Lussen

Om code **herhaaldelijk** ***(Eng. repeatedly)*** uit te voeren gebruiken we **lussen** ***(Eng. loops).*** In JavaScript zijn er verschillende soorten lussen mogelijk, maar ze doen allen in essentie hetzelfde: ze herhalen een actie meerdere keren totdat een bepaalde conditie niet waar is of een einde bereikt is. De verschillende soorten lussen bieden verschillende manieren aan om het start- en eindpunt te bepalen.

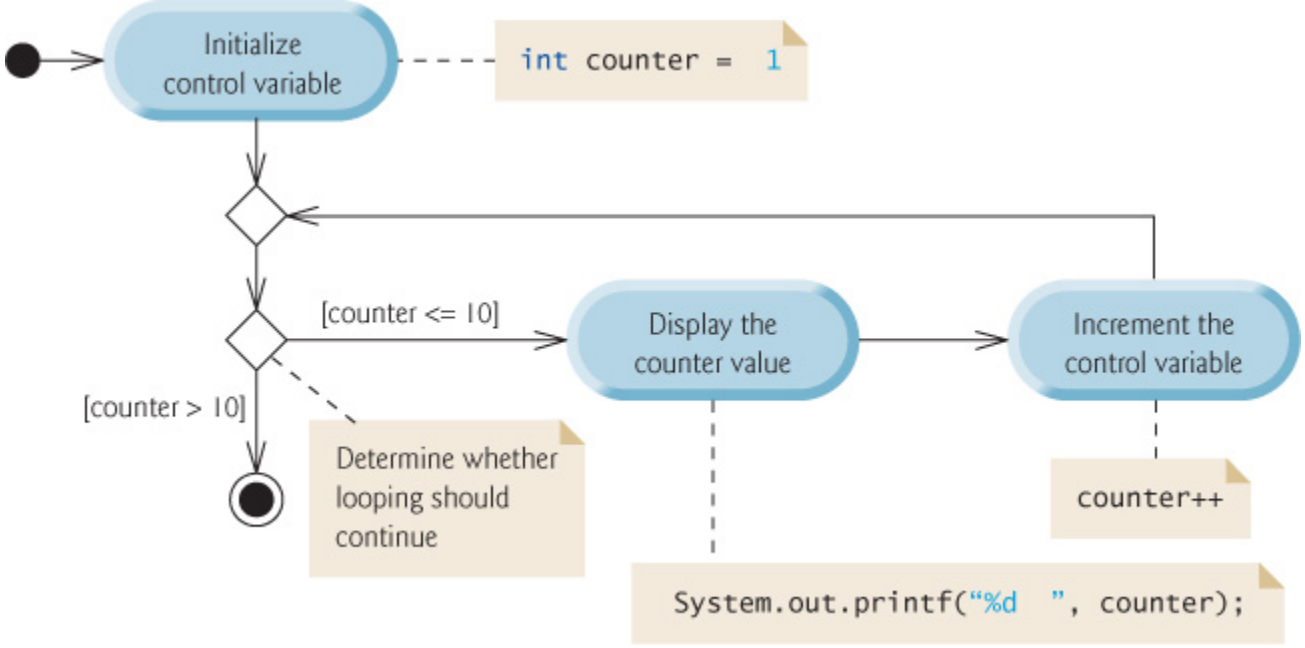
De mogelijk iteratie statements zijn: `for`, `for…in`, `for…of`, `while` en `do…while`.

`for` statement

Een `for` statement herhaalt een anctie totdat een bepaalde conditie **niet waar** of `false` is. Een `for` statement ziet er als volgt uit:

```js
for («initial-expression»; «condition»; «increment-expression»)
  «statement»;
```

> Wanneer een `for`-lus wordt uitgevoerd, zal het volgende gebeuren:
>
> 1. `«initial-expression»` wordt uitgevoerd. Meestal bevat deze een of meerdere variabelen die dienen als interators. Deze variabelen worden gedeclareerd en geïnitialiseerd.
> 2. De `«condition»` wordt geëvalueerd. Als de waarde van de conditie waar of `true` is, dan volgt de volgende iteratie. Is de conditie niet waar of `false`, dan wordt de `for`-lus beëindigd.
> 3. Indien `true`, dan worden de acties binnen het statement uitgevoerd. Gebruik altijd een block statement (`{ … }`) om acties te groeperen.
> 4. Vervolgens wordt de update expressie `«increment-expression»` uitgevoerd.
> 5. Het programma keert terug naar stap 2.



UML Activity Diagram for the for Statement. Bron: //www.oreilly.com/library/view/javatm-how-to/9780133813036/ch05lev2sec8.html>

Het UML Activity Diagram in Java op de bovenstaande afbeelding geeft een visuele weergave voor een `for` -lus. We initialiseren eerst een variabele `counter` met als waarde `1` gevolgd door het testen van een conditie, in dit geval met de `counter` kleiner of gelijk aan `10` . Indien de conditie waar is zulle de acties binnen een block statement uitgevoerd worden, in dit geval geven we de waarde van de variabele `counter` weer in een outputscherm. Tenslotte **verhogen** ***(Eng. increment)*** we de variabele `counter` via de code `counter++` .

Het UML-schema kunnen we in JavaScript als volgt vertalen:

./js_essentials/loops/counter.js

```js
for (let counter = 1; counter <= 10; counter++) {
  console.log(`${counter}  `);
}
```

```
$ node counter.js
1
2
3
4
5
6
7
8
9
10
```

Voorbeeld:

Als voorbeeld wensen we een matrix van het wildcard-symbool (`*`) te genereren in de console. Gegeven de variabelen:

- `rows` : aantal te genereren rijen
- `cols` : aantal te genereren kolommen
- `output` : de string die we opbouwen via de `for` -lus.

./js_essentials/loops/rows_n_cols.js (00) js

```
const rows = 10;
const cols = 10;
let output = "";

for (let row = 0; row < rows; row++) {
  for (let col = 0; col < cols; col++) {
    output += "* ";
  }
  output += "\n";
}

console.log(output);
```

```
$ node rows_n_cols.js
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
```

We doorlopen de rijen en voor iedere rij doorlopen we de kolommen. Deze werkwijze is gekend als een **nested loop**. Een lus in een lus, een loop in een loop. Voor iedere kolom binnen een rij breiden we de string `output` uit met het wildcard teken en incrementeren we vervolgens de iterator `c` . Zijn alle kolommen binnen een rij doorlopen, dan voegen we een return (`\n`) toe aan de `output` string. Zijn alle rijen doorlopen, dan printen we de waarde van de `output` variabele uit in de console.

De bovenstaande code kan efficiënter geschreven worden door gebruik te maken van wiskundige operatoren `*` en `%`. Op deze manier kunnen we de nested loop vermijden.

./js_essentials/loops/rows_n_cols.js (01) js

```
const rows = 10;
const cols = 10;
let output = "";

for (let i = 0; i < rows * cols; i++) {
  output += "* ";
  if ((i + 1) % cols === 0) {
    output += "\n";
  }
}

console.log(output);
```

```
$ node rows_n_cols.js
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
```

Het totaal aantal te genereren wildcards kunnen we berekenen via het product van `rows` en `cols`. Indien de rest van de deling `(i + 1) / cols` absoluut gelijk is aan `0`, dan voegen we een return toe aan de `output` string.

./js_essentials/loops/days_of_week.js js

```
const daysOfWeek = [
  "Sunday",
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday",
];
```

```
10
11  for (let i = 0; i < daysOfWeek.length; i++) {
12    console.log(`daysOfWeek[${i}]:`, daysOfWeek[i]);
13  }
```

```
$ node days_of_week.js
daysOfWeek[0]: Sunday
daysOfWeek[1]: Monday
daysOfWeek[2]: Tuesday
daysOfWeek[3]: Wednesday
daysOfWeek[4]: Thursday
daysOfWeek[5]: Friday
daysOfWeek[6]: Saturday
```

`while`

Een `while` statement voert acties uit zolang de gespecificeerde conditie waar of `true` is. Een `while` statement ziet er als volgt uit:

```js
1  while («condition»)
2    «statement»;
```

Als de conditie waar of `true` is zal het statement uitgevoerd worden. Wordt de conditie niet waar of `false`, dan stopt de `while`-lus of loop en zal het programma na deze lus verder gaan. Het testen van de conditie gebeurt voordat het statement binnenin deze loop uitgevoerd zal worden. Net zoals bij de andere lussen gebruik je bij voorkeur een block statement (`{ … }`) om één of meerder acties of statements te groeperen.

> ⚠ **Opgelet**
>
> Vermijd een **oneindige lus** ***(Eng. infinite loop)*** in JavaScript!
> Deze zullen de webpagina blokkeren, waardoor er een slechte gebruikersbeleving ontstaat.

./js_essentials/loops/while_true.js js

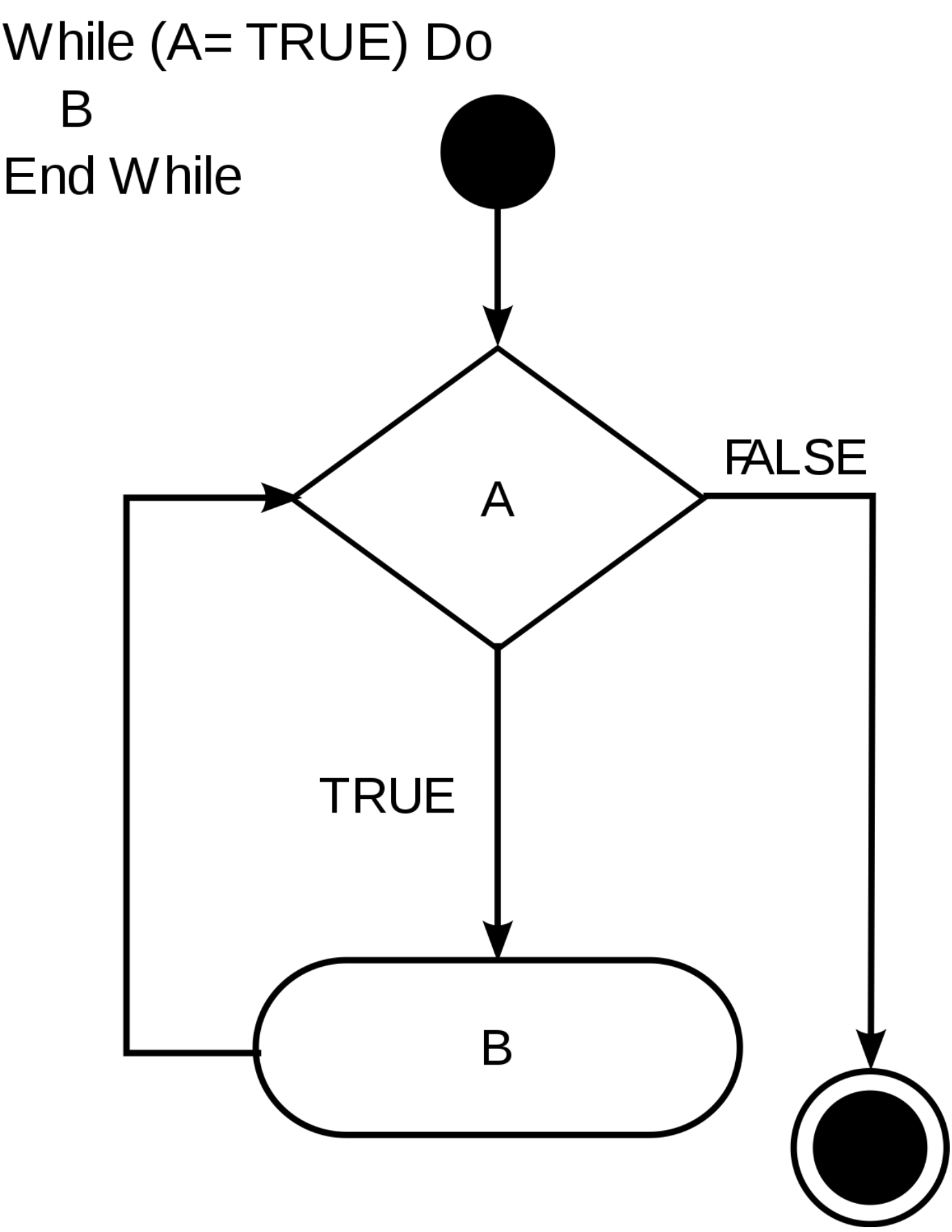
```
1  while (true) {
2    console.log(
```

```
    "You are a very naughty JavaScript programmer!",
    "Do not use while (true)!",
    "It can be OK in Python though."
  );
}
```

```
$ node while_true.js
You are a very naughty JavaScript programmer! Do not use while (true)! It can be OK in Python
You are a very naughty JavaScript programmer! Do not use while (true)! It can be OK in Python
You are a very naughty JavaScript programmer! Do not use while (true)! It can be OK in Python
You are a very naughty JavaScript programmer! Do not use while (true)! It can be OK in Python
…
You are a very naughty JavaScript programmer! Do not use while (true)! It can be OK in Python
^C
```

Tip

Stop een script met `Ctrl` + `C` (`^C`).



While loop - Wikipedia. Bron: https://en.wikipedia.org/wiki/While_loop

In het bovenstaande UML-diagram voor de `while`-lus evalueren we eerst de conditie. Zolang deze conditie `A` waar of `true` is wordt het statement `B` uitgevoerd. Wordt de conditie `false`, dan stopt de iteratie en zal het programma de instructies na deze `while`-lus uitvoeren.

Voorbeeld:

./js_essentials/loops/moves.js

```js
let isPlaying = true;
let nMoves = 0;

while (isPlaying) {
  if (999 < nMoves + 1) {
    isPlaying = false;
  } else {
    nMoves++;
  }
}

console.log("Number of moves:", nMoves);
```

```
$ node moves.js
Number of moves: 999
```

In het voorbeeld initialiseren ze twee variabelen, namelijk:

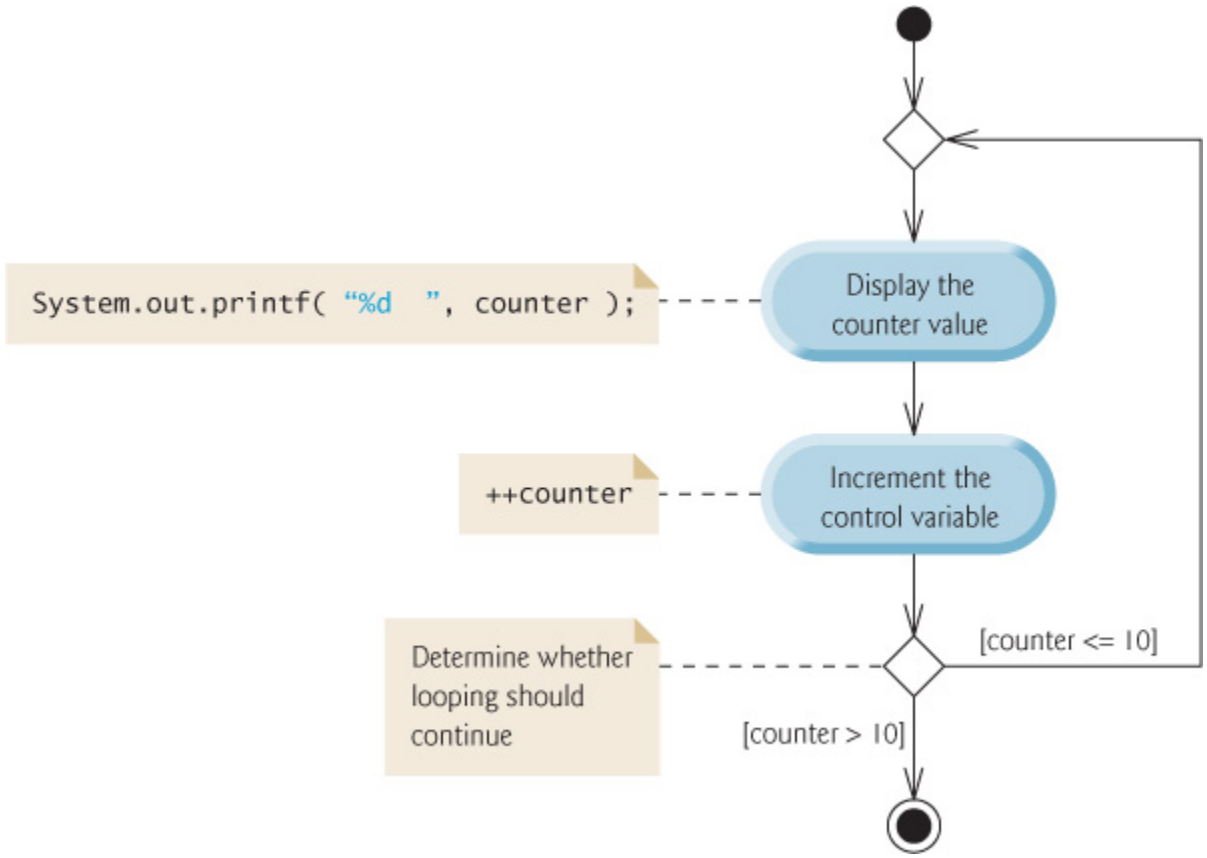
- `isPlaying`
  Status van een game bijhouden, mogelijke waarden: `true` of `false`
- `nMoves`
  Aantal gemaakte moves in een game met een maximum van `999` .

Zolang (`while`) de variabele `isPlaying` de waarde `true` bevat, dan kunnen we verder spelen. Binnen het block statement testen we de incrementatie van het aantal verplaatsingen, via de variabele `nMoves` , groter is dan `999` . Indien dit niet het geval is, kunnen we verder spelen en maken we een verplaatsing. Indien de conditie in het `if` -statement `true` is, kennen we de waarde `false` to aan de variabele `isPlaying` . Bij de volgende lus wordt de conditie `false` in het `while` -statement. Het programma voert vervolgens de instructies uit na het `while` -statement.

do…while

De `do…while` -lus is gelijkaardig met de `while` -lus, maar voert de acties of instructies binnen het block statement minstens eenmaal uit. Als vervolgens de conditie de waarde `true` bevat in het `while` statement, zullen de acties

terug uitgevoerd worden. Is de conditie `false`, de zal het programma de `do...while`-lus verlaten en verder de volgende instructies uitvoeren.



UML Activity Diagram for the do…while Repetition Statement. Bron: //www.oreilly.com/library/view/javatm-how-to/9780133813036/ch05lev2sec8.html>

Het UML-schema kunnen we in JavaScript als volgt vertalen:

./js_essentials/loops/do_while.js

```js
let counter = 0;

do {
  console.log(counter);
  counter++;
} while (counter <= 10);
```

```
$ node do_while.js
0
1
2
3
4
5
6
```

```
7
8
9
10
```

`for…in`

Een `for…in` statement itereert een gespecificeerde variabele over alle **telbare eigenschappen** *(Eng. enumerable properties)* van een object. Voor elke eigenschap worden de acties uitgevoerd binnen het block statement. Een `for…in` statement ziet er als volgt uit:

```js
for («variable» in «object»)
  «statement»;
```

Voorbeeld:

./js_essentials/loops/for_in.js

```js
let person = {
  firstName: "Jane",
  surName: "Doe",
  sex: "Female",
};

for (let prop in person) {
  console.log(`The property '${prop}' has the value '${person[prop]}'.`);
}
```

```
$ node for_in.js
The property 'firstName' has the value 'Jane'.
The property 'surName' has the value 'Doe'.
The property 'sex' has the value 'Female'.
```

In het voorbeeld definiëren we een Object via de **literal notation**. Dit object bevat 3 eigenschappen, namelijk: `firstName`, `surName` en `sex`. Aan elke eigenschap wordt er een waarde toegekend. Om vervolgens alle aanwezige eigenschappen uit dit object op te lijsten, kunnen we best gebruik maken van een `for…in`-lus. De

variabele `prop` bevat de naam van de eigenschap. Om de waarde van deze eigenschap op te vragen gebruiken we de associatieve array van dit object via `person[prop]` .

./js_essentials/loops/days_of_week_for_in.js

```js
let daysOfWeek = [
  "Sunday",
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday",
];
daysOfWeek.info = "Represents the days of the week";

for (const prop in daysOfWeek) {
  console.log(prop);
}
```

```
$ node days_of_week_for_in.js
0
1
2
3
4
5
6
info
```

Het `for…in` statement kan gebruikt worden om alle elementen binnen de array op te vragen, maar is daar eigenlijk niet de beste oplossing voor. Naast deze elementen worden ook alle gedefinieerde eigenschappen opgelijst. Bij voorkeur gebruiken we de `for` of `for…of` -lus of de `Array.forEach` methode.

for…of

Het `for…of` statement creëert een lus die itereert over iterabele objects (zoals: `Array` , `Map` , `Set` …) en roept vervolgens een variabele aan met een waarde die kan verschillen bij iedere iteratie.

Een `for…of` statement ziet er als volgt uit:

```js
for («variable» of «object»)
  «statement»;
```

Voorbeeld:

./js_essentials/loops/days_of_week_for_of.js

```js
let daysOfWeek = [
  "Sunday",
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday",
];
daysOfWeek.info = "Represents the days of the week";

for (const element of daysOfWeek) {
  console.log(element);
}

for (const prop in daysOfWeek) {
  console.log(prop);
}
```

```
$ node days_of_week_for_of.js
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
0
1
2
3
4
5
6
info
```

In het voorbeeld definiëren we een array via de **literal notation**. Deze array bevat zeven elementen, namelijk: `"Sunday"` … `"Saturday"` . Om vervolgens de waarden van alle elementen op te vragen kunnen we gebruik maken van een `for…of` -lus.

`break` en `continue`

`break` - en `continue` -statements hebben invloed op lussen. Een `continue` -statement beëindigt de executie van de huidige iteratie in een lus. Een `break` -statement beëindigt of stopt de lus. `break` wordt ook gebruikt in `switch…case` en dit om een match te beëindigen en vervolgens de `switch` te verlaten.

`break`

./js_essentials/loops/break.js

```js
for (let i = 1; i <= 8; i++) {
  if (i == 6) break;
  console.log(i);
}
```

```
$ node break.js
1
2
3
4
5
```

Met het `break` -statement kunnen we een lus stoppen wanneer een bepaalde conditie `true` is. De lus wordt hierdoor verlaten en het programma gaat verder na deze `for` de volgende instructies uitvoeren. Het `break` -statement wordt vooral toegepast in `switch…case` -statements.

`continue`

./js_essentials/loops/continue.js

```js
```

```
for (let i = 1; i <= 8; i++) {
  if (i % 2 == 1) continue;
  console.log(i);
}
```

```
$ node continue.js
2
4
6
8
```

In het voorbeeld willen we enkel de even getallen schrijven naar het outputscherm. Bij iedere iteratie gaan we na of de variabele `i` **oneven** ***(Eng. odd)*** is.

> ⚠ **Opgelet**
>
> Vermijd het gebruik van `break` en `continue` in geneste lussen – een lus in een lus. Dit schept verwarring voor andere programmeurs.

Labled statement

Maar het is wel degelijk mogelijk! In dit geval hebben deze statements in de lus die hen omsluit. Wil je lussen aanroepen op een ander niveau, dan moet je werken met **gelabelde statements** ***(Eng. labled statements).***

Een label geeft een naam aan een bepaald statement, dat o.a. een `while`-lus kan zijn. Op een andere plek in de code kunnen we refereren naar dit statement via `continue` of `break` keyword gevolgd door het label.

./js_essentials/loops/labled_statement.js

```
even: for (let i = 1; i <= 10; i++) {
  if (i % 2 == 1) continue even;
  console.log(i);
}
```

```
$ node labled_statement.js
2
```

```
4
6
8
10
```

← Control Flow

Functies →

Laatst bijgewerkt: 25/9/2022 17:19:32