



JavaScript

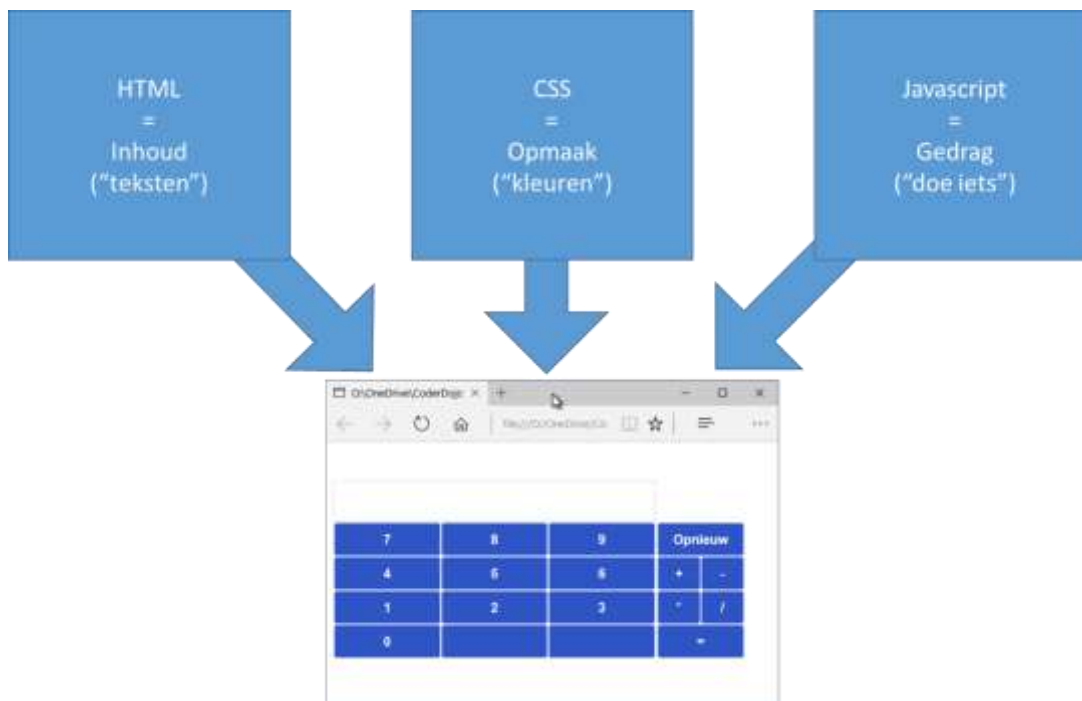
Met Javascript breng je je browser tot leven. Javascript zorgt ervoor dat teksten en plaatjes in een webpagina kunnen bewegen. Javascript laat iets op een webpagina gebeuren als jij wat intikt of met je muis beweegt. En zo kan je ook spelletjes maken in Javascript.

Een rekenmachine

We gaan een rekenmachine maken in Javascript. Daarvoor hebben we drie dingen nodig:

- HTML
- CSS
- Javascript

Samen ziet dat er zo uit.



In dit plaatje zie je hoe HTML, CSS en Javascript samenwerken om in de browser wat te laten gebeuren. Alles wat je in Javascript leert, heeft deze drie dingen nodig. Ze zitten ieder in hun eigen tekstbestand.

We gebruiken Visual Studio Code om te programmeren. Als dit programma nog niet geïnstalleerd is, kan je het hier downloaden: <https://code.visualstudio.com/>

De belangrijkste commando's in Visual Studio Code zijn:

Toets	Menu	Waarvoor
CTRL+N	File > New File	Maak een nieuw bestand
CTRL+S	File > Save	Bewaar een bestand
CTRL+K CTRL+O	File > Open Folder	Bekijk de inhoud van een map
CTRL+K S	File > Save All	Alle bestanden bewaren

HTML

We beginnen met de HTML.

1. Start Visual Studio Code op je computer
2. Open het bestand **Javascript\Opdrachten\Rekenmachine\rekenmachine.html**
3. Kopieer en plak de onderstaande tekst naar Visual Studio Code.

```
<html>
<head>
  <link href="rekenmachine.css" rel="stylesheet" type="text/css" />
</head>
<body>
  <div id="uitkomst"></div>
  <div id="som"></div>
  <table id="getallen">
    <tr><td>7</td><td>8</td><td>9</td></tr>
    <tr><td>4</td><td>5</td><td>6</td></tr>
    <tr><td>1</td><td>2</td><td>3</td></tr>
    <tr><td>0</td><td></td><td></td></tr>
  </table>
  <table id="functies">
    <tr><td colspan="2">Opnieuw</td></tr>
    <tr><td>+</td><td>-</td></tr>
    <tr><td>*</td><td>/</td></tr>
    <tr><td colspan="2" id="bereken">=</td></tr>
  </table>
  <script src="rekenmachine.js" type="text/javascript"></script>
</body>
</html>
```

4. Sla het bestand op. Gebruik hiervoor het menu **File > Save**, of druk op **CTRL+S**.

Als je dit bestand opent in een browser, dan ziet er een beetje raar uit. Ga maar eens via de Verkenner naar de plek waar je net het bestand hebt bewaard. Klik met rechts op het bestand en kies open.

TIP! Je kunt in Visual Studio Code ook een webserver gebruiken. Een webserver is een programma dat HTML pagina's naar de browser kan sturen. Je installeert hem zo:

Druk op CTRL+P en type

```
ext install ritwickdey.liveserver
```

Druk op ENTER om de installatie te starten. Als het klaar is, zie je rechtsonder in je scherm:



Klik hierop en een browser wordt gestart om je pagina te bekijken.

CSS

Dat het er raar uit ziet komt omdat we nog geen opmaak hebben gemaakt. CSS staat voor 'Cascading Style Sheets'. We noemen de opmaak ook wel stijl, of style in het Engels. Tijd om de pagina een mooie stijl te geven.

1. Open het bestand **Javascript\Opdrachten\Rekenmachine\rekenmachine.css**
2. Kopieer en plak de onderstaande tekst naar Visual Studio Code.

```
@import url(https://fonts.googleapis.com/css?family=Press+Start+2P);
body {
  font-family: Arial;
  font-size: 25px;
}
td {
  width: 50px;
  height: 40px;
  background-color: #2F54C6;
  text-align: center;
  cursor: pointer;
  border-radius: 3px;
  font-weight: bold;
  color: #fff;
}
#getallen {
  float: left;
  width: 400px;
}
#functies {
  float: left;
}
#uitkomst {
  height: 40px;
  font-family: 'Press Start 2P', cursive;
  text-align: right;
  width: 390px;
}
#som {
  height: 40px;
  border: 1px solid #ddd;
  width: 390px;
  text-align: right;
  line-height: 30px;
  padding: 4px;
  font-family: 'Press Start 2P', cursive;
}
```

3. Sla het bestand op.

Voor allerlei elementen in de HTML bepaalt de stijlsheet het lettertype, de grootte, en de kleur. Er zijn heel veel stijlen mogelijk, maar met wat er nu in `rekenmachine.css` staat zit de rekenmachine er al een stuk beter uit. Probeer maar.

Heb je ook gezien hoe de HTML in `rekenmachine.html` weet dat de stijlen in `rekenmachine.css` te vinden zijn?

Javascript

De pagina lijkt wel op een rekenmachine. Maar hij werkt nog niet zo. Nu moet Javascript erbij komen om ervoor te zorgen dat de rekenmachine ook wat gaat doen. Omdat deze Dojo gaat over Javascript en niet over HTML en CSS staan we iets langer stil bij de code die je nu gaat toevoegen.

1. Druk in Visual Studio Code op **CTRL+N** om een nieuw bestand te maken.
2. Kopieer en plak de onderstaande tekst naar Visual Studio Code.

```
var somtekst = "";
var resultaat = 0;
var nieuweSom = true;
```

Met deze regels hebben we 3 variabelen gemaakt. In elke variabele kan iets worden bewaard en later opnieuw worden gebruikt. Je ziet gelijk 3 soorten variabelen.

De variabele **somtekst** bevat een stukje tekst, ook wel *string* genoemd. Dat zie je door gebruik van de aanhalingstekens. De variabele **resultaat** is een getal of *number*. We beginnen met 0. De variabele **nieuweSom** is een zogenaamde booleaanse variabele. Die kan waar of niet waar zijn, ofwel *true* en *false*.

Onder deze variabelen voegen we een functie toe. Een functie is een stukje code met een eigen naam. Dat we die code een naam geven is handig, want dan kunnen we verderop in het Javascript bestand zo'n functie aanroepen, ofwel laten uitvoeren. Op het moment dat wij dat willen.

3. Kopieer en plak de onderstaande tekst naar Visual Studio Code.

```
function VoegGetalToe(e)
{
    if (e.target.tagName == 'TD')
    {
        somtekst = somtekst + e.target.innerText;
        document.getElementById("som").innerText = somtekst;
    }
    nieuweSom = false;
}
```

De code van deze functie zorgt ervoor dat een getal van de rekenmachine wordt toegevoegd aan de tekst van de som. Straks kan je dus op 3, 6, 9 of elke ander getal in de rekenmachine klikken, en het getal verschijnt op het scherm.

Niet alleen getallen maar ook rekenfuncties moeten we toevoegen. We kunnen kiezen uit plus (+), min (-), vermenigvuldigen (*) en delen (/). Voor het gemak gebruiken we gelijk de rekenfuncties die Javascript ook kent. We gebruiken dus niet 'x' voor keersommen en ook niet ':' voor delen.

4. Kopieer en plak de onderstaande tekst naar Visual Studio Code.

```
function VoegRekenFunctieToe(e)
{
    if (e.target.tagName == 'TD')
    {
        if(e.target.innerText != '=' && e.target.innerText != "Opnieuw")
        {
            somtekst = somtekst + ' ' + e.target.innerText + ' ';
        }

        document.getElementById("som").innerText = somtekst;

        if(e.target.innerText == "Opnieuw" || nieuweSom )
        {
            somtekst = "";
        }
    }
}
```

Dit stukje code zorgt er dus voor dat de rekenfunctie wordt toegevoegd aan de tekst van de som.

Nog 1 functie hebben we nodig. Namelijk eentje die er voor zorgt dat de som ook wordt berekend, en dat de uitkomst op de pagina komt te staan.

5. Kopieer en plak de onderstaande tekst naar Visual Studio Code.

```
function ToonUitkomst()
{
    resultaat = eval(somtekst);
    if(somtekst == "") resultaat = "";
    document.getElementById("uitkomst").innerText = resultaat;
    nieuweSom = true;
}
```

We hadden helemaal bovenin een variabele met de naam **resultaat**, weet je dat nog? In deze functie krijgt deze variabele de waarde van de uitkomst van de som. Als de somtekst leeg is, is ook het resultaat leeg. Dat gebeurt door de regel die begin met **if(somtekst ==**. In vorige functies zag je ook al *if*. En *if*, Engels voor *als*, vraagt dus aan de computer of iets is (==) of iets misschien niet is (!=). De uitkomst van deze vraag kan waar zijn (*true*) of niet waar (*false*). Als het waar is, gaat de computer door, en anders slaat hij een stukje code over.

We hebben nog 1 ding nodig. We moeten nog vertellen op welk moment die functies mogen worden uitgevoerd.

6. Kopieer en plak de onderstaande tekst naar Visual Studio Code.

```
document.getElementById("getallen").onclick = VoegGetalToe;
document.getElementById("functies").onclick = VoegRekenFunctieToe;
document.getElementById("bereken").onclick = ToonUitkomst;
```

Deze code geeft aan de knoppen in de HTML door dat als iemand er op klikt, de functie moet worden uitgevoerd.

In de HTML hebben we een tabel met een *ID* “**getallen**”. Kan je die vinden? Als we ergens in de tabel klikken, dan gaan we naar de functie **VoegGetalToe**. In die functie staat ook (let op: zet deze regel niet onderaan je javascript-bestand):

```
if (e.target.tagName == 'TD')
```

Er wordt hier gecontroleerd of er op een cel in de tabel is geklikt. Zo ja, dan wordt het getal aan de som toegevoegd. Hetzelfde doen we met de functie *VoegRekenFunctieToe*.

De laatste regel laat de som uitrekenen en de uitkomst zien als je op de knop “**bereken**” klikt.

Succes!

Je hebt nu de eerste stappen gezet om Javascript te leren. Tijd voor wat fun!

Spelen met je naam

Wist je dat je ook dingen kunt laten bewegen met Javascript? In deze opdracht ga je zien hoe.

1. Start Visual Studio Code en open de map **javascript\opdrachten\naamfun**.
2. Maak een nieuw bestand in Visual Studio Code (met CTRL-N).
3. Kopieer en plak de volgende code.

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="https://code.jquery.com/jquery-1.10.2.min.js"></script>
    <script type="text/javascript" src="alfabet.js"></script>

  </head>
  <body>
    <canvas id="tekening"></canvas>
    <script type="text/javascript" src="bubbles.js"></script>
    <script type="text/javascript" src="naamfun.js"></script>
  </body>
</html>
```

4. Bewaar het bestand in de map **javascript\opdrachten\naamfun**. Daar staan al een paar andere bestanden, zoals jquery.js, alfabet.js en bubbles.js.

In die bestanden zit al heel veel Javascript code. Dat scheelt intikken.

5. Maar nu een nieuw bestand in Visual Studio Code, en plak de volgende code erin.


```

var naam = "CoderDojo";

var red = [0, 100, 63];
var orange = [40, 100, 60];
var green = [75, 100, 40];
var blue = [196, 77, 55];
var purple = [280, 50, 60];
var letterColors = [red, orange, green, blue, purple];

drawName(naam, letterColors);


if(10 < 3)
{
    bubbleShape = 'square';
}
else
{
    bubbleShape = 'circle';
}

bounceBubbles();

```

6. Bewaar dit bestand in de map **naamfun** onder de naam naamfun.js.

Ga naar de Verkenner op je computer, zoek de map naamfun en open het bestand naamfun.html in een browser. Dat ziet er kleurig uit, niet waar?

Als je het rechtsonder in Visual Studio Code  ziet staan, kan je daar ook op klikken en naamfun.html openen.

Je ziet dat de tekst CoderDojo verschijnt. Kan je daar je eigen naam van maken?

De tekst wordt gemaakt met allemaal gekleurde bolletjes. Weet je hoe je er blokjes van kunt maken? Probeer maar.

Ok, je hebt al aardig wat ervaring nu met Javascript. Durf je de uitdaging aan om een spel te maken, helemaal in Javascript? Ga dan door met je volgende missie.

CoderDojo Snake game - opdracht 1

In deze dojo ga je een snake game maken in de browser. We gebruiken daarvoor de programmeertaal JavaScript. Je hebt geen kennis van deze taal nodig om deze dojo te kunnen volgen, alles wordt uitgelegd.

De enige applicaties die je nodig hebt is een tekstverwerker om de code te schrijven en een webbrowser om het resultaat te bekijken.

Beginsituatie

Je begint met vier bestanden:

- `index.html` - Dit is de webpagina waarin de game zal komen.
- `onrequestframe.js` en `timestamp.js` - Dit zijn twee hulp-scripts die we later in de dojo gaan gebruiken. Wat deze bestanden doen is geen onderdeel van de dojo.
- `snake.js` - In dit bestand ga je de game maken. Als je hier iets in wijzigt dan kun je het resultaat daarvan bekijken in de browser door `index.html` te openen.

De achtergrond tekenen

Het eerste dat we gaan doen is het tekenen van de achtergrond. Voeg daarvoor de volgende code toe aan `snake.js`:

```
(function () {  
    var canvas = document.getElementById("canvas"),  
        context = canvas.getContext("2d"),  
        width = canvas.width,  
        height = canvas.height;  
  
    function draw() {  
        context.fillStyle = "white";  
        context.fillRect(0, 0, width, height);  
        context.strokeStyle = "black";  
        context.strokeRect(0, 0, width, height);  
    }  
  
    draw();  
})();
```

Sla de wijzigingen op en open `index.html` in een webbrowser. Je ziet nu een wit vlak met een zwarte lijn.

Hoe werkt het script?

Om te beginnen zetten we alles tussen haakjes en maken we een functie.

```
(function () {  
    ...  
})();
```

Dit is een JavaScript trucje om te zorgen dat alles wat binnen de accolades ({ en }) staat alleen bekend is binnen de haakjes. Waarom dat handig is, is voor deze dojo niet belangrijk, maar als je dat wilt weten dan kun je het aan je begeleider vragen.

Hoe een functie precies werkt wordt zodadelijk uitgelegd als we daadwerkelijk gaan tekenen.

Variabelen

Vervolgens maken we vier "variabelen":

```
var canvas = document.getElementById("canvas"),
    context = canvas.getContext("2d"),
    width = canvas.width,
    height = canvas.height;
```

Een variable is een naampje die we aan een bepaalde waarde kunnen geven, zodat we later naar die waarde kunnen verwijzen via de naam i.p.v. dat we telkens de waarde zelf op moeten zoeken.

De eerste variable, *canvas*, verwijst naar het canvas-element in `index.html`. Dit is een speciaal Html5 element waarmee je tekeningen kan maken op de webpagina. De tweede variable die we nodig hebben is *context*. Dit is een onderdeel van het canvas en dit object bevat de methodes waarmee we daadwerkelijk gaan tekenen.

Vervolgens slaan we nog de breedte (*width*) en hoogte (*height*) van het canvas-object op, zodat we die later kunnen gebruiken.

Een functie maken

Nu we alle nodige variabelen hebben, gaan we een functie definiëren met de naam *draw*.

```
function draw() {
    ...
}
```

Een functie in JavaScript maak je door eerst het woord `function` te gebruiken, gevolgd door de naam die je aan de functie wilt geven en twee haakjes. Waar die haakjes voor dienen zul je later nog zien. Voor nu moet je onthouden dat je ze altijd nodig hebt. Tot slot zet je alles wat onderdeel moet zijn van de functie tussen accolades.

De volgende vier regels zijn dus onderdeel van de *draw*-functie:

```
context.fillStyle = "white";
context.fillRect(0, 0, width, height);
context.strokeStyle = "black";
context.strokeRect(0, 0, width, height);
```

Deze code tekent twee rechthoeken op het scherm, dankzij de functies *fillRect* (dit tekent een rechthoek van punt (0,0) met de breedte en hoogte van het canvas die we eerder opgeslagen hebben en vult het met de kleur die in de variable *fillStyle* is opgeslagen) en *strokeRect* (dit tekent een rechthoek met dezelfde afmetingen, maar

in plaats van het te vullen met een kleur tekent het een rand met de kleur die in *strokeStyle* is opgeslagen).

Belangrijk: Wat je als laatste tekent staat op het canvas bovenaan. Kijk maar eens wat er gebeurt als je de eerste twee en laatste twee regels omdraait.

Speel ook eens met de kleuren die je aan *fillStyle* en *strokeStyle* meegeeft (de meeste engelse namen voor kleuren zouden moeten werken) en met de afmetingen van de rechthoeken. Wat gebeurt er op het scherm als je de pagina herlaadt?

De functie aanroepen

We hebben nu een functie gemaakt genaamd *draw*. Maar als we hier stoppen dan gebeurt er nog niks. Om de functie daadwerkelijk uit te voeren moet je hem namelijk aanroepen. Dat doen we in de laatste regel:

```
draw();
```

Een functie aanroepen doe je door de naam van de functie te typen en daarna de haakjes te typen. De ; aan het einde van de regel is simpelweg bedoeld om tegen JavaScript te zeggen dat je klaar bent met deze regel.

BORING!

Ok, we hebben nu een wit vlak met een zwarte rand getekend. Dat was spannend, NOT!. In het volgende deel wordt het interessanter, dan gaan we de snake tekenen...

CoderDojo Snake game - opdracht 2

In dit deel gaan we snake tekenen. Hij zal nog niet bewegen, dat gaan we doen in een volgende opdracht, maar in ieder geval hoeven we niet meer alleen naar het saaie witte scherm te staren.

Nieuwe variabelen

Om bij te kunnen houden waar de snake is en hoe groot hij is hebben we twee nieuwe variabelen nodig. Voeg deze toe aan de lijst van variabelen aan het begin van snake.js:

```
var canvas = document.getElementById("canvas"),
    context = canvas.getContext("2d"),
    width = canvas.width,
    height = canvas.height,
    snake_array,
    cellwidth = 10;
```

De eerste variable, snake_array, zal alle delen (we gaan de snake in blokjes tekenen) bewaren en de tweede variable, cellwidth, bepaalt hoe groot elk blokje van de snake zal zijn. Zie je dat we bij de variable snake_array geen waarde hebben ingevuld? Het is niet verplicht om bij het maken van een variable deze ook meteen een waarde te geven.

De snake maken

De volgende stap is om de nieuwe variabelen te gebruiken om een snake te maken waarmee je het spel begint. Dit doen we weer in een functie zodat we die aan kunnen roepen wanneer we dat willen:

```
function create_snake() {
    var length = 5, i;
    snake_array = [];
    for (i = length - 1; i >= 0; i--) {
        snake_array.push({x: i, y: 0});
    }
}
```

We noemen de functie create_snake. Het eerste dat we doen is twee variabelen declareren; length en i. Omdat we deze variabelen declareren binnen de functie create_snake zijn ze ook alleen maar binnen die functie beschikbaar. De variable length krijgt de waarde 5. Dit is de beginlengte van de snake.

Ten tweede geven we de variable snake_array (die wel buiten de functie beschikbaar is) een waarde []. Dit is een array, ofwel een lijstje van waarden. Arrays zijn heel handig, omdat je er van alles in kan stoppen.

Tot slot maken we een for-loop die vijf elementen aan snake_array toevoegd. Een for-loop is een stukje code die een aantal keer achter elkaar uitgevoerd wordt, zolang een bepaalde voorwaarde waar is (in dit geval $i \geq 0$). Je geeft een beginwaarde op voor i ($i = \text{length} - 1$, dus $i = 4$) en na elke keer dat de code is uitgevoerd verander je de waarde van i ($i--$, dit betekent: maak i 1 kleiner).

De regel snake_array.push({x: i, y: 0}); voegt een object met een x-coördinaat en een y-coördinaat toe aan snake_array. Dit stelt de positie voor van elk deel van de snake. Omdat we i gebruiken voor de x-coördinaat en omdat die elke keer verandert

krijgen we vijf verschillende posities. Na het uitvoeren van deze code ziet `snake_array` er zo uit (dit hoef je niet in `snake.js` te zetten):

```
snake_array = [
  {x: 4, y: 0},
  {x: 3, y: 0},
  {x: 2, y: 0},
  {x: 1, y: 0},
  {x: 0, y: 0}
];
```

Zoals je in de vorige opdracht hebt geleerd is er nu nog niks gebeurd, omdat we de functie `create_snake` nog nergens aanroepen. Zet daarom onderaan het script (boven `draw()`;) de aanroep:

```
create_snake();
draw();
```

De snake tekenen

Als je de webpagina nu laadt, dan zie je nog steeds alleen het witte vlak met de zwarte rand. Toch is de snake nu gemaakt en opgeslagen in `snake_array`. Alleen moeten we, om dat te kunnen zien, de snake wel nog tekenen.

Daarvoor maken we een nieuwe functie, `draw_snake`. Zet de onderstaande code boven de `draw`-functie:

```
function draw_snake() {
  var cell, i;
  for (i = 0; i < snake_array.length; i++) {
    cell = snake_array[i];
    context.fillStyle = "blue";
    context.fillRect(cell.x * cellwidth, cell.y * cellwidth, cellwidth,
cellwidth);
    context.strokeStyle = "white";
    context.strokeRect(cell.x * cellwidth, cell.y * cellwidth, cellwidth,
cellwidth);
  }
}
```

Deze functie gebruikt weer een *for-loop* om elk element van de snake te tekenen. Elk element wordt getekend als een blauw blok met een witte rand, d.m.v.

de `fillRect` en `strokeRect` functies die je al eerder hebt gezien. De positie en grootte van het blok wordt bepaald door de x- en y-coördinaten die we in `snake_array` hebben opgeslagen en de `cellwidth` variabele die we eerder hebben gemaakt.

Binnen de *for-loop* zie je ook hoe je een enkel element uit een array kan halen. De regel `cell = snake_array[i];` doet dit. Als `i` de waarde 0 heeft, dan wordt het eerste element opgehaald, is `i = 1`, dan wordt het tweede element opgehaald, enz.

Nu moeten we deze nieuwe functie alleen nog ergens aanroepen. Dat doen we in dit geval in de `draw`-functie:

```
function draw() {
  context.fillStyle = "white";
  context.fillRect(0, 0, width, height);
  context.strokeStyle = "black";
  context.strokeRect(0, 0, width, height);

  draw_snake();
}
```

Weet je nog dat wat je als laatste tekent bovenaan staat? Daarom moeten we `draw_snake` aanroepen nadat we de achtergrond tekenen, anders zou de achtergrond boven de snake staan en zien we nog niets. Als je nu het script opslaat en de webpagina laadt, dan zie je links bovenaan de snake. Is dat de positie die je had verwacht? Speel eens met de x- en y-waarden in `create_snake` en kijk wat er gebeurt met de snake op het scherm.

Beweging toevoegen

Nu we een snake op het scherm hebben wordt het tijd om deze te laten bewegen. Dat is wat je in de volgende opdracht gaat doen.

CoderDojo Snake game - opdracht 3

In deze opdracht ga je leren hoe je de snake kan laten bewegen. Dit wordt best wel een complexe en lange opdracht, dus probeer het te volgen en als je iets niet begrijpt, vraag het dan aan je begeleider.

De positie van de snake aanpassen

Het eerste dat we gaan doen is een functie maken die de positie van de snake kan aanpassen, deze noemen we `update`. Voeg deze functie toe na de `draw`-functie:

```
function update() {  
    var nx = snake_array[0].x,  
        tail = snake_array.pop();  
  
    tail.x = nx + 1;  
  
    snake_array.unshift(tail);  
}
```

Deze functie definieert twee variabelen, `nx` en `tail`. De eerste variable is de x-coördinaat van het eerste element in `snake_array` en de tweede variable is het laatste element uit `snake_array`. Belangrijk om te weten is dat de functie die we gebruiken om het laatste element te vinden, `pop`, niet alleen het element terug geeft, maar deze ook verwijdert uit de array.

Vervolgens wordt de x-coördinaat van het laatste element aangepast zodat het 1 hoger is dan de x-coördinaat van het eerste element. Tot slot wordt het laatste element teruggezet in `snake_array` met de `unshift`-functie. Wat deze functie doet is het element dat je meegeeft vooraan de array zetten en alle andere elementen een plek op te schuiven. Begrijp je al wat dit doet?

De game loop

Nu wordt het tijd om de game loop te bouwen. Een game loop is een standaard concept in het maken van spelletjes. Dit zorgt er voor dat er telkens nieuwe plaatjes op het scherm getekend worden die je als speler kan beïnvloeden met je toetsenbord en muis of met een game-controller.

Een game loop maak je door een functie te maken die heel vaak opnieuw wordt aangeroepen. In die functie die je normaal gesproken twee dingen: de positie van je objecten in het spel bijwerken en de spelwereld opnieuw tekenen. De functies om dit te doen hebben we al, namelijk `update` en `draw`. Wat we dus nog nodig hebben is een functie om telkens aan te roepen en een manier om die functie heel vaak aan te roepen.

Om dit voor elkaar te krijgen vervang je de volgende code:

```
create_snake();  
draw();
```

met:

```
function run() {  
    update();
```



```

    draw();
}

function init() {
    create_snake();

    window.onEachFrame(run);
}

init();

```

run is de functie die we telkens opnieuw gaan uitvoeren. Deze doet simpelweg wat hierboven beschreven was, namelijk update en draw aanroepen. Met de init-functie kunnen we het spel starten. Deze maakt onze spelobjecten (alleen de snake op dit moment) en start de game loop (door de run-functie aan window.onEachFrame mee te geven). Tot slot roepen we init(); aan om het spel te starten.

Kijk maar eens wat er nu gebeurt als je het script opslaat en de webpagina opent. Wow, dat ging snel, de snake vliegt over het scherm! Dat komt omdat de functie window.onEachFrame zo vaak mogelijk aangeroepen wordt (op moderne computers is dat ongeveer 60 keer per seconde). Dat is prima voor de draw-functie, want dat zorgt er voor dat de animaties mooi vloeiend verlopen, maar het is minder prima voor de update-functie omdat de snake nu veel te snel gaat om te kunnen controleren.

De snelheid van update aanpassen

Om de snelheid waarmee update aangeroepen wordt aan te passen in de run-functie hebben we eerst wat nieuwe variabelen nodig. Deze zetten we bovenaan aan het script bij de rest van de variabelen, zodat deze er zo uit zien:

```

var canvas = document.getElementById("canvas"),
    context = canvas.getContext("2d"),
    width = canvas.width,
    height = canvas.height,
    snake_array,
    cellwidth = 10,
    now,
    last = window.timestamp(),
    dt = 0,
    step = 0.05;

```

De nieuwe variabelen gaan we als volgt gebruiken:

- now - Deze variable zal elke keer dat run wordt aangeroepen bijgewerkt worden om de huidige tijd te bepalen
- last - Deze variable zal de tijd bewaren van de vorige keer dat run werd aangeroepen. We geven deze variable om te beginnen de huidige tijd met de window.timestamp-functie.
- dt - In deze variable gaan we het verschil tussen now en last in seconden bewaren
- step - Deze variable bepaalt hoe vaak de update-functie wordt uitgevoerd. De waarde die we er aan geven, 0.05, betekent dat update elke 0.05 seconden wordt aangeroepen (20 keer per seconde, i.p.v. de 60 keer per seconde die het was)

Om deze nieuwe variabelen te gebruiken passen we de `run`-functie aan:

```
function run() {
  now = window.timestamp();
  dt = dt + Math.min(1, (now - last) / 1000);

  while (dt > step) {
    dt = dt - step;
    update();
  }

  last = now;

  draw();
}
```

Zoals hierboven is uitgelegd zorgt dit ervoor dat `update` maar 20 keer per seconde wordt aangeroepen, i.p.v. 60 keer. Het mooie is dat we de snelheid van de snake nu kunnen aanpassen door de waarde van de `step`-variable aan te passen. Maken we de waarde van `step` groter, dan gaat de snake langzamer. Maken we de waarde kleiner, dan gaat de snake sneller. Experimenteer maar eens met een paar verschillende waarden en kijk wat er gebeurt.

`window.onEachFrame` **en** `window.timestamp`

In deze opdracht hebben we twee functies gebruikt, namelijk `window.onEachFrame` en `window.timestamp`. Je vraagt je misschien af waar deze functies vandaan komen? Deze komen uit de help-scripts `onrequestframe.js` en `timestamp.js` waar we het in opdracht 1 over hadden. Als je wilt, kan je de scripts bekijken om uit te vogelen hoe ze werken.

Vervolg

We hebben nu een bewegende snake, maar hij vliegt nog wel heel snel het scherm uit zonder dat we daar iets aan kunnen doen. In de volgende opdracht gaan we er voor zorgen dat je de snake kan besturen.

CoderDojo Snake game - opdracht 4

In deze opdracht ga je leren hoe je de snake kan besturen met de pijltjestoetsen.

Richting bepalen

Om te beginnen hebben we een manier nodig om te bepalen in welke richting de snake beweegt. Daarvoor voegen we een nieuwe variable, *direction*, toe aan de lijst met variabelen aan het begin van het script:

```
var canvas = document.getElementById("canvas"),
    context = canvas.getContext("2d"),
    width = canvas.width,
    height = canvas.height,
    snake_array,
    cellwidth = 10,
    now,
    dt = 0,
    last = window.timestamp(),
    step = 0.05,
    direction;
```

Deze variable geven we in de *init*-functie de beginwaarde *right*:

```
function init() {
    direction = "right";
    create_snake();

    window.onEachFrame(run);
}
```

Dit zorgt er voor dat de snake naar rechts zal bewegen als het spel begint.

Van richting veranderen

Vervolgens moeten we de *update*-functie aanpassen om de richting van de snake te veranderen op basis van de waarde van *direction*:

```
function update() {
    var nx = snake_array[0].x,
        ny = snake_array[0].y,
        tail = snake_array.pop();

    if (direction === "right") {
        nx = nx + 1;
    } else if (direction === "left") {
        nx = nx - 1;
    } else if (direction === "up") {
        ny = ny - 1;
    } else if (direction === "down") {
        ny = ny + 1;
    }

    tail.x = nx;
    tail.y = ny;

    snake_array.unshift(tail);
}
```

Naast de x-positie van het eerste element, `nx`, bewaren we nu ook de y-positie in de variable `ny`. Daarna kijken we wat de waarde van `direction` is. Is dit *right* of *left*, dan passen we `nx` aan. Is het *up* of *down*, dan passen we `ny` aan. Tot slot geven we deze x- en y-positie door aan `tail` en voegen deze weer toe aan `snake_array`.

Toetsenbord input

Tot slot willen we de waarde van de variable `direction` kunnen aanpassen met het toetsenbord. Daarvoor moeten we een functie maken die wordt aangeroepen als de gebruiker een toets indrukt. Voeg de volgende code toe, onder de `init`-functie, maar boven de `init()`; aanroep:

```
document.onkeydown = function (e) {
    var key = e.which;

    if (key === 37 && direction !== "right") {
        direction = "left";
    } else if (key === 38 && direction !== "down") {
        direction = "up";
    } else if (key === 39 && direction !== "left") {
        direction = "right";
    } else if (key === 40 && direction !== "up") {
        direction = "down";
    }
};
```

Het indrukken van een toets wordt een *event* genoemd. Er zijn allerlei soorten events, niet alleen voor het indrukken van toetsen, maar ook voor het laden van de pagina, het bewegen van de muis en het klikken op links of buttons in de webpagina. Om iets zinnigs te doen met zo'n event moet je een functie maken en die koppelen aan dat event. Dat doen we door de functie aan `document.onkeydown` te geven. De functie heeft geen naam. Dat hoeft ook niet omdat we de functie niet zelf willen aanroepen, maar alleen als het `keydown`-event plaats vindt. Wel heeft de functie een variable `e` tussen de haakjes staan. Dit noemen we een *parameter*. Op deze manier kan de aanroeper van de functie een waarde meegeven die we vervolgens in de functie kunnen gebruiken. In dit geval zorgt JavaScript er voor dat de waarde `e` gevuld wordt met informatie over het event. Zo kunnen we met `e.which` achterhalen welke toets ingedrukt is.

Vervolgens bepalen we wat de nieuwe waarde van `direction` moet worden. We zijn geïnteresseerd in vier waardes van `e.which`:

- 37 - Dit is de *pijltje naar links* toets
- 38 - Dit is de *pijltje omhoog* toets
- 39 - Dit is de *pijltje naar rechts* toets
- 40 - Dits is de *pijltje omlaag* toets

Voordat we de `direction` aanpassen controleren we ook nog of we niet op dit moment in de tegenovergestelde richting gaan. Het is namelijk niet toegestaan om direct om te draaien. Dus `if (key === 37 && direction !== "right")` betekent: Als we *pijltje naar links* hebben ingetoetst EN we gaan niet naar rechts. Als beide waar zijn, dan wordt `direction` veranderd in de waarde *left*.

Als je nu de webpagina laadt, dan kan je de snake besturen met de pijltjes-toetsen.

Botsingen!

De volgende stap is om te voorkomen dat de snake uit het speelveld kan raken, of door zichzelf heen kan gaan. Dat gaan we oplossen in de volgende opdracht.

CoderDojo Snake game - opdracht 5

We kunnen nu de snake bewegen, maar hij vliegt nog wel het scherm uit als we niet oppassen. Ook kan de snake door zichzelf heen bewegen, wat niet de bedoeling is. Om dit te voorkomen moeten we een techniek toepassen die *collision detection* heet (vertaald: botsings-detectie)

Botsing met de randen

We gaan eerst controleren of de snake de randen van het speelveld heeft geraakt. Om dat te doen gaan we de huidige inhoud van `update` splitsen in twee functies. Zet de volgende code boven de `update`-functie:

```
function get_new_position() {
    var nx = snake_array[0].x,
        ny = snake_array[0].y;

    if (direction === "right") {
        nx = nx + 1;
    } else if (direction === "left") {
        nx = nx - 1;
    } else if (direction === "up") {
        ny = ny - 1;
    } else if (direction === "down") {
        ny = ny + 1;
    }

    return {x: nx, y: ny};
}

function update_position(position) {
    var tail = snake_array.pop();

    tail.x = position.x;
    tail.y = position.y;

    snake_array.unshift(tail);
}
```

We maken gebruik van de techniek die we in de vorige opdracht hebben geleerd om een *parameter* aan de `update_position`-functie mee te geven. Die `position` parameter krijgen we door de `get_new_position`-functie te laten eindigen met het woord `return`, gevolgd door een object met de juiste x- en y-coördinaat.

Nu hebben we nog een functie nodig die ons kan vertellen of de nieuwe positie een van de randen raakt. Zet deze code ook boven de `update`-functie:

```
function check_wall_collision(position) {
    return position.x === -1
        || position.x === width / cellwidth
        || position.y === -1
        || position.y === height / cellwidth;
}
```

Hier controleren we of er aan een van vier voorwaarden is voldaan. Als de x-positie -1 is OF als de x-positie gelijk is aan `width / cellwidth` OF als de y-positie -1 is OF als de y-positie gelijk is aan `height / cellwidth`. Dit zijn de voorwaarden die ons vertellen of we een van de randen geraakt hebben.\

Nu kunnen we de `update`-functie aanpassen:

```
function update() {
    var position = get_new_position();

    if (check_wall_collision(position)) {
        init();
        return;
    }

    update_position(position);
}
```

Eerst wordt de nieuwe positie bepaald met `get_new_position`. Vervolgens wordt gecontroleerd of we de rand hebben geraakt. Als dat zo is, dan herstarten we het spel (met `init()`). Als we de rand niet geraakt hebben, dan passen we de positie van de snake aan met `update_position`.

Probeer je nieuwe code uit en bekijk wat er gebeurt als je een van de randen raakt.

Botsing met jezelf

Ook als de snake botst met zichzelf moet het spel opnieuw gestart worden. Hiervoor hebben we maar een extra functie nodig. Zet deze functie

onder `check_wall_collision`:

```
function check_self_collision(position) {
    var i;
    for (i = 0; i < snake_array.length; i++) {
        if (snake_array[i].x === position.x && snake_array[i].y === position.y) {
            return true;
        }
    }
    return false;
}
```

Deze functie controleert elk element in de `snake_array` en kijkt of de *parameter* `position` dezelfde plek inneemt. Als dat zo is, dan botst de snake met zichzelf.

Nu moeten we weer de `update`-functie aanpassen:

```
function update() {
    var position = get_new_position();

    if (check_wall_collision(position) || check_self_collision(position)) {
        init();
        return;
    }

    update_position(position);
}
```

En nu kan de snake ook niet meer straffeloos met zichzelf botsen. Als je moeite hebt om dit te zien in de browser, pas dan de beginlengte van de snake aan in de `create_snake`-functie.

Tijd voor wat voedsel...

We zijn nu wel lang genoeg met de snake bezig geweest. Het wordt tijd om voor voedsel te zorgen dat de slang kan opeten om groter te worden. Dat gaan we in de volgende opdracht doen.

CoderDojo Snake game - opdracht 6

Tot nu toe hebben we ons alleen maar met de snake zelf bezig gehouden. Het wordt tijd om de snake ook een doel te geven in de vorm van wat heerlijk voedsel.

Voedsel maken

Eerst moeten we voedsel maken voordat de snake het op kan eten. Stap één om dat voor elkaar te krijgen is, zoals zo vaak, een nieuwe food-variable maken:

```
var canvas = document.getElementById("canvas"),
    context = canvas.getContext("2d"),
    width = canvas.width,
    height = canvas.height,
    snake_array,
    cellwidth = 10,
    now,
    dt = 0,
    last = window.timestamp(),
    step = 0.05,
    direction,
    food;
```

Verder gaat het maken van een food element hetzelfde als het maken van de snake. Eerst maken we een functie `create_food`, vervolgens maken we een functie `draw_food` en tot slot moeten we zorgen dat deze functies op de juiste plekken aangeroepen worden. Hier is de `create_food`-functie, zet deze onder `create_snake`:

```
function create_food() {
    food = {
        x: Math.round(Math.random() * (width - cellwidth) / cellwidth),
        y: Math.round(Math.random() * (height - cellwidth) / cellwidth)
    };
}
```

Dit ziet er erg ingewikkeld uit, maar het idee is heel simpel: We maken een object op een willekeurige positie ergens in het speelveld. Deze nieuwe functie roepen we vervolgens aan in de `init`-functie:

```
function init() {
    direction = "right";
    create_snake();
    create_food();

    window.onEachFrame(run);
}
```

Vervolgens maken we de `draw_food`-functie. Vervang de `draw_snake`-functie door de volgende code:

```
function draw_cell(cell) {
    context.fillStyle = "blue";
    context.fillRect(cell.x * cellwidth, cell.y * cellwidth, cellwidth,
cellwidth);
    context.strokeStyle = "white";
    context.strokeRect(cell.x * cellwidth, cell.y * cellwidth, cellwidth,
cellwidth);
}

function draw_snake() {
```

```

    var i;
    for (i = 0; i < snake_array.length; i++) {
        draw_cell(snake_array[i]);
    }
}

function draw_food() {
    draw_cell(food);
}

```

Zie je wat we hier hebben gedaan? In plaats van nogmaals in `draw_food` de `fillRect` en `strokeRect` functies aan te roepen hebben we een nieuwe functie `draw_cell` gemaakt die een enkel blokje tekent. In de `draw_snake`-functie roepen we deze functie nu aan voor elk element van `snake` en in `draw_food` roepen we dezelfde functie aan voor het `food`-element. Als laatste moeten we niet vergeten de `draw_food`-functie aan te roepen in de `draw`-functie:

```

function draw() {
    context.fillStyle = "white";
    context.fillRect(0, 0, width, height);
    context.strokeStyle = "black";
    context.strokeRect(0, 0, width, height);

    draw_snake();
    draw_food();
}

```

Voedsel eten

Om het voedsel op te eten moeten we drie dingen doen:

- We moeten controleren of we ons op de plek van een voedsel-blokje bevinden
- Als dat zo is, dan moet de snake een blokje langer worden
- En er moet een nieuw voedsel-blokje gemaakt worden

Laten we beginnen met controleren of we ons op de plek van een voedsel-blokje bevinden. Zet de volgende functie onder `check_self_collision`:

```

function check_food_collision(position) {
    return position.x === food.x && position.y === food.y;
}

```

Deze functie is vrij simpel, hij controleert of de x- en y-coördinaat van de parameter `position` overeen komt met de coördinaten van `food`.

Vervolgens passen we de `update`-functie aan:

```

function update() {
    var position = get_new_position();

    if (check_wall_collision(position) || check_self_collision(position)) {
        init();
        return;
    }

    if (check_food_collision(position)) {
        eat_food();
        create_food();
    } else {

```

```
        update_position(position);
    }
}
```

In plaats van simpelweg altijd `update_position` aan te roepen, doen we dat nu alleen als we niet op de plek van een voedsel-blokje zijn terecht gekomen. Als dat wel zo is, dan roepen we `eat_food` aan (die we nog moeten maken) om de snake langer te maken en daarna `create_food` (die we al hebben) om een nieuw voedsel-blokje te maken.

Nu hoeven we alleen nog maar `eat_food` te maken. Vervang de `update_position`-functie met de volgende code:

```
function eat_food() {
    snake_array.unshift(food);
}

function update_position(position) {
    snake_array.pop();

    snake_array.unshift(position);
}
```

We voegen simpelweg het `food`-element toe aan de voorkant van de `snake_array`. We hebben ook `update_position` aangepast door de `tail`-variable weg te gooien. Het bleek dat we die helemaal niet nodig hadden.

Probeer het spel nu maar eens te spelen. Elke keer als je over een voedsel-blokje gaat wordt de snake langer. Hoe lang houdt jij het vol?

Score bijhouden

In de laatste opdracht gaan we nog een ding toevoegen, namelijk een score. Elke keer als je een voedsel-blokje oppakt krijg je een punt. Op die manier hoef je niet alle blokjes van de snake te tellen om te weten hoe goed je bent en kun je makkelijker opscheppen tegen je vriendjes!

CoderDojo Snake game - opdracht 7

In deze laatste opdracht ga je een score-teller toevoegen om de game helemaal af te maken.

Score bijhouden

Om de score bij te houden hebben we weer een nieuwe variable nodig, laten we deze score noemen (dit is de laatste, ik beloof het):

```
var canvas = document.getElementById("canvas"),
    context = canvas.getContext("2d"),
    width = canvas.width,
    height = canvas.height,
    snake_array,
    cellwidth = 10,
    now,
    dt = 0,
    last = window.timestamp(),
    step = 0.05,
    direction,
    food,
    score;
```

De beginwaarde hiervan is uiteraard 0, dat zetten we in de init-functie:

```
function init() {
    direction = "right";
    score = 0;
    create_snake();
    create_food();

    window.onEachFrame(run);
}
```

En elke keer als we een blokje opeten, maken we de score 1 hoger:

```
function update() {
    var position = get_new_position();

    if (check_wall_collision(position) || check_self_collision(position)) {
        init();
        return;
    }

    if (check_food_collision(position)) {
        score = score + 1;
        eat_food();
        create_food();
    } else {
        update_position(position);
    }
}
```

Score laten zien

Nu hoeven we alleen nog maar de score te laten zien. Hiervoor maken we een `draw_score`-functie (zet deze boven de `draw`-functie):

```
function draw_score() {  
    var score_text = "Score: " + score;  
    context.fillStyle = "red";  
    context.fillText(score_text, 5, height - 5);  
}
```

Als je het tot hier hebt kunnen volgen, dan snap je vast wel wat er hier gebeurt. We maken een tekst met daarin de huidige score en tekenen deze op het scherm met de `fillText`-functie. Kun je zonder te spieken inschatten waar op het scherm de tekst komt te staan?

Tot slot moeten we de `draw_score`-functie alleen nog aanroepen in de `draw`-functie:

```
function draw() {  
    context.fillStyle = "white";  
    context.fillRect(0, 0, width, height);  
    context.strokeStyle = "black";  
    context.strokeRect(0, 0, width, height);  
  
    draw_snake();  
    draw_food();  
    draw_score();  
}
```

En nu is je game helemaal af! Veel plezier!

Helemaal af?

Is de game nu helemaal af? Nee, een game is namelijk nooit af. Je kan altijd nog dingen bij bedenken. Probeer bijvoorbeeld maar eens om de snake steeds sneller te laten gaan voor elke 10 blokjes die je op eet.