# Literatuur

## 0. Benodigdheden

### TypeScript

TypeScript (ook wel *TS*) is een scripttaal (programmeertaal) die op de taal JavaScript (ook wel *JS*) is gebouwd. Het wordt gebruikt om allerlei soorten programma's te maken zoals web-, mobile-, desktop- en serverapplicaties.

TypeScript is *backwards-compatible* met JavaScript, wat betekent dat alles wat je in JavaScript schrijft je ook in TypeScript kan gebruiken. Daarentegen wordt TS steeds veel gebruikt aangezien deze taal features bevat die het schrijven van schonere code en grotere applicaties overzichtelijker maakt.

TypeScript bestanden eindigen op `.ts` en worden, zodra je programma gedraaid wordt, gecompileerd naar `.js` bestanden die door een webbrowser gedraaid kunnen worden.

#### Handige weetjes

Regeleindes worden aangegeven met een `;` (puntkomma).

Als je je programma draait, kan je tekst tonen door middel van `console.log("Hier komt je tekst.");`

### Node.JS

Voor deze workshop wordt het programma Node.JS gebruikt om JavaScript (en daarmee TypeScript) bestanden te kunnen draaien zonder een browser. Ga naar deze webpagina om Node.JS te installeren.

### npm

Om externe packages/libraries met code te installeren en te gebruiken in onze eigen projecten, gebruiken we npm (Node Package Manager). Dit wordt automatisch door Node.JS geinstalleerd.

Om een package in je project te installeren, gebruik je `npm install <packagename>`. Als je bijvoorbeeld de library `react` will installeren, typ je `npm install react`.

Voordat je een Node.JS programma voor de eerste keer draait, moet je in je terminal `npm install` uitvoeren. Hiermee worden alle benodigde packages voor je geinstalleerd, en ben je klaar het programma te draaien.

## 1. Variabelen

### Definitie

In een programma wil je vaak waarden opslaan en later opnieuw kunnen gebruiken. Hiervoor gebruiken we variabelen. Dit zijn verwijzingen naar waarden in het geheugen van de computer.

Variabelen bestaan uit twee onderdelen: het type en de waarde. Een variabele `name` heeft bijvoorbeeld het type `string` (een stuk tekst) en de waarde `John`.

## Types

Elke variabele heeft een type. Dit geeft aan welke waarden er aan de variabele toegewezen kunnen worden. Als je een variabele declareert met het type `number`, kan je waarden met dat type eraan toewijzen (in dit geval, alleen getallen; `1`, `235.92`, etc.).

De meeste types hebben zelf ook waarden en functies.

## Gebruik

In TypeScript (en JavaScript) kan je variabelen op meerdere manieren aanmaken; `var`, `let` en `const`. In deze workshop gebruiken we `let`.

Laten we een variabele aanmaken om een getal op te slaan.

```
let age: number = 21;
```

Je kan dit in het Engels lezen als "**Let** the variable **age** be a **number** with the value **21**".

1. `let` - **Keyword** Dit vertelt TypeScript dat we een **nieuwe** variabele declareren.
2. `age` - **Naam** De naam van de variabele.
3. `: number` - **Type (optioneel)** Het type van de variabele. Dit kan in dit geval weggelaten worden, aangezien TypeScript aan de waarde die aan `age` wordt gegeven (`21`) kan zien wat het type van de variabele is.
4. `=` - **Toewijzing** In het Engels "assignment". Hiermee geef je aan wat de waarde van de variabele moet zijn.
5. `21` - **Waarde / Expression** De waarde die aan de variabele gegeven wordt. Dit kan een directe waarde zijn (`200`, `'Hello'`, `true`, etc.) of een andere variabele, in welk geval de waarde hiervan wordt gekopieerd.
6. `;` - **Regeleinde (optioneel)** De puntkomma geeft aan dat dat het einde van de regel code is. Hoewel dit teken weggelaten kan worden, zullen we hem in deze workshop wel gebruiken gezien het feit dat deze in veel programmeertalen verreist is en in veel grote TypeScript projecten wel gebruikt wordt.

## 'any' en 'undefined'

Variabelen kunnen ook gedeclareerd worden zonder een waarde toe te wijzen. Om aan te geven dat een variabele geen waarde heeft, wordt `undefined` gebruikt. Je kan dit zien als een waarde die aangeeft dat een variabele 'leeg' is. `undefined` kan aan elke variabele toegewezen worden, ongeacht het type van de variabele.

Zoals eerder is aangegeven, kan je ook variabelen declareren zonder een type te geven. Als je een variabele aanmaakt zonder een type *en* waarde aan te geven (zoals `let firstName;`), krijgt het automatisch het type `any` toegewezen. Dit type zorgt ervoor dat de variabele elk soort waarde kan opslaan. Dit wordt in de meeste gevallen gezien als slechte code en zorgt dat er gemakkelijker fouten in de code kunnen ontstaan. **Het wordt sterk aangeraden altijd een type en/of waarde te geven bij de declaratie van variabelen.**

Later in het programma kan de variabele een nieuwe waarde krijgen (mits het type van die waarde overeenkomt met het type dat is aangegeven bij de declaratie van de variabele).

## Voorbeeld

Hier een voorbeeld van verschillende variabelen.

*(Let op: in TypeScript kan je opmerkingen toevoegen met twee slashes (`//`). Alles wat na die tekens komt, wordt genegeerd door TypeScript. Opmerkingen (comments) worden vaak gebruikt voor het uitleggen wat een regel code doet, of om notities achter te laten voor jezelf of andere programmeurs.*

```
let firstName = 'Johan'; // string, 'Johan'
let lastName = 'van Dam'; // string, 'van Dam'

let fullName = firstName + ' ' + lastName; // string, 'Johan van Dam'

let nameLength = fullName.length; // number, 13

let age = 36; // number, 36
let tenYearsOlder = age + 10; // number, 46

age = 'Pizza'; // FOUT.
// Een string kan niet toegewezen worden aan een variabele met type number.

let counter = 0; // number, 0
counter = counter + 1; // counter heeft nu de waarde 1.

let copyOfCounter = counter; // number, 1

let buttonIsPressed: boolean; // boolean, null
buttonIsPressed = false; // boolean, false
buttonIsPressed = true; // boolean, true

let variableWithAnyType; // any, undefined
variableWithAnyType = 1; // any, 1
variableWithAnyType = 'Hello'; // any, Hello
variableWithAnyType = true; // any, true
```

# 2. Functies

## Definitie

Functies zijn stukjes code/processen. Op dezelfde manier dat je een waarde kan opslaan met variabelen, kan je met functies een blok code definieren en later aanroepen. Er gaat iets in, er komt iets uit.

Functies kunnen parameters (*arguments*) aannemen om berekeningen uit te voeren. Ook kunnen functies een waarde teruggeven, bijvoorbeeld om het resultaat van een som aan een variabele toe te wijzen.

## Gebruik

In TypeScript (en JavaScript) kan je functies definieren met het woord `function`.

Laten we een functie maken die twee getallen bij elkaar optelt.

```
function sum(first: number, second: number): number {
    return first + second;
}
```

We kunnen later de functie op deze manier aanroepen.

```
let x = sum(10, 5); // number, 15
```

De functie genaamd `sum` neemt twee parameters aan, `first` met het type `number` en `second` met het type `number`, en geeft een `number` terug. In de inhoud (ookwel *body*) van de functie worden `first` en `second` bij elkaar opgeteld en teruggeven (met `return`) aan hetgene wat de functie aanroept.

1. `function` - **Keyword** Dit vertelt TypeScript dat we een **nieuwe** functie declareren.
2. `sum` - **Naam** De naam van de functie.
3. `(first: number, second: number)` - **Parameters (arguments)** De waarden of variabelen die de functie nodig heeft om te werken. In dit geval zijn er twee arguments, `first` en `second`, maar een functie kan ook meer dan twee of helemaal geen arguments hebben. Als je geen arguments nodig hebt, dan hou je dit gedeelte leeg en schrijf je simpelweg `()`.
4. `: number` - **(Return)type (optioneel)** Het type dat de functie teruggeeft aan hetgene wat hem aangeroepen heeft. Dit kan een normaal type zijn (`number`, `string`, etc.), maar als de functie niets teruggeeft, dan kan je hier `void` neerzetten.

   Het type kan in dit geval weggelaten worden, aangezien TypeScript aan de body van de functie kan zien wat er terug wordt gegeven; de expressie `first + second` resulteert in een nieuw getal, en dus weet TypeScript dat deze functie een `number` teruggeeft. Je kan ook functies maken die niets teruggeven.
5. `{ ... }` - **Body** Dit opent en sluit de body (inhoud) van de functie. Alles wat tussen de `{` en `}` geschreven wordt, behoort tot de functie en wordt uitgevoerd wanneer de functie aangeroepen wordt.

6. `return ...;` - **Return statement** Dit geeft de waarde die na het woord `return` komt (in dit geval `first + second`) terug aan hetgene wat de functie aanroept. In het voorbeeld wordt de variabele `x` aangemaakt en wordt, na het aanroepen van de functie `sum(10, 5)` de waarde teruggegeven aan `x`.

## Voorbeeld

```
function greet(name: string): void {
    // console.log is een functie die de meegegeven string op het scherm toont.
    console.log('Hallo ' + name + '!');
}

greet('Andreas'); // Toont 'Hallo Andreas!' op het scherm.
greet('Simon'); // Toont 'Hallo Simon!' op het scherm.
greet(); // FOUT. De parameter name is niet meegegeven.

let name = 'Cassandra'; // string, 'Cassandra'
let upperCaseName = name.toUpperCase(); // string, 'CASSANDRA'
```

# 3. If-else

## Definitie

Het komt vaak voor dat, tijdens het draaien van je programma, je keuzes wilt maken op basis van variabelen. Als je bijvoorbeeld de gebruiker wilt vertellen dat een knop is ingedrukt, moet je eerst weten of de knop ingedrukt is of niet.

Om dit te doen maken we gebruik van `if` en `else` statements. Je kan het zien als een stukje code dat zegt "**als** dit gebeurt, doe dan dit, **en anders** dat".

## Gebruik

In TypeScript (en JavaScript) kan je met `if` en `else` statements, afhankelijk van een waarde, een stukje code - de body - uit laten voeren.

Om te checken of een variabele een bepaalde waarde heeft, kan je gebruik maken van drie opeenvolgende gelijkheidstekens (`===`). In de meeste programmeertalen hoef je maar twee gelijkheidstekens te gebruiken (`==`) en ook TypeScript ondersteunt dit, maar het gebruik hiervan wordt niet aangeraden.

Het vergelijken van variabelen met `===` resulteert in een `boolean` waarde - d.w.z. `true` of `false` - en op basis van die waarde wordt de body wel of niet uitgevoerd.

Naast de `if` kan er ook het woord `else` gebruikt worden. Dit geeft aan wat er moet gebeuren als de conditie (de vergelijking in de `if`) resulteert in `false`.

## Gebruik

In het volgende voorbeeld wordt alleen `'Welkom!'` op het scherm getoond **als** de naam van de gebruiker (`name`) de waarde `Jeroen` heeft.

```
let name = 'Jeroen'; // string, 'Jeroen'

if (name === 'Jeroen') { // true. Dit blok wordt uitgevoerd.
    console.log('Welkom!');
}

// Toont 'Welkom!' op het scherm.
```

> "**Als** `name` gelijk is aan 'Jeroen', voer dan `console.log('Welkom!')` uit."

```
let buttonIsPressed = false; // boolean, false

if (buttonIsPressed === true) { // false
    console.log('De knop is ingedrukt.');
} else { // Dit blok wordt uitgevoerd.
    console.log('De knop is NIET ingedrukt.');
}

// Toont 'De knop is NIET ingedrukt.' op het scherm.
```

> "**Als** `buttonIsPressed` gelijk is aan `true`, voer dan `console.log('De knop is ingedrukt.')` uit. **Zo niet,** voer dan `console.log('De knop is NIET ingedrukt.');` uit."

In het geval dat de waarde die gecheckt wordt al een `boolean` is, kan het `=== true` gedeelte overgeslagen worden.

```
let buttonIsPressed = false; // boolean, false

if (buttonIsPressed) { // De waarde is al een boolean, false
    console.log('De knop is ingedrukt.');
} else { // Dit blok wordt uitgevoerd.
    console.log('De knop is NIET ingedrukt.');
}

// Toont 'De knop is NIET ingedrukt.' op het scherm.
```

Getallen kunnen ook worden vergeleken met andere getallen met de tekens is-gelijk-aan (`===`), kleiner-dan (`<`), kleiner-dan-of-gelijk-aan (`<=`), groter-dan (`>`) of groter-dan-of-gelijk-aan (`>=`).

```
let length = 170; // number, 170

if (length < 175) { // true. Dit blok wordt uitgevoerd.
    console.log('Je bent kleiner dan 175 cm.')
} else {
    console.log('Je bent groter dan 175 cm.')
}
```

`if` en `else` statements kunnen ook achtereen worden gezet voor complexere condities.

```
if (length < 170) {
    console.log('Je bent minder lang dan gemiddeld.');
} else if (length >= 170) {
    console.log('Je bent langer dan gemiddeld.')
}
```

Condities kunnen gecombineerd worden om op meerdere factoren tegelijk te checken.

Om te checken of A **en** B waar zijn, gebruikt men tweemaal het en-teken (`&&`).

```
if (computerIsOn && userIsLoggedIn) {
    console.log('Welkom bij Windows.');
}
```

Om te checken of A **of** B waar zijn, gebruikt men tweemaal het sluisteken/verticaal streepje (`||`).

```
if (frontDoorbellIsPressed || backDoorbellIsPressed) {
    console.log('Dingdong!');
}
```

# 4. Async

## Definitie

Stel je voor, je hebt een robot. Een hele domme robot. Deze robot kan je een velletje papier met instructies geven die achter elkaar worden uitgevoerd. We moeten hem heel specifiek vertellen wat hij moet doen.

In dit scenario staat de robot voor Node.JS en de papieren instructies voor onze TypeScript code. Node.JS voert onze code regel voor regel uit maar kan wel duizenden van dit soort operaties per seconden uitvoeren. Daar staat wel tegenover dat Node.JS, net zoals onze robot, maar één ding tegelijk kan doen.

Stel we geven de robot instructies om koffie te zetten en daarna om dozen te stapelen. De robot zal eerst naar het koffiezetapparaat lopen en het aanzetten, een kopje eronder zetten en dan op de knop drukken. Dan wordt de koffie gemaakt, maar dit duurt lang. De robot wacht wel een volle minuut tot de koffie klaar is, brengt het naar ons toe en pas daarna gaat hij dozen stapelen. In dit geval heeft de robot veel tijd verspild aan het wachten op de koffie. Zou het niet veel efficienter zijn om, tijdens dat de koffie gezet wordt, alvast dozen te gaan stapelen. Zodra het apparaat eenmaal klaar is, kan de robot een seintje krijgen en de koffie brengen. Daarna kan hij weer door met stapelen.

Zo'n zelfde soort situatie is wat vaak voor kan komen in apps. Af en toe moeten er lange berekeningen gedaan worden of wordt er data opgehaald van het internet. Dit kan allemaal relatief lang duren. Als hier niets aan gedaan zou zijn, kan het voorkomen dat je app spontaan vastloopt elke keer als je een stukje data op moet halen. Je programma is aan het wachten tot die data terug komt, en kan in de tussentijd niet het scherm verversen.

Om dit te voorkomen kunnen we gebruik maken van *asynchrone functies*. Dit zijn functies die erg lijken op normale functies, maar waarvan het resultaat niet direct bekend is (bijvoorbeeld bij het sturen van een berichtje of het verwerken van grote berekeningen). Deze *async* functies zorgen ervoor dat we een process kunnen laten draaien "in de achtergrond" en niet hoeven te wachten op het resultaat. We kunnen vervolgens TypeScript vertellen wat er moet gebeuren zodra het resultaat eenmaal is aangekomen.

## Gebruik

In deze workshop gebruiken we de Star Wars API. We kunnen met de `swapi` variabele via het internet informatie over acteurs en films van Star Wars opvragen. Aangezien deze informatie niet direct beschikbaar is, zijn alle functies async.

Beide functies van de `swapi`, `getPerson(id)` en `getFilm(id)`, geven ons de waarde `Promise` terug. We kunnen hiermee de functie `.then` aanroepen en het een functie geven die uitgevoerd moet worden wanneer de informatie is opgehaald. Dit heet een *callback* functie.

We kunnen bijvoorbeeld informatie over de acteur met het id `1` opvragen. Dat doen we als volgt:

```
import { swapi } from './backend/swapi';

swapi.getPerson(1);
```

Dit geeft ons een `Promise<Person>`. Dat betekent dat we niet gelijk een waarde van `Person` hebben, maar ons wordt **beloofd** dat we later een `Person` krijgen.

We kunnen een `Promise` een callback functie geven met `.then(callback)`. Dit is een functie die zelf een functie inneemt als parameter en deze uitvoert zodra de informatie van de `Promise` is opgehaald.

Laten we TypeScript vertellen wat er moet gebeuren als de informatie is opgehaald. Laten we alle informatie over de persoon op het scherm tonen.

```
function printPerson(person: Person): void {
    console.log(person);
}

swapi.getPerson(1).then(printPerson);
```

De onderste regel kunnen we in het Engels lezen als "**Get** this person and **then print** the information."

We kunnen ook in plaats van van te voren een functie te definiëren ook rechtstreeks binnen de `.then(...)` een functie meegeven. Het volgende stukje code doet precies hetzelfde als hetgene hiervoor.

```
swapi.getPerson(1).then((person) => console.log(person));
```

De functie `(person) => console.log(person)` is een functie net zoals de `printPerson` van net, maar hij wordt ter plekke aangemaakt, in plaats van dat we hem van tevoren definiëren als `printPerson`. Het enige verschil is het pijltje (`=>`) dat we nodig hebben.

We kunnen ook een langere functie meegeven door na het pijltje de haakjes toe te voegen net zoals bij normale functies.

```
swapi.getPerson(1).then((person) => {
    console.log(person);
    // Hier kunnen meer regels staan!
});
```

Aangezien we binnen de callback functie volledige toegang hebben tot het `person` object, kunnen we alles erover opzoeken. Laten we alleen de naam van de acteur op het scherm tonen.

```
swapi.getPerson(1).then((person) => console.log(person.name));

// of

swapi.getPerson(1).then((person) => {
    console.log(person.name);
});
```