Functies, objecten, arrays en exceptions

Leereenheid 5

Functies, objecten, arrays en exceptions

INTRODUCTIE

Functies zijn in feite een manier om te voorkomen dat er op allerlei plekken in de code dezelfde stukjes code voorkomen. JavaScript kent uiteraard functies, en in deze leereenheid krijgt u te zien hoe u functies definieert en gebruikt in JavaScript. Daarbij wordt aandacht besteed aan het verschil tussen 'pure' functies en functies met side-effect.

In de vorige leereenheid heeft u gezien dat variabelen en expressies als waarde een Number, een String of een Boolean kunnen krijgen; in deze leereenheid leert u twee andere typen kennen: objecten en arrays. Dit zijn zogenaamde referentiële typen: een variabele krijgt niet het object of de array zelf als waarde, maar een verwijzing naar een object of een array.

U leert ook een aantal standaard-objecten kennen in JavaScript, en u maakt kennis met een aantal nieuwe bewerkingen op strings.

Ook besteden we aandacht aan exception handling: een manier om te voorkomen dat een programma vastloopt wanneer zich een onverwachte situatie voordoet.

De leereenheid sluit af met een aanpak om functies te ontwerpen en te testen. Deze aanpak wordt in de rest van de cursus steeds gebruikt.

LEERDOELEN

Na het bestuderen van deze leereenheid wordt verwacht dat u

- weet wat een pure functie is en pure functies kunt schrijven
- weet wat de scope is van variabelen en van formele parameters
- weet wat argumenten en formele parameters zijn en hoe parameteroverdracht plaatsvindt
- de werking van een JavaScript-programma kunt nagaan met behulp van geheugenmodellen
- weet wat de execution stack is en hiermee de loop van een programma kunt voorspellen bij functieaanroepen en exceptions
- weet wat een closure is
- weet dat een functie als variabele, als parameter en als terugkeerwaarde kan optreden
- met arrays en objecten kunt werken
- weet wat refactoren is en refactoren kunt uitvoeren
- weet welke soort fouten in een programma kunnen optreden en hoe die vermeden kunnen worden
- exception handling kunt toepassen
- weet wat de signatuur van een functie is
- een systematische aanpak kent voor het ontwerpen, implementeren en testen van functies en deze aanpak kunt toepassen.

Studeeraanwijzing

Weblink: JavaScript referentie

Het is handig om bij het bestuderen van deze leereenheid de behandelde typen en functies op te zoeken in de JavaScript-referentie, zie weblink.

LEERKERN

1 Chapter 3, paragrafen 1 en 2

Pure functies

Een pure functie is een functie zonder neveneffecten, dat wil zeggen: de terugkeerwaarde is alleen afhankelijk van de parameters en er vinden geen veranderingen van de omgeving van de functie plaats. Hierdoor is het resultaat van zo'n functie ook altijd reproduceerbaar. Zo'n functie kunnen we voorstellen als een black box met een invoer en uitvoer. Een pure functie komt ook overeen met het wiskundige begrip functie: de uitvoer is te schrijven als een functie van de invoer, bijvoorbeeld

```
y = f(a,b)
```

Scope

In leereenheid 4 zagen we dat de scope van een variabele begint bij de eerste regel van het programma waarin deze gebruikt wordt. Deze uitspraak moeten we nu wat nuanceren. Als een variabele gedeclareerd wordt in een functie, dan is deze zichtbaar (*lokale* zichtbaarheid) vanaf de eerste regel van deze functie tot en met de laatste regel, maar niet daarbuiten. Variabelen die binnen een functie gedeclareerd worden zijn *tijdelijke variabelen*.

Tijdelijke variabelen

Keyword var

Het is belangrijk om te weten dat een variabele die binnen een functie wordt gedeclareerd alleen maar lokaal is wanneer het *keyword var* wordt gebruikt. Wanneer u een variabele zonder dat keyword declareert, is de variabele globaal!

Programmeeraanwijzing: Declareer variabelen altijd met var
Declareer een variabele altijd met het keyword var. Dan kunt u nooit per ongeluk binnen een functie een globale variabele declareren of overschrijven.

In een functie zijn ook de variabelen zichtbaar die in de omgeving (environment) buiten de functie gedeclareerd zijn (*globale* zichtbaarheid).

Side-effect

Wanneer in een functie de waarde van een globale variabele wordt aangepast, heet dat een *side-effect.* Een eigenschap van een pure functie is dat deze geen side-effects heeft. Een pure functie mag dus geen waarden veranderen van globale variabelen, of een globale variabele introduceren.

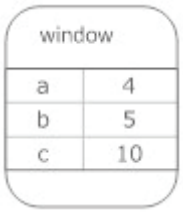
Als de naam van een gedeclareerde variabele in een functie dezelfde is als die van een erbuiten gedeclareerde variabele, 'overschaduwt' de lokale variabele de globale variabele. We illustreren bovenstaande met behulp van het geheugenmodel waarbij we voor de functie een soortgelijke voorstelling gebruiken als wanneer we de parameters als lokale variabelen beschouwen en de return als een pseudo-variabele die de terugkeerwaarde bevat. De globale omgeving waarbinnen we werken, wordt in JavaScript `window` genoemd.

Argumenten en formele parameters

Bij de uitvoering van een functie worden de waarden van de *argumenten* van de aanroep gekopieerd naar de *formele parameters* van de functie. Ter illustratie gebruiken we het volgende programma waarin de functie `xFactor` wordt aangeroepen met als argumenten `1` en `2`:

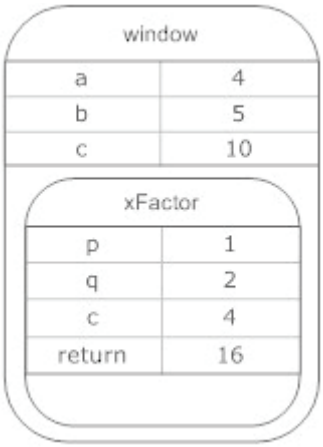
```
var a = 4,
    b = 5,
    c = 10;
function xFactor(p, q) {
  var c = 4;
  return a + b + c + p + q;
}
print(xFactor(1, 2));
```

De situatie net voor de functieaanroep kunnen we weergeven zoals in figuur 5.1.



FIGUUR 5.1 Geheugenmodel net voor de functieaanroep

Figuur 5.2 toont de situatie aan het eind van de verwerking van de functie. Hierbij zijn de waarden van de argumenten `1` en `2` gekopieerd naar de overeenkomstige formele parameters `p` en `q` en bevat de pseudo-variabele `result` het resultaat van de berekening. Binnen de `window` omgeving is een tijdelijke omgeving `xFactor` ontstaan.



FIGUUR 5.2 Geheugenmodel aan het eind van de functieaanroep

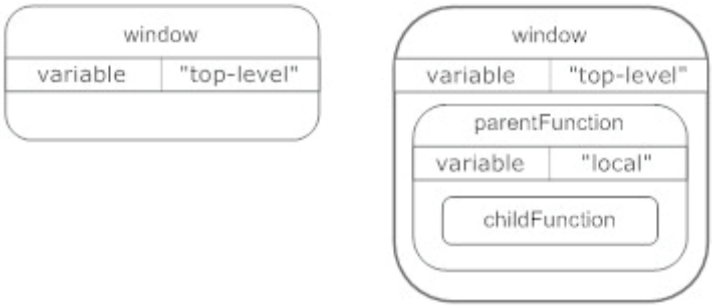
De eindsituatie is weer de situatie van figuur 5.1; de omgeving van `xFactor` is verdwenen. We zien in figuur 5.2 een variabele `c` in de `window`-omgeving en een variabele `c` binnen `xFactor`. Bij het bepalen van het resultaat van de functie wordt de waarde `4` voor de lokale variabele `c` gebruikt: de lokale `c` 'overschaduwt' de globale `c`.

Het begin van paragraaf 2 toont dat twee verschillende functies elkaars variabelen niet kunnen zien, wat ook logisch is op basis van het voorgaande. Als een functie binnen een andere functie gedefineerd wordt (een interne functie), kan dat wel zoals paragraaf 2.1 toont.

We herhalen hier de code van paragraaf 2.1 en tonen het geheugenmodel dat daar bij hoort.

```
1 var variable = "top-level";
2 function parentFunction() {
3   var variable = "local";
4   function childFunction() {
5     print(variable);
6   }
7   childFunction();
8 }
9 parentFunction();
```

Figuur 5.3 toont het geheugenmodel na regel 1 en tijdens de verwerking van regel 9 op het moment dat `parentFunction` is aangeroepen die op zijn beurt `childFunction` heeft aangeroepen. Als in `childFunction` de waarde van `variable` wordt geprint, dan wordt de waarde uit die omgeving gebruikt, dus wordt 'local' afgedrukt. De omgevingen omvatten elkaar.



FIGUUR 5.3 Geheugenmodel bij code van paragraaf 2.1

De inhoud van paragraaf 5.2 zal ons niet verbazen. JavaScript kent geen block-scope zoals Java. Daarom zal het toevoegen van { en } om een block aan te geven geen gevolg hebben voor de scope van variabelen.

Lexical scoping

Deze manier waarop de scope van variabelen werkt wordt *lexical scoping* genoemd. Het volgende voorbeeld illustreert het gevaar van het ontbreken van block-scope:

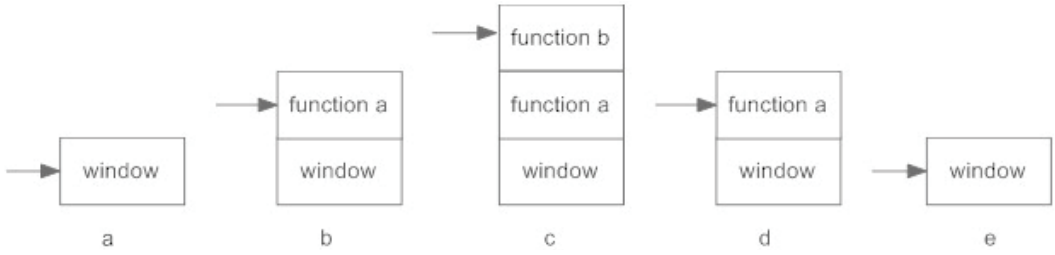
```
var x = 1,
    test = true;
if (test) {
  var x = 10;
}
print(x); // drukt 10 af!

terwijl

var x = 1,
    test = true;
function y(t){
  if (t) {
    var x = 10;
  }
}
y(test);
print(x);  // drukt 1 af!
```

Dus alleen binnen een functie is er sprake van een lokale scope.

De JavaScript-runtime-omgeving houdt ook steeds bij welke programmaregel moet worden uitgevoerd. Als tijdens de uitvoering van een programma een functie wordt aangeroepen, wordt de omgeving van het programma met de regel waar het programma gebleven is vastgelegd, zodat het programma na afloop van de functie weer vervolgd kan worden. Als de functie op haar beurt ook weer een functie aanroept, gebeurt hetzelfde.

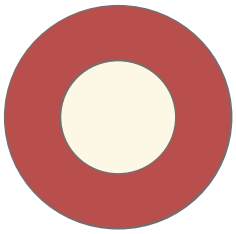
Execution stack

De runtime-omgeving van JavaScript gebruikt hiervoor een stukje geheugen dat de *execution stack*, kortweg *stack*, genoemd wordt. De stack is een wachtrij volgens het principe Last In First Out (LIFO). In figuur 5.4 hebben we de situaties weergegeven waarbij een functie `a` aangeroepen wordt (b) die op zijn beurt een functie `b` aanroept (c). De stack neemt weer af als functie `b` klaar is (c) en als daarna functie `a` afgelopen is (e).



FIGUUR 5.4 Groei en afname van de stack

OPGAVE 5.1
Schrijf een pure functie die de oppervlakte berekent van een cirkelschijf, het donkere deel van figuur 5.5.



FIGUUR 5.5 Cirkelschijf

De oppervlakte van een cirkel wordt gegeven door de formule
O = π*R*R waarbij R de straal van de cirkel voorstelt en π = 3.14.

2 Paragraaf 3: Closure

Deze paragraaf toont een eigenschap van functies in JavaScript die geen equivalent kent in Java: een functie kan een andere functie als terugkeerwaarde geven. Ook kunnen we aan een variabele niet alleen de waarde van een functie toekennen maar ook een referentie naar een functie.

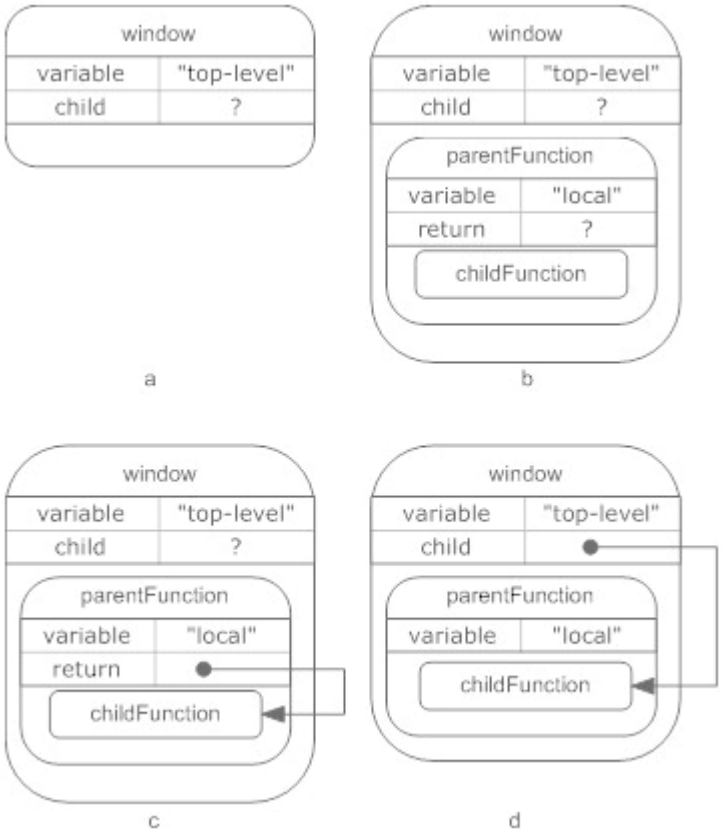
```
1  var variable = "top-level";
2  function parentFunction() {
3    var variable = "local";
4    function childFunction() {
5      print(variable);
6    }
7    return childFunction;
8  }
9  var child = parentFunction();
10 child(); // drukt local af
```

De functie `parentFunction` heeft als terugkeerwaarde een referentie naar `childFunction`. In regel 9 krijgt variabele `child` deze referentie. In regel 10 zien we dat de variabele `child` nu gebruikt kan worden om de functie aan te roepen door achter de naam van de functie `()` te zetten.

Het bijzondere is nu dat `childFunction` gebruikmaakt van de omgeving waarbinnen deze functie is gedeclareerd, dus hier `parentFunction`. Daarom zal de waarde 'local' afgedrukt worden.

In JavaScript heeft een functie altijd toegang tot de context waarin de functie gecreëerd is. Dat is gemakkelijk te begrijpen: op het moment dat de functie `childFunction` wordt teruggegeven, is de code van `parentFunction` uitgevoerd. Daarbij is voor de variabele `variable` de waarde 'local' ingevuld.

Figuur 5.6 toont (a) het geheugenmodel na regel 1 (waarbij hoisting van variabele `child` heeft plaatsgevonden), (b) aan het begin van de uitvoering van `parentFunction`, (c) aan het eind van de uitvoering van `parentFunction` en (d) na afloop van regel 9.



FIGUUR 5.6 Geheugenmodel bij code van paragraaf 3

Closure

Een *closure* is een functie met een bijbehorende omgeving van lokale variabelen. De variabelen en hun waarden blijven bestaan na afloop van de aanroep van de functie, doordat een functie die aan deze variabelen refereert ter beschikking blijft (doordat de functie als terugkeerwaarde is gegeven).

We bekijken nog een voorbeeld:

```
function getFunctie(x) {
  return function hoogOp() {
    x += 1;
    return x;
  };
}

var f = getFunctie(3);
print(f());  // 4
print(f());  //5
```

We zien dat zelfs de waarde van de parameter `x` die weer als lokale variabele gebruikt wordt, bewaard blijft. Het is niet mogelijk direct `x` te benaderen. Met een closure kunnen we dus information hiding toepassen en op deze wijze private variabelen realiseren (u zult dat verderop in deze cursus uitgebreider tegenkomen).

3 Paragrafen 4 tot en met 6

Paragraaf 4 beschrijft dat ook bij functies hoisting optreedt. Hierdoor is het toegestaan een functie aan te roepen voordat deze gedefinieerd is.

Functiedeclaratie
Functie-expressie

De manier waarop we in paragraaf 1 functies definieerden, wordt *functiedeclaratie* genoemd. Paragraaf 5 beschrijft een alternatieve manier die een *functie-expressie* genoemd wordt. De waarde van de functie expressie wordt toegekend aan een variabele; de functie zelf is anoniem. De variabele bevat een referentie naar de functie. Door achter de naam van de variabele () te plaatsen, wordt de functie uitgevoerd.

Bij functie-expressies treedt *geen* hoisting op!

Het volgende fragment toont de twee manieren onder elkaar.

```
function hoogOp(getal) {
  return getal + 1;
}
print(hoogOp(3));

var hoogOp2 = function(getal) {
  return getal + 1;
};
print(hoogOp2(3));
print(hoogOp2);
print(typeof hoogOp2);
```

We zien dat de functies op dezelfde manier worden aangeroepen.

Het resultaat van bovenstaande code is:

```
4
4
function(getal) {
  return getal + 1;
}
function
```

Het type van variabele `hoogOp` is `function`, waarmee een Function-object wordt bedoeld.

Functie als terugkeerwaarde

Functies kunnen als resultaat een functie opleveren.

OPGAVE 5.2
Schrijf een functie `isWortelVan` met een numerieke parameter `g` die een functie teruggeeft met parameter `n` waarmee getest kan worden of `n` een wortel is van `g`. Test het resultaat met het gegeven dat `4` een wortel van `16` is.

We zien dat het in JavaScript soms handig is dat een functie een andere functie als terugkeerwaarde heeft. Later zullen we zien dat functies ook als parameter van een andere functie gebruikt kunnen worden.

Direct aangeroepen functie

Soms is het handig om een functie te definiëren en daarna direct aan te roepen. Dit kan niet met een functiedeclaratie maar wel met een functie-expressie, mits we deze maar tussen haakjes zetten. Zo'n functie kan dan anoniem zijn:

```
(function() {
  //code
})();
```

Omdat een functie een lokale scope kent, zijn variabelen die binnen de functie zijn gedeclareerd ook alleen zichtbaar binnen die functie en kunnen de namen van deze variabelen zonder probleem op andere plaatsen gebruikt worden. Direct aangeroepen functies vormen de basis van het module-patroon dat later in deze cursus besproken zal worden.

Samenvattend kunnen we zeggen dat een functie-expressie een function-object oplevert. Functie-objecten zijn in JavaScript 'first class'. Dat wil zeggen dat ze aldus gebruikt kunnen worden:
– als parameter in een (andere) functie
– als terugkeerwaarde van een functie
– om toe te kennen aan een variabele
– om opgeslagen te worden in een object of array.
Verderop in deze cursus komen we hierop terug.

Aantal parameters in een functie-aanroep

Paragraaf 7 toont het flexibele karakter van JavaScript. Als in een functieaanroep te weinig parameters worden meegegeven, krijgen de overeenkomstige formele parameters de waarde `undefined`. Er treedt echter geen exception op. Als er te veel parameters worden meegegeven, worden de extra parameters gewoon genegeerd.

4 Chapter 4, paragraaf 1

Paragraaf 1 is enigszins misleidend. In de vorige leereenheid hebben we gezien dat JavaScript onder andere de primitieve typen number, boolean en string kent. Een property is te vergelijken met een attribuut in Java en bestaat uit een naam en een waarde. Hoe kan een waarde of variabele van een primitief type dan properties bezitten?

Een primitief type kan *geen* properties bezitten. De JavaScript-interpreter converteert een primitief type automatisch naar een overeenkomstig objecttype als de aangeroepen property daar bij hoort.

JavaScript kent naast het primitieve type string ook `String` objecten. In de JavaScript reference kunt u zien dat het objecttype String vijf properties kent: `arity`, `caller`, `constructor`, `length` en `name`, waarvan alleen `length` nu betekenis voor ons heeft.

Ook booleans en numbers kennen bijbehorende objecttypen. Het volgende programmafragment drukt 11 af.

```
var text = "purple haze";
print(text.length);
```

Bij de uitvoering van dit stukje code creëert de JavaScript-interpreter een nieuw `String` object op basis van de waarde van de variabele `text`. Dit nieuwe object kent een property `length` waarvan de waarde wordt opgevraagd en door de functie `print` wordt afgedrukt. Het nieuwe object wordt daarna weer vernietigd.

Properties kunnen op twee manieren benaderd worden: met de puntnotatie die ook in Java gebruikt wordt of door de naam als string tussen [en] te zetten. De tweede manier is handig wanneer u een variabele wilt gebruiken om aan de waarde van een property te komen; in alle andere gevallen gebruikt u de puntnotatie.

Programmeeraanwijzing: **Gebruik de puntnotatie**
Gebruik de puntnotatie om aan de waarde van een property van een object te komen. Alleen wanneer u een variabele wilt gebruiken als naam voor een property, gebruikt u de notatie met [en].

5 Paragraaf 2

Ten opzichte van een klassengebaseerde taal als Java is het opvallend dat we in JavaScript objecten kunnen maken zonder dat daar een klasse voor nodig is. We kunnen deze objecten 'on the fly' maken. Ze bevatten properties die op hun beurt bestaan uit een combinatie van een naam en een bijbehorende waarde, het geheel tussen twee accolades geplaatst.

Literal objects

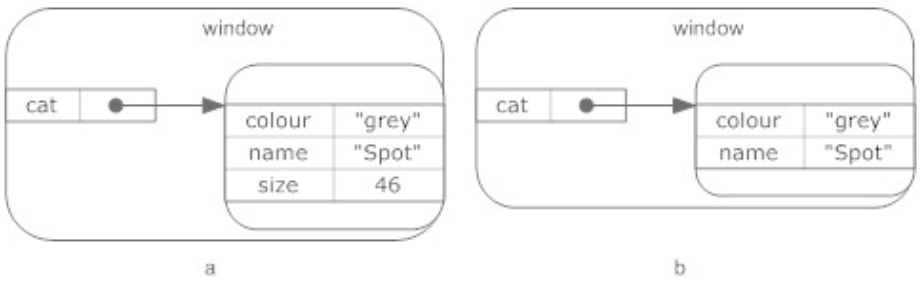
Dergelijke objecten worden wel *literal objects* genoemd.

JavaScript-object

In *JavaScript* kunnen we een *object* beschouwen als een dynamische verzameling properties. We gebruiken zo'n object om samenhangende gegevens te bewaren.

We herhalen hier de code van het eerste programma van paragraaf 2 en tonen u het bijbehorende geheugenmodel.

```
var cat = {
      colour : "grey",
      name : "Spot",
      size : 46
   };
2 cat.size = 47;
3 show(cat.size);
4 delete cat.size;
5 show(cat.size);
6 show(cat);
```



FIGUUR 5.7 Geheugenmodel bij de code van paragraaf 2.

Figuur 5.7 a toont de situatie na regel 1, figuur 5.7 b die na regel 4.

Let op!

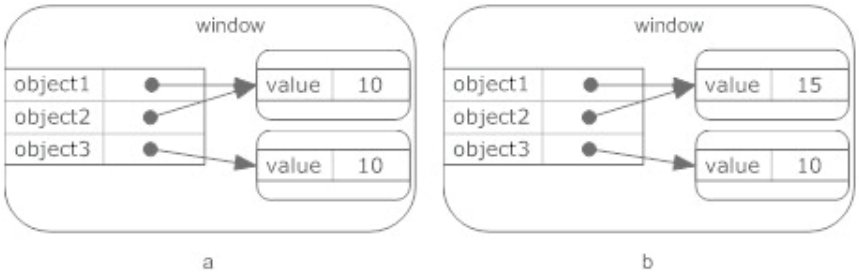
Aan het eind van de inleiding van paragraaf 2 staat dat *waarden immutable* zijn; *variabelen* zijn dat niet.

Alias

Als twee variabelen naar hetzelfde object verwijzen, spreken we van een *alias*. We lichten dit begrip toe aan de hand van het geheugenmodel op basis van de code uit paragraaf 2.1

```
1 var object1 = {
    value: 10
  };
2 var object2 = object1;
  var object3 = {
    value : 10
  };
4 show(object1 === object2);
5 show(object1 === object3);
6 object1.value = 15;
7 show(object2.value);
8 show(object3.value);
```

Figuur 5.8 a toont de situatie na regel 3; figuur 5.8 b na regel 5.



FIGUUR 5.8 Geheugenmodel bij de code van paragraaf 2.1

De variabelen `object1` en `object2` zijn aliassen. Met `===` testen we of de referenties naar twee objecten gelijk zijn; dat wil zeggen, we testen of ze naar hetzelfde object verwijzen. Merk op dat `object3` verwijst naar een ander object met dezelfde 'inhoud' als `object1` en `object2`. Daarom geeft regel 4 de waarde `true` en regel 5 `false`.

Het begrip alias en de manier waarop we referenties van objecten vergelijken, is volkomen analoog aan de manier waarop dat in Java gebeurt. In volgende leereenheden komen we nog uitgebreid terug op objecten.

OPGAVE 5.3
Gegeven de volgende programma code:

```
1 var obj1 = {
    getal : 6
  };
2 var obj2 = {
    getal : 3
  };
3 var obj3 = {
    ref : obj2
  };
4 var obj4 = obj2;
5 obj4.getal = obj1.getal;
6 print(obj3.ref.getal);
7 print(obj3.ref === obj4);
```

Teken een geheugenmodel na uitvoering van de code van regel 4 en na regel 5. Voorspel op basis van het laatste model wat bij regel 6 en 7 afgedrukt zal worden.

6 Paragraaf 3: Arrays

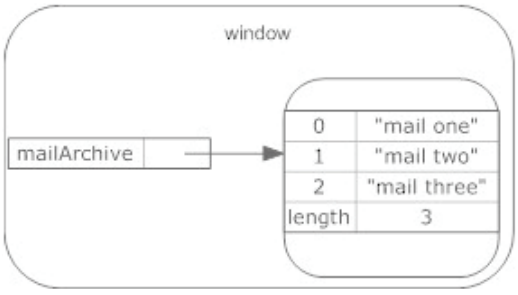
Array literal

Paragraaf 3 toont hoe u een *array literal* in JavaScript kunt maken en hoe u de elementen van zo'n array kunt doorlopen met een `for`-lus. Hierbij maken we gebruik van property `length` die het aantal elementen van een array aangeeft.

Als we van een array met `typeof` het type opvragen, blijkt dat het type `object` is. Arrays in JavaScript zijn dus objecten. Na uitvoering van de regel

```
var mailArchive = ["mail one", "mail two", "mail three"];
```

krijgen we het volgende geheugenmodel, zie figuur 5.9.



FIGUUR 5.9 Geheugenmodel bij een array

We hebben in het model ook property `length` weergegeven.

Constructorfunctie

Een andere manier om een array te maken is met behulp van een constructorfunctie `Array`. Constructorfuncties laten we altijd beginnen met een hoofdletter. De leereenheid over modulen en prototype gaat gedetailleerd in op constructor-functies.

We roepen de constructorfunctie aan met `new` en geven het aantal elementen als parameter mee. Om een array van 20 elementen te maken gebruikt u de volgende code:

```
var ar = new Array(20);
```

De manier waarop we hier een array maken lijkt op het aanroepen van een constructor in Java.

Tweedimensionale array

Als elementen van een array kunnen we andere arrays opnemen. Op die manier kunnen we meerdimensionale arrays realiseren. Als voorbeeld tonen we u een situatie van 'Boter-kaas-en-eieren' waarin we symbolen X, O en _ gebruiken voor respectievelijk een kruis, een nul en een blanco veld:

```
var bord = [
            ["X", "O ", "_"],
            ["X ", "O ", "_"],
            ["O", "X ", "_"]
           ];
```

7 Paragraaf 4: Properties and methods of strings and arrays

Methode

Deze paragraaf bespreekt een bijzondere property van objecten: een property met als waarde een functiedefinitie. Zo'n property wordt een *methode* genoemd. Ieder object erft een aantal methoden, die per type kunnen verschillen.

Keyword this

Bij een methode kunnen we gebruikmaken van de overige properties van het object via het *keyword this*.

push, pop, join en split

Een string-object kent vele methoden, waaronder `toUppercase` en `slice`. Arrays kennen onder andere een methode *push* waarmee elementen aan een array kunnen worden toegevoegd en een methode *pop* om het laatste element te verwijderen. De methode `join` maakt een string-representatie van een array, en `split`, een methode van string, levert een array die opgebouwd is uit de elementen van de string.

We herhalen hier de code na exercise 4.5 die we aangevuld hebben, zodat de code ook kan draaien. Ook hebben we de namen van variabelen wat aangepast en netjes overal { en } gebruikt. Maar eerst geven we in gestructureerd Nederlands weer *wat* het programma doet.

Pseudocode

Zo'n weergave noemen we een weergave in *pseudocode*.

```
haal de emails op
creëer de verzameling livingCats die bestaat uit {Spot,
true}
voor iedere email doe:
  lees de email en splits deze op in aparte paragrafen
  voor iedere paragraaf doe:
    als de paragraaf begint met "born"
      lees de namen van de katten uit de paragraaf
      voor iedere naam doe:
        voeg de naam toe aan livingCats
    anders
      als de paragraaf begint met "died"
        lees de namen van de katten uit de paragraaf
        voor iedere naam doe:
          verwijder de naam uit livingCats
```

De JavaScript-code luidt:

```
1  function retrieveMails() {
2  return ["born 05/04/2006 (mother Lady Penelope):
3    Red Lion, Doctor Hobbles the 3rd, Littele Iroquois"];
4  }

5  function catNames(paragraph) {
6    var colon = paragraph.indexOf(":");
7    return paragraph.slice(colon + 2).split(", ");
8  }

9  function startsWith(string, pattern) {
10   return string.slice(0, pattern.length) === pattern;
11 }
12 var mailArchive = retrieveMails(),
       livingCats = {"Spot": true},
       paragraphs,
       names;

13 for (var mailNr = 0; mailNr < mailArchive.length; mailNr++) {
14   paragraphs = mailArchive[mailNr].split("\n");

15 for (var parNr = 0; parNr < paragraphs.length; parNr++) {
16   if (startsWith(paragraphs[parNr], "born")) {
17       names = catNames(paragraphs[parNr]);
18     for (var nameNr = 0; nameNr < names.length; nameNr++) {
19       livingCats[names[nameNr]] = true;
20       }
21     } else {
22       if (startsWith(paragraphs[parNr], "died")) {
23         names = catNames(paragraphs[parNr]);
24         for (var nameNr = 0; nameNr < names.length; nameNr++) {
25           delete livingCats[names[nameNr]];
26         }
27       }
28     }
29   }
30 }
31 show(livingCats);
```

Regels 1 t/m 11 bevatten functies die nodig zijn voor de rest van het programma. In regel 7 roepen we met behulp van de dot-notatie meerdere functies achter elkaar aan. Eerst wordt dan de meest linkse functie uitgevoerd, op het resultaat de volgende enzovoort.

Regel 12 initialiseert `mailArchive`, een array die de e-mails bevat, en `livingCats` met een object met property `Spot` en waarde `true`.

Regels 13 t/m 30 bevatten een lus die over alle e-mails van `mailArchive` loopt. `mailArchive[mailNr]` bevat een specifieke e-mail in de vorm van een string. Deze string wordt op basis van een nieuwe regel (“\n”) gesplitst en als array aan variabele `paragraphs` toegewezen (regel 14).

Regels 15 t/m 29 bevatten een lus over de paragrafen van de e-mail. Er wordt gekeken of deze begint met “born” (regel 16) of “died” (regel 22). Afhankelijk van die voorwaarde worden de namen in de betreffende paragraaf toegevoegd aan livingCats of eruit verwijderd.

Als we naar de code kijken, zien we dat regels 17 t/m 20 en regels 23 t/m 27 veel op elkaar lijken. De code kan dan ook nog beter, wat in de volgende paragraaf ook gaat gebeuren.

Refactoring

Het verbeteren van de code zonder dat de functionaliteit verandert wordt *refactoring* genoemd. Voor een programmeur is vaardigheid in het refactoren van groot belang. Uiteindelijk zal de code door refactoren beter leesbaar en gemakkelijker onderhoudbaar worden. Refactoren moet eigenlijk een tweede natuur worden van een goede programmeur. Door voortdurend te refactoren, zal dit proces steeds in kleine stapjes verlopen en zo goed beheersbaar zijn.

Martin Fowler, een goeroe op het gebied van refactoring, schrijft in zijn boek 'Refactoring':

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Bad smells

Fowler heeft het in zijn boek over '*bad smells*', code dus die stinkt. De code van het vorige programma doet dat ook, omdat in dit geval een deel van de code bijna letterlijk herhaald wordt. De remedie is van deze repeterende code een functie te maken en die dan twee keer aan te roepen.

operator in

De paragraaf eindigt met een voorbeeld van de operator *in* waarmee we in een `for`-lus de index van een array verkrijgen. Deze index kunnen we dan gebruiken om de elementen van de array te benaderen:

```
var vruchten = ["appel", "peer", "banaan"],
    vrucht;
for (vrucht in vruchten){
  print("index: " + vrucht + " waarde: " +
         vruchten[vrucht]);
}
```

Met als uitvoer:

```
index: 0 waarde: appel
index: 1 waarde: peer
index: 2 waarde: banaan
```

Let op!

Het gebruik van `for-in` wordt echter *afgeraden* bij een array.

Standaard-manier om array te doorlopen

Ter vergelijking tonen we u de *standaard-manier van het doorlopen van een array*:

```
var vruchten = ["appel", "peer", "banaan"],
    i,
    lengte = vruchten.length;
for (i = 0; i < lengte; i += 1) {
  print("index: " + i + " waarde: " + vruchten[i]);
}
```

Doorlopen array met forEach

Ten slotte tonen we u nog het gebruik van functie *forEach* van een array voor het doorlopen van de elementen van een array. Deze functie heeft als parameter een functie met een parameter die de waarde van een array-element representeert:

```
var vruchten = ["appel", "peer", "banaan"];
vruchten.forEach(function(vrucht) {
  print("waarde: " + vrucht);
});
```

Bij de eerste twee manieren hebben we een expliciete index; bij de laatste niet. Bij de tweede manier moeten we zelf de begin- en eindwaarden van de index aangeven. Bij de eerste en derde manier gebeurt dat impliciet door het gebruik van `in` of `forEach`.

We kunnen `in` ook gebruiken om de properties van een object op te vragen:

```
var obj = {
  naam : "Web applicaties",
  afkorting : "WAC",
  code : "T58311"
}
for (var prop in obj){
  print(prop + ": " + obj[prop]);
}
```

Met als uitvoer:

```
naam: Web applicaties
afkorting: WAC
code: T58311
```

Object.keys

Voor literal objecten werkt operator `in` goed, maar bij objecten die we met een functie maken (zie verderop in deze cursus) niet: we krijgen dan ook de properties van objecten waarvan het eigen object erft. Een betere manier is dan om door middel van *Object.keys* een array van alleen eigen properties te verkrijgen en die te doorlopen. Toegepast op het vorige voorbeeld krijgen we dan:

```
var obj = {
      naam : "Web applicaties",
      afkorting : "WAC",
      code : "T58311"
    },
    keys = Object.keys(obj);
keys.forEach(function(prop){
  print(prop + ": " + obj[prop]);
});
```

met dezelfde uitvoer als eerder. U ziet hier een voorbeeld van een situatie waarin u niet met de puntnotatie voor properties van objecten kunt werken maar de [en] nodig heeft.

Performance

Hoewel leesbaarheid van een programma erg belangrijk is, moeten we niet vergeten dat we te maken hebben met een interpreter die de JavaScript-opdrachten uitvoert. Met name bij het doorlopen van een lus waarbij het aantal herhalingen groot is, kan *performance* een rol spelen.

De standaard-manier om een `for`-lus te schrijven is niet optimaal: bij elke herhaling wordt de lengte van een array opnieuw bepaald. We kunnen ervoor zorgen dat dit eenmalig gebeurt door deze berekening voor de lus te zetten. Daarnaast is het ook beter om de lusvariabele al aan het begin van het programma te zetten. Zo kan bijvoorbeeld regel 14 uit het vorige programma

```
14 for (var mailNr = 0; mailNr < mailArchive.length;
mailNr++) {
```

omgezet worden naar de volgende vorm:

```
var mailNr,
    mailLength = mailArchive.length;
for (mailNr = 0; mailNr < mailLength; mailNr++) {
```

We zullen dit soort wijzigingen verder niet consequent doorvoeren, te meer daar het tekstboek het ook niet doet. Maar als uw code te traag is, weet u nu hoe u dat enigszins kunt verhelpen.

OPGAVE 5.4

Schrijf een stukje code dat de namen en waarden afdrukt van de properties van het object dat door de volgende code gegeven wordt:

```
var obj = {
  Aap :  true,
  Noot : false,
  Mies : true
};
```

8 **Paragraaf 5: Elegant code**

Voor het lezen van de namen van de katten en het toevoegen of verwijderen ervan maken we twee functies `addToSet` en `removeFromSet`. Als parameter geven we de verzameling katten mee en een array met de namen die toegevoegd of verwijderd moeten worden. We kijken naar de code van `addToSet`:

```
function addToSet(set, values) {
  for (var i = 0; i < values.length; i++) {
    set[values[i]] = true;
  }
}
```

Formele parameters van referentietype

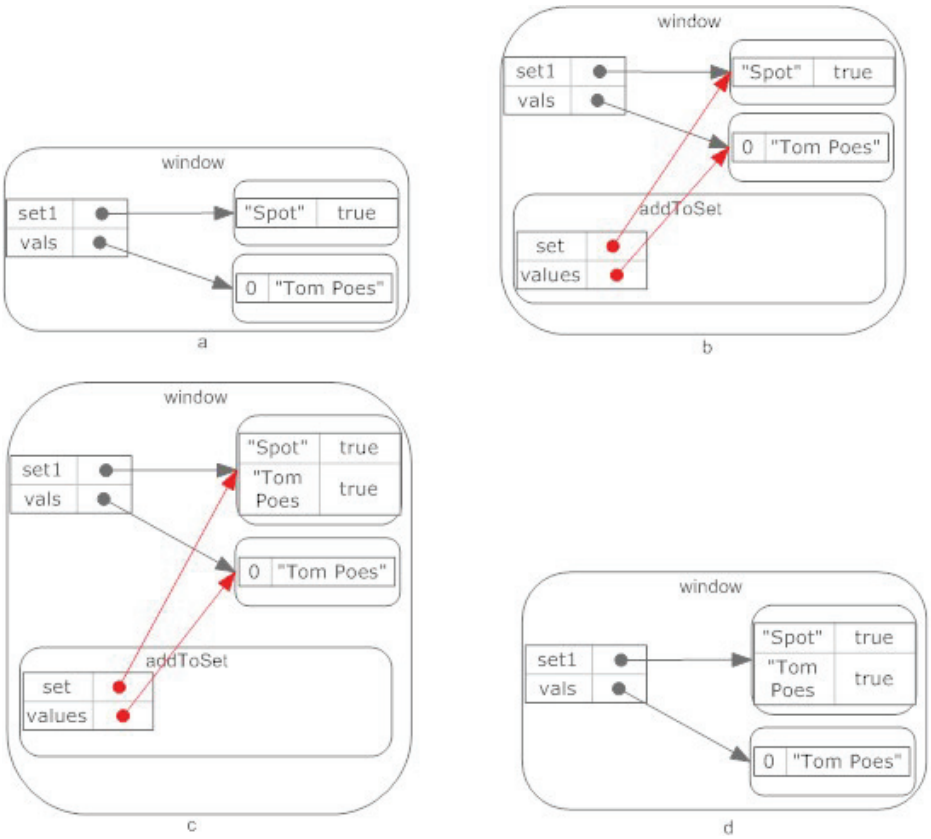
We gaan hier wat dieper op deze functie in omdat we hier voor het eerst te maken hebben met een functie waarvan de *parameters* geen primitieve typen zijn maar *objecten*. Ook bij zo'n functie wordt de inhoud van de argumenten gekopieerd naar de formele parameters; het zijn hier echter geen waarden van een primitief type maar *referenties*.

We lichten het gebruik van `addToSet` toe met een stukje code waarin we de functie aanroepen en op basis daarvan stellen we het geheugenmodel op.

```
1  function addToSet(set, values) {
2    for (var i = 0; i < values.length; i++) {
3      set[values[i]] = true;
4    }
5  }

6  var set1 = {
         "Spot" : true
       },
7      vals = ["Tom Poes"];
8  addToSet(set1,vals);
```

Merk eerst op dat we om de properties van `set` te benaderen niet de dot-notatie gebruiken maar die met [] omdat de naam van de property als variabele bekend is.

Figuur 5.10 toont het geheugenmodel



FIGUUR 5.10 Geheugenmodel bij de aanroep van addToSet

Figuur 5.10 a toont de situatie net voor de aanroep van regel 8; `set` wijst naar een object, `vals` naar een array.

Figuur 5.10 b toont de situatie net na de aanroep van regel 8; `set` verwijst naar het object waar `set1` naar verwijst en `values` verwijst naar de array waar `vals` naar verwijst.

Figuur 5.10 c toont de situatie na afloop van de lus van regels 2 t/m 4: het object heeft er een property bij gekregen. Na afloop van de functie-aanroep wordt de omgeving weer opgeruimd en krijgen we de situatie van figuur 5.10 d. Het uiteindelijke resultaat is dat object `set1` uitgebreid is met property " Tom Poes".

Let op!

Het gedrag van een parameter van een primitief type is dus anders dan het gedrag van een parameter van het referentietype. In het eerste geval wordt het corresponderende argument van de aanroep niet gewijzigd; in het tweede geval kan dat wel het geval zijn.

We merken ook op dat het resultaat van een toekenningsopdracht aan een property afhankelijk is van het bestaan van de property. Als een property bestaat, dan wordt door een toekenningsopdracht de waarde aangepast. Als de property niet bestaat, wordt het object uitgebreid met de property met bijbehorende waarde.

OPGAVE 5.5

Herschrijf functie `addToSet` zodat het een pure functie wordt. Hoe moet vervolgens de aanroep van deze functie luiden?

9 Paragraaf 6: Date

Constructor in combinatie met new

In deze paragraaf zien we een tweede manier om objecten te creëren: met behulp van een *constructor* en het keyword *new*. Objecten hebben, zoals we al gezien hebben, in tegenstelling tot bijvoorbeeld Java geen bijbehorende klasse. Sterker nog: JavaScript kent het gehele begrip klasse niet.

Later in deze cursus zullen we zien dat ook bij objecten een vorm van overerving mogelijk is: dat wordt in JavaScript geregeld door een Prototype. Als we een `Date`-object creëren, erven we een groot aantal methoden waarmee we bijvoorbeeld het jaar en de maand van een `Date`-object kunnen opvragen.

Net als in Java lopen de maanden van 0 t/m 11. Wees voorzichtig bij het vergelijken van `Date`-objecten. Bedenk dat `===` nagaat of de referenties gelijk zijn. Om de *inhoud* van twee verschillende `Date`-objecten te vergelijken, is het handig om functie `getTime` te gebruiken. Deze functie levert het aantal milliseconden dat verstreken is sinds 1 januari 1970. Als twee `Date`-objecten dezelfde datum representeren, zal `getTime` dezelfde waarde opleveren.

Exercise 4.6 is een opgave waarin een paar valkuilen zitten. Een ervan is dat het maandnummer van een `Date`-object één lager is dan het nummer dat wij in het dagelijks leven gebruiken. Dit betekent dat we niet alleen de string van de paragraaf uit elkaar moeten halen, maar in ieder geval ook de maand moeten omzetten naar een number om te kunnen aftrekken.

OPGAVE 5.6
Schrijf een functie `dagenTussen` met parameters `datum1` en `datum2`, die het volledig aantal dagen bepaalt dat verlopen is tussen `datum1` en `datum2`. Beide parameters zijn `Date`-objecten. Het resultaat kan ook negatief zijn. Het resultaat moet een geheel getal zijn. Raadpleeg eventueel de JavaScript reference object `Math`.

10 **Paragrafen 7, 8 en 9**

Deze paragrafen behandelen geen echt nieuwe, belangrijke zaken maar moeten wel bestudeerd worden.

OPGAVE 5.7
Schrijf een functie `maakIntArray` met parameter `n` die een array van `n` integers teruggeeft. De integers zijn random getallen tussen `0` en `n` (grenzen niet inbegrepen).
Arrays kennen een standaardfunctie `indexOf` met een parameter `element`, die de (eerste) index oplevert van `element` als `element` in de array voorkomt. Als `element` niet voorkomt is het resultaat `-1`.
Schrijf ten slotte een programma om een array van `1000` getallen te genereren en na te gaan of `35` in die array voorkomt en als dat zo is de index af te drukken.

11 **Chapter 5, paragrafen 1 en 2.1**

Vraag: Wat zien gebruikers als een webpagina fouten in JavaScript bevat?
Antwoord: Niets, dat wil zeggen geen foutmeldingen!

Programmeerfouten

Omdat een browser een 'vergevingsgezinde' omgeving is waarin een programma draait, worden *programmeerfouten* niet standaard gesignaleerd. Die fouten kunnen daarentegen wel leiden tot ongewenste resultaten van een programma. Moderne browsers kunnen wel zo geconfigureerd worden dat ze fouten aangeven, maar dat is voor de gebruiker alleen maar vervelend. Het is dan ook zaak programmeerfouten te voorkomen. Wat daaraan bijdraagt is om eerst een goed ontwerp van de code te maken. Het zorgvuldig testen van functies kan fouten aan het licht brengen, zodat u ze kunt verhelpen.

Twee soorten fouten

Paragraaf 1 beschrijft *twee soorten fouten*: fouten met betrekking tot het gegevenstype van een parameter en fouten waarbij het type wel juist is maar de waarde niet. In beide gevallen zal er wel een (foutief) resultaat berekend worden.

Dit soort fouten kan voorkomen worden door in de documentatie van een functie nauwkeurig op te nemen wat het verwachte gegevenstype van een parameter is en wat het bijbehorende waardenbereik is. We kunnen deze documentatie in de vorm van commentaar aan een functie toevoegen. De programmeur die de functie gebruikt (meestal bent u dat zelf!) moet er dan voor zorgen dat bij de aanroep van de functie gecontroleerd wordt of het type en/of de waarde voldoen aan de specificaties. We tonen u hier nogmaals functie `power` maar nu voorzien van commentaar:

Documentatie in de vorm van commentaar

```
/*
Parameters:
  base     - een getal waarvan de macht bepaald moet worden
  exponent - een geheel getal groter dan of gelijk aan 0
             dat de macht voorstelt waartoe base verheven
             moet worden
*/
function power(base, exponent) {
  var result = 1;
  for (var count = 0; count < exponent; count++)
    result *= base;
  return result;
}
```

Preconditie

De eisen die we stellen aan parameters worden de *preconditie* van de functie genoemd. We hebben hier dus inderdaad met een preconditie te maken: `base` moet een geheel getal zijn en exponent moet een geheel getal `>= 0` zijn.

Testen vooraf

De programmeur heeft de verantwoordelijkheid vóór het aanroepen van deze functie te zorgen dat de preconditie geldig is.

Het tweede type fouten (waarbij het type wel juist is maar de waarde niet) kan soms verholpen worden met commentaar zoals in het voorbeeld van functie `power`, maar meestal moeten we het op een andere manier doen. Om aan te geven dat het resultaat van een functie een uitzonderingssituatie betreft, kunnen we soms een speciale waarde teruggeven. Deze manier van werken kennen we al, bijvoorbeeld bij het gebruik van array-methode *indexOf*. Als het argument niet voorkomt, wordt de waarde `-1` als resultaat opgeleverd.

Testen achteraf

De programmeur heeft in dit geval de verantwoordelijkheid na de aanroep van de functie te controleren of het resultaat niet de bijzondere waarde is.

Probleem bij de tweede oplossing is dat er niet altijd een waarde te verzinnen is die als terugkeerwaarde gekozen kan worden om een uitzonderingssituatie te signaleren. Daarnaast kan het steeds achteraf testen van de terugkeerwaarde vervelend zijn, vooral bij een herhaling.

12 Paragraaf 2.2: Exception handling

Exception

Moderne, objectgeoriënteerde talen kennen een mechanisme voor het afhandelen van fouten, dat *exception handling* genoemd wordt. Het idee is dat als op een bepaalde plaats in een programma een onverwachte situatie ontstaat, de runtime-omgeving of de programmeur een *exception* opgooit die op een andere plaats in het programma afgehandeld ('gevangen') wordt.

try-catch

Het opvangen van een exception gebeurt met de `try-catch`-constructie:

```
try {
  //code die mogelijk een exception kan opgooien
}  catch(exception) {
  //wat er moet gebeuren bij een exception
}  finally {
  //optioneel om de boel bijvoorbeeld weer op te ruimen
}
```

De werking van het mechanisme is aldus: als tijdens de verwerking van het `try`-blok een exception optreedt, wordt direct naar de code van het `catch`-blok gesprongen en de code daarvan uitgevoerd. Als er geen exception optreedt wordt het `catch`-blok overgeslagen.

De parameter `exception` bevat informatie over de exception, die we in het `catch`-blok kunnen gebruiken. Deze informatie is helaas afhankelijk van het type browser dat gebruikt wordt, maar alle browsers hebben in ieder geval property `message` gemeenschappelijk.

finally

De `try-catch`-constructie kent nog een optioneel onderdeel `finally` dat gebruikt kan worden voor 'cleanup code'. Als een `finally`-blok aanwezig is, wordt de code ervan *altijd* uitgevoerd, ook al treedt er geen exception op.

We bekijken een voorbeeld waarbij de runtime-omgeving een exception opgooit. Als we een array maken maar als aantal elementen een negatief getal als argument meegeven, gooit de runtime-omgeving een exception op. Als u de volgende regel in de console intypt

```
var ar = new Array(-20);
```

ziet u de volgende exception:

```
Exception: RangeError: invalid array length
```

Als u de volgende code intypt

```
try {
  var ar = new Array(-20);
}
catch(error) {
  print("Fout opgetreden:",error.message);
}
```

ziet u

```
Fout opgetreden:invalid array length
```

Keyword throw

We kunnen als programmeur ook zelf een exception opgooien. Dat doen we met het *keyword throw*. We kunnen `throw` gebruiken in combinatie met een string of object zoals in de paragraaf van het tekstboek gebeurt, maar mooier is het om een nieuw `Error`-object te maken:

```
throw new Error("Er is iets misgegaan");
```

Als tijdens de uitvoering van een programma deze regel uitgevoerd wordt, gebeurt het volgende:

Het programma-blok dat de `throw`-clausule bevat, wordt direct beëindigd. Dit betekent bijvoorbeeld dat een `if`-constructie of een `for`-lus direct worden beëindigd. Daarna gaat dit proces door totdat een

try-catch-constructie gevonden is waarvan de throw-clausule direct of indirect een onderdeel is. Bij dit proces worden ook eventuele functies beëindigd, waarbij de runtime-omgeving gebruikmaakt van de callstack. Als er geen try-catch-constructie gevonden wordt, moet de browser de exception afhandelen.

Mogelijke exception in commentaar opnemen

Als een functie één of meer exceptions kan opgooien, is het zinvol dat bij de documentatie te vermelden, zodat de programmeur de aanroep van de functie in een try…catch-blok kan plaatsen.

OPGAVE 5.8

Gegeven is het volgende programma:

```
/*
  Parameters:
    p: willekeurig getal
    q: willekeurig getal
*/
function delen(p,q){
  if (q === 0){
    throw new Error("delen door nul niet toegestaan");
  }
  return p/q;
}

function invoer() {
  var a = 6,
      i ,
      res = 0 ;
  for (i = -2; i < 2; i += 1){
    res = res + delen(a,i);
  }
  return res;
}

try {
  print(invoer());
}
catch(error) {
  print(error.message);
};
```

Speel zelf voor runtime-omgeving en bedenk wat afgedrukt wordt door dit programma.

We illustreren de verschillende manieren waarop we met fouten kunnen omgaan nog met een voorbeeld.

In de wiskunde en statistiek wordt vaak het begrip faculteit gebruikt. De *faculteit* van een natuurlijk getal n, genoteerd als n! (n *faculteit*), is gedefinieerd als het product van de getallen 1 tot en met n. Daarbij is ook nog 0! gedefinieerd als 1.

We kunnen voor het berekenen van n! een functie `faculteit` schrijven:

```
/*
Parameters:
  n:  geheel getal >= 0
*/
function fac(n) {
  var res = 1,
      i;
  for(i = 1;i <= n;i += 1) {
    res = res * i;
  }
  return res;
}
```

Deze functie werkt correct als de gebruiker ervoor zorgt dat `n` inderdaad een natuurlijk getal is (een positief getal). Faculteiten kunnen snel groot worden. Zo kan het programma nog net correct `171!` berekenen (althans een benadering daarvan) maar bij nog hogere waarden voor `n` is de uitkomst `infinity`. Daarom moeten we nog de eis `n<172` toevoegen aan de documentatie.

Stel nu dat we een function `func` hebben die een resultaat `n` oplevert waarvan we de faculteit willen berekenen. We moeten dan testen of n een integer is en, zo ja, of `n >= 0` en `< 172` is.

Het testen of een variabele een integer-waarde heeft, is nog niet zo eenvoudig in JavaScript; er is geen standaardfunctie voor. Als we verderop in deze cursus reguliere expressies behandeld hebben, kunnen we dat probleem oplossen. Voorlopig doen we even of er een standaardfunctie `isInteger` beschikbaar is.

We moeten voor het gegeven probleem het volgende stukje code opnemen om aan de preconditie en het juiste gegevenstype voor de parameter te voldoen:

```
var n = func();
if (isInteger(n) && n >= 0 && n < 172) {
  print(fac(n));
};
```

Een alternatief is om in `fac` zelf de controle op parameter n te leggen. In dat geval is er geen preconditie nodig. We krijgen dan:

```
/*
  Parameters:
    n:  geheel getal >= 0

  Throws:
    Error als n >= 172 of n < 0
*/
function fac(n) {
  var res = 1,
      i;
  if (isInteger(n) && n >= 0 && n < 171) {
    for(i = 1; i <= n; i += 1) {
      res = res * i;
    }
```

```
    return res;
  } else {
    throw new Error("Faculteit kan niet berekend worden");
  }
}

var n = func();
try {
  print(fac(n));
}
catch(error) {
  print(error.message);
}
```

13 **Systematisch functies ontwerpen**

Voordat u een goede functie kunt schrijven, is het belangrijk eerst een ontwerp voor die functie te maken. U heeft hiervan al een paar informele voorbeelden gezien, bijvoorbeeld het gebruik van pseudo-taal. In deze paragraaf gaan we dat meer systematisch aanpakken en u een werkwijze aan de hand doen die we vanaf nu zoveel mogelijk zullen toepassen. Hoewel deze werkwijze voor simpele gevallen alleen maar extra werk lijkt op te leveren, zal zij u wel degelijk helpen bij complexere problemen. Deze werkwijze is voor een belangrijk deel ontleend aan de cursus 'Introduction to Systematic Program Design' van Gregor Kiczales en gebaseerd op het boek 'How to Design Programs' van Felleisen, Findler, Flatt en Krishnamurthi.

We gebruiken bij het ontwerpen van een functie de volgende stappen:

1 Leg signatuur en doel van de functie vast; schrijf de functie heading en maak een stub.
2 Bedenk een aantal testgevallen voor de functie.
3 Geef in pseudocode aan hoe het gewenste resultaat bereikt kan worden; voor eenvoudige gevallen kan deze stap overgeslagen worden.
4 Schrijf de body van de functie.
5 Test de functie op basis van de testgevallen.

Signatuur

Met de *signatuur* geven we de *naam* van de functie aan, de *namen* en *typen* van de *parameters* en het *type* van de *terugkeerwaarde*. We geven deze signatuur in de vorm van commentaar. U weet inmiddels dat we in JavaScript geen expliciete types kunnen specificeren, maar het is zinvol te weten voor welk type parameters de functie dient.

Stub

Een *stub* is een eerste implementatie van de functie, die iets teruggeeft van het juiste type, die gebruikt kan worden om het gedrag van de functie te simuleren. Met zo'n stub kunnen de testen in ieder geval uitgevoerd worden, hoewel het resultaat bijna altijd zal zijn dat de testen falen.

We hebben eerder functie `power` bestudeerd. We tonen u nu het ontwerp van deze functie, waarbij we gebruikmaken van het stappenplan.

1 Leg signatuur en doel van de functie vast; schrijf de functie heading en maak een stub.

```
/*
  Function: power(base: number,exponent: integer) -> number

  Goal: berekent base tot de macht exponent

  Parameters:
    base     - een getal waarvan we de macht moeten geven
    exponent - een geheel getal groter dan of gelijk aan 0
               dat de macht voorstelt waartoe base verheven
               moet worden

  Returns:
    base tot de macht exponent

*/
```

De notatie van de parameters is steeds *naam: type*. Bij de typen kunnen we ook integers aangeven, hoewel JavaScript daar geen eigen type voor heeft. Na de `->` zetten we het type van de terugkeerwaarde.

Op basis van deze specificatie kunnen we een stub in JavaScript schrijven, bijvoorbeeld:

```
function power(base, exponent){
  return 1;
}
```

Deze implementatie van functie power levert altijd `1`. Bij stap 4 zullen we deze code vervagen door de 'echte' code.

2 Bedenk een aantal testgevallen voor de functie. Het systematisch bedenken van goede testgevallen is een probleem op zich. We maken hier onderscheid tussen gehele en decimale getallen, al dan niet positief. De verwachte uitkomst kunnen we eventueel als een expressie formuleren. Dit laatste heeft als voordeel dat we al een beetje een idee voor een algoritme krijgen.

```
/*
  Tests:
  power(3, 1) = 3
  power(-3, 1) = -3
  power(2, 4) = 2 * 2 * 2 * 2
  power(-2, 4) = -2 * -2 * -2 *-2
  power(10, 5) = 10 * 10 * 10 * 10 * 10
  power(-3.1, 2) = -3.1 * -3.1
  power(-3.1, 3) = -3.1 * -3.1 * -3.1
  power(0, 1) = 0
  power(5, 0) = 1
  power(0, 0) = 1
*/
```

De laatste drie testen ontlenen we aan de wiskunde: 0 tot iedere macht levert 0 behalve 0 tot de macht 0 dat 1 levert. Ieder getal tot de macht nul levert 1.

3 Op basis van de testgevallen kunnen we zien dat we herhaald moeten vermenigvuldigen en wel precies zo vaak als parameter `exponent` voorstelt. We beginnen met een startwaarde die we in een lus steeds wijzigen. Waar een sommatie een startwaarde van `0` heeft, moeten we bij herhaald vermenigvuldigen een startwaarde van `1` gebruiken.

```
/*
  res = 1
  herhaal exponent keer
    res = res * base
  return res
*/
```

4 We zetten de pseudocode om naar JavaScript en combineren de code met die van de heading

```
function power(base, exponent){
  var res = 1,
      count;
  for (count = 0; count < exponent; count++) {
    res *= base;
  }
  return res;
}
```

5 Test de functie op basis van de testgevallen.
In de volgende leereenheid wordt een framework om systematisch tests te schrijven behandeld. Nu behelpen we ons met conditionele expressies:

```
print(<test> ? "OK" : "Fout in test");
```

Om alles wat te stroomlijnen, definiëren we twee string-constanten `OK` en `NOK` om de tekst te representeren. De eerste test luidt dan:

```
const OK = "Test OK",
      NOK = "Test not OK";
print(power(3,1) === 3 ? OK : NOK);
```

Als we zo de testen afwerken, ontstaan problemen bij de testen met grote getallen en met decimale getallen. Omdat JavaScript geen integers kent, worden alle getallen met eindige precisie opgeslagen en dus afgerond. Het vergelijken van grote getallen en decimale getallen levert meestal onjuiste uitkomsten. Zo wordt in ons voorbeeld power(-3.1,2) berekend met als uitkomst 9.610000000000001. Als we de uitkomst vergelijken met 9.61 levert dit `false` op.

We kunnen dit probleem oplossen door de uitkomst af te ronden op het aantal decimalen van het getal waarmee de uitkomst vergeleken wordt, hier dus 9.61, dat wil zeggen twee decimalen. Een andere mogelijkheid is om een klein getal *eps* te definiëren en na te gaan of het verschil van de berekende waarde en de verwachte waarde kleiner is dan `eps`. Wel moeten we dan de absolute waarde van het verschil nemen.

eps

$$|\text{berekende waarde} - \text{referentie}| < \text{eps}$$

Voor `eps` kunnen we bijvoorbeeld kiezen `1e-12`. De absolute waarde kunnen we berekenen met `Math.abs`. We gebruiken `eps` voor al onze testgevallen.

Als een test faalt, zien we alleen dat de test niet `OK` is. Het is duidelijker als we daarbij ook zowel de actuele waarde als de verwachte waarde afdrukken. Voor het testen kunnen we dan de functie `test` gebruiken:

```
const OK = "Test OK",
      NOK = "Test not OK",
      EPS = 1e-12;

function test(actual,expected){
  print(Math.abs(expected-actual) < EPS ? OK : NOK +
        " expected:" + expected + " actual: "+ actual);
}
```

Zo krijgen we voor het testen:

```
test(power(3, 1), 3);
test(power(-3, 1), -3);
test(power(2, 4), 16);
test(power(-2, 4), 16);
test(power(10, 5) ,100000);
test(power(-3.1, 2), (-3.1 * -3.1));
test(power(-3.1, 3), (-3.1 * -3.1 * -3.1));
test(power(0, 1), 0);
test(power(5, 0), 1);
test(power(0, 0), 1);
```

Als bij het testen blijkt dat een of meer testen falen, moeten we eerst nagaan of de test zelf wel valide is. Als dat het geval is, is er iets met de functie mis en zal deze verbeterd moeten worden.

De complete code van de functie luidt:

```
const OK = "Test OK",
      NOK = "Test not OK",
      EPS = 1e-12;

/*
  Function: power(base: number, exponent: integer) -> number

  Goal:
    berekent base tot de macht exponent

  Parameters:
    base     - een getal waarvan we de macht moeten bepalen
    exponent - een geheel getal groter dan of gelijk aan 0
               dat de macht voorstelt waartoe base verheven
               moet worden

  Returns:
    base tot de macht exponent


*/
```

```
function power(base, exponent){
  var res = 1,
      count;
  for (count = 0; count < exponent; count++){
    res *= base;
  }
  return res;
}
```

Testen:

```
test(power(3, 1), 3);
test(power(-3, 1), -3);
test(power(2, 4), 16);
test(power(-2, 4), 16);
test(power(10, 5), 100000);
test(power(-3.1, 2), (-3.1 * -3.1));
test(power(-3.1, 3), (-3.1 * -3.1 * -3.1));
test(power(0, 1), 0);
test(power(5, 0), 1);
test(power(0, 0), 1);
```

We zullen de code van de testen meestal in een apart bestand zetten, zoals we in de volgende leereenheid zullen zien.

ZELFTOETS

1 Gegeven is de volgende code:

```
var aantal = 5;
function verdubbel(aantal) {
  return 2 * aantal;
}
var uitkomst = verdubbel(4);
```

a Welke waarde heeft `uitkomst` na het uitvoeren van deze code? Teken een geheugenmodel dat de situatie weergeeft op het moment dat function `verdubbel` op het punt staat beëindigd te worden.

b Wat is de waarde van `uitkomst` als we de volgorde van de code aldus wijzigen?

```
var aantal = 5,
    uitkomst = verdubbel(4);
function verdubbel(aantal) {
  return 2 * aantal;
}
```

c Wat is de waarde van `uitkomst` als we een functie door middel van een functie-expressie specificeren?

```
var aantal = 5,
    uitkomst = verdubbel(4)
    verdubbel = function(aantal) {
      return 2 * aantal;
    };
```

2 Gegeven is de volgende JavaScript-code:

```
var lijst1 = [1, 2, 3],
    lijst2 = [5, 6, 7, 8];
```

a We willen een nieuwe lijst `lijst3` die bestaat uit de elementen van `lijst1` gevolgd door die van `lijst2`, maar deze laatste in omgekeerde volgorde. Schrijf hiervoor een programma dat gebruikmaakt van herhalingsstructuren. Geef eerst een opzet in pseudocode.
b Kijk daarna in de JavaScript-reference of u geschikte functies kunt vinden zodat u met minder code het probleem kunt oplossen. Geef de code.

3 Gegeven de volgende programmacode:

```
var obj1 = {
      a : 0,
      b : 1,
      c : 3
    },
    ar1 = [1, 2, 5, 10],
    s = "b";
ar2 = ar1;
ar2[obj1[s]] = 8;
print(ar1);
```

Wat wordt door dit programma afgedrukt? Maak gebruik van een geheugenmodel.

4 Opgave 1 van de zelftoets van leereenheid 4 is niet robuust. Als u in plaats van een getal tekst invoert, zal ten slotte de waarde `NaN` als gemiddelde afgedrukt worden. Maak het programma robuust zodat het aangeeft dat de invoer verkeerd is en ten slotte het correcte gemiddelde bepaalt.
HINT: schrijf een function `leesGetal` die een exception opgooit als de invoer geen correct getal is en anders het getal teruggeeft. Gebruik dan deze functie in uw programma.

5 Schrijf een functie `calcBMI` die de Body Mass Index (BMI) berekent op basis van lichaamsgewicht en lengte. De BMI is een index voor het gewicht in verhouding tot de lichaamslengte. De BMI geeft een schatting van het gezondheidsrisico met betrekking tot het gewicht. De index wordt berekend door het gewicht in kilogram te delen door het kwadraat van de lengte in meters. Het is gebruikelijk dat als parameters het gewicht in kilogram en de lengte in centimeter worden gebruikt. De BMI-index kan bepaald worden als de lengte ligt tussen 75 en 225 cm en als het gewicht ligt tussen 30 en 250 kg.

Pas de ontwerpstappen toe op dit probleem. Laat de functie een exception opgooien als het gewicht of de lengte niet tussen de gegeven grenzen ligt.

TERUGKOPPELING

1 Uitwerking van de opgaven

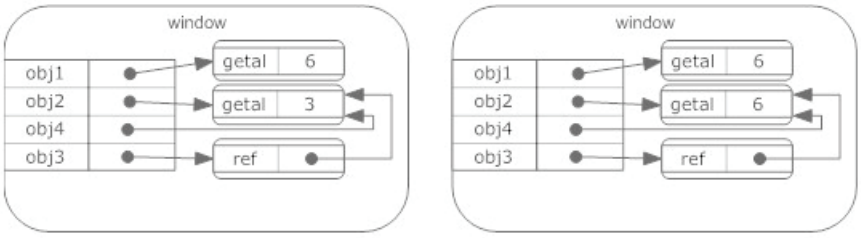
5.1 De oppervlakte kan berekend worden door de oppervlakte van de witte cirkel van die van de rode af te trekken. Voor het berekenen van de oppervlakte van een cirkel schrijven we een functie `cirkelOppervlakte` met parameter `r`. Voor de oppervlakte van de schijf schrijven we een functie `cirkelSchijfOppervlakte` met twee parameters `r1` en `r2`.

```
function cirkelOppervlakte(r) {
  return Math.PI * r * r;
}

function cirkelSchijfOppervlakte(r1, r2) {
  return cirkelOppervlakte(r1) - cirkelOppervlakte(r2);
}

var r1 = 5,
    r2 = 4;
print(cirkelSchijfOppervlakte(r1, r2));
```

5.2 De code luidt:

```
function isWortelVan(g) {
  return function(n) {
    return n * n === g;
  }
}

var test = isWortelVan(16);
print(test(4));
```

5.3 Het geheugenmodel ziet eruit zoals figuur 5.11 toont



FIGUUR 5.11 Geheugenmodel bij opgave 5.3

Er zijn drie aliassen die naar `{getal: 3}` wijzen: `obj2`, `obj3` en `obj4`. Als dan in regel 5 de waarde van `obj4` veranderd wordt, veranderen `obj2` en `obj3` mee. Het resultaat van de print-opdrachten is dan :

```
6
true
```

5.4 De code luidt:

```
var obj = {
      Aap :  true,
      Noot : false,
      Mies : true
    },
    keys = Object.keys(obj),
    lengte = keys.length,
    i;

for (i = 0; i < lengte; i += 1){
  print(keys[i] + ':' + obj[keys[i]]);
}
```

5.5 De code luidt:

```
function addToSet(set, vals) {
  var mijnSet = {},
      i,
      prop,
      nvals = vals.length;
  for (prop in set) {
    mijnSet[prop] = true;
  }
  for(i = 0 ; i < nvals; i += 1) {
    mijnSet[vals[i]] = true;
  }
  return mijnSet;
}
```

De functie kopieert eerst de properties van `set` naar `mijnSet` en voegt daarna de elementen van `vals` toe. Ten slotte wordt `mijnSet` als terugkeerwaarde opgeleverd. De aanroep moet nu luiden:

```
cats = addToSet(cats, names);
```

5.6 De code luidt:

```
function dagenTussen(dat1, dat2) {
  const URENPERDAG = 24,
        MINUTENPERUUR = 60,
        SECONDENPERMINUUT = 60,
        MILLISECONDEN = 1000;
  return Math.round((dat2 - dat1) / URENPERDAG
    / MINUTENPERUUR
    / SECONDENPERMINUUT
    / MILLISECONDEN, 0);
}

show(dagenTussen(dat1, dat2));
```

5.7 De code luidt:

```
function getRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}

function maakArray(n) {
  var ar = [],
      i;
  const MIN = 1;
  for(i = 1; i <= n; i += 1) {
    ar.push(getRandomInt(MIN, n));
  }
  return ar;
}

var     ar = maakArray(AANTAL),
    index = ar.indexOf(ZOEK);
const AANTAL = 1000,
      ZOEK = 35;
if (index !== -1) {
  show("",
       ZOEK,
       " komt voor in array",
       "nl op positie ",
       index);
}
else {
  show("",
       ZOEK,
       " komt niet voor in array");
}
```

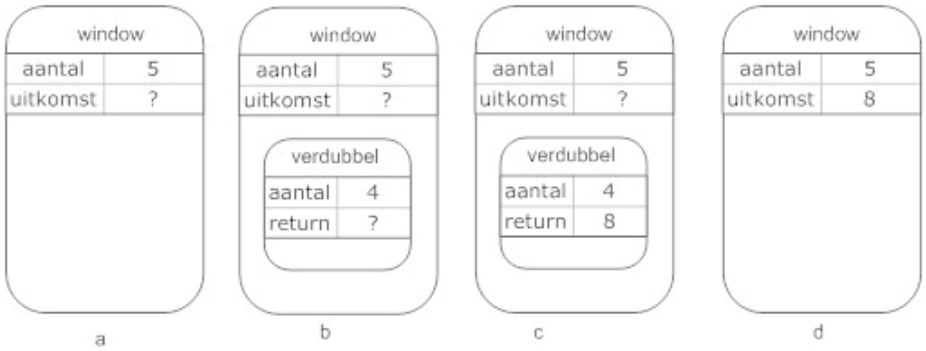
5.8 Het programma roept herhaald de functie `delen` aan. Deze methode gooit een exception op als parameter `q` de waarde `0` heeft. De lus-constructie van function `invoer` roept op een bepaald moment `delen(6,0)` aan. Hierdoor wordt de `throw`-clausule van function `delen` uitgevoerd. Hierna beëindigt de runtime-omgeving eerst de `if`-constructie van function `delen`. Omdat in delen geen omhullende `try-catch`-constructie te vinden is, wordt function `delen` beëindigd en gaat de runtime-omgeving verder met de regel van function `invoer` waarin `delen` werd aangeroepen. De `for`-lus wordt dan beëindigd en ook function `invoer` in zijn geheel. Daarna wordt verdergegaan met de regel waarin de aanroep van function `invoer` staat. Omdat deze aanroep in een `try-catch`-blok staat, wordt de `catch` van dit blok uitgevoerd. Hier wordt `error.message` afgedrukt. Deze message is de tekst die we aan function `Error` hebben meegegeven in de `throw`-clausule. Dus de tekst

```
delen door nul niet toegestaan
```

wordt afgedrukt.

2 Uitwerking van de zelftoets

1 In geval a hebben we een globale variabele `aantal` met waarde `5` en een parameterwaarde die in functie `verdubbel` als lokale variabele gebruikt wordt; de lokale `aantal` 'overschaduwt' de globale `aantal`. De lokale `aantal` krijgt bij aanroep waarde `4` en wordt gebruikt in de `return` die waarde `8` teruggeeft. `uitkomst` heeft dus de waarde `8` na afloop. Figuur 5.12 toont het geheugenmodel.



FIGUUR 5.12 Geheugenmodel bij opgave 1

In geval b hebben we hetzelfde antwoord, hoewel de functie wordt aangeroepen voordat deze gedefinieerd is. Hoisting bij functiedeclaratie zorgt ervoor dat de functiedefinitie voor de runtime-omgeving toch voor de aanroep staat en dus hetzelfde resultaat geeft als bij 1.

In geval c treedt een foutmelding op, omdat bij een functie-expressie geen hoisting optreedt. De aanroep van `verdubbel` zal een foutmelding geven omdat deze functie nog niet gedefinieerd is.

2 a Opzet in pseudocode:

```
Creëer lege array lijst3
Doorloop lijst1 en voeg de elementen toe aan lijst3
Doorloop lijst2 in omgekeerde volgorde en voeg de elementen
toe aan lijst3
```

Code:

```
var lijst1 = [1, 2, 3],
    lijst2 = [5, 6, 7, 8],
    aantal1 = lijst1.length,
    aantal2 = lijst2.length,
    i,
    j,
    lijst3 = [];

for(i = 0;i < aantal1;i += 1){
  lijst3.push(lijst1[i]);
}
for (i = aantal2-1;i >= 0; i -= 1){
  lijst3.push(lijst2[i]);
}
print(lijst3);
```

b Voor het aan elkaar plakken van arrays bestaat functie `concat` en voor het omkeren van de volgorde van de elementen van een array bestaat functie `reverse`. De code wordt dan:

```
var lijst1 = [1, 2, 3],
    lijst2 = [5, 6, 7, 8],
    lijst3;

lijst3 = lijst1.concat(lijst2.reverse());
print(lijst3);
```

3 [1, 8, 5, 10]

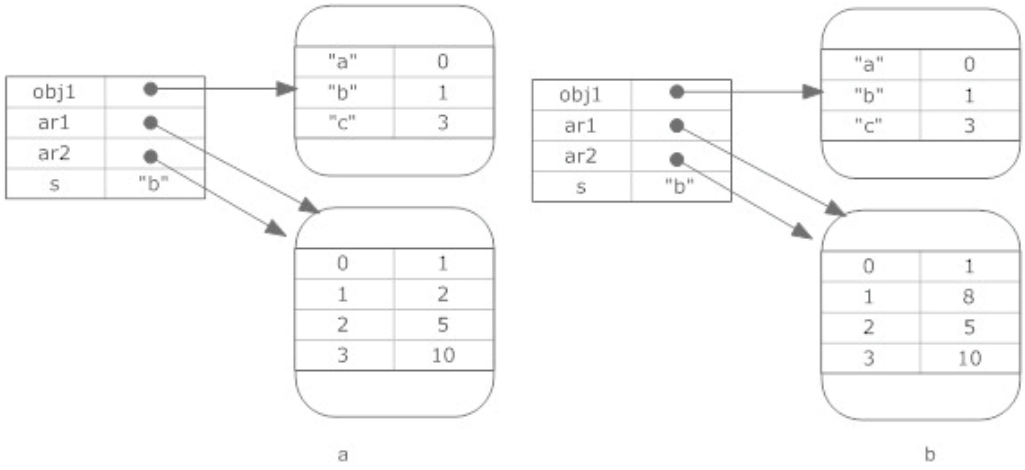
Toelichting: Na de variabelendeclaratie geeft figuur 5.12 a de situatie weer: `ar1` en `ar2` zijn aliassen. Bij de evaluatie van

```
ar2[obj1[s]] = 8;
```

worden de rechte haken van binnen naar buiten geëvalueerd.

`s` heeft de waarde "b ", dus wordt `obj1["b"]` opgezocht `= 1` en dit levert `ar2[1]` die de waarde `8` krijgt, zie figuur 5.13 b.

Omdat `ar1` en `ar2` aliassen zijn, wordt bij het afdrukken van `ar1` de nieuwe waarde afgedrukt.



FIGUUR 5.13 Geheugenmodel bij opgave 2

4 De code luidt:

```
var invoer,
    aantal = 0,
    som = 0,
    gemiddelde;
const STOPGETAL = 777;

function leesGetal(tekst) {
  var invoer = Number(prompt(tekst, ""));
  if (isNaN(invoer)) {
    throw new Error("Invoer is geen geldig getal");
  }
  return invoer;
}
/* Vraagt net zolang aan de gebruiker om een getal totdat
deze een geldig getal heeft ingevoerd
*/
function leesDoorGetal(tekst) {
  var ok = false,
      invoer;
  while (!ok) {
  try {
    invoer = leesGetal(tekst,"");
    ok = true;
  }
  catch(e) {
    alert(e.message);
  }
 }
 return invoer;
}

print("Dit programma berekent het gemiddelde van de door u
ingevoerde getallen");
invoer = leesDoorGetal("Voer een getal in, 777=stoppen","");
while(invoer !== STOPGETAL) {
  som = som + invoer;
  aantal += 1;
  invoer = leesDoorGetal("Voer een getal in, 777=stoppen",
"");
}
if (aantal > 0) {
  gemiddelde = som / aantal;
  print("Het gemiddelde = " + gemiddelde);
}
else {
  print("Het gemiddelde kon niet bepaald worden");
}
```

5 De uitwerking luidt:

```
/*
function calcBMI(lengte:integer,gewicht:number)->number
Goal: berekent de BMI op basis van gewicht(kg)/lengte(m)^2

Parameters:
  lengte:  lichaamslengte in cm
           (minimum 75 cm, maximum 225 cm)
  gewicht: lichaamsgewicht in kg
           (minimum 30 kg, maximum 250 kg)

Returns: de BMI

Throws: exception als een of meer parameters een waarde
hebben buiten aangegeven interval

*/

/* Stub
function calcBMI(lengte, gewicht){
    return 25.0;
}
*/

/*Tests:
calcBMI(180, 80) = 80 / (1.8 * 1.8)
calcBMI(100, 50) = 50 / (1 * 1)
calcBMI(200, 50) = 50 / (2 * 2)
calcBMI(300, 100) = exception
calcBMI(74, 50) = exception
calcBMI(75, 50) = 50 / (0.75 * 0.75)
*/
```

Implementatie:

```
const OK = "Test OK",
      NOK = "Test not OK",
      EPS = 1e-2;

function test(actual, expected) {
    print(Math.abs(expected-actual) < EPS ? OK : NOK +
          " expected: " + expected + " actual: " + actual);
}



/*
function calcBMI(lengte:integer,gewicht:number)->number
Goal: berekent BMI-index op basis van gewicht
(kg)/lengte(m)^2

Parameters:
  lengte:  lichaamslengte in cm
           (minimum 75 cm, maximum 225 cm)
  gewicht: lichaamsgewicht in kg
           (minimum 30 kg, maximum 250 kg)

Returns: de BMI-index
Throws exception als een of meer parameters een waarde
hebben buiten aangegeven interval

*/

function calcBMI(lengte, gewicht) {
  if (lengte < 75 ||
       lengte > 225 ||
       gewicht < 30 ||
       gewicht > 250) {
    throw new Error("Lengte of gewicht onjuist: lengte moet
tussen 75 en 225 liggen; gewicht tussen 30 en 250");
  }
  return gewicht / ((lengte / 100) * (lengte / 100));
}

test(calcBMI(180, 80), 80 / (1.8 * 1.8));
test(calcBMI(100, 50), 50 / (1 * 1));
test(calcBMI(200, 50), 50 / (2 * 2));
test(calcBMI(75, 50), 50 / (0.75 * 0.75));

try {
  calcBMI(300,100);
}
catch(error) {
    print(error.message);
}

try {
  calcBMI(74,50);
}
catch(error) {
    print(error.message);
}
```