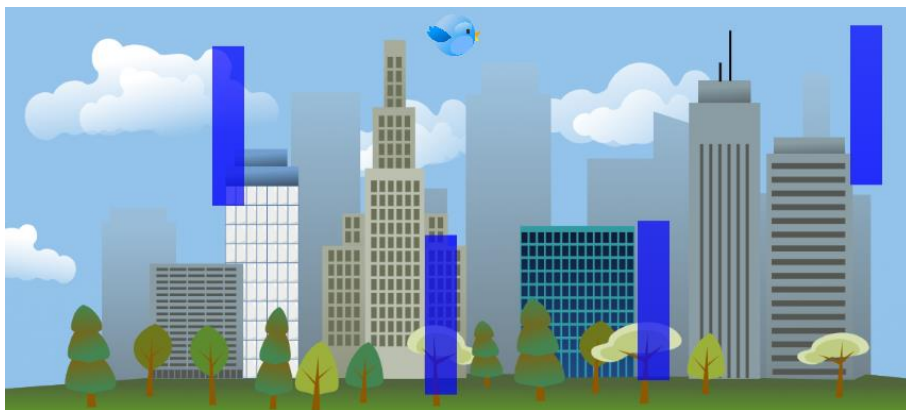
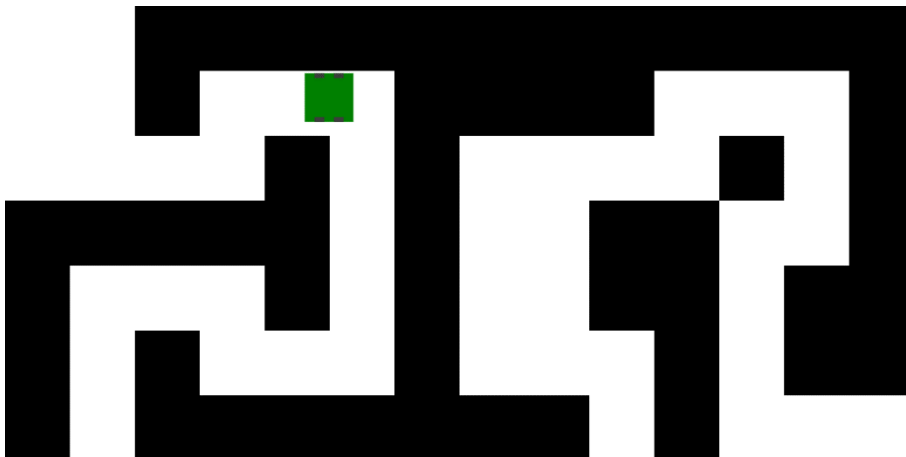




games maken en ervaren



Domein J: Keuzethema Programmeerparadigma's
Domein P: Keuzethema User experience

INHOUD

Inleiding op de module	4
H1 kennismaken met P5	5
1.1 het canvas: ons speelveld	5
1.2 P5-functies: parameters en argumenten	8
1.3 variabelen	10
1.4 draw is een loopfunctie	11
1.5 zelf functies maken	14
1.6 Voorwaarden: if en else	16
1.7 vaste herhalingen met een for-loop	19
1.8 VERDIEPING: recursie 	23
H2 objecten	27
2.1 Inleiding (camelCase)	27
2.2 afbeeldingen	27
2.3 lijsten: arrays	30
2.4 animatie: een array van plaatjes	32
2.5 VERDIEPING: spritesheets 	35
2.6 zelf objecten maken	37
2.7 een object dat ja of nee antwoordt	42
2.8 een klasse van objecten	45
2.9 een array van objecten	48
H3 game design	Fout! Bladwijzer niet gedefinieerd.
3.1 Inleiding (Wat is een spel?)	Fout! Bladwijzer niet gedefinieerd.
3.2 Het object spel & flowcharts	Fout! Bladwijzer niet gedefinieerd.
3.3 Acties en gebeurtenissen	Fout! Bladwijzer niet gedefinieerd.
3.4 Vaste patronen: design patterns	Fout! Bladwijzer niet gedefinieerd.
3.5 Levels en score	Fout! Bladwijzer niet gedefinieerd.
3.6 Tijd	Fout! Bladwijzer niet gedefinieerd.
3.7 Geluid sterparagraaf	Fout! Bladwijzer niet gedefinieerd.
H4 gamification	Fout! Bladwijzer niet gedefinieerd.
4.1 Inleiding	Fout! Bladwijzer niet gedefinieerd.
4.2 Levels	Fout! Bladwijzer niet gedefinieerd.

INLEIDING OP DE MODULE

In deze module leer je stap voor stap programmeren in de taal Javascript met P5. P5 of *Processing* is een aanvulling op Javascript. Zo'n aanvulling heet een bibliotheek (*library*). P5 is speciaal gemaakt om het programmeren makkelijker te maken voor mensen die net beginnen met programmeren. Maar vergis je niet: je kunt het zo ingewikkeld maken als je zelf wilt!

In hoofdstuk 1 leer je de basisprincipes van het programmeren. Als je al eerder geprogrammeerd hebt, kun je sneller door dit hoofdstuk heen. In hoofdstuk 2 zoomen we in op een speciale programmeertechniek die *object-georiënteerd programmeren* heet. Het werken met objecten gaat ons helpen bij ons uiteindelijke doel: spellen programmeren in Javascript. Hoofdstuk 3 is helemaal gewijd aan het ontwerpen en ervaren van games.

Voordat je kunt beginnen met programmeren in P5, moet je een omgeving inrichten om in te werken en te testen. Dat kan op vele manieren online of met speciale programma's. Daarom leggen we dat hier niet uit. Wel kun je in Appendix A lezen hoe je thuis aan de slag zou kunnen gaan. Jouw leraar vertelt je hoe jullie op school gaan werken en zorgt dat jij de bestanden krijgt om mee te werken. Meer informatie over de bestanden en hoe ze samenwerken vind je in Appendix B.

Als je coderegels wilt lezen of aanpassen kijk je in een *editor*. Als je het resultaat wil bekijken, gebruik je een *browser* (zoals Chrome of Firefox).

Deze module is zo gemaakt dat je leert programmeren door het maken van opdrachten. Dit betekent dat niet alles dat je leert ook in de theorie terug te vinden is. Wel zijn er voorbeelden beschikbaar om je op weg te helpen. Heel handig is bovendien de *reference* van P5, waarin je uitleg krijgt bij alle beschikbare P5-commando's.

Onderstaande links zijn belangrijke bronnen van informatie:

Overzicht en uitleg van P5-commando's:	https://p5js.org/reference
Interactieve site met <i>keycodes</i> :	http://keycode.info
Overzicht van html-kleurnamen:	https://htmlcolorcodes.com/color-names
Uitgebreide naslag voor ontwikkelaars #1:	https://developer.mozilla.org/nl/docs/Web/JavaScript
Uitgebreide naslag voor ontwikkelaars #2:	http://devdocs.io/javascript
Uitgebreide Engelstalige uitleg:	https://javascript.info

H1 KENNISMAKEN MET P5

1.1 het canvas: ons speelveld

Als je *voorbeeld 1* in je browser opent, zie je figuur 1.1 als resultaat. Deze tekening is geprogrammeerd met Javascript. De code bestaat uit twee belangrijke delen: de **setup** en **draw**.

In de *setup* van een programma (figuur 1.2) geef je de begininstellingen van het programma. We gaan hier niet alle regels één voor één uitleggen.

Belangrijk voor nu zijn:

- `createCanvas(1000,500);`
Maak een speelveld van 1000 pixels breed en 500 pixels hoog
- `background('orange');`
Geef het speelveld een oranje achtergrond

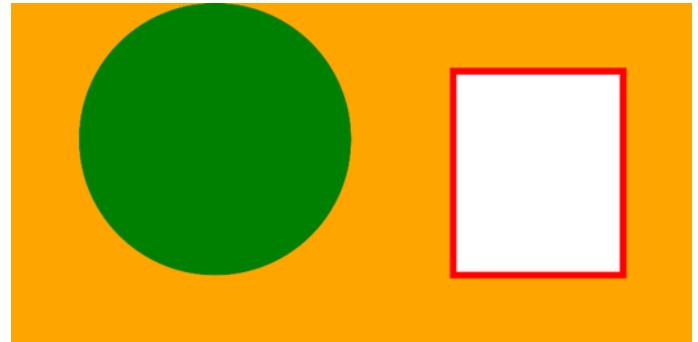
Het tweede deel van de code staat in figuur 1.3. Dit is ons hoofdprogramma: de *draw*. In dit voorbeeld staan hier commando's om vormen te tekenen in het canvas.

Het **canvas** kun je zien als een veld van pixels. Je verwijst naar een bepaald punt met een x-waarde en een y-waarde. Belangrijk om te weten: het punt [0,0] ligt linksboven en een grotere y-waarde betekent dat je naar beneden gaat (en niet omhoog, zoals in een assenstelsel bij wiskunde).

In figuur 1.4 zie je het resultaat van figuur 1.1 met een raster van hokjes van 100 × 100 pixels. Die helpen je bij het begrijpen van de code in figuur 1.3:

- `ellipse(300,200,400);`
Teken een ellips (een cirkel is een ellips die net zo breed is als hij hoog is) met het **middelpunt** in [300,200] en een breedte en hoogte van 400 pixels.
- `rect(650,100,250,300);`
Teken een rechthoek met als **positie linksboven** [650,100] en een breedte van 250 en een hoogte van 300.
- `fill('white');`
Kies een vulkleur. Merk op dat je eerst een vulkleur kiest en dan pas een vorm tekent.
- `stroke('red');`
`strokeWeight(10);`
Teken de figuur met een rode rand van 10 pixels.
- `noStroke();`
Teken de figuur zonder rand.

We gaan nu ons eerste programma schrijven.



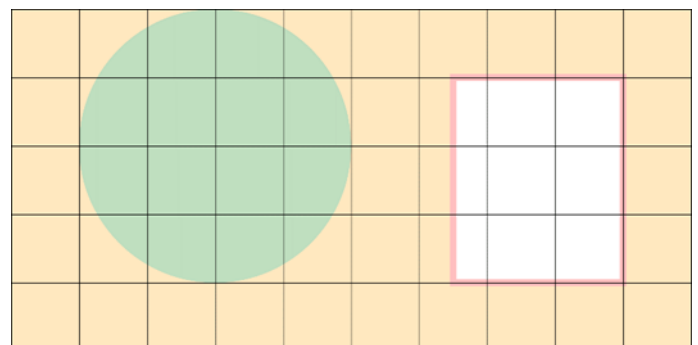
FIGUUR 1.1

```
function setup() {  
  var myCanvas = createCanvas(1000,500);  
  background('orange');  
  myCanvas.parent('processing');  
  noLoop();  
}
```

FIGUUR 1.2

```
function draw() {  
  // groene cirkel zonder rand  
  noStroke();  
  fill('green');  
  ellipse(300,200,400);  
  // witte rechthoek met rode rand  
  stroke('red');  
  fill('white');  
  strokeWeight(10);  
  rect(650,100,250,300);  
}
```

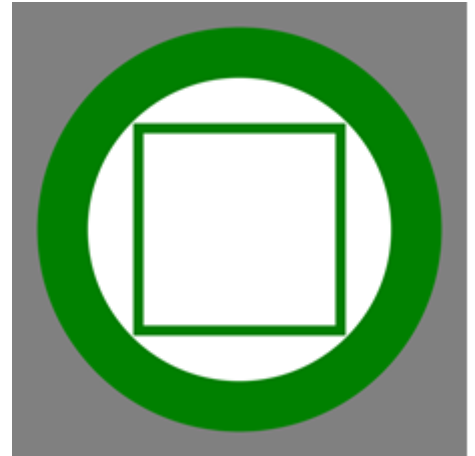
FIGUUR 1.3



FIGUUR 1.4

✓ Opdracht 1 vormen tekenen I

1. Open *H1O1.js* (Hoofdstuk 1 Opdracht 1) in jouw *editor*. Dit is de code van *voorbeeld 1*. Bekijk het resultaat in de *browser*.
2. Pas de code aan zodat het canvas grijs (*grey*) van kleur wordt.
3. Pas de code aan zodat het canvas nog maar 450 pixels hoog is.
4. Zorg dat de groene cirkel zowel boven als links op een afstand van 25 pixels van de rand van het canvas komt te staan.
5. Voeg een tweede, witte cirkel toe met hetzelfde middelpunt als de groene cirkel en een diameter van 300.
6. Pas de regel voor de rechthoek aan, naar:
`rect(125, 125, 200, 200);`
7. Pas de code aan zodat het eindresultaat uit figuur 1.5 verschijnt met een vierkant canvas.
8. Er zijn nu twee regels met `ellipse`. Voorspel wat je zal zien als je de beide regels van plek laat wisselen. Probeer daarna uit.



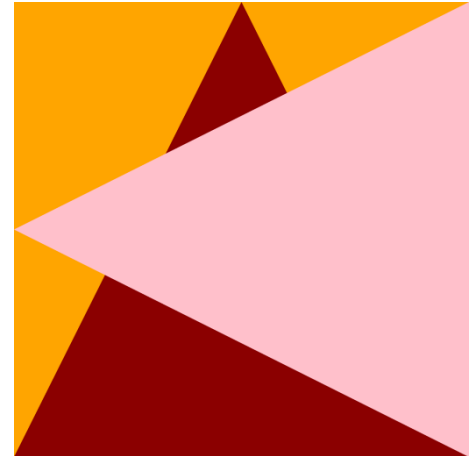
FIGUUR 1.5

Opdracht 2 vormen tekenen II

Als je in de *P5-reference* kijkt onder vormen (*shape*:

<https://p5js.org/reference/#group-Shape>) dan zie je een flink aantal commando's om vormen te tekenen. We proberen er een aantal uit.

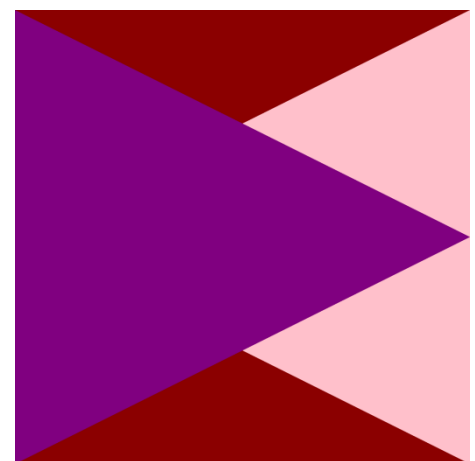
9. Open *H1O2.js* in jouw *editor*. Bekijk het resultaat in de *browser*.
10. De code bevat het commando `triangle`. Gebruik de *reference* om uit te zoeken wat de betekenis is van de zes getallen tussen de haakjes.
11. Pas de code aan, zodat het beeld uit figuur 1.6 ontstaat. (Het Engelse woord voor roze is *pink*.)
12. Gebruik de kleur paars (*purple*) om figuur 7 na te bouwen. Wie goed, kijkt heeft maar twee driehoeken nodig!



FIGUUR 1.6

Per hoekpunt van de driehoek gebruiken we een x-waarde en een y-waarde. Op dezelfde manier kun je met het commando `quad` ook een vierhoek maken die niet per se rechthoekig is. Dat wordt al gauw onoverzichtelijk. Bovendien: hoe maak je dan een vijf- of zeshoek? P5 heeft een algemenere manier om een vorm met hoekpunten te tekenen. Zo'n vorm heet toepasselijk een *shape*. Hier een voorbeeld:

```
strokeWeight(5);
stroke('pink');
fill('darkred');
beginShape();
vertex(225, 115);
vertex(300, 225);
vertex(225, 335);
vertex(0, 225);
endShape(CLOSE);
```



FIGUUR 1.7

13. Bekijk *voorbeeld 2* dat laat zien hoe je met `shape` werkt. Wat is een *vertex*? Wat doet `CLOSE`?
14. Neem de code hierboven over en plaats het in de *draw* onder de regels voor de driehoeken.

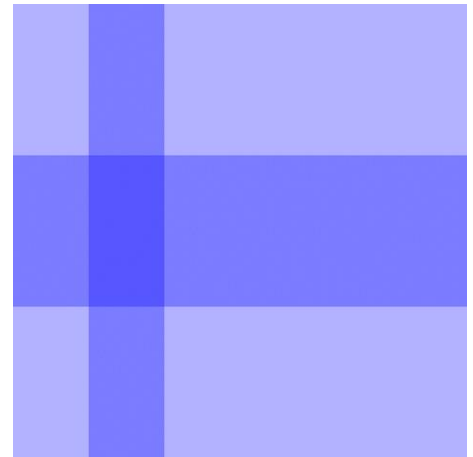
Opdracht 3 kleuren en doorzichtigheid

In de eerste opdrachten en voorbeelden hebben we kleur aangegeven met Engelse woorden zoals *pink* en *blue*. Als je wel eens een website hebt gemaakt, ben je misschien bekend met **hexadecimale kleurcodes** zoals `#FFC0CB`. Van een tekenprogramma zoals Photoshop of de natuurkunde-les ken je misschien het maken van kleuren met de primaire kleuren rood, groen en blauw in het **RGB**-systeem. Met P5 kun je ze allebei gebruiken. Zie de *reference* onder kleur:

<https://p5js.org/reference/#group-Color>

Met 255 rood (de maximale waarde), 192 groen en 203 blauw ofwel (255,192,203) maak je de kleur *pink*, net als met `#FFC0CB`. Deze site laat zien welke kleurnamen horen bij de kleurcoderingen:

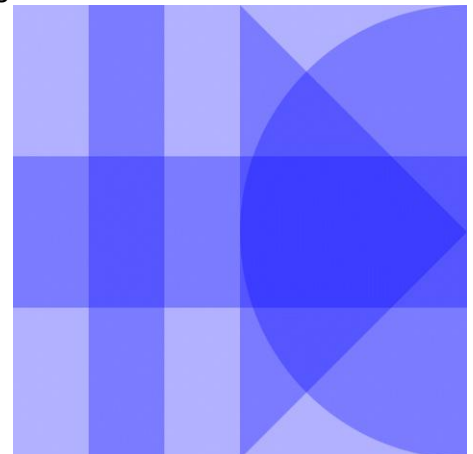
<https://htmlcolorcodes.com/color-names>



FIGUUR 1.8

Natuurlijk kun je ook je eigen kleurcombinaties maken.

15. Open *H1O3.js* in jouw *editor*. Bekijk het resultaat in de *browser*.
16. In regel 12 (kleurinstelling) en regel 13 (vorm) wordt een rood rechthoek getekend. Verklaar waarom je deze rode horizontale balk niet op het scherm ziet.
17. Vervang de regel die zorgt voor een groene kleur door `fill(0,255,0)`; Bekijk het resultaat.
18. We hebben nu een andere kleur groen. Gebruik bovenstaande link om de juiste kleurcodering voor de kleur *green* te vinden.
19. Vervang de regel die zorgt voor een blauwe kleur door een hexadecimale kleurcode die hetzelfde resultaat geeft als *blue*. Gebruik opnieuw de link naar de webpagina met kleurnamen.
20. In de code staan twee regels met `//` ervoor (er staat ook eentje in de *setup!*). Verwijder de `//` voor beide regels. Eén van de regels die je nu actief hebt gemaakt is `fill(0,0,255,0.3)`; Verwijder alle andere regels met het commando `fill`.
21. Als het goed is zie je als je de code uitvoert het resultaat uit figuur 1.8. We zien nu ook de horizontale balk die eerst onzichtbaar was. Wat is blijkbaar de betekenis van de `0.3` tussen de haakjes van `fill`?
22. Controleer je voorspelling door de waarde `0.3` te laten variëren tussen 0 en 1. Let op: gebruik een punt en geen komma. Wat zie je als je precies 0 of 1 invult?
23. Voeg een cirkel toe met een diameter van 450 en als middelpunt `[450,225]`. Zorg dat de vulkleur weer `0.3` bevat.
24. Voeg een driehoek toe zodanig dat de tekening in figuur 1.9 ontstaat. Kies daarna je eigen favoriete kleur.



FIGUUR 1.9

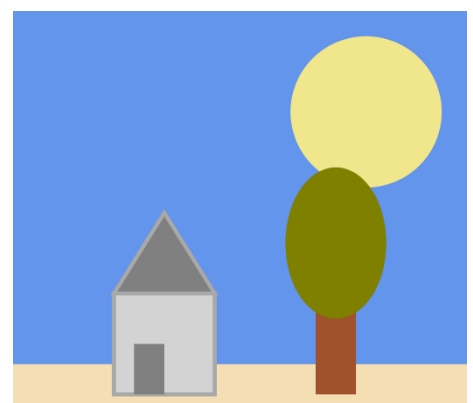
Opdracht 4 huis met boom

In deze opdracht oefenen we nog een keertje met het tekenen van vormen, randen en kleuren. Ons doel is de afbeelding in figuur 1.10.

25. Open *H1O4.js* in jouw *editor*. Bekijk het resultaat in de *browser*.

In de code staan meerdere commentaarregels met opdrachten om de tekening compleet te maken. Als je een programma aanpast, doe je er goed aan om dit in stapjes te doen. Kijk elke keer even of een stap gelukt is, voordat je verder gaat.

26. Voer de opdrachten in de code uit zodat de afbeelding in figuur 1.10 verschijnt. Gebruik de extra instructies in de code.



FIGUUR 1.10

1.2 P5-functies: parameters en argumenten

In de eerste paragraaf heb je gewerkt met onder andere `fill`, `stroke`, `strokeWeight`, `rect`, `noStroke` en `ellipse` (figuur 1.11). We hebben dit tot nu toe commando's genoemd. Vanaf nu gebruiken we de term **functie**. Met een functie geef je de computer de opdracht om iets uit te voeren. Het was je misschien al opgevallen dat achter elke functie twee haakjes staan. Tussen die haakjes zet je de extra informatie die de functie nodig heeft om goed te werken.

Als je in de *reference* kijkt bij de uitleg van de functie `strokeWeight`, dan zie je het volgende:

Syntax

```
strokeWeight(weight)
```

Parameters

`weight` Number: the weight (in pixels) of the stroke

```
fill('green');  
stroke('red');  
strokeWeight(10);  
rect(50,100,250,300);  
noStroke();
```

FIGUUR 1.11

Met de term **syntax** (of: *syntaxis*) wordt het geheel aan taalregels van een programmeertaal bedoeld. De syntax vertelt je hoe je de taal, in ons geval Javascript, moet schrijven.

In de beschrijving staat een **functie** `strokeWeight`. In de uitleg zien we dat deze functie een gegeven nodig heeft om goed te kunnen werken. Zo'n gegeven heet een **parameter**. In dit geval is dit de lijndikte `weight`.

Een parameter heeft een algemene naam, zoals `weight`. Als je de functie wilt gebruiken, moet je een specifieke waarde voor de parameter meegeven: `strokeWeight(10)`

In dit voorbeeld heet de waarde `10` een **argument**. Niet alle functies hebben een argument nodig.

Voorbeeld: met de functie `noStroke()` geef je aan dat je geen rand om een vorm wilt.

In de volgende opgaven maak je kennis met een aantal P5-functies die erg handig kunnen als je tekent.

Opdracht 5 verplaatsen: `translate`

Stel dat je het huis in figuur 1.10 naar rechts wilt verplaatsen met 15 pixels. Dan moet je zowel het huis, de deur als het dak aanpassen. Dat is veel werk, maar gelukkig kunnen we de functie `translate(x,y)` gebruiken. Hiermee verplaats je de oorsprong (met de coördinaten $x = 0$ en $y = 0$) naar een andere plaats op het canvas. Daarom heeft `translate` twee parameters x en y om de oorsprong horizontaal en verticaal te kunnen verplaatsen.

27. Open `H1O5.js` in jouw *editor*. Bekijk het resultaat in de *browser*. Dit is het eindresultaat van opdracht 4.
28. In regel 14 staat de functie `translate`. Hiermee willen we het huis verplaatsen.
Pas de argumenten aan, zodat $x = 90$ en $y = -10$. Wat is de betekenis van het minteken? Voorspel wat het resultaat van deze aanpassing zal zijn.
29. Klopte jouw voorspelling? In regel 26 staat nog een regel met `translate` die is uitgeschakeld met `//`. Haal de `//` weg. Begrijp je waarom nu alleen het huis verplaatst (en net nog niet)?

Deze opgave laat zien hoe `translate` werkt, maar normaal gesproken pas je dit wel anders toe: als je de oorsprong verplaatst, kun je juist makkelijk vanaf de coördinaten $[0,0]$ tekenen. Dit proberen we uit.

```
function draw() {  
  noStroke();  
  fill('tan');  
  //translate(25,25);  
  rect(0,0,400,400);  
  //translate(200,50);  
  fill('peru');  
  rect(0,0,150,150);  
}
```

FIGUUR 1.12

30. Bekijk de code uit figuur 1.12 en maak een schets van de tekening die zal ontstaan en een tweede schets van de tekening wanneer de `//` voor beide regels met `translate` worden weggehaald.
31. Verwijder alle coderegels in de `draw` en kopieer de code uit figuur 1.12. Klopte jouw tekening?

Opdracht 6 push & pop

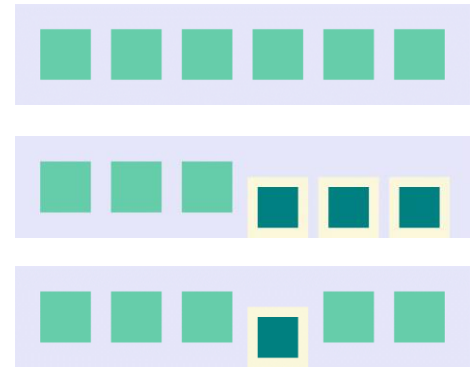
Een ontwerper heeft een rij van zes vierkanten gemaakt zoals in de bovenste afbeelding van figuur 1.13. Het uiteindelijke beeld moet één vierkant bevatten dat er anders uitziet, zoals in de onderste afbeelding. De ontwerper heeft een paar aanpassingen gedaan voor de vierde figuur, met de middelste afbeelding als resultaat.

32. De programmeercode van de ontwerper staat in *H106.js*. Open dit bestand in jouw *editor*. De aanpassingen die de ontwerper heeft gedaan zijn in het bestand gemarkeerd.

Als je in P5 een andere tekeninstelling kiest zoals een vulkleur, dan zal de computer deze vulkleur normaal gesproken blijven gebruiken totdat je weer een andere kleur kiest. Dat geldt voor alle tekeninstellingen.

Als je terug wilt naar de vorige instellingen zou je alles weer terug moeten zetten zoals het was. Dat kan wel, maar is veel werk. De functies **push** en **pop** bieden uitkomst. Met **push** sla je de huidige tekeninstellingen op in het geheugen van de computer. Je kunt daarna de instellingen zoals **fill**, **stroke** maar ook **translate** tijdelijk aanpassen voor een stukje van de tekening. Ben je klaar en wil je weer verder tekenen met de bewaarde instellingen? Dan kun je die terughalen met **pop**.

33. Voeg voor de eerste aanpassing van het vierde vierkant de regel **push()**; in. Bekijk het resultaat: is er iets veranderd?
34. Voeg nu na het tekenen van het vierde vierkant de regel **pop()**; toe. Controleer of jouw tekening nu overeenkomt met de onderste afbeelding uit figuur 1.13.
35. Geef de vijf gelijke vierkanten in één handeling de kleur *thistle*.
36. Wat verwacht je te zien als we de regel **pop()**; niet na het vierde maar na het vijfde vierkant plaatsen? Probeer het uit en verklaar wat je ziet.



FIGUUR 1.13



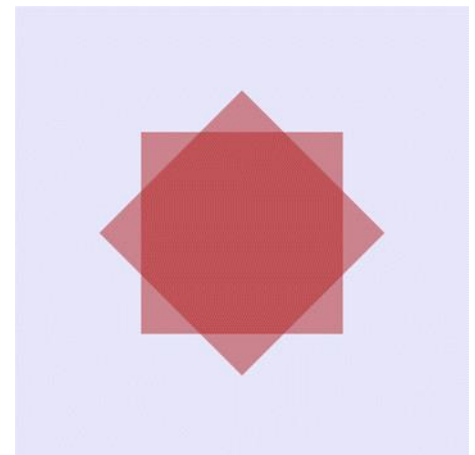
Opdracht 7 draaien I

In figuur 1.14 zie je twee half-doorzichtige vierkanten: het onderste vierkant is *normaal*, de bovenste is gedraaid over 45° zodat een ruit ontstaat.

In opgave 2 hebben we gezien dat je met de functie **shape** willekeurige vormen kunt maken met hoekpunten. In principe zou je hiermee ook een ruit kunnen tekenen, maar dat is wel omslachtig. Het is makkelijker om gewoon een vierkant te draaien. Hiervoor is de functie **rotate** (Engels: draaien of roteren).

37. Open *H107.js* in jouw *editor* en bekijk het resultaat.
38. In regel 15 staat **rotate(0)**. Verander het argument van de functie in 7. Rond welk punt is het vierkant gedraaid? En over hoeveel graden is hij gedraaid?

Waarschijnlijk is het resultaat anders dan je verwacht. Eén van de redenen is dat Javascript hoeken niet in graden maar radialen rekent.



FIGUUR 1.14

39. Voeg de regel **angleMode(DEGREES)**; toe aan **setup** en geef **rotate** als argument 45 mee.

P5 heeft een aantal standaard tekeninstellingen. Eén daarvan is dat bij een bewerking op een vierkant of rechthoek vanaf linksboven wordt gerekend. Daarom draait het vierkant niet om zijn as.

40. Voeg de regel **rectMode(CENTER)**; toe aan de *setup* en bekijk het resultaat.
41. Pas de argumenten van **translate** aan, zodat de figuur weer in het midden van het canvas staat.
42. Plaats een geel vierkant met zijde 50 in het midden van het vierkant. Neem als RGB-kleurcode 255,225,0. Gebruik **push** en **pop** om te zorgen dat het vierkant niet gedraaid staat als een ruit.

1.3 variabelen

In de laatste opgaven hebben we meerdere vierkanten moeten tekenen met steeds dezelfde zijde. Hiervoor hebben we elke keer opnieuw hetzelfde getal als argument meegegeven aan de *rect*-functie. Voor zes vierkanten betekent dit twaalf keer (lengte en breedte) dezelfde waarde invullen!

Als je de vierkanten iets groter wilt maken, moet je dus ook op twaalf verschillende plaatsen die lengte van een zijde aanpassen. Zou het niet fijn zijn als we de computer één keer konden vertellen dat de *zijde* van de vierkanten bijvoorbeeld 150 pixels moet zijn? En dat hij het daarna zelf onthoudt? Dat kan met een **variabele**.

```
var lengte = 5;
var breedte = 8;
var omtrek;

function draw() {
  omtrek = 2*lengte + 2*breedte;
  text("De omtrek is " + omtrek, 50, 50);
  rect(0, 0, lengte, breedte);
}
```

FIGUUR 1.15

Als je een nieuwe variabele maakt of **declareert**, dan geef je een naam aan een klein stukje van het geheugen van de computer. In de code van figuur 1.15 worden drie variabelen gedeclareerd. Twee daarvan krijgen bovendien een beginwaarde mee. Dat heet **initialisatie**.

De variabele *omtrek* krijgt geen beginwaarde, omdat we die door de computer willen laten berekenen. De opdracht daarvoor staat in de *draw*: `omtrek = 2*lengte + 2*breedte`;

De functie *text* zet de tekst *De omtrek is 24* op het scherm met:

```
text("De omtrek is " + omtrek, 50, 50);
```

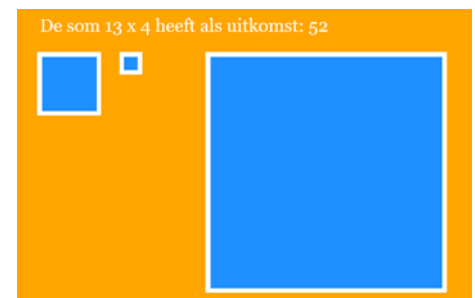
Gewone tekst zet je tussen aanhalingstekens, getallen en variabelen niet. Je 'plakt' de losse stukjes aan elkaar door er een `+` tussen te zetten.

Naast de variabelen die je zelf kunt maken, kent P5 ook een aantal eigen variabelen zoals *mouseX*, *mouseY*, *width* en *height*. Hun betekenis leer je in opgave 9.



Opdracht 8 variabelen

43. Open *H108.js* in jouw *editor*. Dit is de code van *voorbeeld 3*. Bekijk het resultaat in de *browser*.
44. Geef variabele *B* de waarde 11 en bekijk het resultaat.
45. Zorg dat de rekensom $21 - 9 = 12$ op het scherm verschijnt. (Let ook op de tekst!)
46. In figuur 1.16 wordt de rekensom $13 \times 4 = 52$ getoond. Zorg dat hetzelfde beeld op jouw scherm verschijnt. Pas de grootte van het canvas aan, zodat ook het grote vierkant in het beeld past.



FIGUUR 1.16

Opdracht 9 P5-variabelen en tekst

47. Open *H109.js* in jouw *editor*. Dit is de code van *voorbeeld 4*. Bekijk het resultaat in de *browser*.
48. In welke variabele slaat P5 het aantal pixels van de hoogte van jouw browserscherm op?
49. In welke variabele slaat P5 het aantal pixels van de hoogte van jouw canvas op?

Regel 17 bevat de code: `text("mouseX:" + mouseX + "\nmouseY:" + mouseY, mouseX, mouseY)`; die ervoor zorgt dat er een stuk tekst meebeweegt met de x-positie (*mouseX*) en y-positie (*mouseY*) van de muis.

50. Verklaar dat deze gele tekst achter het blauwe tekstvak verdwijnt.
51. In regel 17 staat `"\nmouseY:"`. Toch zie je alleen *mouseY* in beeld. Haal `\n` weg. Wat zie je?
52. We ronden de positie van de muis af naar hele pixels met de functie *round*. Verander regel 17 naar:
`text("mouseX:" + round(mouseX) + " mouseY:" + round(mouseY), mouseX, mouseY);`
53. In de code staan vier functies die beïnvloeden hoe de tekst wordt getoond (*textFont*, *textSize*, *textLeading* en *textAlign*). Zoek uit wat deze functies doen en experimenteer ermee. Gebruik eventueel de *reference*: <https://p5js.org/reference/#group-Typography>

Opdracht 10 simpel tekenprogramma

We gaan de P5-variabelen `mouseX` en `mouseY` gebruiken om zelf een tekening te maken.

54. Open `H1O10.js` in jouw *editor*. Bekijk het resultaat in de *browser*. Hoe heeft de programmeur er zonder hoofdrekenen voor gezorgd dat de stip precies in het midden staat?
55. Pas de argumenten van de functie `ellipse` aan, zodat de stip jouw muis volgt.

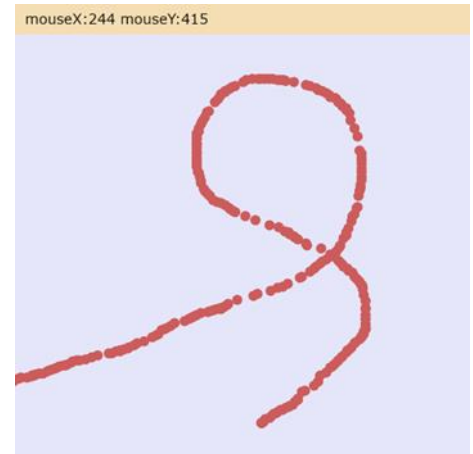
Het is je misschien opgevallen dat `background('lavender');` in dit programma niet in de *setup* maar in de *draw* is geplaatst. Waarom?

56. Verplaats de genoemde regel van de *draw* naar de *setup*. Wat is er veranderd?

Nu de achtergrondkleur in de *setup* wordt ingesteld, wordt deze alleen in het begin gekleurd. Alles wat daarna verandert, blijft in beeld staan.

We kunnen nu tekenen, maar de tekst bovenaan wordt onleesbaar. Gelukkig is er een simpele oplossing:

57. Voeg aan het begin van de *draw* een regel toe die een rechthoek tekent met de (vul-) kleur `wheat`. Gebruik de standaard P5-variabele voor de canvasbreedte en zorg voor een hoogte van 30 pixels zodat bovenaan een balk wordt getekend zoals in figuur 1.17.



FIGUUR 1.17

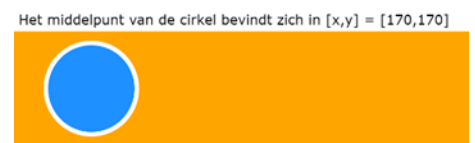
1.4 draw is een loopfunctie

Alle programma's en voorbeelden die je tot nu toe hebt gezien hebben een *setup* met de regel `noLoop();`. Was het je opgevallen dat deze regel in de laatste twee opgaven was uitgeschakeld met `//`?

In paragraaf 1.1 is genoemd dat in de *setup* de begininstellingen van het programma worden beschreven en dat de *draw* het hoofdprogramma is. In P5 is *draw* wel een bijzonder geval, want de code binnen de *draw* wordt telkens opnieuw uitgevoerd. Een functie die regels code steeds opnieuw herhaalt noemen we een **loopfunctie** (*loop* is Engels voor 'lus' of 'herhalingslus'). In opgave 9 en 10 hebben we voor het eerst het voordeel van zo'n herhalingslus gezien: omdat de *draw* steeds opnieuw wordt uitgevoerd, kan worden gevolgd wat de actuele plaats van de muis is. Met die informatie krijgt de canvastekening steeds een update. Hoe vaak gebeurt dat eigenlijk?

In *voorbeeld 5* (waarvan figuur 1.18 een screenshot toont) zie je hoe een cirkel zich van links naar rechts verplaatst. Hierbij gebruikt men:

- o in de *setup*: `frameRate(10);`
Dit zorgt ervoor dat de animatie met 10 *frames* (beeldjes) per seconde wordt getekend, ofwel: dat de functie *draw* 10x per seconde wordt uitgevoerd.
- o aan het einde van de *draw*: `horizontaal += 2;`
Dit zorgt ervoor dat de variabele `horizontaal` die bepaalt op welke breedte de cirkel wordt getekend elk frame met 2 wordt opgehoogd, zodat de cirkel steeds verder naar rechts wordt getekend.



FIGUUR 1.18

De regel `horizontaal += 2;` had ook geschreven kunnen worden als `horizontaal = horizontaal+2;` Dat is iets langer, maar voor sommige mensen wel duidelijker. Je moet deze regel lezen als: *de nieuwe waarde van de variabele 'horizontaal' is gelijk aan de oude waarde van de variabele 'horizontaal' plus 2*. Javascript heeft handige, korte notaties voor berekeningen. Wil je één optellen of aftrekken van de waarde van een variabele, dan gebruik je `horizontaal++`; en `horizontaal--`; Verdubbelen? Gebruik `horizontaal *= 2;` Halveren? Gebruik `horizontaal /= 2;` (of `horizontaal *= 0.5;`)

Het is niet altijd nodig om de *draw*-functie eindeloos uit te voeren. Als de code binnen de *draw* maar één keer moet worden uitgevoerd of als je het herhalen wilt stopzetten, dan gebruik je `noLoop();`



Opdracht 11 automatische bewegingen

58. Open *H1O11.js* in jouw *editor*. Dit is de code van *voorbeeld 5*.
Bekijk het resultaat in de *browser*.

De blauwe cirkel noemen we cirkel A. Voor de beginpositie van A gebruiken we de variabele *horizontaalA*. We gaan een tweede cirkel B maken. Zie figuur 1.19.

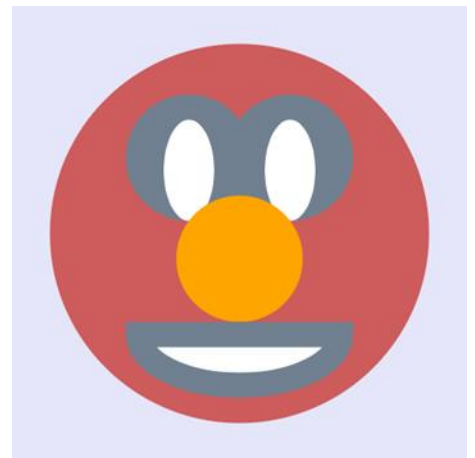


FIGUUR 1.19

59. Declareer voor de beginpositie van B een variabele *horizontaalB* met de waarde 500.
60. Teken cirkel B even groot als A, maar met de kleur *darkred*. Natuurlijk maak je voor de horizontale positie gebruik van *horizontaalB*.
61. Zorg dat de positie van B elke keer dat de *draw* wordt uitgevoerd met 1 wordt verhoogd.
62. Pas het argument van de functie *frameRate* aan naar 50. Wat is het resultaat?
63. Pas de tekst aan zodat er komt te staan:
positie A = 454 positie B = 667 (Dit is maar één voorbeeld: de waarden veranderen steeds.)

Opdracht 12 maak kennis met JOS

In deze opgave maken we kennis met JOS. JOS is een *game character*: een poppetje dat we in meerdere opgaven tegen zullen komen (figuur 1.20). Zijn naam is een afkorting van *Javascript Object Sprite*. Een *sprite* is veelgebruikte term voor een tweedimensionaal plaatje of animatie. In hoofdstuk 2 leer je wat een object is.



FIGUUR 1.20

64. Open *H1O12.js* in jouw *editor*. Bekijk het resultaat. Je ziet een behoorlijk aantal regels die ervoor zorgen dat JOS wordt getekend, maar het is niet nodig om die nu te bestuderen.
65. Voeg de regel *xJOS--*; toe aan het eind van de *draw* en bekijk het resultaat. Wat betekent deze regel?
66. Gebruik *yJOS--*; zodat JOS naar linksboven beweegt.
67. Pas de regel aan naar *yJOS -= 2*; zodat JOS twee keer zo snel omhoog beweegt als dat hij naar links beweegt.
68. Pas regel 17 (*translate*) aan, zodat JOS meebeweegt met jouw muis.

Opdracht 13 P5-functies voor beperken en schalen

In de vorige opdracht was het mogelijk om JOS van het canvas te laten verdwijnen. In veel spellen is het de bedoeling dat je *game character* zichtbaar blijft, ofwel: op het canvas blijft. Voor jou als programmeur betekent dit dat je de beweging van JOS moet inperken.

69. Open *H1O13.js* in jouw *editor*. Bekijk het resultaat. Zorg dat je de muis heen en weer beweegt!

Als het goed is heb je gemerkt dat je JOS wel kunt bewegen, maar dat je niet meer alle vrijheid hebt. Oorzaak is de regel *xJOS = constrain(mouseX, 100, 450)*; (*constrain* is Engels voor beperken). JOS mag bewegen volgens de horizontale muispositie (*mouseX*), maar alleen tussen de waarden 100 en 450.

70. Pas de code aan, zodat JOS links en rechts precies tot de rand van het canvas kan bewegen.
HINT: in welke regel kun je zien hoe groot JOS is?

71. Breid de code uit, zodat JOS ook boven en onder precies tot de rand van het canvas kan bewegen.

Als we Jos wat kleiner willen tekenen, lijkt dat een hele klus, omdat hij is opgebouwd met flink wat regels programmeercode. Gelukkig heeft P5 een functie om de omvang van tekeningen te schalen: *scale(1)*; Met de waarde 1 wordt alles op normale grootte (100%) getekend, met b.v. *scale(0.5)*; op 50%.

72. Teken Jos op 50% van zijn normale grootte. Wat zie je? LET OP: volgt Jos de muis nog wel goed?
73. Verplaats de regel *scale(0.5)*; zodat hij meteen na *push()*; staat. Probleem opgelost?



Opdracht 14 snelheid

In deze opgave gaan we JOS uit zichzelf laten bewegen.

74. Open *H1O14.js* in jouw *editor* en bekijk het resultaat.
75. In regel 15 is `yJOS--`; uitgezet met `//`. Haal deze weg en bekijk het resultaat.
76. Wat gebeurt er als JOS bovenaan is? Waarom?
77. Declareer een variabele *snelheid* en geef deze de waarde 17.
78. Pas de regel is `yJOS--`; aan zodat JOS niet met stapjes van 1 maar van 17 naar boven beweegt. Gebruik hiervoor de variabele *snelheidJOS*.

Jos beweegt nu sneller naar boven, maar zijn beweging is niet heel natuurlijk. Als jij iets omhoog gooit, dan gaat het steeds langzamer omhoog, omdat de snelheid afneemt.

79. Maak onder de regel die je net hebt aangepast een nieuwe regel die ervoor zorgt dat de snelheid van JOS bij elke loop van de *draw* afneemt met 0,5.
(LET OP: in Javascript gebruik je niet een komma maar een punt.)
80. Zorg dat de *snelheid* bovenin wordt getoond, naast *x* en *y*.
81. Verklaar de beweging die je nu ziet. Leg bovendien uit waarom de snelheid bovenin blijft veranderen, ook als JOS weer 'op de grond' staat.

Opdracht 15 een functie om JOS te tekenen

We hebben kennis gemaakt met JOS, een poppetje dat je nog vaker tegen zult komen in deze module. Er zijn behoorlijk wat coderegels nodig om hem te tekenen. Het zou mooi zijn als we JOS met één commando (functie) op het scherm zouden kunnen krijgen.

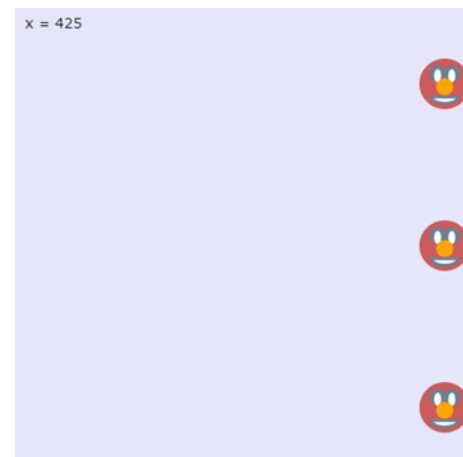
82. Open *H1O15.js* in jouw *editor*. In regel 17 wordt de functie *tekenJos* aangeroepen. Deze functie heeft twee parameters die bepalen waar JOS wordt getekend.
83. JOS wordt nu horizontaal in het midden getekend. Zorg dat (het middelpunt van) JOS 75 pixels vanaf de linkerkant van het canvas wordt getekend.

```
translate(0,160);
tekenJos(xJOS,yJOS);
```

FIGUUR 1.21

De functie *tekenJos* is natuurlijk geen standaard-functie van JS of P5. Hij wordt gemaakt in de regels 20 t/m 40. Daar zie je alle regels terug waarmee JOS kan worden getekend. Het gebruik van de functie lijkt meer werk (hoewel de code in de *draw* nu wel overzichtelijker is). Functies worden pas echt handig als je ze vaker gebruikt.

84. Voeg onder regel 16 de twee coderegels toe uit figuur 1.21 en bekijk het resultaat. Voeg beide regels daarna nogmaals toe, zodat je JOS drie keer op het scherm ziet.
85. Gebruik de eigenschappen van de loopfunctie *draw* om alle drie de versies van JOS naar rechts te laten bewegen met stapjes van 3 pixels.
86. Gebruik de functie *constrain* om te zorgen dat de poppetjes stoppen op het moment dat ze de rechterkant van het canvas hebben bereikt (zie figuur 1.22).



FIGUUR 1.22



Opdracht 16 obfuscator I: schuivende bollen

Een *obfuscator* is een klein programmaatje dat programmeercode onleesbaar kan maken. Dit kan handig zijn wanneer je jouw code niet met anderen wil of mag delen.

87. Bekijk *OBF01*. De bijbehorende code is onleesbaar gemaakt m.b.v. <https://obfuscator.io>
88. Open *H1O16.js* in jouw *editor*. Breid deze code uit zodat je hetzelfde resultaat bereikt als in *OBF01*.

1.5 zelf functies maken

In opdracht 15 heb je voor het eerst kunnen zien dat je zelf **functies** kunt maken; in dit geval: `function tekenJos(x,y)`.

Aan de hand van *voorbeeld 6* gaan we functies nader bestuderen.

In figuur 1.23 zie je een functie die een huis tekent. Als je zelf een functie maakt, begin je met `function` gevolgd door de (door jou bedachte) naam van de functie. Achter die naam moet je altijd haakjes `()` zetten, dus ook als de functie geen **parameters** heeft. Tussen de accolades met `{}` zet je vervolgens alle regels code die moeten worden uitgevoerd als de functie wordt aangeroepen.

Let op: de functie maak je buiten de *draw*. Het **aanroepen** van de functie doen we eerst wel in de *draw* met (alleen) `tekenHuis()`;

Pas als je de functie aanroept, wordt het huis ook echt getekend!

figuur 1.24 toont een functie met een **parameter**: `tekenBoom(x)`; . Dit betekent dat je aan de functie een waarde meegeeft waar hij mee aan de slag kan. Deze waarde wordt bijvoorbeeld gebruikt voor de rechthoekige stam van de boom: `rect(x,130,40,130)`;

```
function tekenHuis() {  
  push();  
  strokeWeight(4);  
  stroke('darkgrey');  
  fill('lightgray');  
  rect(100,180,100,100);  
  // zie vb 6 voor meer code  
  pop();  
}
```

FIGUUR 1.23

```
function tekenBoom(x) {  
  push();  
  noStroke();  
  fill('sienna');  
  rect(x,130,40,130);  
  fill('olive');  
  ellipse(x+20,130,100,150);  
  pop();  
}
```

FIGUUR 1.24



FIGUUR 1.25

Het gebruik van parameters heeft enorme voordelen. Het huis wordt altijd op dezelfde plek getekend, maar de plek waar de boom getekend wordt kun je nu zelf sturen door bij het aanroepen een attribuut mee te geven: `tekenBoom(700)`;

Als je één keer een functie hebt gemaakt, kun je hem zo vaak aanroepen als je wilt. Twee of drie bomen tekenen is nu een fluitje van een cent.

Merk op dat we binnen de functies steeds `push()`; en `pop()`; hebben gebruikt. Dit voorkomt dat je per ongeluk tekeninstellingen (zoals de vulkleur) 'aan laat staan' als je de functie hebt gebruikt.



Opdracht 17 functies maken en aanroepen

89. Open *H1O17.js* in jouw *editor*. Deze gebruikt de code van *voorbeeld 6*. Bekijk het resultaat in de *browser*.
90. Voeg drie bomen toe aan de tekening. Vul voor de horizontale positie de attributen 50,150 en 250 in. Zorg dat de bomen achter het huis komen te staan.
91. Maak een nieuwe functie (buiten de *draw*!) door de code uit figuur 1.26 over te nemen.
92. De functie regel `tekenZon` heeft twee parameters. Welke betekenis hebben deze parameters?
93. Voeg de regel `tekenZon(500,1)`; toe na regel 11. Wat verwacht je te zien? Kijk of je voorspelling klopt.
94. Pas de regel aan naar: `tekenZon(mouseX,schaal)`; . Wat verwacht je te zien? Kijk opnieuw of je voorspelling klopt.

```
function tekenZon(x,s) {  
  push()  
  fill('red');  
  scale(s);  
  ellipse(x,200,300,300);  
  pop();  
}
```

FIGUUR 1.26

Opdracht 18 vallende ster

In de theorie hebben we functies met nul en één parameter gezien, maar je kunt zoveel parameters toevoegen aan een functie als je zelf wilt. In deze opgave kijken we naar een functie met drie parameters.

95. Open *H1O18.js* in jouw *editor*. Bekijk het resultaat in de *browser*.
96. In de loopfunctie *draw* staan maar twee regels die de twee functies *background* en *tekenSter* aanroepen. Voeg twee regels toe, zodat bij elke loop de variabele *yPositie* met 1 wordt verhoogd en de variabele *xPositie* met 5 wordt verhoogd.
97. Voorspel wat je gaat zien als je *//* voor *background* zet. Controleer je voorspelling.
98. Misschien herinner je jezelf nog dat de vierde parameter van *background* de mate van doorzichtigheid aangeeft. Haal de *//* voor *background* weer weg en geef de laatste parameter het attribuut 0.1 mee. Bekijk het resultaat. Kun je verklaren wat je ziet?

De functie *tekenSter* heeft op dit moment twee parameters. In regel 3 is een variabele *schaal* gedeclareerd, waarmee we zelf willen regelen met welke grootte de ster wordt getekend. Daarvoor is binnen *tekenSter* al de regel *scale(1)*; toegevoegd.

99. Voer de volgende opdrachten uit:
 - Pas de functie *tekenSter* aan zodat een derde parameter *s* wordt gebruikt voor het tekenen van de ster op een bepaalde schaal
 - Zorg dat er bij de aanroep van de functie *tekenSter* (in de *draw*) gebruik gemaakt wordt van de variabele *schaal*.
100. Zorg dat de variabele *schaal* bij elke loop met 0,05 wordt verhoogd. Bekijk nu het eindresultaat.

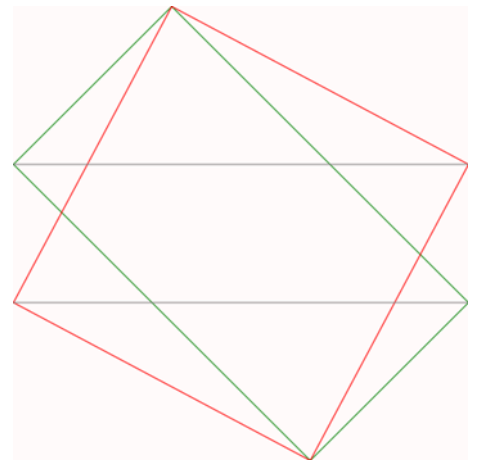
★ Opdracht 19 obfuscator II: lijnenspel

101. Bekijk *OBFO2*. De bijbehorende code is onleesbaar gemaakt m.b.v. <https://obfuscator.io>
102. Open *H1O19.js* in jouw *editor*. Bekijk het resultaat in de *browser*. Je ziet nu alleen twee grijze lijnen.

De bedoeling is dat je *OBFO2* probeert na te bouwen. Om je te helpen is *function tekenLijnen(p)* alvast voor je gemaakt. Hierin zie je voor ons nog onbekende code *line(0,p,width,p)*. Gebruik eventueel de *reference* om de precieze werking te achterhalen.

In de code is al een begin gemaakt met de twee functies *tekenRechthoek(p)* en *tekenVierkant(p)*.

103. Vul beide functies aan zodat het beeld overeenkomt met *OBFO2*.
104. Wat is de betekenis van de regels 16 t/m 18, denk je?



FIGUUR 1.27

Opdracht 20 JOS laten groeien

Tot nu toe heb jij als programmeur beslist hoe de tekeningen eruitzien en hoe vormen bewegen. Het leuke van spellen is nu juist dat de speler zelf iets kan veranderen. In deze opgave maken we voor het eerst kennis met coderegels die daarvoor kunnen zorgen. In de volgende paragraaf krijg je er meer uitleg bij.

105. Open *H1O20.js* in jouw *editor*. Bekijk het resultaat in de *browser*.
106. Bekijk de coderegels. We maken hier opnieuw gebruik van dezelfde functie om JOS te tekenen.
107. Sommige programmeerregels lezen als Engelse zinnen. Wat betekent *if (keyIsPressed == true)* denk je?
108. Hoe zou jij het woord *else* vertalen?
109. Verander deze regel in *if (mouseIsPressed == true)*. Dus: verander het woord *key* in *mouse*. Welk resultaat verwacht je?
110. Wat gebeurt er als de variabele *zoomniveau* kleiner dan 0 wordt?

1.6 Voorwaarden: if en else

Als je opdracht 20 hebt gemaakt, *dan* heb je al gewerkt met voorwaarden. Bijna alle computerprogramma's reageren op de wensen van de gebruiker van het programma. Ze doen dus niet zomaar iets, maar alleen *als* de gebruiker daar om vraagt.

Wanneer een opdracht niet altijd moet worden uitgevoerd, maar alleen in speciale gevallen, spreek je van een **voorwaarde**. Alleen wanneer aan de voorwaarde is voldaan, wordt de opdracht uitgevoerd. In *voorbeeld 7* zie je een bewegende bal die, *als* hij de rand bereikt, omkeert qua snelheid. De bijbehorende code zie je in figuur 1.28.

Een voorwaarde begint standaard met `if ()`. Tussen de haakjes zet je een voorwaarde of eis waaraan moet worden voldaan om één of meerdere coderegels uit te voeren, zoals `if (x > 880)`. Wat de computer moet doen als aan de eis is voldaan, staat tussen `{ }`.

Als je in *voorbeeld 7* met de muis klikt, wordt de cirkel soms groen, maar soms ook niet, ook al klik je met de muis. Dat komt omdat er een dubbele voorwaarde is gebruikt:

```
if (mouseIsPressed == true && snelheid == 5) { fill('green'); }
```

De snelheid wisselt hier tussen (+) 5 (naar rechts) en -5 (naar links). Alleen als het waar is dat er op de muis wordt gedrukt **en** tegelijkertijd ook geldt dat de snelheid 5 is, wordt de vulkleur groen gebruikt. Als je meerdere eisen wilt stellen, gebruik je `&&`. Merk op dat je om te kijken of de snelheid 5 is een dubbele `=` (`==`) moet gebruiken. Een enkele `=` gebruik je om een waarde aan een variabele toe te kennen.

Als je nog eens kritisch kijkt naar de code in figuur 1.28 dan zie je dat dat er eigenlijk twee voorwaarden zijn waarbij dezelfde handeling moet plaatsvinden (namelijk: snelheid omkeren). Dit kunnen we samenvoegen tot één voorwaarde: als `x` groter dan 880 **of** kleiner dan 120 is, moet de snelheid omkeren. Dat doe je zo: `if (x > 880 || x < 120) { snelheid = -1*snelheid; }`

Als je *of* wilt aangeven gebruik je dus `||`. De elementen `||`, `&&`, `==` en `>` zijn voorbeelden van **logische operatoren**. In de opgaven gaan we er mee oefenen.

In *voorbeeld 7* wordt de cirkel groen als aan bepaalde eisen is voldaan. Maar wat moet er gebeuren als dat niet zo is? Dat staat beschreven in het gedeelte met `else { }`. Een `else` maak je altijd in combinatie met een `if`. De vaste vorm zie je in figuur 1.29.

Merk op dat er achter de `else` geen nieuw voorwaarde komt. De voorwaarde tussen `()` is altijd iets dat WAAR (*true*) of NIET WAAR (*false*) is. Later komen we hier nog uitgebreid op terug.

```
if (x > 880) {
    snelheid = -1*snelheid;
}

if (x < 120) {
    snelheid = -1*snelheid;
}

x += snelheid;
ellipse(x,170,200);
```

FIGUUR 1.28

```
if ( VOORWAARDE ) {
    wat moet er gebeuren
    als het WAAR is?
}

else {
    wat moet er gebeuren
    als het NIET WAAR is?
}
```

FIGUUR 1.29



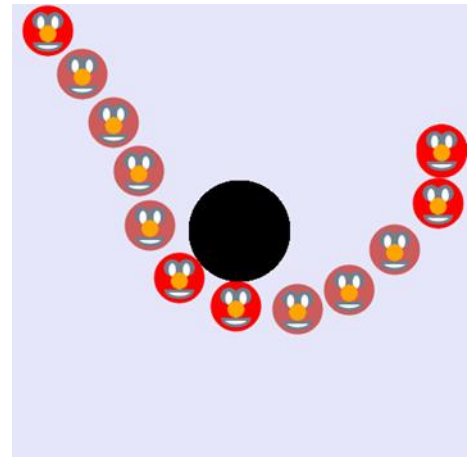
Opdracht 21 oefenen met if & else

111. Open `H1O21.js` in jouw *editor*. Dit is de code van *voorbeeld 7*. Bekijk het resultaat in de *browser*.
112. De code bevat de regels uit figuur 1.28. In de theorie wordt uitgelegd hoe deze twee losse voorwaarden kunnen worden samengevoegd tot één `if`. Voor dat uit in jouw bestand.
113. In de voorwaarde voor de vulkleur staat `&& snelheid == 5`. Voorspel wat je ziet als `==` wordt veranderd in: A) `snelheid < 5`; B) `snelheid > 5`; C) `snelheid >= 5`;
Check vervolgens of je voorspelling klopt.
114. Zorg dat de cirkel groen wordt als er wordt geklikt **of** als de cirkel naar rechts beweegt.
115. Zorg dat de cirkel groen is als hij naar rechts beweegt, blauw als hij naar links beweegt en rood als er iemand klikt.
116. Zorg dat de diameter van de cirkel alleen 100 is als er iemand klikt en weer 200 als er niemand klikt.
117. Wat gebeurt er als je bij `mouseIsPressed == true` in plaats van `true` kiest voor `false`?

Opdracht 22 pas op de randen: afstand bepalen

Bij veel spellen is het belangrijk om vast te stellen of je als speler iets hebt geraakt of dat je er voldoende dichtbij in de buurt bent. Maar hoe stel je nu vast of dit het geval is?

In figuur 1.30 zie je JOS op verschillende plaatsen in een canvas met een zwarte bol. Als je goed kijkt zie je dat hij zowel in de buurt van een rand als in de buurt van de zwarte bol een extra rood gezicht krijgt. Dit gedrag van JOS gaan we in stappen programmeren.



FIGUUR 1.30

118. Open *H1O22.js* in jouw editor. Bekijk het resultaat in de browser. Beweeg hierbij JOS langs alle randen van het canvas. Jos verandert alleen rechts van kleur. De functie *tekenJOS* heeft hiervoor een extra parameter *kleur* gekregen.
119. Bestudeer de code. In welke regels is ervoor gezorgd dat JOS niet buiten beeld kan verdwijnen?
120. Verklaar het gebruik van de waarde 25 in deze coderegels.
121. Hoe dicht moet JOS bij de rand zijn om van kleur te veranderen? Hoeveel pixels is dit? In welke coderegel kun je dit zien?
122. Pas de code aan zodat JOS ook aan de linkerkant rood kleurt als hij te dichtbij de rand komt. Houd dezelfde marge aan als aan de rechterkant.
123. Pas de code verder aan zodat JOS ook boven en onder rood kleurt (met dezelfde marge).

Nu JOS rood kleurt bij de randen van het canvas, is de zwarte bol aan de beurt. In figuur 1.31 bevindt het middelpunt van Jos zich in het punt $[x,y] = [315,105]$. Omdat het canvas 450 pixels breed en hoog is en de zwarte bol in het midden staat, bevindt het middelpunt van de zwarte bol zich in $[x,y] = [225,225]$ ($= 450/2$).

In de onderbouw heb je geleerd hoe je met de Stelling van Pythagoras de afstand tussen twee punten kunt bepalen. Als we in de gele driehoek van figuur 1.31 de horizontale zijde *a* noemen en de verticale zijde *b* dan kunnen we over de schuine zijde *c* zeggen:

$$a^2 + b^2 = c^2$$

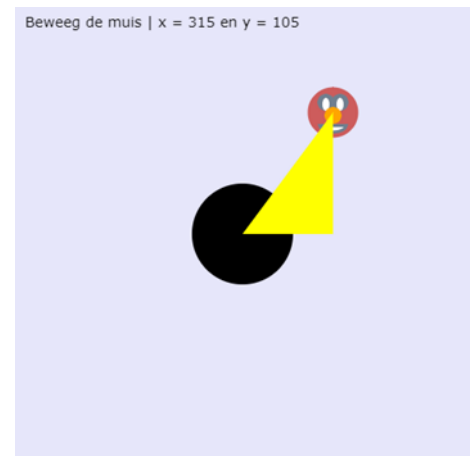
Met $a = 315 - 225 = 90$ en $b = 225 - 105 = 120$ volgt dan voor *c*:

$$c^2 = 90^2 + 120^2 = 22.500 \Leftrightarrow c = \sqrt{22.500} = 150$$

P5 heeft een ingebouwde functie die dit voor je uitrekent: de *dist*-functie (*distance* is Engels voor afstand). De coderegel

afstand = `dist(315,105,225,225)`;

zorgt ervoor dat de variabele afstand nu de waarde 150 krijgt.



FIGUUR 1.31

124. Gebruik bovenstaande coderegel (met `dist`) en pas deze aan zodat de afstand tussen de punt van de muis (de cursor; het middelpunt van JOS!) en het middelpunt van de zwarte bol wordt berekend.

We willen dat JOS rood kleurt als hij zich 5 pixels (of minder) van de zwarte bol bevindt. Dit doen we door de functie *dist* te gebruiken.

125. Leg uit dat de variabele afstand op dat moment **niet** de waarde 5 heeft. Welke waarde heeft de variabele afstand dan wel?
126. Pas de code aan, zodat JOS niet alleen rood kleurt in de buurt van de randen van het canvas, maar ook wanneer hij in de buurt van de zwarte bol komt.
127. Zorg dat bovenin, behalve de reeds getoonde tekst, ook de actuele waarde van de variabele afstand wordt getoond.
128. Pas de code aan, zodat de achtergrond geel (*yellow*) kleurt als JOS zich op de linkerhelft van het canvas bevindt en wit (*white*) kleurt als JOS zich op de rechterhelft van het canvas bevindt. Natuurlijk zorg je ervoor dat de tekst bovenaan nog steeds zichtbaar is!

Opdracht 23 `keysDown`: raak het andere blokje

We hebben al kennis gemaakt met een standaard P5-functie die reageert als er wordt geklikt (bij `mouseIsPressed`). Er zijn ook functies die reageren op het toetsenbord. In deze opgave kijken we naar bij `keyIsDown`. De functie `mouseIsPressed` heeft geen parameter, want er is maar één muis, maar aan bij `keyIsDown` moet je een argument meegeven. In `H1O23.js` hebben we al een beginnetje gemaakt.

129. Open `H1O23.js` in jouw *editor*. Bekijk het resultaat in de *browser*. Druk op de pijltjestoetsen. Welke richtingen werken?

130. Het blokje blijft in het canvas, door regel 23:

```
y = constrain(y, 0, height - 100);
```

Verklaar het gebruik van `height - 100` in deze functie.

131. Pas de code aan, zodat het blokje ook naar links en rechts kan bewegen (met de bijbehorende pijltjestoetsen).

HINT: Gebruik **LEFT** en **RIGHT**.

132. Het blokje kan nu wel links en rechts het canvas verlaten. Blokkeer dit met de functie `constrain`.

Het was je vast al opgevallen dat het rechterblokje fel groen (*chartreuse*) kleurt, als het vierkantje zich op een bepaalde hoogte bevindt zoals in figuur 1.32. Om precies te zijn is gezorgd dat het blokje kleurt als de hoogtes van beide blokjes elkaar (deels) overlappen. Dit is bereikt met `if (y >= 75 && y <= 225)`.

133. Leg uit waarom hier voor de waarden `75` en `225` is gekozen.

134. Waarom staat er in deze coderegel `&&` en niet `||`? Voorspel wat er gebeurt als we dit aanpassen. Controleer jouw voorspelling. Vergeet niet om daarna de `&&` weer terug te zetten.

135. Breid de code uit, zodat het rechterblokje groen kleurt wanneer het geraakt wordt door het vierkant.



FIGUUR 1.32

★ Opdracht 24 jager en prooi

In de vorige opdracht hebben we kunnen zien hoe je de computer kunt laten vaststellen of een vierkant een blokje raakt. Als we het blokje ook (met andere toetsen) kunnen laten besturen, kunnen twee spelers op hetzelfde toetsenbord tegen elkaar spelen. De ene is de jager (vierkant) die de prooi (blokje) moet pakken.

Als uitgangspunt nemen we het eindresultaat van de vorige opdracht. De prooi moet te besturen zijn met de toetsen `w`, `a`, `d` en `s`. Voor de pijltoetsen gebruikten we `keyIsDown(DOWN_ARROW)`. Voor de `a` is dit we `keyIsDown(65)`. Maar hoe weet je nu dat `65` hoort bij de 'a'? De site <http://keycode.info> helpt je daar bij.

136. Open `H1O24.js` in jouw *editor* en bestudeer de code. Let in het bijzonder op de vier variabelen die bovenin gedeclareerd zijn. Deze moet je gebruiken voor deze opdracht.

137. Breid de code uit, zodat een tweede speler het blokje (prooi) kan besturen met `w`, `a`, `d` en `s`.

138. De prooi kan nu het canvas verlaten. Los dit probleem op.

De prooi kan nu bewegen. In figuur 1.33 heeft de prooi zich naar linksboven verplaatst terwijl de jager zich rechtsonder bevindt. Toch kleurt de prooi felgroen, alsof hij geraakt wordt. Begrijp je waarom?

139. Pas de code aan, zodat de prooi alleen felgroen kleurt wanneer de jager hem ook echt raakt.

140. Onderaan is een functie `eindScherm` gemaakt. Zorg dat deze wordt aangeroepen op het moment dat het 'raak' is.



FIGUUR 1.33

★ Opdracht 25 obfuscator III: raak het doel

141. Bekijk `OBF03`. Gebruik hierbij de pijltjestoetsen (links / rechts).

142. Open `H1O25.js` in jouw *editor* en lees de aanwijzingen in de code. Bekijk het huidige tussenresultaat in de *browser*. De bal weerkaatst boven en onder, maar verdwijnt rechts uit beeld.

143. Pas de code aan tot het resultaat van `OBF03`.



FIGUUR 1.34

1.7 vaste herhalingen met een for-loop

In paragraaf 1.5 hebben we functies gebruikt voor het tekenen van een huis en een boom. Het schrijven van een aparte functie is vooral handig, wanneer je de functie vaker wilt gebruiken. In plaats van alle code voor het tekenen van een huis steeds opnieuw in te typen, kun je dan volstaan met het herhalen van de functie `tekenHuis` (figuur 1.35). Dit lukt nog wel voor vier huizen, maar wat nu als je een stad wilt tekenen? We kunnen de computer vragen om een aantal stappen te herhalen met een **for-loop** zoals in figuur 1.36. Deze code doet exact hetzelfde als de code uit figuur 1.35!

Qua vorm lijkt een for-loop op het maken van een voorwaarde met `if`. Eerst is er een gedeelte tussen met `()` waarin staat voor welke situaties er iets moet gebeuren, gevolgd door `{ }`. Hiertussen zet je de code die moet worden uitgevoerd (teken een huis en ga daarna een stukje opzij zodat het volgende huis op andere nieuwe plek komt).

Bij een for-loop zet je tussen de haakjes `()` drie dingen:

- `var n = 1;`
Je kiest een variabele die meestal als een teller gaat werken. Je geeft hem een naam (in dit geval `n` maar het hoeft niet per se één letter te zijn) en een **beginwaarde** (`1`)
- `n <= 4;`
De teller gaat een aantal stapjes doorlopen tot een zekere **eindwaarde**. De code `n <= 4;` is een **voorwaarde**: de herhaling gaat door totdat de variabele `n` de waarde `4` heeft bereikt, ofwel: zolang aan de voorwaarde wordt voldaan.
- `n++;`
Dit betekent: hoog de variabele `n` steeds met 1 op. In de meeste gevallen wordt dit gebruikt, maar bijvoorbeeld `n--` of `n += 3` mag ook.

```
tekenHuis();  
translate(200,0);  
  
tekenHuis();  
translate(200,0);  
  
tekenHuis();  
translate(200,0);  
  
tekenHuis();  
translate(200,0);
```

FIGUUR 1.35

```
for (var n = 1; n <= 4; n++) {  
    tekenHuis();  
    translate(200,0);  
}
```

FIGUUR 1.36

```
for (var n = 1; n <= 4; n++) {  
    tekenHuis();  
    tekenBoom(n);  
    translate(200,0);  
}
```

FIGUUR 1.37

Met de code uit figuur 1.37 loopt de variabele `n` niet langs 1, 2, 3 tot 4, maar van 0 langs 1 en 2 tot 3, want de 4 doet niet mee vanwege het `<=`-teken. Dat zijn nog steeds vier stapjes en dus worden er nog steeds vier huizen getekend! Het is onder programmeurs gebruikelijker om het tellen te beginnen bij 0 (dan bij 1).

figuur 1.37 toont nog een truc die we in de opdrachten zullen gaan gebruiken. De variabele `n` kun je in de herhaling gebruiken voor een berekening of, in dit geval, als attribuut voor de functie `tekenBoom(n)`.



Opdracht 26 oefenen met vaste herhalingen

144. Open `H1O26.js` in jouw editor. Dit is de code van voorbeeld 8. Bekijk het resultaat in de browser.
145. Zet `//` voor regel 12 (`translate(125,0);`) zodat deze niet meer wordt uitgevoerd.
146. Pas de for-loop aan zodat er geteld wordt vanaf 0 tot (en dus niet tot en met!) 5. Bekijk het resultaat.

De functie `tekenBoom(x)` heeft een parameter `x` waarvoor als waarde de waarde van `n` wordt ingevuld. Als `n` groter wordt, wordt de boom ook groter getekend (zie figuur 1.38).

147. Hoe breed en hoe hoog is de ellips waarmee de grootste boom is getekend?
148. Met een paar extra aanpassingen kun je figuur 1.39 nabouwen. Kijk goed wat er allemaal veranderd is en bouw de tekening na.



FIGUUR 1.38



FIGUUR 1.39

Opdracht 27 JOS op herhaling

In deze opdracht gebruiken we een for-loop om JOS meerdere keren op het scherm te tekenen. Afhankelijk van het aantal keren dat we JOS tekenen wordt het canvas hiervoor opgedeeld in blokken van gelijke *breedte*.



FIGUUR 1.40

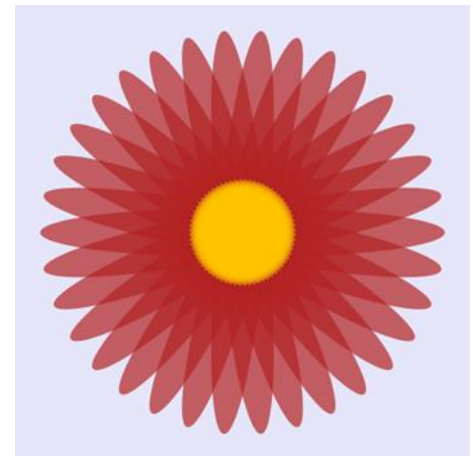
149. Open *H1O27.js* in jouw *editor*. Bekijk het resultaat in de *browser*. Wat gebeurt er als je op het pijltje naar rechts drukt?
150. Welke waarde heeft de variabele *breedte* op dit moment?
151. Zorg dat de waarde van *breedte* afhangt van de variabele *aantal*.
Voorbeeld: als *aantal* = 5, dan moet gelden dat *breedte* = $1000 / 5 = 200$
152. Hoewel we *aantal* nu kunnen verhogen, zien we nog steeds maar vier versies van JOS. Zorg dat dit afhankelijk wordt van de variabele *aantal*.
153. Zorg dat we *aantal* ook (met 1) kunnen verlagen als op het pijltje naar links wordt gedrukt.
154. Het is nu mogelijk dat het aantal gelijk wordt aan 0 of zelfs negatief wordt. Zorg dat *aantal* minimaal 1 blijft.
155. De functie *tekenJos* krijgt als argument *2* mee. Welke betekenis heeft de waarde *2* hier?
156. Wat verwacht je te zien als deze (*2*) wordt veranderd in (*2 + n / 2*)? Controleer je voorspelling.

★ Opdracht 28 draaien II

In opgave 7 heb je kennis gemaakt met de functie *rotate* om dingen te laten draaien. We gaan deze functie gebruiken binnen een for-loop.

157. Open *H1O28.js* in jouw *editor* en bekijk het resultaat. Met behulp van regel 22 wordt nu één *blad* van een rode bloem getekend. Daarna wordt er eenmalig gedraaid m.b.v. regel 23.
158. Voeg een for-loop toe zodat regel 22 en 23 vaker (gebruik de variabele *aantal*) worden herhaald.

De gele binnenkant van de bloem in figuur 1.41 is gemaakt door een vierkant met zijdes van 75 pixels en een reeds in de code gegeven gele vulkleur te draaien op vergelijkbare manier als de rode bladeren.

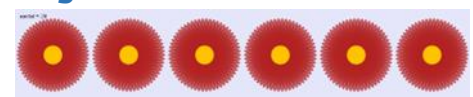


FIGUUR 1.41

159. Maak een aparte herhaling om de gele binnenkant te tekenen.
160. Pas de ellips en het vierkant aan naar eigen smaak en kies je eigen vulkleur. Hoe ziet jouw mooiste bloem eruit?

★ Opdracht 29 obfuscator IV: bloemenrij

De bloemenrij in figuur 1.42 is gemaakt door slim gebruik te maken van het eindresultaat van de vorige opgave. Het volgende is gedaan:



FIGUUR 1.42

- Alle coderegels die gezamenlijk één bloem tekenen zijn samengevoegd in een functie *tekenBloem*.
- Aan deze functie is de functie *scale* toegevoegd, zodat een bloem kleiner kan worden getekend.
- Er is een nieuwe variabele voor het aantal bloemen gemaakt genaamd *Nbloemen*.
- Deze variabele is gebruikt in een for-loop waarin de functie *tekenBloem* wordt aangeroepen.

161. Bekijk *OBFO4* die de code verbergt waarmee de tekening van de bloemenrij in figuur 1.42 is verkregen (met *aantal* = 29).
162. Open *H1O29.js* in jouw *editor*. Dit is het eindresultaat van de vorige opgave.

Als je een programma schrijft is het verstandig dat je dit in stapjes doet, waarbij je steeds even tussentijds controleert of je nog op de goede weg bent. Dit voorkomt dat je te veel fouten tegelijk terug moet vinden.

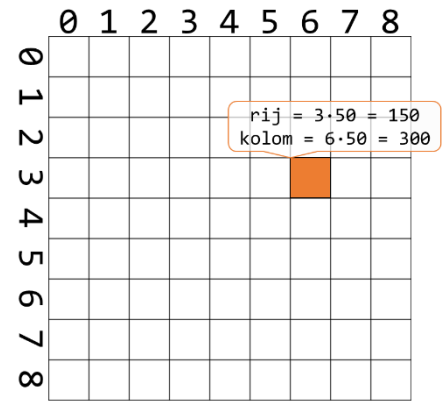
163. Pas de code aan volgens de gegeven stappen, zodat een bloemenrij verschijnt.

Opdracht 30 raster: een herhaling in een herhaling

Er zijn een hoop games en andere programma's, waar het handig is om te werken (en te denken) met een raster. figuur 1.43 toont een raster van 9×9 vakjes. Het canvas is hier 450×450 dus één vakje of cel heeft een zijde van $450 / 9 = 50$.

Bij een raster (b.v. in Excel) spreek je horizontaal van een **rij** en verticaal van een **kolom**. De oranje gemarkeerde cel in figuur 1.43 bevindt zich in de 7^e kolom en de 4^e rij. Let op: omdat we bij 0 beginnen te tellen, hoort hier kolomnummer 6 en rijnummer 3 bij.

Dat lijkt onhandig, maar is het niet, want om een vierkant te tekenen moeten we aan de functie `rect` als parameters meegeven hoeveel we opzij (300) en omlaag (150) moeten gaan voordat we gaan tekenen. Omdat bij 0 beginnen te tellen, kunnen we rekenen met 6 en 3.



FIGUUR 1.43

164. Open `H1O30.js` in jouw editor. Bekijk het resultaat in de browser.

165. Bestudeer de code: er wordt hier één rij getekend met negen kolommen (van één cel).

Het raster in figuur 1.43 is een herhaling in een herhaling: met de eerste herhaling maken we de rij met negen cellen die je nu ziet. Het tekenen van die rij wordt vervolgens negen keer herhaald.

166. Breid de code uit, zodat je een raster krijgt. Gebruik een variabele `rij` op dezelfde manier als nu de variabele `kolom` in de for-loop is gebruikt.

167. Gebruik een `if` binnen de herhaling om dezelfde cel als in figuur 1.43 oranje (*orange*) te kleuren.

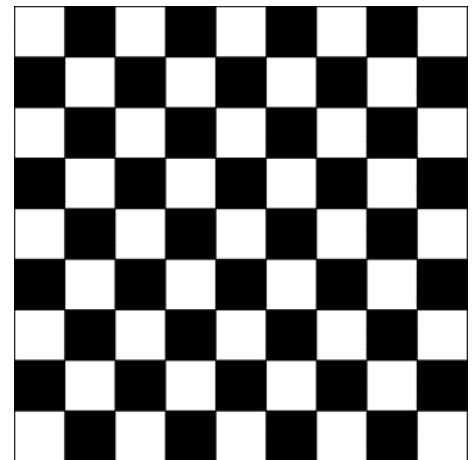
★ Opdracht 31 blokpatronen

In figuur 1.44 zie je een blokpatroon dat is verkregen op basis van de code uit de vorige opgave met de volgende aanpassingen:

- Er is een variabele `kleur` gemaakt voor de vulkleur van de cellen
- Er is een `if - else` aan toegevoegd volgens de redenering:
Als de huidige kleur wit is, dan moet de kleur voor de eerstvolgende cel zwart zijn en andersom

168. Open `H1O31.js` in jouw editor en bekijk het resultaat.

169. Pas de code aan volgens bovenstaande aanpassingen, zodat het patroon uit figuur 1.44 verschijnt.



FIGUUR 1.44

Een dambord is niet 9×9 maar 10×10 met dezelfde structuur.

170. Pas het canvas aan tot 501×501 en maak een dambord.

Opdracht 32 de random-functie

Met de functie `random` kun je de computer een willekeurig getal laten kiezen. Dit getal kun je vervolgens gebruiken om iets op een willekeurige plaats te tekenen of bijvoorbeeld een willekeurige kleur te geven.

171. Open `H1O32.js` in jouw editor. Bekijk het resultaat in de browser. Hoe wordt de celkleur bepaald?

172. Haal de `//` in regel 14 weg en bekijk het resultaat.

Met de functie `random` kiest computer kiest nu zelf een getal tussen 0 en 255 voor de variabele `R`.

173. Zorg dat de computer ook voor de variabelen `G` en `B` een getal tussen 0 en 255 kiest.

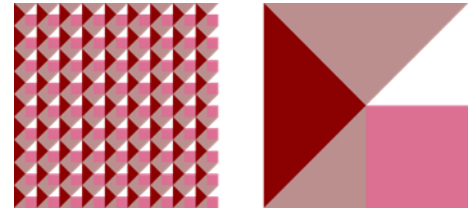
174. Voorspel wat je ziet als je de regel voor de vulkleur verandert in `fill(R,R,R);`.

175. Pas de code aan zodat alle cellen in één rij dezelfde willekeurige kleur krijgen.

Opdracht 33 tegelpatronen

In veel culturen en in het bijzonder de islamitische cultuur zijn decoraties ontstaan die gebaseerd zijn op de herhaling van elementaire afbeeldingen of **tegels**. We kunnen het raster uit de vorige opgaven inzetten om zelf een mooi tegelpatroon te ontwerpen.

176. Open *H1O33.js* in jouw *editor*. Je ziet nu de code waarmee het patroon in figuur 1.45 is getekend.
177. Pas kleuren en vormen aan en ontwerp je eigen tegel.



FIGUUR 1.45



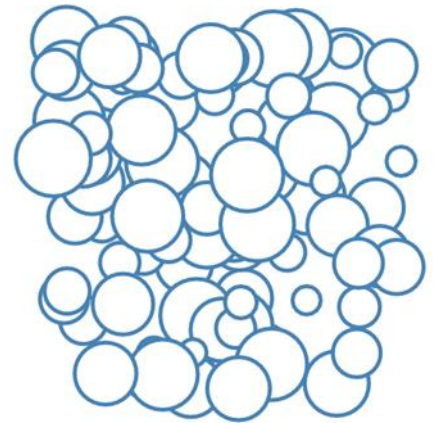
Opdracht 34 obfuscator V: random ringen

De 100 ringen in figuur 1.46 zijn op een willekeurige plek getekend.

178. Bekijk *OBF05* die het resultaat toont maar de code verbergt.

Het programma is gemaakt op basis van de volgende keuzes:

- Het middelpunt van de cirkels is zo gekozen, dat zowel voor de x-waarde als de y-waarde geldt dat dit een willekeurig gekozen waarde is tussen de 50 en 400.
- De diameter van de cirkels is een willekeurige waarde tussen de 25 en 75



FIGUUR 1.46

179. Open *H1O34.js* in jouw *editor*. Hierin is al een begin gemaakt.
180. Breid de code uit zodat deze aan bovenstaande eisen voldoet.
181. Zet `//` voor `noLoop()`; zodat er steeds 100 nieuwe cirkels verschijnen.



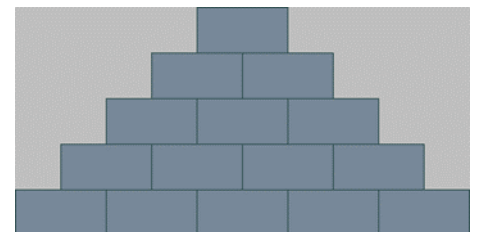
Opdracht 35 piramide

De *stenen* in figuur 1.47 zijn gestapeld in de vorm van een piramide. Er zit een duidelijke vorm van herhaling in het patroon, maar hoe leg je die uit aan een computer?

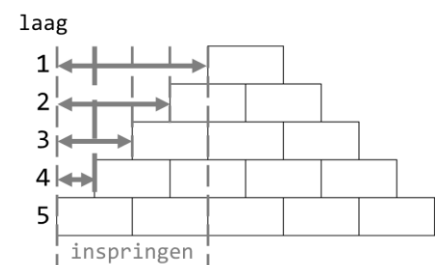
Een logische keus is om de piramide op te delen in lagen en die te nummeren. Aan de hand van figuur 1.48 merken we het volgende op:

- Het aantal stenen in een laag is gelijk aan het nummer van de laag
- Bij alle lagen behalve de onderste moet je eerst een stukje inspringen voordat je de eerste steen kunt tekenen.
- Het inspringen gaat met stapjes van een halve steenbreedte
- Een steen is twee keer zo breed als hij hoog is

182. Open *H1O35.js* in jouw *editor*. Bekijk het resultaat.
183. Vervang `tekenRij(4)` door een herhaling die voor elke naam opnieuw `tekenRij(laag)` aanroept voor elk laagnummer en vervolgens omlaag gaat naar de volgende laag.



FIGUUR 1.47



FIGUUR 1.48

In de functie `tekenRij(aantalStenen)` is een variabele `inspringen`. Het is de bedoeling dat de waarde van deze variabele gelijk is aan het aantal pixels dat je eerst naar rechts moet voor je een rij stenen tekent.

184. Pas de regel met deze variabele aan, zodat wordt berekend hoeveel pixels er naar rechts moet worden ingesprongen voordat de rij met stenen wordt getekend.
185. Verander de waarde van `aantalLagen` naar 10. Krijg je een keurige piramide met tien lagen?

1.8 VERDIEPING: recursie ☆

In de vorige opdracht heb je het tekenen van een piramide geprogrammeerd. In figuur 1.50 staan de coderegels die we daarvoor hebben gebruikt. De code in figuur 1.50 levert dezelfde piramide op.

```
function draw() {  
  for (var laag=1; laag<=aantalLagen; laag++){  
    tekenRij(laag);  
    translate(0, hoogte);  
  }  
}  
  
function tekenRij(aantalStenen) {  
  inspringen =  
    (aantalLagen-aantalStenen)*0.5*breedte;  
  push();  
  translate(inspringen, 0);  
  for (var steen=0; steen<4; steen++) {  
    rect(breedte*steen, 0, breedte, hoogte);  
  }  
  pop();  
}
```

FIGUUR 1.50

Hoewel beide codes hetzelfde eindresultaat opleveren, is de gebruikte programmeertechniek totaal anders.

De variant rechts gebruikt **recursie**. Recursie is een programmeertechniek die je gebruikt in situaties waarbij een opdracht logisch opgedeeld kan worden in meerdere opdrachten die hetzelfde zijn qua vorm, maar dan eenvoudiger.

Kenmerken voor recursie is dat er een functie is die zichzelf aanroept. De functie `tekenPiramide` in figuur 1.50 bevat zelf weer de coderegels `tekenPiramide`. Het idee daarachter is als volgt:

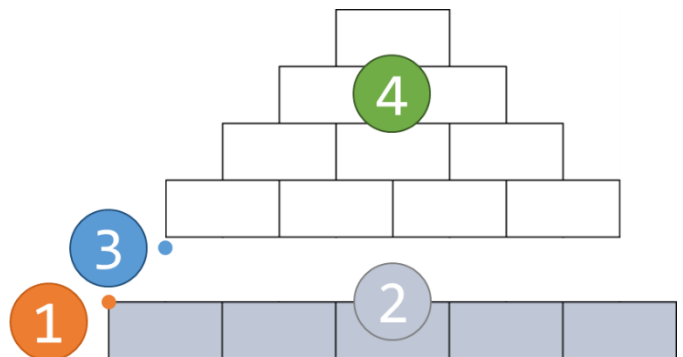
Als je een piramide van $n = 5$ lagen van onderaf begint te tekenen, dan heb je na het tekenen van de onderste laag stenen daarna nog 4 lagen te gaan. Die 4 lagen vormen zelf ook een piramide! Als we na het tekenen van één laag de functie `tekenPiramide` voor $n = 4$ uitvoeren, krijgen we uiteindelijk een piramide van 5 lagen. In stappen (zie figuur 1.51):

- Verplaats (`translate`) de tekenpositie naar het begin van de eerste laag (1)
- Teken de eerste laag stenen van in totaal $n = 5$ (2)
- Verplaats de tekenpositie naar het beginpunt van de bovenliggende laag (3)
- Geef de opdracht om vanaf dat punt een nieuwe piramide te tekenen voor $n-1 = 4$ (4)
- Herhaal dit, zolang de piramide nog niet klaar is, dus zolang $n > 0$.

In figuur 1.50 zie je deze stappen vertaald naar programmeercode. Dit is de code van *voorbeeld 9*. In deze paragraaf gaan we recursie inzetten voor het oplossen van problemen.

```
function draw() {  
  translate(0, height-hoogte);  
  tekenPiramide(aantalLagen);  
}  
  
function tekenPiramide(n) {  
  if (n>0) {  
    for (var nr=0; nr<n; nr++) {  
      rect(nr*breedte, 0, breedte, hoogte);  
    }  
    translate(breedte/2, -hoogte);  
    n--;  
    tekenPiramide(n);  
  }  
}
```

FIGUUR 1.50



FIGUUR 1.51

★ Opdracht 36 piramide

In opdracht 35 heb je een piramide geprogrammeerd. In *voorbeeld 9* wordt dezelfde piramide geprogrammeerd, maar nu met recursie.

186. Open *H1O36.js* in jouw *editor*. Bekijk het resultaat in de browser.

187. Zorg dat een piramide met 10 lagen wordt getekend.

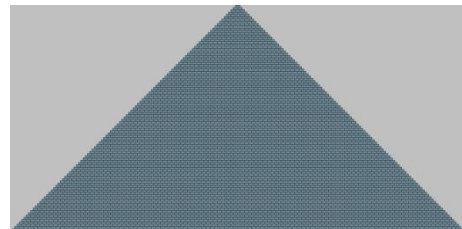
Hoewel de functie `tekenPiramide` zichzelf aanroept, gaat het programma niet oneindig lang door. Dit komt omdat de programmeur een voorwaarde heeft ingebouwd.

188. Wanneer stopt de functie met de uitvoer? Ofwel: wat is de **stopconditie**?

Als we het aantal lagen groter maken, neemt de grootte van het canvas ook toe. Het canvas is op dit moment $10 \times 90 = 900$ pixels breed en 450 pixels hoog. We willen bereiken dat het canvas altijd een grootte van 900×450 pixels heeft en dat het programma zelf uitrekent hoe breed en hoog de stenen dan kunnen worden.

189. Pas de code aan, zodat bovenstaand doel wordt bereikt.

190. Maak een piramide met 100 lagen.



FIGUUR 1.52

★ Opdracht 37 Droste-effect

In figuur 1.53 zie je een wereldberoemd cacaoblik van de Nederlandse firma Droste. Op het cacaoblik staat een serveerster die een dienblad vasthoudt met daarop een cacaoblik met daarop een serveerster die een dienblad vasthoudt met een cacaoblik met daarop... etc. Deze eindeloze herhaling wordt het **Droste-effect** genoemd. Het is een voorbeeld van recursie.

191. Open *H1O37.js* in jouw *editor*. Bekijk het resultaat in de browser. Je ziet een kamer met een deur. Aan de wand hangt een groot zwart schilderij.

We willen dat op het schilderij het beeld van de kamer wordt herhaald. Dit beeld wordt gemaakt met de functie `tekenKamer`. Als we hier recursie willen toepassen, moeten we zorgen dat de functie `tekenKamer` zichzelf aanroept.

192. Voeg aan het eind van de functie de volgende regel toe:
`tekenKamer(0.5);`

Als het goed is zie je nu het Droste-effect.

193. Gaat dit programma oneindig door? Of is er een stopconditie? Zo ja, onder welke voorwaarde stopt dit programma?

We kunnen nu onderdelen toevoegen aan de kamer en kleuren aanpassen, zoals het voorbeeld in figuur 1.54.

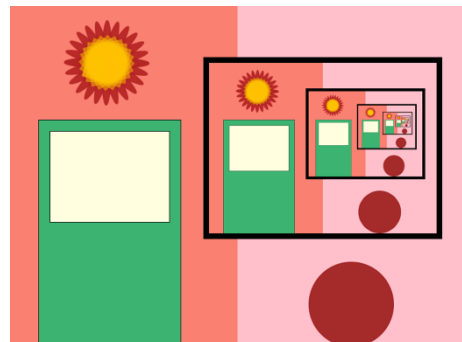
194. Voeg een grote bal toe onder het schilderij. Bekijk het resultaat.

195. Pas de kleuren van de kamer aan, zodat ze passen bij jouw smaak.

196. Voeg minimaal één ander object toe aan de ruimte. In figuur 1.54 hebben wij gekozen voor de bloem die we eerder dit hoofdstuk hebt getekend, maar je kunt ook kiezen voor iets geheel nieuws.



FIGUUR 1.53



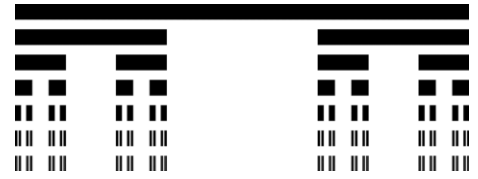
FIGUUR 1.54

★ Opdracht 38 Cantorverzameling

Bij de vorige opdracht hebben we de functie `scale` gebruikt om het Droste-effect te creëren. Dat is een handige truc, maar zeker niet noodzakelijk. In deze opgave kijken we naar de **Cantorverzameling**.

De tekening in figuur 1.55 is gemaakt door een functie `cantor` te schrijven. Deze functie voert de volgende stappen uit:

- Teken een rechthoek met een zekere lengte (breedte canvas)
- Verplaats een stukje naar onderen
- Teken nu twee keer een rechthoek met $1/3$ deel van de vorige lengte, namelijk éénmaal vooraan en eenmaal tot aan het eind, ofwel vanaf $2/3$ deel van de rechthoek erboven
- Herhaal bovenstaande stappen



FIGUUR 1.55

197. Open `H1O38.js` in jouw *editor*. Bestudeer de functie `cantor`. Bekijk het resultaat in de browser.

Op dit moment zien we maar één rechthoek en, voor het gemak, de lengte van die rechthoek. We kunnen van de functie `cantor` een recursieve functie maken door te zorgen dat de functie zichzelf aanroept.

198. Haal de `//` weg in regel 23 en bekijk het resultaat.

199. Kopieer regel 23 naar regel 24 en pas de parameter `x` aan zodanig dat de tweede rechthoek op $2/3$ deel van de vorige rechthoek eveneens verschijnt.

De code bevat een stopconditie. Als de lengte kleiner wordt dan 1, dan stopt de herhaling.

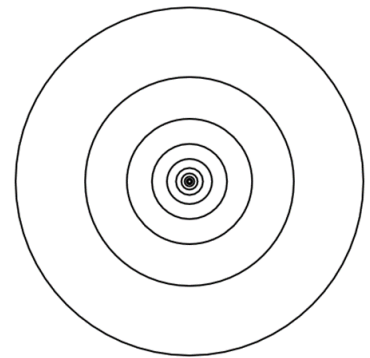
200. Verwijder de stopconditie uit de code, zodat het programma steeds doorgaat.

201. Verklaar dat de herhalingen aan de linkerkant nog te zien zijn, maar aan de rechterkant niet.

★ Opdracht 39 fractal van cirkels

Als je inzoomt op de Cantorverzameling in figuur 1.55, dan zie je eigenlijk hetzelfde patroon als in de huidige uitgezoomde versie. Het maakt niet uit hoe ver je inzoomt: je ziet steeds eenzelfde patroon. Dat is een kenmerk van een **fractal**. Fractals worden doorgaans getekend met behulp van recursie.

De vorm in figuur 1.56 is gemaakt met behulp van recursie.



FIGUUR 1.56

202. Open `H1O39.js` in jouw *editor*. Bestudeer de functie `tekenCirkel`. Bekijk het resultaat in de browser.

Wat is de betekenis van het getal dat linksboven in beeld staat?

203. Pas de stopconditie aan, zodat het programma pas stopt als de diameter van de cirkel 5 is. Hoeveel cirkels worden er nu getekend?

Net als bij de Cantorverzameling kunnen we onze recursieve functie interessanter maken door te zorgen dat de functie zichzelf meer dan eens aanroept.

204. Maak een schets op papier van wat je verwacht te zien als regel 24 wordt aangepast tot:

```
tekenCirkel(x + 0.5*D, y, D*0.5);
```

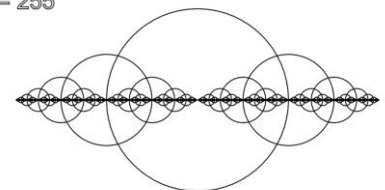
205. Controleer je verwachting.

206. Voeg in regel 25 een vergelijkbare regel toe, zodat, vanuit het midden naar links toe, hetzelfde patroon ontstaat als naar rechts.

207. In de code hierboven staat `D*0.5`. Hoeveel cirkels krijg je als je de `0.5` in `0.6` verandert voor beide regels met `tekenCirkel`?

208. En hoeveel zijn het er voor `0.75`?

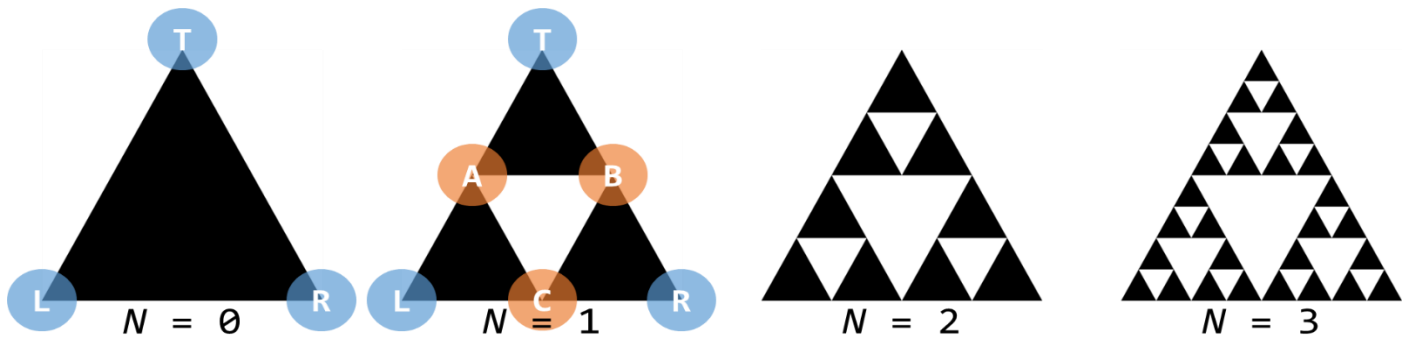
N = 255



FIGUUR 1.57

★ Opdracht 40 Sierpinski-driehoek

Met recursie kunnen we met slechts enkele regels code complexe figuren maken. Een voorbeeld hiervan is de Sierpinski-driehoek. In figuur 1.58 zie je zijn eerste ontwikkelstappen. Herken je het recursieve patroon?



FIGUUR 1.58

Als uitgangspunt nemen we een eerste driehoek ($N=0$) bestaande uit drie punten T (top), L (links) en R (rechts). Voor $N=1$ zien we het patroon van een **Sierpinski-driehoek**: er zijn nu drie driehoeken, getekend vanaf de punten T , L en R en de middelpunten van de zijden A , B en C . Als we voor elk van de drie driehoeken deze stap herhalen krijgen we negen driehoeken ($3 \cdot 3 = 9$; $N=2$) en daarna zeventig driehoeken ($3 \cdot 3 \cdot 3 = 27$; $N=3$) enzovoorts. Dit patroon is zeer geschikt om met behulp van recursie te programmeren!

We hebben de Sierpinski-driehoek al deels voor je geprogrammeerd. In de code zijn voor de in de figuur gemarkeerde punten T , L en R al variabelen gemaakt voor hun x - en y -coördinaten. Datzelfde geldt voor de punten A en B .

209. Open `H1O40.js` in jouw editor. Merk op dat in regel 1 de waarde van N kan worden ingesteld.
210. Bekijk de regels die de x - en y -coördinaten van de punten T , L en R beschrijven. Begrijp je ze?
211. In de functie `sierpinski` zijn de punten A en B beschreven op basis van T , L en R . Voor het punt C kloppen de coderegels nog niet. Pas de regels voor C_x en C_y aan.
212. Bekijk het resultaat in de browser. Als je het goed gedaan hebt zie je de driehoek TAB.
213. Verhoog N naar 2 en vervolgens naar 3. Wat zie je?

De driehoek TAB zie je, omdat de functie `sierpinski` zich in regel 45 zelf aanroept voor de punten T , A en B . Let op: als parameter wordt nu $n - 1$ meegegeven. Dit zorgt voor $n = 0$, waardoor de functie naar het `else`-deel gaat. Daar wordt (pas) de driehoek getekend.

Om het volledige patroon (voor $N=1$ in figuur 1.58) te krijgen moeten we de functie `sierpinski` zichzelf ook laten aanroepen voor de driehoeken ALC en BCR.

214. Voeg twee coderegels toe om dit te bereiken (in de regels 46 en 47).
215. Zet N weer terug naar 1. Voer de code uit en controleer of het resultaat klopt met figuur 1.58.
216. Verhoog N naar 2 en vervolgens naar 3. Komt dit overeen met figuur 1.58?
217. De waarde van N is beperkt tot een maximum van 10. Hoeveel driehoeken zie je dan op het scherm? (Natuurlijk ga je niet tellen, maar rekenen!)

De functie `sierpinski` roept zichzelf nu drie keer aan. Overzie jij in welke volgorde de driehoeken nu worden getekend?

218. Maak een schets op papier voor $N=3$ en geef hierin met een volgnummer aan in welke volgorde de driehoeken volgens jou worden getekend.
219. Zorg dat N de waarde 3 heeft en haal de `//` weg voor regel 50, zodat de driehoeken in een steeds lichtere grijs tint worden getekend. Klopt jouw voorspelling bij de vorige vraag?
220. Maak een functie `kiesKleur` die een willekeurige (random) kleur instelt.
221. Roep de functie `kiesKleur` aan, vlak voor de regel met `triangle`. (De grijs tinten komen hiermee te vervallen).
222. Verander N (niet te hoog!) tot het eindresultaat je bevalt.

H2 OBJECTEN

2.1 Inleiding (camelCase)

In hoofdstuk 1 heb je kennis gemaakt met het programmeren in P5 (Processing). Daarbij lag de nadruk op het toepassen van een aantal basisprincipes zoals variabelen, functies, herhalingen en keuzes die je waarschijnlijk al bij eerdere programmeerlessen had gezien. Misschien is het je opgevallen dat we in de code soms hoofdletters gebruiken en som niet:

```
var naam = 'JOS';           // een variabele (met tekst) bestaande uit één woord
var aantalBomen = 2;        // een variabele (numeriek) bestaande uit twee woorden
noFill();                   // een bestaande P5-functie
createCanvas(50,20);        // een bestaande P5-functie
function tekenStenen() { }  // een zelfgemaakte functie
class Vijand() { }          // een klasse van een object
voldemort = new Vijand()    // een instantie van de klasse Vijand
```

Deze stijl van notatie heet (*lower*) *camel case* (*camelCase*). Belangrijkste kenmerk is dat bij samengestelde woorden of zinnen (zoals *noFill* of *aantalBomen*) de afzonderlijke elementen beginnen met een hoofdletter, behalve de eerste. Deze manier van noteren ken je vast van de *iPhone*. Behalve *lower* bestaat er ook *Upper CamelCase* die je zonder dat je het wist al kent van *PlayStation* en *SpongeBob*.

De laatste twee voorbeelden met *Vijand* wijken af van de afspraak, (want het begint met een hoofdletter). Deze notatie gaan we gebruiken voor het maken van (een klasse van) **objecten**. In dit hoofdstuk leer je wat objecten zijn. We beginnen met een paar bijzondere objecten: afbeeldingen en lijsten.

2.2 afbeeldingen

In figuur 2.1 zie je een beeld uit *voorbeeld 10* waarin twee **afbeeldingen** of **sprites** worden gebruikt: een rij bomen als achtergrond en een kever die door de lucht vliegt. Om die plaatjes vooraf te kunnen laden gebruiken we `preload()`:

```
function preload() {
  bomen = loadImage("images/bomen.jpg");
  kever = loadImage("images/sprites/kever.png");
}
```

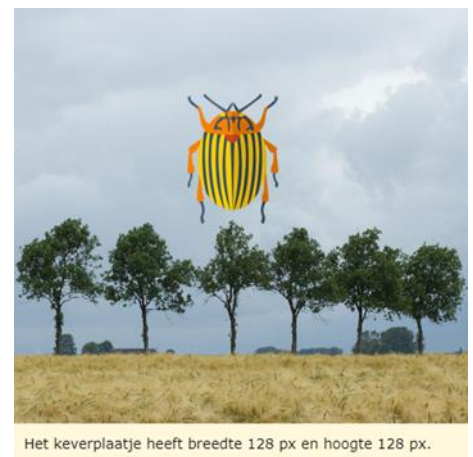
De afbeeldingen staan in aparte bestanden in jouw werkomgeving in de map *images*. Tussen de "" geef je aan waar het plaatje staat. Het bestand *kever.png* is bijzonder, omdat hij een transparante achtergrond heeft, waardoor het lijkt alsof kever door de lucht vliegt.

Net als een getal of een tekst, kun je een plaatje opslaan in een variabele (zoals hier *bomen* en *kever*). In het vervolg van je programma kun je verwijzen naar deze variabelen. P5 weet dan zelf dat het om een variabele van het type afbeelding gaat. Een afbeelding is een voorbeeld van een (bijzonder) **object**.

Kenmerkend voor een object is dat je er **eigenschappen** aan kunt koppelen. Zo heeft een afbeelding een breedte en hoogte in pixels. Door eerst de naam van het object te noemen en met een punt aan de eigenschap te verbinden, kun je de eigenschap gebruiken of opvragen: *kever.width* en *kever.height*.

Voorbeeld 10 laat twee manieren zien om de afbeeldingen ook echt in het canvas te tonen:

`background(bomen);` en `image(kever, 40, 60);`. In de volgende opgaven ga je hier mee werken. Bovendien leer je een aantal extra trucs die je met afbeeldingen kunt uithalen.



FIGUUR 2.1



Opdracht 1 afbeeldingen laden en tonen

1. Open *H2O1.js* in jouw *editor*. Dit is de code van *voorbeeld 10* met enkele toevoegingen. Bekijk het resultaat in de *browser*.
2. Pas regel 20 aan naar `background(kater)`; zodat we als achtergrond het beeld uit figuur 2.2 krijgen.
3. In de `preload` is de bijbehorende jpg-afbeelding geladen. De bestandsnaam is de naam van de kater. Hoe heet hij?
4. We kunnen dezelfde afbeelding ook laden met regel 21. Verwijder de `//` voor `image(kater, 0, 0)`; en bekijk het resultaat. Probeer uit te leggen waarom we niet hetzelfde zien.
5. Verander regel 20 in `background('grey')`; en verander de getallen `0, 0` in regel 21. Welke functies hebben ze?
6. Pas regel 21 aan tot `image(kater, 25, 25, 400, 400)`; . Wat zie je? Wat is de betekenis van `400, 400`?
7. Haal de `//` weg bij regel 24 zodat de kever uit *voorbeeld 10* weer te zien is. Verklaar de beweging die de kever maakt.
8. Zorg dat kever met grootte `30 x 30` (pixels) wordt afgebeeld.



FIGUUR 2.2

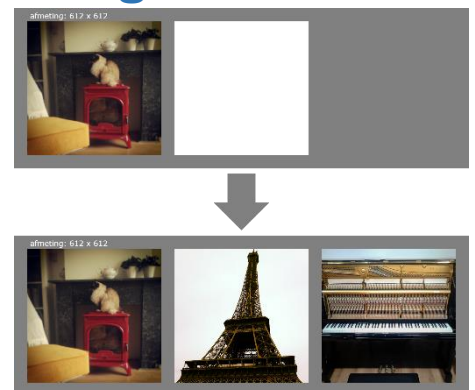
Opdracht 2 meer oefenen met afbeeldingen

9. Open *H2O2.js* in jouw *editor*. Bekijk het resultaat in de *browser*. Je ziet het bovenste deel van figuur 2.3.

In de map *images* heb je behalve het bestand *briecck.jpg* ook afbeeldingen van een toren en een piano.

10. Voeg de afbeelding van de toren toe op de wit gemarkeerde plek. Let op: je moet de afbeelding ook laden met `preload`.
11. Voeg ook de foto van de piano toe. Zorg dat de onderlinge afstand tussen de foto's steeds hetzelfde is.

Boven de foto van de kater staan de afmetingen van de foto in pixels. De breedte en de hoogte zijn eigenschappen van het object `kater`. Ze worden opgevraagd met `kater.width` en `kater.height`.



FIGUUR 2.3

12. Voeg vergelijkbare tekst toe voor de toren en de piano. Welke afbeelding heeft de meeste pixels?

Opdracht 3 bewegende foto I

13. Open *H2O3.js* in jouw *editor* en je *browser*.
14. Wat is de beginwaarde van de variabele `strandX`?

We gaan nu **dezelfde** foto (figuur 2.4) twee keer achter elkaar tonen, zodanig dat de foto's precies op elkaar aansluiten. In regel 16 staat de bijbehorende coderegel:

```
image(strand, strandX + strand.width, 0)
```

15. Wat is de beginwaarde van `strandX + strand.width`?
Ofwel: waar wordt deze foto in eerste instantie getekend?
16. Haal de `//` weg in regel 16 en bekijk het resultaat.
Zie jij waar de foto's op elkaar aansluiten?

We hebben nu een bewegende achtergrond, maar na korte tijd zijn de foto's allebei uit beeld. Moeten we nu een derde foto toevoegen?

Nee! Er zijn nooit meer dan twee foto's tegelijkertijd in beeld. Als de foto's zich over één fotobreedte (`strand.width`; dit is hier ook de breedte van het canvas!) hebben verplaatst, kunnen we de beginsituatie weer herstellen door `strandX` terug te zetten naar de beginwaarde (`0`).

17. Voer deze opdracht uit. HINT: gebruik een *if*-constructie om dit voor elkaar te krijgen.



FIGUUR 2.4



Opdracht 4 obfuscator VI: bewegende foto II

18. Bekijk *OBFO6*. Deze toont de bewegende foto van de vorige opdracht. Als je de pijl naar rechts indrukt, verandert de bewegingsrichting van de foto.
19. Open *H2O4.js* in jouw *editor*. Bekijk het resultaat in de *browser*.

Als je nu de pijl naar rechts indrukt, zie je dat het mis gaat als *strandX* groter is dan 0 (zie figuur 2.5) of kleiner dan -600.

20. Breid de code uit zodat jouw resultaat hetzelfde is als *OBFO6*.

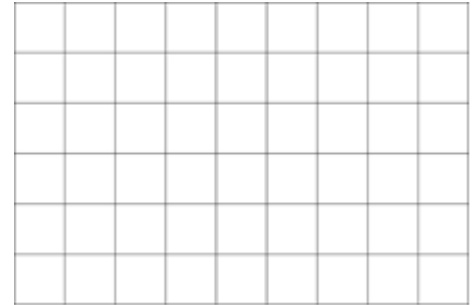


FIGUUR 2.5

Opdracht 5 overloper I: intro

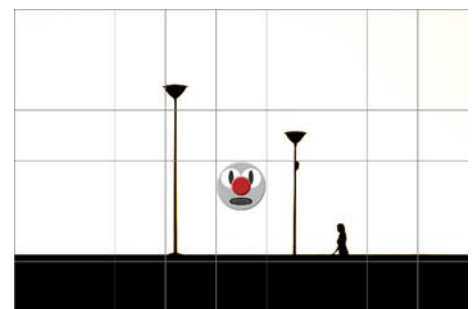
In dit hoofdstuk gaan we in stappen een spel maken dat *overloper* heet. Bij elke stap vertellen we je meer over de inhoud en de regels van het spel. Kortgezegd moet de speler proberen vanaf de linkerkant van het scherm de overkant te bereiken, waarbij hij obstakels moet ontwijken. Als *game character* gebruiken we Jos: onze *Javascript Object Sprite* die je nog kent uit Hoofdstuk 1.

In deze opgave maken we het speelveld en een raster (figuur 2.6) dat ons helpt bij de ontwikkeling van het spel.



FIGUUR 2.6

21. Open *H2O5.js* in jouw *editor*. Bekijk het resultaat.
22. Het raster bestaat uit cellen (vierkantjes) met een lengte die hier *celGrootte* heet. In welke regel wordt deze lengte berekend? Wat is de afmeting van de cellen (in pixels)?
23. De functie *tekenRaster()* tekent op dit moment maar één cel. Maak een dubbele herhaling met een *for-loop* om het volledige raster te tekenen. Volg de instructies in het bestand of kijk terug naar de opgave in hoofdstuk 1 waar je dit hebt geleerd.
24. In de *preload* is een afbeeldingsobject *brug* gemaakt. Voeg deze afbeelding in als achtergrond. Let op: zorg dat het raster over de foto heen wordt getekend.
25. Voeg de sprite *spriteJos* in zodat je het eindresultaat in figuur 2.7 krijgt. Gebruik de variabelen *xJos* en *yJos* om Jos op de juiste plek te krijgen.



FIGUUR 2.7

Opdracht 6 overloper II: interactie met een sprite

26. Open *H2O6.js* in jouw *editor*. Bekijk het resultaat. Druk hierbij op de pijl naar rechts. Hoe groot zijn de stappen die Jos maakt?
27. Breid de besturing uit, zodat Jos ook naar links en naar boven en onder kan.

Als Jos aan de linker- of rechterkant van het canvas is, kan hij niet verder. Dit is gemaakt met de regel:

```
xJos = constrain(xJos,0,width - celGrootte);
```

28. Leg uit waarom de functie *constrain* gebruik maakt van het attribuut *width - celGrootte*.
29. Zorg dat Jos ook boven en onder op het speelveld blijft.

Van sommige stukjes code kun je voorspellen wat ze doen, ook al bevatten ze een functie die je nog niet eerder hebt gezien.

30. Probeer de code in figuur 2.8 te lezen. Voorspel wat het resultaat zal zijn van deze code.
31. Kopieer de code in figuur 2.8 naar jouw programma. Wat gebeurt er? En waar / wanneer precies? Klopt dit met jouw voorspelling?

```
if (xJos == 6*celGrootte
    && yJos == 4*celGrootte) {
    spriteJos.hide();
}
```

FIGUUR 2.8

2.3 lijsten: arrays

In de vorige paragraaf hebben we leren werken met afbeeldingen. Een afbeelding is een bijzonder voorbeeld van een object. Kenmerkend voor objecten is dat ze eigenschappen hebben zoals een breedte (`spriteJos.width`) en dat je ze kunt vragen om iets te doen, zoals zichzelf verbergen (`spriteJos.hide()`). Zie figuur 2.9.

Ook een lijst of **array** is een object. Het biedt de mogelijkheid om een reeks van gegevens die bij elkaar horen op te slaan onder één naam.

Dat zoiets handig kan zijn, laat de code in figuur 2.10 zien. Als je alle namen van je klasgenoten (of spelers van je game!) wilt gebruiken, wordt het wel erg onhandig, als je die namen één voor één in aparte variabelen moet opslaan. De variabele `namenKlas` is hier een array: een lijst met alle namen op één geheugenplek (onder één noemer).

Een array is een reeds bestaande soort object (of klasse). Een nieuwe lijst maak je met `new Array`. Alle elementen van de lijst krijgen een volgnummer. We beginnen weer te tellen bij 0, dus let goed op: het tweede element ("`Alice`" in figuur 2.10) heeft volgnummer 1, want bij "`Bob`" hoort volgnummer 0.

Omdat een array een object is, heeft het net als een afbeelding eigenschappen en kun je hem vragen om iets te doen (figuur 2.11).

Om de inhoud van het derde element (met volgnummer 2) op te vragen uit de array `klas` gebruik je `klas[2]`; en met `klas.length` tel je het aantal elementen in de lijst.

Er zijn heel veel voorgeprogrammeerde handelingen die je aan een array kunt vragen. De belangrijkste voor ons staan in figuur 2.11:

- Met `push()` kun je iets toevoegen aan je lijst. Tussen haakjes vul je in wat je wilt toevoegen aan je lijst. Dit argument komt standaard achteraan in de lijst te staan.
- Met `pop()` verwijder je het laatste element.
- Met `shift()` verwijder je juist het eerste element.
- Met `sort()` sorteer je de array (alfabetisch / numeriek).

Omdat de elementen in een array een volgnummer meekrijgen, kun je ze met een for-loop (zie § 1.7) één voor één bij langs. Door `klas.length` te gebruiken, bekijkt de computer zelf hoe vaak de loop moet worden herhaald:

```
for (var n = 0; n < klas.length; n++) {  
    text(klas[n], 65, 40 * (n + 1));  
}
```

Bovenstaande code levert het resultaat in figuur 2.12. De teller `n` krijgt achtereenvolgens de waarde 0 t/m 3 (dus tot 4), want er wordt geteld *tot* (en dus niet *tot en met*) `klas.length = 4`. De namen staan 65 pixels uit de kantlijn. In dit geval komen de namen onder elkaar, omdat voor de verticale positie opnieuw de teller `n` is gebruikt. Omdat `n` steeds groter wordt, komen de namen steeds verder naar beneden.

```
// eigenschappen  
breedte =  
    spriteJos.width;  
  
hoogte =  
    spriteJos.height;
```

```
// handelingen  
spriteJos.hide();
```

FIGUUR 2.9

```
var naam1 = "Bob";  
var naam2 = "Alice";  
var naam3 = "Eve";  
  
// alle namen in één lijst:  
var klas = new Array("Bob", "Alice", "Eve");
```

FIGUUR 2.10

```
// eigenschappen  
element3 = klas[2];  
  
aantalLeerlingen =  
    klas.length;
```

```
// handelingen  
klas.push("Trent");  
klas.pop();  
klas.shift();  
klas.sort();
```

FIGUUR 2.11



FIGUUR 2.12

✓ Opdracht 7 oefenen met arrays

32. Open *H2O7.js* in jouw editor. Dit is de code van *voorbeeld 11* met enkele aanpassingen. Bekijk het resultaat in de browser. De array *vierkanten* bevat de lengte van een zijde van drie vierkanten.
33. Gebruik de functie `push()` om in regel 12 jouw eigen naam aan de array namen toe te voegen.
34. Voeg in regel 13 een coderegel toe, zodat de array namen alfabetisch wordt gesorteerd. Zie theorie.
35. In regel 18 staat een for-loop. Wat is de grootste waarde die teller zal aannemen?
36. Met regel 21 wordt de omtrek van de vierkanten berekend en getoond. Regel 22 is voor de oppervlakte. Deze regel is nog niet af. Vul de regel aan, zodat de waarden uit figuur 2.13 verschijnen.
37. Voeg in regel 14 een coderegel toe om het eerste element uit de array *vierkanten* te verwijderen. Zie theorie.

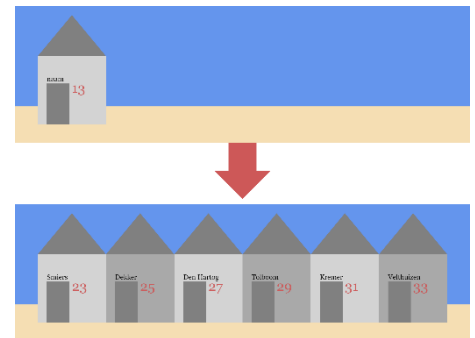
23	■	omtrek = 92	opp = 529
18	■	omtrek = 72	opp = 324
11	■	omtrek = 44	opp = 121
30	■	omtrek = 120	opp = 900

FIGUUR 2.13

Opdracht 8 huizenrij

38. Open *H2O8.js* in jouw editor. Bekijk het resultaat in de browser. Je ziet het bovenste deel van figuur 2.14.

Het is de bedoeling dat we het onderste beeld uit figuur 2.14 gaan programmeren. We zien zes huizen. Alle huizen hebben een nummer en een eigenaar. De bijbehorende gegevens staan in de twee arrays *huisNummers* en *huisEigenaren* (zie bovenaan in de code).



FIGUUR 2.14

39. Maak een for-loop die de huidige regels 18 en 19 herhaalt. Gebruik de lengte van de array *huisNummers* om het juiste aantal herhalingen te krijgen.
40. Alle huizen hebben nummer 13, door het tweede argument van `tekenHuis(kleur, 13)`; Vervang het argument 13 door een stukje code, zodanig dat alle huizen een huisnummer uit de array *huisNummers* krijgen.
41. Voer de volgende stappen uit om de namen van de huiseigenaren op de huizen te krijgen:
 - Voeg een extra parameter naam toe aan de functie `tekenHuis`.
 - Pas de regel `text("naam", 20, 165)`; aan, zodat deze gebruik maakt van deze parameter
 - Pas de for-loop aan, zodat er bij het aanroepen van de functie `tekenHuis` gebruik gemaakt wordt van de array *huisEigenaren*.

De huizen in figuur 2.14 hebben afwisselend de kleur *lightgray* en *darkgrey* hebben. Dat is geprogrammeerd door in de for-loop na het tekenen van een huis toe te voegen:
Als de waarde van de variabele kleur gelijk is aan "lightgray", maak hem dan "darkgrey" en andersom.

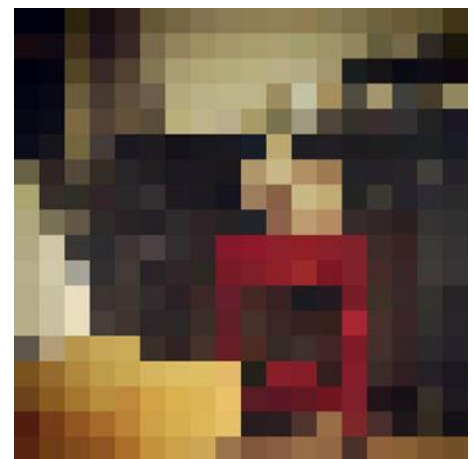
42. Gebruik een if-else om de huizen van grijs tint te laten wisselen.

★ Opdracht 9 obfuscator VII: pixels

43. Bekijk *OBFO7*. Je ziet een mozaïek-versie van de kater in figuur 2.2, net als in figuur 2.15.

Als een foto-object geladen is, kun je met de functie `loadPixels()` van elke pixel van de foto informatie opvragen. *Voorbeeld 12* demonstreert hoe je met `get` kleurinformatie (in RGB-formaat) kunt opvragen. Voor elke pixel bevat het plaatje een array voor de hoeveelheid rood, groen en blauw van de pixelkleur.

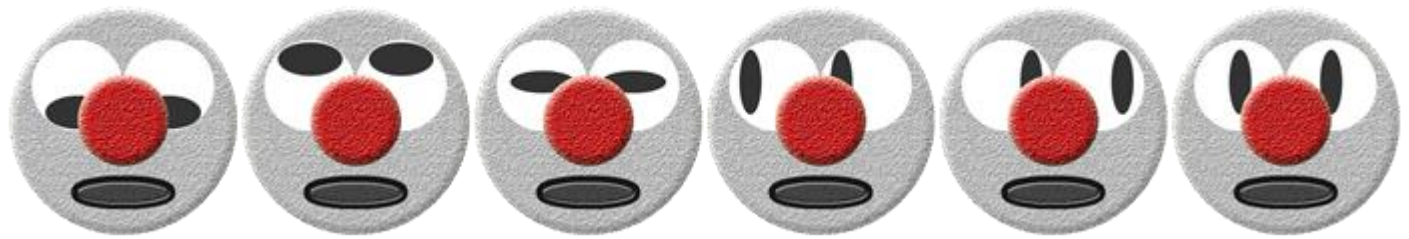
44. Open *H2O9.js* in jouw editor. Dit is *voorbeeld 12*. Bestudeer de code en in het bijzonder de manier waarop kleurinformatie wordt opgevraagd.
45. Bouw op basis van de gegeven code *OBFO7* na.



FIGUUR 2.15

2.4 animatie: een array van plaatjes

Een **sprite** kan één plaatje zijn, maar de term sprite wordt meestal gebruikt voor een bewegende plaatje of animatie. De beweging of animatie ontstaat door verschillende beeldjes of **frames** achter elkaar te tonen. Vroeger werden voor het maken van tekenfilms 24 frames getekend voor maar één seconde film. Als je de frames goed op elkaar laat aansluiten, creëer je daarmee de illusie van een vloeiende beweging.



FIGUUR 2.17

Hierboven zie je zes beeldjes van Jos, die we nummer van 0 t/m 5. In voorbeeld 13 zijn deze beeldjes gebruikt om een array van plaatjes te maken. De gebruikte code staat in figuur 2.16:

- o `var animatie = [];`
Met `[]` maak je een nieuwe, lege array.
- o `loadImage("Jos-" + b + ".png");`
In de for-loop wordt steeds een ander beeldje geladen, want de variabele `b` loopt van 0 t/m 5. We laden daarom achtereenvolgens de bestanden `Jos-0.png` t/m `Jos-5.png`.
- o `animatie.push(nieuw_beeldje);`
De functie `push` voegt steeds een nieuw element toe aan de array `animatie`. Na het uitvoeren van de for-loop hebben we dus een lijst met beeldjes.

```
var animatie = [];           // maak een lege array
var aantalBeeldjes = 6;

function preload() {
  for (var b = 0; b < aantalBeeldjes; b++) {
    nieuw_beeldje =
      loadImage("Jos-" + b + ".png");
    animatie.push(nieuw_beeldje);
  }
}
```

FIGUUR 2.16

De volgende stap is om die beeldjes of frames achter elkaar te tonen. In hoofdstuk 1 hebben we kennis gemaakt met de loopfunctie `draw` die steeds opnieuw wordt uitgevoerd. De frequentie – het aantal frames per seconde – stel je in met `frameRate(2)` (voor 2 frames per seconde; elke frame duurt dus $1 / 2 = 0,5$ seconde). P5 kent een standaardvariabele `frameCount` die bijhoudt hoe vaak de loopfunctie is uitgevoerd vanaf het moment dat het programma geladen is.

In figuur 2.18 zie je hoe je in de `draw` steeds een ander beeldje kunt laden:

- o `nummer++;`
We gebruiken een variabele om bij te houden wat het juiste nummer van het frame is. Als het plaatje getoond is wordt het nummer met 1 verhoogd.
- o `image(animatie[nummer], 0, 0);`
laat het juiste plaatje uit onze array met beeldjes zien. Voorbeeld: `animatie[3]` is het 4^e (!) plaatje.
- o `if (nummer == aantalBeeldjes) {`
 `nummer = 0;`
}
Het laatste plaatje heeft nummer 5, dus nummer 6 bestaat niet. Als we bij dit nummer zijn, moet hij opnieuw beginnen bij 0.

```
var nummer = 0;

function draw() {
  image(animatie[nummer], 0, 0);
  nummer++;
  if (nummer == aantalBeeldjes) {
    nummer = 0;
  }
}
```

FIGUUR 2.18

In de volgende paragraaf leren we een andere tactiek kennen, waarbij alle beeldjes in één bestand staan.

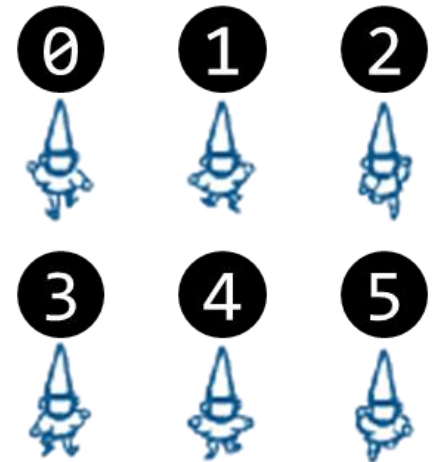


Opdracht 10 oefenen met animaties

46. Open *H2O10.js* in jouw *editor*. Dit is de code van *voorbeeld 13*. Bekijk het resultaat in de *browser*.
47. Voorspel wat je ziet als je regel 23 aanpast tot `image(animatie[nummer], 80, 160, 300, 300);`
Controleer je voorspelling.
48. Wat verandert er aan wat je ziet wanneer je de *framerate* ophoogt van 2 naar 5?
49. Als we alleen de eerste drie beeldjes (zie figuur 2.17) in de array laden, bewegen de ogen van Jos alleen nog van beneden naar boven (en weer terug). Verwerk dit in jouw programma.

Opdracht 11 frameCount en modulus

Op het internet zijn verschillende programma's te vinden om zelf sprites mee te tekenen. Ook zijn er talloze creatievelingen die hun werk met de wereld delen, zoals [Mitchell Vizensky](#). Van hem gebruiken we een serie beeldjes van een tovenaars (bron). De beeldjes zijn al voor je gedownload in jouw werkomgeving in de submap *images/sprites/wizard/opdracht_11A* met zes beeldjes (figuur 2.19).



FIGUUR 2.19

50. Open *H2O11.js* in jouw *editor* en bestudeer de code.
51. Leg uit wat de betekenis is van de regel:
`breedte = animatie[0].width;`
52. De gebruikte plaatjes zijn maar klein. Ze worden groter op het canvas getoond dan ze werkelijk zijn. Hoeveel keer groter?

In de theorie maken we gebruik van een variabele *nummer* om het juiste beeldje uit de array te halen. Met een truc kunnen we ook gebruik maken van *frameCount*. Deze wiskundetruc heet **modulus**.

De modulus is een bovengrens. In ons geval **6** (`aantalBeeldjes = 6`). Met modulo-rekenen (hiervoor wordt het procentteken gebruikt) kunnen we nu de *restwaarde van een deling* bepalen. Dat klinkt ingewikkeld maar is met een paar voorbeelden is het te begrijpen:

- 10 % 6 = 4** : Als je 10 deelt door 6, dan past de 6 er één keer in en houd je nog **4** over.
- 12 % 6 = 0** : Als je 12 deelt door 6, dan past de 6 precies twee keer. Je houdt geen (**0**) restant over.
- 5 % 6 = 5** : Als je 5 deelt door 6, dan past de 6 er nul keer in en houd je dus **5** over.

De *frameCount* bevat het aantal keren dat de loopfunctie *draw* is uitgevoerd en wordt dus steeds met 1 verhoogd. Als we als volgnummer voor de array met beeldjes `frameCount % aantalBeeldjes` invullen, dan is de uitkomst achtereenvolgens 0, 1, 2, 3, 4, 5 en dan opnieuw 0, 1, 2, 3, 4, 5, etc.

53. Vervang jouw *draw* door de code in figuur 2.20. Bekijk het resultaat. (De regels voor de achtergrond en de teksten zijn niet veranderd.)

```
function draw() {
    background('lavender');
    nummer = frameCount % aantalBeeldjes;
    image(animatie[nummer], 150, 0, breedte, hoogte);
    text("frameCount=" + frameCount, 5, 40);
    text("nummer=" + nummer, 5, 70);
}
```

FIGUUR 2.20

Als je afbeeldingen in je programma laadt, moet je goed letten op de bestandslocatie (de map of *directory*) en de precieze bestandsnaam. Als de computer de afbeelding niet kan vinden omdat je een klein foutje hebt gemaakt, laat hij eindeloos *Loading...* zien.

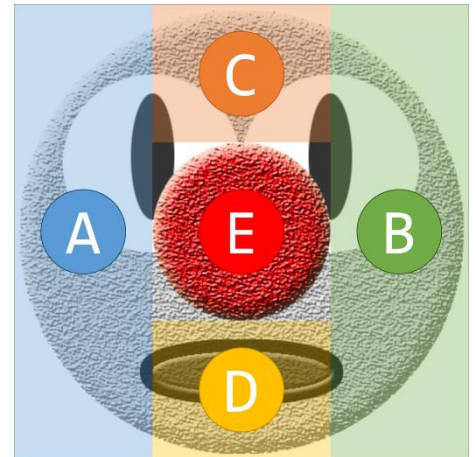
In jouw werkomgeving staat in de map *wizard* in de submap *opdracht_11B*. Deze bevat een andere serie sprites met andere bestandsnamen en een ander aantal.

54. Bekijk de inhoud van de submap *opdracht_11B*. Hoeveel beeldjes zijn er? Hoe is de bestandsnaam opgebouwd?
55. Pas de *preload* aan zodat de beeldjes uit deze nieuwe map worden geladen. Vergeet niet om voor de variabele `aantalBeeldjes` het juiste aantal frames in te vullen! Bekijk het resultaat.



Opdracht 12 obfuscator VIII: kijkrichting

In *OB08* reageren de ogen van Jos op de positie van de muis. De programmeur heeft dit gedaan door de afbeelding van Jos in gedachten op te delen in vijf delen A, B, C, D en E (figuur 2.21). Als de muis in gebied A is kijkt Jos naar links, in B naar rechts, in C naar boven en in D naar beneden. Als de muis zich in E bevindt kijkt Jos recht vooruit (net als wanneer de pagina wordt geladen).



FIGUUR 2.21

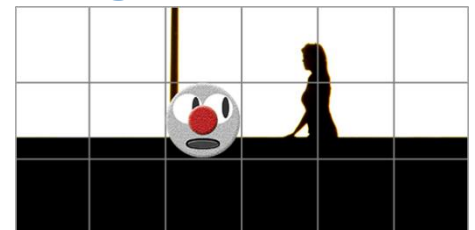
56. Bekijk *OBF08* en beweeg daarbij je muis over de figuur. Stel vast welk plaatje in welke situatie wordt geladen en welk volgnummer van de array animatie hoort bij dit plaatje.

Ter herinnering: voor de positie van de muis heeft P5 de standaardvariabelen `mouseX` en `mouseY`.

57. Open *H2O12.js* in jouw *editor* en bestudeer de code.
58. Bouw op basis van de gegeven code *OBF08* na. HINT:
Gebruik *if* en *else* om de variabele nummer de juiste waarde te geven.

Opdracht 13 overloper III: sprites invoegen

We gaan weer een stukje aan ons spel *overloper* (figuur 2.23) toevoegen. We gebruiken de bestanden in de map *images/sprites/Jos100px* om Jos meer tot leven te laten komen.

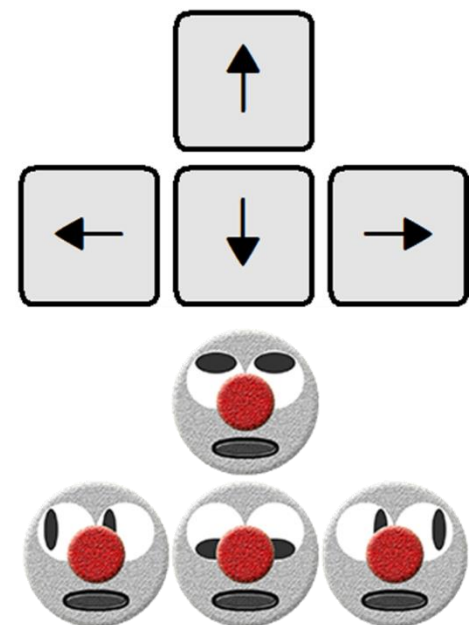


FIGUUR 2.23

59. Open *H2O13.js* in jouw *editor* en bestudeer de code. Dit is het eindresultaat van *overloper II*. Wel is alvast een lege array animatie gemaakt in regel 5 en is regel 14 toegevoegd om het eerste beeldje uit de map te laden.
60. Gebruik een for-loop om alle zes beeldjes uit de genoemde map te laden in de array animatie.
61. In regel 7 is de variabele nummer gedeclareerd met de waarde 3. Gebruik deze variabele om te zorgen dat het plaatje met volgnummer 3 wordt getoond als het programma wordt geladen.

In de vorige *overloper*-opgave hebben we er al voor gezorgd dat Jos te besturen is met de pijltjestoetsen. Het doel van deze opgave is dat aan Jos te zien is naar welke kant hij als laatste gelopen is, zoals in figuur 2.23.

Voorbeeld: Als op de linker pijl is gedrukt, willen we dat Jos vanaf dat moment naar links kijkt (tot er op een andere pijl wordt gedrukt). In figuur 2.24 zie je de code die nodig is om dit te bereiken.



FIGUUR 2.22

62. Pas het programma aan zodat aan bovenstaande eisen wordt voldaan (voor alle bewegingsrichtingen).

De plaatjes die we nu gebruiken zijn allemaal precies 100×100 pixels. Die afmetingen passen precies bij onze keuzes voor de grootte van ons raster en de afmetingen van het canvas. Want: de grootte van één cel is nu exact gelijk aan de grootte van één afbeelding. Wat nu als dit niet zo is?

63. Verdubbel het aantal rijen en kolommen (zie regel 1 en 2).
Bekijk het resultaat: hoe groot is Jos t.o.v. het raster? Hoe beweegt hij nu?
64. Pas de regel met *image* aan, zodat Jos weer precies in één cel past.

```
if (keyIsDown(LEFT_ARROW)) {
    xJos -= celGrootte;
    nummer = 2;
}
```

FIGUUR 2.24

2.5 VERDIEPING: spritesheets ☆

In de vorige paragraaf hebben we geleerd om animaties te maken met een array van plaatjes. In de praktijk wordt vaak gewerkt met maar één grote afbeelding waarin alle losse beeldjes achter elkaar zijn geplaatst, zoals in figuur 2.25. Deze **spritesheet** heeft frames van 460×460 pixels, dus de hele afbeelding is één frame is 4140×920 pixels groot ($9 \times 460 = 4140$ en $2 \times 460 = 920$).



FIGUUR 2.25

Om een afbeelding in het canvas te tonen gebruiken we de functie `image`. We hebben al gezien dat je aan deze functie niet altijd hetzelfde aantal parameters hoeft mee te geven. Ter herinnering:

- `image(kater,10,25);`
Plaats het object (of: de afbeelding) `kater` en toon het (op ware grootte) op positie $x = 10$ en $y = 25$
- `image(kater,10,25,50,40);`
Toon `kater` op positie $x = 10$ en $y = 25$, maar nu met *breedte* 50 en *hoogte* 40

Ons doel is om maar een klein stukje van onze spritesheet te tonen. Hiervoor is extra informatie nodig. Om het **groen gemarkeerde deel** uit de spritesheet van figuur 2.25 te selecteren, gebruiken we:

- `image(spriteSheet,10,25,50,50,1380,0,460,460);`
Toon `spriteSheet` op positie $x = 10$ en $y = 25$, met *breedte* en *hoogte* 50.
Ga binnen de gebruikte afbeelding eerst naar de linkerbovenhoek van het groene vlak:
1380 opzij en 0 naar beneden. Selecteer vanaf dat punt een gebied van 460 bij 460 pixels.

Dit zijn wel erg veel attributen. We zetten het daarom nog eens rustig op een rij voor het **geel gemarkeerde frame** dat je selecteert met `image(spriteSheet,10,25,50,50,2300,460,60,460)`. De betekenis van deze waarden wordt toegelicht in onderstaande tabel waarin we vier delen zien:

Waar op het canvas moet de afbeelding worden geplaatst?		Hoe groot moet de afbeelding worden weergegeven?		Waar zit de linkerbovenhoek van het (gele) vlak?		Welk gedeelte wil je vanaf de linkerbovenhoek selecteren?	
x canvas	y canvas	breedte	hoogte	x sprite	y sprite	breedte	hoogte
10	25	50	50	2300	460	460	460
				5×460	1×460	sBr	sHo
x	y	br	ho	kolom x sBr	rij x sHo	sBr	sHo

De tabel laat zien dat je kunt werken met de volgnummers voor rij en kolom. In *voorbeeld 14* zijn de variabelen zo gemaakt dat de volgende twee regels dezelfde uitkomst hebben:

```
image(spriteSheet,10,25,50,50,2300,460,460,460);
image(spriteSheet,x,y,br,ho,kolom*sBr,rij*sHo,sBr,sHo);
```

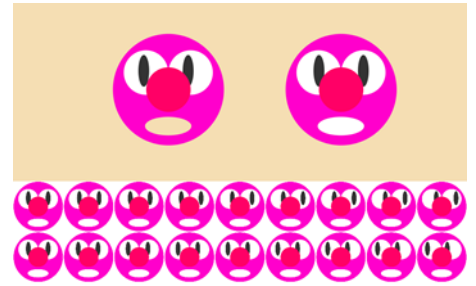
Het gebruik van al die variabelen is in eerste instantie ingewikkelder, maar zorgt wel dat we, als we een andere sprite willen gebruiken, niets meer aan onze code hoeven te veranderen om het te laten werken! Je hoeft alleen nog zelf het aantal rijen en kolommen te tellen en daarna doet de computer de rest.

Als we ook nog de truc met het modulo rekenen uit de vorige paragraaf toepassen, kunnen we het kolomnummer laten oplopen met `frameCount % aantalSpriteKolommen`. Als we de sprite naar links willen laten kijken zoals in de onderste rij (`rij = 1`) van figuur 2.25 dan komen we tot:

```
image(spriteSheet,x,y,br,ho,(frameCount % aantalSpriteKolommen)*sBr,1*sHo,sBr,sHo);
```

Opdracht 14 oefenen met spritesheets

65. Open *H2O14.js* in jouw *editor*. Bekijk het resultaat in de *browser*. Je ziet een variant op *voorbeeld 14* met links een animatie en rechts één frame uit de animatie (figuur 2.26).
66. Bestudeer de code. Welk frame wordt op dit moment rechts getoond in figuur 2.26?
67. Er is één frame waarbij ons *game character* met de naam Alice de inkleuring van de mond mist (zie figuur 2.26 links). Pas regel 38 aan, zodat dit frame steeds aan de rechterkant wordt getoond. Welke getallen moet je invullen?
68. Verander de waarde van de variabele *schaal* in 0.33. Waarom verandert het rechterplaatje niet mee?
69. Hoe snel kun jij wisselen tussen van links naar rechts kijken? Pas de *framerate* aan, zodat Alice net zo snel van kijkrichting wisselt als jij.



FIGUUR 2.26

Opdracht 15 Pony

Online zijn vele spritesheets te vinden. Kijk vooral zelf rond of maak je eigen spritesheet. Hiervoor zijn gratis programma's beschikbaar. In deze opgave gebruiken we de spritesheet *Pony* ([bron](#)) van TrueFrenzy ([Markus Boch](#)). De spritesheet (figuur 2.27) is al voor je gedownload in jouw werkomgeving in de map *images/sprites*.



FIGUUR 2.27

70. Open *H2O15.js* in jouw *editor*. Als je kijkt naar het resultaat in de *browser* dan zie je *Pony* die alleen vooruit loopt. Ofwel: alleen de beeldjes uit de bovenste rij worden doorlopen.
71. Zorg dat de animatie één voor één alle rijen doorloopt en weer terugkeert naar de bovenste rij als alle frames zijn geweest.



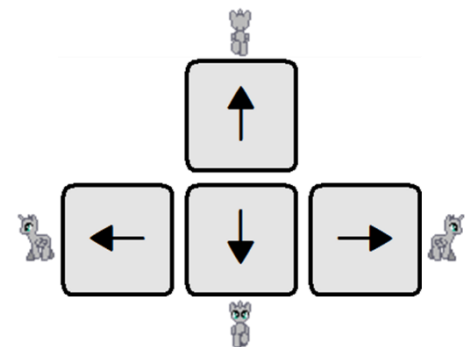
Opdracht 16 obfuscator IX: Pony besturen

De spritesheet in figuur 2.27 bevat vier rijen van animaties die je zou kunnen opvatten als vier looprichtingen. Eerder hebben we al geleerd hoe we een object kunnen laten reageren op de pijltjestoetsen om het over het canvas te laten bewegen.

72. Bekijk *OBFO9* en druk op de pijltjestoetsen. Je kunt *Pony* over het canvas laten lopen.

We gaan *OBFO9* nabouwen. Denk hierbij aan de volgende aandachtspunten en hints:

- De besturing gaat met de pijltjestoetsen. In figuur 2.28 kun je zien welke richting bij welke rij van de spritesheet hoort.
- Als *Pony* van looprichting wisselt, betekent dit voor de animatie dat de waarde van de variabele *rij* moet veranderen.
- Als *Pony* de een rand van het canvas bereikt, kan hij niet verder lopen. Dit is bereikt met de functie **constrain**. Gebruik hierbij de breedte van het frame, zoals die op het canvas verschijnt.



FIGUUR 2.28

73. Bouw *OBFO9* na volgens bovenstaande aanwijzingen.

OBFO9 bevat wel iets vreemds: de animatie gaat door, ook als we geen enkele pijltjestoets indrukken. We willen eigenlijk dat *Pony* helemaal stil blijft staan (op hetzelfde frame) als er niet op een toets wordt gedrukt.

74. Pas jouw code aan zodat ook aan deze extra eis wordt voldaan.

2.6 zelf objecten maken

In de vorige paragrafen heb je kennis gemaakt met de bijzondere objecten afbeelding en lijst (array). Ze zijn bijzonder, omdat hun werking afwijkt van andere objecten, maar belangrijk voor nu is dat we hebben gezien dat ze eigenschappen hebben en dat je ze kunt vragen om een handeling uit te voeren (figuur 2.29).

AFBEELDINGEN	LIJSTEN (ARRAYS)	OBJECT kever
<pre>// attributen: eigenschappen spriteJos.width; spriteJos.height; // methodes: handelingen spriteJos.hide();</pre>	<pre>// attributen: eigenschappen klas.length; klas[2]; // methodes: handelingen klas.sort(); klas.push("Trent");</pre>	<pre>// attributen: eigenschappen kever.x; kever.y; // methodes: handelingen kever.beweeg();</pre>

FIGUUR 2.29

Het maken van programma's met objecten wordt **object-georiënteerd** programmeren genoemd. Het is een bepaalde tactiek van programmeren, of programmeer-**paradigma**, waarbij je eigenschappen en handelingen koppelt onder één noemer: het **object**. We leggen het uit met een voorbeeld:

In *voorbeeld 10* (zie ook § 2.2) laten we een kever over het canvas bewegen. Er is een variabele kever voor de bijbehorende sprite en er zijn variabelen keverX en keverY om zijn positie mee te bepalen. Daarnaast is er code om de kever te laten bewegen.

Eigenlijk is dit onhandig, want dit zijn allemaal *losse* gegevens, terwijl ze eigenlijk *bij elkaar* horen. Net zoals bij de afbeeldingen en lijsten, willen we de kever vragen om informatie over zichzelf te geven of om iets voor ons te doen, zoals in het rechterblok van figuur 2.29. We willen van de kever een **object** maken.

In *voorbeeld 15* is de kever van *voorbeeld 10* nogmaals te zien, maar nu object-georiënteerd. De declaratie van ons object kever zie je in figuur 2.30.

Een eigenschap van een object heet **attribuut**. In dit voorbeeld zijn er drie attributen: x, y en sprite. De eerste twee hebben we meteen een waarde gegeven, maar sprite blijft nog even *leeg* (omdat we daar straks in de **preload** een afbeelding inladen). Als je een attribuut nog niet meteen een waarde meegeeft, kun je dat aangeven met **null**. Let goed op het gebruik van { }, : en , bij het maken van een object. Dat komt heel precies!

Een handeling van een object heet **methode**. Als we de kever vragen om te bewegen via de methode **beweeg()**, dan worden drie regels uitgevoerd. Deze regels gebruiken de attributen x, y en sprite, maar wel steeds voorafgegaan door **this**.

```
var kever = {
  // attribuut
  x: 100,
  y: 150,
  sprite: null,

  // methode
  beweeg() {
    this.x += random(-5,5);
    this.y += random(-5,5);
    image(this.sprite,this.x,this.y);
  }
};
```

FIGUUR 2.30

Het gebruik van **this** is even wennen. Het is een verwijzing naar de eigenaar van een attribuut of methode; in ons geval kever. Het benadrukt dat het gaat om eigenschappen en handelingen van *dit* object. Dat gebruiken van **this** doe je alleen bij het maken van een object. In het hoofdprogramma gebruik je de naam die je aan het object hebt gegeven. Hier een voorbeeld voor het gebruik van een attribuut:

```
text("De kever bevindt zich op x-positie" + kever.x,0,0);
```

en een voorbeeld voor het gebruik van een methode:

```
kever.beweeg();
```



Opdracht 17 werken met objecten

75. Open *H2O17.js* in jouw *editor*. Dit is de code van *voorbeeld 15* met enkele kleine aanpassingen. Bekijk het resultaat in de *browser*.
76. De tekst onderaan verandert op dit moment niet mee als de positie van de kever verandert. Pas de code aan, zodat de actuele waarden van *x* en *y* worden getoond.
77. Voeg een attribuut naam toe aan het object *kever* en geef als waarde een zelfbedachte naam mee. LET OP: omdat een naam een tekst is, moet deze tussen aanhalingstekens.
78. Pas de getoonde tekst aan zodat in plaats van “*Het object kever*” de door jou gekozen naam verschijnt. Gebruik de code *kever.naam* voor het tonen van de naam.

Opdracht 18 Jos als object

79. Open *H2O18.js* in jouw *editor*. Bekijk het resultaat in de *browser*. Je ziet de getekende versie van Jos uit hoofdstuk 1.

In hoofdstuk 1 werd Jos getekend met een zelfgemaakte functie. De coderegels staan nu als methode teken binnen het object *jos*.



FIGUUR 2.31

80. Hoeveel attributen heeft het object *jos*?
81. Wat is op dit moment de waarde van het attribuut *jos.x*?
82. De methode *teken* heeft nu als parameter *muispositieX*. De bedoeling is dat Jos ook echt gaat meebewegen met de muis. Pas de regel *jos.teken(500)*; aan, zodat Jos reageert op de *x*-positie van de muis.

We willen dat Jos groter wordt als hij naar rechts beweegt en kleiner als hij naar links beweegt.

83. Voeg de volgend regel toe aan *teken* in regel 9: *this.schaal = this.x / (0.25*width)*;
84. Zorg dat in de tekst bovenaan behalve de *x*-positie ook de schaal van Jos wordt getoond.
85. Pas de code aan, zodat Jos ook reageert op de *y*-positie van de muis. Doe de volgende stappen:
 - Zorg dat de methode *teken* een extra parameter *muispositieY* krijgt.
 - Gebruik de functie *constrain* om te zorgen dat *y* (van Jos) tussen de 100 en 150 blijft.
 - Zorg dat bij het aanroepen van de methode *teken* de *y*-positie van de muis als parameter wordt meegegeven.

Opdracht 19 overloper IV: het raster als object

In deze opgave gaan we het paradigma van object-georiënteerd programmeren toepassen op het spel *overloper* dat als rode draad door dit hoofdstuk loopt. We beginnen met het raster.

86. Open *H2O19.js* in jouw *editor* en bestudeer de code. Dit is qua werking het eindresultaat van *overloper III*.
87. Maak een object *raster* met de attributen *aantalRijen*, *aantalKolommen* en *celGrootte* en geef ze achtereenvolgens de waarden *6*, *6* en *null* mee.
88. Voeg de methode *berekenCelGrootte()* toe aan het object *raster* volgens het voorbeeld in figuur 2.32.
89. Om de *celGrootte* te berekenen, moet deze methode nog wel worden aangeroepen. Vraag binnen de *setup* het object *raster* om de methode *berekenCelGrootte()* uit te voeren.
90. De *oude* variabele *celGrootte* willen we niet meer gebruiken. Pas alle coderegels die *celGrootte* gebruiken aan, zodat ze het attribuut *raster.celGrootte* gebruiken.
91. Voeg de methode *teken()* toe aan het object *raster*, zodat de regel *raster.teken()*; zorgt voor het tekenen van het raster. Gebruik de coderegels uit de *oude* functie *tekenRaster()* als basis.
92. Welke *oude coderegels* kun je nu allemaal verwijderen?
93. Voeg de regel *raster.teken()*; toe aan de *draw*. Wordt het raster nog steeds getekend?

```
berekenCelGrootte() {
  this.celGrootte =
    width/this.aantalKolommen;
}
```

FIGUUR 2.32

Opdracht 20 overloper V: het bijzondere object jos

In de vorige opgave heb je zelf van het raster een object gemaakt. In dit geval hebben we alvast het object jos voor je gemaakt. Doel van deze opgave is dat je begrijpt wat ze betekenen.

94. Open *H2O20.js* in jouw *editor*. Hoeveel attributen heeft object jos? En welke methodes heeft jos?

Eén van de attributen van jos is de *stapGrootte*. In eerste instantie heeft deze geen waarde (*null*).

95. Door welke coderegel krijgt *stapGrootte* wel een waarde? Welke waarde is dat?

96. Verklaar door naar de code te kijken dat jos bij een beweging naar links een andere afstand aflegt dan in de overige richtingen.

97. Pas de code aan, zodat jos bij een beweging naar links ook een afstand *jos.stapGrootte* aflegt.

Eén van de attributen van jos is zelf een object, namelijk de array *animatie*. In de *preload* worden alle afbeeldingen (frames) die we gebruiken in de array *animatie* geladen.

98. Leg in je eigen woorden uit wat de betekenis is van *jos.animatie.push(frame)*; (regel 61).

99. Doe een voorspelling over wat je op het scherm zal zien als de coderegel

```
text(jos.animatie[3].width,5,15);
```

aan het eind van *draw* wordt toegevoegd. Controleer daarna of je voorspelling klopt.

100. Maak het aantal rijen en kolommen van het raster drie keer zo groot (18 respectievelijk 27).

101. De *draw* is ons hoofdprogramma. Uit hoeveel coderegels bestaat *draw* nog, nu we werken met objecten?

Opdracht 21 zoekspelletje I

Zie je de donkergele cirkel op de foto in figuur 2.33? We gaan een spel maken waarbij de speler de cirkel moet vinden en aanklikken. Te makkelijk? Elke keer dat je goed klikt, verschijnt de cirkel op een andere plek, maar dan kleiner en iets meer doorzichtig.

102. Open *H2O21.js* in jouw *editor* en bestudeer de code. Voer de code uit. Wat gebeurt er als je op een toets drukt?

Het zwarte scherm dat verschijnt als je op een toets drukt, is bedoeld als hulp voor de programmeur (of als *cheat* voor de speler!):

Als je de cirkel kwijt bent, kan het zijn dat je een programmeerfout hebt gemaakt of dat je hem zelf niet kunt vinden. Door op een knop te drukken, kun je vaststellen wat er aan de hand is.



FIGUUR 2.33

103. Verklaar dat de cirkel op een andere plek verschijnt als je de pagina opnieuw laadt.

Het object *cirkel* heeft een methode *controleerRaak()* die de volgende stappen moet gaan uitvoeren:

- Bepalen van de afstand van de muis tot het middelpunt van de cirkel. Hiervoor is al een coderegel: `afstandMuisCirkel = dist(mouseX,mouseY,this.x,this.y);` (Zie de uitleg van *dist* in opdracht 22 van H1 of de *reference*: <https://p5js.org/reference/#/p5/dist>)
- Als het raak is, moet het attribuut *alpha* (die de doorzichtigheid van *cirkel* bepaalt) 20% kleiner worden, ofwel de vorige waarde maal 80%. Bovendien moet de cirkel een nieuwe plek kiezen.

104. Breid de methode *controleerRaak()* uit zodat aan alle eisen is voldaan. Gebruik een *if*-structuur en het feit dat het *raak* is, als de onderlinge afstand kleiner of gelijk is aan de straal van de cirkel. Hint: in hoofdstuk 1 hebben we `if (mouseIsPressed == true)` gebruikt, om te controleren of de speler met de muis klikt.

105. Het object *cirkel* heeft een attribuut *aantalRaak*. Zorg dat deze met één wordt verhoogd, als iemand raak klikt. Maak vervolgens een coderegel in de *draw* die zorgt dat je op het scherm kunt zien hoe vaak je raak hebt geklikt. Gebruik hierbij de *text*-functie.

★ Opdracht 22 natuurlijk bewegen

In *voorbeeld 16* zie je een bal die tegen de wanden van het canvas terugkaatst. De bal is hier een object. De beweging van de bal lijkt misschien wel op een beweging die je vaker in spellen ziet, maar het lijkt niet erg op de beweging van een echte bal. Die stuitert meer zoals in figuur 2.34.



FIGUUR 2.34

In deze opgave gaan we de bal natuurlijker laten vallen en stuiten.

106. Open *H2O22.js* in jouw *editor* en bestudeer de code.
107. Hoeveel attributen heeft object *bal*? En welke methodes heeft *bal*?
108. Wat is de waarde van *bal.y* na het uitvoeren van de *setup*?

In de methode *beweeg* staan de coderegels:

```
if (this.x <= this.straal || this.x >= canvas.width - this.straal) {  
  this.snelheidX *= -this.damping; }
```

Als een bal ergens tegenaan botst, verliest het doorgaans een deel van zijn snelheid. In dit geval verliest hij nog geen snelheid, omdat *this.damping* de waarde *1.0* heeft. Als dit wordt veranderd naar *0.95* verliest de bal 5% van zijn snelheid, want de nieuwe waarde is nog 95% (*0.95*) van de vorige waarde. Merk op: door *canvas.width* te gebruiken, maken we hier gebruik van het feit dat *canvas* zelf ook een object is!

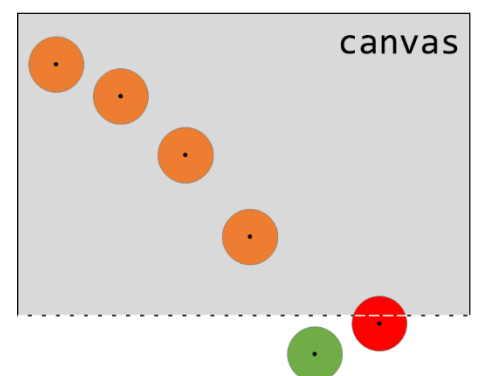
109. Voor welke (getals-) waarden van *this.x* weerkaatst de bal hier?
110. Stel *this.damping* in op *0.95* en bekijk het resultaat.

Als je voldoende geduld hebt bij de laatste opdracht, dan heb je misschien gezien dat de bal op een gegeven moment tegen de rand van het canvas blijft *plakken*. Dat probleem lossen we straks in deze opgave op. Eerst gaan we de beweging van de bal natuurlijker maken.

De bal beweegt op dit moment in een rechte lijn. In het echt valt een bal naar beneden, omdat de aarde eraan trekt. Tijdens het vallen gaat de bal steeds sneller: er is een *versnelling*. Als de bal weer omhoog beweegt, remt hij juist af, tot het hoogste punt waar de snelheid 0 is.

111. Geef het attribuut *snelheidY* de beginwaarde 0 en *damping* de waarde 0.9.
112. Voeg een nieuw attribuut *versnelling* toe met beginwaarde 0.2.
113. Voeg als eerste regel van de methode *beweeg* de volgende coderegel toe:
this.snelheidY += this.versnelling
Bij elke stap in de beweging wordt nu 0.2 opgeteld bij de snelheid van dat moment.
114. Bekijk het resultaat. Hoe zorgt deze regel ervoor dat de bal afremt als hij naar boven gaat?

We zien opnieuw dat de bal op een zeker moment aan de rand van het canvas blijft *plakken*. Hoe kan dat? figuur 2.35 toont schematisch wat er gebeurt. De hoogte van de bal wordt na elke loop van *draw* opnieuw berekend. De bal verplaatst zich hierdoor in stapjes. Op een bepaald moment komt de bal onder de rand van het canvas (groen gekleurd). De verticale snelheid moet dan omkeren, want de bal moet weer omhoog: *this.snelheidY *= -0.95*;



FIGUUR 2.35

Nu kan het gebeuren (het hoeft niet!), dat het middelpunt van de bal bij de eerstvolgende stap nog steeds onder de rand van het canvas zit (rood gekleurd). Het programma zal hierop reageren door opnieuw de snelheid om te keren, waardoor de bal weer naar beneden gaat. Dat was niet de bedoeling! Een manier om dit op te lossen is om de bal precies aan de onderkant van het canvas te plaatsen, op het moment dat deze op de groen gemarkeerde plek belandt: *this.y = canvas.height - this.straal*;

115. Leg uit waarom *this.straal* hier nog van de hoogte van het canvas wordt afgetrokken.
116. Pas de 2^e if-structuur aan zodat er komt te staan: *if (this.y >= canvas.height - this.straal)* { zodat deze alleen nog reageert als de bal onderaan het canvas is.
117. Voeg daarna aan de regel *this.y = canvas.height - this.straal*; toe aan deze if-structuur.
118. Ziet het resultaat er *echt* uit? Pas desnoods de *damping* aan totdat je tevreden bent.



Opdracht 23 obfuscator X: speelkaarten kiezen

Een pak met speelkaarten bestaat uit 52 verschillende kaarten: 13 verschillende kaarten in de soorten klaver, schoppen, ruiten en harten. Als je *OBF10* bekijkt, kun je als speler willekeurig vier kaarten uit het pak van 52 kiezen door met de muis te klikken. Het resultaat ziet eruit zoals in figuur 2.36.

119. Bekijk *OBF10* en klik een aantal keren met je muis.
120. Open *H2O23.js* in jouw editor en bestudeer de code. Bekijk het resultaat in de *browser*. Het scherm toont telkens opnieuw een willekeurig gekozen speelkaart.



FIGUUR 2.36

De code bevat al drie objecten:

- Het object `kaartSoorten` is een array met de vier soorten kaarten in het spel, afgekort met letters. Deze lijst wordt gebruikt voor het laden van de plaatjes.
- Het object `kaartSpeel` is een array met daarin alle 52 afbeeldingen behorende bij de verschillende kaarten. (Die afbeeldingen zijn eigenlijk ook nog objecten!)
- Het object `speler` dat een kaart uit `kaartSpeel` kan trekken (`trekKaart()`) en zijn getrokken kaarten (attribuut `getrokkenKaarten`) op het scherm kan laten zien (methode `toonKaarten`). Hierbij wordt de wiskundefunctie `floor` gebruikt, die alle kommagetallen (die door `random` worden gekozen) naar beneden afrondt. Zie ook de *reference*: <https://p5js.org/reference/#/p5/floor>

De code van *H2O23.js* is nog niet af. Een aantal stappen die nog moeten worden gezet zijn:

- De methode `toonKaarten` toont op dit moment de laatst getrokken kaart, maar moet alle getrokken kaarten netjes op een rij tonen.
- Op dit moment wordt automatisch steeds een nieuwe kaart getrokken, maar dit moet alleen als er wordt geklikt en er minder dan vier kaarten zijn getrokken.
- Als er al vier kaarten zijn getrokken en er toch weer wordt geklikt, moet er een tekst verschijnen.

121. Pas de code aan, zodat het gedrag van jouw programma gelijk is aan *OBF10*.

In theorie is het nu nog mogelijk dat je twee keer dezelfde kaart trekt, zoals in figuur 2.37 (tenzij je daar al aan hebt gedacht!).

122. Breid de methode `trekKaart` uit, zodat deze voorkomt dat er twee keer dezelfde kaart wordt getrokken. Een mogelijke oplossing is: controleer voordat de getrokken kaart aan `getrokkenKaarten` wordt toegevoegd of deze al in de lijst zit.



FIGUUR 2.37

Opdracht 24 dobbelstenen I

123. Open *H2O24.js* in jouw *editor* en bestudeer de code. Voer de code uit. Wat gebeurt er als je met je muis klikt?

De code om het object `dobbelsteen` te maken is best uitgebreid. Dit komt vooral door de methode `teken()` die regelt dat de witte stippen op de steen bij elk aantal ogen op de juiste plaats worden getekend. Deze methode is helemaal klaar.

In de methode `gooi` staat `this.ogen = floor(random(0,6)) + 1`; De wiskundefunctie `floor` rondt alle kommagetallen (die door `random` worden gekozen) naar beneden af.

124. Wat is het verschil in mogelijke uitkomsten tussen `round(random(0,6))` en `floor(random(0,6))`?
125. Geef de dobbelsteen zijden van 200 pixels en zwarte stippen.
126. Maak een attribuut `totaal` dat het totaal van alle worpen bijhoudt en zorg dat dit totaal steeds leesbaar is in het canvas.



FIGUUR 2.38

2.7 een object dat ja of nee antwoordt

Objecten hebben attributen en methodes. Met een methode kan een handeling worden verricht. Dit kan ook betekenen het programma (of een ander object) aan een object iets vraagt en dat het antwoord geeft.

In hoofdstuk 1 heb je misschien *jager* en *prooi* gemaakt, waarin je een vierkantje (de *jager*) met de pijltjestoetsen kan besturen om een rechthoek (de *prooi*) te raken. Als het raak is, verandert de prooi van kleur. Hierbij moet het programma voortdurend de vraag stellen: *raakt de jager de prooi of niet?*

In *voorbeeld 17* is *jager* en *prooi* opnieuw gemaakt, maar nu object-georiënteerde paradigma. Het programma stelt steeds twee vragen:

- *Jager, zit je vlakbij de rand?*
- *Prooi, wordt je geraakt door jager?*

Dit zijn vragen met slechts twee mogelijke uitkomsten: *ja* of *nee* of in het Engels **true** of **false**.

In figuur 2.39 zie de methode `vlakbijRand()` van het object *jager* dat een antwoord teruggeeft met `return`. Dat antwoord is **true** of **false** op basis van de x-positie `this.x` van het object *jager*.

De methode doet zijn werk pas als je hem aanroept. Dat zie je in figuur 2.40: als de *jager* zich te dicht bij de rand van het canvas bevindt, wordt de achtergrond van het canvas rood.

Merk op: tussen de `()` van de `if` staat nu geen vergelijking meer (zoals `hoogte > 150`). In plaats daarvan roepen we de methode `vlakbijRand()` aan. Die geeft antwoord met *ja* of *nee*.

De vraag aan de *prooi* of hij geraakt wordt door de *jager* is anders dan de vraag aan *jager* of hij vlakbij de rand zit, omdat het object *prooi* hiervoor informatie nodig heeft van het object *jager* (waar ben je?).

figuur 2.41 toont het voor nu belangrijkste deel van de code waarmee het object *prooi* is gemaakt. Het attribuut `benGeraakt` is hier geen getal of tekst, maar bevat de waarde **false**. Een attribuut of variabele die waar of niet waar is, heet een **boolean**.

De methode `wordJeGeraakt` heeft een zekere *vijand* als parameter. Dat is een object! Door eigenschappen van die *vijand* te vergelijken met zijn eigen eigenschappen, weet *prooi* of hij geraakt wordt. Is dit het geval, dan krijgt het attribuut `benGeraakt` de waarde **true**. De boolean `benGeraakt` wordt daarna in de methode `teken()` gebruikt om de vulkleur te bepalen.

We vragen de *prooi* of hij wordt geraakt door de *jager* met de coderegel:

```
prooi.wordJeGeraakt(jager);
```

Merk op dat hier als *vijand* *jager* wordt ingevuld.

Voorbeeld 17 is nog niet helemaal af. Dat doen we in de volgende opgave.

```
vlakbijRand() {  
  if (this.x < 10 || this.x > 990) {  
    return true;  
  }  
  else {  
    return false;  
  }  
}
```

FIGUUR 2.39

```
if (jager.vlakbijRand()) {  
  background('red');  
}  
else {  
  background('orange');  
}
```

FIGUUR 2.40

```
var prooi = {  
  x: 800,  
  y: 175,  
  benGeraakt: false,  
  wordJeGeraakt(vijand) {  
    if (vijand.x >= this.x - vijand.zijde) {  
      this.benGeraakt = true;  
    }  
  },  
  teken() {  
    if (this.benGeraakt) {  
      fill('white');  
    }  
    else {  
      fill('green');  
    }  
  }  
}
```

FIGUUR 2.41

✓ Opdracht 25 jager en prooi

127. Open *H2O25.js* in jouw *editor*. Dit is de code van *voorbeeld 17*. Bekijk het resultaat in de *browser*.
128. Bekijk de methode `vlakbijRand()` van het object *jager*. Hoeveel pixels mag de jager maximaal van de rand van het canvas zitten als hij wil voorkomen dat de achtergrond rood kleurt?
129. De methode `vlakbijRand()` werkt nu alleen voor de linker- en rechterrاند van het canvas. Pas de code aan, zodat deze ook voor de boven- en onderrand van het canvas werkt.
130. Ook de methode `wordJeGeraakt` van het object *prooi* is nog niet af. Wat gaat er mis?
131. Vul de methode `wordJeGeraakt` aan, zodat de prooi wit kleurt als hij geraakt is door de jager.
132. We willen dat de prooi weer tot leven komt, als de jager te dicht bij de rand komt. Pas het programma aan, zodat de boolean `benGeraakt` weer de waarde `true` krijgt, als het object *jager* zich vlakbij (of tegen) de rand bevindt.

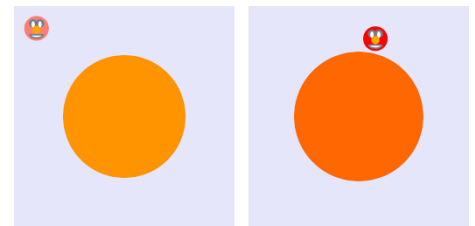
Opdracht 26 heet!

In deze opdracht gebruiken we opnieuw het object *Jos* dat we in opdracht 18 voor het eerst hebben gebruikt. Jos bevindt zich in de buurt van een gevaarlijk vuur. Als hij te dichtbij komt krijgt hij een rood (*red*; zie figuur 2.42 rechts) hoofd. Gaat hij weer verder weg staan dan kleurt zijn hoofd weer zalmroze (*salmon*; zie figuur 2.42 links).

133. Open *H2O26.js* in jouw *editor* en bestudeer de code. Voer de code uit.
134. Zoek uit hoe het object *vuur* van kleur en grootte wisselt.

Op dit moment kleurt het hoofd van Jos nog niet als hij te dichtbij het vuur komt. Hiervoor is binnen het object *jos* al wel een methode `isVlakbij` gemaakt. Deze maakt gebruik van de `dist`-functie.

135. Bestudeer de methode `isVlakbij`. Beschrijf in je eigen woorden wanneer deze methode de waarde `true` als antwoord teruggeeft.
136. In figuur 2.43 staat een if-else-structuur die op de met // gemarkeerde plek in de `draw` moet worden geplaatst. Tussen de haakjes (.....) moet gebruik gemaakt worden van de methode `isVlakbij`.
Pas de code aan zodat het gezicht van Jos rood kleurt als hij te dicht in de buurt van het vuur komt.



FIGUUR 2.42

```
if ( ..... ) {  
    jos.kleur = 'red';  
}  
else {  
    jos.kleur = 'salmon';  
}
```

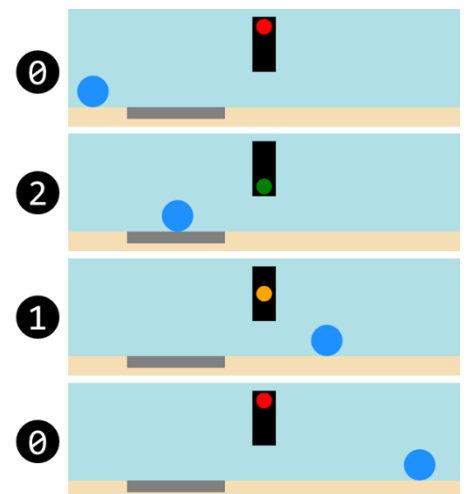
FIGUUR 2.43

★ Opdracht 27 obfuscator XI: stoplicht

In figuur 2.44 zie je een stoplicht dat verbonden is met een detectielus. Als er iets over de grijze rechthoek rolt, gaat het stoplicht op groen (2). Als de lus niets meer detecteert gaat de tijd lopen. Op een bepaald moment gaat het stoplicht op oranje (1) en even later naar rood (0).

Het object *stoplicht* gebruikt de methode `bepaalStand` om het stoplicht op rood, oranje of groen te zetten. Als parameter gebruikt deze methode de boolean `ikVoelIets` van het object *detectielus*. Het is de bedoeling dat `ikVoelIets` van waarde kan veranderen met behulp van de methode `detecteer` van het object *detectielus*.

137. Bekijk *OBF11*.
138. Open *H2O27.js* in jouw *editor* en bestudeer de code.
139. Pas de methode `detecteer` aan zodanig dat jouw programma hetzelfde gedrag vertoont als *OBF11*.



FIGUUR 2.44

Opdracht 28 overloper VI: een vijand voor jos

Bij de introductie van het spel *overloper* is verteld dat het de bedoeling is dat Jos de overkant bereikt en dat hij daarbij obstakels moet ontwijken. figuur 2.45 toont een eerste obstakel: Alice.

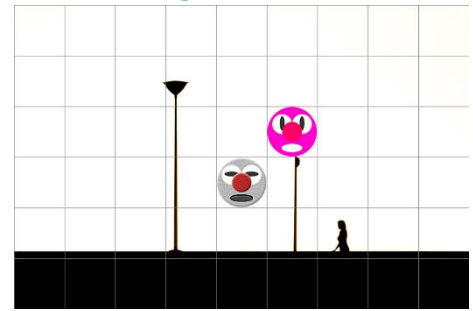
140. Open *H2O28.js* in jouw *editor* en bestudeer de code. Dit is het eindresultaat van opdracht 20, aangevuld met een nieuw object Alice. Kijk in het bijzonder naar de code van dit object.

Alice staat nu stil, maar ook voor haar is er een methode om te bewegen. Hierin wordt de wiskundefunctie `floor` gebruikt, die alle kommagetallen (die door `random` worden gekozen) naar beneden afrondt. Zie ook de *reference*: <https://p5js.org/reference/#/p5/floor>

141. Bestudeer de methode `beweeg()` van Alice en voorspel welk resultaat deze methode zal hebben.
142. Zorg dat Alice gaat bewegen (in de `draw`). Klopt de manier van bewegen met jouw voorspelling?
143. Is het mogelijk dat Alice gedurende een frame stil blijft staan? Leg uit.

Het spel moet stoppen als Jos door Alice wordt geraakt. Om dat te bereiken is in de `draw` de code uit figuur 2.46 toegevoegd die gebruik maakt van de methode `wordtGeraakt` van Jos. We stellen hiermee de vraag of Jos geraakt wordt door Alice. Zo ja, dan stopt het spel.

144. Pas de methode `wordtGeraakt` aan, zodat deze waarde `true` teruggeeft, als Jos en Alice zich op dezelfde plaats bevinden.



FIGUUR 2.45

```
if (jos.wordtGeraakt(alice))
{
  noLoop();
}
```

FIGUUR 2.46

Opdracht 29 overloper VII: gehaald

We hebben al veel gewerkt aan het spel *overloper*, maar op dit moment kun je het spel nog niet winnen. Er zijn vele manieren waarop we dit zouden kunnen programmeren. De vraag *heeft Jos de overkant gehaald* is ook te beantwoorden met ja of nee. Gezien het thema van deze paragraaf kiezen we daarom bij de oplossing voor een boolean gehaald als attribuut van het object jos.

145. Open *H2O29.js* in jouw *editor* en bestudeer de code. Dit is het eindresultaat van opdracht 28.
146. Voeg een attribuut `gehaald` toe aan het object jos en geef deze de beginwaarde `false`.

Op dit moment kan Jos niet buiten het canvas omdat zowel `jos.x` als `jos.y` wordt beperkt door een `constrain`-functie. Voor `jos.x` is dit:

```
this.x = constrain(this.x, 0, canvas.width - raster.celGrootte);
```

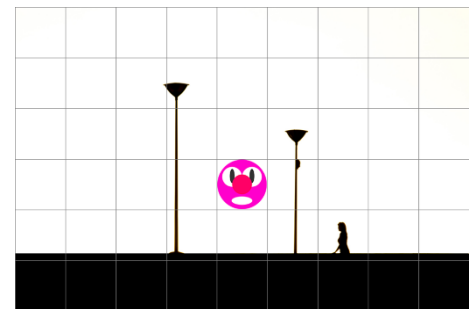
147. Verander de laatste parameter van deze coderegels naar `canvas.width`. Welk resultaat verwacht je? Controleer het resultaat.
148. Wat is de waarde van `jos.x` als Jos rechts uit beeld is verdwenen?

Als Jos rechts uit beeld is verdwenen (zoals in figuur 2.47), moet de boolean `gehaald` de waarde `false` krijgen.

149. Voeg onderaan de methode `beweeg` (van jos) een if-structuur toe die dit regelt. Gebruik het antwoord op de vorige vraag.

Als de overkant is gehaald, willen we dat het spel stopt en in beeld komt te staan dat je hebt gewonnen, zoals in figuur 2.48.

150. Voeg een if-structuur toe aan de `draw` zodat er een eindscherm verschijnt als de overkant is gehaald.



FIGUUR 2.47

Je hebt gewonnen!

FIGUUR 2.48

2.8 een klasse van objecten

Het spel *overloper* loopt als rode draad door dit hoofdstuk. In de vorige paragraaf is een vijand (Alice) toegevoegd. Stel nu dat we een tweede vijand willen toevoegen om het spel lastiger te maken. Dat kan door alle code van het object Alice te kopiëren en daarna een nieuwe naam voor dit object te bedenken. Erg logisch is dat niet, want we krijgen hierdoor allemaal 'dubbele code', bijvoorbeeld voor de methode `beweeg`. En wat als we een kleine aanpassing willen doen? Moet dat dan op twee plaatsen in de code? Veel liever zouden we één basiscode schrijven waar we steeds nieuwe versies van kunnen maken. Zo'n basis heet een **klasse**. Een nieuw exemplaar ervan heet een **instantie** van de klasse.

Instantie, klasse en object zijn lastige begrippen. Je kunt het vergelijken met het maken van voorwerpen met een 3D-printer. Eerst maak je een basis op de computer met een 3D-tekenprogramma. Dit is de klasse. Vervolgens kun je met de 3D-printer zoveel instanties printen als je wilt. Die instanties zijn objecten.

In *voorbeeld 18* zie je de bomen uit figuur 2.49. Het zijn drie objecten van dezelfde klasse. Daarom hebben ze veel overeenkomsten:

- De bomen hebben allemaal dezelfde eigenschappen. Dit zijn de attributen `leeftijd`, `kleur` en `x`. Hun waarde verschilt wel per instantie van de klasse `Boom`!
- De bomen worden allemaal volgens dezelfde tactiek getekend.
- De bomen kunnen allemaal groeien (zie boven en onder).

In figuur 2.50 zie je de code waarmee de algemene basis voor alle bomen, de **klasse**, is gemaakt. Let goed op het gebruik van `{}` en merk op dat er hier niet, zoals bij een object, gebruik gemaakt wordt van komma's. Een klassenaam begint met een hoofdletter.

Een instantie van de klasse `Boom` maak je met een regel zoals:

```
boom1 = new Boom(1, 'olive', 130);
```

Dit kun je lezen als: maak een nieuwe versie (instantie) van de klasse `Boom` en geef dit object de naam `boom1`. Zorg dat de leeftijd van deze boom `1` is, de kleur van de bladeren `'olive'` en de x-positie `130`. Deze drie argumenten verwijzen naar de drie parameters van (de) **constructor** waarmee een nieuwe instantie wordt *geconstrueerd*. De klasse `Boom` heeft hiermee drie attributen `leeftijd`, `kleur` en `x` gekregen.

Met de klasse kunnen we nu eenvoudig nieuwe instanties maken:

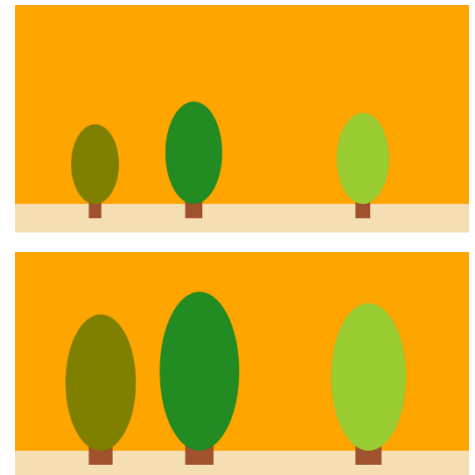
```
boom2 = new Boom(5, 'forestgreen', 300);
```

```
boom3 = new Boom(3, 'yellowgreen', 600);
```

De objecten bestaan op dit moment alleen in het geheugen van de computer. Net als in de vorige paragraaf moeten we methodes aanroepen om de objecten op het scherm te tekenen en in dit geval de boom te laten groeien. In figuur 2.51 zie je dat dit nog steeds gaat zoals je in de vorige paragraaf hebt geleerd.

Wanneer gebruik je een klasse? Daar zijn niet alle informatici het over eens. Sommigen vinden dat je alleen een klasse hoeft te maken als je meer *dezelfde* (vergelijkbare) objecten in je programma gebruikt. Dat scheelt veel dubbel werk en zorgt ervoor dat je in één handeling alle instanties kunt aanpassen.

Anderen stellen dat je altijd een klasse moet maken, ook als je er uiteindelijk maar één object mee maakt. Dit heeft in ieder geval als voordeel dat je werkt met één codestijl voor alle objecten. Mede daarom volgen we in deze module vanaf nu die laatste tactiek.



FIGUUR 2.49

```
class Boom {  
    constructor(td,kl,x) {  
        this.leeftijd = td;  
        this.kleur = kl;  
        this.x = x;  
    }  
    groei() {  
        if (this.leeftijd<20) {  
            this.leeftijd++;  
        }  
    }  
    teken() {  
        // zie bestand  
    }  
}
```

FIGUUR 2.50

```
function draw() {  
    boom1.teken();  
    boom1.groei();  
}
```

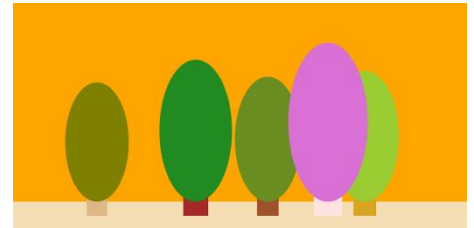
FIGUUR 2.51

✓ Opdracht 30 de klasse Boom

151. Open *H2O30.js* in jouw *editor*. Dit is de code van *voorbeeld 18*. Bekijk het resultaat in de *browser*.
152. De bomen stoppen op een zeker moment met groeien. Welke leeftijd hebben ze dan?
153. Voeg twee objecten toe met de namen *Boom4* en *Boom5*. Kies zelf een geschikte leeftijd, kleur en plek voor de bomen. Natuurlijk zorg je er ook voor dat de bomen op het scherm verschijnen en groeien.
154. Alle stammen hebben nu dezelfde kleur. Welke kleur is dat? Pas de kleur van de stam aan naar *burlywood*.

We willen zorgen dat alle bomen een eigen kleur van de stam kunnen krijgen, zoals in figuur 2.52. Dat kan met de volgende drie stappen:

- Aan de constructor moet een attribuut *kleurStam* worden toegevoegd.
- Bij het aanmaken van instanties van *Boom* moet worden meegegeven wat de kleur van de stam van die (instantie van) *Boom* is.
- Bij het tekenen van de bomen moet gebruik gemaakt worden van het nieuwe attribuut *kleurStam*.

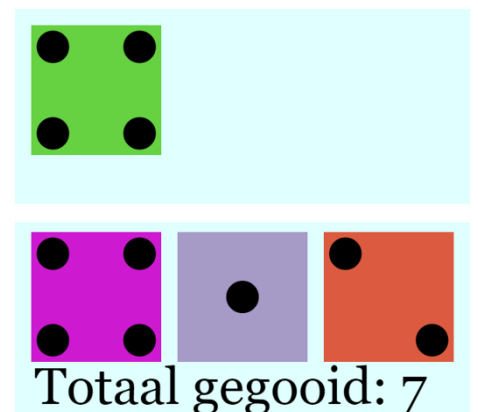


FIGUUR 2.52

155. Voer bovenstaande stappen uit. Kies zelf kleuren voor de stammen van de verschillende bomen.

Opdracht 31 dobbelstenen II

156. Open *H2O31.js* in jouw *editor* en bestudeer de code. Deze bevat het object *dobbelSteen* uit opdracht 24.
157. Pas de code van het object *dobbelSteen* aan naar een klasse *DobbelSteen*. Zorg er hierbij voor dat bij het aanmaken van een instantie van de klasse alleen het attribuut *x* (de *x*-positie van de dobbelsteen) ingesteld hoeft te worden.
158. Maak een instantie van de klasse *DobbelSteen* door aan de setup de volgende coderegel toe te voegen:
`dob1 = new DobbelSteen(25);`
159. Pas de coderegels in de *draw* aan zodat *dob1* wordt gegooid en getekend als er met de muis wordt geklikt.
160. Voeg twee nieuwe instanties *dob2* en *dob3* toe en zorg dat alle drie de dobbelstenen tegelijkertijd gegooid en getoond worden.
161. Breid het programma uit, zodat het totaal aantal ogen van de worp op het scherm wordt getoond, zoals in figuur 2.53.



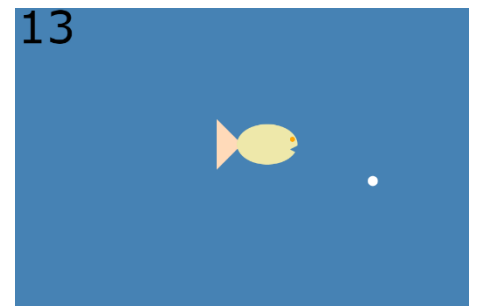
FIGUUR 2.53

★ Opdracht 32 obfuscator XII: (b)-eet

Bij het spel *(b)-eet* bestuur je een vis die een prooi wil opeten. Lukt dat niet, dan ben je af; lukt het wel, dan kun je een tweede prooi vangen.

162. Bekijk *OBF12* en probeer een aantal prooien te vangen door de pijltoetsen voor naar boven en beneden te gebruiken. Als dat lukt gaat de vis sneller zwemmen en wordt de prooi op een nieuwe random verticale plek geplaatst.
163. Open *H2O32.js* in jouw *editor* en bestudeer de code. Bekijk het resultaat in de *browser*.
164. Maak het spel af door de volgende stappen te doen:

- Pas de methode *zwem()* van de klasse *Vis* aan, zodat een vis omhoog en omlaag kan worden bestuurd.
- Zorg dat *garnaal* een nieuwe willekeurig gekozen verticale positie krijgt, als hij is opgegeten.
- Zorg dat de snelheid van gup met 3 toeneemt als hij *garnaal* heeft opgegeten.

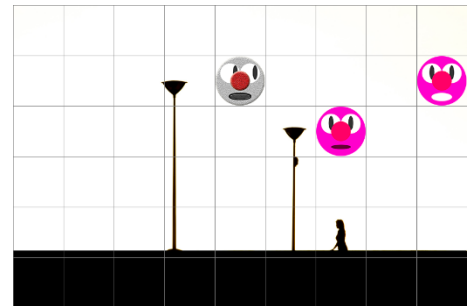


FIGUUR 2.54

Opdracht 33 overloper VIII: werken met klassen

Vanaf nu willen we voor alle objecten werken vanuit een klasse, dus ook voor de objecten in *overloper*. In deze opgave hebben we alvast een begin voor je gemaakt. Er is een klasse *Jos* gemaakt. Het object *eve* is een instantie van de klasse *Jos*. Vanaf nu heet het object waarmee we spelen dus *Eve*! Daarnaast is er een klasse *Vijand* gemaakt. Het object *alice* is een instantie van *Vijand*.

165. Open *H2O33.js* in jouw *editor* en bestudeer de code. Bekijk het resultaat in de browser.
166. Voor het object *raster* is nog geen klasse gemaakt. Schrijf de code om tot een klasse *Raster* en maak hiermee een instantie *raster* (let op het verschil: de klasse is met een hoofdletter, het object is met kleine letter) die wordt gebruikt.
167. Zorg dat het raster weer verschijnt. Bedenk zelf welke aanpassingen hiervoor nog nodig zijn in de *setup* en de *draw*.



FIGUUR 2.55

Om het spel moeilijker te maken, voegen we een tweede vijand toe met de objectnaam *bob*. Bob lijkt op Alice, maar heeft een andere mond (zie figuur 2.55). De bijbehorende afbeelding vindt je in de map: *images/sprites/Bob100px/Bob.png*

168. Voeg het object *bob* toe aan het programma. Zorg er hierbij voor dat hij zijn eigen sprite en startpositie krijgt.
169. Heb je er al aan gedacht dat het spel nu is afgelopen als *Jos* door *Alice* of door *Bob* wordt geraakt? Zo niet, breid dan je programma uit zodat ook aan deze eis is voldaan.

Opdracht 34 overloper IX: een eigen plek

Misschien heb je al gezien dat het in de vorige opgave mogelijk is dat *Alice* en *Bob* zich op dezelfde plek bevinden. Hun afbeeldingen staan dan over elkaar, waardoor het lijkt alsof er nog maar één vijand is. Hoe kunnen we dit voorkomen? Een mogelijk tactiek is de volgende:

- Laat zowel *Alice* als *Bob* bewegen.
- Controleer daarna of *Alice* en *Bob* zich op dezelfde plek bevinden.
- Is dat het geval? Vraag dan aan *Bob* om zich nogmaals te bewegen.

170. Open *H2O34.js* in jouw *editor* en bestudeer de code. Dit is de eindversie van de vorige opgave.
171. Voeg een *if*-constructie in de *draw* toe die controleert of *Alice* en *Bob* zich op dezelfde plek bevinden. Als dit het geval is, moet *bob.beweeg()* nogmaals worden uitgevoerd.
172. Bedenk minimaal twee bezwaren tegen de gekozen aanpak.

Volgende opgave: regel om en om en regel niet op dezelfde plek. Overloper steropdracht? Geef bijbehorende code en laat uitleggen. Ze moeten allebei bewegen, dan is het goed.

Opdracht 35 overloper X: wie is aan de beurt?

In het spel *overloper* kunnen alle objecten nu tegelijkertijd bewegen. We kunnen er een tactisch spel van maken door om en om *Eve* (de speler) en haar vijanden *Alice* en *Bob* (de computer) een stap te laten zetten en te zorgen dat *Eve* niet meer naar links (achteruit) kan.

173. Open *H2O35.js* in jouw *editor* en bestudeer de code. Hoe zie je in de code dat *Eve* niet meer naar links kan bewegen?
174. De klasse *Jos* heeft een nieuw attribuut. Het is een boolean. Wat is de naam van deze boolean? Wat is de beginwaarde?
175. Voeg de code in figuur 2.56 toe aan de *draw* onder *raster.teken()*. Hoe bewegen de drie objecten nu?
176. Welke regel moet op de plaats van *//* worden toegevoegd zodat de objecten om en om bewegen?

```
if (eve.aanDeBeurt) {  
    eve.beweeg();  
}  
else {  
    alice.beweeg();  
    bob.beweeg();  
    //  
}
```

FIGUUR 2.56

2.9 een array van objecten

In de vorige paragraaf heb je kennis gemaakt met klassen. Als je één keer een **klasse** hebt gemaakt, is het heel eenvoudig om een nieuwe **instantie** van die klasse te maken met `new`, zoals met de drie bomen in *voorbeeld 18*.

Door het gebruik van een klasse hoeven we methodes zoals `groei()` en `teken()` slechts één keer te programmeren. Daarna roepen we dezelfde methode voor elke instantie van de klasse `Boom` opnieuw aan, zoals in *figuur 2.57*.

Toch ziet de code er een beetje onhandig uit. En stel nu eens dat je niet drie maar tien bomen hebt zoals in *figuur 2.58*? Of zelfs honderd bomen? In dat geval wil je alle bomen in één handeling vragen om te groeien en niet elke boom apart.

Eerder in dit hoofdstuk hebben we gezien dat je een **array** kunt gebruiken om gegevens die bij elkaar horen onder één objectnaam te bewaren. Voor het maken van de animaties (sprites) hebben we al een lijst van objecten (namelijk afbeeldingen) gemaakt. De code om een lijst van objecten van de klasse `Boom` te maken is vergelijkbaar:

```
var bomen = []; // maakt een lege array

function setup() {
  for (var b = 0; b < 10; b++) {
    bomen.push(new Boom());
  }
}
```

Met de methode `push` voeg je een element toe aan een lijst. Om het extra beknopt te houden staat tussen de `()` van de methode niet een object, maar de opdracht om een object te maken: `new Boom()`. Met de herhaling (`for`) vraag je nu tien keer om een nieuwe boom aan de lijst met bomen toe te voegen. In *voorbeeld 19* is het hele programma uitgewerkt.

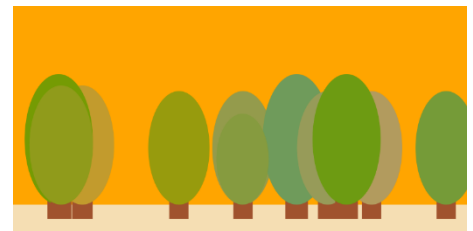
In *figuur 2.57* zie je hoe in *voorbeeld 18* één voor één aan alle bomen werd gevraagd om te groeien. Hoe doen we dat, nu we een hele lijst met bomen hebben? We gebruiken opnieuw een `for`-loop om de lijst (met de naam) bomen te doorlopen; *figuur 2.59* toont de exacte code.

Ter herinnering: met `bomen[3]` duiden we het (vierde!) element uit de lijst aan. In dit geval is dit element een object van de klasse `Boom` met de methodes `groei()` en `teken()`. Omdat de teller `n` loopt van 0 tot en met 9, vragen we hier met slechts een paar regels code aan tien bomen om te groeien en zichzelf te tekenen. En willen we honderd of duizend bomen, dan kan dat nog steeds met deze regels!

```
function draw() {
  boom1.teken();
  boom2.teken();
  boom3.teken();

  boom1.groei();
  boom2.groei();
  boom3.groei();
}
```

FIGUUR 2.57



FIGUUR 2.58

```
for(n=0;n<bomen.length;n++)
{
  bomen[n].teken();
  bomen[n].groei();
}
```

FIGUUR 2.59



Opdracht 36 een array van bomen

177. Open `H2O36.js` in jouw editor. Dit is de code van *voorbeeld 19*. Bekijk het resultaat in de browser.
178. Pas de code aan zodat er niet tien maar twintig bomen worden getekend.
179. Wat verwacht je te zien als aan het begin van de `draw` de coderegel `bomen[10].G = 0;` wordt toegevoegd? Controleer je voorspelling.
180. De bomen kunnen op dit moment 20 jaar oud worden. Pas dit aan naar tien jaar.

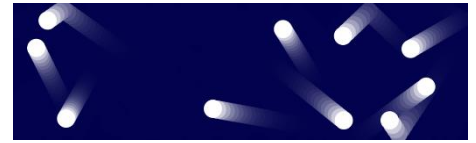
Als bomen tien jaar oud zijn, gaan ze dood. Dit betekent voor het programma dat ze niet meer moeten worden getekend.

181. Gebruik een `if`-constructie in de `for`-loop van de `draw` om dit te programmeren. Als je het programma uitvoert heb je als het goed is aan het eind geen bomen meer over!

Opdracht 37 knikkerbak I

In *voorbeeld 16* is een object `bal` gemaakt dat tegen de wanden van het canvas weerkaatst. Op basis van de code is een klasse `Knikker` gemaakt waarmee we een grote bak met knikkers gaan maken.

182. Open `H2O37.js` in jouw *editor* en bestudeer de code. Bekijk het resultaat in de *browser*.
183. In regel 29 is een lege array `knikkerVerzameling` gemaakt. Vul deze lijst met tien instanties van de klasse `Knikker` op de manier die in de theorie wordt beschreven.
184. Gebruik een `for`-loop in de `draw` om alle knikkers te laten bewegen en te tekenen, zoals in figuur 2.60.



FIGUUR 2.60

Opdracht 38 knikkerbak II: interactie met de muis

We gaan de knikkerbak uit de vorige opgave uitbreiden zodat het programma reageert op de muis, door gebruik te maken van de boolean `mouseIsPressed` die standaard in Processing zit en waar je al eerder kennis mee hebt gemaakt.

185. Open `H2O38.js` in jouw *editor* en bestudeer de code. Voorspel op basis van de code wat je zult zien als er met de muis wordt geklikt.
186. Bekijk het resultaat in de *browser*. Klopt je voorspelling? Begrijp je wat er gebeurt?

Ten opzichte van de vorige opgave is het vullen van de lijst met knikkers veranderd. In de `setup` staat nu: `knikkerVerzameling.push(new Knikker(random(20,980),random(20,280),'white'));`

187. Welke betekenis hebben de drie parameters die aan `Knikker` worden meegegeven?

Als op de muis wordt gedrukt, willen we bereiken dat er een nieuwe knikker wordt toegevoegd aan de lijst. Deze knikker moet verschijnen op de plek waar de muis zich bevindt als er wordt geklikt.

188. Pas de code aan, zodat dit doel wordt bereikt. Maak de nieuwe knikkers rood, zodat je goed kunt zien welke de nieuwe zijn.

In figuur 2.61 is zojuist één keer met de muis geklikt. Toch zie je meerdere rode knikkers.



FIGUUR 2.61

189. Geef hiervoor de verklaring.



Opdracht 39 knikkerbak III: beweeg naar de muis

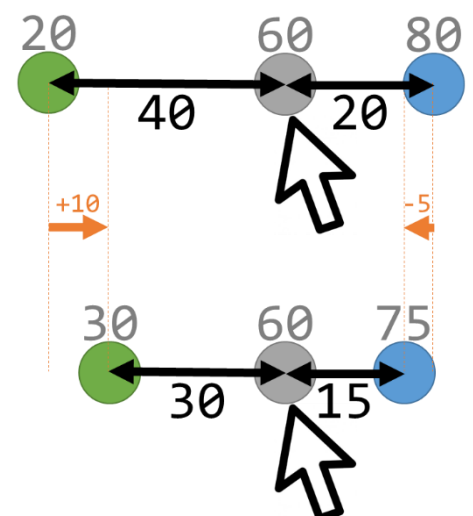
In figuur 2.62 zie je een veel gebruikt principe bij het maken van games: **lineaire interpolatie**. Het wordt onder andere gebruikt om objecten naar elkaar toe te laten bewegen.

In de tekening bevindt de groene knikker zich in eerst op $x = 20$ en de muis op $x = 60$. De onderlinge afstand is dus $60 - 20 = 40$. We willen programmeren dat de groene knikker een stap zet die 25% (kwart) van de onderlinge afstand is. Die stap is dus $40 \cdot 0,25 = 10$.

In een wiskundige formule met variabelen is de grootte van de stap:

$$stap = (x_{\text{muis}} - x_{\text{knikker}}) \cdot 0,25$$

190. De groene knikker beweegt naar rechts, maar de blauwe knikker beweegt naar links. Klopt de formule ook voor blauw?
191. Open `H2O39.js` in jouw *editor* en bestudeer de code. De klasse `Knikker` bevat een methode `gaNaarMuis` die nu nog leeg is.
192. Vul de methode `gaNaarMuis` zodanig aan dat de knikker zich bij elke aanroep zowel in de x-richting als de y-richting 25% van de onderlinge afstand richting de muis beweegt.
193. Verander het percentage naar 5% en bekijk het eindresultaat.

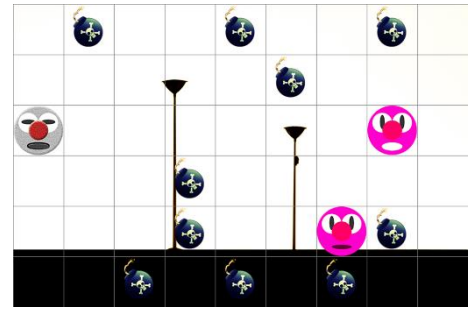


FIGUUR 2.62

Opdracht 40 overloper XI: een array met bommen

Om het spel *overloper* uitdagender te maken gaan we het raster vullen met een aantal bommen. Als je Eve op een bom laat stappen, ben je af. Het eindresultaat ziet eruit zoals in figuur 2.63, maar we beginnen eerst met één bom.

Voor Eve is een nieuw attribuut `staOpBom` gemaakt. Het is de bedoeling dat deze boolean straks de waarde is `true` krijgt, als Eve zich op de plek van een bom bevindt.



FIGUUR 2.63

194. Open *H2O40.js* in jouw *editor* en bestudeer de code. Hierin is een nieuwe klasse `Bom` opgenomen, waarmee één instantie is gemaakt.

De **constructor** van de klasse `Bom` bevat de volgende regels:

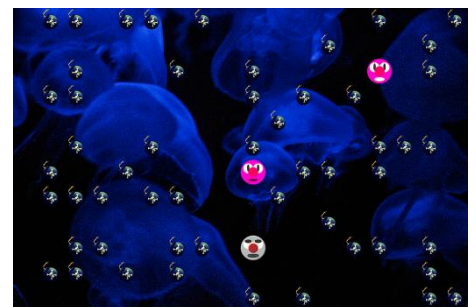
```
this.x = floor(random(1,raster.aantalKolommen))*raster.celGrootte;  
this.y = floor(random(0,raster.aantalRijen))*raster.celGrootte;
```

195. Welke mogelijke waarden hebben `this.x` en `this.y`?
196. Bedenk een mogelijke reden waarom de programmeur ervoor gekozen heeft om voor `this.x` als argument van de functie `random` een `1` in te vullen in plaats van `0`.
197. In regel 73 is een lege array `bommenArray` gemaakt. Vul deze lijst met tien instanties van `Bom`.
198. Gebruik een for-loop in de `draw` om alle bommen te tonen. Als je nu nog coderegels met `bom1` hebt, moet je die verwijderen.
199. Controleer het resultaat. Hoeveel bommen zie je?
200. Geef **twee** redenen waarom je, als je het spel laadt, niet altijd tien bommen ziet.

Alice en Bob kunnen gewoon over bommen heenlopen, maar als Eve op een bom staat, moet het spel afgelopen zijn. Om dit te bereiken is voor Eve (in de klasse `Jos`) de methode `staatOp` gemaakt, met als parameter de array met bommen. Deze wordt al aangeroepen (met `Eve.staatOp(bommenArray)`), maar bevat nog niet de juiste code. Om dit te bereiken moeten de volgende stappen worden gezet:

- Maak een for-loop die één voor één de bommen uit de array langsloopt
- Controleer voor elke bom of de positie van Eve overeenkomt met die van de bom
- Is dat zo? Verander dan `staOpBom` in `true`.
- Klaar met de lijst? Geef dan de waarde van `staOpBom` terug.

201. Pas de methode `staatOp` aan op basis van bovenstaande stappen en controleer het eindresultaat.
202. Leg uit dat het in theorie mogelijk is dat er een spel ontstaat dat niet te winnen is.
203. Voer de volgende stappen uit om tot een eindversie van *overloper* te komen, vergelijkbaar met figuur 2.64:
- Pas het raster aan zodat het 18 × 12 cellen bevat
 - Kies voor dit grotere speelveld (216 cellen!) zelf een geschikt aantal bommen.
 - Kies zelf een mooie afbeelding als achtergrond
 - Zet het tekenen van het raster uit



FIGUUR 2.64

★ Opdracht 41 obfuscator XIII: bouncer

Een laatste uitdaging in dit hoofdstuk: bij het spel *bouncer* kun je door met je muis te klikken nieuwe ballen laten stuiteren, totdat één van de ballen de grond heeft geraakt. Maar pas op: vervolgens moet jij alle gestuiterde ballen weggklikken, voordat ze voor een tweede keer de grond raakt. Hoeveel risico neem jij?

204. Bekijk *OBF13*. Probeer het spel een paar keer uit.
205. Open *H2O41.js* in jouw *editor* en bestudeer de code. Hierin is al geprogrammeerd dat je nieuwe ballen kunt maken, totdat één van de ballen gestuiterd is.
206. Vul *H2O41.js* aan zodat deze zich net zo gedraagt als *OBF13*.