



Belajar TypeScript

Anggie Bratadinata

www.masputih.com | [@abratadinata](https://twitter.com/abratadinata)

Daftar Isi

1. Introduksi	1
1.1. Hello TS	2
1.2. Variabel & Tipe Data	3
1.3. Custom Type	7
1.4. Union Type	8
1.5. Type Assertion	8
1.6. Let atau Var ?	8
1.7. Function	9
1.8. Namespace	12
2. Interface & Class	16
2.1. Abstraksi	18
2.2. Turunan	19
2.3. Generics	23
2.4. Variabel & Function Statis	28
3. Modul	31
3.1. Export & Import	32
3.2. Ekspor Lebih Dari Satu Kelas	39
3.3. Ekspor Function	40
4. Contoh Aplikasi Kalkulator	42
4.1. Optimasi	49
4.2. Minifikasi	52
5. Debugging	54
5.1. Source Maps	54
5.2. Minifikasi	57
6. Library Pihak Ketiga	59
6.1. Ambient Declaration	59
6.2. Type Definition File	59
7. Penutup	66
7.1. Dari Sini Terus ke Mana?	66

Bab 1. Introduksi

TypeScript adalah bahasa pemrograman berbasis JavaScript yang menambahkan fitur *strong-typing* & konsep pemrograman OOP klasik (*class*, *interface*). Di dalam dokumentasinya, TypeScript disebut sebagai *super-set* dari JavaScript, artinya semua kode JavaScript adalah kode TypeScript juga. Kompiler TypeScript menterjemahkan (*transpile*) sintaks TypeScript ke dalam JavaScript standar kayak yang sudah kita kenal.

Tentunya untuk sintaks/konsep OOP belum didukung di JavaScript hanya dipakai oleh *TypeScript Compiler* (TSC) untuk memverifikasi kode TypeScript yang kita tulis & nggak ada di file JavaScript hasil kompilasi. Bukan berarti konsep ini nggak berguna, justru sebaliknya adanya fitur ini membuat kita bisa menulis aplikasi yang kompleks dengan relatif lebih mudah tanpa perlu pusing mikirin dukungan browser (hasilnya toh tetap JavaScript).

Dengan seting default, kode JavaScript hasil proses kompilasi adalah kode standard yang bisa dijalankan di semua browser modern yang mendukung ECMAScript 5 (JavaScript 1.5). Kalo kita lagi sial dan harus mendukung browser jadul yang hanya support ECMAScript 3.0 (JavaScript 1.3), misalnya Internet Explorer 8, kita bisa atur *compiler* supaya hanya *generate* kode yang kompatibel dengan JS1.3.

Jadi kayak yang bisa kita baca di website nya,

Mulai dengan JavaScript, diakhiri dengan JavaScript.

Untuk instalasi TSC kita butuh Node. Jadi silakan install Node kalo belum ada di komputer Anda, terus instal TSC pake NPM.

```
npm install -g typescript
```

Selanjutnya, pastikan `tsc` bisa dijalankan di terminal. Kalo perintah `tsc` nggak ditemukan, periksa lagi apa TSC udah masuk di `PATH`.

```
tsc -v
```

TSC yang saya pakai adalah versi 1.8.10 jadi output perintah di atas adalah `Version 1.8.10`.



Sebelum Anda mulai belajar TypeScript, sangat saya sarankan Anda benar-benar paham JavaScript karena lebih mudah belajar TypeScript sambil lihat file JavaScript hasil kompilasinya.

Editor/IDE yang udah support TypeScript antara lain:

- IntelliJ IDEA
- SublimeText (pake plugin)
- Atom (pake plugin)
- Visual Studio Code
- Visual Studio



Saya pake [Atom](#) + plugin [Atom-TypeScript](#). Tapi sampe Bab 2 kita nggak butuh editor, cukup pake [TypeScript Playground](#).

1.1. Hello TS

Kita bikin file bernama `hello.ts` yang isinya begini:

```
console.log('Hello typescript')
```

Buka terminal, terus jalanin perintah

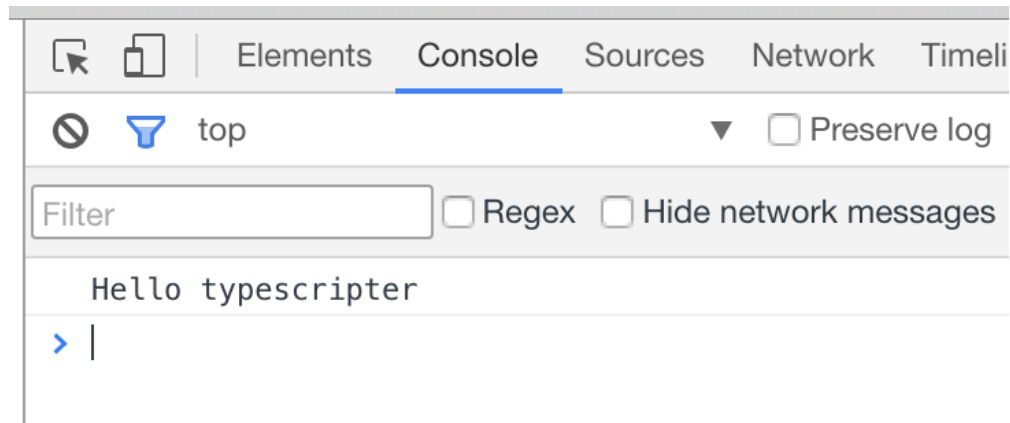
```
tsc hello.ts
```

Compiler akan meng-*compile* file `hello.ts` jadi file `hello.js`. Selanjutnya, kita bikin file `hello.html` yang isinya:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <script src="hello.js"></script>
  </head>
  <body>

  </body>
</html>
```

Buka file di browser & liat `console`.



1.2. Variabel & Tipe Data

Pertama kita kenalan dulu dengan tipe data dasar dan cara bikin variabel.

1.2.1. Deklarasi vs Definisi

Kalo Anda belum paham apa bedanya **deklarasi** & **definisi**. Gampangnya begini:

Deklarasi artinya kita bikin variabel tanpa nilai. Kalo function hanya nama & parameter, tanpa isi (*body*). **Definisi** artinya variabel kita kasih nilai. Function kita kasih kode *body*-nya.

```

let a:number; // deklarasi
a = 1;         // definisi

let b:string = 'hello'; // definisi

add(a:number,b:number):number; // deklarasi
add(a:number,b:number):number{ // definisi
    return a + b;
}

```

1.2.2. Any

Any artinya sebarang tipe. Variabel dengan tipe data ini bisa diberi sebarang nilai. Sama dengan variabel di JavaScript standar.

```

let a:any = 0;
a = 'hello';
a = [1,2];

```

Semua variabel yg dideklarasikan tanpa tipe data otomatis bertipe *any* & disebut *implicit any* (lawannya *explicit any*). Tapi lebih baik kita pake *any* secara eksplisit.

```

let b = 1; // implicit
let b:any = 1

```

1.2.3. Boolean

Tipe **boolean** (pake huruf **b** kecil).

```

let finished:boolean = false;
finished = 'no'; // error

```

1.2.4. Number

Tipe data **number** dipake nggak hanya untuk bilangan desimal/integer, tapi juga dipake untuk bilangan hexadesimal (**0x**), oktal (**0o**) dan bilangan biner (**0b**);

```
let start:number = 1.2;
let hexColor:number = 0xFFFF; //0x = hexa
let oct:number = 0o755; //0o = octal
let binary:number = 0b101; //0b = binary
```

1.2.5. String

Tipe `string` dipake untuk data berbentuk teks.

```
let color:string = 'red';
```

Template String

Typescript juga menyediakan *template string* yang dipake untuk substitusi teks. String yang pake template dibuka-tutup pake *backtick* (```) bukan *single-quote* (`'`) atau *quote* (`"`)

```
let name:string = 'Jon Snow';
let age:number = 35;

let info:string = `Nama saya ${name}. Tahun depan usia saya ${age
+1} tahun`;
```

1.2.6. Enum

Enum kita pake untuk mendefinisikan data yang berisi sekumpulan konstanta numerik.

```
enum Switch{
    ON, OFF
}

console.log(Switch.ON); // 0
console.log(Switch.OFF); // 1
```

1.2.7. Array

Untuk bikin variabel bertipe `array` kita harus mendeklarasikan tipe data setiap element di dalamnya. Array semacam ini disebut juga *typed-array* & dalam bahasa lain kayak ActionScript 3.0 atau C++ dikenal dengan nama *Vector*.

```
let a:number[] = [100,200]; // array numerik
let b:any[] = [1,2,'a','b']; // array campuran
```

Kita juga pake sintaks *Generic Array*.

```
let a:Array<number> = [100,200];
let b:Array<any> = [1,2,'a','b'];
```



Generics kita bahas lebih lanjut di bagian lain.

Array Iterator

Untuk memproses Array, TypeScript nyediain fitur namanya *Iterator*. Jadi untuk memproses setiap elemen atau index dari array kita bisa pake `for .. of`

```
let a:number[] = [1,2,3,4,5];
for(let n of a){
  console.log(n);
}
```



Iterator sebenarnya bisa dipake untuk tipe selain Array tapi hanya kalo kita pilih ECMAScript 6 untuk output TSC

1.2.8. Tuple

Tuple adalah array yang jumlah awal elemennya *fixed*. Tipe data setiap elemennya boleh berbeda. Nilai awal *tuple* harus sesuai dengan urutan yang ditentukan waktu kita deklarasi variabelnya.

```
//bikin tuple 2 elemen, yg pertama string, kedua numerik
let tup:[string,number];
tup = ['xx',100]; //OK
tup = [1000,'xxx']; // error, yg pertama harus string
```

Elemen tuple bisa ditambah/dikurangi kayak array biasa. Data yang bisa dimasukin ke sebuah Tuple hanya data yang sama dengan tipe yang kita tentukan di deklarasinya.


```
let tup:[string,number] = ['a',1];
tup.push(100); // OK
tup.push('xx'); // OK

let tup2:[string,string] = ['a','b'];
tup2.push(100); //error
```

1.3. Custom Type

Kita bisa bikin tipe data kustom secara *inline* kayak begini:

```
let mycar:{ brand:string, color:string };
mycar = { brand:'toyota',color:'white'}
```

Cara *inline* kayak di atas hanya bisa dipake untuk satu variabel. Cara yang lebih baik adalah pake *interface*.

Listing 1. Interface

```
interface Car{
  brand:string;
  color:string;
}

let myCar:Car = {
  brand:'toyota',color:'white'
}

let yourCar:Car = {
  brand:'mazda',color:'red'
}

let theirCar:Car = {
  brand:'audi',
  year:2001 // error, atribut 'year' nggak ada di tipe Car
}
```



Untuk tipe data kustom yang lebih kompleks, kita bahas nanti di bagian Class & Interface.

1.4. Union Type

Union-type kita pake untuk deklarasi variabel untuk tipe data lebih dari satu (mirip **any** tapi terbatas).

```
let a:string|number; // a bisa string atau number

a = 100; // OK
a = 'xx'; // OK

a = false; // error, a nggak bisa boolean
```

1.5. Type Assertion

Type Assertion adalah cara untuk memproses data sesuai tipenya. Dalam bahasa lain disebut *Type Cast*.

```
let a:any = 1;

console.log( (a as number).toFixed(2) );
```

1.6. Let atau Var ?

ECMAScript 2015 (dikenal juga dengan nama ECMAScript 6/ES6) memperkenalkan cara baru untuk deklarasi variabel dengan kata kunci **let**. Apa bedanya dengan **var**? Beda *scope*. *Scope* dari **var** adalah *function* di mana variabel dideklarasikan. **let** bersifat *block-scoped*, hanya berlaku di dalam blok di mana variabel kita deklarasikan.

```
function testVar(){
  var a=1;
  var n=0;
  while(n < 5){
    var a = 10 * n;
    console.log('[var:loop] a',a); // 0 - 40
    n++;
  }

  console.log('[var:fn]',a); // 40
}

testVar();
```

Dalam contoh di atas, nilai `a` di akhir function `testVar()` nilainya sama dengan nilai `a` di dalam loop. Kita lihat bedanya dengan `let`.

```
function testLet(){
  var a=1;
  var n=0;
  while(n < 5){
    let a = n * 10;
    console.log('[let::loop] a',a); // 0 - 40
    n++;
  }

  console.log('[let:fn]',a); // 1
}

testLet();
```

Kita liat nilai `a` di akhir function sama dengan nilai `a` di awal function karena `a` yang di dalam loop *scope*-nya beda.

1.7. Function

Kita bisa tentukan tipe data setiap parameter sebuah function & nilai baliknya (`return`).

```
function mul(a:number,b:number):number{
  return a * b;
}
console.log( mul(2,3) ); // 6
console.log( mul(2) ); //error, function mul() butuh 2 argumen
```

Parameter yang opsional kita deklarasikan pake tanda tanya.

```
//parameter b opsional
function mul(a:number,b?:number):number{
  b = b || 1; // kalo b nggak ada, anggap nilainya 1
  return a * b;
}

console.log(mul(2)); // 2
console.log(mul(3,4)); // 12
```

Parameter opsional function di atas bisa kita ubah jadi parameter dengan *default value* kayak di bawah.

```
//parameter b opsional & nilai defaultnya = 1
function mul(a:number,b:number = 1):number{
  return a * b;
}

console.log(mul(2)); // 2
console.log(mul(3,4)); // 12
```

Function tanpa nilai balik (*return*), kita kasih *void*.

```
function log(msg:any):void{
  console.log( '[LOG]',msg);
}
```

1.7.1. Lambda

Lambda adalah function di mana *scope* dari kata kunci *this* selalu mengacu pada objek di mana function dideklarasikan, bukan objek yang jalanin function itu.

```

let label:any = document.createElement('p');
document.body.appendChild(label);

let btn:any = document.createElement('button');
btn.innerHTML = 'CLICK ME';
document.body.appendChild(btn);

btn.onclick = function(event:any):void{
    console.log('label',this.label); // undefined
    this.label.textContent = 'button clicked'; // error
};

```

Di baris 9 contoh di atas, `this.label` bernilai `undefined` karena `this` mengacu pada objek `btn`. Kalo `btn` kita klik, hasilnya error di console:

```

VM1205:9 Uncaught TypeError: Cannot set property 'textContent' of
undefined

```

Sekarang kita ganti *onclick-callback*-nya pake *lambda*. Sintaksnya `() => {}`.

```

btn.onclick = (event:any) => {
    console.log('label',this.label);
    this.label.textContent = 'button clicked';
};

```

Lambda di atas jadi JavaScript kayak ini:

```

var _this = this;
btn.onclick = function (event) {
    console.log('label', _this.label);
    _this.label.textContent = 'button clicked';
};

```

Kalo `btn` diklik, nggak ada error & muncul teks `button clicked` kayak yang kita mau.

1.7.2. Rest Parameter

Kalo Anda kenal ActionScript 3.0, pasti kenal yang namanya *rest-parameter* untuk bikin function yang jumlah parameternya nggak diketahui. Di TS sintaksnya sama dengan

ActionScript 3.0, pake tiga titik

```
function testRest(firstname:string, ...othernames:string[]):string{
    return firstname + ' ' + othernames.join(' ')
}

console.log(testRest('thyrion', 'lannister'));
console.log(testRest('jon', 'snow', 'of', 'winterfell'));
```

1.8. Namespace

kayak yang kita tahu, bikin variabel atau function dalam *scope* global (*window*) adalah salah satu *bad practice* dalam JavaScript. Praktek umum yang banyak dipake untuk menghindarinya adalah dengan membungkus kode dalam *Immediately Invoked Function Execution* (IIFE).

```
(function(){
    //kode kita
})();
```

TypeScript mempermudah pengaturan *scope* lewat apa yang disebut *Namespace*. Contohnya kayak begini:

```
namespace Vehicle{
    let name:string = null;
    function startEngine(){
        console.log(name+ ' starting')
    }
}
```

JavaScript hasil kompilasinya kayak di bawah ini. Kodenya sama dengan IIFE di mana variabel *name* & function *startEngine* ada dalam *scope* objek *Vehicle*.

```
var Vehicle;
(function (Vehicle) {
  var name = null;
  function startEngine() {
    console.log(name + ' starting');
  }
})(Vehicle || (Vehicle = {}));
```

Kode di atas nggak terlalu berguna kalo variabel & function di dalamnya nggak bisa dipake sama skrip lain. Untuk mengekspos isi sebuah `namespace`, kita pake `export`.

```
namespace Vehicle{
  export let name = null;
  export function startEngine() {
    console.log(name + ' starting');
  }
}
```

Output JavaScriptnya jadi kayak di bawah ini. Perhatikan variabel `name` dan function `startEngine()` sekarang jadi atribut objek `Vehicle`.

```
var Vehicle;
(function (Vehicle) {
  Vehicle.name = null;
  function startEngine() {
    console.log(Vehicle.name + ' starting');
  }
  Vehicle.startEngine = startEngine;
})(Vehicle || (Vehicle = {}));
```

Kode yang di-ekspor bisa dipake di kode lain di luar `namespace`.

```
Vehicle.name = 'tank';
Vehicle.startEngine();
```



Khusus untuk variabel, supaya bisa diekspor harus dikasih nilai misalnya `null`. Kalo hanya deklarasi saja (tanpa nilai), variabelnya nggak akan diekspor karena dianggap *dead-code* oleh *compiler*.

Namespace bisa juga kita pake untuk mengorganisir kode. Kita bisa tulis kode di beberapa file terpisah & semuanya pake *namespace* yang sama.

Kita coba bikin direktori baru & kita buat file TypeScript kayak begini:

```
//file vehicle.ts
namespace Vehicle{
    export let name:string = null;
    export function startEngine(){
        console.log(name + ' starting');
    }
}
```

```
//file shoot.ts
namespace Vehicle{
    export function shoot(){
        console.log(name+' shooting')
    }
}
```

```
//file index.ts
Vehicle.name = 'tank';
Vehicle.startEngine();
Vehicle.shoot();
```

Terus kita *compile* dengan perintah (di terminal):

```
tsc *.ts
```

Outputnya adalah file `index.js`, `vehicle.js` & `shoot.js`.

Lanjut, kita buat file `index.html` yg isinya begini:

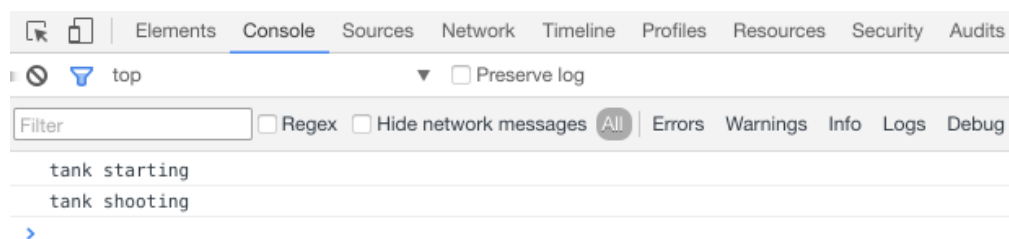
Listing 2. File index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <script src="vehicle.js" charset="utf-8"></script>
    <script src="shoot.js" charset="utf-8"></script>
    <script src="index.js" charset="utf-8"></script>
  </head>
  <body>

  </body>
</html>
```



Buka file `index.html` di browser & lihat console.



Bab 2. Interface & Class

TypeScript dibuat oleh Anders Heljsberg. Jenius yang bikin Bahasa Pascal & C#. Jadi wajar kalo OOP jadi salah satu fitur utama TypeScript. Jadi nggak hanya memperkenalkan konsep Class, TypeScript juga membawa konsep lain yang ada di bahasa OOP semacam C# kayak *interface*, *generics*, & *access modifier*.

Sama dengan bahasa lain, *interface* dipake untuk mendefinisikan properti sebuah objek. Hanya deklarasi aja, definisinya harus di tulis di kode lain yang mengimplementasikan *interface* itu. Jadi *interface* ibaratnya sebuah perjanjian/kontrak, kalo objek A memiliki/mengimplementasi *interface* B, maka objek A pasti punya properti yang didefinisikan di *interface* B.

Sebagai contoh lihat kode berikut:

```
interface IStartable{
    //hanya deklarasi
    started:boolean;
    stopped:boolean;
    start():void;
    stop():void;
}
```

Listing 3. Implementas interface

```
class Car implements IStartable{
    //implementasi IStartable,
    //bikin definisi variabel & function
    started:boolean = false;
    stopped:boolean = true;
    start():void{
        console.log('starting');
    }
    stop():void{
        console.log('stopping');
    }
}
```



Penulisan variabel dalam sebuah kelas/interface nggak pakai kata kunci `let` atau `var`. Function juga nggak pakai kata kunci `function`.

Sebuah kelas bisa mengimplementasikan lebih dari satu *interface*.

Listing 4. Implementasi lebih dari satu interface

```
interface IFly{
    fly():void;
}

interface IStartable{
    started:boolean;
    stopped:boolean;
    start():void;
    stop():void;
}

class Plane implements IStartable, IFly{

    //implementasi IStartable
    started:boolean = false;
    stopped:boolean = true;
    start():void{
        console.log('starting');
    }
    stop():void{
        console.log('stopping');
    }

    //implementasi IFly
    fly():void{
        console.log('flying')
    }
}
```

Kalo Anda belum pernah bikin program pake bahasa yang *full OOP* kayak Java atau ActionScript 3.0, pasti Anda tanya

Terus apa gunanya interface selain bikin banyak ngetik?

Jawabannya: *Abstraksi*.

2.1. Abstraksi

Abstraksi (*abstraction*) menawarkan fleksibilitas dalam penulisan kode. Caranya kita pake tipe data yang abstrak atau generik setiap kali kita bikin variabel. Maksudnya supaya variabel yang kita buat nggak terikat pada satu macam tipe data / objek aja, tapi bisa dipake bermacam objek yang kompatibel. Abstraksi bisa bantu kita menyederhanakan kode karena variabel yang kita buat bisa dipake untuk menyimpan data yang beda-beda tapi masih kompatibel satu sama lain tergantung situasi. Dengan kata lain, variabel yang kita buat bersifat polimorfis.

Polimorfisme asalnya dari bahasa Yunani (polys, morphs) yang artinya banyak bentuk. Dalam bahasa OOP, fitur ini adalah implementasi dari *Liskov Substitution Principle* yang bunyinya (kurang lebih):

Kalo tipe S adalah turunan tipe T, maka semua objek bertipe T bisa digantiin sama objek bertipe S tanpa perlu mengubah kode

Interface adalah tipe data abstrak karena hanya berisi deklarasi variabel & function, tanpa nilai/isi.

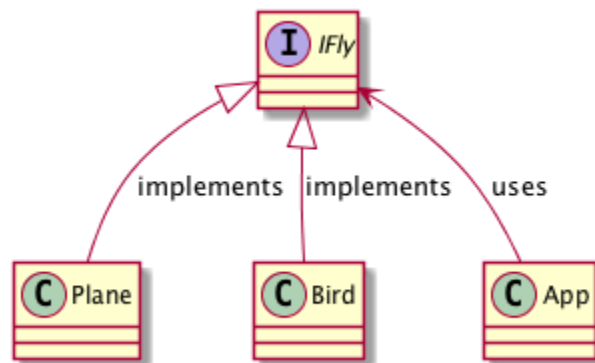
Kita bikin contoh lain.

```
class Plane implements IFly{
  fly():void{
    console.log('Flying with engine');
  }
}

class Bird implements IFly{
  fly():void{
    console.log('Flying with wings');
  }
}

let flyQueue:IFly[] = [ new Plane(), new Bird()];
let f:IFly;
while(flyQueue.length > 0){
  f = flyQueue.shift();
  f.fly();
}
```

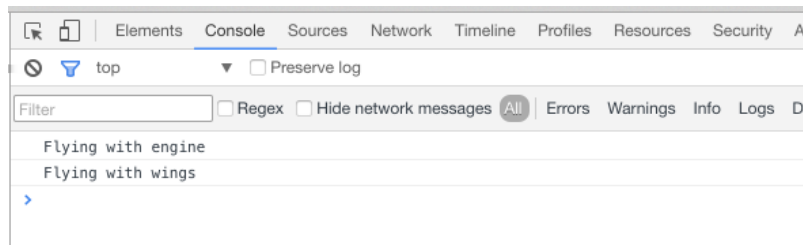
Variabel `f` di atas bersifat polimorfis karena bisa dikasih objek `Plane` atau `Bird` atau objek lain selama objek itu adalah implementasi dari `IFly`.



Gambar 1. Diagram UML



Dalam TypeScript sebuah objek dianggap kompatibel dengan sebuah interface kalo semua atribut (variabel & function) dari interface ada di objek itu. Nggak 100% sama dengan interface di bahasa lain. Maklum, bagaimanapun TypeScript tetap JavaScript.



Abstraksi nggak hanya kita praktekan pake *interface* tapi juga bisa pake turunan (*inheritance*).

2.2. Turunan

Turunan (*inheritance*) adalah struktur di mana sebuah tipe data (*sub-type*) kita bikin berdasarkan tipe lain (*super-type*). Mirip dengan *interface*, bedanya *super-type* nggak hanya berisi deklarasi tapi juga berisi definisi. Semua atribut (variabel & function) yang dimiliki oleh *super-type* diwariskan ke *sub-type* kecuali yang bersifat pribadi (*private*). *Sub-type* bisa melakukan *override* yaitu mengganti atau menambah kode sebuah function yang diwarisi dari *super-type* dengan kodenya sendiri. Jadi bisa dibilang *super-type* lebih generik sementara *sub-type* lebih spesifik.

Kita bikin *super-type* kayak ini:

Listing 5. Kelas Animal

```
//super-type
class Animal {
  private _planet:string = 'earth'
  name:string;
  constructor(name:string){
    this.name = name;
  }
  move():void{
    console.log(this.name, ' is moving');
  }
  eat():void{
    console.log(this.name, ' is eating');
  }
  sleep():void{
    console.log(this.name, ' is sleeping');
  }
}
```

Dan *sub-type*-nya (turunannya) kayak di bawah ini.

Listing 6. Turunan

```
class Dog extends Animal{
  constructor(){
    // jalanin constructor pake kode yang ada di Animal
    super('dog');
  }
  //override
  move():void{
    console.log(this.name, ' walking');
  }
}

class Duck extends Animal{
  constructor(){
    super('duck');
  }
  move():void{
    super.move();// jalanin move punya Animal
    console.log(this.name, ' swimming'); // kode tambahan
  }
}

let animals:Animal[] = [ new Dog(), new Duck()];
for(let i:number = 0; i < animals.length; i++){
  let an:Animal = animals[i];
  an.move();
  an.eat();
  an.sleep();
}
```

Dog & Duck mewarisi semua atribut **Animal** kecuali variabel **_planet** yang bersifat **private**. Kalo kita coba akses variabel **_planet** di dalam **Dog**, TSC pasti komplain:

```
class Dog extends Animal{
  constructor(){
    super('dog');
    this._planet = 'moon'; //error
  }
}
```

```

18
19 class Dog extends Animal {
20   co Property '_planet' is private and only accessible within class 'Animal'.
21   (property) Animal._planet: string
22   this._planet
23 }
24 }
25

```

2.2.1. Access Modifier

TypeScript nyediain tiga macam *access modifier*. Buat Anda yang nggak pernah koding pake Java / ActionScript 3.0 atau sejenisnya: *access modifier* menentukan objek apa yang boleh dan nggak boleh mengakses variabel atau function.

- **public** : terbuka untuk umum. Siapa aja boleh pake.
- **private** : milik pribadi. Objek lain nggak boleh pake.
- **protected** : Hanya objek yang bersangkutan & turunannya yang boleh pake.

Listing 7. Access Modifier

```
class Human{
    private fingerprint:any = 'human fingerprint';
    protected dna:any = 'human dna';
    constructor(){}
    kill(){}
}

class Child extends Human{
    constructor(){
        super();

        //error, fingerprint itu private
        this.fingerprint = 'child fingerprint';

        //ok, dna protected, boleh diakses sama turunan Human
        this.dna = 'child dna';

    }
}

class Animal{
    owner:Human;
    constructor(){
        this.owner = new Human();

        // error, dna itu protected
        console.log(this.owner.dna);

        //error juga, apalagi private
        console.log(this.owner.fingerprint);

        //ok, kill() itu public
        this.owner.kill();
    }
}
```

2.3. Generics

Dalam pengembangan aplikasi yang kompleks, komponen yang kita buat nggak hanya perlu punya API yang jelas & konsisten, tapi juga sebisa mungkin *reusable*. Bahasa kayak Java, C#, & C++ menawarkan fitur yang disebut *Generics* yang membuat kita bisa bikin komponen

yang fleksibel & bisa bekerja dengan tipe data yang berbeda.

Generics bisa dibilang mirip `any`, bedanya kalo kita pake `any` TSC nggak bisa memvalidasi (*type-checking*) kode secara akurat karena nggak ada info tentang tipe data yang dipake. Kalo kita pake editor yang mendukung *Intellisense* ini juga nggak jalan karena kurang info. Sebaliknya, kalo pake *Generics* kita tetap bisa mengandalkan TSC untuk validasi kode & editor yang kita pake nggak kehilangan *Intellisense*-nya.

Listing 8. Generic Array

```
class GenericArray<T>{

    private _items:T[];

    constructor(a:T[]){
        this._items = [];
        for(let i:number = 0; i < a.length ;i++){
            this._items.push(a[i]);
        }
    }
    getItemAt(i:number):T{
        return this._items[i];
    }
    getLastItem():T{
        return this._items[this._items.length-1];
    }
    addItem(item:T):void{
        this._items.push(item);
    }
    removeItemAt(n:number):T[]{
        this._items.splice(n,1);
        return this._items;
    }
}
```

Coba bikin `GenericArray` dengan tipe data `number`.

```
let nums:GenericArray<number> = new GenericArray<number>([1,2,3]);
nums.getItemAt(0).toFixed(2);
```

Kalo editor kita mendukung TS & punya *intellisense* kayak TS playground, waktu kita ketik `nums.getItemAt(0)`, kita bisa liat daftar API untuk tipe `number`.

```
let nums:GenericArray<number> = new GenericArray<number>([1,2,3]);
nums.getItemAt(0).
```

- ✧ toString (method) Number.toString(radix?: number): string
Returns a string representation of an object.
- ✧ toFixed
- ✧ toExponential
- ✧ toPrecision
- ✧ valueOf
- ✧ toLocaleString

Kalo kita pake `string` yang muncul ya daftar API-nya `string`.

```
let strs:GenericArray<string> = new GenericArray<string>(['a','b',
,'c']);
strs.getItemAt(0).toUpperCase();
```

```
let strs:GenericArray<string> = new GenericArray<string>(['a','b','c']);
strs.getItemAt(0).
```

- ✧ slice (method) String.slice(start?: number, end?: number): string
Returns a section of a string.
- ✧ toString
- ✧ charCodeAt
- ✧ concat
- ✧ indexOf
- ✧ lastIndexOf
- ✧ localeCompare
- ✧ match
- ✧ replace
- ✧ search
- ✧ charAt
- ✧ split

Kalo kita coba masukan data yang nggak kompatibel, misalnya begini:

```
strs.addItem(100);
```

Jadinya error.

```
let strs:GenericArray<string> = new GenericArray<string>(['a','b','c']);
strs.addItem(100);
```

Argument of type 'number' is not assignable to parameter of type 'string'.

Contoh lain, pake kelas.

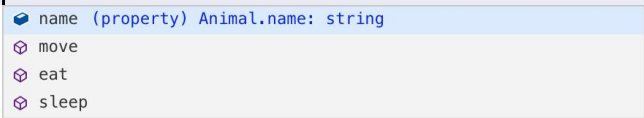
Listing 9. Generic Array Object

```
class Animal{
    name:string = null;
    constructor(name:string){
        this.name = name;
    }
    move(){ }
    eat(){ }
    sleep(){ }
}

class Bird extends Animal{
    constructor(){
        super('Bird');
    }
}

let zoo:GenericArray<Animal> = new GenericArray<Animal>([ new
Animal('dog'), new Animal('duck') ]);
zoo.getItemAt(0).move();
zoo.addItem(new Bird());
```

```
let zoo:GenericArray<Animal> = new GenericArray<Animal>([ new Animal('dog')]);
zoo.getItemAt(0).|
```



Kalo cuma bikin `array` kayak di atas, kita sebenarnya nggak perlu pake Generics, *vector* /*typed-array* aja cukup. Jadi kita coba yang sedikit lebih rumit, bikin *linked-list*.

Listing 10. Linked-list pake Generics

```
class GenericItem<T>{
    value:T;
    next:GenericItem<T>;
    prev:GenericItem<T>;
    constructor(value:T,prev?:GenericItem<T>,next?:GenericItem<T>){
        this.value = value;
        this.prev = prev;
        this.next = next;
    }
}
```

```

}

class GenericList<T>{

    private _items:GenericItem<T>[];

    //parameter constructor berupa tuple supaya
    //list minimal punya satu item,
    //nggak boleh kosong
    constructor(a:[T]){
        this._items = [];
        let currentItem:GenericItem<T> = null;
        let prevItem:GenericItem<T> = null;
        for(let i:number = 0; i < a.length ;i++){
            currentItem = new GenericItem<T>(a[i]);
            prevItem = this.getLastItem();
            if(prevItem){
                currentItem.prev = prevItem;
                prevItem.next = currentItem;
            }
            this._items.push(currentItem);
        }
    }
    getItemAt(i:number):GenericItem<T>{
        return this._items[i];
    }
    getLastItem():GenericItem<T>{
        return this._items[this._items.length-1];
    }
    addItem(item:T):void{
        let newItem:GenericItem<T> = new GenericItem<T>(item);
        let lastItem:GenericItem<T> = this.getLastItem();
        if(lastItem){
            newItem.prev = lastItem;
            lastItem.next = newItem;
        }
        this._items.push(newItem);
    }
    count():number{
        return this._items.length;
    }
}

class Animal{
    name:string = null;

```

```

    age:number = 0;
    constructor(name:string,age:number){
        this.name = name;
        this.age = age;
    }
    move(){ console.log(this.name, ' moving') }
    eat(){ console.log(this.name, ' eating') }
    sleep(){ console.log(this.name, ' sleeping') }
}

class Bird extends Animal{
    constructor(age:number){
        super('Bird',age);
    }
}

let zoo:GenericList<Animal> = new GenericList<Animal>([new
Animal('snake',2)]);
zoo.addItem(new Animal('dog',4));
zoo.addItem(new Animal('duck',2));
zoo.addItem(new Bird(3));

let an:Animal = zoo.getFirstItem().value;
let n:number = 0;
let totalAge:number = 0;
while(true){
    let item:GenericItem<Animal> = zoo.getItemAt(n);
    item.value.move();
    item.value.eat();
    item.value.sleep();
    totalAge += item.value.age;
    if(!item.next){
        break;
    }

    n++;
}

console.log('Average Age',(totalAge/zoo.count()).toFixed(2));

```

2.4. Variabel & Function Statis

Static function/variabel dimiliki oleh kelas, bukan objek dari kelas itu. Bedanya begini,

Listing 11. Variabel & function static

```
class Animal{
  name:string;
  static home:string = 'earth';
  constructor(name:string){
    this.name = name;
  }
  static migrate(planet:string):void{
    Animal.home = planet;
  }
  getHome():string{
    return Animal.home;
  }
}

class Dog extends Animal{
  constructor(){
    super('dog')
  }
}

class Duck extends Animal{
  constructor(){
    super('duck');
  }
}

var anim:Animal = new Animal('unknown');
var dog:Dog = new Dog();
var duck:Duck = new Duck();

console.log(anim.home);//error, undefined
console.log(dog.home);//error, undefined

console.log(Animal.home);//earth
console.log(anim.getHome());//earth
console.log(dog.getHome());//earth
console.log(duck.getHome());//earth

Animal.migrate('mars');
console.log(Animal.home);//mars
console.log(anim.getHome());//mars
console.log(dog.getHome());//mars
```

```
console.log(duck.getHome()); //mars
```

`anim.home` & `dog.home` jadi error & nilainya `undefined` karena `home` itu variabel statis punya kelas `Animal` bukan objek kelas itu atau objek dari kelas turunannya. Karena variabel ini punya kelas `Animal`, kalo nilainya diganti, semua objek dari kelas itu & turunannya tahu perubahan itu.

Lawan dari *static function/variable* adalah *instance function/variable*. Semua variabel & function yang bukan statis adalah milik *instance* atau objek dari kelas yang bersangkutan.

Static function, kayak `Animal.migrate()` dalam contoh di atas hanya bisa mengakses variabel atau function lain yang sama-sama statis. Dia nggak bisa akses *instance variable/function*. Sebaliknya, *instance function* bisa mengakses *static variable/function*, contohnya `getHome()` di atas.

Bab 3. Modul

Dalam bab ini kita belajar tentang modul & cara bikin aplikasi yang sifatnya modular. Sebagai contoh kita bikin aplikasi kalkulator.

TSC bisa menghasilkan modul JavaScript dalam beberapa format:

- CommonJS kayak NodeJS
- AMD (*Asynchronous Module Definition*) ala RequireJS
- gabungan keduanya yang disebut UMD (*Universal Module Definition*)
- SystemJS
- ES6

Kalo pake CommonJS, kita harus install Webpack/Browserify untuk mengkonversi format CommonJS ke JavaScript standar jadi ada proses kompilasi lagi. Habis .ts → .js (common) terus .js (common) → Webpack/Browserify → .js (standar). ES6 juga sama, harus dikompilasi lagi pake Babel. AMD, UMD, & SystemJS cukup satu kali kompilasi.

Kita coba liat output masing-masing format modul tapi sebelumnya, kita bikin dulu struktur direktori untuk aplikasi Kalkulator.

```
calc
├─ js/
├─ file *.js
├─ typescript/
└─ file *.ts
```

js : isinya file JavaScript hasil kompilasi

typescript: isinya file TS & file konfigurasi (**tsconfig.json**).

Bikin direktorinya, terus jalanin perintah ini di direktori **typescript/**:

```
tsc --init
```

Perintah itu bikin file **tsconfig.json** yang isinya begini:

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": false,
    "sourceMap": false
  },
  "exclude": [
    "node_modules"
  ]
}
```

3.1. Export & Import

Untuk bikin modul, kita harus pake kata kunci `export`. Masih di direktori `typescript/`, kita bikin file `command.ts` & `calc.ts`. Isinya kode di bawah ini.

Listing 12. File `typescript/command.ts`

```
export class Command{
  name:string;
  constructor(name:string = 'cmd'){
    this.name = name;
  }
  execute(){
    console.log(this.name, 'execute');
  }
}
```

Biar bisa pake modul, kita perlu `import` dulu modulnya.

Listing 13. File `typescript/calculator.ts`

```
//import modul Command dari file command.ts
import { Command } from './command';
export class Calculator{
    //pake modul Command
    cmd:Command;
    constructor(){
        this.cmd = new Command();
        this.cmd.execute();
    }
}
```

Karena setiap modul ekspor objek yang isinya bisa lebih dari satu kelas, untuk impor modul kita pake notasi objek `{ }`.

```
import { Command } from './command';
```

Bukan begini,

```
import Command from './command';
```

Buka file `tsconfig.json` & tambahkan kode berikut di bawah `"exclude:[]"`:

```
"files":[
    "calculator.ts",
    "command.ts"
]
```

Terus tambahkan `outDir` di bagian `compilerOptions`, arahkan ke direktori `js/`.

Konfigurasinya jadi begini:

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": false,
    "sourceMap": false,
    "outDir": "../js/"
  },
  "exclude": [
    "node_modules"
  ],
  "files": [
    "command.ts",
    "calc.ts"
  ]
}
```

Sekarang buka terminal & jalankan perintah berikut (di direktori `typescript/`):

```
tsc --listFiles
```

Karena kita pake `tsconfig.json` perintah kompilasinya jadi lebih sederhana. Opsi `--listFiles` kita pake untuk liat file apa aja yang diproses sama TSC.

Di komputer saya, perintah di atas nampilin daftar file kayak di bawah ini. Di komputer Anda pasti beda.

```
/Users/boss/.nvm/versions/node/v5.10.0/lib/node_modules/typescript/
lib/lib.d.ts
/Users/boss/workspaces/personal/tutorial/belajar-
typescript/ex/03/typescript/command.ts
calc.ts
```

Kita coba liat output JavaScriptnya. Buka file `js/calc.js` Karena pake format modul CommonJS, isinya kayak ini:

Listing 14. File `js/calc.js` format `CommonJS`

```
"use strict";
var command_1 = require('./command');
var Calculator = (function () {
  function Calculator() {
    console.log('Calculator');
    this.cmd = new command_1.Command();
    this.cmd.execute();
  }
  return Calculator;
})();
exports.Calculator = Calculator;
```

Coba pake format `AMD`. Ganti setting `module` di `tsconfig.json`.

```
"module": "commonjs"
```

Jadi begini

```
"module": "amd"
```

Terus *compile* lagi.

Listing 15. File `js/calc.js` format `AMD`

```
define(["require", "exports", './command'],
function (require, exports, command_1) {
  "use strict";
  var Calculator = (function () {
    function Calculator() {
      console.log('Calculator');
      this.cmd = new command_1.Command();
      this.cmd.execute();
    }
    return Calculator;
  })();
  exports.Calculator = Calculator;
});
```

Begini outputnya kalo modulnya `umd`. Bisa langsung dipake di `NodeJS` & `web` (+ `RequireJS`).

Listing 16. File js/calc.js format UMD

```
(function (factory) {  
    if (typeof module === 'object' && typeof module.exports ===  
    'object') {  
        var v = factory(require, exports); if (v !== undefined)  
module.exports = v;  
    }  
    else if (typeof define === 'function' && define.amd) {  
        define(["require", "exports", './command'], factory);  
    }  
})(function (require, exports) {  
    "use strict";  
    var command_1 = require('./command');  
    var Calculator = (function () {  
        function Calculator() {  
            console.log('Calculator');  
            this.cmd = new command_1.Command();  
            this.cmd.execute();  
        }  
        return Calculator;  
    })();  
    exports.Calculator = Calculator;  
});
```

Output modul **system**:

Listing 17. File `js/calcul.js` format SystemJS

```
System.register(['./command'], function(exports_1, context_1) {
    "use strict";
    var __moduleName = context_1 && context_1.id;
    var command_1;
    var Calculator;
    return {
        setters:[
            function (command_1_1) {
                command_1 = command_1_1;
            },
        ],
        execute: function() {
            Calculator = (function () {
                function Calculator() {
                    console.log('Calculator');
                    this.cmd = new command_1.Command();
                    this.cmd.execute();
                }
                return Calculator;
            })();
            exports_1("Calculator", Calculator);
        }
    };
});
```



Dalam tutorial ini kita pake SystemJS aja, karena saya pikir itu yang paling sederhana dibanding format lainnya.

Kita bikin file `main.ts` yang nantinya jadi *entry point* aplikasi kita.

```
import { Calculator } from './calc';
let calc:Calculator = new Calculator();
```

Buka `tsconfig.json` & tambahin `main.ts` ke dalam daftar `files`. Terus *compile*.

Lanjut bikin file `index.html` di root (direktori `calc/`). Isinya begini:

Listing 18. File index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <script

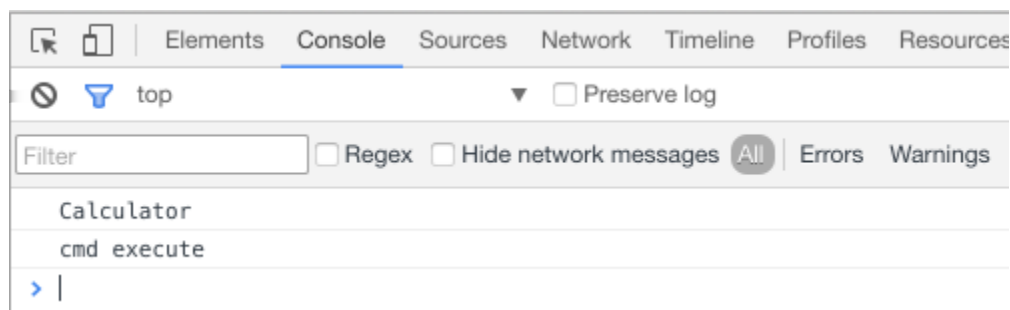
src="http://cdnjs.cloudflare.com/ajax/libs/systemjs/0.19.27/system.
js">
    </script>
    <script>
      System.config({
        baseUrl: 'js',
        defaultJSExtensions: true
      });
      System.import('js/main.js');
    </script>
  </body>
</html>
```

Berikutnya kita bikin *local server*. Kalo Anda punya PHP 5.6, bisa pake server bawaan PHP. Caranya jalanin perintah berikut di direktori `calc/`

```
php -S localhost:9000 -t .
```

Kalo Anda pake WAMP / MAMP silakan atur sendiri. Bisa juga pake aplikasi gratis namanya Fenix yang bisa diunduh di <http://fenixwebserver.com/>.

Buka alamat `localhost:9000/` di browser & liat *console*.



3.2. Ekspor Lebih Dari Satu Kelas

Satu file TS bisa berisi lebih dari satu `export`. Kita coba tambahkan kelas `CommandQueue` di modul `command`.

Listing 19. File typescript/command.ts

```
export class CommandQueue{
  private _commands:Command[] = [];
  constructor(){}
  add(cmd:Command):void{
    this._commands.push(cmd);
  }
  execute(){
    console.log('queue execute');
  }
}
```

Terus kita impor semua kelas dari `command.ts`.

Listing 20. File typescript/calc.ts

```
//impor semua kelas dari command.ts
import * as Commands from './command';

export class Calculator{
  cmd:Commands.Command;
  queue:Commands.CommandQueue;
  constructor(){
    this.cmd = new Commands.Command();
    this.cmd.execute();

    this.queue = new Commands.CommandQueue();
    this.queue.execute();
  }
}
```



Untuk impor semua kelas / function yang di ekspor sebuah modul, kita harus bikin nama alias (`as`) untuk objek yang diekspor modul itu

Gimana kalo kita nggak butuh semua kelas, tapi hanya sebagian aja?

Untuk impor lebih dari satu kelas, caranya sama. Pake notasi objek & nama semua kelas yang mau kita impor. Kalo `command.ts` ekspor banyak kelas tapi kita hanya butuh `Command` & `CommandQueue`, kita tulis impor kayak contoh di bawah. Sintaksnya sama dengan impor satu kelas.

Listing 21. File typescript/calculator.ts

```
//hanya impor kelas Command & CommandQueue
//nggak pake alias (as) lagi
import { Command, CommandQueue } from './command';

export class Calculator{
  cmd:Commands.Command;
  queue:Commands.CommandQueue;
  constructor(){
    this.cmd = new Commands.Command();
    this.cmd.execute();

    this.queue = new Commands.CommandQueue();
    this.queue.execute();
  }
}
```

3.3. Ekspor Function

Sebagai alternatif *static function*, kita juga bisa langsung ekspor function dari sebuah file TS. Jadi lebih simpel karena nggak perlu bikin kelas.

Kita bikin file `utils.ts`.

Listing 22. File typescript/utils.ts

```
export function log(...msg):void{
  console.log('[Calc]',msg);
}
```

Importnya sama dengan import kelas.

Listing 23. File typescript/calculator.ts

```
import {Command, CommandQueue} from './command';

//import semua function
import * as Utils from './utils';

export class Calculator{
  cmd:Command;
  constructor(){
    this.cmd = new Command();
    //pake Utils.log()
    Utils.log('Calc constructor');
  }
}
```

Kalo mau impor function `log()` aja dari `Utils`, sama dengan *selective import* di atas.

Listing 24. File typescript/calculator.ts

```
import {Command, CommandQueue} from './command';

//import log() aja
import { log } from './utils';

export class Calculator{
  cmd:Command;
  constructor(){
    this.cmd = new Command();
    //nggak pake Utils. lagi
    log('Calc constructor');
  }
}
```

Di bagian selanjutnya, kita liat kode lengkap aplikasi Kalkulator ini.

Bab 4. Contoh Aplikasi Kalkulator

Di bawah ini kode dari aplikasi kalkulator yang dibuat pake konsep yang udah kita bahas. Sengaja saya nggak jelasin detil logikanya karena topik tutorial ini bukan Kalkulator tapi TypeScript & semua dasarnya udah dijelasin di bab-bab sebelumnya.

Sekedar ngingetin, struktur direktorinya kayak begini:

```
calc/  
  |_ index.html  
  |_ typescript/  
    |_ *.ts  
  |_ js/  
    |_ *.js
```

Listing 25. File index.html

```
<!DOCTYPE html>  
<html>  
  <head>  
    <meta charset="utf-8">  
    <title></title>  
    <style>  
      #calculator{  
        width:210px;  
        background: #FFCC66;  
        padding:4px;  
      }  
      button{  
        width:41px;  
        height:40px;  
        margin:4px;  
        background: #FF8000;  
        color:#fff;  
      }  
  
      button:hover{  
        background: #CC6600;  
        cursor: pointer;  
      }  
  
      .display{
```

```

        background: #4C4C4C;
        color: #CCFF66;
        font-size: 2em;
        padding: 2px;
        text-align: right;
    }
</style>

</head>
<body>

    <div id="calculator">
        <div class="display">0
        </div>
        <div class="left">
            <div class="row">
                <button class="num">9</button>
                <button class="num">8</button>
                <button class="num">7</button>
                <button class="cmd add">+</button>
            </div>
            <div class="row">
                <button class="num">6</button>
                <button class="num">5</button>
                <button class="num">4</button>
                <button class="cmd sub">-</button>
            </div>
            <div class="row">
                <button class="num">3</button>
                <button class="num">2</button>
                <button class="num">1</button>
                <button class="cmd div">/</button>
            </div>

            <div class="row">
                <button class="num">0</button>
                <button class="cmd clear">C</button>
                <button class="cmd result">=</button>
                <button class="cmd mul">x</button>
            </div>
        </div>
    </div>

<script>

```

```

src="http://cdnjs.cloudflare.com/ajax/libs/systemjs/0.19.27/system.
js">
  </script>
  <script>
    System.config({
      baseUrl: 'js',
      defaultJSExtensions: true
    });
    System.import('js/main');
  </script>
</body>
</html>

```

Listing 26. File typescript/command.ts

```

export class Command{
  static Accumulator:number = 0;

  name:string;
  input:number = 0;

  constructor(name:string='cmd'){
    this.name = name;
  }
  insert(num:string):void{
    this.input = parseInt(this.input+num);
    this.log(`${this.name} insert ${num} => ${this.input}`);
  }
  execute():number{
    Command.Accumulator = this.input;
    return Command.Accumulator;
  }
  log(msg?:string):void{
    if(msg){
      console.log(msg);
    }else{
      console.log(this.name + ' , input:'+ this.input + ' , acc:'
+Command.Accumulator);
    }
  }
}

```

Listing 27. File typescript/adder.ts

```
import { Command } from './command';
export class Adder extends Command{
    constructor(){
        super('Add');
    }
    execute():number{
        this.log();
        Command.Accumulator = Command.Accumulator + this.input;
        return Command.Accumulator;
    }
}
```

Listing 28. File typescript/subtractor.ts

```
import { Command } from './command';
export class Subtractor extends Command{
    constructor(){
        super('Sub')
    }
    execute():number{
        this.log();
        Command.Accumulator = Command.Accumulator - this.input;
        return Command.Accumulator;
    }
}
```

Listing 29. File typescript/multiplier.ts

```
import { Command } from './command';
export class Multiplier extends Command{
    constructor(){
        super('Mul');
    }
    execute():number{
        this.log();
        Command.Accumulator = Command.Accumulator * this.input;
        return Command.Accumulator;
    }
}
```

Listing 30. File typescript/divisor.ts

```
import { Command } from './command';
export class Divisor extends Command{
  constructor(){
    super('Div');
  }
  execute():number{
    this.log();
    Command.Accumulator = Command.Accumulator / this.input;
    return Command.Accumulator;
  }
}
```

Listing 31. File typescript/utils.ts

```
export function getRoot(id:string):any{
  return document.getElementById(id);
}

export function getButtons(rootId:string,selector:string):any[] {
  let root:any = getRoot(rootId);
  let btns:any = root.querySelectorAll(selector);
  let buttonArray:any[] = [];
  for(let i:number =0;i<btns.length;i++){
    buttonArray.push(btns[i]);
  }
  return buttonArray;
}

export function getDisplay(rootId:string):any{
  let root:any = getRoot(rootId);
  return root.querySelectorAll('.display')[0];
}
```

Listing 32. File typescript/calc.ts

```
import { Command } from './command';
import { Adder } from './adder';
import { Subtractor } from './subtractor';
import { Multiplier } from './multiplier';
import { Divisor } from './divisor';
```



```

import * as Utils from './utils';

export class Calculator{
  cmd:Command;
  root:any;
  display:any;
  cmdButtons:any[];
  numButtons:any[];
  constructor(id:string){
    this.cmd = new Command();
    this.root = Utils.getRoot(id);

    this.numButtons = Utils.getButtons(id, '.num');
    this.numButtons.forEach(btn=>{
      btn.onclick = ()=>{
        this.cmd.insert(btn.textContent);
        this.display.textContent = this.cmd.input;
      }
    });

    this.cmdButtons = Utils.getButtons(id, '.cmd');
    this.cmdButtons.forEach(btn=>{
      btn.onclick = ()=>{
        let cmd:Command;
        let className:string = /cmd\s(.*)/gi.exec(btn.className)[
1];

        if(className === 'clear'){
          this.clear();
        }else if(className === 'result'){
          this.getResult();
          this.cmd = new Command();
          this.cmd.insert(Command.Accumulator.toString());
        }else{
          switch(className){
            case 'add':
              cmd = new Adder();
              break;
            case 'sub':
              cmd = new Subtractor();
              break;
            case 'mul':
              cmd = new Multiplier();
              break;
            case 'div':
              cmd = new Divisor();

```

```

        break;
    }
    this.switchCommand(cmd);
}
}
}))
this.display = Utils.getDisplay(id);

console.log('Calc constructor');
}
switchCommand(newCmd:Command):void{
    if(!newCmd) return;
    console.log('switching command ',newCmd.name);
    this.getResult();
    this.cmd = newCmd;
}
getResult():void{
    let result:number = this.cmd.execute();
    this.display.textContent = result;
}
clear():void{
    this.display.textContent = 0;
    Command.Accumulator = 0;
    this.cmd = new Command();
}
}
}

```

Listing 33. File typescript/main.ts

```

import { Calculator } from './calc';
let calc:Calculator = new Calculator('calculator');

```

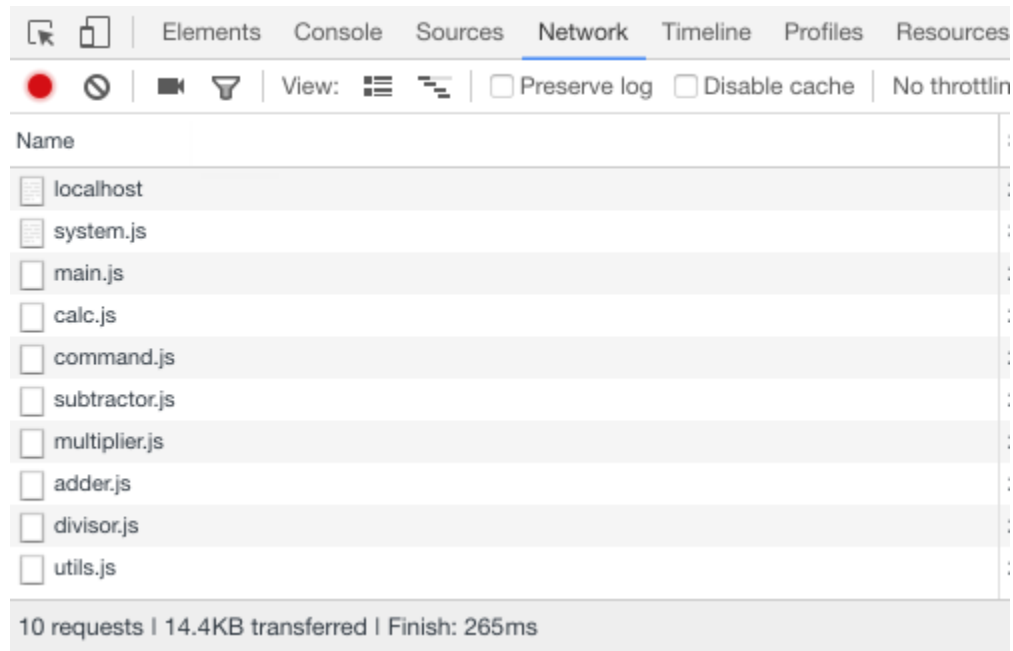
Listing 34. File typescript/tsconfig.json

```
{
  "compilerOptions": {
    "module": "system",
    "target": "es5",
    "noImplicitAny": false,
    "sourceMap": false,
    "outDir": "../js/"
  },
  "exclude": [
    "node_modules"
  ],
  "files": [
    "utils.ts",
    "calc.ts",
    "command.ts",
    "adder.ts",
    "subtractor.ts",
    "divisor.ts",
    "multiplier.ts",
    "main.ts"
  ]
}
```

Coba *compile* & buka `localhost:9000` (atau alamat server Anda) di browser.

4.1. Optimasi

Kalo kita buka Network panel, kita liat ada 10 file yang diunduh.



Untuk optimasi *bandwidth* & mempercepat *loading*, kita bisa kurangi jumlah *request* dengan cara gabungin (*concatenate*) semua skrip jadi satu file JavaScript.



Kalo server Anda udah mendukung HTTP 2.0 nggak perlu lagi gabungin skrip

Buka file `tsconfig.json`, hapus `outDir` & ganti dengan `outFile`.

Listing 35. File typescript/tsconfig.json

```
{
  "compilerOptions": {
    "module": "system",
    "target": "es5",
    "noImplicitAny": false,
    "sourceMap": false,
    "outFile": "../js/index.js"
  },
  "exclude": [
    "node_modules"
  ],
  "files": [
    "utils.ts",
    "calc.ts",
    "command.ts",
    "adder.ts",
    "subtractor.ts",
    "divisor.ts",
    "multiplier.ts",
    "main.ts"
  ]
}
```

Compile, terus buka file `js/index.js`. Kita liat semua modul ada dalam satu file ini.

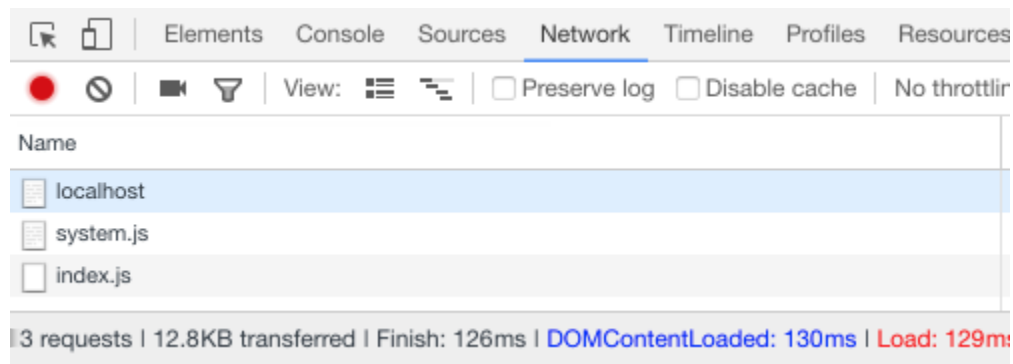
Buka file `index.html` & ganti kode di bawah ini (bagian sebelum `</body>`).

```
<script
src="http://cdnjs.cloudflare.com/ajax/libs/systemjs/0.19.27/system.
js">
</script>
<script>
  System.config({
    baseUrl: 'js',
    defaultJSExtensions: true
  });
  System.import('js/main');
</script>
</body>
```

Jadi begini:

```
<script
src="http://cdnjs.cloudflare.com/ajax/libs/systemjs/0.19.27/system.
js">
</script>
<script src="js/index.js"></script>
<script>
  //import modul "main" dari index.js
  System.import('main');
</script>
</body>
```

Buka file di browser & liat network panel. Sekarang file yang diunduh cuma 3 bukan 10. Ukuran total file yang diunduh sekitar 12.7KB (cache) atau 31.4KB (tanpa cache).



4.2. Minifikasi

Optimasi selanjutnya adalah dengan ngecilin ukuran file lewat proses minifikasi.

Instal paket `uglify-js` lewat NPM.

```
npm install -g uglify-js
```

Buka direktori `calc` di terminal.

```
uglifyjs js/index.js -o js/index.min.js --mangle
```

Buka file `index.html`, ganti `index.js` jadi `index.min.js`.

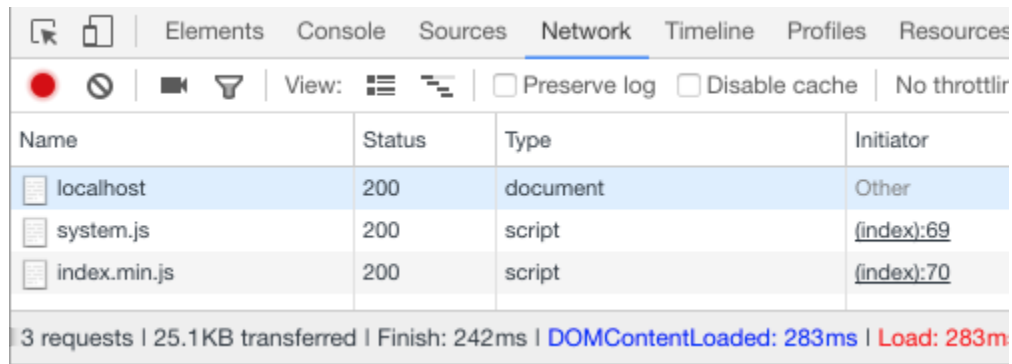
```
<script src="js/index.js"></script>
```

Jadi begini:

```
<script src="js/index.min.js"></script>
```

Buka di browser & liat *Network Panel* lagi.

Hasil minifikasi tanpa cache

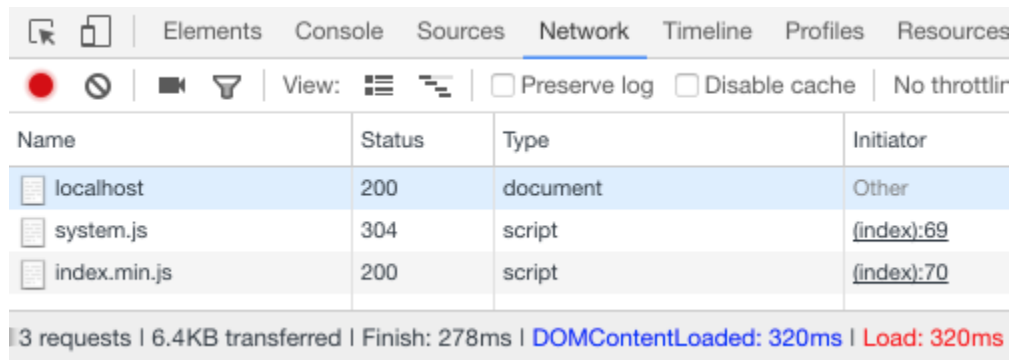


The screenshot shows the Chrome DevTools Network Panel with the 'Network' tab selected. The 'View' dropdown is set to 'Resources'. The table lists three requests: 'localhost' (Status: 200, Type: document, Initiator: Other), 'system.js' (Status: 200, Type: script, Initiator: (index):69), and 'index.min.js' (Status: 200, Type: script, Initiator: (index):70). The summary at the bottom indicates 3 requests, 25.1KB transferred, and a total load time of 283ms.

Name	Status	Type	Initiator
localhost	200	document	Other
system.js	200	script	(index):69
index.min.js	200	script	(index):70

3 requests | 25.1KB transferred | Finish: 242ms | DOMContentLoaded: 283ms | Load: 283ms

Hasil minifikasi pake cache



The screenshot shows the Chrome DevTools Network Panel with the 'Network' tab selected. The 'View' dropdown is set to 'Resources'. The table lists three requests: 'localhost' (Status: 200, Type: document, Initiator: Other), 'system.js' (Status: 304, Type: script, Initiator: (index):69), and 'index.min.js' (Status: 200, Type: script, Initiator: (index):70). The summary at the bottom indicates 3 requests, 6.4KB transferred, and a total load time of 320ms.

Name	Status	Type	Initiator
localhost	200	document	Other
system.js	304	script	(index):69
index.min.js	200	script	(index):70

3 requests | 6.4KB transferred | Finish: 278ms | DOMContentLoaded: 320ms | Load: 320ms


Proses minifikasi ngecilin ukuran file `index.js` dari 11KB jadi 6KB (`index.min.js`). Hampir 50%. Bandwidth totalnya jadi 6.5KB (cache) & 25.1KB (tanpa cache). Lumayan.

Bab 5. Debugging

Sebagus-bagusnya bahasa pemrograman & *compiler*, kalo *debugging*-nya susah, ya percuma, cuma bikin repot. Saya nggak tahu Anda gimana, tapi saya sendiri bukan programmer dewa yang nggak pernah bikin error. *Sering malah* :)

TSC menterjemahkan sintaks TypeScript ke dalam JavaScript standar yang dipake sama browser. Jadi kalo ada error, yang ditunjukkan browser adalah error di JS bukan di TS. Susah mau ngelacak error sebenarnya ada di mana. Kita harus buka file TS nya untuk cari kode aslinya yang lokasinya pasti nggak sama dengan yang ditunjukkan browser. Apalagi kalo kita pake penggabungan skrip. Kalo ditambah proses minifikasi malah *impossible*.

Contohnya nih,



```
✖ Uncaught ReferenceError: someobj is not defined
    n.execute      @ index.min.js:1
    t.getResult    @ index.min.js:1
    t.switchCommand @ index.min.js:1
    t.onclick      @ index.min.js:1
```

Terus gimana dong?

Pake *Source Maps*.

5.1. Source Maps

Untuk bikin *source map* kita ganti setingan TSC. Buka file `tsconfig.json`. Ganti setting `sourceMap` jadi `true`.

Listing 36. File typescript/tsconfig.json pake source-map.

```
{
  "compilerOptions": {
    "module": "system",
    "target": "es5",
    "noImplicitAny": false,
    "sourceMap": true,
    "outFile": "../js/index.js"
  },
  "exclude": [
    "node_modules"
  ],
  "files": [
    "utils.ts",
    "calc.ts",
    "command.ts",
    "adder.ts",
    "subtractor.ts",
    "divisor.ts",
    "multiplier.ts",
    "main.ts"
  ]
}
```

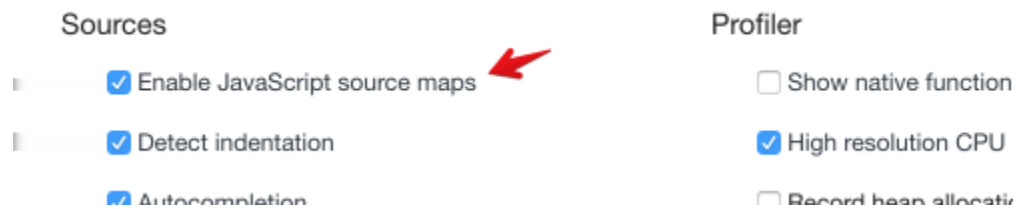
Buka file `typescript/adder.ts`. Kasih *breakpoint*.

Listing 37. File typescript/adder.ts

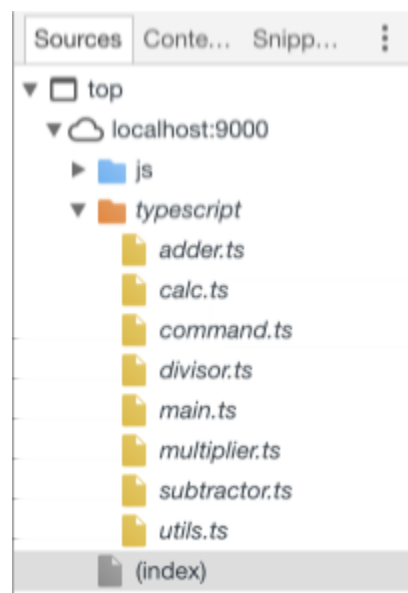
```
import { Command } from './command';
export class Adder extends Command{
  constructor(){
    super('Add');
  }
  execute():number{
    this.log();
    Command.Accumulator = Command.Accumulator + this.input;
    debugger; //BREAKPOINT DI SINI
    return Command.Accumulator;
  }
}
```

Jalanin TSC. Kita liat ada file baru namanya `index.js.map`.

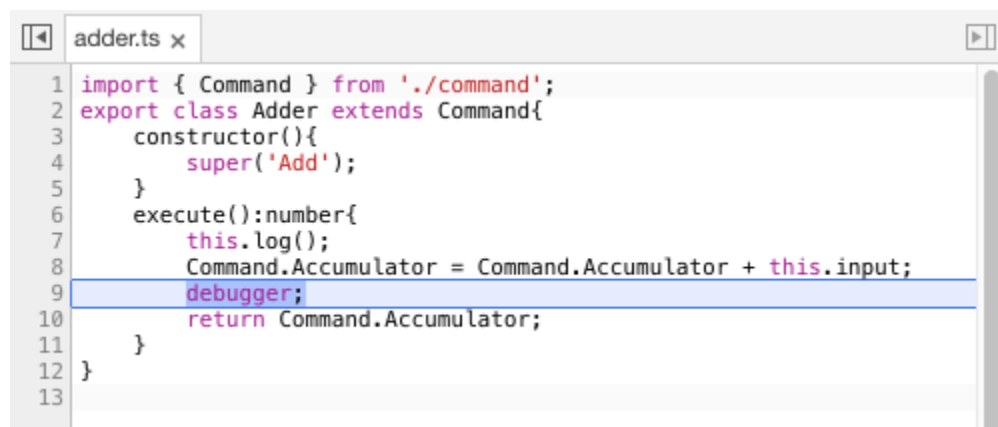
Berikutnya, pastiin setting source-map di *Developer Tools* sudah aktif. Kalo Anda pake browser selain Chrome, silakan cek dokumentasinya.



Sekarang kita coba pake `index.js`, bukan `index.min.js`, terus buka file `index.html` di browser. Kita liat file TS nya ikut diunduh.



Klik tombol `+` di kalkulator 2x. Aplikasi akan berhenti di *breakpoint* kita bisa liat file `adder.ts` tepat di lokasi statemen `debugger`.

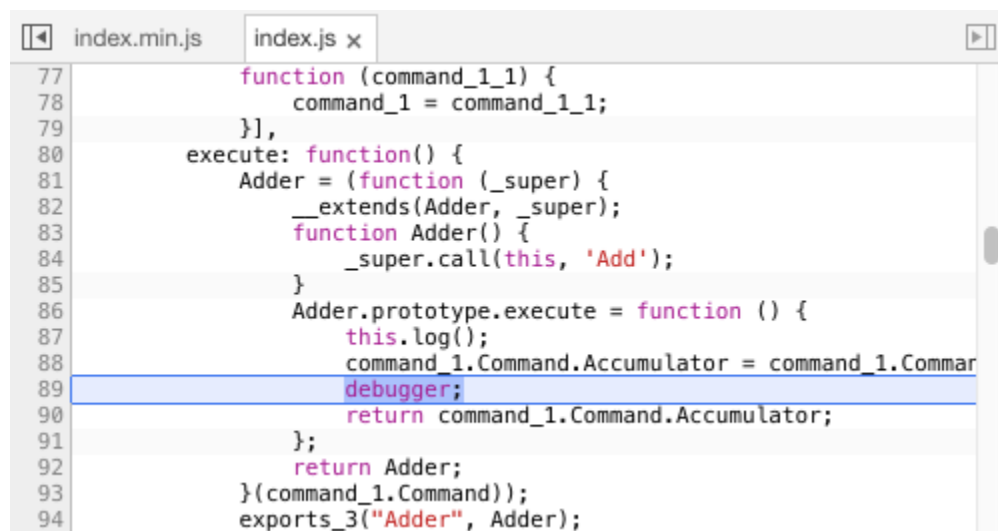


5.2. Minifikasi

Kita minifikasi pake UglifyJS lagi tapi kali ini kita bikin *source-map*-nya juga. Buka direktori `calc/js/` di terminal & jalanin perintah:

```
uglifyjs index.js -o index.min.js --source-map index.min.js.map  
--mangle
```

Kita coba pake `index.min.js`, tapi nggak ada file TS yang diunduh & kode berhenti di `index.js`. Tentu nggak begitu bermanfaat.

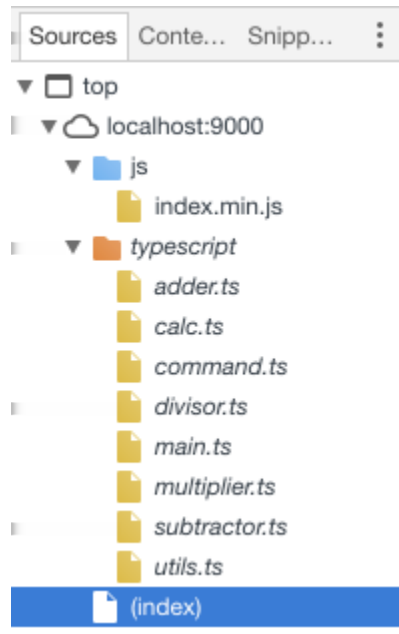


Ini karena Uglify tahunya cuma file `index.js`, dia nggak tau apa-apa tentang file TS.

Kita udah punya file `index.js.map` yang dibikin TSC. File ini kita pake untuk input perintah `uglifyjs` lewat opsi `--in-source-map`. Hasilnya, Uglify punya info tentang file TS yang dia dapat dari `index.js.map`.

```
uglifyjs --in-source-map index.js.map -o index.min.js --source-map  
index.min.js.map --mangle
```

Coba lagi buka di browser. File TS sekarang ikut diunduh.



Klik tombol + 2x. Kode tepat berhenti di `adder.ts`. Sama kayak kalo kita pake `index.js` sebelumnya.

```
adder.ts x
1 import { Command } from './command';
2 export class Adder extends Command{
3   constructor(){
4     super('Add');
5   }
6   execute():number{
7     this.log();
8     Command.Accumulator = Command.Accumulator + this.input;
9     debugger;
10    return Command.Accumulator;
11  }
12 }
13
```

Bab 6. Library Pihak Ketiga

Dalam contoh kode sebelumnya, kita hanya pake kode yang semuanya kita tulis pake TypeScript. Dalam bab ini, kita bahas cara pake *library* eksternal kayak JQuery.

6.1. Ambient Declaration

Ambient (bukan *ambeien*) *declaration* adalah deklarasi objek/*library* yang bukan bagian dari kode aplikasi kita tapi tetap kita butuhkan & nantinya harus kita muat sendiri ke dalam web page pake `<script>`.

```
$( '.btn' ).click( function() {  
    console.log( 'CLICKED' )  
})
```

TSC pasti komplain begini:

```
error TS2304: Cannot find name '$'
```

Tapi kalo kita pake *ambient declaration*, TSC nggak komplain lagi.

```
//ambient  
declare var $;  
  
$( '.btn' ).click( function() {  
    console.log( 'CLICKED' )  
})
```

TSC nggak akan memvalidasi *ambient* variabel `$` jadi nggak ada error tapi juga nggak ada *intellisense* & *type-checking*.

Nggak asik dong ...

6.2. Type Definition File

Masih ada jalan biar kita dapat fitur *type-checking* & *intellisense* biarpun kita pake *ambient declaration*. Caranya adalah dengan pake file yang disebut *Type Definition*. File ini berekstensi `.d.ts`.

Di mana kita dapetin file `.d.ts` ? Kita bisa unduh dari repositori publik pake aplikasi namanya [Typings](#)

Kita coba bikin proyek baru di direktori, `extlib/`. Struktur direktorinya kayak begini:

```
extlib/  
  |_ typescript/  
  |_ js/  
  index.html
```

Bikin file `typescript/index.ts`, isinya:

```
//ambient  
declare var $;  
  
$( '.btn' ).click(function() {  
  console.log( 'CLICKED' )  
})
```

Terus bikin juga file `index.html`.

Listing 38. File index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <script
src="//cdnjs.cloudflare.com/ajax/libs/jquery/2.2.3/jquery.js"></scr
ipt>
    <style>
      .btn{
        width:100px;
        text-align: center;
        background: black;
        color:white;
      }
    </style>
  </head>
  <body>

  <div class="btn">
    <p>Click Me</p>
  </div>

  <script
src="//cdnjs.cloudflare.com/ajax/libs/systemjs/0.19.27/system.js"><
/script>
  <script>
    System.config({
      baseUrl:'js',
      defaultJSExtensions:true
    });
    System.import('js/index')
  </script>
</body>
</html>
```

Jalanin `tsc --init` di direktori `typescript/` & ubah isinya jadi begini:

Listing 39. File `typescript/tsconfig.json`

```
{
  "compilerOptions": {
    "module": "system",
    "target": "es5",
    "noImplicitAny": false,
    "sourceMap": false,
    "outDir": "../js"
  },
  "exclude": [
    "node_modules"
  ],
  "files": [
    "index.ts"
  ]
}
```

Coba *compile*. Karena ada `declare var $;` di `index.ts`, kita bisa pake JQuery (\$) tanpa ada masalah.

Terus gimana caranya biar kita dapet *intellisense* & *type-checking* untuk JQuery?

Pertama kita install `typings` dulu.

```
npm install -g typings
```

Buka direktori `typescript/` di terminal & jalanin perintah :

```
typings init
```

Perintah di atas bikin file `typings.json` yang isinya di bawah ini.

```
{
  "version": false,
  "dependencies": {}
}
```

Sekarang coba cari JQuery.


```
typings search jquery
```

Muncul daftar *Type Definition* yang berhubungan sama JQuery kayak begini.

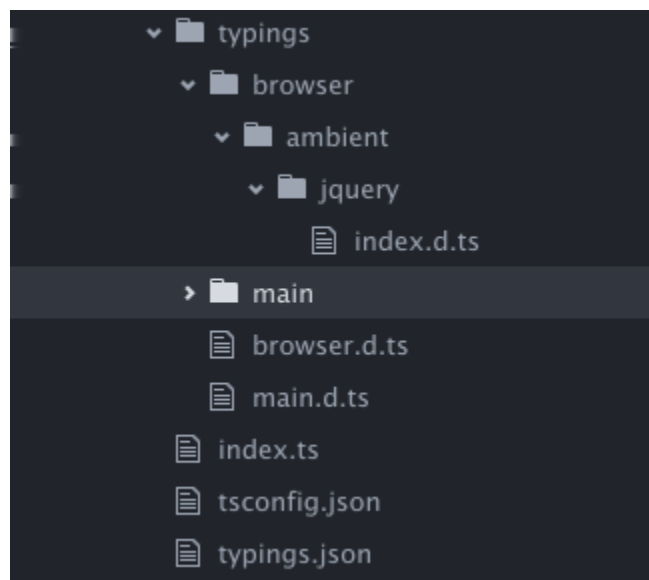
```
$ typings search jquery
Viewing 20 of 104

NAME                SOURCE  HOMEPAGE
chai-jquery         dt      https://github.com/chaijs/chai-jquery
jasmine-jquery      dt      https://github.com/velesin/jasmine-jquery
jquery              dt      http://jquery.com/
jquery-ajax-chain   dt      https://github.com/humana-fragilitas/jQuery-Ajax-Chain/
jquery-backstretch  dt      https://github.com/srobbin/jquery-backstretch
jquery-cropbox       dt      https://github.com/acornejo/jquery-cropbox
jquery-easy-loading  dt      http://carlosbonetti.github.io/jquery-loading/
jquery-fullscreen   dt      https://github.com/kayahr/jquery-fullscreen-plugin
jquery-galleria      dt      https://github.com/aino/galleria
jquery-handsontable  dt      http://handsontable.com
jquery-jsonrpcclient dt      https://github.com/Textalk/jquery.jsonrpcclient.js
jquery-knob          dt      http://anthonyterrien.com/knob/
```

Sekarang instal JQuery.

```
typings install 'dt!jquery' --save --ambient
```

Perintah di atas mengunduh file `jquery.d.ts` untuk JQuery ke direktori `typescript/typings/`. Opsi `--ambient` kita pake karena kita pake jquery sebagai variabel *ambient* bukan modul.



File `typings.json` isinya jadi begini.

```
{
  "version": false,
  "dependencies": {},
  "ambientDependencies": {
    "jquery": "registry:dt/jquery#1.10.0+20160417213236"
  }
}
```



Kalo Anda pake VCS misalnya Git/SVN, abaikan direktori `typings/` cukup `typings.json` aja yang ikut di-commit ke VCS

Selanjutnya, masukan file `typings/browser.d.ts` ke dalam `tsconfig.json`.

Listing 40. File typescript/tsconfig.json

```
{
  "compilerOptions": {
    "module": "system",
    "target": "es5",
    "noImplicitAny": false,
    "sourceMap": false,
    "outDir": "../js"
  },
  "exclude": [
    "node_modules"
  ],
  "files": [
    "typings/browser.d.ts",
    "index.ts"
  ]
}
```



Kenapa bukan `jquery/index.d.ts`? Karena semua file `.d.ts` yang kita install nantinya dimuat dalam `browser.d.ts`, jadi kita nggak perlu masukan setiap file `d.ts` satu-satu ke dalam `tsconfig.json`.

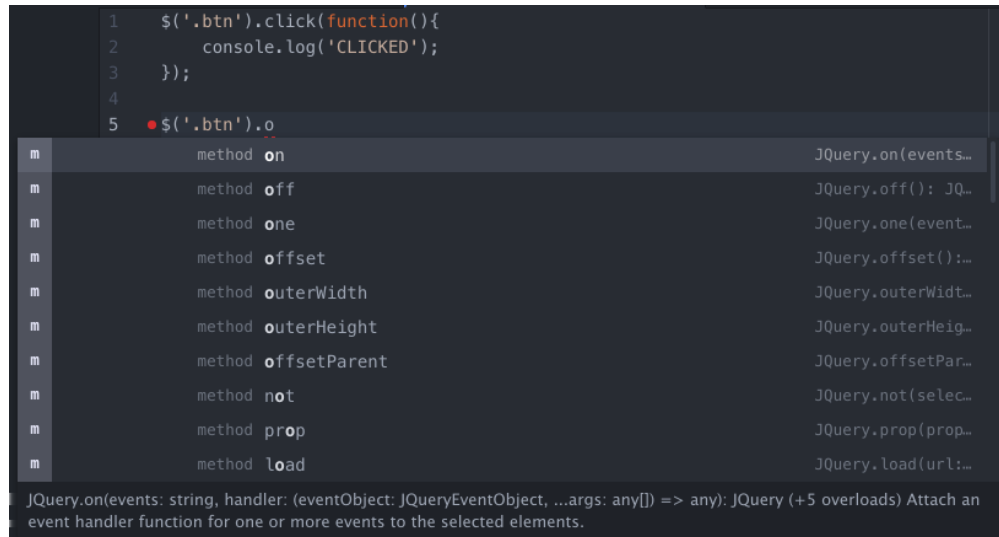


Kenapa `browser.d.ts`? Karena `main.d.ts` itu untuk aplikasi server pake NodeJS

Kita nggak perlu lagi pake `declare var $` di `index.ts`.

```
$( '.btn' ).click(function(){
    console.log('CLICKED')
})
```

Kalo kita *compile* sekarang TSC nggak komplain lagi. Plus, kita dapet *intellisense*.



Gambar 2. Intellisense JQuery di Atom Editor

Dapet *type-checking* juga. Jadi kalo kita salah masukin argumen misalnya begini:

```
$( '.btn' ).ajaxSend('hello');
```

TSC bakal komplain.

```
error TS2345: Argument of type 'string' is not assignable to
parameter of type '(event: JQueryEventObject, jqXHR: JQueryXHR,
ajaxOptions: JQueryAjaxSettings) => any'
```

Bab 7. Penutup

Dari tutorial ini kita udah belajar banyak tentang TypeScript. Kita bisa paham kenapa Google pake TypeScript untuk AngularJS 2.0, bukan bahasa punya mereka sendiri misalnya AtScript atau Google Closure.

Untuk aplikasi yang relatif sederhana, ya nggak perlu pake TypeScript. Tapi kalo kita bikin aplikasi yang kerumitannya medium atau super, TypeScript bisa jadi sangat membantu. Apalagi kalo kita bekerja secara tim, banyak orang yang ikut nulis kode.

Sistem *static-typing* minimal bisa menjamin nggak ada error pada dalam penulisan kode. *Bug* yang ditemuin pada waktu aplikasi berjalan itu lain konteks & nggak bisa dihindari 100%. Tapi kita bisa yakin kode yang ditulis nggak ngaco. Kalopun ada *bug*, kemungkinan besar akibat salah logika, bukan salah tulis kode. Jadi setidaknya TypeScript bisa mengeliminasi satu dari beberapa sumber *bug*.

7.1. Dari Sini Terus ke Mana?

Yang kita bahas dalam tutorial ini baru sebagian aja dari fitur TypeScript. Masih banyak lagi fitur lainnya yang sengaja nggak saya bahas karena:

1. Saya sendiri nggak gitu paham (yay!)
2. Menurut saya nggak terlalu penting (*mixin*, *decorator*, *overload*, dsb)
3. Bisa Anda baca sendiri di [TypeScript Handbook](#) atau di ebooknya Bang Basarat Ali [TypeScript Deep Dive](#) yang bisa dibaca/unduh gratis di Gitbook.

Untuk kelanjutannya saya sendiri mungkin belajar:

1. AngularJS 2.0
2. ReactJS (JSX/TSX)

Jadi, sekian tutorial dari saya. Silakan terusin sendiri belajarnya.

Kalo ada pertanyaan bisa kontak saya di sini 📧 :

- ✉ anggie.bratadinata@gmail.com
- 🐦 [@abratadinata](#)