Mengenal JavaScript

Anggie Bratadinata | *www.masputih.com*

DAFTAR ISI

BAB 1. INTRODUKSI

JavaScript adalah “bahasa web-browser”. Tanpa JavaScript, konten yang ditampilkan dalam browser akan tetap statis, tidak dinamis dan interaktif. Bahasa yang dulu tidak populer ini, dalam beberapa tahun terakhir menjadi salah satu bahasa penting yang wajib dikuasai oleh web developer. Bahkan saat ini JavaScript juga makin populer sebagai bahasa pemrograman server menggunakan program yang disebut *NodeJS* yang berbasis *V8 JavaScript Engine* buatan Google yang juga digunakan oleh browser populer yaitu Google Chrome.

Di sisi browser (client), kita semua pasti pernah mendengar atau menggunakan library seperti JQuery, Dojo, YUI, dan sebagainya yang memungkinkan kita membuat aplikasi/website yang menarik & interaktif tanpa harus bersusahpayah mengatasi perbedaan JavaScript engine yang berbeda antara browser yang satu dengan yang lain.

Dari sekian banyak library, yang paling populer adalah JQuery yang memungkinkan kita menambahkan elemen-elemen atraktif dengan mudah. Sayangnya, masih banyak di antara pengguna JQuery yang bahkan tidak paham JavaScript sama sekali sehingga mereka bergantung 100% pada library ini bahkan untuk menyelesaikan permasalahan yang sangat sederhana sekalipun. Sebagai contoh, dulu saya pernah melihat di forum StackOverflow seseorang bertanya tentang cara membaca “cookies” dengan JavaScript, ironisnya, jawaban paling populer adalah “*pakai Jquery plugin …*”.

Memang tidak ada salahnya mengandalkan JQuery, tetapi kita harus ingat bahwa JQuery dibuat untuk sekedar membantu kita menyelesaikan pekerjaan, bukan untuk menggantikan JavaScript. Bahasa browser adalah JavaScript, bukan JQuery. Untuk membuat website mungkin ini bukan masalah besar. Namun kalau kita membuat aplikasi, pemahaman tentang JavaScript adalah wajib walaupun pada prakteknya kita menggunakan *library* untuk mempermudah pekerjaan kita. Kualitas produk akhir tetap tergantung pada pemahaman kita tentang browser, html, CSS, dan JavaScript.

Menurut saya pribadi, JavaScript adalah bahasa yang mudah dipelajari tetapi tidak mudah dikuasai karena untuk menguasai sebuah bahasa, kita tidak hanya perlu tahu “bagaimana” tetapi juga “kenapa”. Semua orang bisa menulis kode JavaScript, tetapi tidak semua paham kenapa sebuah kode ditulis dengan cara tertentu. Sifat JavaScript yang *dynamic-typing* seringkali mempersulit proses debugging dan mempermudah kita melakukan kesalahan tanpa kita sadari.

Sifat ini juga menyebabkan sampai sekarang belum ada editor atau IDE yang 100% support JavaScript dengan segala fiturnya (*code hint*, *mass refactoring*, *intellisense*, dll). Beda dengan bahasa yang bersifat *static-typing* seperti Java, C#, dan ActionScript 3 di mana setiap IDE bisa mengenali setiap baris kode dari struktur project kita. Ditambah lagi perbedaan JavaScript engine di setiap browser yang kadang membuat frustrasi.

> **Dynamic-typing** : Kita tidak perlu mendeklarasikan tipe data sebuah variabel sebelum menggunakannya. Akibatnya, pada saat program dieksekusi sebuah variabel bisa merujuk pada data bertipe string pada satu waktu, dan data numerik pada waktu lain.
>
> **Static-typing**: Kita wajib mendeklarasikan tipe data sebuah variabel. Selama program berjalan, variabel tersebut hanya bisa merujuk pada data dengan tipe yang telah ditentukan.

Dalam buku ini, kita akan mempelajari dasar-dasar bahasa JavaScript dan mengenal kelebihan serta kekurangannya agar kita memiliki pondasi yang cukup untuk belajar materi yang lebih kompleks. Selain materi yang benar-benar dasar, dalam buku ini saya juga membahas pengenalan topik yang sedikit rumit yaitu:

- Scope-chain
- Closure
- Class & Object
- Inheritance

1.1 ECMAScript 5

JavaScript adalah bahasa pemrograman yang dibuat mengikuti spesifikasi standar yang disebut ECMAScript dan saat ini versi termodern dari ECMAScript adalah versi 5. Sebagian besar browser modern sudah mendukung ECMAScript 5 walaupun tidak ada implementasi yang 100% sama.

Seperti biasa, Microsoft sedikit terlambat dalam implementasinya dan saat ini hanya Internet Explorer 9 ke atas yang bisa disebut kompatibel dengan ECMAScript 5. Untungnya beberapa developer membuat library untuk menutupi kekurangan Internet Explorer versi 7 & 8 sehingga kita tidak perlu bersusah payah menulis kode khusus untuk browser ini.

Library yang ditujukan untuk mengatasi atau menutupi kekurangan sebuah browser dikenal dengan sebutan *Polyfill* atau *Shim*. Berikut ini daftar beberapa polyfill yang bisa kita gunakan: *https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills*

1.2 Testing Environment

Untuk menjalankan contoh-contoh kode dalam buku ini, kita menggunakan *JavaScript console* yang ada di browser Chrome (buka menu `JavaScript Console`) atau Firefox dengan extension Firebug.

> Saya menggunakan Chrome dan *screenshot* yang ada di buku ini saya ambil dari JavaScript console jadi jika Anda menggunakan Firefox+Firebug tampilan console akan berbeda.

Semua contoh kode kita tulis sebagai *embedded script* dalam blok `<script></script>` dalam file html kecuali jika ada keterangan bahwa kode tersebut harus ditulis dalam file `.js` yang terpisah.

Berikut ini contoh file html untuk *embedded script*.

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title></title>
    </head>
    <body>

       <script>
           //kode

       </script>

    </body>
</html>
```

Berikut ini contoh file html untuk memuat file .js eksternal.

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title></title>
    </head>
    <body>

       <script src="main.js"></script>
       <script>
           //kode lain

       </script>
```

```
    </body>
</html>
```

Untuk editor kode, saya menggunakan Microsoft Webmatrix 2[1] yang bisa Anda dapatkan secara gratis. Anda bebas menggunakan editor apa saja namun saya sarankan pilih editor yang mendukung *code-hinting* atau *intellisense* JavaScript.

Di dalam contoh-contoh kode saya sering melakukan *logging* ke JavaScript *console* dengan perintah `console.log()`. Perintah ini bisa di-*support* juga oleh Firebug. Saya kurang tahu apakah *browser* lain juga memiliki fitur yang sama.

[1] http://www.microsoft.com/web/webmatrix/

BAB 2. SINTAKS, VARIABEL & TIPE DATA

2.1 Variabel

Variabel adalah kode yang merujuk pada sebuah lokasi di memori (RAM) di mana sebuah data berada. Variabel tidak berisi data tetapi hanya merupakan referensi atau rujukan sebuah data di memori. Jadi secara teknis, pernyataan "*variabel A bernilai 2*" sebenarnya kurang tepat namun pernyataan ini lebih mudah dipahami dan ditulis daripada "*variabel A merujuk pada data numerik bernilai 2 di memori*".

Satu buah data bisa dirujuk oleh lebih dari satu variabel. Dalam kode berikut, variabel `myCar` dan `wifeCar` sama-sama merujuk pada satu data yaitu objek `Car` yang sama:

```
var myCar = new Car();
var wifeCar = myCar;
```

Ada dua langkah yang diperlukan untuk menggunakan variabel yaitu:

1. Deklarasi
2. Inisialisasi atau definisi

Kita mendeklarasikan variabel dengan menggunakan kata kunci (*keyword*) `var` seperti contoh berikut:

```
var firstName;
var last_name;
var email;
var wheel4;
```

Kita bisa menggunakan kombinasi huruf, angka, dan *underscore* untuk nama variabel tetapi nama variabel tidak boleh diawali dengan angka. Contoh berikut ini tidak valid:

```
var 2wheel;
```

Pada saat variabel dideklarasikan, nilainya adalah `undefined` sampai kita melakukan inisialisasi. Inisialisasi sebuah variabel berarti memberi nilai awal pada variabel tersebut. Inisialisasi dan deklarasi bisa dilakukan dalam satu baris atau baris yang terpisah.

```
//Deklarasi & inisialisasi dalam satu baris
var car_brand = 'Honda'

//Inisialisasi terpisah
var car_brand;
car_brand = 'Honda';
```

Kita juga bisa mendeklarasikan beberapa variabel sekaligus hanya dengan satu *keyword* var di mana antara satu variabel dengan yang lain dipisahkan dengan koma. Seperti contoh berikut:

```
//semua dalam satu baris
var brand = 'honda',type='mpv', numberOfWheels = 4, price;

//baris terpisah (lebih baik daripada satu baris)
var brand = 'honda',
    type='mpv',
    numberOfWheels = 4,
    price;
```

JavaScript memungkinkan kita membuat data dengan dua notasi yaitu literal dan konstruktor (dengan *keyword* `new`). Notasi literal lebih disukai untuk tipe data dasar seperti `Object`, `Array`, `Number`, dan `String` sedangkan untuk notasi konstruktor kita gunakan untuk *custom type* (tipe data yang kita buat sendiri).

```
//notasi literal
var a = [1,2,3];
var obj = {};
var n = 123;

//notasi objek (tidak disarankan)
var a = new Array(1,2,3);
var obj = new Object();
var n = new Number(123);
```

2.2 String

String adalah data yang berisi deretan karakter yang digunakan untuk merepresentasikan sebuah teks. String diawali dan diakhiri dengan tanda kutip ganda atau kutip tunggal.

```
//dengan tanda kutip ganda
var brand = "Honda";
//dengan tanda kutip tunggal
var brand = 'Honda';

//angka yang dilingkupi tanda kutip
//berubah menjadi string (teks) jadi kita tidak
//bisa melakukan operasi matematika
```

```
var n = '123456';

//penjumlahan string dengan number
//menyebabkan number dikonversi menjadi string
//hasilnya adalah penggabungan string
var j = ''+123456;//hasilnya sama dengan string '123456'

//teks yang berisi tanda kutip tunggal harus
//diawali dan diakhiri dengan kutip ganda
var error = "Can't find user";
//atau menggunakan escape character "\"
var error = 'Can\'t find user';
```

2.3 Number

Number adalah representasi data numerik. Dalam JavaScript, data ini secara default berjenis *floating point* (desimal) tetapi kita juga bisa menggunakan bilangan oktal (basis 8) dan heksadesimal (basis 16).

```
var n = 1;
var mass = 1.5;
//oktal berawalan 0
var o = 0377;
//heksadesimal berawalan 0x
var color = 0xFFFFFF;
```

JavaScript memiliki dua tipe data khusus yang berkaitan dengan number yaitu `Infinity` dan `NaN`. `Infinity` adalah data numerik yang nilainya sangat besar (atau sangat kecil) tidak terhingga, melebihi batas yang bisa diproses oleh JavaScript. Infinity bisa bernilai positif atau negatif. `NaN` adalah singkatan dari *Not a Number* yang merupakan nilai ekuivalen dari `undefined` tetapi khusus untuk data numerik.

```
var kecilSekali = -Infinity;
var besarSekali = Infinity;

//perkalian data numerik dengan string
//menghasilkan NaN
var a = 10 * "20";

//operasi numerik yang melibatkan NaN
//akan menghasilkan NaN juga
var b = 10 * 10 + NaN
```

Selain operator matematika standar (*, /, +, -,%), kita juga bisa menggunakan beberapa *shortcut* seperti contoh berikut.

```
var a = 10;
//kalikan a dengan 2 dan simpan dalam variabel a lagi
//sama dengan a = a * 2
```

```
a *= 2;
console.log(a); // 20

var b = 20;
//bagi b dengan 2 dan simpan hasilnya dalam variabel b lagi
//sama dengan b = b / 2
b /= 2;
console.log(b); //10

//jumlahkan a dengan 1 dan simpan hasilnya
//dalam variabel a lagi
//sama dengan a = a + 1 atau ++a
a++;
console.log(a); //21

//kurangi b dengan 1 dan simpan hasilnya
//dalam variabel b lagi
//sama dengan b = b - 1 atau --b
b--;
console.log(b); //9

//ambil sisa pembagian b dengan 2 (modulus) dan simpan
//hasilnya dalam variabel b lagi
//sama dengan b = b % 2
b %= 2;
console.log(b);//1
```

Kita perlu berhati-hati dalam menggunakan operator `--` dan `++` dalam operasi perbandingan karena posisi mereka terhadap variabel yang bersangkutan akan mempengaruhi hasil akhirnya. Berikut ini contohnya.

```
var i = 1;
console.log(i++ == 1); //true
console.log(i);//2

var i = 1;
console.log(++i == 1); //false
console.log(i);//2

var i = 1;
console.log(i-- == 0); //false
console.log(i);//0

var i = 1;
console.log(--i == 0); //true
console.log(i);//0
```

Jika posisi operator `++` dan `--` ada di belakang variabel, maka variabel tersebut akan dibandingkan terlebih dahulu, baru kemudian nilainya dinaikkan (`++`) atau diturunkan (`--`). Kedua operasi ini disebut *post-increment* dan *post-decrement*.

Sebaliknya, jika kedua operator tersebut diletakkan di depan variabel, maka nilainya akan dinaikkan atau diturunkan dulu baru kemudian dibandingkan. Kedua operasi ini di sebut *pre-*

increment dan *pre-decrement*.

2.4 Array

Array adalah struktur data sederhana berisi deretan data (elemen) yang bisa diakses dengan menggunakan nomor indeks atau *key*. Indeks sebuah array dimulai dari nol.

```
//buat array 3 elemen
var a = [1,2,3];

//tampilkan data pada index pertama (0)
console.log(a[0]);//output: 1

//tampilkan data pada index terakhir (2)
console.log(a[2]);//output: 3
```

Array yang menggunakan *key* (string) sebagai identitas, disebut *associative-array*. Untuk mengakses elemennya kita harus menggunakan *key* bukan nomor indeks. *Key* ditulis dalam tanda kurung siku atau diawali dengan tanda titik (*dot-notation*).

```
var info = [];
info['name'] = 'John';
info['age'] = 40;
//dengan dot notation
info.sex = 'male';

console.log(info['sex']);//male
console.log(info.name);//John

//Associative-array tidak mengenal indeks
console.log(info[0]);//undefined
```

2.4.1 Array Function

Array memiliki banyak built-in function untuk memanipulasi elemen, membuat duplikat, dan lain-lain. Beberapa di antaranya bisa kita coba dengan kode berikut.

```
var myArray = [0];
//tambahkan elemen di index 1
myArray[1] = 5;
console.log(myArray);//0,5

//tambahkan elemen baru di belakang
myArray.push(10);
console.log(myArray); //0,5,10

//tambahkan elemen baru di depan
myArray.unshift(100);
console.log(myArray); //100,0,5,10
```

```
//tambahkan elemen baru di depan elemen terakhir
myArray.splice(myArray.length - 1, 0, 300);
console.log(myArray); //100,0,5,300,10

//hapus elemen terakhir
myArray.pop();
console.log(myArray); //100,0,5,300

//hapus elemen pertama
myArray.shift();
console.log(myArray); //0,5,300

//gabungkan dengan array lain
var otherArray = ['a', 'b', 'c'];
myArray = myArray.concat(otherArray);
console.log(myArray); //0,5,300,a,b,c


//buat array baru berisi sebagian elemen myArray
//antara index 0 - terakhir
var partial = myArray.slice(1, myArray.length - 1);
console.log(partial); //5,300,a,b
//myArray tidak berubah
console.log(myArray); //0,5,300,a,b,c

//buat duplikat
var myCopy = myArray.concat();
console.log(myCopy); //0,5,300,a,b,c

//buat string dari array dengan pemisah '|'
var s = myArray.join('|');
console.log(s); //0|5|300|a|b|c
```

2.5 Object

`Object` adalah "mbahnya" semua tipe data dalam JavaScript. Dengan kata lain, semua tipe data adalah turunan dari `Object`. Untuk membuat objek, kita menggunakan notasi literal `{ }`.

```
var car = {};
```

`Objek` memiliki properties dan untuk membuat properties kita bisa menggunakan cara yang sama dengan *associative-array* atau menggunakan notasi objek (*key-value*) seperti contoh berikut.

```
//notasi objek, lebih disukai
//setiap properti dipisahkan oleh koma & key-value dipisahkan
//oleh titik-dua.
//Tidak boleh menggunakan keyword var
var myCar = {
  brand:'Honda',
  year:2011
};
```

```
//cara yg sama dengan sintaks associative-array
var yourCar = {};
yourCar['brand'] = 'Toyota';
yourCar['year'] = 2012;
```

Sama seperti *associative-array*, kita mengakses properti dengan *key* dalam kurung siku atau *dot-notation*.

```
console.log(yourCar.brand);
console.log(yourCar['brand']);
```

2.6 Introspeksi

Introspeksi adalah proses pengecekan sebuah objek yang kita lakukan ketika kita ingin mengetahui apa tipe data sebuah objek dan apa saja variabel dan function yang ada di dalamnya. JavaScript memiliki tiga *built-in function* untuk melakukan introspeksi yaitu:

- `typeof` : memeriksa tipe data sebuah variabel
- `instanceof` : untuk mengecek apakah sebuah data merupakan instance (objek) dari sebuah kelas
- `hasOwnProperty` : untuk mengecek apakah sebuah objek memiliki properti (*key*)

```
var a = {
    name: 'bob',
    die: function () { }
};

console.log(a.hasOwnProperty('name')); //true
console.log(a.hasOwnProperty('die'));//true
console.log(a.hasOwnProperty('kill'));//false

var b = 100;
var c = '100';
var d = function () { };
console.log(typeof a); //object
console.log(typeof b); //number
console.log(typeof c); //string
console.log(typeof d); //function

var myClass = function () {
    var name;

}

var myObject = new myClass();

console.log(myObject instanceof myClass); //true
```

Salah satu kegunaan introspeksi adalah untuk melakukan validasi data. Misalkan kita

membuat function yang hanya menerima data bertipe string, maka kita bisa melakukan validasi menggunakan `typeof` seperti contoh berikut:

```
function validateName(value){
  if(typeof value == 'string){
    //kode yang dieksekusi jika value benar bertipe string
  }
}
```

2.7 Boolean dan Kondisional

Operasi kondisional adalah proses eksekusi kode jika suatu syarat terpenuhi. Jika syarat tidak terpenuhi dan ada kode alternatif, maka kode alternatif itulah yang akan dieksekusi. Untuk melakukan operasi kondisional kita menggunakan data bertipe *Boolean* yaitu data yang hanya bisa bernilai `true` atau `false`.

```
var a = true;
typeof a; //boolean

//sama seperti Number, nilai boolean yang dibungkus
//oleh tanda petik berubah menjadi string
var b = 'true';
typeof b;//string
```

2.7.1 Logika

JavaScript memiliki tiga operator logika yaitu:

- **!** : kebalikan (negasi)
- **&&** : logika AND
- **||** : logika OR

Operator ! akan menghasilkan nilai kebalikan dari data di mana operator tersebut disematkan.

```
var a = true;
console.log(a);//true
console.log(!a);//false

var b = false;
console.log(!b);//true
```

Logika AND (&&) menghasilkan nilai true hanya jika kedua ekspresi di sebelah kiri dan kanan operator tersebut bernilai true.

```
var a = true;
var b = true;
```

```
console.log(a && b);//true

a = false;
console.log(a && b);//false

a = b = false;
console.log(a && b);//false

a = b = true;
var c = false;
console.log(a && b && c);//false
```

Logika OR (||) menghasilkan nilai true jika minimal salah satu dari ekspresi di sebelah kiri dan kanan bernilai true.

```
var a = true;
var b = true;

console.log(a || b);//true

a = false;
console.log(a || b);//true

a = b = false;
console.log(a || b);//false

b = true;
var c = false;
console.log(a || b || c);//true
```

Sebagai kuis, coba perkirakan apa tampilan di console (`true` atau `false`) tanpa menjalankan kode berikut di browser.

```
var a = true;
var b = false;
var c = false;
var d = true;

console.log(a || b && !c && !d)
```

2.7.2 Perbandingan

Berikut ini beberapa operator perbandingan yang tersedia dalam JavaScript berikut contoh kodenya.

Operator	True jika …

==	Data di sisi kiri sama dengan yang di sisi kanan. Sebelum perbandingan dilakukan JavaScript engine akan mencoba melakukan konversi tipe data sehingga kedua operand bertipe sama jika mungkin.
===	Data di sisi kiri sama dengan yang di sisi kanan dan keduanya mengacu pada data yang sama. Perbandingan dilakukan tanpa konversi.
!=	Data di sisi kiri tidak sama dengan yang di sisi kanan setelah konversi tipe data.
!==	Data di sisi kiri tidak sama dengan yang di sisi kanan ATAU keduanya bernilai sama tetapi berbeda tipe data. Operasi ini tanpa konversi tipe data.
>	Data di sisi kiri lebih besar daripada data di sisi kanan
>=	Data di sisi kiri lebih besar daripada atau sama dengan data di sisi kanan
<	Data di sisi kiri kurang dari data di sisi kanan
<=	Data di sisi kiri kurang dari atau sama dengan data di sisi kanan

```
var myClass = function () {
    name: 'bob'
};

var myClass2 = function () {
    name: 'bob'
};

var a = new myClass();
var b = a;
var c = myClass2();

console.log(a == b); //true
console.log(a == c); //false
console.log(a === c); //false
console.log(a != b); //false
console.log(b !== c); //true

var d = '1';
var e = 1;

console.log(d == e); //true karena ada konversi tipe data
console.log(d != e);//false karena ada konversi
console.log(d === e); //false, tidak ada konversi
console.log(d !== e);//true, tidak ada konversi

console.log( 1 > 2 );//false
console.log( 2 > 1 );//true
console.log( 2 >= 1 );//true
console.log( 1 <= 1 );//true

//kasus menarik, NaN tidak sama dengan
```

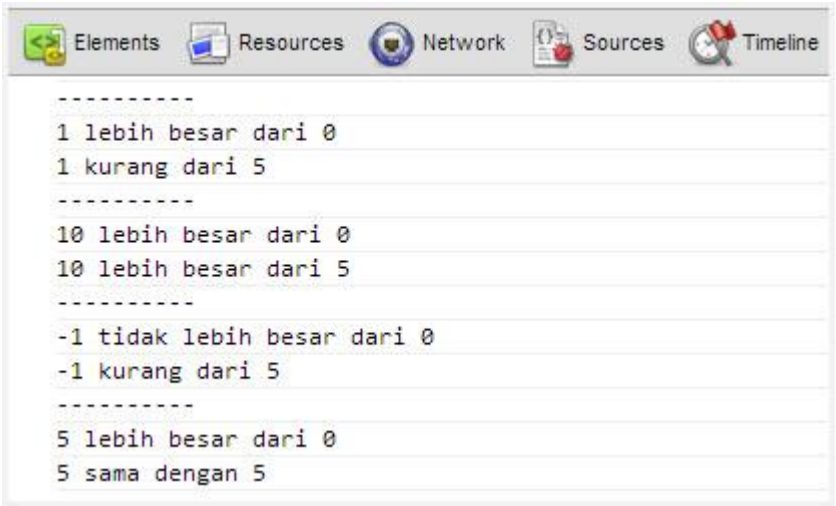
```
//apapun bahkan dirinya sendiri
console.log( NaN == NaN );//false
```

2.7.3 Pencabangan

Kode program yang kita tulis tidak akan terlepas dari pencabangan. Dua pernyataan yang kita gunakan untuk pencabangan adalah if dan if-else.

```
var a = 5;

function check(n) {
    console.log('----------');
    if (n > 0) {
        console.log(n + ' lebih besar dari 0');
    } else {
        console.log(n + ' tidak lebih besar dari 0')
    };

    if (n > a) {
        console.log(n + ' lebih besar dari ' + a);
    } else if (n < a) {
        console.log(n + ' kurang dari ' + a);
    } else {
        console.log(n + ' sama dengan ' + a);
    }
}

check(1);
check(10);
check(-1);
check(5);
```

Hasil eksekusi kode di atas adalah seperti berikut.



Gambar 2-1 Contoh Pencabangan

2.7.4 Ternary Operator

Operator ini digunakan sebagai jalan pintas (*shortcut*) untuk pencabangan sederhana. Sebagai contoh, misalkan kita punya function seperti berikut.

```
function check(c) {

  if (c > 0) {
      console.log('ok');
  } else {
      console.log('not ok');
  };

}
```

Kode di atas bisa ditulis dengan ternary operator seperti berikut. Di mana baris `(c > 0 ) ? true : false` fungsinya sama dengan blok kode `if-else` di atas.

```
function check(c){
  var status = ( c > 0 ) ? 'ok' : 'not ok';
  console.log(status);
}
```

2.7.5 switch-case

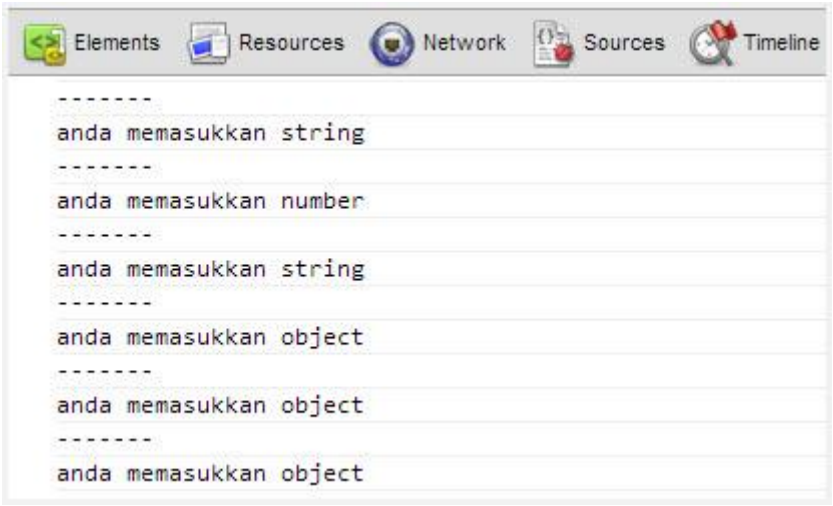
Switch-case kita gunakan untuk pencabangan dengan banyak kondisi sebagai pengganti if-else. Sintaks dasarnya adalah

```
switch(variabel yang diuji){
  case kondisi pertama:
    //kode yang dieksekusi jika kondisi pertama terpenuhi
   break;//keluar dari switch-case
  case kondisi kedua:
    //kode yang dieksekusi jika kondisi kedua terpenuhi
   break
  default:
   //kode yang dieksekusi jika kondisi-kondisi di atas tidak
   //ada yang terpenuhi
}
```

Di bawah ini contoh *switch-case* sederhana di mana variabel `status` berisi pesan berbeda-beda tergantung tipe data yang dikirim ke *function* `checkType`.

```
function checkType(n){
    console.log('-------');
    var status;
    var t = typeof n;
    switch (t) {
```

```
        case 'string':
            status = 'anda memasukkan string';
            break;
        case 'number':
            status = 'anda memasukkan angka';
            break;
        default:
            //jika t bukan string ataupun number
            status = 'anda memasukkan object';
    }

    console.log(status);
}

checkType('a');//anda memasukkan string
checkType(100); //anda memasukkan angka
checkType('100');//anda memasukkan string
checkType(null); //anda memasukkan object
checkType(undefined); //anda memasukkan object
checkType([1, 2, 3]); //anda memasukkan object
```



Gambar 2-2 Contoh Switch-case

2.8 Perulangan

Perulangan adalah pemrosesan sekumpulan data atau eksekusi kode sebanyak beberapa kali. JavaScript memiliki empat operator perulangan yaitu *for*, *for-in*, *while*,dan *do-while*.

2.8.1 for

for kita gunakan untuk melakukan perulangan dengan batas yang kita tentukan. Batas ini harus berupa angka.

```
var a = [];
```

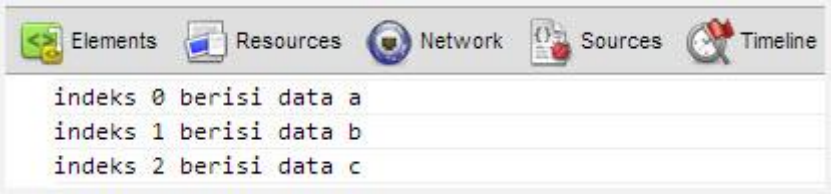
```
for (var i = 0; i < 10; i++) {
    a.push(i);
}

console.log(a);
console.log('panjang a = ' + a.length + " elemen");

//gandakan nilai a dan simpan hasilnya dalam array b
var b = [];
for (var j = 0; j < a.length; j++) {
    b.push(a[j] * 2);
}

console.log(b);

//cari data bernilai 5 dalam array a dan hentikan pencarian
//begitu data ditemukan
var c;
//sintaks alternatif, lebih efisien karena panjang array a
//disimpan dalam variabel len
for (var k = 0,len = a.length; k < len; k++) {
    if (a[k] === 5) {
        c = a[k];
        //data ditemukan, hentikan loop dan keluar dari loop
        break;
    }
}
//perhatikan nilai k terakhir tidak sama dengan
//panjang array a
console.log('data: ' + c + ' ditemukan di indeks ' + k);
```

2.8.2 for-in

for-in hanya bisa digunakan untuk memproses array atau object. Berbeda dengan *for*, *while*, dan *do-while* yang bersifat generik. Berikut ini contoh perulangan for-in atas sebuah array.

```
var list = ['a', 'b', 'c'];
var message;
for (var i in list) {
    message = 'indeks ' + i + ' berisi data ' + list[index];
    console.log(message);
};
```

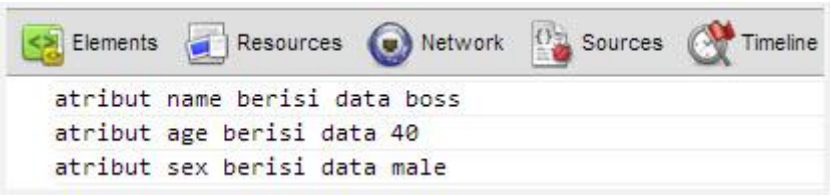
Hasil eksekusi kode di atas adalah seperti gambar di bawah ini.



Gambar 2-3 Output hasil eksekusi for-in atas sebuah array

for-in juga bisa kita gunakan untuk memproses semua atribut (key) sebuah object seperti contoh berikut ini.

```
var person = {
    'name':'boss',
    'age':40,
    'sex':'male'
}
var message;
for (var attr in person) {
  message = 'atribut '+attr+ ' berisi data ' + person[attr];
  console.log(message);
}
```



Gambar 2-4 Hasil eksekusi for-in atas sebuah object

2.8.3 while

while adalah perulangan yang paling sederhana. Perulangan dilakukan selama kondisi untuk berhenti (*break condition*) belum terpenuhi. Dalam eksekusinya, *break condition* diuji terlebih dahulu baru kemudian kode dijalankan. Berikut ini contohnya.

```
var a = [];
var i = 0;

//selama i kurang dari 4
//break condition : i == 4
while (i < 4) {
    //simpan i dalam array
    a.push(i);

    //naikkan i untuk proses selanjutnya
    i++;
};

console.log(a);//[0,1,2,3]
```

Kode di atas juga bisa kita tulis dalam bentuk berikut.

```
var a = [];
var i = 0;
while (i++ < 4){
    a.push(i);
};

console.log(a);//0,1,2,3
```

Jika kita mengganti `i++` dalam contoh di atas dengan `++i` maka kita akan mendapat hasil yang berbeda. Jadi kita perlu berhati-hati dalam menggunakan *while* dengan *shortcut* seperti itu. Ini juga berlaku untuk operasi kondisional seperti yang kita pelajari dalam bab sebelumnya.

Berikut ini contoh kode di atas dengan ekspresi `++i` sebagai pengganti `i++`.

```
var a = [];
var i = 0;
while (++i < 4){
    a.push(i);
};

console.log(a);//1,2,3
```

2.8.4 do-while

do-while sedikit berbeda dengan while tetapi kalau kita tidak paham perbedaannya, kode kita tidak akan bekerja dengan benar. Berikut contoh do-while yang mirip dengan contoh while di atas.

```
var a = [];
var i = 0;

do {
    a.push(i);
    i++;
} while (i < 4);

console.log(a); //[0,1,2,3]
```

Hasil eksekusinya memang sama, tetapi ada perbedaan mendasar antara *do-while* dan *while*. *Do-while* akan memproses blok kode di dalamnya terlebih dahulu, baru kemudian menguji *break condition* sehingga blok kode diproses minimal satu kali. Sebaliknya *while* akan menguji *break condition* dulu baru memproses blok kode dan mungkin saja blok kode tidak pernah diproses. Berikut contoh yang menunjukkan perbedaan keduanya.

```
var b = [];
```

```
var i = 5;
while (i < 4) {
    b.push(i);
    i++;
}
//b tidak memiliki elemen (kosong) karena blok kode tidak
//pernah dieksekusi
console.log(b);//[]

var a = [];
var i = 5;

do {
    a.push(i);
    i++;
} while (i < 4);

//a memiliki satu elemen karena blok kode
//dieksekusi minimal satu kali
console.log(a); //[5]
```

BAB 3. FUNCTION

Dalam JavaScript, *function* adalah *first-class object*. Artinya, *function* bisa digunakan secara mandiri (*standalone*) atau sebagai bagian dari objek atau *function* lain. Secara *default*, *function* yang kita tulis adalah *global function* yang bisa diakses dari kode lain dalam satu *window* yang sama kecuali jika *function* tersebut merupakan bagian dari sebuah object atau function lain.

Global function yang kita buat dalam sebuah file `script1.js` bisa diakses/dieksekusi oleh kode lain yang berada dalam file misalnya `script2.js` dan sebaliknya, jika kedua skrip tersebut dimuat oleh halaman html yang sama.

3.1 Function Sederhana

Setiap *function* terdiri dari dua bagian yaitu *signature* dan *body*. *Signature* sebuah function adalah nama dan parameter sedangkan *body* adalah seluruh kode di antara kurung kurawal.

```
function sum(a,b){ //signature, nama = sum, parameter = a & b
  return a + b;    //body
};

//eksekusi function, tampilkan hasilnya di console
console.log(sum(1,2));//3
```

Karena function juga merupakan sebuah data, kita bisa menulis kode di atas sebagai sebuah variabel yang berisi *function* tanpa nama (anonim) seperti berikut:

```
var sum = function(a,b){
            return a + b;
          };
```

Setiap function memiliki nilai balik (*return value*) yang dikirimkan ke kode yang mengeksekusinya. Kalau kita tidak secara eksplisit mendefinisikan nilai balik sebuah function, JavaScript *engine* akan membuatnya secara otomatis dengan nilai `undefined`. Dua function di bawah memiliki nilai balik yang sama, perbedaannya `fn_A` memiliki nilai balik eksplisit sedangkan `fn_B` implisit.

```
//nilai balik eksplisit
function fn_A(){
  return undefined;
```

```
}

//nilai balik implisit
function fn_B(){
}
```

Sebuah *function* bisa menjadi bagian (*child*) dari *function* lain.

```
function getCircleArea(r){

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

  return 2 * pi_r();
};
```

3.2 Scope Chain

Setiap *function* yang dieksekusi memiliki *scope* atau ruang lingkup yang menentukan variabel dan *function* lain yang bisa diakses oleh *function* tersebut. Sama dengan variabel global, *global function* bekerja dalam konteks *window* dan bisa diakses dari manapun.

```
var n = 10;
var fn_A = function(){
  console.log(this);//window
  console.log(n);//10
}
```

Sebuah *child function* bisa mengakses variabel atau *function* yang dimiliki oleh *parent function*. Sebaliknya *parent function* tidak bisa mengakses variabel atau *function* yang dimiliki oleh *child*.

```
var fn = function() {
    var a = 10;

    function fn_test() {
        var b = 20;
        console.log('#fn_test');
        console.log(a);
    }

    function fn_test2() {
        console.log('#fn_test2');
        //karena function ini tidak punya variabel a
        //yang dipakai adalah variabel a milik root function
        console.log(a);

        function fn_test2_child() {
            console.log('##fn_test2_child');
            console.log('a');
```

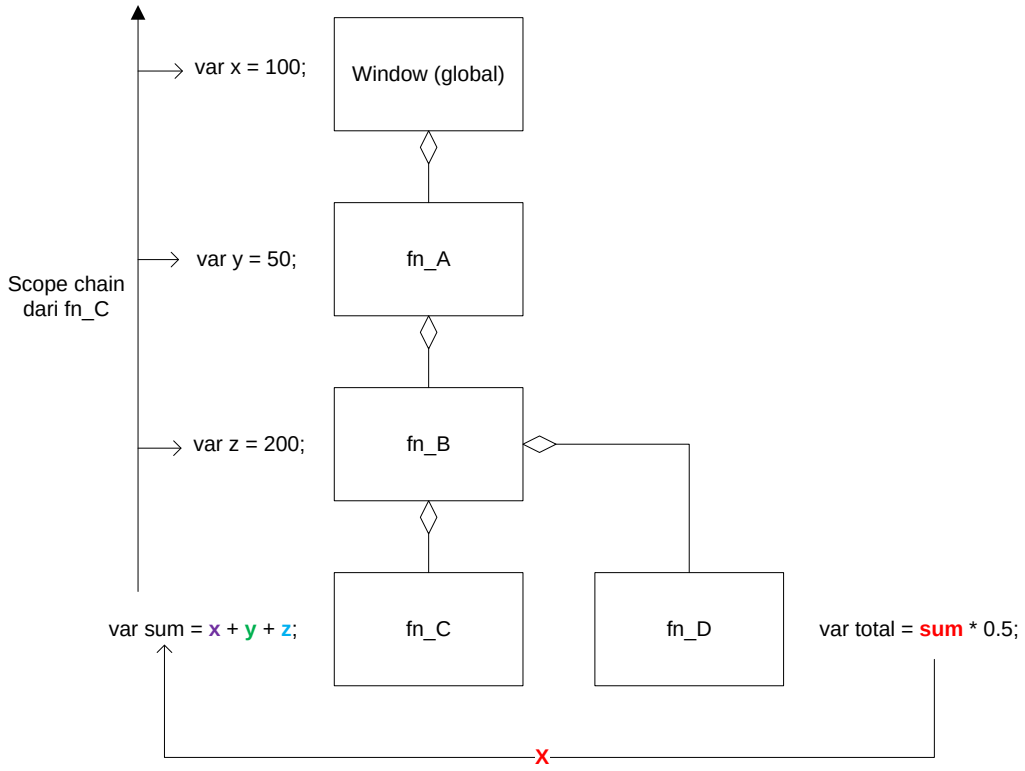
```
            //a bisa diakses karena a berada dalam
            //scope chain
            console.log(a);//10
            console.log('##call fn_test()');
            //fn_test() ada dalam scope chain
            //sehingga bisa diakses juga
            fn_test();
        }

        fn_test2_child();
    }

    fn_test();
    fn_test2();

    console.log('fn');
    //b tidak dikenali oleh root function
    console.log(b);//undefined
}

//jalankan function fn()
fn();
```

Ketika sebuah *function* tidak bisa menemukan variabel atau function di dalam *scope*-nya sendiri, *function* tersebut akan mencari variabel atau *function* di dalam *scope parent function* kemudian *parent* dari *parent function* dan seterusnya sampai *global scope*. Struktur ini disebut *scope chain*. Jika sampai *global scope* variabel atau *function* yang dicari tidak ditemukan maka variabel atau *function* tersebut bernilai *undefined*.

```
var x = 100;

function fn_A() {
    var y = 50;

    function fn_B() {
        var z = 200;

        function fn_C() {
            var sum = x + y + z;
            console.log('sum = ' + sum)
        }

        function fn_D() {
            var total = sum * 0.5;//ERROR!
            console.log('total = ' + total)
        }

        fn_C();
        fn_D();
    }

    fn_B();
}
```

```
fn_A();
```

Dalam contoh kode di atas, `fn_C` tidak memiliki variabel `x`,`y`, dan `z` sehingga function ini akan mencari variabel-variabel tersebut dalam scope chain. Variabel z ditemukan di *parent function* (`fn_B`), variabel y ditemukan di parent dari `fn_B` yaitu `fn_A`, dan variabel `x` ditemukan di *parent* dari `fn_A` (window/global). Variabel `sum` yang ada di *scope* `fn_C` tidak bisa ditemukan oleh `fn_D` karena hubungan antara `fn_D` dan `fn_C` adalah *sibling*, bukan *parent-child*, sehingga `sum` akan bernilai `undefined`. *Scope chain* `fn_C` dalam kode di atas dapat divisualisasikan dengan gambar berikut.



Gambar 3-1 Scope Chain

3.3 Callback

Dalam praktek pemrograman JavaScript, kita akan menemui banyak sekali penggunaan *callback* yaitu sebuah function yang dikirim ke function lain sebagai argumen sehingga bisa dieksekusi oleh function tujuan. Berikut ini contoh *callback*.

```
function add(a, b) {
    var sum = a() + b();
```

```
    console.log(sum);
}

function fn_A() {
    return 1;
}

function fn_B() {
    return 2;
}

//perhatikan : fn_A dan fn_B tanpa
//tanda kurung di belakangnya karena
//kita ingin mengirim function bukan hasil eksekusinya
add(fn_A, fn_B);//output: 3
```

Callback juga bisa diimplementasikan dengan *function* anonim. Contoh di atas bisa kita tulis sebagai berikut:

```
function add(a, b) {
    var sum = a() + b();
    console.log(sum);
}

function fn_A() {
    return 1;
}

//fn_B diganti oleh function anonim
add(fn_A, function(){return 2;});
```

3.4 Self-Invoking Function

Self-invoking function adalah function yang mengeksekusi dirinya sendiri segera pada saat kode function tersebut selesai dibaca oleh JavaScript *engine*. Banyak developer yang lebih suka menyebutnya sebagai *Immediately-invoked Function Execution* (IIFE, dieja: *iffy*) untuk menghindari kerancuan dengan istilah *recursive function* yang juga berarti *function* yang mengeksekusi dirinya sendiri tetapi dalam konteks yang berbeda. Fitur ini adalah fitur unik dalam bahasa JavaScript yang setahu saya tidak ada dalam bahasa lain kecuali mungkin bahasa *functional* seperti Haskell dan Erlang.

Berikut ini contoh dari *self-invoking function*. Begitu halaman HTML dimuat oleh browser, kita akan melihat output log di JavaScript *console*. Perhatikan tambahan tanda kurung, satu sebelum kata *function*, tiga setelah kurung kurawal penutup *body*. Karena function ini otomatis dieksekusi, kita tidak perlu memberinya nama.

```
(function() {
    console.log('self-invoking function');
```

```
}());

//self-invoking function dengan parameter
(function(name) {
    console.log('halo,' + name);
}('boss'));
```

Gambar 3-2 Hasil eksekusi Self-invoking Function

setelah browser di-refresh

3.5 Return Function

Seperti telah kita bahas dalam materi sebelumnya, function dalam JavaScript adalah sebuah tipe data sama seperti `Object`, `String`, `Array`, dan `Number`. Karena itu kita bisa membuat *function* yang memiliki nilai balik berupa sebuah *function*.

```
function fn_A(){
  console.log('Hello');
  return function(){
    console.log('World');
  }
}

var myFunc = fn_A(); //Hello

//myFunc sekarang berisi function yang
//dikirim kembali oleh fn_A
myFunc(); //output: World
```

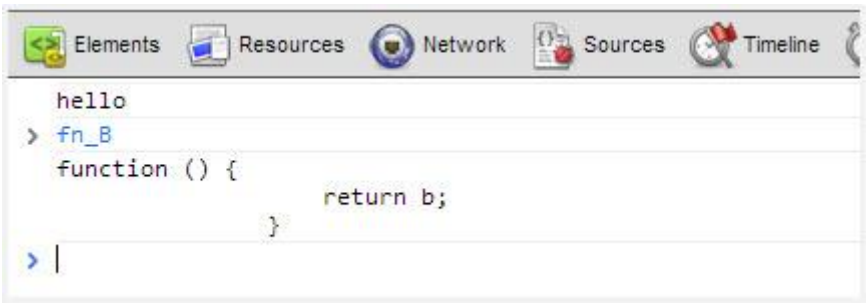
3.6 Closure

Closure adalah konsep yang sedikit sulit dipahami karena tidak ada penjelasan formal dalam spesifikasi JavaScript yang dengan gamblang mendeskripsikannya. Pada dasarnya, *closure* adalah mekanisme untuk mengubah *scope* sebuah variabel atau *function* pada saat eksekusi. Perhatikan contoh berikut:

```
function fn_A(){
    var b = 'hello';
    //closure
    return function(){
        return b;
    }
}
```

```
//simpan hasil eksekusi fn_A sebagai variabel
//fn_B di global space
var fn_B = fn_A();
//fn_B sekarang memiliki akses ke
//variabel b dalam scope fn_A
console.log(fn_B());//hello

console.log(b); //undefined
```

Kalau kita ketikkan perintah `fn_B` (tanpa tanda kurung di belakangnya) dalam konsol, kita bisa melihat struktur `fn_B` seperti di bawah ini yang berarti `fn_B` tetap berada dalam *global scope*. Tentu timbul pertanyaan, kenapa `fn_B` dalam *global scope* memiliki akses ke variabel `b` dalam *scope* `fn_A`. Padahal kalau kita lihat dalam contoh kode di atas, baris terakhir akan menghasilkan pesan `undefined` yang berarti `b` tidak dikenali dalam *global scope*.



Gambar 3-3 Test Closure

Dalam contoh di atas, `fn_B` memiliki akses ke variabel `b` karena *function* `fn_A` menciptakan sebuah *closure* pada saat dieksekusi. *Closure* tersebut memiliki referensi permanen terhadap variabel `b` dan tetap ada bahkan setelah `fn_A` selesai dieksekusi.

Contoh kode di atas juga bisa kita ubah sehingga `fn_A` langsung membuat closure dan menyimpannya sebagai `fn_B` seperti di bawah ini.

```
var fn_B;

function fn_A(){
    var b = 'hello';
    //buat closure & simpan sebagai fn_B
    fn_B = function(){
        return b;
    }
};

fn_A();
console.log(fn_B());//hello
```

Dari contoh di atas, jelas *closure* adalah fitur yang sangat bermanfaat karena fitur ini memungkinkan kita mengatur *scope* sebuah *function* sesuai kebutuhan. Hanya perlu kita ingat,

closure membuat kode kita kurang *readable* karena perubahan *scope* terjadi di belakang layar. Jadi kita perlu hati-hati dalam menggunakannya dan sebaiknya semua pemakaian *closure* diberi komentar yang cukup.

Di bawah ini contoh kode di mana *closure* menimbulkan masalah kalau kita tidak cermat. Misalkan kita ingin membuat sebuah `array` berisi *closure* yang merekam nilai variabel `i` pada saat *closure* tersebut dibuat.

```
function fn(){
    var a = [];
    var i;
    for (i = 0; i < 3; i++) {
        //buat closure & simpan dalam array
        a[i] = function() {
            return i;
        }
    }

    return a;
}

var myArray = fn();
console.log(myArray[0]());//3
console.log(myArray[1]());//3
console.log(myArray[2]());//3
```

Kita lihat hasil eksekusi *closure* dalam `myArray` di tiga baris terakhir semua menghasilkan nilai 3, bukan 0,1,2 seperti yang kita harapkan. Apa yang terjadi di sini adalah akibat *closure* berisi referensi ke variabel `i` dan bukan nilai sebenarnya. Pada saat `fn_A` selesai di eksekusi nilai variabel `i` adalah 3 jadi nilai inilah yang dikembalikan oleh semua *closure* pada saat mereka dieksekusi.

Bug tersebut bisa kita perbaiki dengan menggunakan *closure* tambahan sehingga kode menjadi seperti berikut ini di mana kita mengirim variabel `i` sebagai argumen ke sebuah *self-invoking function* yang kemudian membuat *closure*. *Closure* yang dihasilkan tidak memiliki referensi langsung ke `i` tetapi memiliki referensi ke `x` yang nilainya sama dengan `i` pada saat *self-invoking function* dieksekusi.

```
function fn() {
    var a = [];
    var i;
    for (i = 0; i < 3; i++) {
        //buat closure & simpan dalam array
        a[i] = (function(x){
                    return function(){
                            return x;
                        };
                }(i));//kirim i sebagai argumen
    }

    return a;
```

```
}

var myArray = fn();
console.log(myArray[0]()); //0
console.log(myArray[1]()); //1
console.log(myArray[2]()); //2
```

Contoh pemakaian *closure* yang berikutnya adalah dalam membuat *global function* yang memiliki akses ke sebuah *local variable*. Dengan kata lain, kita menyembunyikan sebuah variabel dan kita ingin variabel tersebut hanya bisa diakses melalui *global function* yang ditentukan, misalnya untuk keperluan validasi. Dalam contoh ini, variabel `name` hanya bisa diubah melalui function `setName()` dan nilainya harus berupa `string`.

```
//global function (getter & setter)
var getName, setName;

(function(){
    //variabel lokal, tidak bisa diakses langsung
    var name = 'boss';
    //closure untuk mengakses local variable
    getName = function(){
        return name;
    };
    setName = function(value){
        //name harus berupa string
        if (typeof value == 'string'){
            name = value;
        }
    };

}());

console.log(getName());//boss

setName('bob');
console.log(getName());//bob

setName(123);
//name tidak berubah karena 123 bukan string
console.log(getName());//bob
```

BAB 4. OOP

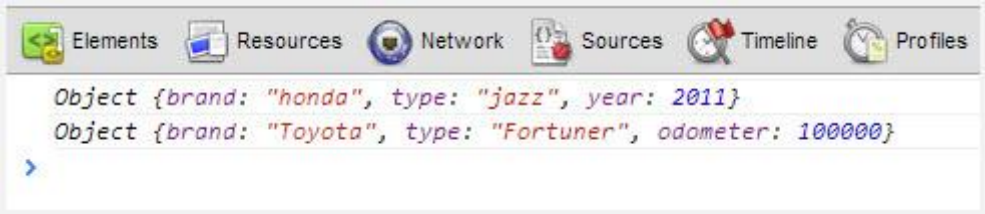
JavaScript adalah bahasa yang unik. Di satu sisi bahasa ini memiliki karakteristik bahasa *functional* di mana *function* adalah *first-class object*. Di sisi lain, JavaScript mendukung konsep OOP yaitu turunan (*inheritance*) dalam bentuk *prototype* yang dikenal dengan istilah *prototypal inheritance*. Dalam bab ini kita akan belajar mengenai implementasi OOP dalam JavaScript.

4.1 Object Properties & Methods

Kita bisa membuat objek baru di luar yang sudah ada (*built-in*) dalam JavaScript dengan menggunakan notasi *object-literal* atau *function*. Sebagai contoh, misalnya kita ingin membuat tipe data Car dengan notasi object-literal. Semua *key* dalam object ini disebut properti dari objek Car. Setelah objek ini dibuat kita bisa mengubah, menambah dan menghapus properti.

```
//buat objek Car dengan properti brand,type,dan year
var Car = {
    brand: 'honda',
    type: 'jazz',
    year: 2011
};

console.log(Car);

//ubah properti dari objek yang sudah kita buat
car.brand = 'Toyota';
car.type = 'Fortuner';

//hapus properti year
delete(car.year);

//tambah properti baru
car.odometer = 100000;

console.log(Car);
```

Gambar ***Gambar 4-1*** menunjukkan hasil eksekusi kode di atas.



Gambar 4-1 Manipulasi Object Literal

Objek bisa memiliki properti berupa *function*. Dalam konteks OOP, properti ini disebut sebagai *method*.

```
var car = {
    brand: 'honda',
    type: 'jazz',
    year: 2011,
    drive: function () {
        console.log('driving');
        //object tidak memiliki scope-chain
        //jadi kita perlu menggunakan 'this'
        //sebagai referensi ke objek ini sendiri
        this.odometer++;
    },
    stop: function () {
        console.log('stopped');
    }
};

car.drive();//driving
car.stop();//stopped
console.log(car.odometer);//100001
```

4.2 Constructor Function

Dari contoh sebelumnya bisa kita lihat bahwa semua properti dari objek yang dibuat dengan notasi object-literal tidak memiliki proteksi yang dikenal dengan nama access-modifier dalam bahasa lain. Hal ini tidak menjadi masalah kalau objek yang kita buat adalah objek yang hanya kita gunakan untuk menyimpan struktur data sederhana. Untuk objek yang lebih kompleks hal ini sangat tidak disarankan, terutama jika objek yang bersangkutan diakses di banyak bagian program yang lain.

Selain itu, dengan menggunakan *object-literal*, kita tidak bisa membuat lebih dari satu objek tanpa melakukan *copy-paste*. Tentu sangat merepotkan jika suatu waktu kita ingin menambah atau menghapus properti, kita harus melakukannya berulang kali sesuai jumlah objek yang ada.

Sebagai solusinya, kita bisa membuat *class* menggunakan *constructor function*. *Class* adalah cetak biru dari sebuah objek. Kita bisa membuat banyak objek dari satu *class* dengan menggunakan keyword `new`. Di bawah ini contoh implementasinya dalam pembuatan class `Car`

dan cara membuat objek dari *class* ini. Untuk membuat properti dan *function* yang bisa diakses dari luar objek (*public*), kita harus menggunakan *keyword* `this`.

```
var Car = function(){

    this.brand = 'Honda';
    this.type = 'jazz';
    this.year = 2011;

    this.drive = function () {
        return 'driving';
    }

    this.stop = function () {
        return 'stopped';
    };

}

//buat objek Car
var car = new Car();
console.log(car.brand); //Honda
console.log(car.type); //jazz
console.log(car.year); //2011
console.log(car.drive()); //driving
console.log(car.stop()); //stopped

//buat objek Car baru
var car2 = new Car();
//ubah propertinya
car2.brand = 'Toyota';
car2.type = 'Fortuner';

console.log(car2.brand); //Toyota
console.log(car2.type); //Fortuner
console.log(car2.year); //2011
console.log(car2.drive()); //driving
console.log(car2.stop()); //stopped
```

Kita lihat *constructor function* sangat mempermudah pembuatan banyak objek dari class yang sama. Proses pembuatan objek dikenal dengan nama instansiasi dan objek yang dihasilkan disebut *instance* dari *class* yang bersangkutan.

Masalahnya, dengan cara di atas, objek yang kita buat tetap tidak memiliki proteksi dalam bentuk *private member* (variabel dan *function*) . Properti objek tetap bisa diubah dengan bebas oleh objek atau kode lain seperti *object literal*. Solusinya adalah :

- Menggunakan keyword var untuk private variable
- Membuat *function* yang merupakan *child-function* bukan properti

Misalkan kita ingin membuat properti `odometer` sebagai *private property* yang hanya bisa dibaca namun tidak bisa diubah langsung dari luar. Nilai `odometer` hanya bisa berubah pada saat

function `drive` dieksekusi. Perubahan kedua adalah penambahan *private function* `brake()` yang juga tidak bisa diakses langsung dari luar. Kita tahu dari pembahasan *function* yang lalu, bahwa *child function* tidak bisa diakses dari luar, jadi kita buat `brake` sebagai *child function* di dalam *constructor*. *Class* `Car` kita ubah seperti berikut.

```
var Car = function () {

    this.brand = 'Honda';
    this.type = 'jazz';
    this.year = 2011;

    //private property, gunakan keyword var, bukan 'this'
    var odometer = 0;

    //public function
    this.drive = function () {
        odometer++;
        return 'driving';
    }

    this.stop = function () {
        //panggil private function
        brake();
        return 'stopped';
    };

    this.getOdo = function () {
        return odometer;
    }

    //private function, dalam bentuk
    //child function
    function brake(){
       console.log('braking');
    }
};

//TESTING

var car = new Car();
console.log(car.brand); //Honda
console.log(car.type); //jazz
console.log(car.year); //2011

console.log(car.getOdo()); //0
console.log(car.drive()); //driving
console.log(car.stop()); //braking, stopped

//setelah drive(), nilai odometer berubah
console.log(car.getOdo()); //1


//coba ubah langsung odometer
car.odometer = 1000;
//odometer tetap 1
console.log(car.getOdo());//1
```

```
//coba panggil brake()
car.brake();//!ERROR undefined
```

4.3 Inheritance

Inheritance atau turunan adalah objek yang dibuat berdasarkan objek lain sehingga objek yang turunan tersebut memiliki karakteristik yang serupa dengan objek asalnya. JavaScript tidak mendukung *inheritance* berdasar class tetapi berdasar objek. *Class* dalam JavaScript berbentuk *function* dan *function* adalah sebuah objek, jadi pada dasarnya kita membuat turunan dari sebuah objek bukan dari class objek tersebut. Ini berbeda dengan bahasa lain misalnya Java, PHP, dan ActionScript 3.

Untuk membuat turunan sebuah objek, misalnya objek B sebagai turunan objek A, kita lakukan dua langkah:

1. Mengeset objek A sebagai `prototype` class B
2. Mereset `constructor` class B.

Di bawah ini kode untuk membuat class baru yaitu Tank sebagai turunan class Car.

```
var Car = function () {

    this.brand = 'Honda';
    this.type = 'jazz';
    this.year = 2011;

    //private
    var odometer = 0;

    this.drive = function () {
        odometer++;
        return 'driving';
    }

    this.stop = function () {
        brake();
        return 'stopped';
    };

    this.getOdo = function () {
        return odometer;
    }
    //private
    function brake() {
        console.log('braking');
    }
};

//buat turunan Car
var Tank = function () {
    this.fire : function(){
      return 'firing';
    }
```

```
};
//1. Set objek Car sebagai prototype
Tank.prototype = new Car();
//2. Reset constructor
Tank.prototype.constructor = Car;

//TESTING
var myTank = new Tank();
myTank.brand = "Abrams"
myTank.type = "Main battle tank";
console.log(myTank.brand);//Abrams
console.log(myTank.type);//Main battle tank
//function yang diwarisi oleh Tank dari Car
console.log(myTank.getOdo());//0
console.log(myTank.drive());//driving
console.log(myTank.stop());//braking,stopped
console.log(myTank.getOdo());//1
//function yang hanya ada di Tank, tidak ada di Car
console.log(myTank.fire());//firing
```

Masih banyak materi mengenai OOP yang perlu kita pelajari misalnya *prototype*, *mixin*, *parasitic inheritance*, dan lain-lain. Tetapi materi-materi tersebut cukup kompleks dan termasuk kategori *advanced* dan saya pikir kurang sesuai untuk buku yang ditujukan untuk programmer JavaScript pemula seperti buku ini. Mungkin lain kali kalau ada kesempatan saya akan menulis buku khusus OOP dengan JavaScript.

BAB 5. PENUTUP

Saya harap banyak yang Anda pelajari dari buku ini. Selain itu, saya juga berharap Anda bisa membagikan buku ini kepada teman-teman Anda supaya kita bisa belajar menjadi programmer handal bersama-sama. Seperti ungkapan dalam bahasa Inggris, "*The more, the merrier*" yang artinya kurang lebih "Lebih rame, lebih asik".

> Kalau Anda punya ide mengenai materi pemrograman lain dalam bahasa JavaScript, ActionScript 3, C, C++, Objective-C, Java, Ruby, Python, atau PHP yang cocok untuk semua level programmer dan menarik untuk dibahas dalam bentuk buku silakan kirim email ke *mas_ab@masputih.com*.

5.1 What next?

Buku ini hanya berisi materi yang menurut saya wajib dipahami oleh semua programmer JavaScript pemula. Untuk menjadi programmer yang handal dan memiliki nilai jual, masih banyak yang harus kita pelajari diantaranya:

- OOP & Design Patterns
- DOM Scripting dengan JavaScript murni (tanpa Jquery, mootools, atau library lainnya)
- AJAX
- JavaScript framework : **Backbone***, **Knockout***, Ember, Angular
- Modular application development menggunakan RequireJS

** Saya berencana menulis buku mengenai Backbone dan Knockout tetapi karena materinya kompleks dan juga butuh banyak waktu, kemungkinan nanti saya jual dalam bentuk ebook atau hardcopy dengan harga murah atau donationware*.

5.2 Referensi

Berikut ini buku-buku dan sumber lain yang saya rekomendasikan untuk Anda baca jika Anda ingin mempelajari JavaScript lebih lanjut.

- **JavaScript : The Good Parts**, *Douglas Crockford, O'Reilly*

- **Object Oriented JavaScript**, *Stoyan Stefanov, Packtpub*
- **High Performance JavaScript**, *Nicholas C. Zakas, O'Reilly*
- **DOM Scripting**, *Jeremy Keith, Friends of Ed (Apress)*
- **Mozilla Developer Network**
    - *https://developer.mozilla.org/en-US/docs/JavaScript/Reference*
- **Microsoft Developer Network**
    - *http://msdn.microsoft.com/en-us/library/ie/yek4tbz0(v=vs.94).aspx*