes zijn evenwaardig.

De notatie `document.frmTwee.voornaam` is de standaard JS **object** notatie waarbij `frmTwee` een **property** is van `document` en `voornaam` een **property** van `frmTwee`.

De notatie `document.frmTwee['voornaam']` gebruiken we vooral als we programmatorisch de `name` moeten berekenen.

*Is het `name` attribuut wel nodig als er een `id` is?*

**het `name` attribuut van een control mag je nooit achterwege laten!**

Als een formcontrol geen `name` attribuut heeft, zal zijn waarde niet beschikbaar zijn in het *naam=waarde* array dat **gesubmit** wordt.

Op het vlak van gegevens doorsturen is het `id` attribuut handig om het control te manipuleren of te valideren, maar niet essentieel.

Noteer ook dat het niet noodzakelijk is, dat `name` en `id` identiek zijn.

## 9.3 formulierelementen kookboek

Hoe lees je de waarde van een set keuzerondjes? hoe lees je een *multi-select* keuzelijst? Hoe update ik de waarde van één veld door middel van een ander? deze en andere recepten vind je hier kort gedemonstreerd.
Als je vind dat er nog een receptje bijgevoegd moet worden, laat maar weten.

### 9.3.1 De waarde van een tekstveld lezen en schrijven

Laat ons alle `input` elementen van het `type` "text" samen met `textarea` benoemen als *tekstvelden*:

```
<input type="text" name="voornaam" id="voornaam" value="typ uw voornaam hier" />
<textarea name="commentaar" id="commentaar" cols="40" rows="6">
  uw commentaar hier</textarea>
```

Een `input` element is dus een EMPTY element, het heeft dus geen zin er een *textNode* van te gaan zoeken.

- de **ingegeven waarde** van het veld kan je schrijven en lezen met de DOM `value` property
- De **standaardwaarde** van zo'n veld kan je in het HTML **attribuut** `value` plaatsen zoals hierboven.
  Je kan deze standaardwaarde schrijven en lezen met de DOM `defaultValue` property.
- De `defaultValue` property **wordt niet beïnvloed** door de `value` property

In dit voorbeeld wordt een lege string als waarde ingegeven zodra het veld de `focus` krijgt. Daardoor 'verdwijnt' de `defaultValue`, maar keert terug bij het verlaten van het veld als de `value` nog steeds een lege string is (en dus niet ingevuld werd):

```
var eVoornaam = document.getElementById('voornaam');
eVoornaam.onfocus= function (){
  this.value = "";
}
eVoornaam.onblur= function (){
  this.value = (this.value=="")?eVoornaam.defaultValue:this.value;
}
```

Opmerkingen:

- `this` is hier het control zelf omdat het in een event handler opgeroepen wordt
- bedenk ook dat met deze methode, als je niets invult, de `defaultValue` doorgestuurd wordt naar de server bij het `submit` van het form!
  Die waarden zal je dus moeten opvangen of een andere techniek gebruiken.

Het verbazende is dat dit evengoed werkt met een textarea, probeer maar even en vervang het voornaam veld door het commentaarveld.

### 9.3.2 Welk keuzerondje is aangevinkt?

Een set keuzerondjes zijn een aantal `input type="radio"` elementen met **dezelfde** `name` waarvan de aangeduide waarde doorgegeven wordt als het formulier gesubmit wordt:

```
<p>kies een kleur:
  <label>rood<input type="radio" name="kleur" id="rodekleur" value="rood" /></label>
  <label>geel<input type="radio" name="kleur" id="gelekleur" value="geel"
  checked="checked" /></label>
  <label>blauw<input type="radio" name="kleur" id="blauwekleur" value="blauw"
  /></label>
</p>
```

Merk op dat – indien je ze een `id` wil geven – deze niet dezelfde mag zijn. Het is meestal onnodig een `id` te gebruiken want we gebruiken bij voorkeur DOM0 om ze te bereiken als 'set':

```
var kleurKeuzes = document.frmTwee.kleur;
```

- Infeite kan je de groep beter beschouwen als een **array** van keuzerondjes waarin `document.frmTwee.kleur[0]` het eerste keuzerondje is, `document.frmTwee.kleur[1]` het tweede, etc...
  De `document.frmTwee.kleur.length` geeft het aantal keuzerondjes aan.
- Het aangevinkte keuzerondje heeft een DOM eigenschap `checked=true`
- Je kan één van de rondjes standaard aanvinken door het HTML attribuut `checked="checked"` mee te geven

Om dus te weten te komen welk keuzerondje aangevinkt is en wat zijn `value` is kunnen we als volgt tewerk gaan:

```
var kleurKeuzes = document.frmTwee.kleur;
for(var i=0;i<kleurKeuzes.length;i++){
        if(kleurKeuzes[i].checked==true){
                break;
        }
  }
  alert("keuzerondje met index " + i + " gekozen met waarde " + kleurKeuzes[i].value);
```

Opmerkingen:

- we raden je aan **altijd** één van de keuzerondjes een standaard `checked="checked"` mee te geven
- de lus controleert de checked eigenschap en breekt af als hij die vindt: de waarde van `i` is dus diegene met de eigenschap

### 9.3.3 Welke checkboxes zijn aangevinkt?

Checkboxes zijn `input type="checkbox"` elementen die dezelfde `name` kunnen – maar niet hoeven te – hebben. Is er sprake van een groep, zoals hieronder dan worden alle aangeduide waarden meegestuurd met de submit:

```
<p>waar bent u ooit geweest:
  <label>Brugge<input type="checkbox" name="stad" value="Brugge" checked="checked"
  /></label>
  <label>Gent<input type="checkbox" name="stad" value="Gent" /></label>
  <label>Oostende<input type="checkbox" name="stad" value="Oostende" /></label>
</p><p>
  <label>Wil u graag spam ontvangen?<input type="checkbox" name="spam" value="ja"
  checked="checked" /></label>
</p>
```

- ook hier kan je de checkbox standaard aanvinken met `checked="checked"`

Van een enkele radiobutton vragen we gewoon de `checked` property op:

```
function spamKrijgen(){
        alert(document.frmTwee.spam.checked + "aangevinkt") ;
}
```

Voor een groep moeten we opnieuw lussen doorheen de groep zoals bij keuzerondjes:

```
function welkeStad(){
  var steden = document.frmTwee.stad; //infeite een array
  for(var i=0;i<steden.length;i++){
                if(steden[i].checked==true){
                        alert(steden[i].value + " bezocht")
                }
        }
}
```

### 9.3.4 Wat is gekozen in een keuzelijst?

Een keuzelijst is een `select` element met een aantal `option` elementen als *children*:

```
<select name="muziek" id="muziek">
    <option value="" selected="selected">-- kies uw muziek --</option>
    <option value="kl">klassiek</option>
    <option value="jz">jazz</option>
    <option value="sc">schlager</option>
    <option value="pp">pop</option>
    <option value="funk">funk</option>
</select>
```

Even uw geheugen opfrissen:

- de `option` elementen kunnen **wel** of **niet** een `value` attribuut hebben: zoniet wordt de zichtbare **tekstwaarde** de doorgegeven waarde, anders de `value`

- een `option` element kan **standaard geselecteerd** worden met een `selected="selected"` attribuut
- het *huidige geselecteerde item* heeft een *index* gegeven door de `selectedIndex` DOM property. Let op! dit geeft zijn positie in de `option`'s lijst, niet zijn `value`
- indien er **niets geselecteerd** is, dan is de `selectedIndex` gelijk aan `-1`
- dikwijls zet men in een `select` een eerste item met de `value` "" (lege string) en de tekst "*kies hier uw …*". Deze waarde moet je natuurlijk kunnen detecteren: via zijn `value.`
- een `select` element kan een `size` attribuut hebben, bv `size="4".` In dat geval wordt het een open lijst met vier zichtbare rijen, geen 'dropdown' lijst meer
- het `select` element kan een `multiple="multiple"` attribuut hebben: dan kan je meerdere items aanduiden. Praktisch enkel te gebruiken als je terzelfdertijd ook `size` gebruikt (zie verder)

De eenvoudigste manier om alles over het geselecteerde item te weten te komen is als volgt:

```
function welkeMuziek(){
  // een single-select select element
  var muziek     = document.frmTwee.muziek; // of document.getElementById('muziek')
  var index      = muziek.selectedIndex;
  var waarde     = muziek.value;
  var tekst      = muziek.options[index].firstChild.nodeValue;
  var str        = "selectedIndex:" + index + " value:" + waarde + " tekst:" + tekst
  alert(str);
}
```

Met een test erbij wordt dat:

```
function welkeMuziek(){
  // een single-select select element
  var muziek     = document.frmTwee.muziek; // of document.getElementById('muziek')
  var index      = muziek.selectedIndex;
  var waarde     = muziek.value;
  var tekst      = muziek.options[index].firstChild.nodeValue;
  if(index == -1 || waarde == ""){
         alert("maak eerst een geldige keuze uit de keuzelijst, aub")
  }
  else{
         var str = "selectedIndex:" + index + " value:" + waarde + " tekst:" + tekst
         alert(str);
         }
}
```

### 9.3.5 Wat is gekozen in een multi-select keuzelijst?

Voor een meerkeuze-keuzelijst is de situatie wat complexer: we kunnen niet zomaar `value` gebruiken. Deze geeft ons gewoonweg het eerste item en niet de andere geselecteerde.

Om het onderscheid te maken tussen een meerkeuze lijst en een gewone gebruiken we de `type` property die voor een `select` slechts één van deze 2 waarden kan hebben: "*select-single*" of "*select-multiple*".

De functie in het onderstaande voorbeeld returnt een array van subarrays waarin de index, de value en de tekst van de geselecteerde items zitten. De functie kan zowel "*select-single*" als "*select-multiple*" behandelen.

```
var cursus      = document.getElementById("cursus");
console.log(welkeKeuzes(cursus));


function welkeKeuzes(e){
  // zowel voor select-multiple als voor select-single SELECT elementen
  var tag               = e.tagName;
  var type              = e.type;
  var arrSelected       = new Array();


  if(tag == "SELECT"){
   if(type == "select-multiple"){
    for (var j=0;j<e.options.length; j++){
     if (e.options[j].selected) {
         arrSelected.push(j , e.options[j].value , e.options[j].firstChild.nodeValue)
         }
    }
   }
   else if(type == "select-one"){
     arrSelected.push(e.selectedIndex, e.value,
                      e.options[e.selectedIndex].firstChild.nodeValue);
         }
  }
  return arrSelected;
}
```

Bespreking:

- De functie controleert of het doorgegeven element `e` wel een select is
- Daarna bekijkt hij het `type` en in het geval van een *select-multiple*
- Lust hij doorheen alle `option` elementen en kikt of ze `selected` zijn
- De waarden van een geselecteerd `option` element worden in het *arrSelected* geplaatst
- Een console statement toont het gereturnde array

### 9.3.6 Hoe toon/verberg | disable/enable ik een veld via een checkbox?

Veronderstel de volgende formelementen:

```
<p>
<label><input type="checkbox" name="retourcheck" id="retourcheck"  />
retour?</label>
</p>
<p>
<label>datum heenvlucht:
<input type="text" name="heendatum" id="heendatum"  title="datum heenvlucht" />
</label>
<label id="retourlabel" style="display:none" >datum heenvlucht:
<input type="text" name="retourdatum" id="retourdatum" title="datum terugvlucht" />
</label>
</p>
```

Bij het begin staat het checkboxje *retourcheck* uitgevinkt. Het label *retourlabel* met het veld *retourdatum* is niet te zien want er staat een **inline** style in `display:none.`

Dan kunnen we de volgende eventhandler maken voor de checkbox:

```
var retourcheck    = document.getElementById('retourcheck');
var retourlabel    = document.getElementById('retourlabel');
var retourdatum    = document.getElementById('retourdatum');
retourcheck.onclick = function(){
  toonVerbergVeld(this,retourlabel);
}

//************************
function toonVerbergVeld(ch,e){
// toggled de display van een element
// @ch   = this = checkbox
// @e    = welk element te togglen

  if(e && ch.tagName == "INPUT" && ch.type == "checkbox"){
        var aan = (ch.checked);
        if(aan == true){
              e.style.display = "inline";
              e.focus();
        }
        else{
              e.style.display = "none";
        }
  }
}
```

Bespreking:

- We gebruiken de `onclick` event listener ipv `onchange` voor de checkbox omdat IE met deze laatste een probleem heeft
- De eventhandler *toonVerbergVeld* heeft twee argumenten: `this`, de checkbox zelf en *e*, het element dat hij moet wisselen

- Hier geven we *retourlabel* door als wisselelement, dat het invulveld *retourdatum* bevat
- We stellen de inline `display` property in op "*inline*" of "*none*", afhankelijk van de waarde van `ch.checked`: `true` of `false`
- We plaatsen de `focus` of het veld bij aan==true : omdat het label het invulveld bevat, zal de cursor goed staan in de meeste browsers.

Waar je echter ook rekening mee moet houden is het feit dat een veld verbergen de waarde er in niet wijzigt. Als je dus bv. de retourdatum invult en die daarna verbergt, hij zijn waarde blijft bevatten en die ook zal doorsturen naar de server…
we passen ons script dus als volgt aan:

```
…
if(e && ch.tagName == "INPUT" && ch.type == "checkbox"){
  var velden = e.getElementsByTagName('input'); //alle velden die in e zitten
   var aan = (ch.checked);
          if(aan == true){
                  e.style.display = "inline";
                  e.focus();
          }
          else{
                  e.style.display = "none";
                  for(var i=0;i<velden.length;i++){
                          velden[i].value ="";
                  }
          }
  }
…
```

Willen we een veld enkel activeren/deactiveren met een checkbox, dan is het script erg gelijkaardig. Veronderstel deze formelementen:

```
<p>
<label><input type="checkbox" name="spamcheck" id="spamcheck"  /> wil u graag hopen
  spam ontvangen?</label>
<label>emailadres: <input type="text" name="spamemail" id="spamemail"  title="uw
  emailadres" disabled="disabled" /></label>
</p>
```

Het invulveld *spamemail* is gedeactiveerd door het `disabled="disabled"` attribuut.
Een script als eventhandler:

```
var spamcheck     = document.getElementById('spamcheck');
var spamemail     = document.getElementById('spamemail');
spamcheck.onclick = function(){
                enableDisableVeld(this,spamemail);
                }
```

```
function enableDisableVeld(ch,e){
// enabled/disabled een veld
// @ch = this = checkbox
// @e = welk element te togglen
if(e && ch.tagName=="INPUT" && ch.type=="checkbox"){
  e.disabled = (!ch.checked);
  var onbelangrijk = (ch.checked)?e.focus():null;
  }
}
```

Bespreking:

- De property `disabled` van het element *e* is het omgekeerde van de property `checked` van *ch*: dus als de checkbox *checked* is (`true`), dan moet het veld niet *disabled* zijn (`false`)
- Daarna voeren we een *vuil trukje* uit: we gebruiken de **ternary operator** om te beslissen of we een `focus()` uitvoeren op het veld *e* of niet.
  De ternary operator heeft de bedoeling een waarde in de variabele *onbelangrijk* te plaatsen, maar dat interesseert ons hier niet

# 10 Elementen dynamisch opmaken

'DHTML' is de term die vroeger gebruikt werd voor het dynamisch manipuleren en opmaken van html. Zo werden bijvoorbeeld *roll-over* effecten, *drop-down menu's* en *in-page tabblad* navigatie DHTML toepassingen genoemd.

Infeite gaat het hier om DOM Scripting, Javascript en CSS, 'DHTML' is geen zuivere techniek maar een samenraapsel van technieken.

In dit korte hoofdstukje beperken we ons tot kennis die je nodig hebt voor de **dynamische opmaak** van elementen, waarbij een hele goede kennis van CSS essentieel is: hoe meer je van CSS afweet, hoe meer je kan bereiken.

## *10.1 Inline style of stylesheet ?*

De opmaak van een element kan bepaald worden door een html-attribuut of door CSS styles. Vaste html attributen gebruiken we simpelweg niet meer voor opmaak: "scheiding van opmaak en inhoud", alle opmaak gebeurt via CSS.
CSS opmaak wordt geregeld door de cascade: de opmaak kan van één of meerdere **stylesheets** komen of van een **inline style rule**.
Daarbij worden een aantal opties toegepast:

het dynamisch toepassen van **inline style**
het toekennen/verwijderen van een `class` attribuut zodat het element reageert op een aanwezige CSS-rule in het stylesheet
gebruik maken van contextuele CSS
het dynamisch wisselen van stylesheet

Hoewel het ook mogelijk is de stylesheet-rules van een stylesheet dynamisch te wijzigen, wordt dat quasi nooit gebruikt omdat deze techniek erg complex is en het sop de kolen niet waard.

We overlopen even kort de mogelijkheden en gaan er daarna dieper op in:

**inline style toepassen:**

Inline styles zijn de style rules in het `style` **attribuut** van een element. Dat kan ook via scripting gezet/gelezen worden.
Een inline style toepassen op een element heeft enkele duidelijk voordelen:

- een inline style heeft **altijd de overhand** op stylesheet rules (uitzondering `!important`)
- erg korte en duidelijk code

Daarom wordt dit meestal gebruikt voor het snelle werk: toepassen van één style rule (hoewel ook meerdere style rules mogelijk zijn).

```
var pree = document.getElementsByTagName('pre')
pree.style.backgroundColor = 'rgb(128,44,255)';
```

`class` toekennen/verwijderen:

Het dynamisch toekennen/verwijderen van een `class` wordt meestal gebruikt om meer complexe opmaak toe te kennen: style rules al aanwezig in het stylesheet. Aanpassing van de opmaak gebeurt in het stylesheet en staat los van de code. Een `class` kan even snel weer verwijderd worden.

**contextuele CSS:**

Hieronder verstaan we het effect dat je krijgt als je een element **verplaatst** in de DOM tree: veronderstel een `h1` in een `div`. Als je die `h1` verplaatst (via DOM scripting) naar een andere container dan kan zijn opmaak totaal verschillen zonder er iets voor te moeten doen:

```
h1 {font-family: Georgia, Times New Roman, Times, serif;}

#div.links h1 {
  font-size: 24px;
  color:blue;
}
#div.rechts h1 {
  font-size: 18px;
  color:red;
}
```

Deze techniek kan enkel in heel specifieke situaties gebruikt worden en steunt volledig op de aanwezige CSS.

**stylesheet switcher:**

In sommige situaties wil je de mogelijkheid bieden aan de gebruiker om te wisselen van stylesheet.

## *10.2 Inline styles wijzigen*

Een inline style instellen gebeurt eenvoudig met het `style` object van een elementNode. Het is volledig cross-browser:

```
elementNode.style.backgroundColor = 'rgb(128,44,255)';
```

*denk er aan dat de CSS styles, bv. "`background-color`" , in DOM vertaald worden naar **`properties`**, zoals hier "`backgroundColor`".*

*De volledige lijst van geldige CSS2 properties kan je vinden op*

*http://www.w3.org/TR/2000/REC-DOM-Level-2-Style-20001113/css.html#CSS-CSS2Properties*

Om de inline `style` van één element te wijzigen moeten we het enkel refereren en de style:

```
var divOutput = document.getElementById('output');
divOutput.style.display='none';
```

In dit voorbeeldje 'verdwijnt' het element divOutput door zijn `display` property op 'none' te zetten.

Om een collection elementen te wijzigen, moet je doorheen alle elementen van de collection lussen:

```
var divs = document.getElementsByTagName('div');  //collection
for(var i=0;i<divs.length;i++){
        divs[i].style.border = "1px solid red";
  }
```

Om een inline style te wissen, stel je de property in op een lege string:

```
divs[i].style.border = "";
```

Om meerdere style rules terzelfdertijd inline in te stellen gebruiken we de property `cssText` van het `style` object:

```
divs[i].style.cssText = "border:1px solid red; background-color:#CCFFCC;color:blue;";
```

`cssText` bevat dus de echte CSS style rules zoals in een selector: let op de schrijfwijze.
Om dit ongedaan te maken stel je opnieuw de eigenschap in op een lege string.
Noteer dat daarmee alle inline opmaak verwijderd wordt, ook deze via de aparte properties ingesteld.

```
divs[i].style.cssText = "";
```

Om een inline style te lezen moet je eerste bedenken dat een bepaalde property afwezig kan zijn: enkel als die ingesteld geweest is – al of niet op een lege string – is die aanwezig:

```
if(divs[i].style.display=="block") { divs[i].style.display="none" }
```

Dit voorbeeld probeert de items in de collection te verbergen. Helaas gaat de auteur ervan uit dat de `div's` een `display` property hebben en dat deze ook nog eens op "block" staat, enkel omdat ze nu zichtbaar zijn?

Neen, als de pagina geladen wordt hebben deze elementen waarschijnlijk geen enkele inline style, misschien een styleheet rule, maar die zijn hier toch van geen belang.
Beter ware:

```
var getoond = divs[i].style.display;
if((getoond=="")||(getoond=="block")) divs[i].style.display="none";
```

Of een volledige 'toggle' functie:

```
divs[i].style.display=
  ((divs[i].style.display=="")||(divs[i].style.display=="block"))?"none":"block";
```

## *10.3 class wijzigen*

De beste en waarschijnlijk ook de snelste tactiek gebruikt in 'DHTML' is het toekennen of verwijderen van een **CSS class** aan de *elementNode*. Die classes zet je natuurlijk klaar in het stylesheet. Bij elke wijziging van class wordt de DOM tree opnieuw opgemaakt en laten we het dus aan CSS over om de wijzigingen te doen.
Deze techniek is minstens even snel als een inline style zetten.

De property `className` returnt een `string` met de inhoud van `class`:

```
alert(document.getElementById('testdiv').className)
```

Eén praktisch probleempje is dat je meerdere *classes* aan een element mag toewijzen. In een `class` attribuut zouden deze gescheiden staan door spaties:

```
<div id="testdiv" class="kolom rood links"></div>
```

Als we dus een *class* toevoegen aan een element, moeten we opletten dat we de aanwezige classes niet overschrijven:

```
node.className += " "  + klasse;
```

Hierboven wordt een klasse achteraan toegevoegd en er wordt een spatie voorzien als tussen ruimte.
Ook bij het verwijderen moeten we enkel het gevraagde wissen en de rest laten staan:

```
var r = new RegExp(klasse);
node.className = node.className.replace(r,'');
```

Hier gebruiken we een regular expression met de `replace` functie om de klasse te verwijderen.

Hieronder een functie die zowel classes kan toevoegen als verwijderen voor enkele elementen én voor collecties:

```
function changeClass(nodes,klasse,actie){
// voegt een class  toe of verwijdert die aan een elementNode
/*
@nodes   : enkel elementNode of een collection van elementNodes, minstens 1
@klass   : string met className
@actie   : "+" of "-"
*/
// één node of collection(array)?
if((nodes.constructor!=Array)&&(nodes.nodeType==1)){
         nodes=[nodes];
  }
switch (actie) {
         case "+" :
                addClass(nodes, klasse);
                break;
         case "-" :
                removeClass(nodes,klasse);
                break;
         default :
                throw new Error('verkeerd argument: gebruik een + of een -')
  }
  function addClass(nodes, klasse){
  //voegt een class toe
         for(var i=0;i<nodes.length;i++){
                nodes[i].className += " "  + klasse;
         }
  }
  function removeClass(nodes, klasse){
  //verwijdert een class
         var r = new RegExp(klasse);
         for(var i=0;i<nodes.length;i++){
                nodes[i].className =nodes[i].className.replace(r,'');
         }
  }
}
```

## *10.4 Stylesheets manipuleren*

Het `document` bevat een `styleSheets` collectie:

`document.styleSheets[i]`

Deze collectie bevat zowel

externe stylesheets: via een `link` element
interne stylesheets: in een `style` element in de pagina.

Stylesheets gekoppeld via een `@import`-rule worden niet meegeteld en zitten dus verwerkt in het stylesheet waarin ze geïmporteerd worden.
De volgorde van deze stylesheets volgt de flow van het document, dus `document.styleSheets[0]` is het eerste stylesheet dat gelezen wordt en `document.styleSheets[1]` het tweede. De vraag is natuurlijk of jij die volgorde kent?
Een alternatieve manier van benaderen – en **aan te raden** om verwarring te vermijden - is een `id` geven aan het stylesheet!

```
var SS2 = document.getElementById('SS2');
```

### 10.4.1 Stylesheets uitzetten

Een `styleSheet` object heeft een `boolean` eigenschap `disabled` waarmee je het kunt aan/afzetten

```
var SS2 = document.getElementById('SS2');
SS2.disabled = true;
```

of

```
document.styleSheets[1].disabled = true
```

### 10.4.2 Wisselen van stylesheet

Een `styleSheet` object heeft een `href` eigenschap (W3C readonly - IE read/write) ; voor externe stylesheets (via `link`) bevat dit de locatie van het externe CSS bestand, voor interne stylesheets (via `style`) is dit `null`.

Omdat `href` readonly is in Moz browsers kunnen we beter een andere tactiek volgen: het wijzigen van het `href` attribuut van het `link` element (W3C `ownerNode`, IE `ownerElement`)

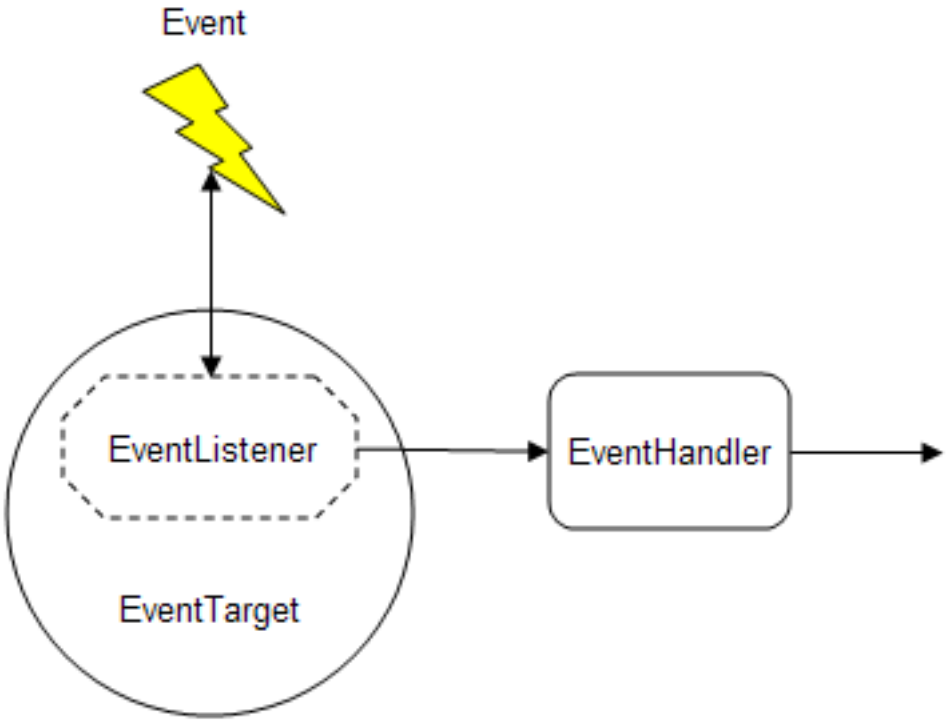
```
document.styleSheets[0].ownerNode/ownerElement.setAttribute('href','nieuwStyleSheet.c
  ss')
```

Denk er aan dat je zeker moet zijn dat het hier een `link` element betreft.

# 11 Events

Elke moderne programmeertaal bevat *Events* : *gebeurtenissen*. Je bent er zo vertrouwd mee dat je het niet meer beseft: de reactie op een klik, een mouse-over op een menuutje, etc...Maar er zijn ook Events die je niet zo duidelijk ziet: het laden van een pagina, het sturen van een formulier, het opwekken van een fout in een script, het wijzigen van de DOMtree.

## *11.1 Wat is een Event?*




De **Event flow** is het proces vanaf de start van een **Event**, het bestaat minstens uit een **Event Listener**, een **Event Target**, een **Event** en een **Event Handler**:
De Event Listener is een stukje code geregistreerd voor een object, de Event Target, dat wacht tot een Event gebeurt.
Er zijn heel wat ingebouwde Event Listeners in JS: zo bevat een hyperlink een ingebouwde listener voor het `click` Event: het gaat vanzelf, je hoeft er niets voor te doen. Maar je kan zelf Event Listeners registreren voor één of meerdere objecten, zie verder.

Het Event is zeer specifiek: een `click` op een knop, een `mouseover` op een foto, een `load` van de pagina, een `readyState` van een Ajax call.
De Event Handler is het programma dat reageert als het Event plaatsgevonden heeft. Ook hier zijn er veel ingebouwde Event Handlers waar je niets voor hoeft te doen: zo laadt een `click` op een hyperlink vanzelf het gevraagde url in het window. Maar opnieuw kan je zelf een Event Handler maken.

Een voorbeeld brengt duidelijkheid. Veronderstel deze HTML:

```
<button id="hide">verberg details</button>
```

En deze JS code:

```
var knop = document.getElementById('hide');
knop.addEventListener('click', verberg);

function verberg(){
  this.parentNode.style.display = 'none';
}
```

De knop heeft een *Event Listener* geregistreerd voor het `click` event. Bij een `click` schiet de *Event Handler* in gang: de functie *verberg().* Deze functie maakt gebruik van het `this` keyword die het *eventTarget* voorstelt: de knop zelf. De functie zoekt de `parentNode` van de knop en verbergt die. De Event Flow is hier duidelijk en eenvoudig.

Bij het klikken op een Submit knop van een formulier gebeurt echter veel meer. Het klikken op zich is bevat al de events `mousedown`, `mouseup` en `click`. Submit knoppen hebben ingebouwde EventListeners voor deze events. Je hoeft er geen speciale registratie voor te doen (m.a.w. je moet er geen `onclick` in zetten). Door Event bubbling wordt echter ook de event `submit` van het formulier afgevuurd.

Dit laatste voorbeeld maakt duidelijk dat het niet altijd zo eenvoudig is als het lijkt: zo vuren dikwijls meerdere Event Listeners af en is het mogelijk voor elk een andere Event Handler te hebben.

## *11.2 Event Handlers in HTML elementen*

Vroeger werden Event Listeners in de HTML van een element geplaatst zoals

```
<div onmouseover="animate()" >…</div>
```

Dit willen we echter ten allen prijze vermijden: we **scheiden scripting van inhoud**. Event Listeners worden nu exclusief geregistreerd via Javascript zoals hieronder beschreven. Doe dit niet meer!

## *11.3 Veel voorkomende events*

Enkele veel voorkomende events en hun EventTarget (het element waarop ze plaatsvinden) . Vooral formulierelementen zijn van belang.

| HTML element | Event | beschrijving |
|---|---|---|
| ***Algemeen*** | | |
| | `click` | Klikken is een opeenvolging van de Events mousedown, mouseup en click. Deze laatste vuurt als de twee eerste voltooid werden.<br>Als de gebruiker de muisknop indrukt boven één element en weer loslaat boven een ander kunnen |

| | | |
|---|---|---|
| | | er problemen ontstaan met het click Event, daarom wordt aangeraden eerder een mouseup te gebruiken om een klik waar te nemen. |
| | dblclick | dubbelklik |
| | mousedown | muisknop ingedrukt |
| | mousemove | muis bewogen |
| | mouseout | pointer niet meer boven Target |
| | mouseover | pointer komt boven Target |
| | mouseup | muisknop losgelaten |
| | touchstart | het aanraken van een touch device (zie verder) |
| | touchmove | het bewegen van een aanrakingspunt |
| ***Formulieren*** | | |
| form | submit | formgegevens worden doorgestuurd |
| form | reset | form reset gebeurt |
| button, label, select, input, textarea | focus | element krijgt de focus |
| button, label, select, input, textarea | blur | element verliest de focus |
| select, input, textarea | change | waarde verandert of andere selectie |
| input, textarea | select | tekst geselecteerd |
| ***Window, document,*** | | |
| window, document, body, frameset, iframe, img | load | document is volledig geladen |
| window, document | unload | document wordt ontladen |

Zoals je merkt zijn hier duidelijk twee groepen in te onderscheiden: Events die direct door de gebruiker afgevuurd worden, zoals een `mousemove`, en interne Events, zoals een `load`, die door de browser of door programma's opgewekt worden of als gevolg van een andere Event.

## *11.4 Event registratie*

Sommige HTML element komen met ingebouwde event listeners en handlers ( hyperlinks, formulieren), andere niet en dan moet je zelf een event registreren.
Er zijn twee manieren om dat te doen:

- volgens de oude manier (DOM 0): eenvoudig, maar beperkt

- volgens de nieuwe manier (DOM 2): iets complexer, maar veel meer mogelijkheden

*Gebruik steeds het nieuwe event registratie model*

### 11.4.1 de oude manier van registratie

In het oude model wordt een Event Listener als een **property** geschreven:

```
element.onclick = doeIets;
function doeIets() {…}
```

Let op de **afwezigheid van haakjes** bij de naam van de Event Handler functie: het is een verwijzing naar de functie. Op die manier wordt de functie uitgevoerd bij de uitvoering van de Event: `onclick()`.

Je kan uiteraard ook een anonieme functie gebruiken:

```
element.onclick = function(){ alert('you rang, myLord?') }
```

Het oude model heeft nadelen:

- er kan slechts één eventhandler per event ingesteld worden
- het verwijderen van de eventhandler verwijdert **alle** eventhandlers voor dit event
- het is dikwijls browser-specifiek, dus moet jeaan browsertesting gaan doen
- er is geen controle over in welke event fase (bubbling of capture) het event opgevangen wordt

### 11.4.2 Het nieuwe Event registratie model

http://www.w3.org/TR/DOM-Level-2-Events/ en http://www.w3.org/TR/DOM-Level-3-Events/ voorzien een nieuwe standaard voor Events. Om een Event te registreren gebruiken we de method `addEventListener()`

```
element.addEventListener(event, handler, useCapture)
```

De argumenten:

- *`event`* is hier een event zoals `click`, `submit`, `mouseover`, `touchstart`.
- *`listener`* is de functie die als Event Handler optreedt
- *`useCapture`* is een boolean die de *Event propagation* bepaalt (zie verder).

Het grote voordeel van deze standaard is dat een element nu meerdere handlers kan hebben voor hetzelfde event:

```
var dief = document.getElementById('speciaal');
if(dief.addEventListener){
  dief.addEventListener('mouseover',oplichten,false);
```

```
  dief.addEventListener('mouseover',kleuren,false);
  dief.addEventListener('mouseover',vergroten,false);
}
```

Je bemerkt hier ook *feature detection*: we testen eerst voor de aanwezigheid van de method via `if(dief.addEventListener)`. Dat is een goede veiligheidsmaatregel, maar je mag ervan opaan dat deze methods nu door alle browsers ondersteund worden. (Ook Internet Explorer vanaf IE9).

De registratie van een event kan verwijderd worden met de method `removeEventListener()`:

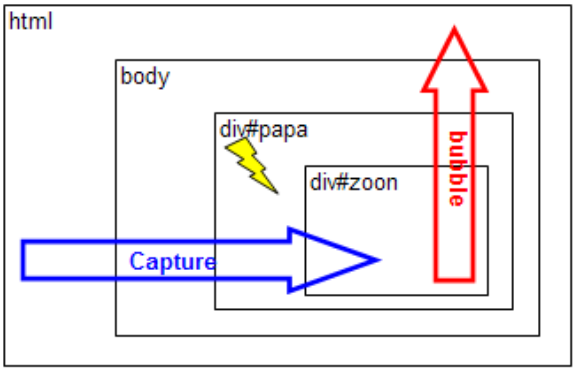
```
element.removeEventListener(event, handler,useCapture)
```

Waarbij alle argumenten exact de argumenten moeten reflecteren die gebruikt warden bij de registratie, anders werkt het niet. Als dus meerdere handlers geregistreerd zijn voor dezelfde eventTarget, kan je er perfect eentje van tussenuit halen.

### 11.4.3 Event fases

Een HTML pagina bestaat infeite uit een reeks van geneste containers, vb. een `p` element zit in een `div` dat zelf in een andere `div` zit in en tenslotte in de `body` dat in het `document` zit.
Wat gebeurt er als één van die containers, bv. `div#papa`, een event geregistreerd heeft - een `click` bijvoorbeeld ?

De `click` wordt infeite uitgevoerd op de childNode `div#zoon`.




In DOM Level 2 doorloopt het event de DOM tree tweemaal:

eerst van 'boven naar beneden': van html tot `div#zoon`:
dit noemen we **Event Capture**
Daarna doorloopt het event de tree nogmaals, deze keer van 'beneden naar boven', dus van `div#zoon` tot `html`:
dit noemen we **Event Bubbling.**

Het klassieke (DOM 0) model kent enkel event bubbling: het event passeert eerst `div#zoon`, maar omdat deze voor `click` geen registratie heeft, gebeurt er niets, daarna gaat het naar `div#papa` die wel een handler heeft, daarna naar `body` enz..
Elke eventhandler voor `click` die het event tegenkomt wordt uitgevoerd.

Bij een DOM level 2 registratie kan je kiezen of een event bij de capture fase (`true`) ofwel bij de bubbling fase (`false`) onderschept moet worden.
IE ondersteunt enkel event bubbling.

In sommige gevallen wil je liever geen bubbling (omdat je bijvoorbeeld een event geregistreerd hebt op een reeks geneste containers): dan kan je dat annuleren voor dat event door de eigenschap `cancelBubble` op `true` te zetten.

## *11.5 het Event object*

Het event zelf is bereikbaar als een object met attributen. Die kunnen we gebruiken om allerlei interessante dingen te weten te komen. Bijvoorbeeld:

- het soort event
- het element waarop het event afgevuurd werd
- het element waarop het event geregistreerd werd
- het element dat de pointer net verlaten heeft
- welke muisknop werd geklikt/losgelaten
- de positie van de pointer
- welke toets werd ingedrukt/losgelaten
- waar precies je met de vinger een touchscreen aanraakte
- etc.…

We vangen het `Event` object als een argument `e` (of eender welke variabelenaam) in de eventhandler:

```
function eventHandler(e){
console.log(e.type)
  ...
}
```

Voornaamste attributen

| Attribuut | beschrijving |
|---|---|
| `type` | returnt een `String` die aangeeft welke soort event plaatsheeft. Bijvoorbeeld "mouseup"," mouseover","submit"," load","touchmove", ... |
| `target` | de ***Event Target*** , is het element waarop het Event momenteel afvuurt |
| `currentTarget` | is het element waarvoor het Event oorspronkelijk **geregistreerd** werd.<br>Dit is meestal identiek aan `target`, maar niet altijd |
| `relatedTarget`<br>(enkel bij `mouseover`) | is het element dat net verlaten werd (vr `mouseover`), of net betreden wordt (vr `mouseout`) |

In de meeste scripts is het vooral van belang te weten op welk element het Event momenteel plaatsheeft: dat element is de `target`. Dus, over welk element beweegt de muis nu, op welk element wordt er geklikt, welk formulier submit, etc...
Een voorbeeld: veronderstelt twee geneste `div` elementen

```
<section id="test">
<div id="outer">
      <div id="inner"></div>
 </div>
</section>
```

met volgende CSS:

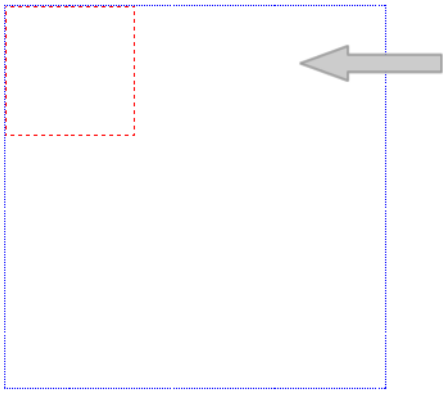
```
#outer{
  width: 300px;
  height:300px;
  border:1px dotted blue;
  }
#inner{
  width: 100px;
  height:100px;
  border:1px dashed red;
  }
```

we registreren een `mouseover` en `mouseout` enkel voor "outer":

```
  var outer = document.getElementById('outer');
  var inner = document.getElementById('inner');
  outer.addEventListener('mouseover',hoverOut);
  outer.addEventListener('mouseout',hoverOut);

  function hover(e){
         displayEventInfo(e);
         this.style.backgroundColor = "orange";
         }
  function hoverOut(e){//this test
         displayEventInfo(e);
         this.style.backgroundColor = "yellow";
         }
function displayEventInfo(evt){
  if(evt){
   txt = "type: "       + evt.type + "\n";
   txt += "eventPhase: " + evt.eventPhase + "\n";
```

```
    txt += "cancelable: "+ evt.cancelable + "\n";
    txt += "bubbles: "   + evt.bubbles + "\n";
    txt += "timeStamp: " + evt.timeStamp + "\n";
    txt += "target: "    + evt.target.nodeName + ", id: " + evt.target.id + "\n";
  if(evt.currentTarget) txt += "currentTarget: " + evt.currentTarget.nodeName +
          ", id: " + evt.currentTarget.id + "\n";
  if(evt.relatedTarget) txt += "relatedTarget: " + evt.relatedTarget.nodeName +
         ", id: " + evt.relatedTarget.id +"\n";
  }
  else{
         txt = "non-W3C Event Model";
  }
  console.log(txt);
}
```

de functies *hover* en *hoverOut* wijzigen de achtergrondkleur van de `div` elementen.
De functie *displayEventInfo* zal ons alle mogelijke info tonen over de events die plaatshebben in de console.
Als we nu met de muis vanuit de pagina over de *outer* `div` bewegen

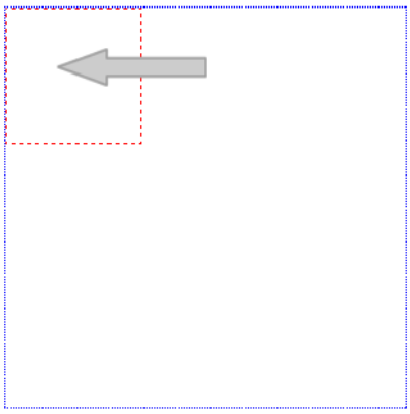


zien we dit:

```
type: mouseover
eventPhase: 2
cancelable: true
bubbles: true
timeStamp: 1372252982207
target: DIV, id: outer
currentTarget: DIV, id: outer
relatedTarget: SECTION, id: test
```

het event `type` is een `mouseover`.
de `target` en `currentTarget` zijn dezelfde: de *outer* `div`.
De `relatedTarget` is het element waar we vandaan komen : de `section`.

Bewegen we verder van *outer* naar *inner*



dan toont de console dit:

```
type: mouseout
eventPhase: 2
cancelable: true
bubbles: true
timeStamp: 1372253388826
target: DIV, id: outer
currentTarget: DIV, id: outer
relatedTarget: DIV, id: inner

type: mouseover
eventPhase: 3
cancelable: true
bubbles: true
timeStamp: 1372253388827
target: DIV, id: inner
currentTarget: DIV, id: outer
relatedTarget: DIV, id: outer
```

Er treden dus twee events op: eerst een `mouseout`, daarna een `mouseover`.
Waarom twee? we hebben toch enkel voor "outer" eventhandlers geregistreerd?
Dat komt omdat het event omhoog *bubbled*: het plant zich voort in zijn children.
Eenmaal de pointer bovenop "*inner*" dan merk je dat de `target` (event target) *inner* is, terwijl `currentTarget` nog steeds *outer* is: het element waarop het event geregistreerd werd. Deze verschillen kunnen kan van pas komen in je script.
De `eventPhase` is daar ook een reflexie van: 2 betekent *target*, 3 betekent *bubble*.

## *11.6 Standaardactie en cancelation*

Veel Events hebben een **standaard actie** die je soms moet tegenhouden omdat deze anders sowieso zal plaatshebben, wat je ook in je event handler stopt.
Een typisch voorbeeld is een hyperlink (`a` element ) die altijd reageert op een `click` en probeert naar het URI te gaan in het `href` attribuut (je bent verplicht daar iets in te zetten). Een ander voorbeeld is het `submit` event van een formulier: dat zal altijd plaatshebben wat er ook voor fout moge gebeuren.

Om die *default action* te kunnen annuleren moet een Event *Cancelable* zijn.
Bijvoorbeeld, de `submit` van een formulier is *Cancelable* en kan dus via een script tegengehouden worden.

Je kan te weten komen of een Event *Cancelable* is door zijn eigenschap `cancelable` op te vragen:

```
alert(e.cancelable)
```

Om de default action van een Event tegen te houden gebruiken we de Event *method* `preventDefault()`

Bijvoorbeeld: hier werd een validatiescript op een formulier gebruikt: alle velden worden eerst gecontroleerd vooraleer de gegevens door te sturen.

```
<form id="bestelform" action="bestelling.php" method="post">
…
<input type="submit" value="plaats bestelling" />
</form>
```

en de Javascript

```
var frm = document.getElementById('bestelform');
frm.addEventListener('submit', validate);

function validate(e){
  e.preventDefault();
  … //controles
  if(bOk===true){
         frm.submit();
  }
}
```

Het argument `e` in de functie *validate()* bevat het *event* zelf: we voeren er de method `preventDefault()` op uit zodat de standaardactie tegengehouden wordt, wat er ook gebeurt. We plaatsen deze method dan ook als eerste statement. Enkel als alle controles OK zijn, voeren we zelf de `submit` uit met de method `submit()` op het `form`.
De `confirm` is een dialoogvenster met een OK - Cancel knop. Het antwoord van de user is dus `true/false`. De returnwaarde van het script wordt nogmaals doorgegeven aan de `onclick` Event Handler.

## 11.7 `this` in event handlers

Het sleutelwoord `this` heeft een verschillende betekenis afhankelijk van de situatie waarin het gebruikt wordt, maar in een eventhandler stelt het steeds de EventTarget voor (het element waarop het event afvuurt), bijvoorbeeld:

```
<a id="klikmij" href="#">klik mij</a>
…
var lienk = document.getElementById('klikmij');
lienk.addEventListener('click',test);
```

```
function test(e){
  e.preventDefault();
  alert(this.nodeName)
}
```

In dit voorbeeld zal je "A" te zien krijgen want de event handler *test()* toont de `nodeName` van `this` en dat is de hyperlink zelf.

Dit kan heel handig zijn als je hele collections van elementen van dezelfde eventhandler voorziet: je hoeft je namelijk niet meer aan te trekken van welkeen er nu juist afgevuurd werd. Veronderstel ons eerder voorbeeld waarin een knop een onderdeel verbergt. Neem nu dat je een heleboel *detail-verberg-knoppen* hebt, elk in zijn eigen detail. Welkeen moet je dan verbergen?

```
…
<div class="detail">
  detailtekst detailtekst detailtekst detailtekst detailtekst detailtekst detailtekst
  <button class="hide">verberg details</button>
</div>
<div class="detail">
  detailtekst detailtekst detailtekst detailtekst detailtekst detailtekst detailtekst
  <button class ="hide">verberg details</button>
</div>

<div class="detail">
  detailtekst detailtekst detailtekst detailtekst detailtekst detailtekst detailtekst
  <button class ="hide">verberg details</button>
</div>
…
```

dan kunnen we het volgende doen:

```
var knoppen = document.querySelectorAll('.hide'); //collection
for(var i=0;i<knoppen.length;i++){
  knoppen[i].addEventListener('click', verberg);
}

function verberg(){
  this.parentNode.style.display = 'none';
}
```

Hier staat het keyword `this` enkel voor dié knop die aangeklikt werd en zal enkel die `parentNode` verborgen worden.

## 11.8 Muis Events

De meeste interactie tussen gebruiker en pagina vloeit voort uit MouseEvents. Daarbij vragen we ons af:

- op welk element klikt/beweegt de muis?
- welke muisknop wordt geklikt?
- wat zijn de coördinaten van de muisaanwijzer t.o.v. het venster / gepositioneerd element
- wordt er gesleept?

### 11.8.1 De MouseEvents

Een overzicht van de handelingen die je kan doen met een muis:

| Actie | Events | beschrijving |
|---|---|---|
| klik | `mousedown`, `mouseup`, `click` | Klikken is een opeenvolging van de Events `mousedown`, `mouseup` en `click`. Deze laatste vuurt als de twee eerste voltooid werden.<br>Als de gebruiker de muisknop indrukt boven één element en weer loslaat boven een ander kunnen er problemen ontstaan met het `click` Event, daarom wordt aangeraden eerder een mouseup te gebruiken om een klik waar te nemen.<br>De eigenschappen `target` en `currentTarget` wijzen dan respectievelijk naar het object waarop het Event afgevuurd werd (klikken of een ander Event) en het object die het Event momenteel verwerkt (waar dus de Event handler aan gekoppeld is en die een EventListener heeft) |
| hover | `mouseover`, `mouseout` | naast `target` en `currentTarget` is ook hier `relatedTarget` van belang |
| dubbelklik | niet in DOM, wel in >js1.2: `dblclick` | Gebruik nooit `clik` en `dblclick` op dezelfde objecten vermits er steeds een `click` voor een `dblclick` plaats heeft. |
| slepen | `mousedown`, `mousemove`, `mouseup` | Slepen ('dragging') is een complexe handeling waarbij je eerst moet een `mousedown` opvangen, gevolgd door een `mousemove`. Tijdens deze laatste moet je het object bewegen (vraagt veel processorcapaciteit) om het tenslotte bij een `mouseup` te laten staan. |

### 11.8.2 Muisknoppen

Op welke knop geklikt wordt kan gedetecteerd worden door de eigenschap van `which` (Moz,FF,N), `button` (IE en W3C) van het Event. De chaos is hier compleet: De waarden van deze eigenschappen zijn ten eerste verschillend en zowel FF als Opera ondersteunen ook `button` enkel voor de rechtermuisknop...
Gelukkig heb je voor de meeste scripts enkel de linkermuisknop nodig. Er wordt wel aangeraden ze na te gaan op het `mousedown/up` event en niet op `click`.

| muisknop | Moz,FF,N | IE | W3C |
|---|---|---|---|
| links | 1 | 1 | 0 |
| midden(wiel) | 2 | 4 | 1 |
| rechts | 3 | 2 | 2 |

Dan is er ook nog de proprietary eventListener `oncontextmenu` die zowel door Mozilla, IE als Opera ondersteund worden. Deze door de browsers ingebouwde event luistert infeite naar een `click` op de rechtermuisknop. Door deze een returnwaarde `false` te geven kan je deze functionaliteit afzetten, maar niet volledig in Opera.

### 11.8.3 Pointer coördinaten

Hier is de situatie beter: de W3C eigenschappen `clientX` en `clientY` van het mouse event geven de coördinaten relatief t.o.v. het window. De eigenschappen `screenX` en `screenY` geven die positie t.o.v. het scherm.

## *11.9 Key Events*

De `keydown`, `keypress` en `keyup` events vuren af als een gebruiker een toets gebruikt.

| Event | beschrijving |
|---|---|
| `keydown` | Vuurt als een toets ingedrukt of ingehouden wordt. Herhaalt zolang de toets ingedrukt blijft |
| `keypress` | Vuurt af als een karakter ingevoerd wordt. Herhaalt zolang de toets ingedrukt blijft |
| `keyup` | Vuurt als de toets losgelaten wordt, nadat de default action uitgevoerd is |

De browsermodellen zijn totaal verschillend: relevante eigenschappen voor deze Events zijn:

| property | browser | beschrijving |
|---|---|---|
| `keyCode` | | voor `keypress`: returnt de Unicode waarde van een toets.<br>Gebruik `String.fromCharCode()` om het oorspronkelijke karakter terug te krijgen<br>voor `keydown` en `keyup`: returnt de key waarde van de toets (eender welke)<br>Mozilla stelt `charCode` en niet `keyCode` in voor een karaktertoets op `keypress` |
| `altKey, ctrlKey,shiftKey` | | returnen een *boolean* waarde om vast te stellen of de Alt, Ctrl of Shift toets werden ingedrukt samen met een andere toets |
| `charCode` | enkel Mozilla | returnt de *Unicode waarde* van een karakter toets. |

| | | |
|---|---|---|
| | | Gebruik `String.fromCharCode()` om het oorspronkelijke karakter terug te krijgen |
| `which` | enkel Mozilla | bevat steeds dezelfde waarde als `charCode` of `keyCode`, welk ook ingevuld is |

Eén van de problemen die opduiken met scripting key Events zijn de key waarden: dit zijn geen ASCII of Unicode waarden en dus heeft een omzetting met `String.fromCharCode()` geen zin. Er bestaat geen goede manier om ze om te zetten.

Om die reden gebruik je best steeds `keypress` als Event Handler om toetsen te testen, zodat je Unicode waarden krijgt.

```
document.frm1.testvak1.onkeypress=luisterToets;
...
function luisterToets(e){

    if(!e) var e = window.event;
    var uCode;

    if(e.keyCode){uCode=e.keyCode}
    if(e.which){uCode=e.which}

    alert(String.fromCharCode(uCode));
}
```

Denk er aan dat deze Unicode geen karakter hoeft voor te stellen, het kan bijvoorbeeld ook een Enter zijn.
Ten tweede hebben sommige toetsen ook een default action in de browser (bijvoorbeeld F11): deze gaan gewoon door, tenzij je ze cancelled.

## *11.10 Touch events*

Met de komst van *touch screens* hebben we ook *Touch Events* nodig.

Dit betekent ook dat we nog in fase van volle ontwikkeling zijn.

De meeste browsers (Firefox, Chrome, Safari, Opera ) ondersteunen het *Touch Events Model* (http://www.w3.org/TR/touch-events).

Microsoft ontwierp een eigen *Pointer Events Model* voor IE10 (vroegere versies geen ondersteuning) (http://www.w3.org/Submission/pointer-events/ ) dat momenteel besproken wordt op het W3C. Het heeft het voordeel dat er geen onderscheid meer gemaakt wordt in "pointing devices": er is maar één soort *pointing event* voor muis, pen, vinger en andere.

Hieronder volgt een uitleg over de Interface van het Touch Events Model.

***Touch point:***

Als je een oppervlak aanraakt met vinger of pen wordt een *touch point* aangemaakt. Je kan ook drie vingers tegelijk gebruiken: dan heb je 3 *touch points*.

Elk *touch point* heeft volgende eigenschappen:

| Property | type | beschrijving |
|---|---|---|
| `clientX` | `long`, readonly | De horizontale coördinaat relatief tov de *viewport* in pixels, zonder scroll offset |
| `clientY` | `long`, readonly | De vertikale coördinaat relatief tov *viewport* in pixels, zonder scroll offset |
| `identifier` | `long`, readonly | Een uniek identificatie nummer voor elk touch point, start vanaf 0, dan 1, 2 , etc… |
| `pageX` | `long`, readonly | De horizontale coördinaat relatief tov de *viewport* in pixels, inclusief de scroll offset |
| `pageY` | `long`, readonly | De vertikale coördinaat relatief tov the *viewport* in pixels, inclusief de scroll offset |
| `screenX` | `long`, readonly | De horizontale coördinaat relatief tov het *scherm* in pixels |
| `screenY` | `long`, readonly | De vertikale coördinaat relatief tov het *scherm* in pixels |
| `target` | `EventTarget` | De EventTarget bij de start van het touch point |

Daarnaast bestaan er ook objecten van het type `touch list`:

***Touch list:***

Is een lijst van *touch points*. Een touch list heeft slechts één eigenschap `length` en één method `item()`.

***Touch events:***

Er zijn 4 verschillende *touch events*:

| Event | beschrijving |
|---|---|
| `touchstart` | bij het begin: punten worden aangemaakt |
| `touchend` | einde: punten worden verwijderd |
| `touchmove` | beweging: punten verplaatst |
| `touchcancel` | annulatie: punten worden verwijderd |

Elk van deze events heeft volgende eigenschappen:

| Property | type | beschrijving |
|---|---|---|
| **altKey** | boolean, readonly | true als de ALT key ingedrukt is tijdens het event, anders false |
| **changedTouches** | TouchList, readonly | een lijst van touches voor elk punt die bijdroeg aan het event.<br>Voor touchstart zijn dat de aanraakpunten bij het begin,<br>voor touchmove zijn dat de punten die verplaatst zijn,<br>voor touchend en touchcancel zijn dat punten die verwijderd werden |
| **ctrlKey** | boolean, readonly | true als de CTRL key ingedrukt is tijdens het event, anders false |
| **metaKey** | boolean, readonly | true als de META KEYMODIFIER geactiveerd is, tijdens het event, anders false |
| **shiftKey** | boolean, readonly | true als de SHIFT key ingedrukt is tijdens het event, anders false |
| **targetTouches** | TouchList, readonly | een lijst van touches voor elk punt bij touchstart EN die OP het target van het huidig event liggen.<br>Subset van touches |
| **touches** | TouchList, readonly | een lijst van **alle** touches die momenteel bestaan |

### Mixing mouse en Touch events

Om dezelfde pagina compatibel te houden op een desktop en op een touch device, vuren browsers een manegeling af van *mouse* en *touch* events.
De volgorde is meestal

`touchstart > [touchmove]+ > touchend >` tijdspanne `> mousemove > mousedown > mouseup > click`

tussen het vuren van deze events stopt de browser een korte tijdspanne om een dubbelklik/dubbeltap mogelijk te maken. Daardoor kan een web app een beetje sloom overkomen. Om dit te vermijden registreren we zelf onze combinatie, enkel voor `touchend` en `click`:

```
/* feature detection */
var clickEvent = ('ontouchstart' in window ? 'touchend' : 'click');
element.addEventListener(clickEvent, function() { ... });
```

# 12 JSON

JSON staat voor *JavaScript Object Notation* (*Douglas Crockford*).

JSON is een **eenvoudig dataformaat** bedoeld om gegevens uit te wisselen, dat gemakkelijk zowel door **mensen** als door **programma's** gelezen kan worden.

Omdat JSON afgeleid is van Javascript, is het voor een Javascript applicatie eenvoudig om de gegevens te lezen en te verwerken - veel makkelijker dan een XML bestand.

In vergelijking met XML is JSON veel lichter: er is geen nood aan ingewikkelde transformaties van XML space naar DOM space, het kent geen interne validatie en heeft geen mogelijkheid voor stylesheet output en opmaak.

JSON data worden vooral gebruikt door **Ajax** applicaties, wordt courant gebruikt door jQuery of Prototype, maar ook HTML5 applicaties maken veelvuldig gebruik van het JSON formaat.

## *12.1 Syntax*

JSON kan twee soorten structuren weergeven:

- een **collectie** *naam:waarde* paren, m.a.w. een **Javascript Object**
- een geordende **lijst van waarden**, m.a.w. een **Array**

Met andere woorden: een JSON pakketje dat doorgegeven wordt tussen twee applicaties, is ofwel een *JSON object* of een *JSON array*.

Zo kan er bijvoorbeeld nooit een echte string doorgegeven worden.

Enkele voorbeelden:

```
{
    "name": "Jack (\"Bee\") Nimble",
      "format": {
          "type":       "rect",
          "width":      1920,
          "height":     1080,
          "interlace":  false,
          "frame rate": 24
      }
  }
```

Deze JSON bestaat uit twee properties met een String en een ander object als waarde.

```
 [
  [0, -1, 0],
  [1, 0, 0],
  [0, 0, 1]
]
```

Deze JSON bestaat uit een Array met subarrays.

Witruimte wordt volledig genegeerd.

## 12.2 datatypes

Welke datatypes ondersteunt JSON? Uiteraard vinden we hier een grote gelijkenis met de JS datatypes, toch is het wat beperkter:

- `Number` (integer, real, or floating point)
- `String:` identiek aan het JS datatype maar steeds in dubbele aanhalingstekens, apostroffen zijn niet toegelaten.
  Karakters kunnen ge-escaped worden met \
- `Boolean`: `true` en `false`
- `Array` (identiek aan JS datatype)
- `Object` (identiek aan JS datatype)
- `null`

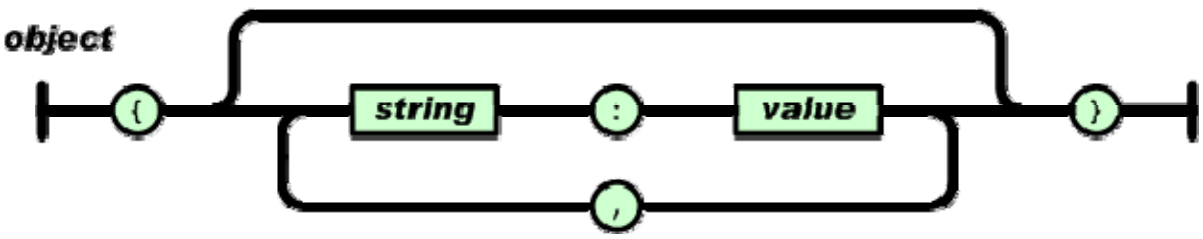
Enkele JS types zoals `undefined` en `NaN` zijn hier afwezig om compatibel te blijven met andere programmeertalen.

**Alle combinaties van deze types zijn toegestaan!**

Hieronder vind je JSON structuren voorgesteld als *'railroad diagrams'* (figuren uit *json.org*) :

### 12.2.1 object

Een **object** wordt voorgesteld in **accolades** als een collectie *naam:waarde* paren (net als JS eigenschappen) gescheiden door een **komma's**.




De naam MOET ook in aanhalingstekens! geen identifier zoals in JS.
De naam mag spaties bevatten of mag zelf een lege string zijn (of dat aan te raden is voor de verwerking is een andere zaak...)

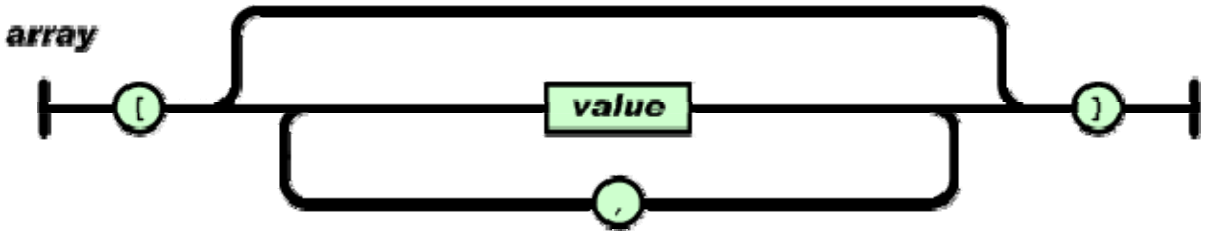
Een voorbeeld:

```
{"voor naam":"Jan",    "familienaam"  :  "Vandorpe"}
```

Dit lijkt erg op een JS object literal, het enige verschil is dat ook de eigenschapsnaam in aanhalingstekens staat.

### 12.2.2 array

Een **array** wordt voorgesteld in **vierkante haakjes** als een collectie waarden gescheiden door een **komma's**.

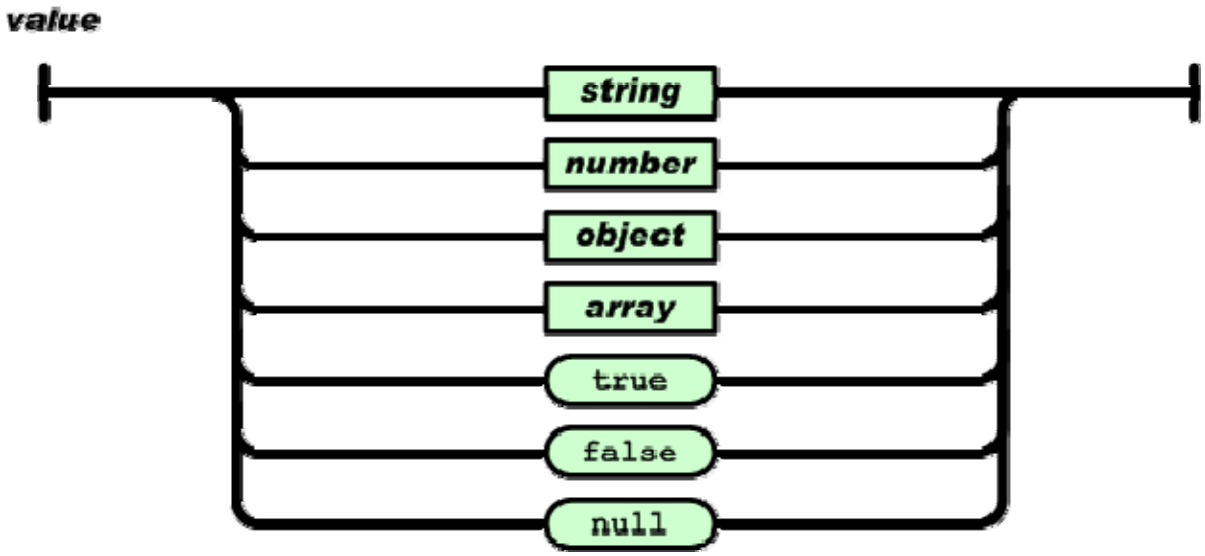



Een voorbeeld:

```
["koekoek","33",{"voornaam":"Jan"},false, 6, null, ["nog","een","array"],"en
  t'vogeltje zei \"koekoek\""]
```

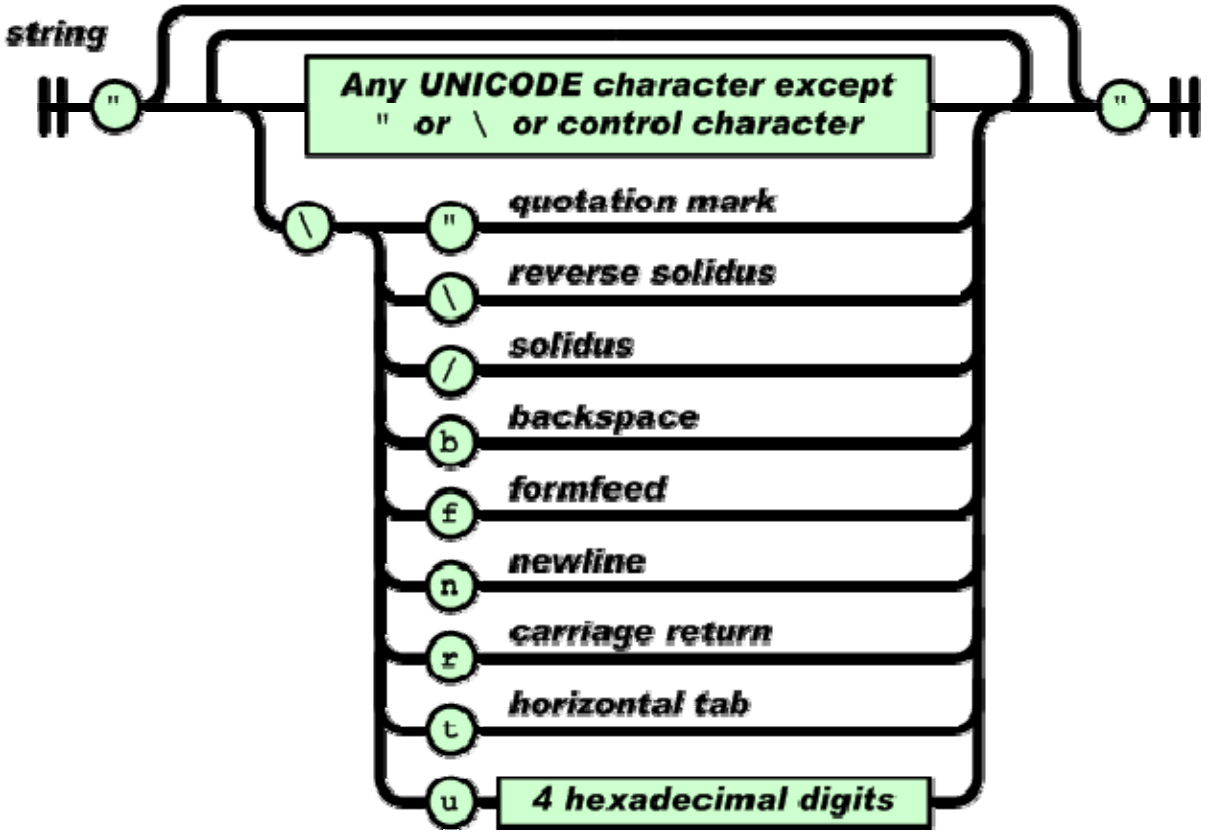
### 12.2.3 waarde

Een **waarde** kan een `String` zijn, een `Number`, een `Boolean`, een `Object`, een `Array` of `null` . Ze kunnen genest worden.



### 12.2.4 string

Een **string** staat steeds **in dubbele aanhalingstekens**. Het bevat *geen* tot *meerdere* unicode karakters. *Backslash* karakters kunnen gebruikt worden om te *escapen*. Een aantal speciale *backslash sequences* zoals een linefeed \n kunnen gebruikt worden.




Een voorbeeld:

```
{"lege string":"","quotes": "&#34; \u0022 %22 0x22 034 &#x22;
  \/\\\"\uCAFE\uBABE\uAB98\uFCDE\ubcda\uef4A\b\f\n\r\t`1~!@#$%^&*()_+-
  =[]{}|;:',./<>?"}
```

### 12.2.5 number

Een **number** is een decimaal getal , niet in aanhalingstekens. Gebruik de punt . als decimaal teken. e of E voor exponenten.

Een voorbeeld:

```
{"integer":1234567890,"real":-9876.543210,"e":0.123456789e-
  12,"E":1.234567890E+34,"":23456789012E66,"zero":0}
```

Het JSON number type kent geen octaal of hexadecimaal getaltype.

### 12.2.6 welke datatypes niet?

JSON ondersteunt de volgende datatypes niet:

- `undefined`
- `NaN`
- `Date`

properties met een waarde `undefined` wordt genegeerd: je vind de property niet terug in het JSON object.

Een `NaN` waarde wordt geconverteerd naar `null`.

Javascript heeft geen eigen `date` datatype, het gebruikt een `Date` object. Om datums om te zetten naar JSON, gebruik een `String` met het ISO8601 formaat (YYY-MM-DD), bijvoorbeeld

```
"datum": "2012-02-29"
```

Om het nadien weer te gebruiken als datum, plaats de string als argument in een nieuw Date object.

Een ander mogelijkheid is het date-time number op te slaan.

## *12.3 van JSON naar Javascript en omgekeerd*

Een JSON object is in essentie een `String`.

Om een javascript `Array` of `Object` in een JSON String om te zetten wordt het ge-*serialized*: geschreven als een letterlijk `Array` of `Object` waarvan alle identifiers in aanhalingstekens geplaatst worden.

Bijvoorbeeld: we starten met een JS object:

```
var oPersoon = {
  naam:"Jean Smits",
  beroepen:["coördinator","psycholoog","voorvechter","verpleger"],
  leeftijd: 56
}
```

omgezet naar een JSON object:

```
var jsonPersoon = {
  "naam":"Jean Smits",
  "beroepen":["coördinator","psycholoog","voorvechter","verpleger"],
  "leeftijd": 56
}
```

Een zuiver `Array` ondergaat geen veranderingen:

```
var aaCoordinaten      = [[55.3124,3.8801],[53.4455, 4.8877]]
var jsonCoordinaten    = [[55.3124,3.8801],[53.4455, 4.8877]]
```

Sommige van deze voorbeelden kan je natuurlijk manueel schrijven, maar als er speciale karakters aan te pas komen is het beter de omzetting te doen via een library.

De meest moderne browsers hebben ***native JSON support***, m.a.w. ze hebben een ingebouwde object `JSON` die de omzetting kan doen. Deze heeft slechts twee methods:

- `JSON.stringify()` om een Array/Object naar een JSON object te converteren
- `JSON.parse()` om een JSON object terug naar een Javascript structuur om te zetten

Voorbeelden:

```
var oPersoon    = JSON.parse(jsonPersoon);
var jsonPersoon = JSON.stringify(oPersoon);
```

**Syntax:**

```
jsonString = JSON.stringify(value [, replacer [, space]])
```

De method `stringify` zet een object of array om naar een jsonstring en heeft de volgende argumenten:

- `value`: verplicht, een Javascript object of array
- `replacer`: optioneel, een functie of een array van properties die moeten geconverteerd worden. Andere poperties worden genegeerd.
- `space`: optioneel, een karakter die zal gebruikt worden om witruimet te creëren in het eindresultaat, of een gatl met het aantal spaties dat hiervoor gebruikt zal worden.

```
jsObject = JSON.parse(value, reviver)
```

De method `parse` zet een jsonstring terug om naar een object.

- de `value` is de jsonstring die omgezet wordt naar een object of een array
- het optionele argument `reviver` is een functie waarmee je selectief kunt omvormen: het leest alle key's en values zodat je er een filter mee kan bouwen voor het te maken Javascript object

Het is altijd aan te raden aan *feature detection* te doen – het is niet gegarandeerd dat de browser van de gebruiker *native support* heeft.

Maar FF3.5+, alle Webkit browsers(Chrome, Safari), IE 8+, Opera 10.5+ hebben *native JSON support*. Ook de populaire Javascript libraries (jQuery, …) maken hiervan gebruik.

Indien geen native support aanwezig is, kan je gebruik maken van één van de speciale JSON libs zoals *json2.js* (https://github.com/douglascrockford/JSON-js).

Ook de Javascript functie `eval()` is in staat een JSON object te interpreteren, maar dat wordt **sterk afgeraden** wegens *security issues*.

JSONLint is een online json validator: http://jsonlint.com/

# 13 XML Primer

## 13.1 Introductie

Deze “*Primer*” is een introductie in XML voor Javascript en XHTML op "*Need to know*" basis. Het is voor deze cursus nodig dat je de basisprincipes van XML kent en voor sommige onderwerpen zoals Ajax is het ook nodig dat je de structuur van een XML document begrijpt.

## 13.2 Wat is XML?

**Extensible Markup Language** (**XML**) is een **standaard** van het World Wide Web Consortium (**W3C**) voor de syntax van de taal waarmee men gestructureerde gegevens kan weergeven als gewone tekst.

Met XML is dus **een vorm van gegevensopslag**, net als een ander databanksysteem.

XML lijkt erg op HTML, het ziet er uit als *free-style* HTML! Dat komt omdat beide talen van dezelfde voorvader afstammen: SGML.

XHTML is een standaard van HTML waarop de strikte XML-regels toegepast worden.

Er zijn verschillende middelen (=talen) waarmee je XML kunt bevragen en bewerken.

Zo is er bijvoorbeeld XPATH, een expressietaal specifiek voor XML. XPATH is voor XML een beetje hetzelfde als wat SQL is voor relationele databanken.

Voor ons is DOM van veel groter belang: het is een voorstelling van het XML document in de vorm van een boomstructuur, de *document tree*. DOM stamt dus oorspronkelijk uit de XML wereld, maar Javascript gebruikt het vooral om HTML te manipuleren.

Een **voorbeeld** maakt veel duidelijk.

In ons opleidingscentrum hebben we een bibliotheek. Deze boeken worden als naslagwerk of als cursus gebruikt. We willen nu een systeem opzetten waarbij we al onze boeken snel en eenvoudig kunnen raadplegen en deze informatie in verschillende vormen weergeven. Daarvoor slaan we de gegevens op in een XML bestand.

```
<?xml version="1.0" encoding="UTF-8"?>
<Boeken>
  <Boek boeknr="405" isbn="1590590104">
    <Titel>Database Programming with C#</Titel>
    <Schrijver>Thomsen</Schrijver>
    <Uitgever>Apress</Uitgever>
    <Curriculums>
      <Curriculum nr="38">
        <Naam>.NET ontwikkelaar met C#</Naam>
        <Beroepencode>750412</Beroepencode>
      </Curriculum>
    </Curriculums>
```

```
    <Onderwerpen>
      <Onderwerp nr="11">C#</Onderwerp>
    </Onderwerpen>
  </Boek>
  <Boek boeknr="404" isbn="0672322358">
…
  </Boek>
</Boeken>
```

### 13.2.1 Toepassing:

XML wordt voornamelijk gebruikt voor:

- **doorgeven van gegevens tussen applicaties**

  XML is vooral populair als medium voor het doorgeven van gegevens tussen computerprogramma's, niet zozeer als zuivere databank.

  Het is efficiënter om massa's productgegevens bij te houden in binaire vorm, bijvoorbeeld in een Oracle, MSSql of MySQL database, maar als een ander programma zoals een web-applicatie, snel enkele van die gegevens nodig heeft, dan kan de RDBS die doorgeven in XML vorm, die licht is en universeel begrepen wordt

- **opslaan van instellingen**

  De meeste moderne programma's slaan voorkeurinstellingen van zowel documenten als van het programma op in een XML vorm, denk maar aan het Open Office formaat, of de `docx` van Microsoft Office

### 13.2.2 Voordelen en nadelen

Waarom deze gegevens niet opslaan in binaire vorm?

**Voordelen** van XML

- Het **tekstformaat** maakt het platformonafhankelijk: Windows, Linux, Mac, …
- Het is **leesbaar** zowel voor mensen als voor machines
- Het ondersteunt **Unicode**, zodat talen en karakters geen probleem vormen
- Het is **zelf documenterend**: de tags verklaren wat er in zit
- De strikte syntax maakt het makkelijk **controleerbaar** voor andere programmeertalen
- De **hiërarchische structuur** is bruikbaar voor bijna alle soorten gegevens: er kan gemakkelijk een boomstructuur mee opgebouwd worden: DOM

**Nadelen**:

- De strikte syntax en verbositeit zijn overbodig bij binaire opslag
- andere tekstopslag formaten (vb. JSON) , zijn veel compacter dan XML
- Geen data types: XML kent geen integer, string, boolean, die moeten uit de inhoud afgeleid worden
- Niet-hiërarchische relaties zijn een probleem om aan te geven

## 13.3 De regels

Een XML bestand moet ***Well-Formed*** zijn. Dat betekent dat het aan een setje regels moet voldoen. We overlopen hier de voornaamste regels.

Een *parser*, dat is een programma die XML kan interpreteren, kan de *Well-formednes* van een document controleren en eventueel weigeren het verder te verwerken. Alle recente browsers hebben een XML-parser ingebouwd.

### 13.3.1 Elementen

In XML maak je je eigen elementen. Een **element** bestaat uit een ***tag*** en zijn inhoud:

```
<boek>Mijn dagboek</boek>
<Boek><Titel>Database Programming with C#></Titel></Boek>
<br />
```

- Een tag moet altijd **afgesloten** worden: ofwel door een eindtag, ofwel – bij leeg element – door een slash aan het einde van die tag
- een **elementnaam** mag niet met een getal of een speciaal teken beginnen en mag geen spaties bevatten.
  *Fout*: `<123boek>,<#boek>,<mijn boek>`.
  *Goed:* `<boek123>,<boek#>,<mijn_boek>`.
- aangezien XML **Hoofdlettergevoelig** is zijn `<Boek>, <boek>` en `<BOEK>`, drie verschillende elementen.
  De *case* van de eindtag moet ook overeenkomen met deze van de begintag
- De **inhoud** van een element kan
  - leeg zijn (EMPTY element)
  - tekst bevatten
  - andere elementen bevatten
- elementen moeten **correct genest** zijn.
  *Fout:*
  `<Boek><Titel>Database Programming with C#</Boek></Titel>`
  *Goed:*
  `<Boek><Titel>Database Programming with C#></Titel></Boek>`

### 13.3.2 Attributen

Een element kan één of meerdere attributen bevatten:

```
<Boek boeknr="405" isbn="1590590104">
        …
   <Curriculums>
     <Curriculum nr="38">
       <Naam>.NET ontwikkelaar met C#</Naam>
       <Beroepencode>750412</Beroepencode>
```

```
    </Curriculum>
  </Curriculums>
  <Onderwerpen>
    <Onderwerp nr="11">C#</Onderwerp>
  </Onderwerpen>
</Boek>
```

- Attributen volgen dezelfde naamgevingregels als elementen: niet beginnen met een getal of een speciaal karakter
- een attribuut moet **uniek** zijn binnen dat element, er kunnen bv .geen twee `type` attributen binnen hetzelfde element zijn
- een attribuut kan ook leeg zijn, dan wordt hij weergegeven door een lege string , bv. `<Onderwerp nr="">`
- de volgorde van de attributen speelt normaal geen rol

Je bent natuurlijk vertrouwd met de attributen van XHTML die ook deze regels beantwoorden

### 13.3.3 het XML document

Het XML document is een tekstdocument opgeslagen in Unicode indeling.

Dat heeft het enorme voordeel dat je karakters van eender welke taal kunt gebruiken in het document. Het volgende voorbeeld is een well-formed document:

```
<?xml version="1.0" encoding="UTF-8"?>
<烏語>Китайська мова</烏語>
```

De elementnaam is in het Chinees, de inhoud in het Cyrillisch.

#### 13.3.3.1 de XML declaratie

De XML declaratie staat als eerste regel in het document, maar is niet verplicht:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Deze declaratie duidt de XML versie aan (er is voorlopig maar 1 versie) en de karakterset waarin het document opgeslagen wordt.

Zowel UTF-8 als UTF-16 zijn Unicode indelingen.

#### 13.3.3.2 commentaar

Je kan overal waar je commentaar zetten. Commentaar wordt niet geïnterpreteerd:

```
<-- dit is commentaar -->
```

Commentaar tags zijn dezelfde als in HTML.

#### 13.3.3.3 witruimte

Spaties, tabs en regeleinden worden door de XML bewaard. Als er dus meerdere spaties tussen woorden staan dan behoudt XML die.

Browsers daarentegen doen hun best die witruimte te negeren en te “normaliseren”. zo zal een browser de spaties tussen de woorden herleiden to één.
Neem bijvoorbeeld het volgende element:

```
<boek>
<Naam>   .NET
  ontwikkelaar
  met C#
  </Naam>
</boek
```

dan zal een zuivere xmlparser die niet wijzigen terwijl Internet Explorer er

```
<boek>
<Naam>.NET ontwikkelaar met C#</Naam>
</boek
```

van maakt.

#### 13.3.3.4 Eén root element

Een well-formed XML doc kan slechts één rootelement bevatten. Het is aan jou om die te kiezen. Voor XHTML is `html` het root-element.

## *13.4 Uw XML == mijn XML ?*

XML laat je dus je elementen kiezen, zodat je een eigen taal kunt maken over je gegevens. Zo kan je bijvoorbeeld een XML “taal” maken voor personeelsdata, of om de gegevens van dierensoorten en hun afstamming op te slaan, of eender wat eigenlijk.
De vrijheid van een XML document kan een probleem vormen als je je gegevens wil **delen met anderen:** dan moet je twee problemen aanpakken:

- een uniforme “woordenschat” en een structuur afspreken zodat iedereen dezelfde gebruikt
- hoe ervoor zorgen dat jouw elementen niet in conflict komen met gelijk genoemde elementen in een XML bestand van een andere “woordenschat”?

Eenmaal er overeenstemming bereikt is over elementennamen en hun hiërarchische structuur, leg je deze woordenschat vast in een **standaard** of **xml vocabulaire.**
Conflicten tussen verschillende *vocabulaires* lossen we op door gebruik te maken van **Namespaces**.

### 13.4.1 XML standaarden

Elke afgeleide taal kan worden vastgelegd in **standaarden en vocabulaires**.
Enkele voorbeelden van de vele honderden specificaties zijn:

- XML zelf, heeft zijn standaard op het W3C
- MATHML, een XML taal voor wiskunde
- Open Office XML, een standaard voor open office applicaties

- Travel XML, een standaard voor reisorganisaties
- RSS, een XML taal voor feeds (syndication)
- XHTML, een XML versie van HTML

Die standaard bestaat uit een document waarin de regels gespecificeerd zijn.
Aan de hand van de standaard kan een document **gevalideerd** worden.
Dat document is ofwel

- een **DTD** (**Document Type Definition**) (`.dtd` extensie) of
- een **XML Schema** (`.xsd` extensie).

Beide documenten zijn leesbaar zowel door mensen als door machines.

- Een **DTD** is een wat oudere manier van valideren, vooral gebruikt voor verwerking van teksten.
  HTML en XHTML worden nog steeds met een DTD gevalideerd. Dat merk je aan de DOCTYPE declaratie bovenaan een HTML document

  Een DTD is zelf geen XML en heeft daardoor beperkingen: het kan niet met *Namespaces* overweg en is erg zwak op gebied van *data types*
- Een **Schema** is een XML document op zich en heeft veel meer mogelijkheden

Het is niet de bedoeling de gedetailleerde werking van deze documenten hier te bespreken, we geven enkele voorbeelden en je moet je bewust zijn van hun bestaan en hun doel. Voor een gedetailleerde uitleg verwijzen we je naar de XML cursussen.

## *13.5 Validatie*

Een applicatie kan dan het XML document valideren aan de hand van de DTD of het Schema, en bepalen of het **valid** is.
Twee voorbeelden:

### 13.5.1 XHTML wordt gevalideerd met een DTD

XHTML past de regels van XML toe op HTML, zowel op gebied van *Well-Formedness* als op gebied van validatie.
Zo ben je in XHTML verplicht een tag af te sluiten: `<li> </li>`, anders is het niet *Well-formed*.
Je kan ook in een `ul` element enkel een `li` element zetten, niets anders. Dat is een *validatieregel* die je kunt vinden in het DTD.
Het DTD van een HTML document vind je in de DOCTYPE declaratie:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
…
```

Daarin staat ergens de volgende lijnen:

```
…
<!ELEMENT ul (li)+>
```

```
<!ATTLIST ul
  %attrs;
  type        %ULStyle;      #IMPLIED
  compact     (compact)      #IMPLIED
  >
…
```

wat zoveel betekent als: een `ul` element kan enkel en moet minstens één `li` element bevatten en kan enkel de attributen `type` en `compact` hebben, naast de core attributes.

Browsers hebben dat dtd op het W3C niet echt nodig, ze hebben er een ingebouwd versie van.

### 13.5.2 Een schema voor boeken.xml

Een *Schema* wordt opgeslagen als een .XSD bestand. Voor ons voorbeeldbestand boeken kunnen we zelf een *Schema* maken *(3_12_XSD_schema.xsd):*

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="boek">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="boek_id" type="xs:integer"/>
                <xs:element name="titel" type="xs:string"/>
                <xs:element name="schrijver" type="xs:string"/>
                <xs:element name="uitgever" type="xs:string"/>
                <xs:element name="isbn_nr" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Je bemerkt dat een schema zelf XML is. Dit schema bepaalt dat:

- er een element *boek* kan bestaan
- dat, in volgorde, de elementen *boek_id, titel, schrijver, uitgever* en *isbn_nr* moet bevatten
- voor elk van die geneste elementen wordt een primitief datatype gegeven

Bemerk ook de *NameSpace* voor XML Schema: die kan in het Schema zelf staan, of in het XML document.

Onderstaand voorbeeld zal dus geldig valideren ten opzichte van ons schema:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<boek xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="3_12_XSD_schema.xsd">
    <boek_id>1</boek_id>
    <titel>
    Linux in A Nutshell : A Desktop Quick Reference (3rd Edition)
    </titel>
    <schrijver>
    Ellen Siever (Editor), Jessica P. Hekman, Stephen Figgins,
Stephen Spainhour
    </schrijver>
```

```
    <uitgever>O' Reilly</uitgever>
    <isbn_nr>2</isbn_nr>
</boek>
```

Elke afwijking van dit schema zal door de *parser* als een ongeldige validatie beschouwd worden.

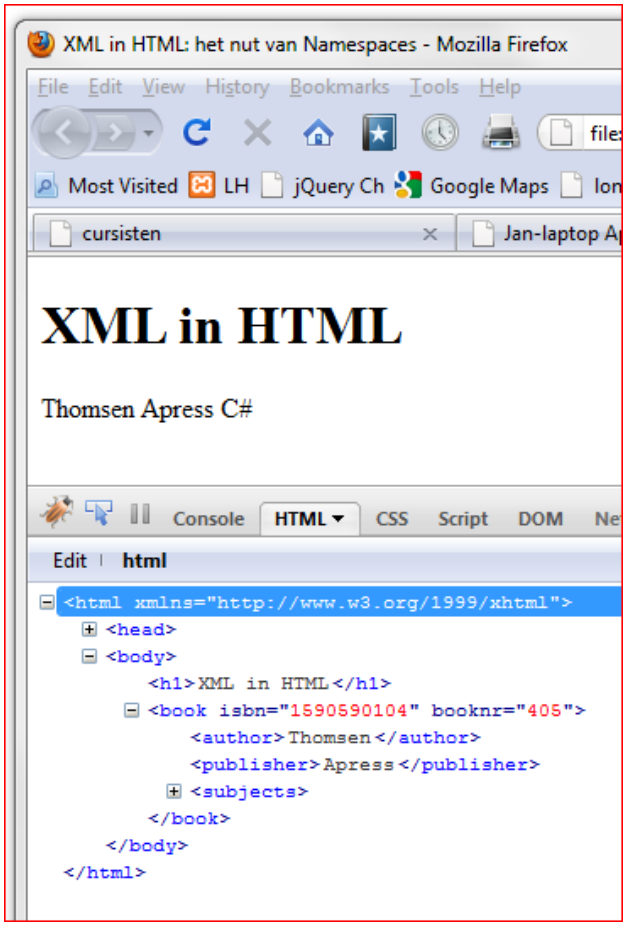
*Verwar **Well-Formed** niet met **Valid:***
*- een well-formed document gebruikt de correcte syntax*
*- een valid document is well-formed en volgt ook de regels van zijn DTD of Schema*

## 13.6 het nut van Namespaces

Veronderstel dat je een deel van *book.xml* (engelse versie) in een HTML bestand wil verwerken:

```
 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>XML in HTML: het nut van Namespaces</title>
</head>
<body>
<h1>XML in HTML</h1>
 <book booknr="405" isbn="1590590104">
    <title>Database Programming with C#</title>
    <author>Thomsen</author>
    <publisher>Apress</Uitgever>
    <subjects>
      <subject nr="11">C#</subject>
    </subjects>
  </book>
</body>
</html>
```

Als we dit bekijken in *FireFox* en *FireBug* dan zien we:




dan bemerk je dat het `<title>` element van `<book>` verdwenen is. Tijdens de opbouw van de DOM komt de browser tot de vaststelling dat er maar één `<title>` element kan zijn in een document.

Dit is een conflict dat opgelost kan worden door ***Namespaces*** te gebruiken.

Een ***Namespace*** verzamelt een aantal elementen in één groep, zodat bijvoorbeeld de browser of de applicatie weet dat hij ze verschillend moet interpreteren.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>XML in HTML: het nut van Namespaces</title>
</head>
<body>
<h1>XML in HTML</h1>
<bk:book xmlns:bk="http://www.vdab.be/bookvoorbeelden/versie1"
         booknr="405" isbn="1590590104" >
  <bk:title>Database Programming with C#</bk:title>
  <bk:author>Thomsen</bk:author>
  <bk:publisher>Apress</bk:Uitgever>
  <bk:subjects>
    <bk:subject nr="11">C#</bk:subject>
```

```
  </bk:subjects>
</bk:book>
</body>
</html>
```

Nu worden alle elementen in het XML fragment getoond door de browser.

Hier hebben we in het element `book` een eigen *Namespace* ingesteld en alle elementen die eronder vallen aangeduid met de prefix `bk:`.
Bemerk ook dat de *Namespace* van het `html` element aangegeven is: `xmlns="http://www.w3.org/1999/xhtml"`, hier automatisch aangemaakt door Dreamweaver.

Een *Namespace* kan je in een element aangeven met het **attribuut** `xmlns`, eventueel gevolgd door een **prefix**, en gelijkgesteld aan een **URL**.

**`xmlns:prefix=URL`**

- de prefix, `:bk`, kan je vrij kiezen
- het URL mag fictief zijn en moet niet noodzakelijk naar één of ander document verwijzen
- een prefix mag je achterwege laten: dat is de ***default NameSpace,*** die alle elementen groepeert die geen prefix hebben.
  Hier is `xhtml` de *default NameSpace*

Voor sommige standaarden gebruikt men bij consensus een welbepaalde prefix en URL aangeven, bijvoorbeeld:

```
<?xml version="1.0" encoding="UTF-8"?>
<boek xmlns="http://www.vdab.be/XML"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.vdab.be/XML 3_14_met_namespace.xsd">
  <boek_id>1</boek_id>
  <titel>
  Linux in A Nutshell : A Desktop Quick Reference (3rd Edition)
  </titel>
  <schrijver>
  Ellen Siever (Editor), Jessica P. Hekman, Stephen Figgins, Stephen Spainhour
  </schrijver>
  <uitgever>O' Reilly</uitgever>
  <isbn_nr>2</isbn_nr>
</boek>
```

In dit voorbeeld:

- is `xmlns=http://www.vdab.be/XML` de *default NameSpace*

- is `xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"` de NameSpace voor XML *Schema*
- wordt met `xsi:schemaLocation=http://www.vdab.be/XML 3_14_met_namespace.xsd` de plaats van het schema aangegeven

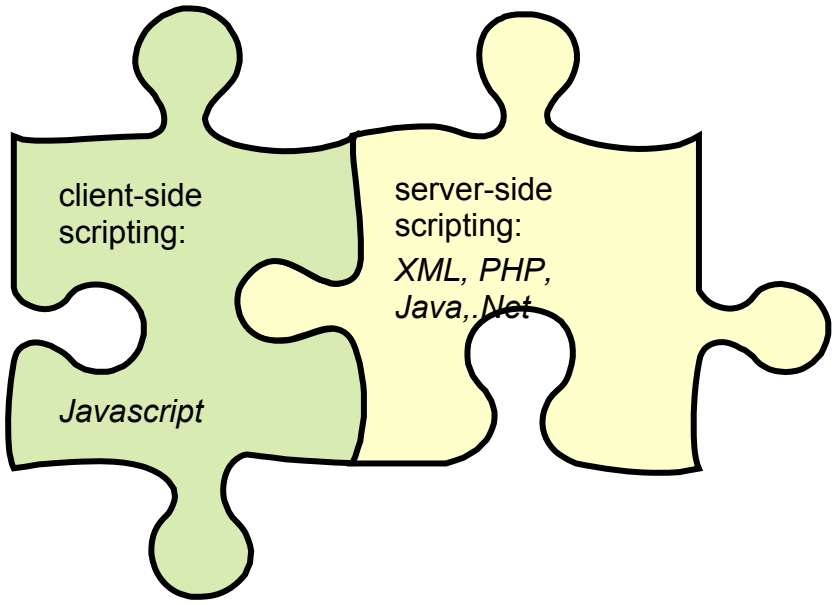
We gebruiken hier een Schema (zie hieronder), maar dat Schema zal niet werken als er niet verwezen wordt naar zijn eigen NameSpace.

# 14 Ajax

*Ajax komt enkel aan bod in de "Javascript Advanced" module. In de jQuery cursus wordt ook een Ajax toepassing gebruikt, dus kan deze theorie nuttig zijn.*

**Ajax** (***Asynchronous JavaScript and XML***) is de techniek waarbij Javascript gegevens asynchroon uitwisselt met de server. Dat gaat via het `XMLHttpRequest` object. Die gegevens kunnen in verschillende vormen uitgewisseld worden: XML, JSON, HTML en zelf gewone tekstbestanden.

Voor een Ajax toepassing heb je dus twee zijden nodig: een ***client-side*** programma dat gegevens vraagt en die nadien verwerkt en een ***server-side*** script dat luistert naar een Ajax call en de gevraagde gegevens levert.




Ajax is dus geen programmeertaal; het is een techniek die minstens twee programma's combineert: de client-side kant op de computer van de gebruiker en de server-side kant op de webserver.

## *14.1 Een voorbeeld:*

Laat ons een "klassieke" webpagina vergelijken met een "Ajax versie".

Veronderstel een *profiel* pagina, zoals je vindt op *social networking* sites zoals Facebook, Flickr, etc…

*Je wilt je email-adres wijzigen:*

**De klassieke versie:**

- je klikt op de knop "*profiel wijzigen*"
- je komt op een formulierpagina terecht met alle gegevens in tekstveldjes
- je wijzigt en klikt op "*wijzigingen opslaan*"
- je krijgt opnieuw je profielpagina te zien, dit keer met het gewijzigd adres

**De Ajax versie:**

- je klikt op je emailadres in je profielpagina
- de tekst verandert in een tekstveldje, er verschijnt een "*Opslaan*" en een "*Terugzetten*" knopje ernaast
- je wijzigt en klikt op "*Opslaan*"
- je emailadres is gewijzigd

**Hoe werkt de klassieke versie?**

voor de klassieke versie zijn minstens twee html pagina's en een serverside script nodig:

- *profiel.html*, de profielpagina zelf
- *profiel_wijzigen.html*, het formulier waarin je wijzigingen kunt aanbrengen
- *profiel_wijzigen.php*, het serverside script dat de wijzigingen in de database zet

voor de klassieke versie wordt via http de server meerdere malen gecontacteerd:

1. om de formulierpagina op te vragen
2. om de gewijzigde gegevens door te geven aan het serverside script (die de database contacteert)
3. om de gewijzigde profiel pagina opnieuw te tonen

**Hoe werkt de Ajax versie?**

voor de Ajax versie is slecht één html pagina en een serverside script nodig:

- *profiel.html*, de profielpagina zelf
- *ajax_profiel_wijzigen.php*, het serverside script dat de wijzigingen in de database zet

voor de Ajax versie wordt via xmlhttp de server slechts eenmaal gecontacteerd:

1. om de gewijzigde gegevens door te geven aan het serverside

Hieruit blijkt duidelijk dat er veel meer minder verkeer is client-server bij een Ajax applicatie, maar ook dat het serverside Ajax script specifiek gemaakt is om de Ajaxcall te ondersteunen.

Nog eens op een rijtje:

**klassiek**:

- synchrone communicatie via een `HttpRequest`
- een **volledig nieuwe pagina** wordt opgestuurd, de browser laadt de volledige HTML code weer in
- er moet gewacht worden op het resultaat van de call: de code is ***blocking***
- de gebruiker moet wachten tot de pagina ingeladen is en kan ondertussen niets doen (hij ziet een *'progress-bar'* in de statusbalk van zijn browser)

**Ajax**:

- asynchrone communicatie via een `XmlHttpRequest`:

- de server stuurt **enkel de gegevens** waarnaar gevraagd is, geen volledige pagina
- Javascript verwerkt de gegevens in de pagina
- De code is asynchroon, dus ***non-blocking***.
- de gebruiker moet niet wachten en kan ondertussen verder scrollen, kiezen

### 14.1.1 Voor- en nadelen van Ajax

Alle voordelen zijn voor de gebruiker, de ontwikkelaar krijgt de meeste nadelen…

Voordelen:

- een Ajax-pagina is **veel interactiever** voor de gebruiker, veel aangenamer om mee te werken. Hij kan zijn omgeving veel meer personaliseren en kan dat direct toepassen
- de veelheid aan interactiviteit kan ultiem leiden naar 1-pagina websites waar alle functionaliteit voorzien is
- Ajax applicaties zijn dikwijls onafhankelijke scripts die erg "*portable"* zijn: mits enkele kleine aanpassingen zijn ze op andere sites direct bruikbaar
- een Ajax verzoek gaat meestal snel en verbruikt veel minder bandbreedte

Nadelen

- moeilijker client-side scripting: Ajax scripts zijn complex. JS Libraries kunnen een oplossing bieden
- meestal moeten ook speciale serverside scripts ontworpen worden, die zijn echter niet noodzakelijk ingewikkelder dan een ander.
- de "*Same origin policy*" van de browsers is een rem op de ontwikkelingsmogelijkheden
- de "*Back*" knop van de browser brengt niet de vorige toestand terug, maar de vorige pagina, en dat brengt de gebruiker in verwarring
- moeilijk om "*state*" te bewaren: als de gebruiker de volgende keer naar de pagina terugkeert, verwacht hij zijn wijzigingen en aanpassingen terug te vinden. Dat kan opgelost worden door degelijk scripting
- het is niet altijd duidelijk voor de gebruiker dat er iets gewijzigd is op de pagina door één van zijn acties, bijvoorbeeld een keuze uit een lijst werkt andere lijsten bij: de ontwikkelaar moet elke wijziging klaar en helder aanduiden
- Ajax sites veronderstellen een continue en snelle internetverbinding, iets wat niet heeft

### 14.1.2 Websites

Ajax technologie is de belangrijkste steunpilaar van Web2 en Web3 websites. Zonder Ajax geen onmiddellijke interactiviteit. Als voorbeeld lijsten we hier enkele vooraanstaande websites die integraal steunen op Ajaxtechnologie:

- Google Maps: (maps.google.com): slepen van de kaart, plaatsen van markers, toevoegen van lagen, etc, etc..
- Flickr foto site: (www.flickr.com): opladen van foto's, online aanpassen van teksten, tags en titels
- Facebook: (www.facebook.com) online aanpassen van profielpagina

- Twitter (www.twitter.com) communicatie van korte berichtjes tussen kennissen via website en GSM
-

## 14.2 het XMLHttpRequest object

De manier waarop je een `XMLHttpRequest` object aanmaakt, verschilt tussen de browsers. Microsoft maakte tot IE6 gebruik van een `ActiveX` object, de huidige versies gebruiken nu ook het standaard `XMLHttpRequest` object. Enkele IE klonen gebruiken nog één van de `ActiveX` objecten.

Een cross-browser script gaat zo:

```
var xmlhttp;
if (window.XMLHttpRequest) {
        xmlhttp  = new XMLHttpRequest();
  }
else if (window.ActiveXObject) {
  try {
        xmlhttp  = new ActiveXObject("Msxml2.XMLHTTP");
  }
  catch (e){
        try {
                xmlhttp  = new ActiveXObject("Microsoft.XMLHTTP");
        }
        catch (e){}
  }
        }
  else {
        throw new Error("XMLHttpRequest niet ondersteund");
}
return xmlhttp;
```

Bespreking:

- we doen aan objectdetection, geen browserdetection
- Mozilla, Safari, en IE7+ hebben een `XMLHttpRequest` class waarvan we een nieuw object afleiden: hier de var *xmlhttp*
- zoniet proberen we één van de `ActiveXObject` die de browser kan hebben
- indien geen van deze lukt, kunnen we onze Ajax applicatie wel vergeten, dus werpen we een Error

Eenmaal dit object aangemaakt kunnen we gebruiken maken van zijn properties, methods en events.

Een overzicht

| Properties | omschrijving |
|---|---|
| `readyState` | integer die de huidige stand van verwerking aangeeft: |

| | |
|---|---|
| | 0 = unitialized<br>1 = open<br>2 = sent<br>3 = receiving<br>4 = loaded , het document is beschikbaar |
| `responseText` | string:<br>Tekstversie van de gegevens doorgestuurd door de server |
| `responseXML` | DOMdocument:<br>de doorgestuurde gegevens als DOM document |
| `status` | long.<br>numerische code returnt door de server die de status van de connectie toont. Beschikbaar zodra de `readyState` 3 is. Hier enkele van de vele codes:<br>200 = OK<br>204 = no content<br>400 = bad request<br>401 = unauthorized<br>403 = forbidden<br>404 = niet gevonden<br>50X = fout |
| `statusText` | string:<br>tekst geassocieerd met de statuscode |
| **methods** | **omschrijving** |
| `abort()` | stopt het huidige request |
| `getAllResponseHeaders()` | string. Returnt de volledige set headers: labels en values |
| `getResponseHeader(label)` | string. Haalt een specifieke header op |
| `open(method, url [,asyncFlag [,userName [,password]]])` | specificeert de `url` die opgehaald moet worden: deze kan relatief of absoluut zijn.<br>De `method` kan een waarde 'GET','POST', 'PUT', 'DELETE' of 'HEAD' hebben.<br>De `asyncFlag` is een `boolean` die indien `true` het script *asynchroon* uitvoert.<br>Met het `onreadystatechange` event weet je dan wanneer het klaar is.<br>Een eventuele `userName` en `password` kan meegegeven worden. |
| `overrideMimeType(mime-type)` | Een method die het *mime-type* van de doorgestuurde gegevens kan overschrijven. Sommige browsers eisen |

| | |
|---|---|
| | een mime-type *'text/xml'* |
| `send(content)` | stuurt het request door met een POST-bare string of DOM object |
| `setRequestHeader(label, value)` | Stelt een label/value paar in als header om verstuurt te worden |
| **Events** | **omschrijving** |
| `onreadystatechange` | *event handler* die afvuurt bij elke wijziging van de `readyState` |

Uit bovenstaande blijkt duidelijk dat je niet enkel een XML document kunt halen maar ook gewone tekst: dat laat ons toe ook eenvoudig delimited text binnen te halen maar ook een Javascript en JSON tekst.

Bedenk daar ook steeds bij dat er aan serverzijde ofwel een vast XMLbestand klaar moet staan ofwel een script die het gevraagde (xml, tekst, JSON) levert. In onze onderstaande voorbeelden gebeurt dat meestal met een simpel PHP script.

### 14.2.1 Een delimited tekst inladen

Aan de hand van een eenvoudig voorbeeld leren we de techniek beter kennen: de function *ajax()* hieronder kan bijvoorbeeld aan een knop gekoppeld worden.

```
var url ="produce_tekst.php" ;
function ajax() {
  var xmlhttp;
  if (window.XMLHttpRequest) {
     xmlhttp = new XMLHttpRequest();
 }
 else if(window.ActiveXObject) {
     xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
 }
  else {
     throw new Error("XMLHttpRequest not supported");
 }

if (xmlhttp) {
  xmlhttp.open("GET", url, true);
  xmlhttp.onreadystatechange = function(){
         if ((xmlhttp.readyState == 4) && (xmlhttp.status == 200 )) {
                maakLijst(xmlhttp.responseText);
         }
  };
  xmlhttp.send(null);
 }
}
```

Bespreking:

- eerst wordt een `XMLHttpRequest` object aangemaakt zoals eerder beschreven
- als de var *xmlhttp* niet `undefined` is, gaan we door
- met de method `open()` doen we een request
  - met de method "GET": gebruik hoofdletters, let op de aanhalingstekens (enkel of dubbel)
  - voor de `url` *produce_tekst.php* op de server
  - de derde parameter, met de waarde `true`, zorgt ervoor dat het request asynchroon verloopt
- merk op dat er op dit ogenblik nog niets verstuurd is, het request is enkel 'klaargemaakt' voor verzending
- `xmlhttp.onreadystatechange` is een **Eventhandler**: we zorgen ervoor dat we klaar zijn om het antwoord te verwerken als het binnenkomt
  - `onreadystatechange` bevat een anonieme functie die de `readyState` en de `status` van het *xmlhttp* object checkt:
  - indien `readyState` *loaded* is (4) en de `status` is *OK* (200) dan kunnen we ervan uitgaan dat het gevraagde correct ingeladen is
  - in dat geval voeren we de functie *maakLijst()* uit met de `responseText` inhoud van de *xmlhttp* object
- nu alles in gereedheid gebracht is voor ontvangst wordt het request verstuurd met `send()`. Het argument in de method – hier met waarde `null` – kan eventueel een *naam=waarde* paar zijn in geval van POST. zie verder.

Opmerking:

- de test

  ```
  if ((xmlhttp.readyState == 4) && (xmlhttp.status == 200 ))
  ```

  is in ons voorbeeld efficiënt maar kan eventueel opgesplitst worden in

  ```
  if(xmlhttp.readyState == 4){
        if(xmlhttp.status == 200 ){
              maakLijst(…)
        }
  }
  ```

  zodat je meer mogelijkheden hebt om te reageren op eventuele problemen. Noteer wel dat je de `status` enkel kunt controleren als de `readyState` minimaal 3 is.

De functie *maakLijst()* anticipeert een delimited string zoals *"Vandorpe Jan=js;Devos Inge=basis;Smits Jean=stripverhalen "* en toont die via DOM method in de pagina:

```
function maakLijst(tekst){
```

```
  // verwacht een gewone string delimited met puntcomma's
  var arrItems = tekst.split(';');
  var eUl = document.createElement('ul');
  // lus doorheen alle items van het array Items
        for (var i=0;i<arrItems.length;i++){
              var eLi = document.createElement('li');
              eLi.appendChild(document.createTextNode(arrItems[i]));
              eUl.appendChild(eLi);
        }
  demoDiv.appendChild(eUl);
}
```

Bespreking:

- het argument *tekst* die deze functie verwacht is dus de `responseText` eigenschap van het `XMLHttpRequest` object, een string dus
- deze wordt opgesplitst volgens de puntkomma's en er wordt een lijstje mee aangemaakt

### 14.2.2 argumenten doorgeven aan de server

Net zoals bij een 'gewone' HTTPRequest kan je ook met `XMLHttpRequest` gegevens meegeven aan de server zodat deze erop kan reageren. Zo zou je reisbureau-pagina de naam van het land kunnen meegeven zodat je de relevante hotellijst terugkrijgt.

De manier waarop dit gebeurt verschilt afhankelijk of je een GET of een POST gebruikt in de `XMLHttpRequest.open()` method.

Voor een GET gebruiken we één of meerdere *naam=waarde* paren die we achteraan de url hangen. Ons voorbeeld kan bijvoorbeeld een specifieke lijst opvragen:

In bovenstaande code moet enkel het url gewijzigd worden:

```
var url ="produce_tekst.php?welkeLijst=2" ;
…
xmlhttp.open("GET", url, true);
…
xmlhttp.send(null);
…
```

Het serverside script kan deze 'Querystring' lezen, analyseren en de gevraagde gegevens terugsturen.

Gebruik je een POST dan geven we de gegevens mee in de `send()` method waar het argument dan niet meer `null` is. We moeten in dat geval ook de *Content-Type* van de header wijzigen:

```
var url ="produce_tekst.php" ;
…
xmlhttp.open("POST", url, true);
xmlhttp.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
```

```
…
xmlhttp.send("welkeLijst=2");
…
```

Opmerking:

- je kunt de method `setRequestHeader()` enkel toepassen na de `open()` method

### 14.2.3 Een JSON object

Een JSON object ( zie relevante hoofdstuk) kan ook verwerkt worden met `responseText`.

Veronderstel dat een serverside script de volgende JSON string doorgeeft:

```
{'ras':'poedel','blaf':function(){return 'woef!'}}
```

dan kunnen we deze terug omzetten naar een object of een array met de `eval()` functie:

```
var strJSON = "{'ras':'poedel','blaf':function(){return 'woef!'}}";
var hond = eval('(' + strJSON + ')');

alert(hond.blaf());
```

Merk op dat een JSON object als `string` ontvangen wordt, en dus binnenkomt als "{....}". Daarom het extra gebruik van haakjes binnen de `eval()` functie.
Door de method *blaf()* uit te voeren bewijzen we het correcte bestaan van het object *hond*.

### 14.2.4 Een Javascript inladen

Ook volledige Javascripts kunnen doorgegeven worden met een Ajax call: dit is een vorm van *Javascript-on-demand* waarbij bepaalde script snippets enkel geladen worden als ze nodig zijn. Opnieuw gebruiken we `responseText`.

Veronderstel een serverside script dat een javascript als string doorgeeft:

```
alert('hello, boys & girls, mashup applicatie nummer 1 is nu online')
```

dan kan dit onmiddellijk uitgevoerd worden met de `eval()` functie:

```
var mashup1 = eval(js);
```

### 14.2.5 Een xml document inladen

Meestal wordt Ajax gebruikt om een XML document door te geven. Dit kan een vast xml bestand zijn, maar wordt meestal door de server *on-the-fly* gegenereerd (.Net, PHP, Java). Voor ons maakt dat weinig uit zolang het maar een geldig XML doc is waarvan we de structuur kennen.

In tegenstelling tot de vorige toepassingen gebruiken we nu `responseXML` om de verwerking te doen.

In dit voorbeeld laden we een XML bestand met usernames en verwerken die in een lijst.

```
  var url ="usernames.xml" ;
  var demoDiv; // de output div

function ajax() {
  var xmlhttp;
 if (window.XMLHttpRequest) {
      xmlhttp = new XMLHttpRequest();
 }
else if(window.ActiveXObject) {
      xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
}
else {
      throw new Error("XMLHttpRequest not supported");
}

if (xmlhttp) {
  xmlhttp.open("GET", url, true);
  xmlhttp.onreadystatechange = function(){
  if ((xmlhttp.readyState == 4)&&(xmlhttp.status == 200 )) {
        maakLijst(xmlhttp.responseXML);
  }
};
xmlhttp.send(null);
}
}
```

Bespreking:

- het enige verschil is dus dat de functie *maakLijst()* deze keer een `XMLDocument` object zal verwerken

De functie *maakLijst()* is nu specifiek aangepast om via DOM het bestand te verwerken:

```
function maakLijst(xmldomdoc){
  // verwacht een xmldomdocument van een gekende structuur
  var namen = xmldomdoc.getElementsByTagName('naam');
  var eUl = document.createElement('ul');
  for (var i=0;i<namen.length;i++){
        var n =namen[i].firstChild.nodeValue.toLowerCase();
```

```
            var eLi = document.createElement('li');
            eLi.appendChild(document.createTextNode(n));
            eUl.appendChild(eLi);
            }
  demoDiv.appendChild(eUl);
}
```

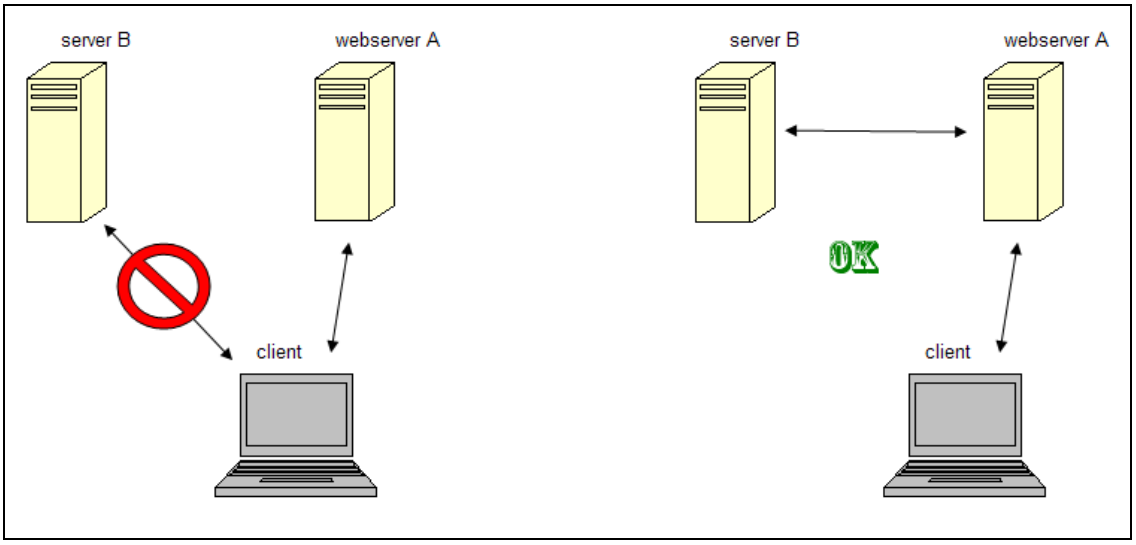
Opmerking:

- indien het opgevraagde xml bestand gegenereerd wordt door een serverscript, let er op dat dat je MIME type op *'application/xml'* instelt, anders wordt het als tekst beschouwd

### 14.2.6 Same origin policy

Web 2.0 applicaties evolueren naar *Mashup's*, dat zijn webapplicaties die gegevens van meer dan één bron combineren, denk maar aan *Google Maps* waarop locaties van alle vdab kantoren te zien zijn.

Een browser hanteert echter een ***same origin policy***. Dat betekent dat hij geen gegevens van een andere server laat ophalen door de client dan van de server waar de pagina zelf vandaan komt.




Praktisch betekent dit, dat een client-side Ajaxscript dat probeert een gegevens-bron te openen, bv. een RSS-feed (xml), de toegang geweigerd zal worden door de browser, als die feed niet van dezelfde bron komt.

'Dezelfde bron' betekent meer specifiek **hetzelfde protocol, domein en poort**.

Neem bijvoorbeeld de pagina `nieuws.html` op `www.mijnwebsite.be`

In deze pagina zit een **Javascript** dat toegang probeert te krijgen tot een aantal andere bronnen. Wat lukt en wat niet?

| URL | Lukt? | reden |
|---|---|---|

| | | |
|---|---|---|
| `http://www.mijnwebsite.be/admin/data.xml` | Ja | |
| `http://www.mijnwebsite.be/php/gegevens.php` | Ja | |
| `https://www.mijnwebsite.be/php/gegevens.php` | Neen | ander protocol |
| `http://www.mijnwebsite.be:80/php/gegevens.php` | Neen | andere poort |
| `http://www.standaard.be/nieuws/rss.php` | Neen | ander domein |

Deze belemmering kan 'omzeild' worden door:

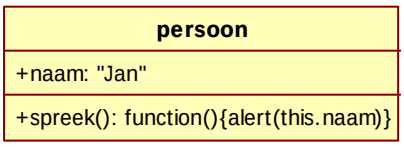
- de beveiliging van de browser uit te schakelen.
  Onwerkbaar want je kunt je gebruikers moeilijk allemaal vragen dat te doen
- *'digitally signed scripts'* te gebruiken.
  Onpraktisch want betalend en niet ondersteund door alle browsers
- de gegevensbron eerst in te laden door een script op dezelfde server en het client-side script zijn gegevens daarvandaan te laten halen.
  De oplossing.

Momenteel is same-origin-policy in zijn huidige staat een belemmering voor veel webapplicaties.

# 15 Object georiënteerd programmeren in JS

## *15.1 objecten in JS*

Een **object** in JS heeft net als in andere talen **attributen** (properties, eigenschappen).
Een property heeft een **naam** en een **waarde**.
Een property die als waarde een functie heeft noemen we een method.




Je kunt gebruik maken van de ingebouwde objecten: `Error, Math, Date, Regular Expressions`, of je eigen objecten aanmaken

Een object kan je aanmaken op verschillende manieren:

- als letterlijk **object** (*literal*) met accolades:

  ```
  var leegObject   = { }
  var ik           = {naam: "Jan"}
  ```

  via een directe lijst van *naam:waarde* paren
- met een **constructor**: een constructor is een functie waarvan je een object afleidt met het new keyword:

  ```
  var mijnObject        = new Object();
  var mijnZus           = new Persoon('Ann');
  ```
- afgeleid van een JS object:

  ```
  var datum             = new Date();
  ```

  dit is infeite ook een constructor

Nu gaan we wat dieper in op elke van deze manieren

### 15.1.1 object literal

De eenvoudigste en snelste manier om terzelfdertijd een object te creëren en eigenschappen in te stellen is een **object literal** (ook wel *object initializer*):

```
var boek = {titel: "Javascript, een vdab handleiding", auteur :"Jean Smits"}
```

De *object literal* syntax gebruikt **accolades** `{ }` om alle **eigenschappen** te bevatten telkens gescheiden door een **komma** van de volgende eigenschap.

Een eigenschap heeft altijd de structuur **naam : waarde**.

De waarde kan eender welk data type zijn: `String, Number, Boolean, object, array, function, undefined, null.`

Merk op dat het onnodig is een dergelijk object toe te wijzen aan een variabele:

```
{titel: "Javascript, een vdab handleiding", auteur :"Jean Smits"}
```

Ook *methods*, *arrays* en andere *objecten* kunnen gebruikt worden als eigenschappen:

```
var ding = {
          naam:"ding",
          californication:true,
          xfactor:12,
          lichaam:{kop:"dik",buik:"vol"},
          kinderen:["blob","blib","tina"],
          spreek: function(){alert(this.naam)}
  };
```

Een object aangemaakt op de letterlijke manier verschilt op geen enkel vlak met een object aangemaakt met een constructor.

### 15.1.2 een Constructor functie

Een andere manier om een object aan te maken, is er een **constructor** functie voor definiëren. Dit is vooral handig als je meerdere objecten van hetzelfde type wil maken, bv. meerdere objecten van het type *Persoon*.

Een constructor is infeite een ***klasse*** uit OOP. Terwijl in andere talen zoals Java en C++ daar een keyword `Class` voor gebruikt wordt, doen we hetzelfde in JS aan de hand van een constructor en zijn `prototype`.


Een constructor is een functie:

```
function Planeet (naam, diameter, afstand, jaar, dag){
  this.naam = naam;
  this.diameter = diameter;
  this.afstand = afstand; // afstand planeet - zon in AU
  this.jaar = jaar;
  this.dag = dag;
  this.AUkm = 149597870 //Astronomical Unit in km = afstand aarde - zon
}
```

- in dit voorbeeld heeft de constructor verschillende argumenten
- Het `this` sleutelwoord refereert in dit geval naar **het object** dat aangemaakt wordt
- die argumenten worden dan gebruikt om de waarde van de properties van het object in te vullen. Een property wordt aan het object gebonden met het `this` keyword.
- een constructor functie wordt bij conventie altijd met een Hoofdletter geschreven

Een nieuw **object** kan nu aangemaakt – ***geïnstantieerd*** – worden door de `new` operator te gebruiken vóór de functie :

```
var aarde  = new Planeet("aarde", 6378, 1, 1, 1, true);
var mars  = new Planeet("mars", 3397, 1.5, 1.9, 1, false);
```

- het is op deze manier dus onnodig te specificeren dat hier een **object** uit ontstaat: de `new` operator doet dat voor ons
- Doordat `Function` ook erft van `Object`, zijn in JS infeite alle objecten afgeleid van `Object`.

Je bemerkt dat onze planeten nog geen **methods** (operaties) hebben, bv. *berekenMassa()* of *roteer()*. Zoals we verder uitleggen zullen we die niet in de constructor functie plaatsen maar aanmaken in zijn `prototype`.

### 15.1.3 de `JS` constructors

Een object kan geïnstantieerd (een instantie) worden van één van de JS ingebouwde constructors/objecten.

Een typisch voorbeeld is de aanmaak van een datum:

```
var vandaag = new Date();
```

maar het kan ook algemener, zo kan je een algemeen object maken met de `Object` constructor:

```
var boek = new Object();
```

*Onze raad: als je geen specifiek object (zoals een datum of een Error,...) nodig hebt, bespaar je de moeite en gebruik een letterlijk object!*

### 15.1.4 een object herkennen

Hoe test je of een variabele een object is? We hebben enkele operatoren en eigenschappen:

**De `typeof` operator**

returnt een `String` met het variabele type:

```
var persoon = {naam:"jan", leeftijd: 45};
var lijst = ['carole',4, undefined];
(typeof(persoon)=="object"); // returnt true
(typeof(lijst)=="object"); // returnt true
typeof null;          // returnt "object"
```

Bemerk in het voorbeeld:

- een `Array` wordt onterecht ook als een object herkend.
  Array is dan ook afgeleid van Object, maar als we het onderscheid willen maken is dat onvoldoende
- `null` wordt ook als object beschouwd...

`typeof` biedt dus niet echt een sluitende test voor objecten.

**De `instanceof` operator**

returnt een Boolean als een object afgeleid is van een ander:

```
(lijst instanceof Array) //returnt true
(wereld instanceof Planeet) //returnt true
({} instanceof Object) //return true
```

**De `constructor` eigenschap**

geeft een referentie terug naar de constructor functie die dit object aangemaakt heeft:

```
var o = {};
var wereld  = new Planeet("aarde", 6378, 1, 1, 1, true);
var lijst = ['carole',4, undefined];
console.log(o.constructor)            //returnt function Object() {[native code]}
console.log (wereld.constructor)      //returnt function Planeet() {[native code]}
console.log (lijst.constructor)       //returnt function Array() {[native code]}
```

deze eigenschap is soms nuttig, meestal ben je niet veel wijzer.

**Conclusie**:

Om goede tests te maken voor `Object` en `Array` moeten we combineren.

Een test voor object:

```
function isObject(iets){
//correcte test voor objecten
//return boolean
return (iets && typeof iets === 'object');
}
```

Deze functie elimineert `null` uit de test en kijkt voor een object. Maar Arrays geven ook een positief resultaat!

Een test voor Array:

```
function isArray(iets){
//correcte test voor arrays
//return boolean
return (iets && typeof iets === 'object' && iets.constructor === Array)
}
```

## *15.2 properties en methods*

Strikt genomen heeft een **object** in JS enkel **properties**. Maar omdat een property ook een functie kan bevatten - die iets uitvoert - noemen we een property die verwijst naar een functie een **method**.
Properties en methods worden ook wel eens members van een object genoemd.

### 15.2.1 Properties

Een object is dus een niet-geordende verzameling naam:waarde paren, een kleine database.

De **naam** van een eigenschap is steeds een `string`, de **waarde** van de eigenschap is **eender welk datatype**, inclusief een ander object of een functie. Bevat de naam van de eigenschap een koppelteken of andere speciale tekens, dan mag je die volledig in aanhalingstekens zetten:

```
var cursist =
{
          naam:"Jan",
          "oog-kleur":"grijs"
}
```

Om een eigenschap van een object aan te spreken hebben we twee mogelijkheden:

- de **dot . notatie**: `object.property`

- de **sleutel notatie**: `object["property"]`

Ze zijn gelijkwaardig.
De sleutel notatie heeft het voordeel dat de property name als `String` meegegeven wordt en dat je die dus programmatorisch kunt berekenen at run-time, bijvoorbeeld in een lus, of via een variabele.

Een eigenschap kan ook altijd later aan een object toegevoegd worden of gewijzigd worden, een `var` keyword is onnodig:

```
var boek = new Object();
boek.titel = "Javascript, een vdab handleiding";
boek.auteurs = new Object();
boek.auteurs.auteur1 = "Jean Smits";
boek.auteurs.auteur2 = "Jan Vandorpe";
boek["auteurs"]["auteur3"] = "Jules Vernes";
boek.publicatieDatum = new Date(2008, 07, 30);
boek.aantalPaginas;
boek.commentaren = [“goed”,”slecht”,”wazegde?”];
boek.uitgever = undefined;
boek['editie'] = "hardcover";
```

Een eigenschap die geen waarde toegekend krijgt heeft de waarde `undefined`, in dit voorbeeld is boek.aantalPaginas `undefined.`

De letterlijke notatie is echter evenwaardig en heel wat korter:

```
var boek = {
  titel:"Javascript, een vdab handleiding",
  auteurs: {
          auteur1: "Jean Smits",
          auteur2: "Jan Vandorpe"
  },
  publicatieDatum: new Date(2008, 07, 30),
  aantalPaginas,
  commentaren : [“goed”,”slecht”,”wazegde?”],
  uitgever : undefined,
  editie: "hardcover"
}
```

Een property kan verwijderd worden met de `delete` operator:

```
delete boek.uitgever;
```

Vanaf nu zal deze `undefined` opleveren.

### 15.2.2 Enumeratie

De term enumeratie wordt hier gebruikt om het overlopen van alle eigenschappen aan te geven.
Om alle eigenschappen van een object te doorlopen gebruiken we een `for in` loop waarbij we van de sleutel notatie gebruik maken:

```
for (var key in obj ){
        alert( key + ": " + obj[key] + "\n")
  }
```

Opmerking: deze lus zal `boek.aantalPaginas` niet tonen wegens `undefined`

### 15.2.3 methods

Een **method** van een object is een **property** die verwijst naar een `function`.
Dat kan in de constructor zelf - als **anonieme functie** - of als verwijzing naar een benoemde functie:

```
function Planeet (naam, diameter, afstand, jaar, dag){
  //properties
  ...
  //methods
  this.volume = function(){
        return (4 / 3) * Math.PI * Math.pow(this.diameter/2 , 3)
  }
  this.toon = toonPlaneet;
}



function toonPlaneet(){
  strPlaneet = "planeet: " + this.naam + "\n";
  strPlaneet += "diameter: " + this.diameter + "\n";
  strPlaneet += "afstand: " + this.afstand + "\n";
  strPlaneet += "jaar: " + this.jaar + "\n";
  strPlaneet += "dag: " + this.dag;
  strPlaneet += "leefbaar: " + this.leefbaar;
  return strPlaneet;
}
```

In dit voorbeeld maakt de method `volume` gebruik van een anonieme functie, terwijl de method `toon` verwijst naar een benoemde functie. Merk ook de verwijzing naar de externe functie op: **geen haakjes**, want de property `toon` **refereert** nu naar toonPlaneet() en bevat niet het resultaat van toonPlaneet().

De functies refereren naar het object zelf via het `this` sleutelwoord.

Om de method toe te passen gebruiken we dus de **naam van de property aangevuld met een paar haakjes**, toegepast vanuit het object:

```
alert(mars.volume())
alert(wereld.toon())
```

**Belangrijke opmerking**: in dit voorbeeld wordt de method meegegeven in de Constructor functie. Dat is geen goed idee, want dat betekent dat alle geïnstantieerde objecten een copy van deze methode meekrijgen. Objecten moeten methods "*delen*", en daarom moeten we ze declareren in hun `prototype`, zie verder.

### 15.2.4 het globale Object

In Javascript zijn alle variabelen gebonden aan het *globale Object*. In een browser is het `window` object het *globale Object*.

```
var global = this;
alert(global.toString()); //returned [object Window]
```

Het `this` keyword, gebruikt buiten een object, dus bijvoorbeeld in een los statement of een losstaande functie, verwijst in dat geval steeds naar het `window` object.

Alle globale variabelen zijn properties van `window`:

```
var naam= "Jan";
leeftijd = 54;
function halloWorld(){
        var str1 = "hallo";
        str2 = " World";
        alert(str1 + str2);
};
```

De variabelen, *naam*, *leeftijd*, *halloWorld*, zijn globale variabelen: ze zijn properties van `window` *(window.naam, window.leeftijd, window.halloWorld).*

De variabele *str1* niet, die is enkel bekend binnen de functie, maar bij *str2* is het `var` keyword vergeten en die wordt dus ook *global*!

Elke variabele die niet goed gedeclareerd is, dus **zonder** het `var` keyword, wordt een globale variabele.

*Vermijd het gebruik van globale variabelen: encapsuleer ze in functies of in objecten.*

### 15.2.5 uitbreiden van de properties

Een leuk aspect van JS is dat je op elk moment een nieuwe property of method kan toevoegen aan een object:

dat hoeft helemaal niet te gebeuren in de constructor:

```
function Planeet(...){}
...
wereld = new Planeet(...);
...
wereld.californication = true;
```

Vanaf dit moment heeft wereld een eigenschap *californication*.

Hebben andere planeten nu ook deze eigenschap? Neen, als je dat wil moet je
ofwel de oorspronkelijke constructor functie aanpassen
of de eigenschap toevoegen via de `object.prototype` van Planeet

De mogelijkheid om een object op elk ogenblik te voorzien van nieuwe properties noemt men **uitbreiding** of **augmentation**.

## *15.3 Functions zijn objecten*

We hebben het al gehad over functies maar toch is het noodzakelijk nog enkele puntjes te herhalen.

Functies zijn objecten die

- als waarde van een variabele kunnen gebruikt worden
- ook een *property* van een object kunnen zijn
- als een parameter doorgegeven worden aan een andere functie
- ge-returned kunnen worden door een functie
- letterlijk geschreven kunnen worden (anonieme functie, lambda)
- zelf ook properties hebben en toegewezen kunnen worden

De volgende statements zijn identiek:

```
function foo() { }
var foo = function foo() { }
var foo = function() { }
//uitvoering:
foo()
```

Functies hebben properties:

```
function foo(naam){
  this.naam = naam;
}
foo.length              //returns 1
foo.constructor         //returns Function()
typeof foo.prototype    //returns object

foo.auteur = "Jan";
console.log(foo.auteur); //returns "Jan"
```

Opmerkingen:

- de `length` property van een functie telt het aantal argumenten
- de `constructor` property wijst (*is een referentie*) naar de **constructor** die de functie gemaakt heeft. Voor de meeste functies is dat het globale `Function` object. Het is natuurlijk mogelijk een functie te schrijven die andere functies returnt.
- elke functie heeft een `prototype` property, leeg, dat van het type object is
- je mag altijd zelf properties toevoegen

### 15.3.1 de apply() en call() methods

De globale methods `call()` en `apply()` laten je toe een functie of een method van een object toe te passen op een ander object.
Veronderstel een object *mijnGlorifier* met een method *glorify()* die het gebruikt om een naam wat op te poetsen:

```
var mijnGlorifier = {
  auteur: "Jan",
  glorify:function(){
         alert( "Graag een applaus voor de weledelgeboren heer " + this.auteur + " ,
  ↵
                        onze spreker voor vanavond");
  }
}
//uitvoeren
mijnGlorifier.glorify() // returnt de hele tekst voor Jan
```

Als we nu een andere object hebben waarop we dezelfde method willen uitvoeren, kunnen we die oproepen met `call`:

```
var Kurt = {auteur:"Kurt"};
mijnGlorifier.glorify.call(Kurt,null) ; returnt de tekst voor Kurt
```

Het eerste argument in de `call()` functie is het object dat de `this` van de functie vervangt (hier wordt dat *Kurt*). De volgende argumenten zijn deze die je nodig hebt in de oorspronkelijke method (hier geen, dus `null`).

Het enige verschil tussen apply() en call() is de manier waarop die argumenten doorgegeven worden: bij call staan ze apart, gescheiden door komma's, bij apply worden alle argumenten vervat in één array.
In een tweede voorbeeld hebben we een object *naaiMachien* met een method *naai()*. Deze method concateneert alle argumenten die het binnenkrijgt met het `arguments` array:

```
var naaiMachien = {
  tekst: "hallo ",
  naai:function(){
```

```
        for(var i=0;i<arguments.length;i++){
                this.tekst += arguments[i];
                }
        alert( this.tekst);
  }
}
//uitvoeren
naaiMachien.naai('ilse,','etienne,','jean,','inge,','pol,','jan,');
//returnt "hallo ilse, etienne, jean, inge, pol, jan,"
```

Nu kunnen we deze method ook gebruiken in een andere functie, hier de method *zaag()* van een *zaagMachien*:

```
var zaagMachien ={
  tekst: "in stukjes zagen:  ",
  zaag:function(){
        naaiMachien.naai.apply(this,arguments)
  }
}
//uitvoeren:
zaagMachien.zaag('ronny,','sarash,','tine,','paul,')
//returnt "in stukjes zagen: ronny,sarash,tine,paul,"
```

Omdat `arguments` een array is, kunnen we het zo doorgeven in `apply`.

`apply()` en `call()` worden echter vooral gebruikt bij *constructor chaining* (verder uitgelegd).
Een voorbeeld :

```
function  Persoon(naam,leeftijd){
  this.naam = naam;
  this.leeftijd = leeftijd;
  }
function Cursist(naam, leeftijd, opleiding) {
  Persoon.call(this,naam,leeftijd);
  this.opleiding = opleiding
  }
Cursist.prototype =  new Persoon();
Cursist.prototype.spreek= function(){
  alert("hallo, ik ben " + this.naam + " mijn leeftijd is " + this.leeftijd + " en ik
  volg de opleiding " + this.opleiding);
  }
var Jean = new Cursist('Jean', 52, "Javascript PF");
```

```
//uitvoeren:
Jean.spreek()
//returnt " hallo, ik ben Jean mijn leeftijd is 52 en ik volg de opleiding Javascript
  PF"
```

In de constructor *Cursist* roepen we de constructor *Persoon* op om de eigenschappen *naam* en *leeftijd* te kunnen koppelen aan het `this` object.

## *15.4 Inheritance in JS*

Om de hoeveelheid code te verminderen, om aanpassingen zo flexibel mogelijk te maken, gebruiken object-georiënteerde programmeertalen het principe van ***inheritance***.

In OOP heb je het principe van een **klasse** (*Class*) geleerd. Een *Class* is een sjabloon waarvan objecten "afgeleid" kunnen worden: een object is een **instantie** van een klasse.

In Javascript kan je ook met het principe van een klasse werken, alleen bekomen we dat op een andere manier: niet met het keyword `Class`, maar met een constructor en zijn `prototype`.

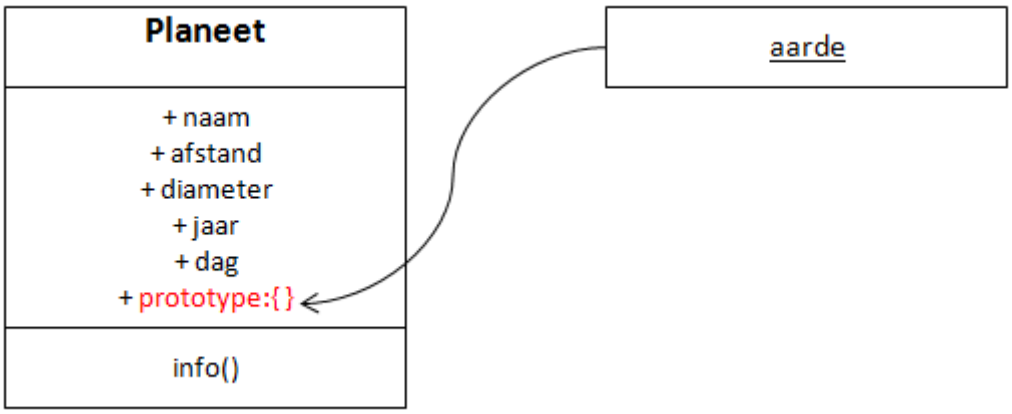
Daarnaast laat Javascript ook *overerving* toe **van object naar object** zonder dat er een klasse aan te pas komt.

### 15.4.1 de `prototype` property

Elke **functie** heeft een *`prototype`* eigenschap, het is een object.

Elk object heeft een verborgen link naar de *`prototype`* van de functie die hem gemaakt heeft, zijn "prototype" dus.

Dus, het object aarde heeft een koppeling naar *`Planeet.prototype`*.
Letterlijke objecten hebben een koppeling naar *`Object.prototype`*.
Functies zelf hebben een koppeling naar *`Function.prototype`*




Dit lege object (`prototype`) heeft geen enkele invloed op de functie zelf (*Planeet*). Gebruik je de functie echter als een constructor, om objecten van af te leiden, dan kan het een grote rol gaan spelen voor die instanties, de afgeleide objecten.

Aan de `prototype` van een constructor kan je altijd attributen toevoegen.

Alle eigenschappen en methods die in de **`prototype`** van de constructor van een object aanwezig zijn, worden geërfd door dat object.

Spreek je een method/property van een object aan, dan kijkt het object:

- **eerst** of hijzelf (de instantie) die heeft, **zoniet**
- kijkt hij of zijn constructor `prototype` die heeft

Ons voorbeeld:

```
function Planeet (naam, diameter, afstand, jaar, dag){
  this.naam = naam;
  this.diameter = diameter;
  …
 this.info = function(){
   return "Ik ben " + this.naam + " met een diameter van " + this.diameter
  }
}
```

Deze constructor heeft een aantal properties en één method.
We leiden een instantie af:

```
var aarde  = new Planeet("aarde", 6378, 1, 1, 1);
console.log(aarde.info())      //returnt "Ik ben aarde met een diameter van 6378"
```

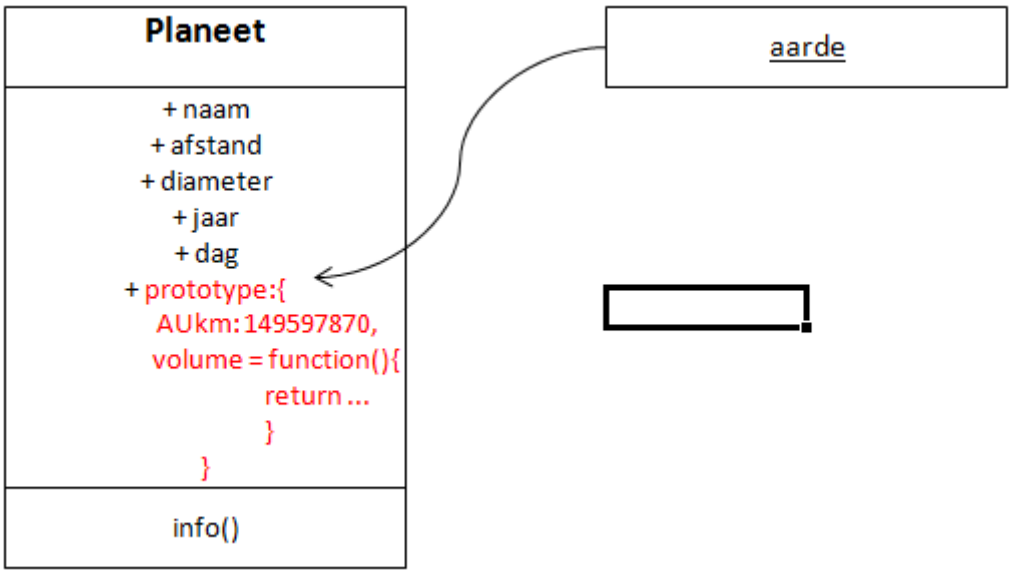
Nu voegen we een property en een method toe aan de *`prototype`* van de constructor:

```
  Planeet.prototype.AUkm = 149597870 //Astronomical Unit in km = aarde – zon
  Planeet.prototype.volume = function(){
         return (4 / 3) * Math.PI * Math.pow(this.diameter/2 , 3)
  }
```

Dan kunnen we nu doen:

```
console.log(aarde.AUkm)        //returnt 149597870
console.log(aarde.volume()) //returnt het volume van de aarde
```

Alle andere instanties van Planeet (mars, venus, …) bezitten nu ook deze property en method.




Wat is het verschil tussen een property/method gedefinieerd in de constructor zelf en die in zijn `prototype`?

- properties/methods **in** de *constructor*:

  worden **gekopieerd** voor elk object.

- properties/methods in de `prototype` van de constructor:

  worden gebruikt/**gedeeld** door alle objecten

In de constructor functie zelf zetten we dus die eigenschappen die **verschillen** voor elk object: *naam, diameter, ….*

In het prototype zetten we dus die eigenschappen die **gemeenschappelijk** zijn (*shared properties, constanten)* en **alle methods**: *AUkm, info(), volume().*

Verbeterd voorbeeld:

```
 function Planeet (naam, diameter, afstand, jaar, dag){
  //properties
  this.naam = naam;
  this.diameter = diameter;
  this.afstand = afstand; // in AU
  this.jaar = jaar;
  this.dag = dag;
}
Planeet.prototype.AUkm = 149597870; // 1AU
Planeet.prototype.volume = function(){
          return (4 / 3) * Math.PI * Math.pow(this.diameter/2 , 3)
  }
Planeet.prototype.info = function(){
```

```
   return "Ik ben " + this.naam + " met een diameter van " + this.diameter
  }
```

Belangrijke opmerking:

- Merk goed op dat het toevoegen aan het prototype altijd **na** de constructor komt, nooit  in de constructor
- maar dat kan veel later gebeuren, eigenlijk op eender welk moment!

Tot nu toe hebben we toegevoegd aan het `prototype`. een andere tactiek is het `prototype` object gewoon **overschrijven**:

```
var o = {
  AUkm: 149597870,
  volume: function(){
          return (4 / 3) * Math.PI * Math.pow(this.diameter/2 , 3)
          },
  info: function(){
          return "Ik ben " + this.naam + " met een diameter van " + this.diameter
          }
}

Planeet.prototype = o;
```

In ons voorbeeld maakt het geen verschil uit. Maar zoals je later zult zien is dat niet altijd het geval: wie weet was `prototype` niet leeg!

Het is ook perfect mogelijk om een constructor function te maken met **enkel** *prototypal* properties, dus erg gelijkend op een *Class*:

```
function Auto(){
 }

 Auto.prototype.wielen = 4;
 Auto.prototype.kleur = "wit";
 Auto.prototype.toonKleur = function(){ alert(this.kleur)};

 var toyota = new Auto();
 var peugeot = new Auto();
```

De constructor function *Auto()* bevat geen code en alle eigenschappen worden in het `prototype` gedeclareerd. De vraag is, of dat in het geval van een auto ook praktisch is?

### 15.4.2 de `constructor` property

De `constructor` property van een object wijst naar zijn constructor:

```
//een letterlijk object
var o1 = {};
o1.constructor  //returnt Object()


//een constructor functie
function Auto(naam){
  this.naam = naam;
 }
Auto.prototype.kleur = "wit";
Auto.prototype.info = function(){
         return "ik ben een " + this.naam + ", mijn kleur is " + ↵
                        this.kleur + ", ik ben een " + this.constructor;
         }
var toyota = new Auto('Toyota');
toyota.kleur = "blauw";
var peugeot = new Auto('Peugeot');


toyota.info(); //returnt "ik ben een Toyota, mijn kleur is blauw, ↵
                                        ik ben een function Auto()"
peugeot.info(); //returnt "ik ben een Peugeot", mijn kleur is wit, ↵
                                        ik ben een function Auto()"
```

De objecten geven correct aan welk hun constructor functie is.

Er ontstaat echter een probleempje als we de *shared properties/methods* niet **toevoegen** maar het `prototype` object **overschrijven**:

```
//een constructor fucntie
function Auto(naam){
  this.naam = naam;
 }
Auto.prototype = {
         kleur: "wit",
         info: function(){
                        return "ik ben een " + this.naam + ", mijn kleur is " + ↵
                        this.kleur + ", ik ben een " + this.constructor;
                }
         }
var toyota = new Auto('Toyota');
toyota.kleur = "blauw";
```

```
var peugeot = new Auto('Peugeot');


toyota.info(); //returnt "ik ben een Toyota, mijn kleur is blauw, ↵
                                    ik ben een function Object()"
peugeot.info(); //returnt "ik ben een Peugeot", mijn kleur is wit, ↵
                                    ik ben een function Object ()"
```

De `constructor` property verwijst niet meer naar de correcte constructor, maar naar `Object`. Dat komt omdat de eigenschap `Auto.prototype.constructor` vernietigd werd door het overschrijven.

Dat zal zeker problemen geven straks bij overerven, daarom herzetten we die eigenschap best na het overschrijven:

```
…
Auto.prototype = {
        kleur: "wit",
        info: function(){
                return "ik ben een " + this.naam + ", mijn kleur is " + ↵
                this.kleur + ", ik ben een " + this.constructor;
            }
        }
Auto.prototype.constructor = Auto;
var toyota = new Auto('Toyota');
…
```

### 15.4.3 Linkage

We hebben het al gezegd:

Spreek je een method/property van een object aan, dan kijkt het object:

- **eerst** of hijzelf die heeft, **zoniet**
- kijkt hij of zijn constructor *`prototype`* die heeft

Dit noemen we *linkage*.

We keren terug naar een lichtjes gewijzigd Auto voorbeeld, veronderstel dit:

```
function Auto(naam){
  this.naam = naam;
 }

 Auto.prototype.wielen  = 4;
 Auto.prototype.kleur   = "wit";
 Auto.prototype.info    = function(){
                            return "ik ben een " + this.naam + ↵
                                   " en mijn kleur is " + this.kleur;
                        };
```

```
var toyota     = new Auto("toyota");
var peugeot    = new Auto("Peugeot");

toyota.info()   //returnt "ik ben een Toyota en mijn kleur is wit";
peugeot.info()  //returnt "ik ben een Peugeot en mijn kleur is wit";
```

Alle auto's hebben een witte kleur omdat ze zelf geen eigenschap kleur hebben en die dus verder opzoeken in hun *constructor's* `prototype`.

Maar als een auto zelf een eigenschap *kleur* heeft wordt het `prototype` niet meer geconsulteerd:

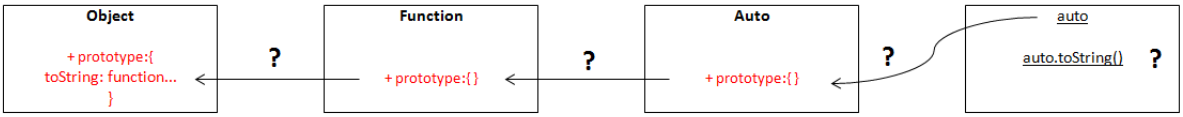
```
toyota.kleur = "blauw";
toyota.info()    //returnt "ik ben een Toyota en mijn kleur is blauw";
peugeot.info()   //returnt "ik ben een Peugeot en mijn kleur is wit";
```

Het zoeken naar de eigenschap wordt gestaakt, zodra die gevonden wordt.

De ***prototype chain*** kan echter helemaal doorgaan tot het *Global Object*: de functie *Auto()* heeft zelf een constructor, nl. `Function` met een `prototype` property, en `Function` is zelf afgeleid van `Object` die een `prototype` heeft.

Als je dus een method zoals `toString()` toepast op één van de auto's, dan krijg je een antwoord (al is het erg summier) want er wordt telkens hoger gezocht langs de ***prototype chain*** tot er iets gevonden wordt:

```
toyota.toString();      //returnt [Object object]
```




Niet de *toyota* instantie, niet de *Auto* constructor, niet `Function` hebben een method `toString()`, maar wel `Object`. Er wordt naar boven in de ketting gezocht tot wel/niet gevonden.
Als we onderweg de method overschrijven onderbreken we de ketting:

```
Auto.prototype.toString = function(){return "Auto object"}
toyota.toString();      //returnt "Auto object"
```

### 15.4.4 Augmentation van Javascript objecten

Zoals je dus merkt kan je elke constructor **achteraf** voorzien van nieuwe eigenschappen of methods door ze aan zijn `prototype` toe te voegen.
Dit geldt ook voor de ingebouwde Javascript objecten, zoals `Object, Array, Function, String, Number`, etc… Dit noemen we ***Augmentation***.

Een eenvoudig voorbeeld: We voorzien het `String` object van een method die elke tekst omkeert:

```
String.prototype.keerOm = function(){
  return Array.prototype.reverse.apply(this.split('')).join('');
}
```

we 'lenen' hier even de `reverse` method van `Array` om toe te passen op `String`. Nu kunnen we die method toepassen op elke `String` variabele:

```
var str = "hallo world";
var str2 = str.keerOm();
console.log(str2);      //returnt "dlrow ollah"
```

*Augmentation van ingebouwde objecten is gevaarlijk en moet je niet licht doen!*

Pas op wat je doet, want voor je het weet overschrijf je een bestaande method. Bedenk dat alle *afgeleide objecten* hierdoor beïnvloed worden.

Het minste wat je kunt doen is testen vooraleer toe te passen:

```
if(!String.prototype.keerOm){
  String.prototype.keerOm = function(){
          return Array.prototype.reverse.apply(this.split('')).join('');
  }
}
```

Toch is *Augmentation* de kortste en meest efficiënte weg om enkele flagrante tekorten in JS weg te werken en nuttige methods toe te voegen:

Hier voegen we bijvoorbeeld een method `trim()` toe aan het `String` object: ze verwijdert witruimte achteraan de `String`.

```
if(!String.prototype.trim){
  String.prototype.trim = function () {
          return this.replace(
                /^\s*(\S*(\s+\S+)*)\s*$/, "$1");
  }
}
```

### 15.4.5 Inheritance

Waar heb je *inheritance*, *overerving*, voor nodig?

In object-georiënteerd programmeren is het handig dat je niet telkens opnieuw attributen en operaties moet toekennen aan een object, maar dat die een aantal gemeenschappelijk attributen en methods al *in zich heeft*, geërfd heeft.

Niets nieuws onder de zon, zal je zeggen, we hebben dat al gedaan: een object afleiden van een constructor functie is *inheritance* toepassen. En je hebt gelijk. maar er zijn andere manieren. Vanaf nu overlopen we de mogelijkheden.

### 15.4.6 Classical inheritance of constructor chaining

Javascript kan net als andere OO talen, gebruikt worden alsof het *Classes* heeft.

Een "*Class"* is dan een constructor functie en zijn prototype augmentation, een voorbeeld:

```
function Vorm(naam){
  this.naam = naam;
  this.tekst = "hallo, ik ben " + this.naam;
}
Vorm.prototype.spreek = function(){
          alert(this.tekst)
  }
```

Volgens de conventie krijgt de **constructor** functie de naam van de *Class* met een Hoofdletter geschreven. Het `this` keyword wordt gebruikt om tijdens de instantiëring de attributen aan het object te koppelen. Methods worden via het `prototype` van de constructor aan de *Class* gebonden.

Daar leiden we dan **instanties** vanaf met het `new` keyword:

```
var blob = new Vorm('blob');
var blobette = new Vorm('blobette');
blob.spreek();          //returnt "hallo, ik ben blob"
blobette.spreek();      //returnt "hallo, ik ben blobette"
```

Om nu een andere *Class* af te leiden van de eerste wordt het wat ingewikkelder:

```
function Veelhoek(naam, zijden){
  Vorm.call(this, naam);
  this.zijden = zijden; //nieuwe prop
}
Veelhoek.prototype = new Vorm();
Veelhoek.prototype.constructor = Veelhoek;
Veelhoek.prototype.type = "Veelhoek";
Veelhoek.prototype.getOpp = function(){
```

```
  return 0; // abstract function
  }
Veelhoek.prototype.spreek = function(){
        alert(this.tekst + ", een " + this.type + "\n aantal zijden: " +
        this.zijden + "\n opp:" + this.getOpp())
  }
```

hier gebeuren 3 stappen:

1. de constructor van de subclass gebruikt de constructor van de superclass:
   - om de argumenten van de *Vorm* functie te kunnen gebruiken, nemen we die over
   - dan roepen we, in de *Veelhoek* constructor, de superclass *Vorm* constructor aan met de `call` functie: het `this` argument zorgt ervoor dat ook de *naam* aan een instantie van een *Veelhoek* gekoppeld wordt
   - een nieuw argument *zijden* verschijnt in de argumentenlijst
2. de *prototype chain* wordt opgezet:
   - de `prototype` van *Veelhoek* wordt een nieuw *Vorm* object, daarmee erven we al diens properties.
     Merk op dat we dat éérst doen: alle *nieuwe* methods worden nadien toegevoegd, zo niet overschrijven we het prototype
   - de `constructor` eigenschap van `prototype` wordt opnieuw op Veelhoek gezet (die was verdwenen na het vorige statement)
3. nieuwe attributen/methods worden toegevoegd aan het prototype van de subclass of er wordt aan overriding gedaan:
   - de eigenschap *Veelhoek.prototype.type* krijgt een nieuwe waarde
   - een nieuwe method *getOpp()* verschijnt. Een abstracte method die geen invulling krijgt
   - de method *spreek()* wordt ook overschreven

Nu kunnen we instanties maken:

```
var ster = new Veelhoek("ster", 9);
ster.spreek();  //returnt "hallo, ik ben ster, een Veelhoek
                            aantal zijden: 9
                            opp: 0"
```

Als we willen kunnen we nog een subclass afleiden van *Veelhoek*:

```
function Rechthoek(naam, lengte, breedte){
  Veelhoek.call(this,naam,4);
  this.lengte = lengte;
  this.breedte = breedte;
}

Rechthoek.prototype = new Veelhoek();
```

```
Rechthoek.prototype.constructor = Rechthoek;
Rechthoek.prototype.type = "Rechthoek";
Rechthoek.prototype.getOpp = function(){
        return  this.lengte * this.breedte
  }
```

Bespreking:

- ook deze constructor doet een `call` naar zijn superClass en geeft daarvoor de nodige argumenten door: de *naam* en een 4 voor het argument *zijden*
- de `prototype` bevat een Veelhoek
- de `prototype.constructor` wordt opnieuw correct op Rechthoek gezet
- twee nieuwe properties doen hun intrede: *lengte* en *breedte*
- de method *getOpp()* wordt hier ingevuld (*overriding*)

Nu kunnen we rechthoeken maken:

```
var spongeBob   = new Rechthoek("spongeBob SquarePants",8,8)
spongeBob.spreek();
var balkje      = new Rechthoek("Balkje Balkema",4,10)
balkje.spreek();
```

Te allen tijde kunnen we aan *augmentation* doen:
veronderstel dat we een algemene functie *enumerate()* hebben, die alle niet-complexe properties overloopt

```
function enumerate(){
var strProps="";
for (var key in this ){
  if(this.propertyIsEnumerable(key)){
        strProps += key + "("+typeof this[key]+"): " + this[key] + "<br />";
  }
}
return strProps;
}
```

dan kunnen we alle instances van alle subclasses in één klap van die method voorzien door het `prototype` van de superclass te *'verhogen'*:

```
Vorm.prototype.lijst = enumerate;
```

### 15.4.7 De extend() functie

Vele programmeertalen hebben een eenvoudig keyword of functie "*extends*" om een Class te doen erven van een andere. Javascript helaas niet….

Een aantal auteurs stellen voor een eigen functie aan je library toe te voegen die de prototypal inheritance vergemakkelijkt.
Deze functie laat een **subClass** erven van een **superClass**.

```
function extend(subClass, superClass){
  //leidt een Class af van een andere

  var F = function(){}
  F.prototype = superClass.prototype;
  subClass.prototype = new F();
  subClass.prototype.constructor = subClass;

  subClass.superclass = superClass.prototype;
  if(subClass.prototype.constructor==Object.prototype.constructor){
         superClass.prototype.constructor = superClass;
         }
  }
```

Bespreking:

- de `prototype` van de subClass wordt een instance van een functie F die zelf het `prototype` van de superClass krijgt
- de `subClass.prototype.constructor` wordt de subClass
- we voegen een extra property `superclass` toe aan onze subClass die ons toelaat de *superClass* te bereiken, bv. in de constructor of vooraleer zijn methods overridden zijn

Nu kunnen we de code van vorig voorbeeld korter schrijven (vergelijkbaar):

```
//===== subclass met extend====================

function Driehoek(naam, basis, hoogte){
  Driehoek.superclass.constructor.call(this, naam, 3);
   //nieuwe prop
  this.basis=basis;
  this.hoogte=hoogte;
}
extend(Driehoek, Veelhoek);

Driehoek.prototype.type = "Driehoek";
Driehoek.prototype.getOpp = function(){
```

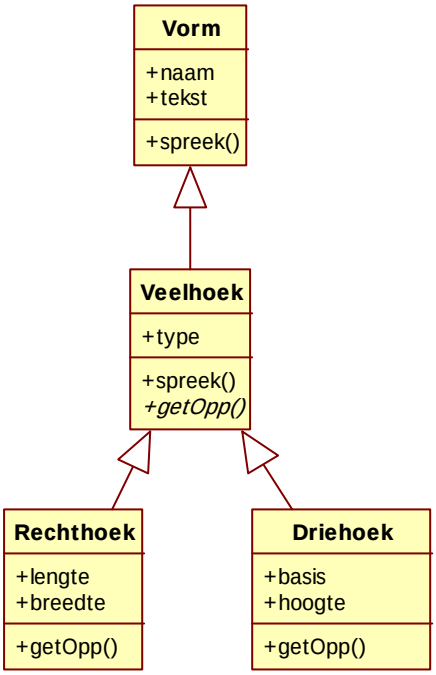
```
  return 1/2 * this.basis * this.hoogte
  }

//Driehoek instances
var piet = new Driehoek("piet", 10, 9);
piet.spreek();
```

Bespreking:

- in de *Driehoek* constructor passen we de `superclass` property toe die *extend* bezit. We hadden evengoed kunnen schrijven

  `Veelhoek.call(this, naam, 3)`
- het overschrijven van de `prototype` is nu volledig vervangen door

  `extend(Driehoek, Veelhoek)`




### 15.4.8 Prototypal inheritance en de clone() functie

Javascript is de enige taal die een andere soort inheritance toelaat, nl directe overerving van object naar object.
In plaats van te werken met *Classes* (dus constructor functies) werken we direct met het `prototype` van een object.

Het principe is als volgt: om een nieuw object attributen/methods te doen erven overschrijven we zijn `prototype` met een ander object.

Om dat vlot te laten verlopen gebruiken we de ***clone()*** functie (soms vind je hem ook onder de naam object()), die we natuurlijk ook aan onze library toevoegen.

```
function clone(o) {
  function F() {}
  F.prototype = o;
  return new F();
}
```

Hoe werkt het? de functie maakt een anonieme functie. Diens `prototype` wordt overschreven door het 'te klonen object'. Dan wordt de anonieme functie gebruikt als constructor om een leeg object aan te maken wiens prototype gelijk is aan het 'te klonen object'. Dit object wordt gereturned.

Een voorbeeld, we maken een letterlijk object *Land*:

```
var Land = {
         naam: "neutraal",
         kleuren:["wit"],
         regios: {},
         spreek: function(){
                return "hallo, ik ben  " + this.naam +
                "\nde kleuren van mijn vlag zijn " + this.kleuren.toString() +
                "\nmijn regio's zijn " + enumerate(this.regios);
                }
  }
```

Opmerkingen:

- We geven dit object variabele ook een HoofdLetter omdat hij als 'sjabloon' zal fungeren
- de functie enumerate gebruiken we enkel om alle properties te overlopen

Dan kunnen we direct andere objecten ervan afleiden met de *clone()* functie:

```
//klonen

var belgie = clone(Land);
belgie.naam = "België";
console.log(belgie.spreek());
// returnt      "hallo, ik ben België
                de kleuren van mijn vlag zijn wit
                mijn regio's zijn: "

var nederland = clone(Land);
nederland.naam = "Nederland";
console.log(nederland.spreek());
// returnt      "hallo, ik ben Nederland
```

```
                de kleuren van mijn vlag zijn wit
                mijn regio's zijn: "
```

De twee 'klonen' hebben de eigenschappen/operaties van *Land* geërfd. De *naam* property is overschreven.

Het is echter belangrijk te beseffen de klonen geen eigen properties/methods hebben: het zijn allemaal referenties naar het prototype object dat een *Land* is.
Als we dus vragen

```
console.log(belgie.kleuren.toString())
```

dan lezen we de property *kleuren* van het object *Land*.
Het is enkel als je een kloon een eigen kopie van een eigenschap geeft dat het prototypal object niet gelezen wordt. Dat gebeurt met het statement

```
belgie.naam = "België"
```

Dat wordt nog eens gedemonstreerd als we een eigenschap wijzigen zonder hem te overschrijven:

```
var belgie = clone(Land);
belgie.naam = "België";
belgie.kleuren.pop(); //verwijdert wit
belgie.kleuren.push("zwart","geel","rood");
console.log(belgie.spreek());
// returnt      "hallo, ik ben België
                de kleuren van mijn vlag zijn zwart, geel, rood
                mijn regio's zijn: "

var nederland = clone(Land);
nederland.naam = "Nederland";
console.log(nederland.spreek());
// returnt      "hallo, ik ben Nederland
                de kleuren van mijn vlag zijn zwart, geel, rood
                mijn regio's zijn: "
```

Ook Nederland heeft de Belgische kleuren gekregen…
Dat komt omdat het wissen en het toevoegen van de kleuren gebeurde in *Land*, *belgie* heeft geen eigen kopie van die eigenschap.
We kunnen dat voorkomen door eerst een eigen kopie van het **datatype** toe te kennen aan de variabele:

```
var belgie = clone(Land);
belgie.naam = "België";
belgie.kleuren = [];
belgie.kleuren.push("zwart","geel","rood");
```

```
console.log(belgie.spreek());
// returnt      "hallo, ik ben België
                de kleuren van mijn vlag zijn zwart, geel, rood
                mijn regio's zijn: "

var nederland = clone(Land);
nederland.naam = "Nederland";
console.log(nederland.spreek());
// returnt      "hallo, ik ben Nederland
                de kleuren van mijn vlag zijn wit
                mijn regio's zijn: "
```

Wat ga je gebruiken, 'Class inheritance' of 'Prototypal inheritance'?

Het is je eigen keuze, prototypal inheritance kan veel eenvoudiger en korter zijn , de meeste mensen verkiezen te werken met 'Classes' omdat ze die goed begrijpen.

# 16 Code Technieken en design patterns in JS

Hier illustreren we enkele programmeertechnieken die je kunt gebruiken, van een eenvoudige techniek tot volwaardig *design patterns*.

De *design patterns* zijn enkel bedoeld voor cursisten die *Javascript Programming Fundamentals* doen. Design patterns zijn universeel, maar hebben in JS diksijls een eigen *twist.*

## *16.1 Code technieken*

### 16.1.1 lazy evaluation

*Lazy evalution* steunt op het principe dat als het resultaat van een eerste logische evaluatie duidelijk is, de rest van de expressie niet meer geëvalueerd wordt:

```
var antwoord = true && false && true && false && true
```

In bovenstaand voorbeeld met de AND operator is al duidelijk na de tweede waarde dat de eindwaarde `false` zal zijn, daarom wordt de rest niet meer geëvalueerd.

Dit geldt evenzeer voor de OR operator:

```
var antwoord = true || false || true
```

Zodra een waarde `true` is, is het eindresultaat `true`, wat de andere waarden ook zijn.

Een eerste toepassing kan je vinden in het **trapsgewijs** testen:

```
if (o && o.readyState == 4) {
        //voer verder uit
  }
```

Het heeft geen zin om de *readyState* property te lezen als er kans is dat het object *o* niet bestaat.

Dit principe kan ook toegepast worden om **optionele argumenten** van een functie van een **standaardwaarde** te voorzien, bijvoorbeeld:

```
function (o){
  var options = o || {};
}
```

Hierboven verwachten we een argument *o* dat optioneel is. Indien het afwezig is, `undefined` of `null`, dan wordt het vervangen door een leeg object.

### 16.1.2 bestaat een variabele?

Onbestaande variabelen kunnen problemen geven:

```
alert("bestaatNiet bevat de waarde: " + bestaatNiet);
```

geeft de **fout** "*bestaatNiet is not defined*"

👎 Doe niet:

```
if(bestaatNiet) { alert("bestaatNiet bevat de waarde: " + bestaatNiet)}
```

want hoewel er geen *runtime error* optreedt, geeft JS nog steeds de fout.
Deze test steunt op het evalueren van `(bestaatNiet)`: als dat `false` geeft gaat de test niet door, enkel indien `true`.
Daarom is hij ook niet sluitend want de variabele *bestaatNiet* kan toch bestaan en de waarde `false` gekregen hebben:

```
var bestaatNiet = false;
if(bestaatNiet) { alert("bestaatNiet bevat de waarde: " + bestaatNiet)}
  // gaat niet door
```

👍 maar wel:

```
if(typeof bestaatNiet !== "undefined") {
  alert("bestaatNiet bevat de waarde: " + bestaatNiet)
}
```

De `typeof` operator geeft **altijd** een `String` als antwoord. We vergelijken dat met "*undefined*".

### 16.1.3 Object/feature detection

Vergelijkbaar met het vorige maar toch anders is het testen van het bestaan van objecten en methodes, typisch de DOM features, vooraleer ze te gebruiken.
Dit heeft meer te maken met "*graceful degrading*" en "*unobtrusive javascript*" in browsers, met andere woorden, een gebruiker die een oudere/andere browser heeft, niet voor het blok zetten, en de website niet afhankelijk maken van het script.

👍 In plaats van aan *browser detection* te doen, controleer of de browser een methode ondersteund:

```
if (window.XMLHttpRequest) {
        xmlhttp = new XMLHttpRequest();
}
else if(window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
}
else {
        throw new Error("XMLHttpRequest niet ondersteund");
```

```
 }
```

Een andere manier van werken is met de `in` operator:

```
if('querySelector' in document && 'localStorage' in window &&
                                                'addEventListener' in window) {
  // moderne html5 browser kan nieuwe apps aan
}
```

### 16.1.4 Function parameters

Enkele technieken om om te gaan met functie parameters.

#### Default waarde voor een optionele parameter

In oudere browsers moeten we de aanwezigheid van de parameter testen.
Met `typeof` :

```
function maal(a,b){
        b = (typeof b !== 'undefined')? b : 1;
        return a * b;
    }
maal(5);        // 5
maal(2,5);      // 10
```

alternatieve methode:

```
function maal(a,b){
        b = b || 1;
        return a * b;
    }
```

Geeft hetzelfde resultaat.

In een nieuwere browser die Ecmascript 6 ondersteunt, kan je ook dit doen:

```
function maal(a, b=1){
        return a * b;
    }
```

#### Options hash

Als je echter een functie hebt met veel parameters, waarvan sommige optioneel, andere niet, kan je vorige techniek nog moeilijk gebruiken.

Gebruik dan een *object waarvan de properties de argumenten* zijn: een *options hash*.

Veronderstel deze functie:

```
function maakTekst(tekst, letterType, letterGrootte, kleur, achtergrondKleur, vet,
  schuin)
{
…
}
```

waarvan enkel de *tekst* parameter verplicht is, dan heb je heel wat werk om de andere argumenten te checken!

Vervang alle optionele parameters door één object met alle parameters als eigenschap:

```
function maakTekst(tekst, opties){
  if (typeof tekst !== 'undefined'){
        // de standaardwaarden
        var settings = {
              kleur   : "blue",
              bgKleur : "yellow",
              type    : "Times New Roman",
              vet     : "normal",
              grootte : "1em",
              schuin  : "italic"
        }

        for (var j in opties || {} ) {
              settings[j] = opties[j];
        }

        var tee = document.createTextNode(tekst);
        var pee = document.createElement('p');
        pee.appendChild(tee);
        with (pee.style){
                    color = settings.kleur;
                    backgroundColor = settings.bgKleur;
                    fontFamily = settings.type;
                    fontWeight = settings.vet;
                    fontSize = (settings.grootte); //eval
                    fontStyle = settings.schuin;
              }
        inhoud.appendChild(pee);
  }
  else {throw new Error('tekst is een verplicht argument')}
}
```

De verplichte parameter *tekst* wordt gecontroleerd.
Een *settings* object voorziet alle standaardwaarden als properties. De lus doorloopt alle properties van *opties* en overschrijft daarmee de waarden van het *settings* object. Dit gebeurt enkel als er een gelijknamige property is in *opties*.
*opties* kan ook helemaal achterwege blijven in dat geval wordt een leeg object voorzien en is de lus dus snel klaar…Daarna worden de waarden van settings toegepast.

Voordelen van een *options hash*:

- volgorde van argumenten is onbelangrijk
- alle eigenschappen zijn optioneel
- het settings object zelf is optioneel
- gemakkelijk uitbreidbaar
- gemakkelijk leesbaar

Bijna alle JS libraries (JQ, BB, dojo, Underscore, …) gebruiken bij deze techniek een `extend` method waarmee 1 object *versmolten* (*merge*) wordt met een tweede. Een voorbeeld in jQuery:

```
oOptions = {a:0, b:7};
oDefaults = {a:1, b:2, c:3};
oResult = $.extend(oDefaults, oOptions); // {a:0, b:7, c:3};
```

### 16.1.5 Lazy function definition

De *Lazy function definition* is een techniek waarbij een functie zichzelf overschrijft tijdens de eerste uitvoering.
Andere termen voor deze techniek zijn *lazy initialization* of *lazy loading* en *memoization* of *memoizing*
Functies die ingewikkelde/langdurige berekeningen doen, maken dikwijls gebruik van gegevens die ingelezen moeten worden (en waarop meestal gewacht moet worden tot de pagina geladen is) waarna er beslissingen gebeuren die de berekeningswijze en uitkomst bepalen.
Het is meestal onnodig om bij een tweede functieoproep die bronnen opnieuw te lezen en opnieuw dezelfde beslissing te nemen. Daarom overschrijft de functie zichzelf met een veel kortere en efficiëntere versie. Dit overschrijven gebeurt enkel bij de eerste uitvoering.
Dit kan zorgen voor een veel snellere uitvoering van het programma.

Een voorbeeld brengt meer duidelijkheid:

> In een Ajax applicatie wordt een method *createXhRObject()* gedefinieerd. Deze method returnt een `XMLHttpRequest` object. Omdat er verschillen zijn in de

browser objecten die zo'n object returnen, doet de functie aan *object detection* en bepaalt eerst welke techniek hij kan gebruiken: `window.XMLHttpRequest` of `window.ActiveXObject`? en in dat laatste geval bestaan er nog verschillende versies van.

Als we deze method zo laten staan dan wordt die beslissingsboom elke keer doorlopen als we een andere Ajaxcall doen! Nochtans weten we al na de eerste uitvoering hoe we het object moeten aanmaken en het is telkens ook hetzelfde object.

Daarom wordt de functie geherdefinieerd en returnt eenvoudigweg het *xmlhttp* object dat we de eerste maal aangemaakt hebben. De snelheid van uitvoering verhoogt hier enorm (en dat is nodig in een Ajaxcall…)

```
…
  createXhrObject: function(){
                 //memoizing
                  var xmlhttp  = '';
                  if (window.XMLHttpRequest) {
                        xmlhttp   = new XMLHttpRequest();
                        }
                  else if (window.ActiveXObject) {
                   try {
                        xmlhttp  = new ActiveXObject("Msxml2.XMLHTTP");
                   }
                   catch (e){
                        try{
                         xmlhttp  = new ActiveXObject("Microsoft.XMLHTTP");
                        }
                        catch (e){}
                   }
                 }

                  this.createXhrObject = function() {
                         return xmlhttp ;
                  }
                  return xmlhttp ;}
}
```

Bespreking:

- alle mogelijkheden worden doorlopen om te bepalen welke *constructor* we moeten gebruiken in de browser van de gebruiker,
- de statement `this.createXhrObject` vervangt de volledige code door het ene correcte object.
  Veronderstel dat `window.XMLHttpRequest` werkt dan is de inhoud van *createXhrObject* nu:

  ```
  createXhrObject: function(){
  ```

```
        return new XMLHttpRequest();
    }
```

Bij een volgende ajax call wordt meteen het correcte object geleverd.

Nog een voorbeeld:

Een functie *getStyle* berekent de reële waarden van een CSS property voor een element. Dit is een gekend geval van 'browserverschillen' waarbij IE op een heel andere manier tewerk gaat dan de andere browsers.

In plaats van telkens opnieuw de testen te doorlopen, herschrijft de functie zichzelf volgens die methode die de browser aankan.

```
function getStyle(element, property) {
        if (document.defaultView && document.defaultView.getComputedStyle) {
           getStyle = function(element, property) {
                       return document.defaultView.getComputedStyle(el,
  '')[property];
                   }
        } else if (el.currentStyle) { // IE
            getStyle = function(element, property) {
                       return el.currentStyle[property];
                   }
        }
        return getStyle(element, property);
  }
```

*Lazy function definition* is een techniek die door de meeste libraries toegepast wordt om op die manier alle mogelijke browserverschillen op een efficiënte en vooral snelle manier op te lossen.

### 16.1.6 Method chaining

Een typisch stukje code uit jQuery

```
$('a').addClass('rood').filter('a[target]').addClass('groen');
```

Demonstreert het principe van *chaining*, of *method chaining*: het uitvoeren van verschillende methods op hetzelfde object.

Bovenstaande code is dus het equivalent van

```
$('a').addClass('rood');
$('a').filter('a[target]');
$('a').addClass('groen');
```

Waarbij `$('a')` een collection is van `a` elementen (geselecteerd op de JQ manier).

Dit zorgt natuurlijk voor een serieuze afslanking van de hoeveelheid code die je moet schrijven. Bijna alle grotere libraries passen dit principe toe.

Hoe werkt het? Kan je het zelf toepassen in *plain vanilla Javascript*?

Als een object method **hetzelfde object returned als waarde**, is de method *chainable*. Dit lukt dus enkel voor methods die geen andere waarde moeten teruggeven.

Veronderstel deze constructor en een instantie ervan:

```
function Persoon(sVoornaam, sFamilienaam, ISOStringGeboortedatum, sGeslacht){
       this.voornaam        = sVoornaam;
       this.familienaam     = sFamilienaam;
       this.geboorteDatum   = new Date(ISOStringGeboortedatum);
       this.geslacht        = sGeslacht.toLowerCase();  // vrouw/man
   }
   Persoon.prototype.getNaam =  function(){
        return this.voornaam + " " + this.familienaam;
    }
   Persoon.prototype.spreek =  function(){
       var vandaag      = new Date();
       var leeftijd     = vandaag.getFullYear() - this.geboorteDatum.getFullYear();
       return "Hi! ik ben " + this.getNaam() + ", ik ben een " + this.geslacht +
                              ", en ik ben " + leeftijd + " jaar oud";
   }
// instantiëring
    var jan = new Persoon('jan','vandorpe','12/14/1954','man')
    console.log(jan.spreek());
```

Dit levert de verwachte resultaten op.

Merk op dat de methods *getNaam()* en *spreek()* een String returnen. Een dergelijke method is niet *chainable* te maken.

Nu voegen we twee methods toe die, dankzij recente wetenschappelijke vooruitgang, mogelijk geworden zijn:

```
    Persoon.prototype.verjong =  function(iJaren){
        var geboorteDag         = this.geboorteDatum.getDate();
        var geboorteMaand       = this.geboorteDatum.getMonth();
        var geboorteJaar        = this.geboorteDatum.getFullYear();
        var nieuwGeboorteJaar   = geboorteJaar + iJaren;
        this.geboorteDatum = new Date(nieuwGeboorteJaar,geboorteMaand,geboorteDag);
    }
    Persoon.prototype.switchGender =  function(){
        if(this.geslacht == "vrouw") this.geslacht = "man";
        else if(this.geslacht == "man") this.geslacht = "vrouw";
    }
```

De method *verjong()* laat toe een aantal jaren van je leeftijd af te schaven, de method *switchGender()* laat je toe *on-the-fly* van geslacht te wisselen…

En we passen dit even toe:

```
var jan = new Persoon('jan','vandorpe','12/14/1954','man')
jan.verjong(10);
jan.switchGender();
console.log(jan.spreek());
```

Het resultaat ziet er al veel knapper uit…

Maar met een eenvoudige ingreep kunnen we deze methods *chainable* maken: ze returnen momenteel geen waarde, ze wijzigen enkel de eigen attributen. Verander:

```
Persoon.prototype.verjong =  function(iJaren){
        var geboorteDag         = this.geboorteDatum.getDate();
        var geboorteMaand       = this.geboorteDatum.getMonth();
        var geboorteJaar        = this.geboorteDatum.getFullYear();
        var nieuwGeboorteJaar   = geboorteJaar + iJaren;
        this.geboorteDatum = new Date(nieuwGeboorteJaar,geboorteMaand,geboorteDag);
        return this;
    }
    Persoon.prototype.switchGender =  function(){
        if(this.geslacht=="vrouw") this.geslacht = "man";
        else if(this.geslacht=="man") this.geslacht = "vrouw";
        return this;
    }
    var jan = new Persoon('jan','vandorpe','12/14/1954','man');
    //chaining
    jan.verjong(10).switchGender().verjong(5).switchGender();
    console.log(jan.spreek());
```

Door een method `this` te laten returnen, is het resultaat van de method opnieuw het object zelf, waarop je dan een volgende method kunt uitvoeren.

In jQuery zijn alle *wrapper set* `$()` methods *chainable* omdat het hier eigenlijk steeds om het jQuery object zelf gaat.

Een opmerking die dikwijls gemaakt wordt over *chaining* is dat het de code moeilijker begrijpbaar maakt. Je kan echter deze code evengoed zo schrijven:

```
//chaining
jan
  .verjong(10)
  .switchGender()
  .verjong(5)
  .switchGender();
```

**Conclusie:**

Elke object method die geen expliciete waarde returned, kan *chainable* gemaakt worden met `return this`.

### 16.1.7 Namespacing en applicaties

Als je de JS Programming Fundamentals doorgenomen hebt, ziet je code er meestal uit als volgt:

```
var bericht = "goedemorgen"; //globale var
var vbDiv1; //globale var

function init(){
  var vbDiv1 = document.getElementById('vb1');
}

function toon(){
  vbDiv1.innerHTML = bericht;
}
```

Sequentieel programmeren dus. Twee globale variabelen en nog eens twee functies die ook in de global Namespace leven.

Waarom ook niet? inderdaad, zolang het bij één enkel script voor één pagina blijft is er geen probleem, maar als je complexere toepassingen maakt met meerdere libraries, kan dit niet meer.

Een manier om de clash van variabelen in de Global Namespace te vermijden is alle functies en variabelen onder te brengen onder **1 globaal object**: een **applicatie.**

De applicatie werkt als **NameSpace** want alles wat we nodig hebben bevindt zich **binnen** het object.

```
var MijnApplicatie = MijnApplicatie || { }
```

De logische operator dient om te controleren of het al niet bestaat.

Alle andere objecten, constanten zijn *properties* van dit object:

```
MynApplicatie.basisModule = { }
MynApplicatie.opmaakModule = { }
MynApplicatie.beginWaarde = 1
```

Ook functies zijn properties van de applicatie of van één van zijn subobjecten:

```
MynApplicatie.init = function () { }
MynApplicatie.opmaakModule.doeIets = function (){ }
```

Als we dus van buiten de applicatie een eigenschap lezen of een functie uitvoeren, moeten we dat altijd via zijn *Namespace* doen:

```
alert(MijnApplicatie.beginWaarde);
MynApplicatie.opmaakModule.doeIets();
```

Zo'n applicatie kan aangemaakt worden met verschillende design patterns, zie verder.

### 16.1.8 Callback functions

*Functional programming* is het gebruik van functies als argumenten in andere functies. Een *callback functie* is een function - al dan niet anonymous - die uitgevoerd wordt binnenin een andere functie of als een event optreedt.

Een voorbeeld met een anonieme functie in een `forEach` structuur:

```
var cursisten =  ['Paul','Adinda','Ilse','Siska'];
cursisten.forEach(function(naam, index){
  console.log( ++index + ") " + naam );
})
```

Een voorbeeld in een NodeJS REST API:

```
app.get('/api', function(request, response) {
    response.send('Filmotheek is actief');
})
```

Waarbij de anonieme functie uitgevoerd wordt als de methode *app.get* uitgevoerd wordt.
Callbacks zijn **closures**, dat wil dus zeggen dat de callback ten allen tijde toegang heeft tot de **scope** van de 'moederfunctie', zelfs als die veel later uitgevoerd wordt.

Callbacks kunnen ook benoemd worden en gebruikt als argument, een voorbeeld:

```
var allUserData = []; //global
// generische log functie
function log (userData) {
        if ( typeof userData === "string") {
            console.log(userData);
        }
        else if ( typeof userData === "object") {
            for (var item in userData) {
                console.log(item + ": " + userData[item]);
            }
        }
}

// functie met twee parameters, de tweede een callback function
```

```
function getInput (options, callback) {
        allUserData.push (options);
        callback (options);
}

//functie oproep
getInput ({name:"Jan", speciality:"JavaScript"}, log);
```

In dit voorbeeld wordt de functie *getInput* gebruikt met 2 parameters:

- Een *options* object
- Een *callback* argument: eender welke (toepaselijke) functie

Merk op dat enkel de naam van de functie gegeven wordt, gebruik geen haakjes ().
Binnen *getInput* wordt de *callback* functie dan uitgevoerd en het *options* object meegegeven. Die gebruikt het om te loggen.

Het gebruik van het `this` keyword kan een probleem opleveren in een callback:

```
var cursist = {
        id: 094545,
        volleNaam: undefined,
        setVolleNaam: function (voorNaam, famNaam)  {
              this.volleNaam = voorNaam + " " + famNaam;
        }
    }

function setCursist(voorNaam, famNaam, callback)  {
        callback (voorNaam, famNaam);
}
//uitvoering
setCursist("Paul","Kiekens",cursist.setVolleNaam);
console.log(cursist.volleNaam); // undefined
```

In dit voorbeeld wordt de object method *cursist.setVolleNaam* uitgevoerd als *callback*. Dit mislukt omdat de *callback* uitgevoerd wordt binnen de context van de functie *setCursist* : op dat moment verwijst `this` naar het globale `window` object, niet naar cursist, dus is *cursist.volleNaam* `undefined`.

Om dit probleem op te lossen bestaan er twee gelijkaardige functies `call` en `apply` . Ze verschillen enkel in de manier waarop je parameters meegeeft.

We passen onze functie *setCursist* aan:

```
function setCursist(voorNaam, famNaam, callback, thisObject)  {
        callback.apply (thisObject,[voorNaam, famNaam]);
}
//uitvoering
setCursist("Paul","Kiekens",cursist.setVolleNaam, cursist);
```

```
console.log(cursist.volleNaam); // "Paul Kiekens"
```

- De JS functie `apply` heeft als eerste argument het object waarop de `this` context geplaatst moet worden
- De andere argumenten volgen in een Array

De functies `call` en `apply` verschillen enkel in syntax:

```
fn.apply(thisArg, [argsArray])
fn.call(thisArg, arg1, arg2, arg3, …)
```

Terwijl bij `apply` de mee te geven argumenten in een Array geplaatst moeten worden, kan je ze bij `call` gewoon als comma-separated list zetten.

Conclusie:
Een callback function is een functie die als argument doorgegeven wordt aan een andere functie. Callback functions zijn closures. Ze worden vooral toegepast bij Eventhandlers en bij asynchrone uitvoering in Ajax calls. Maar bij verweven , meer laagse asynchrone callbacks, waarvan de volgorde van uitvoering en de timing van het resultaat onzeker is, kan dit leiden tot een zogenaamde "*callback hell*": de code is te complex, het resultaat onvoorspelbaar.

### 16.1.9 Promises

Relatief nieuw in Javascript zijn promises. Ze zijn opgenomen in de ES6 proposal.
Javascript is *single-threaded* en dus *inherent* ***blocking***: het ene statement moet wachten op de uitvoering van het vorige statement.
Callbacks en asynchrone calls bieden een oplossing, maar worden snel onoverzichtelijk en foutafhandeling een 'soepje'. Promises bieden daar een oplossing.
Een Javascript *promise* stelt **het resultaat van een asynchrone call** voor.
Een *promise* kan in één van deze toestanden zijn:

- `pending`: de starttoestand, niet `fulfilled` of `rejected`, wachtend
- `fulfilled`: de toestand met een succesvol resultaat
- `rejected`: de toestand van een mislukt resultaat

Een *settled* Promise is ofwel `fulfilled` of `rejected`.
Er bestaat een Standaard waaraan promises moeten voldoen: *Promises/A+* (https://www.promisejs.org/).
Promises zijn in één of andere vorm al een tijdje aanwezig in JS libraries:

- jQuery `deferred`, zijn subtiel verschillend, voldoen niet aan *Promises/A+*
- de Q, when, WinJS en RSVP libraries

Promises worden momenteel door enkele browsers ondersteund (FF, Chrome), voor de anderen heb je een library nodig of een polyfill.
Veronderstel dat we een Ajax method maken die een Promise returned:

```
function get(url) {
            // Return een new promise.
```

```
            return new Promise(function (resolve, reject) {
                // XHR Request
                var req = new XMLHttpRequest();
                req.open('GET', url);
                req.onload = function () {
                    // dit ook bij een 404
                    // dus check status
                    if (req.status == 200) {
                        // Resolve de promise meet de response text
                        resolve(req.response);
                    }
                    else {
                        // anders reject met de status text die een foutbericht bevat
                        reject(Error(req.statusText));
                    }
                };
                //  verwerk network errors
                req.onerror = function () {
                    reject(Error("Network Error"));
                };
                // Maak de request
                req.send();
            });
        }
```

- De promise heeft een static method `resolve()` die de waarde beschikbaar maakt voor de *success* parameter van de `then()` method
- De `reject()` static method returns het promise object met een `rejected` status voor de *failure* parameter van een `then()` function en geeft de reden mee als argument

Veronderstel een bestand *boek.json*:

```
{
    "heading": "<h1>Een boek</h1>",
    "hoofdstukUrls": [
        "hoofdstuk-1.json",
        "hoofdstuk-2.json"
    ]
}
```

en *hoofdstuk1.json*:

```
{
    "chapter": 1,
```

```
    "html": "<h2>Hoofdstuk 1</h2><p>Sed ut perspiciatis unde omnis iste natus error
  sit voluptatem accusantium doloremque laudantium, … nulla pariatur?</p>"
}
```

Nu gebruiken we deze method (uitvoeren in webserver anders CORS fout)

```
get(boek.json').then(function (response) {
      console.log("Success!", response);
}, function (error) {
      console.error("Mislukt!", error);
});
```

Een instantie van een *promise* heeft slechts twee methods : `then()` en `catch()`.
De `then()` functie

- bevat twee *callback* functies als argumenten: respecievelijk
  - de *succes* en
  - de *failure* handler

  Beide kunnen zowel als anonieme functie geschreven worden als benoemd worden. Ze zijn in principe optioneel, de failure handler weglaten gebeurt dikwijls maar dat zou je niet mogen doen.
- returned een *promise*

Omdat `then()` functies zelf een promise returnen, kunnen ze *gechained* worden, bijvoorbeeld:

```
get('story.json')
        .then(function (response) {
                return JSON.parse(response); //return object
         }, function (error) {
                console.error("Failed!", error);
         })
         .then(function(response){
                console.log(response.chapter)
          })
```

In dit verder uitgewerkte voorbeeld hebben we de eerste `then` een returnwaarde gegeven: het geparste JSON object, dat in een tweede `then` opgevangen wordt en geinterpreteerd.

Je kan dus dit principe toepassen om asynchrone calls te *chainen*. We maken eerst een helperfunctie *getJSON* en gebruiken die dan om `then` functies te chainen:

```
function getJSON(url){
    //return een geparste promise
    return get(url).then(JSON.parse);
```

```
    }


getJSON("boek.json")
      .then(function(boek){
            return getJSON(boek.hoofdstukUrls[0]);
        })
      .then(function(hoofdstuk){
            console.log("hoofdstuk 1: ", hoofdstuk);
        })
```

- de functie *getJSON* returned een *promise* met een geparst JSON object
- we roepen de functie aan met het bestand *boek.json* en krijgen dus een *boek* object die
    - met de eerste `then` een *promise* returned met het eerste hoofdstuk als JSON object en
    - met de tweede `then` dit hoofdstuk toont

Op die manier *chainen* we - bij success - de ene async call na de andere.

Foutopvang kan gebeuren door het failure argument van de then functie, hierboven achterwege gelaten. Indien we dat wel invoegen ziet ons script er als volgt uit:

```
getJSON("boek.json")
                .then(
                    function(boek){
                        return getJSON(boek.hoofdstukUrls[0]);
                    },
                    function(error){
                        console.log("Mislukt: ", error);
                    })
                .then(
                    function(hoofdstuk){
                        console.log("hoofdstuk 1: ", hoofdstuk );
                    },
                    function(error){
                        console.log("Mislukt: ", error);
                    })
```

In dit geval wordt ofwel de *succes* functie uitgevoerd, ofwel de *failure* functie, nooit allebei. Indien een fout zich voordoet zal success nooit uitgevoerd worden

Een alternatieve manier van werken is een catch() functie gebruiken:

## 16.2 Design patterns

De rest van dit hoofdstuk bespreekt een aantal formele design patterns.
Een *design pattern* beschrijft een manier van programmeren om een 'slimme', *object georiënteerde* oplossing te bieden voor een probleem.
Een *design pattern* beschrijft formeel de relaties tussen objecten, de eigenlijke toepassing ervan verschilt in elke taal.
Je kunt ze opdelen in een aantal groepen:

- *design patterns* die oplossingen bieden om objecten **aan te maken**
- structurele patronen die beschrijven hoe je objecten **samenstelt** om goed te functioneren
- gedragspatronen die beschrijven hoe objecten kunnen **communiceren** om goed te functioneren

Wij beperken ons tot enkele nuttige *design patterns*, die specifiek zijn voor Javascript. Voor meer verwijzen we naar de literatuur.

### 16.2.1 Singleton

Een **singleton** is het design patroon dat de aanmaak van een **object** voorstelt waarvan er maar **één instantie** is. Een tweede instantie van hetzelfde soort object is en mag er ook niet zijn.

In JS is een singleton heel simpel te maken: maak een **object literal**:

```
var Singleton = {}
```

De "applicatie" die we hierboven beschreven om als *Namespace* te fungeren is een singleton. De basisstructuur van zo'n singleton in Javascript is als volgt:

```
var Singleton = {
  attribuut1: "hallo",
  attribuut2: true,
  method1: function(){
                …
  },
  method2: function(){
                …
  }
}
```

Dus één object met properties en methods.
De attributen zijn daarmee enkel te bereiken via de *namespace* van het object: `Singleton.attribuut1` en `Singleton.method2()`.

Een *singleton* kan ook dienen als *Namespace* voor "subapplicaties". Elke subapplicatie is een property van het *Namespace* object en is een *singleton* op zichzelf:

```
/* MijnApplicatie namespace */
var MijnApplicatie = {
  basisModule: {
          attribuut: "hallo",
          method: function(){}
  },
  opmaakModule: {
          attribuut: 1,
          method: function(){}
  },
  printModule: {
          attribuut: true,
          method: function(){}
  }
}
```

Attributen en methods van één module worden ook via de namespace bereikt: `MijnApplicatie.opmaakModule.method()`.

Opletten met het gebruik van `this`: binnen een object is `this` altijd het object zelf:

```
var MijnApplicatie = {
  naam: "MijnApplicatie",
  getNaam:  function(){
                return this.naam;
                },
  basisModule: {
          naam: "basisModule",
          tekst: "hallo, ik ben ",
          spreek: function(){
                  return this.tekst + MijnApplicatie.getNaam() + "." + this.naam;
                  }
  }
}


MijnApplicatie.basisModule.spreek();
          //returnt "Hallo ik ben MijnApplicatie.basisModule"
```

Het is in principe voldoende de singleton in een javascript tag te zetten of te koppelen om hem als variabele te laden:

```
 <script type="text/javascript">
```

```
  var MijnApplicatie = {
        …
  }

MijnApplicatie.basisModule.spreek();  //voert uit
</script>
```

Een applicatie bevat dikwijls een method *init()* of *start()* om zaken uit te voeren (instellingen) met een uitstel, bijvoorbeeld als het document volledig geladen is.

```
<script type="text/javascript">
  var MijnApplicatie = {
        init: function(){
              var naam = document.getElementById('naam').value;
              …
              }
        …
  }
//============================
window.onload = function(){
        MijnApplicatie.init();
}
</script>
```

Voordelen Singleton:

- zeer eenvoudig aan te maken als *literal*
- overzichtelijk, gestructureerd, leesbaar
- bruikbaar als Namespace

Nadelen:

- alle properties en methods zijn bereikbaar. Geen mogelijkheid tot ***private*** variabelen en methods
- je hebt maar één instantie, dus maar één 'state' van je App.
  Als je meerdere 'states' nodig hebt is dit niet het geschikte pattern

Vooral om echte private members te kunnen gebruiken kunnen we de singleton nog een stap verder laten evolueren tot het Module Pattern.

### 16.2.2 Module pattern

Het *Module Pattern* geeft eveneens een *singleton*, maar die wordt niet als een letterlijk object aangemaakt, maar als de returnwaarde van een functie:

```
var MijnModule = function(){
  //private
  …
  return {
         //public
         …
  }
}()
```

Merk op:

- deze functie wordt onmiddellijk uitgevoerd: let op de dubbele haakjes achteraan ()
- de functie resulteert in een object, dus *MijnModule* is een object (singleton), geen functie

Het voordeel van dit pattern is dat de functie interne variabelen en functies kan bevatten die nooit in de singleton verschijnen: ze zijn ***private***.

De structuur van het *Module Pattern* wordt duidelijk in dit voorbeeld:

```
//Module
var mijnBank = (function() {
       //private
        var saldo = 100;
       function output() {
               return "saldo: " + saldo;
       };
       function reset(bedrag) {
               saldo = bedrag;
       }
      //public
      return {
               init: function(bedrag) {
                           reset(bedrag);
                        },
              toonSaldo: function() {
                        console.log(output());
               },
              interest: function(percent) {
                        saldo = saldo + saldo * percent;
               },
              toonSaldoNaInterest: function(percent) {
                        mijnBank.interest(percent); //oproep aan andere public
                        mijnBank.toonSaldo();
```

```
                }
            }
        }())
//Gebruik
mijnBank.init(1000);
mijnBank.toonSaldoNaInterest(0.2);
console.log(mijnBank.saldo); //undefined
```

Merk op:

- structuur van de Module:
  - de haakjes () aan het einde voeren de variabele *mijnBank* onmiddellijk uit zodat een object gereturned wordt
  - De `return` waarde is een object met de methods *init*, *toonSaldo*, *interest* en *toonSaldoNaInterest*
- Eerst voeren we de method *init()* uit die een startbedrag als saldo instelt. De **public** method *init()* heeft toegang tot de **private** method *reset()*
- Daarna voeren we *toonSaldoNaInterest()* uit die zelf twee andere **public** methods gebruikt: **we zijn verplicht ze op te roepen via hun Namespace**: ***mijnBank***.*interest()*
- Private members, zoals *saldo*, zijn niet toegankelijk, dus afgeschermd.

**Voordelen** van het Module Pattern tov van een Singleton

- Private members zijn afgeschermd
- Public members kunnen vrijelijk gebruik maken van private members en ander public members

**Nadelen**:

- Een public method is verplicht de Namespace van een ander public member te gebruiken (of `this`)

### 16.2.3 Revealing Module pattern

Het *Revealing Module Pattern* is een variant van het vorige die het nadeel ervan probeert op te lossen.
Het enige verschil is dat alle public methods eenvoudigweg doorverwijzen naar private members. Geen enkele public method roept een andere public method op, alle functionaliteit is private.

```
var mijnBank = (function() {
      //private
      var saldo = 100;
      function reset(bedrag) {
               saldo = bedrag;
      }
      function interest(percent) {
               saldo = saldo + saldo*percent;
      }
      function privToonSaldo() {
               console.log("saldo: " + saldo);
```

```
      }
      function privToonSaldoNaInterest(percent) {
               interest(percent); //oproep aan andere public
               privToonSaldo();
      }
      function privInit(bedrag) {
              reset(bedrag);
      }
      //public
      return {
               toonSaldo: privToonSaldo,
               toonSaldoNaInterest: privToonSaldoNaInterest,
               init: privInit
             }
  }())
//Gebruik
mijnBank.init(1000);
mijnBank.toonSaldoNaInterest(0.2);
console.log(mijnBank.saldo) //undefined
```

de public methods zijn gewoon pointers naar private methods. Dat betekent ook nog meer afscherming.

### 16.2.4 Factory

Het *Factory pattern* is een *aanmaak* patroon dat objecten of functies maakt volgens specificaties.
Veronderstel dat we een aantal functies hebben om verschillende soorten `input` elementen te maken:

```
function textbox(name){
  var input = document.createElement('input');
  input.name = name;
  input.type ="text";
  return input;
}
function checkbox(name){
  var input = document.createElement('input');
  input.name = name;
  input.type ="checkbox";
  return input;
}
function radioButton(name){
  var input = document.createElement('input');
  input.name = name;
```

```
  input.type ="radio";
  return input;
}
//toegepast:
var chkBox      = checkbox("junkmail");
var txtBox      = textbox("voornaam");
var radio       = radioButton("mayonaise");
```

Het is duidelijk dat die functies heel wat gemeenschappelijk hebben. Om nog meer soorten `input` elementen te maken moet je functies bijschrijven.

We kunnen dit beter vervangen door een ***Factory***:

```
function InputFactory (){
         this.createInputNode = function(type,name){
         //controle op type is nodig
           var input = document.createElement('input');
            input.name = name;
            return createInputType(type,input);
          }
 }
  function createInputType(type,input){
    switch(type){
      case 'text' :
           input.type ='text';
           return input;
      case 'radio' :
           input.type ='radio';
           return input;
       case 'check' :
           input.type ='checkbox';
           return input;
    }
  }
```

Dan kunnen we de Factory toepassen :

```
var fabriekje   = new InputFactory();
var chkBox      = fabriekje.createInputNode("check", "junkmail");
var txtBox      = fabriekje.createInputNode("text","voornaam");
var radio       = fabriekje.createInputNode("radio","mayonaise");
```

In Javascript bestaat een *Factory* uit een functie of een constructor die een method heeft om objecten aan te maken. Door middel van argumenten bepaalt deze method dus welk soort object aangemaakt moet worden en returnt dat object.

Een *Factory* wordt dus steeds op dezelfde manier opgeroepen.

Nog een voorbeeld: een VormFabriekje produceert verschillende vormen op aanvraag:

```
/* Driehoek Class*/
function Driehoek(basis,hoogte){
  this.basis = basis;
  this.hoogte = hoogte;
}
Driehoek.prototype = {
  type : "Driehoek",
  getOpp : function(){
  return 1/2 * this.basis * this.hoogte
  }
}
/* Rechthoek Class*/
function Rechthoek(lengte, breedte){
  this.lengte = lengte;
  this.breedte = breedte;
}
Rechthoek.prototype = {
  type : "Rechthoek",
  getOpp : function(){
          return  this.lengte * this.breedte
  }
}
/* Factory   */

function VormFabriekje(){}
VormFabriekje.prototype.maakVorm = function (type, settings){
  switch(type){
          case "driehoek":
                  var vorm = new Driehoek(settings.basis,settings.hoogte);
                  return vorm;
          break;
          case "rechthoek":
                  var vorm = new Rechthoek(settings.lengte,settings.breedte);
                  return vorm;
          break;
          }
  }
```

Bemerk dat de *Factory* functie hier aparte constructors gebruikt om de vormen af te leveren.

De eigenlijke aanmaak gaat dan als volgt:

```
/** VormFabriekje  toepassen */
  var fab = new VormFabriekje();
  var d   = fab.maakVorm("driehoek",{basis:5,hoogte:7});
  var r   = fab.maakVorm("rechthoek",{breedte:10,lengte:20});

  console.log(d.type + " met opp :" + d.getOpp());
  console.log(r.type + " met opp :" + r.getOpp());
```

### 16.2.5 Immediately Invoked Function Expressions

IIFEs zijn eigenaardige functie structuren die je soms tegenkomt in JS Modules.
Een voorbeeld van een IIFE:

```
(function (){
        var foo = 'bar';
        console.log(foo)}()); //toont 'bar'
```

Bemerk de haakjes rond de volledige functie declaratie: die zijn nodig want

```
function (){
        var foo = 'bar';
        console.log(foo)}();
```

geeft een syntaxfout, terwijl de volgende statements geen fouten geven:

```
var f = function (){
        var foo = 'bar';
        console.log(foo)}(); //toont 'bar'
```

en

```
function f (){
        var foo = 'bar';
        console.log(foo)}; //toont 'bar'
f();
```

en

```
var f = function (){
        var foo = 'bar';
        console.log(foo)}; //toont 'bar'
f();
```

maar deze laatste functies zijn niet meer anoniem.

Javascript laat niet toe dat je een functie declaratie maakt (het woord function als eerste) en die terzelfdertijd uitvoert. Als je alles in haakjes verpakt, maak je er een expressie van en die kan je wel uitvoeren.

Dit pattern wordt gebruikt om de **variabele scope tot een module te beperken**: vermits JS enkel *function scope* kent, is het niet mogelijk een variabele binnen één

bestand af te schermen van een andere. Laadt je bv twee scripts in met elke een *init()* functie, dan overschrijft de een de ander.

Aan een IIFE kan ook een ander object doorgegeven worden zoals een andere library of zelf het `window` object.

Veronderstel een bestand *shark.js*:

```
(function (window) {
    var me = {};
    me.naam = "shark";
    me.init = function () {
        console.log("hallo, ik ben " + this.naam);
    }
    window.shark = me;
}(window))
```

Bespreking:

- door dit bestand te laden wordt de anonieme functie automatisch uitgevoerd en die
- laadt het `window` object als argument en gebruikt het om
- een *shark* object als globale variabele aan te maken
- dit object bevat een *naam* property en een functie *init*

Die kunnen we dan uit voeren in de html pagina als

```
shark.init();          //toont 'hallo, ik ben shark'
```

Het is op dezelfde manier dat jQuery geladen wordt.

## *16.3 Minification*

*Kortere code = snellere code*.

Dit is eenvoudig gesteld de bedoeling van minification. *Minification* kan bereikt worden op verschillende niveau's:

- Het verwijderen van alle witruimte, line-breaks, commentaar en *pretty-print* (***uglify***)
- Het efficiënter herschrijven van de code, verwijderen van nutteloze code
- De lengte van de code reduceren door variabelenamen te vervangen door zo kort mogelijke variabelen: ***obfuscation*** (vb. strNaam -> sa1)
- Het versmelten van verschillende javascripts tot één bestand. Dit reduceert het aantal HTTP calls naar de server
- Het compresseren met ***gzip***

Dit versnelt in de eerste plaats de uitvoering doordat het bestand (veel) kleiner geworden is en de HTTP request dus sneller klaar is.

Er zijn verscheidene mogelijke **struikelblokken** die door een goede Minifier of door jouw keuzes opgevangen moeten worden:

- Sommige linebreaks dienen als Block-statement
- Vervangen van functies door top-level statements is niet altijd mogelijk

- *obfuscation* maakt het script onleesbaar (wat je misschien wil) maar ook on-debugbaar
- *obfuscation* is niet altijd mogelijk als je met een Framework werkt waar bepaalde objectnamen niet mogen gewijzigd worden (vb. dependency injection in AngularJS)
- een oudere browser kan niet altijd een gzip bestand lezen. Het is aan de server om dat op te vangen

Bij de meeste grote libraries (jQuery, etc..) kan je zowel een minified als een normale versie downloaden. Je gebruikt natuurlijk de minified versie in je productie-site.

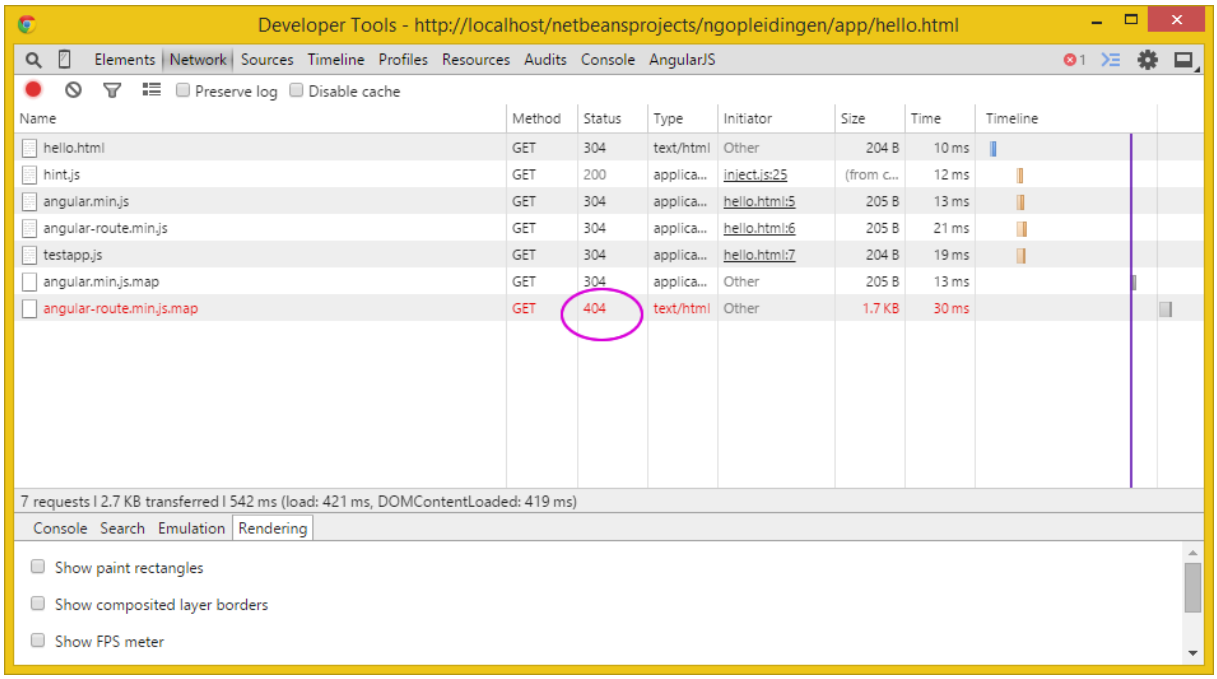
**Gooi zeker nooit je eigen originele script weg!**

Er zijn verschillende tools (sommige doen ook CSS en HTML) die je kunt gebruiken om een script te *minify-en*, enkele belangrijke zijn:

- De **Google Closure Compiler** (online UI en API)
  https://developers.google.com/closure/compiler/
- **JSMin** :
  http://www.crockford.com/javascript/jsmin.html
- De **YUI Compressor**:
  http://yui.github.io/yuicompressor/
- **JSCompress** (online)
  http://jscompress.com

### 16.3.1 Source map files

Het kan gebeuren dat je in de network tab van je *developer tool* een 404 boodscahp krijgt in de aard van "*jquery.min.map not found*".




*Waarom krijg ik een foutboodschap in mijn dev tool?*

Source maps zijn niet nodig voor de werking van het script.

Ze worden ook enkel opgehaald als je developer tools open staat. Maar als in Chrome dev tools de optie "*Enable javascript source maps*" (Settings van de dev tool) aangevinkt staat, zal die klagen als hij dit bestand niet vindt.

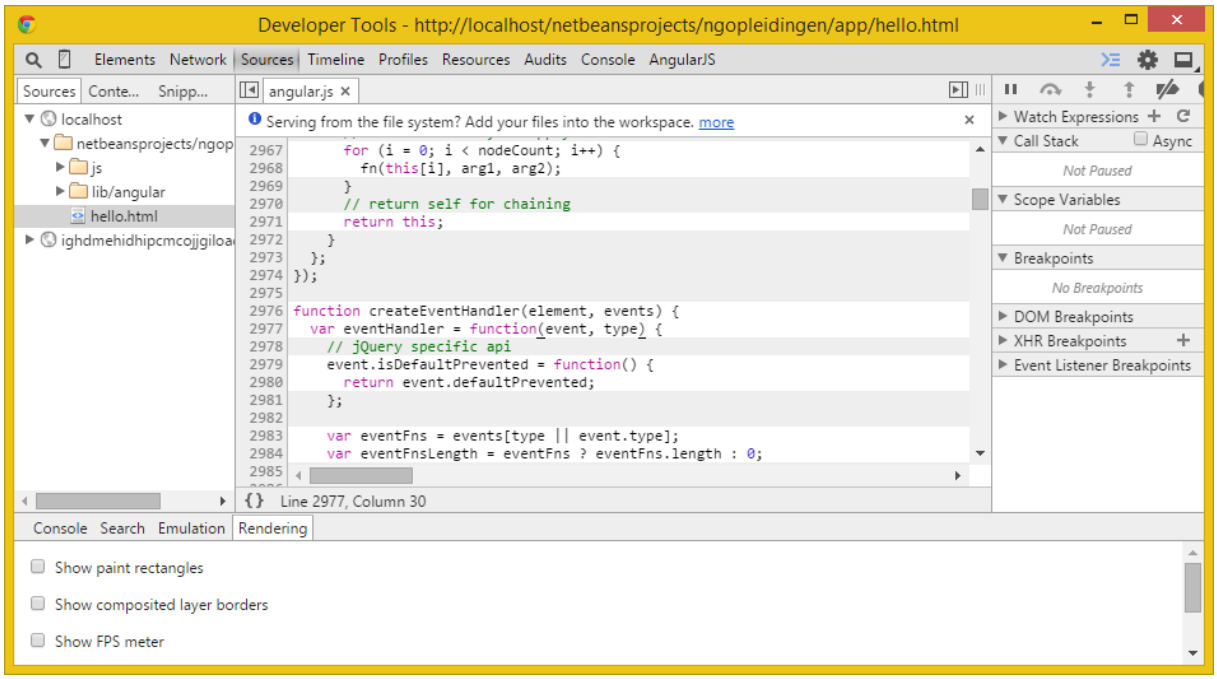
*Wat zijn Source map files?*

Een source map bestand is een JSON bestand met een *.map* extensie. Het bevat informatie over hoe een JS script geminified werd. Source map worden automatisch aangemaakt door de minification engine.

Met zo'n map kan je de originele code terug tevoorschijn halen.

*Kan ik er iets mee doen?*

Dank zij de source map kan je de geminificeerde code inspecteren alsof het het origineel betreft.




Angular code uit de minified versie alsof het niet geminifed was.

## 16.4 Modulair Javascript

Het lijkt er op dat de tijd van de grote "*doe-alles*" libraries, zoals jQuery, Prototype etc.., voorbij is. De nieuwe stijl libraries vallen terug op een kerntaak, waarbij je extra 'modules' kunt laden.

Een module kan een zeer specifiek onderdeel zijn of een algemene 'tool-belt' die in alle applicaties van pas komt.

Modules kunnen verplicht of optioneel zijn. Zijn ze verplicht, dan spreken we van ***dependencies*****:** de applicatie is er van afhankelijk, anders werkt hij niet. Het is ook mogelijk dat de volgorde van lezen van de dependencies belangrijk is.

In deze nieuwe wereld worden momenteel twee modulaire systemen gehanteerd: *AMD* en *commonJS* modules.

### 16.4.1 Asynchronous Module Definition (AMD)

AMD staat voor *Asynchronous Module Format* voor Javascript Modules.

Hoe hebben we tot nu toe zelf een Javascript Module (applicatie) geschreven? Waarschijnlijk op deze manier:

```
(function ($) {
  …
  init: function(){…}
  return {
          start: function(){this.init()}
          }
}(jQuery));
```

- via een onmiddellijk uitvoerende functie (Module Pattern of variant)
- die een globale variabele aanmaakt
- die uitgevoerd wordt via een `script` tag
- dependency van andere modules gebeurt via globale variabelen ($, jQuery)
- deze dependencies zijn fragiel want de volgorde van de `script` tags is cruciaal

De AMD API is een mechanisme om JS **modules** te definiëren en ervoor te zorgen dat dependencies correct geladen worden. Het is afgeleid van het *CommonJS Module* format maar is specifiek voor gebruik in de browser.

AMD

- het laadt modules **asynchroon**, dus enkel als ze gevraagd worden
- maakt **geen globale variabelen** omdat de module vervat zit in een *closure*
- door **string identificatie** kan dezelfde module meermaals opgeroepen worden
- een module kan eender wat bevatten, niet alleen javascript
- ondersteunt plugins
- elke module heeft zijn eigen bestand

AMD compatibele laders zijn bv. *Require, Dojo, Curl*.

**Een AMD module schrijven:**

via de functie `define()` wordt de module gedefiniëerd. Dit is geen *plain vanilla* Javascript functie maar wordt voorzien door de AMD Loader.

De syntax is

```
define(ID?, [dependencies]?, factory)
```

Bijvoorbeeld: een module bevat een simpel object zonder dependencies:

```
define( {
  color: "black",
```

```
  size: "unisize"
} );
```

Bespreking:

- deze module heeft geen `ID` string (gebruik van wordt afgeraden)
- heeft ook geen `dependencies`
- als `factory` bestaat  het uit een simpel object

Een variant: een *factory* function returnt een object zonder dependencies:

```
define( function(){
  //private functies voor instantiering hier
  return {
                color: "black",
                size: "unisize"
        }
});
```

Derde voorbeeld: een module met dependencies:

```
define(['jquery','huppeldepup'], function ($, hpdp) {
   return  { ...}
});
```

Bespreking:

- deze module heeft twee dependencies die geïdentificeerd worden door array van Strings (`['jquery','huppeldepup']`) en waarvan de returnwaarde als variabelen meegegeven worden aan de factory function `($, hpdp)`
- de module mag niet onmiddellijk uitgevoerd worden
- een module mag dus geen globale variabelen maken
- de module wordt uitgevoerd op het moment dat hij geladen wordt. Zijn returnwaarde is de module.

**Een AMD module gebruiken:**

Om nu een AMD module als dependency te gebruiken, neem je de functie `require` (ook geen *plain vanilla JS*, maar voorzien door de *loader*):

```
require(['jquery', 'app/tickets'], function ($ , tickets) {
          $('#test').html(hallo, mijn app hier');
});
```

Bespreking:

- deze functie lijkt erg op `define`, maar voert de code onmiddellijk uit
- de dependencies worden vermeld in een array van Strings: dat is hun bestandsnaam of de naam van een folder waarin ze zitten

- een anonieme functie bevat argumenten waarin de dependencies geladen worden

### 16.4.2 commonJS (CJS)

Het **commonJS** module formaat is in eerste instantie gericht op server-side Javascript.

In tegenstelling tot AMD worden modules niet asynchroon geladen, maar altijd en in de volgorde waarin ze gespecificeerd worden.

Een CJS module definiëren doe je met een `exports` object:

```
//spreekModule.js
exports.tekst = function(naam){ return "Hallo" + naam}
```

en

```
//welkomstModule.js
var spreekModule = require('spreekModule'); //dependency
exports.spreek = function(){return console.log(spreekModule.tekst('Jan'));}
```

Het `exports` object bevat dus alle properties en methods die de module biedt.

Hoe je dat object aanmaakt speelt geen rol.

Zoals het voorbeeld toont gebruik je een CJS module dmv de functie `require`:

```
var mijnModule =  require('mymodule');
```

De string *'mymodule'* is de bestandsnaam zonder `.js` of de naam van een module map.

commonJS modules worden gebruikt door **NodeJS**: alle modules die je dus installeert met *npm* hebben de CJS stuctuur en zo moet je ze dus gebruiken ook

# 17 MVC, SPA en REST

## 17.1 Wat is een SPA?

Een *Single-Page Application* (SPA), is een web applicatie/website die bestaat uit **één** HTML pagina.

Voorbeelden van bekende SPA's zijn Gmail, Google Drive, Facebook, Twitter, Flickr.

SPA's worden typisch gebruikt voor applicaties die intensieve interactie van de gebruiker verwachten terwijl de pagina lange tijd open blijft staan, bijvoorbeeld waarbij de gebruiker teksten schrijft/wijzigt, instellingen verandert, spelletjes speelt, etc…

In een SPA heb je volledige vrijheid van inhoud en opmaak en daardoor krijgt de gebruiker het gevoel dat hij in een native app bezig is, niet op een website.

SPA's moeten slechts éénmaal geladen worden vanaf een server – alle HTML, JavaScript, en CSS – en daarna niet meer. Ze zijn in staat om hun inhoud telkens opnieuw dynamisch aan te maken, in dezelfde of andere vorm (Views). Er is nadien wel degelijk communicatie met een server voor het het laden of schrijven van data. Deze communicatie is zeer dynamisch, en gebeurt als direct gevolg van de handelingen van de gebruiker. Voor een webclient (website) is de technologie gebaseerd op Ajax calls (XHR), de data zelf zijn in JSON formaat.

Dit kan enkel door het gebruik van een vorm van het MVC Pattern, waarbij de gegevens volledig gescheiden zijn van de Views waarin ze terechtkomen.

Hiervoor worden Javascript libraries gebruikt, die één of andere vorm van het MVC model ondersteunen (MVC, MVVM, MV*) : Knockout, Angular , BackBone, Ember, …

De serverkant wordt gereduceerd tot het leveren van het initiële pakket HTML, JS, CSS. De data worden geleverd door een RESTservice (zie verder). Men noemt dit een thin server architecture, door het afslanken van de serverside scripting: bijna alle scripting gebeurt clientside door Javascript.

## 17.2 MVC in Javascript

Het *Model-View-Controller* principe komt uit serverside scripting (Smalltalk, PHP, .Net, Java). Het stelt de scheiding van data (Model), user interface (de View) voorop met een derde component, de Controller, die de logica bevat en alles coordineert.

In Javascript is het niet zuiver bruikbaar en door de flexibiliteit van Javascript kan het geplooid worden naar veel varianten. Elke library/framework past het op zijn eigen manier toe, het mag dan ook geen dogma zijn.

### Het Model

Bevat de data in het applicatie geheugen. In een JS app is het model een ***Class*** die een *entiteit* bevat, bv. een *boek*. Een instantie wordt afgeleid van een Class Constructor (voorzien door de library) waarna je het aangepaste attributen geeft. De library constructor levert allerlei handige methods via inheritance.

Een *Backbone* voorbeeld:

```
var Film = Backbone.Model.extend({
        defaults:{
```

```
                        filmId:0,
                        titel: "",
                        beschrijving: "",
                        genre:"",
                        duur:"",
                        regisseur:"",
                        release:"",
                        foto:"img/filmplaceholder.jpg"
                    }
});
```

Een Model laadt zijn gegevens van een REST server, wijzigingen worden ook naar de server geschreven.

Een Model is ***observable*** en zal zijn observeerders (de Views) **verwittigen** (notify) als hij veranderd (geupdate) is.

Het Model is zich verder onbewust van Controllers of Views.

Veel libraries laten je toe Models te groeperen, vb in BB hebben we Collections. Op die manier kunnen we de groep gebruiken ipv van individuele Model instanties.

#### De View

Een View is de visuele weergave van een Model of een groepModels.

Een View observeert het Model en wordt verwittigd als die veranderd is. Views geven de gebruiker dikwijls de mogelijkheid de gegevens te wijzigen (edit, delete, create).

De taak om het Model daaraan te passen is voor de Controller, niet voor de View. Eenmaal het Model gewijzigd wordt de View daarvan verwittigd en zal die zichzelf opfrissen.

Views in webpagina's bestaan typisch uit templates of uit scripts die HTML produceren.

#### De Controller

De Controller coordineert de verbinding tussen View en Model. Dit is het punt waar de verschillende JS Frameworks het meest van elkaar verschillen. Sommige frameworks hebben echte Controllers die alle verantwoordlijkheid dragen, in andere wordt de Controller rol gedeeltelijk overgenomen door Models en Views zelf (BackBone).

## *17.3 REST*

REST is een afkorting van *Representational State Transfer*.

Bij REST roept een applicatie de diensten op van een andere applicatie via het HTTP protocol.

- De applicatie die de diensten oproept heet de *REST client*.
- De applicatie die de diensten aanbiedt heet de *REST service*.

Een browser is een *client* en de website is een *service*. Maar ook andere applicaties roepen mekaars diensten.
Een client van een reisbureau (geschreven in Java) roept de diensten op van een service van een vliegtuigmaatschappij (geschreven in .net).

REST is gebaseerd op volgende principes.

- De service identificeert de **entities** die hij aanbiedt met URI's.
- De client doet *Requests* naar deze URI's en duidt met de HTTP *method* (`POST`, `GET`, `PUT`, `DELETE`) aan welke **handeling** (toevoegen, lezen, wijzigen, verwijderen) hij op de entity wil uitvoeren.
- Services en clients wisselen data over entities uit in meerdere formaten
  - HTML als de client een browser is
  - XML als de client geschreven is in Java, .net, Cobol, ...
  - JSON als de client geschreven is in JavaScript

- REST gebruikt de HTTP status code van de *Response* om aan te geven of de request wel of niet correct verwerkt werd.
- De data die de service aanbiedt, bevat URI's waarmee de client terug diensten aan de service kan vragen.
  Bijvoorbeeld, de browser krijgt op de URI `/filialen` data over alle filialen.
  Deze data bevat per filiaal een hyperlink met de URI waar je de detail informatie van dat ene filiaal vindt.
  De gebruiker kan met zijn browser deze hyperlink volgen om die detailinformatie te zien. De gebruiker kreeg de URI met deze detailinformatie aangereikt en hoefde die niet zelf te verzinnen.
- De service is *stateless*. Dit houdt in dat de service geen data bijhoudt zoals HTTP *Session variabelen*. Dit principe is moeilijk haalbaar in een service waarvan de clients browsers zijn, maar is wel haalbaar in een service waarvan de clients geen browsers zijn.

### 17.3.1 Entities identificeren met URI's

Elke entity én elke verzameling entities krijgt een URI.

In theorie is deze URI vrij te kiezen. De meeste websites gebruiken echter **betekenisvolle URI's** die aan mensen én zoekrobots verduidelijkt welke data de URI aanbiedt.

De opbouw van een betekenisvolle URI verschilt volgens de data die de URI voorstelt. Je maakt hierbij onderscheid tussen

- de verzameling met alle entities van hetzelfde type

| Voorbeeld URI | Voorgestelde entities |
|---|---|
| `/filialen` | Alle filialen |
| `/werknemers` | Alle werknemers |

- Één entity, aangeduid met zijn unieke identificatie

| Voorbeeld URI | Voorgestelde entity |
|---|---|
| `/filialen/3` | Het filiaal met het nummer 3 |
| `/werknemers/6` | De werknemer met het nummer 6 |

  Variabele onderdelen in de URI (`3` in de URI `/filialen/3`) heten path variabelen. Je stelt bij REST een unieke identificatie **niet** voor als een request parameter (`/filialen?id=3`)

- Een verzameling entities, aangeduid met een groepsnaam

| Voorbeeld URI | Voorgestelde entities |
|---|---|
| `/filialen/hoofdfilialen` | Alle hoofdfilialen |
| `/filialen/bijfilialen` | Alle bijfilialen |

- Een verzameling entities die je zoekt op andere zoekcriteria

  Je zoekt soms een verzameling entities met andere zoekcriteria dan hier boven. Je gebruikt dan request parameters.
  Syntax URI: `/meervoudsvormentities?requestparameters...`

| Voorbeeld URI | Voorgestelde entities |
|---|---|
| `/filialen?beginnaam=de` | Filialen waarvan de naam begint met de |
| `/werknemers?vanafWedde=2000` | Werknemer met een wedde >= 2000 |

  Je kan request parameters ook gebruiken om de sorteervolgorde aan te geven

| Voorbeeld URI | Voorgestelde entities |
|---|---|
| `/filialen?sort=Naam` | Alle filialen, gesorteerd op naam |
| `/filialen?sort=aantalWerknemers` | Alle filialen, gesorteerd op aantal werknemers |

- Associaties tussen entities

  Je kan een associatie tussen entities uitdrukken met een URI.
  Syntax: `/meervoudsvormentities/uniekeidentificatie/naamvandeassociatie`

| Voorbeeld URI | Voorgestelde entities |
|---|---|
| `/filialen/3/werknemers` | De werknemers van het filiaal met het nummer 3 |
| `/werknemers/6/filiaal` | Het filiaal van de werknemer met het nummer 6 |

### 17.3.2 HTTP methods als handelingen

De client doet een request naar de URI die een entity of een verzameling entitites voorstelt. De client duidt met de HTTP method (`GET`, `POST`, `PUT` of `DELETE`) van de request aan *welke handeling* hij wil uitvoeren op deze entity of verzameling entities.

- De client **voegt** een entity **toe** met HTTP `POST`

  De request body bevat de data van de toe te voegen entity.

  Voorbeeld: de client doet een `POST` request naar `/filialen` om een filiaal toe te voegen. De request body bevat de data van het toe te voegen filiaal.

- De client **leest** één of meerdere entities met HTTP `GET`

  De client doet een `GET` request naar de URI die de entity of verzameling entities voorstelt. Hij krijgt een response terug met de data van die entity of entities.

Voorbeelden:

  - De client doet een `GET` request naar `/filialen` om alle filialen te lezen.
  - De client doet een `GET` request naar `/filialen/3` om filiaal 3 te lezen.

- De client **wijzigt** een entity met HTTP `PUT`

  De client doet een `PUT` request naar de URI die de te wijzigen entity voorstelt. De request body bevat de te wijzigen data van de entity.

  Voorbeeld

  - De client doet een `PUT` request naar `/filialen/3` om filiaal 3 te wijzigen. De request body bevat de te wijzigen filiaaldata.

- De client **verwijdert** een entity met HTTP `DELETE`

De client doet een DELETE request naar de URI die hoort bij de te verwijderen entity.

Voorbeeld

- De client doet een DELETE request naar /filialen/3 om filiaal 3 te schrappen.

### 17.3.3 Formaten

De client kan bij een GET request op drie manieren het gewenste formaat (HTML, XML, JSON, ...) aangeven waarmee hij de data in de response wilt ontvangen.

- Het gewenste formaat aangeven in de request header Accept

| Request header Accept | Data formaat response |
|---|---|
| text/html | HTML |
| application/xml | XML |
| application/json | JSON |

- Het gewenste formaat aangeven in een extensie op de URI

| URI extensie | Data formaat response |
|---|---|
| Geen | HTML |
| .xml | XML |
| .json | JSON |

- Het gewenste formaat aangeven in een query parameter

| Query parameter | Data formaat response |
|---|---|
| Geen | HTML |
| xml | XML |
| json | JSON |

Voorbeeld
de client doet een GET request naar /filialen/3 met Accept:application/json
De response bevat dan het filiaal in JSON formaat. De response bevat altijd een header Content-type. Hier zal deze application/json bevatten.

Bij een POST request of een PUT request bevat de request body de toe te voegen of te wijzigen entity. De client geeft dan in de request header Content-Type aan in welk formaat deze entity is uitgedrukt (bvb. application/xml, als de entity is uitgedrukt in XML).

### 17.3.4 Response Status code

De response status code geeft aan of de request correct verwerkt werd:

De meest gebruikte response status codes

- 200 (OK)
  De request is correct verwerkt.
- 201 (Created)
  De client doet een POST request om een entity toe te voegen.
  De response status code bevat 201.
  De response header Location bevat de URI van de nieuwe entity.
- 400 (Bad request)
  De client doet een POST request of een PUT request.
  De entity, die hij meestuurt in die request, bevat verkeerde data (bvb. een werknemer met een negatieve wedde).

De response status code bevat `400`.
De response body kan meer informatie over de fout bevatten.

- `401 (Unauthorized)`
  De client doet een request op een beveiligde URI. De client heeft zich nog niet geauthentificeerd, of heeft een verkeerde gebruikersnaam of paswoord meegegeven. De response status code bevat `401`.
- `403 (Forbidden)`
  De client doet een request op een beveiligde URI. De client heeft zich correct geauthentificeerd, maar heeft niet de juiste rechten om een request te doen op die URI. De response status code bevat `403`.
- `404 (Not found)`
  De client doet een request naar een URI die de service niet kent.
- `405 (Method not allowed)`
  De client doet een request op een URI.
  De service laat de HTTP method van die request niet toe op die URI.
  Voorbeeld: een `DELETE` request naar `http://voorbeeld.com/filialen`
  De service laat enkel `GET` requests toe op deze URI.
  De response status code bevat code `405`. De response header `Allow` bevat de lijst met HTTP methods die de URI wél ondersteunt.
- `406 (Not acceptable)`
  De service ondersteunt de waarde in request header `Accept` niet.
  Voorbeeld: de request header `Accept` bevat `application/xml`.
  De service kan echter enkel responses in JSON formaat produceren.
- `409 (Conflict)`
  De request probeert een entity in een onmogelijke toestand te plaatsen.
  Voorbeeld: een `DELETE` request naar `http://voorbeeld.com/filialen/3`
  Filiaal 3 heeft echter nog werknemers en kan men dus niet verwijderen.
  De response status code bevat `409`.
- `415 (Unsupported Media Type)`
  Een request bevat data in een formaat aangegeven in de header `Content`.
  De service ondersteunt dit formaat echter niet.
  Voorbeeld een `POST` request bevat data in JSON formaat.
  De service kan echter enkel data in XML formaat verwerken. De response status code bevat `415`.
- `500 (Internal server error)`
  De client doet een request. De service heeft interne problemen om die request te verwerken (de database is bijvoorbeeld uitgevallen).
  De response status code bevat `500`.

# 18 Javascript Libraries

Javascript *libraries* (ook *frameworks* genoemd) bieden soms de oplossing voor de overstreste programmeur: sommige bevatten een arsenaal aan code voor alle mogelijke situaties – een toolbox dus (het beste voorbeeld hier is *jQuery*). Andere zijn dan weer gemaakt met een zeer specifiek doel voor ogen, bv. het koppelen van andere libraries *on demand* met *require.*

Als je echter denkt dat je geen Javascript meer moet kennen met een Library, dan ben je fout bezig… in de praktijk is een gedegen kennis van JS noodzakelijk: je kunt een library pas gebruiken als je de standaardtaal kent en begrijpt wat je doet. Wat voor nut heeft een methods zoals *$.extend()* als je niet met objecten kan werken?

## *18.1 Een JS library gebruiken*

met een JS library

- is JS toepassen gemakkelijker, je moet enkel de syntax van de library onder de knie krijgen
- hoef je het wiel niet telkens opnieuw uit te vinden
- leun je op zeer degelijke code gemaakt door ervaren programmeurs
- worden gekende problemen (cross-browser) opgelost
- ben je steeds up-to-date

De grote algemene libraries hebben een volledige **kernbibliotheek** die *DOM*, *Events*, *Ajax* behandelt. Daarmee schrijf je code voor je de meest gebruikte zaken zoals formvalidatie, dhtml, cookies, web 2 apps, etc...
Naast die *code base* vindt je dikwijls extra functionaliteit onder de vorm van *add-on's* of *plug-in's*.

Een library gebruiken doe je op twee mogelijke manieren:

- door hem te downloaden en op je eigen site te plaatsen
- door hem online te gebruiken, op een CDN (Code Developers Network)

Op productiesites gebruik je best de tweede manier.

De meeste libraries kunnen gedownload worden in één of andere vorm van *compression*: indien *minified* is alle witruimte en line-feeds eruit gestript.

## *18.2 Algemene libraries*

Er zijn tegenwoordig zeer veel JS libraries, waarvan sommige algemeen zijn, andere een bepaalde architectuur hebben, nog andere zeer specifiek doeleinden hebben. We kozen er de interessantste uit, we beginnen met de algemene.

### 18.2.1 JQuery

**JQuery** (2006, John Resig) is momenteel zonder twijfel de belangrijkste JS library: in 2013 90+% marktaandeel.
Het is een algemene Javascript library die het werken met JS veel bondiger maakt. Het lost quasi alle browserproblemen op en er bestaan zeer veel plug-ins voor.
Een codevoorbeeld:

```
$("a")
    .filter(".clickme")
      .click(function(){
        alert("You are now leaving the site.");
      })
    .end()
    .filter(".hideme")
      .click(function(){
        $(this).hide();
        return false;
      })
    .end();
```

- korte code, principe van 'chaining', goede node selection
- heel kleine base code (14Kb compressed, 54Kb minified)
- CSS3, XPath expressies
- extra's via plug-ins
- gebruikt door Google, Amazon, Intel
- documentatie OK hoewel summier
- zeer vlotte Ajax methodes

De grote sterkte van jQuery zit in een aantal features, bv. de uitzonderlijke goede *node selection*, het *'chaining'* concept, de vlotte Ajax toepassing én de constante aanpassingen en perfectioneringen.

De vdab cursus "jQuery" leert je ermee werken, een *must*.

jQuery.com

### 18.2.2 JQuery UI

De **JQuery UI** library voorziet talloze User Interface widgets en tools om webpagina's dynamisch op te maken: tabs, accordion, buttons, etc… De jQuery UI is uiteraard een extensie van jQuery (je hebt de basis lib nodig).

### 18.2.3 Mootools

Mootools (2006) is ook een populaire lib die robuste OO programmering toelaat, je zeer flexibel  met de DOM laat werken en Ajax functionaliteit biedt. Ook chaining is ingebouwd.

mootools.net

### 18.2.4 Prototype en script.aculo.us

**Prototype** (2005) is de meest populaire JS library die vooral de kerntaken zoals DOM, Event handling en Ajax voor zijn rekening neemt.

Prototype werkt nauw samen met script.aculo.us dat uitmunt in dhtml en visuele effecten.
In deze voorbeeldcode bemerk je het gebruik van de $ functie, die elementen selecteert en de $$ functie die elementen op CSS selecteert:

```
$$("div.speciaal").map(Element.hide);
$$("a[href='http://']").each(function(element)
  {
  Event.observe(element, 'click', openNewWindow);
  }
 );
```

- hoge kwaliteit, zeer volledige code base, frequente updates
- uitgebreide extension een aanpassing van javascript, $ , $$ , 'classes', is dit nog JS?
- *Ruby-on-Rails* bevat Prototype en scriptaculous
- klein: prototype 54Kb, scriptaculous 32Kb
- object georiënteerde syntax
- uitgebreide documentatie

prototypejs.org

### 18.2.5 YUI

De **Yahoo! User Interface Library** (2006) is een set van JS tools en controls en is vrij te gebruiken als open source BSD license.
In het onvolledige voorbeeldje zie je hoe de code steunt op het gebruik van Namespaces:

```
var myAnim = new YAHOO.util.Anim(
  'test',
  { height: {to: 10} },
  1,
  YAHOO.util.Easing.easeOut );


myAnim.animate();
```

- geen omvorming van Javascript, leunt het meeste aan bij JS standaard, lichte uitbreiding en cross-browser
- niet als één library maar als losse modules: AJAX, DOM manipulation, drag-and-drop, event handling ,
- beperkte toolset (Calendar, Slider, Menu, AutoComplete, Tree View, Container class), groeiend
- object georiënteerde syntax
- uitstekende documentatie
- gebruikt door Yahoo, Flickr, Target, Slashdot

http://yuilibrary.com/

## 18.3 Resource loaders

*JS Resource Loaders* zijn JS libraries die geen andere functie hebben dan te bepalen welke andere libraries/stylesheets je laadt en in welke volgorde.

Sommige frameworks (YUI) hebben een eigen ingebouwde loader, maar er zijn een aantal aparte loaders.

*Heb je een Script Loader nodig om je pagina sneller te laden?*

- Meestal NIET: indien je slechts enkele, kleine scripts hebt en er geen dependency problemen zijn, dan is het antwoord NEEN.
- indien je voorwaardelijk moet beslissen of/welke library je wil gebruiken, of je hebt een *polyfill* nodig om een modern HTML5 feature te emuleren, dan JA
- bestaat je applicatie uit verschillende, grotere *third-party libraries* met eigen dependencies, dan is het antwoord JA

In de meeste gevallen zal het gebruik van een CDN en concateneren van je scripts veel meer snelheid opleveren dan het gebruik van een script loader.

### 18.3.1 RequireJS

*RequireJS* is een *resource loader* (15Kb) die je in staat stelt asynchroon andere libraries te laden. Het past het AMD (*Asynchronous Module Definition*) principe toe. *Require* kan zomaar in de browser gebruikt worden maar veel andere libraries gebruiken het ook. RequireJS kan complexe dependencies zonder moeite aan.

http://requirejs.org/

## 18.4 Javascript MVC Frameworks

Deze libraries laten je toe het *Model-View-Controller* principe te gebruiken in je client-side app. Ze zijn vooral toepasselijk bij applicaties waar veel manipulatie gebeurt in de browser: selecteren, updaten, schrijven, terwijl de server enkel scripts en gegevens voorziet. Voorbeelden zijn webmail systemen, Google docs, etc…

Hier worden een scripts en gegevens eenmalig opgehaald bij de server waarna het meeste werk gebeurt in de browser. Nieuwe pagina's ophalen bij de server gebeurt niet veel. Data worden teruggestuurd naar de server met Ajax.

Je kan uiteraard al die code zelf schrijven, maar deze libs geven jou de tools.

Hier een overzicht van enkele.

### 18.4.1 Backbone

*Backbone* is een heldere library die het MVC model toepast ( hier *Model – Collection – View*). Het script fungeert als Controller en laadt bepaalde Views (die in script tags in de html vervat zijn) als reactie op events. *Backbone's* controllers zijn in staat te reageren op user-friendly URL fragmenten (routing). *Backbone* steunt op *jQuery* en *Underscore*

http://backbonejs.org/

### 18.4.2 Ember

*Ember* is ook een MVC framework dat templates, routing, states ondersteunt. Ember gebruikt daarbij *handlebars* in zijn Views.

http://emberjs.com/

### 18.4.3 Knockout

*Knockout* (KO) is een data-binding library die het MVVM (*Model – View – View Model*) principe toepast, een event-driven afgeleide van MVC. KO gebruikt *data binding:* deze techniek maakt heel wat code (gebruikt om gegevens in de HTML te plaatsen) overbodig.
*Knockout* heeft geen *dependencies*. Het is geïntegreerd met Microsoft .Net producten.

http://knockoutjs.com/

### 18.4.4 AngularJS

AngularJS (by Google) is een volledig framework dat ook het MVC model toepast. Het is speciaal geschikt voor CRUD applicaties. Ook Angular gebruikt *data binding* om de templates te vullen, maar is groter, vollediger maar ook complexer dan KO.

http://angularjs.org/

## *18.5 Mobile JS libraries*

Deze groep spitst zich toe op toepassingen op mobiele toestellen waar andere events gelden en de OS niet altijd garandeerd dat alle JS werkt.

### 18.5.1 JQuery Mobile

De **JQuery Mobile** library is toegespitst op het opmaken van websites voor mobiele toestellen. Ook jQuery Mobile is afhankelijk van de kern library.

## *18.6 Utilities*

Libraries met welbepaalde doeleinden:

### 18.6.1 Modernizr

*Modernizr* is een kleine library die enkel dient om HTML5 en CSS3 ondersteuning te detecteren in browsers. Het enige wat het doet is een aantal `class` waarden in het `html` element plaatsen. Aan de hand hiervan kan je dan je CSS aanpassen of een script mee bouwen. Het lost een probleem dus niet op.
*Modernizr* wordt echter door heel wat CSS Frameworks als *testing suite* gebruikt.

http://modernizr.com/

### 18.6.2 Underscore

*Underscore* is in essentie een verzameling van utitlity functies, allemaal verzameld onder het _. object. Je zou nu denken dat je alles al hebt in *jQuery*, maar toch is deze kleine lib populair en wordt ze dikwijls gebruikt door andere libraries:

underscorejs.org

### 18.6.3 Lo-Dash

*Lodash* is eveneens een *utility-belt* onder de Namespace _. Bijna identiek aan Underscore, maar heeft ook AMD support, customisation, grote snelheid en een nog grotere verzamling functies. Daardoor waarschijnlijk superieur, wint snel aan populariteit!

lodash.com

## *18.7 Javascript Servers*

Een klasse apart zijn de serverpakketten die in JS geschreven zijn. Daar is vooral NodeJS belangrijk.

### 18.7.1 Node.js

*Node.js* is geen gewone JS library, het is in essentie een **platform** waarop applicaties geschreven worden. Die applicaties zijn dus zowel serverside als client-side. Het unieke eraan is dat het werkt op Javascript. Node werkt op de supersnelle *Google V8 Javascript engine*.

Node is *event-driven* en kan daardoor grote hoeveelheden gelijktijdige connecties aan. Dit maakt het ideaal om als webserver te fungeren voor bv. apps die met **Websockets** werken zoals een chat applicatie.

Je schrijft dus zowel de server-side als de client-side van je applicatie in Javascript. Er bestaan reeds zeer veel Modules voor Node, allemaal in het comonJs formaat. Je installeert ze gemakkelijk met *npm*, De *Node Package Manager*, *npm*.

Een voorbeeld van zo'n module is ***Express*** is een webserver overlay. Het helpt je een MVC Framework op te zetten, templates te gebruiken, database connecties te maken, cookies en sessions beheren, routes te definiëren.

## *18.8 Testing libraries*

TDD (test driven development) wordt belangrijker. De volgende libraries/frameworks zijn testers:

- QUnit (http://qunitjs.com/)
- Jasmine https://github.com/pivotal/jasmine
- js-test-driver (https://code.google.com/p/js-test-driver/)
- Karma

# 19 Bijlage A: het canvas element: objecten, properties, methods

canvas is een nieuw HTML element dat deel uitmaakt van de HTML5 specificatie. Het wordt vandaag enkel ondersteund door FireFox, Safari en Chrome.

canvas gebruiken heeft enkel nut als je er op gaat tekenen met Javascript. Als statisch element kan je er weinig mee doen.

Hieronder een opsomming van de voornaamste properties en methods van canvas en zijn afgeleide objecten.

canvas ondersteunt slechts één primitieve vorm, de rechthoek, andere vormen moet je zelf met *Paths* tekenen.

| object | property/method() | return type | beschrijving |
|---|---|---|---|
| canvas | | DOM element | het canvas element |
| | getContext('2d') | CanvasRenderingContext2D | object: een vlak 2D oppervlak met origines 0,0 in de linkerbovenhoek en *x* waarden voor de horizontale dimensie, *y* waarden voor de verticale dimensie |
| 2D context | | object | de returnwaarde van getContext('2d') |
| | *// algemeen* | | |
| | canvas | DOM element | geeft opnieuw het canvas waarvan de context afgeleid is |
| | save() | void | |
| | restore() | void | |
| | *// kleuren* | | |
| | strokeStyle | String | Een geldige CSS kleur of<br>een CanvasGradient object of<br>een CanvasPattern object |
| | fillStyle | String | opvulkleur. idem als hierboven |
| | *// lijn stijlen* | | |
| | lineWidth | number | lijndikte |
| | lineCap | butt \| round \| square | soort einde aan einde lijn |
| | lineJoin | bevel \| round \|miter | soort hoek als twee lijnen samenkomen |
| | miterLimit | | |
| | *// tekst* | | |
| | font | String | een geldige CSS string zoals<br>*"italic 24pt Arial, sans serif"* |
| | fillText() | | tekent de tekst met een opgevulde letter |

| object | property/method() | return type | beschrijving |
|---|---|---|---|
| | strokeText() | | tekent enkel de randen van de tekst |
| *// rechthoeken* | | | |
| | clearRect() | void | wist de rechthoekige vorm en maakt die transparant |
| | fillRect () | void | tekent een opgevulde rechthoek |
| | strokeRect () | void | tekent enkel de rand van de rechthoek |
| | | | |
| *// Paths* | | | |
| | beginPath() | void | start een nieuw path (een lijst van punten) |
| | closePath () | void | sluit het path af en verbindt het laatste punt met het eerste |
| | fill () | void | vult de oppervlakte van het path met de fillStyle |
| | stroke() | | tekent de lijnen van het path |
| | clip() | | |
| | moveTo() | | zet het huidig punt zonder daarbij een lijn te trekken |
| | lineTo() | | zet het huidig punt en trekt er een lijn naartoe van het vorige |
| | bezierCurve() | | tekent een beziercurve |
| | arcTo() | | tekent een boog naar een punt vanaf het huidige |
| | arc() | | tekent een boog of een cirkel |
| | rect() | | tekent een rechthoekig path naar het *path* |
| | | | |

# 20 Bijlage B: Terminologie

Verklarende woordenlijst:

**Containing block**

Is de box waarin een andere box leeft en ten opzichte waarvan het een positie krijgt. De *containing block* is niet noodzakelijk de onmiddellijke *parent* van het *child* block, bijvoorbeeld bij *absoluut gepositioneerde* boxes

**Deprecated**

Een *deprecated* element of attribuut betekent dat het element/attribuut in deze DTD nog legaal is, maar het gebruik wordt nu reeds afgeraden. In een volgende versie van de DTD zal het zeker geschrapt worden. Het is dus nu al beter dat je uitkijkt naar een alternatief

**MIME**

Multipurpose Internet Mail Extensions. MIME is oorspronkelijk ontworpen om meerdere email formaten via het internet te laten sturen, maar is uitgebreid**.** Het omvat nu infeite het *Internet media Type* waarin gezegd wordt welk soort inhoud een internet document bevat samen met de karakterset waarin het gecodeerd is.

**Parsing**

Het analyseren van een input stroom, in dit geval het analyseren van een ingelezen html document en het subsequent tonen (*rendering*) van dat document op het medium (scherm). Een **parser** is het computerprogramma die deze taak uitvoert, in dit geval de browser.

**DTD**

Document Type Defintion. Is een tekstdocument uit de xml wereld waarin de juiste structuur van een xml document vastgelegd wordt. Het verklaart welke elementen en attributen er mogen gebruikt worden, hoe ze genest mogen worden, of het lege elementen zijn, wat de toegelaten waarden voor de attributen zijn, etc…

**Well-Formed**

Een document is Well-formed als zijn elementen juist genest zijn en alle elementen ook juist afgesloten zijn. Een term uit de xml wereld.

**User-Agent**

Een softwareprogramma dat een SGML (html/xhtml) document *parsed*. In het geval van het Web wordt daarmee een *browser* bedoeld. Maar ook GSM software, search engine crawlers en braillelezers zijn User Agents.

**Stack**

De verzameling van software tools die je nodig hebt voor een project. Je moet ze kunnen installeren en er vlot mee werken.

# 21 Bijlage C: Internet referenties

| url | Website |
|---|---|
| de ondersteunende website | |
| www.ict.teno.be/oostende [handleidingen] | css – xhtml – dom - javascript |
| Javascript & DOM specificaties, referenties en validators | |
| developer.mozilla.org/en/docs/Javascript | MVC: Javascript |
| www.w3.org/TR/DOM-Level-2-HTML/html.html | Document Object Model HTML |
| www.w3.org/standards/techs/dom#w3c_all | DOM Modules overview |
| www.w3.org/TR/selectors-api2/ | Selectors API |
| www.json.org | JSON |
| www.jslint.com | JSLint, the Javascript verifier |
| www.whatwg.org/specs/web-apps/current-work/multipage/index.html | HTML5 specificatie |
| andere | |
| www.quirksmode.org | javascript - dom |
| anysurfer.be | Toegankelijkheid |
| | |
| JS libraries | |
| developer.yahoo.com/yui | YUI |
| jquery.com | jQuery |
| nodejs.org | NodeJS |

# 22 COLOFON

**Sectorverantwoordelijke:** Ortaire Uyttersprot

**Cursusverantwoordelijke:** Jean Smits

**Didactiek en lay-out:** Jan Vandorpe

**Medewerkers:** Jan Vandorpe

**Versie:** april 2015

**Nummer dotatielijst:**