



Andre Pratama

# JavaScript Uncover

---

Panduan Belajar JavaScript untuk Pemula



DuniaIlkom

# JavaScript Uncover

Panduan Belajar JavaScript untuk Pemula

DuniaIlkom

Buku ini dijual di <http://leanpub.com/javascriptuncover>

Versi ini diterbitkan pada 2017-06-06



Ini adalah sebuah buku [Leanpub](#). Leanpub memberdayakan penulis dan penerbit dengan proses Lean Publishing. [Lean Publishing](#) adalah model penerbitan ebook dalam-proses menggunakan piranti ringan dan sejumlah iterasi untuk memperoleh masukan dari pembaca, menerapkan pivot hingga Anda dapat mewujudkan komposisi buku yang pas dan menarik.

© 2016 - 2017 DuniaIlkom

# Contents

Ucapan Terimakasih . . . . .	i
Tentang Penulis . . . . .	ii
Kata Pengantar . . . . .	iii
Asumsi / Pengetahuan Dasar . . . . .	v
Contoh Kode Program . . . . .	vi
<b>1. Berkenalan Dengan JavaScript . . . . .</b>	<b>1</b>
1.1 Pengertian JavaScript . . . . .	1
1.2 Samakah JavaScript dengan JAVA? . . . . .	3
1.3 JavaScript Sebagai Client Side Programming Language . . . . .	4
1.4 Contoh Penggunaan JavaScript . . . . .	5
1.5 JavaScript dan DOM . . . . .	7
<b>2. Sejarah dan Perkembangan JavaScript . . . . .</b>	<b>9</b>
2.1 Kemunculan Web Browser Netscape Navigator . . . . .	9
2.2 Kelahiran JavaScript . . . . .	10
2.3 JavaScript sebagai Bahasa Pemrograman Web . . . . .	12
2.4 JScript dan VBScript dari Microsoft . . . . .	12
2.5 Standarisasi JavaScript = ECMAScript . . . . .	13
2.6 ECMAScript 2 . . . . .	14
2.7 ECMAScript 3 . . . . .	15
2.8 ECMAScript 4 dan Akhir Browser War Pertama . . . . .	15
2.9 AJAX dan JavaScript Library . . . . .	16
2.10 ECMAScript 5 . . . . .	17
2.11 Reingkarnasi Netscape: Mozilla Firefox . . . . .	17
2.12 Browser War Kedua . . . . .	18
2.13 ECMAScript 6 . . . . .	18
2.14 ECMAScript 7 . . . . .	19
2.15 Versi ECMAScript Berapa yang Harus Saya Pelajari? . . . . .	19
2.16 Mengenal ECMAScript Engine (JavaScript Engine) . . . . .	20
2.17 Perkembangan JavaScript Saat Ini . . . . .	20
<b>3. Menjalankan Kode Program JavaScript . . . . .</b>	<b>22</b>
3.1 Text Editor dan Web Browser . . . . .	22

## CONTENTS

3.2	Hello World dari JavaScript . . . . .	23
3.3	Cara Menginput JavaScript ke dalam HTML . . . . .	25
3.4	Internal JavaScript . . . . .	26
3.5	External JavaScript . . . . .	30
3.6	Atribut async dan defer . . . . .	34
3.7	Atribut defer . . . . .	37
3.8	Web Developer Tools . . . . .	40
3.9	Function console.log() . . . . .	44
3.10	Tag <noscript> . . . . .	45
<b>4.</b>	<b>Aturan Dasar, Variabel dan Konstanta . . . . .</b>	<b>48</b>
4.1	Statement . . . . .	48
4.2	Case Sensitivity . . . . .	49
4.3	Whitespace . . . . .	49
4.4	Baris Komentar . . . . .	51
<b>5.</b>	<b>Variabel dan Konstanta . . . . .</b>	<b>54</b>
5.1	Variabel dalam JavaScript . . . . .	56
5.2	Aturan Penamaan Variabel (Identifier) . . . . .	60
5.3	Gaya Penulisan Variabel: CamelCase . . . . .	61
5.4	Membuat Variabel Tanpa Perintah var . . . . .	61
5.5	Mengenal Strict Mode . . . . .	62
5.6	Membuat Variabel dengan Perintah let . . . . .	66
5.7	Konstanta dalam JavaScript . . . . .	66
5.8	Identifier dan Literal . . . . .	68
<b>6.</b>	<b>Tipe Data JavaScript . . . . .</b>	<b>69</b>
6.1	Perbedaan Tipe Data Primitive dan Object . . . . .	69
6.2	Tipe Data Number . . . . .	70
6.3	Tipe Data String . . . . .	74
6.4	Tipe Data Boolean . . . . .	80
6.5	Tipe Data Null dan Undefined . . . . .	80
6.6	Operator typeof . . . . .	81
6.7	Tipe Data Array . . . . .	82
<b>7.</b>	<b>Operator JavaScript . . . . .</b>	<b>87</b>
7.1	Operator Aritmatika . . . . .	87
7.2	Operator Increment dan Decrement . . . . .	89
7.3	Operator Perbandingan . . . . .	90
7.4	Operator Logika . . . . .	95
7.5	Operator String . . . . .	100
7.6	Operator Bitwise . . . . .	103
7.7	Operator Assignment . . . . .	106
7.8	Operator Spread . . . . .	108
7.9	Urutan Operator dalam JavaScript . . . . .	109
<b>8.</b>	<b>Struktur Logika dan Perulangan . . . . .</b>	<b>111</b>

## CONTENTS

8.1	Struktur Logika IF . . . . .	111
8.2	Struktur Logika IF ELSE . . . . .	114
8.3	Struktur Logika IF ELSE IF . . . . .	115
8.4	Nested IF ELSE . . . . .	116
8.5	Struktur Logika SWITCH . . . . .	120
8.6	Operator Conditional . . . . .	124
8.7	Perulangan FOR . . . . .	125
8.8	Perulangan Bersarang (Nested Loop) . . . . .	130
8.9	Infinity Loop . . . . .	132
8.10	Perintah Break dan Continue . . . . .	133
8.11	Perulangan WHILE . . . . .	135
8.12	Perulangan DO WHILE . . . . .	137
8.13	Menampilkan Element Array Dengan Perulangan . . . . .	139
8.14	Perulangan FOR OF . . . . .	141
<b>9.</b>	<b>Function . . . . .</b>	<b>143</b>
9.1	Pengertian Function . . . . .	143
9.2	Membuat dan Memanggil Function . . . . .	144
9.3	Mengembalikan Nilai Function . . . . .	146
9.4	Argument Function . . . . .	149
9.5	Default Argument . . . . .	153
9.6	Arguments Object . . . . .	156
9.7	Spread Operator untuk Argument . . . . .	159
9.8	Variable Scope . . . . .	161
9.9	Perbedaan Cara Membuat Variabel var vs let . . . . .	168
9.10	JavaScript Hoisting . . . . .	170
9.11	Function: First-class Citizen . . . . .	176
9.12	Arrow Notation . . . . .	179
<b>10.</b>	<b>JavaScript Object . . . . .</b>	<b>182</b>
10.1	Apa itu Object? . . . . .	182
10.2	Format Dasar Object . . . . .	182
10.3	Object Property . . . . .	183
10.4	Menambah Object Property . . . . .	189
10.5	Mengubah Nilai Object Property . . . . .	190
10.6	Object Method . . . . .	191
10.7	Mengubah Object Method . . . . .	193
10.8	Nested Object . . . . .	194
10.9	Object Reference . . . . .	195
10.10	Perulangan FOR IN . . . . .	198
<b>11.</b>	<b>Object Oriented Programming JavaScript . . . . .</b>	<b>205</b>
11.1	Pengertian OOP . . . . .	205
11.2	Pengertian Class dan Object . . . . .	206
11.3	Membuat Class dan Object . . . . .	207
11.4	Class Sebagai “Object Induk” . . . . .	208

## CONTENTS

11.5	Class Property . . . . .	210
11.6	Pengertian this . . . . .	211
11.7	Argument Constructor . . . . .	212
11.8	Class Method . . . . .	214
11.9	Javascript: Prototype Based Language . . . . .	216
11.10	Membuat Object dengan Constructor Functions . . . . .	217
11.11	Membuat Class Method Dengan Object Prototype . . . . .	219
11.12	Object Bawaan JavaScript . . . . .	220
<b>12.</b>	<b>Number Object . . . . .</b>	<b>225</b>
12.1	Cara Mengakses Property dan Method Object . . . . .	225
12.2	Number Object Property . . . . .	227
12.3	Property Number.EPSILON . . . . .	227
12.4	Property Number.MAX_SAFE_INTEGER . . . . .	227
12.5	Property Number.MAX_VALUE . . . . .	227
12.6	Property Number.MIN_SAFE_INTEGER . . . . .	227
12.7	Property Number.MIN_VALUE . . . . .	228
12.8	Property Number.NaN . . . . .	228
12.9	Property Number.NEGATIVE_INFINITY . . . . .	228
12.10	Property Number.POSITIVE_INFINITY . . . . .	228
12.11	Number Object Method . . . . .	228
12.12	Method Number.isNaN() . . . . .	229
12.13	Method Number.isFinite() . . . . .	229
12.14	Method Number.isInteger() dan Number.isSafeInteger() . . . . .	230
12.15	Method Number.parseFloat() . . . . .	231
12.16	Method Number.parseInt() . . . . .	231
12.17	Number Instance Method . . . . .	233
12.18	Method Number.prototype.toExponential() . . . . .	233
12.19	Method Number.prototype.toFixed() . . . . .	234
12.20	Method Number.prototype.toPrecision() . . . . .	235
12.21	Method Number.prototype.toString() . . . . .	237
12.22	Method Number.prototype.toLocaleString() . . . . .	238
12.23	Method Number.prototype.valueOf() . . . . .	241
<b>13.</b>	<b>Math Object . . . . .</b>	<b>242</b>
13.1	Math Object Property . . . . .	242
13.2	Math Object Method . . . . .	243
13.3	Method Math.ceil(x), Math.floor(x), dan Math.round(x) . . . . .	244
13.4	Method Math.random() . . . . .	245
13.5	Method Math.max() dan Math.min() . . . . .	248
13.6	Method Math.abs() . . . . .	249
13.7	Method Math.pow() . . . . .	249
13.8	Method Math.sqrt() . . . . .	250
13.9	Method Math.log(), Math.log10() dan Math.log2() . . . . .	250
13.10	Method Math.sin(), Math.cos() dan Math.tan() . . . . .	250

## CONTENTS

<b>14. String Object . . . . .</b>	<b>252</b>
14.1    String Object Method . . . . .	252
14.2    Method String.fromCharCode() dan String.fromCodePoint() . . . . .	253
14.3    String Instance Property . . . . .	255
14.4    Property String.prototype.length . . . . .	255
14.5    String Instance Method . . . . .	255
14.6    Method String.prototype.toLowerCase() dan String.prototype.toUpperCase() . .	256
14.7    Method String.prototype.charAt() . . . . .	257
14.8    Method String.prototype.charCodeAt() dan String.prototype.codePointAt() . .	258
14.9    Method String.prototype.substr() . . . . .	259
14.10    Method String.prototype.substring() . . . . .	260
14.11    Method String.prototype.slice() . . . . .	261
14.12    Method String.prototype.split() . . . . .	262
14.13    Method String.prototype.trim() . . . . .	264
14.14    Method String.prototype.concat() . . . . .	265
14.15    Method String.prototype.includes() . . . . .	265
14.16    Method String.prototype.startsWith() dan String.prototype.endsWith() . . .	266
14.17    Method String.prototype.repeat() . . . . .	267
14.18    Method String.prototype.toString() dan String.prototype.valueOf() . . . . .	268
14.19    Method String.prototype.indexOf() . . . . .	268
14.20    Method String.prototype.lastIndexOf() . . . . .	270
14.21    Method String.prototype.search() . . . . .	271
14.22    String.prototype.match() . . . . .	272
14.23    Method String.prototype.replace() . . . . .	273
<b>15. Regular Expression Object . . . . .</b>	<b>275</b>
15.1    Pengertian Regular Expression . . . . .	275
15.2    Cara Membuat RegExp Object . . . . .	275
15.3    RegExp Object Method . . . . .	276
15.4    Method RegExp.prototype.test() . . . . .	276
15.5    Method RegExp.prototype.exec() . . . . .	277
15.6    Pola Reguler Expression . . . . .	278
<b>16. Array Object . . . . .</b>	<b>290</b>
16.1    Array Object Method . . . . .	290
16.2    Method Array.isArray() . . . . .	290
16.3    Array Instance Property . . . . .	291
16.4    Property Array.prototype.length . . . . .	291
16.5    Array Instance Method . . . . .	293
16.6    Method Array.prototype.reverse() . . . . .	294
16.7    Method Array.prototype.concat() . . . . .	294
16.8    Method Array.prototype.slice() . . . . .	296
16.9    Method Array.prototype.splice() . . . . .	296
16.10    Method Array.prototype.join() . . . . .	298
16.11    Method Array.prototype.push() dan Array.prototype.pop() . . . . .	298
16.12    Method Array.prototype.unshift() dan Array.prototype.shift() . . . . .	299

## CONTENTS

16.13	Method Array.prototype.toString() dan Array.prototype.toLocaleString() . . . . .	300
16.14	Method Array.prototype.includes() . . . . .	301
16.15	Method Array.prototype.indexOf() . . . . .	302
16.16	Method Array.prototype.forEach() . . . . .	302
16.17	Method Array.prototype.map() . . . . .	307
16.18	Method Array.prototype.filter() . . . . .	310
16.19	Method Array.prototype.every() . . . . .	311
16.20	Method Array.prototype.some() . . . . .	312
16.21	Method Array.prototype.find() dan Array.prototype.findIndex() . . . . .	313
16.22	Method Array.prototype.reduce() dan Array.prototype.reduceRight() . . . . .	315
16.23	Mehtod Array.prototype.sort() . . . . .	317
<b>17.</b>	<b>Date Object . . . . .</b>	<b>320</b>
17.1	Cara Membuat Date Object . . . . .	320
17.2	Method Getter UTC . . . . .	324
17.3	Method Getter Locale . . . . .	326
17.4	Method Setter UTC . . . . .	328
17.5	Method Setter Locale . . . . .	329
17.6	Latihan Program untuk Date Object . . . . .	330
<b>18.</b>	<b>Global Property dan Global Function . . . . .</b>	<b>338</b>
18.1	Global Property . . . . .	338
18.2	Global Function . . . . .	339
18.3	Function eval() . . . . .	339
18.4	Function isFinite() . . . . .	341
18.5	Function isNaN() . . . . .	341
18.6	Function parseInt() . . . . .	342
18.7	Function parseFloat() . . . . .	342
18.8	Function encodeURI() dan encodeURIComponent() . . . . .	343
18.9	Function decodeURI() dan decodeURIComponent() . . . . .	344
<b>19.</b>	<b>DOM (Document Object Model) . . . . .</b>	<b>345</b>
19.1	Materi Tentang JavaScript Sudah Selesai! . . . . .	345
19.2	Pengertian DOM (Document Object Model) . . . . .	346
19.3	Pengertian BOM (Browser Object Model) . . . . .	348
19.4	Document Object . . . . .	351
19.5	Node Object . . . . .	354
19.6	Menelusuri DOM Tree . . . . .	361
19.7	Node Object Property . . . . .	366
19.8	Mengubah Text dari Node Object . . . . .	370
19.9	Node Object Method . . . . .	374
19.10	Method Document.createElement() dan Document.createTextNode() . . . . .	375
19.11	Method Node.appendChild() . . . . .	376
19.12	Method Node.insertBefore() . . . . .	379
19.13	Method Node.replaceChild() . . . . .	380
19.14	Method Node.removeChild() . . . . .	382

## CONTENTS

19.15	Method Node.cloneNode() . . . . .	383
19.16	Method Node.contains() . . . . .	384
19.17	Method Node.hasChildNodes() . . . . .	385
19.18	Case Study: Latihan Node Object . . . . .	385
<b>20.</b>	<b>Document dan Element Object . . . . .</b>	<b>393</b>
20.1	DOM Level . . . . .	393
20.2	Referensi Property dan Method DOM . . . . .	394
20.3	Mencari Element Node . . . . .	395
20.4	Method document.getElementById() . . . . .	396
20.5	Method document.getElementsByTagName() . . . . .	399
20.6	Method document.getElementsByClassName() . . . . .	402
20.7	Method document.querySelector() . . . . .	403
20.8	Method document.querySelectorAll() . . . . .	406
20.9	Element Object . . . . .	408
20.10	Mengubah Konten Element . . . . .	409
20.11	Property Element.innerHTML . . . . .	409
20.12	Property Element.outerHTML . . . . .	413
20.13	Mengubah Atribut Element . . . . .	414
20.14	Method Element.hasAttribute() . . . . .	415
20.15	Method Element.getAttribute() . . . . .	415
20.16	Method Element.setAttribute() . . . . .	416
20.17	Method Element.removeAttribute() . . . . .	417
20.18	Property Element.attributes . . . . .	418
20.19	Mengubah Style Element . . . . .	420
20.20	Property HTMLElement.style . . . . .	421
20.21	Method Window.getComputedStyle() . . . . .	425
20.22	Property Element.className . . . . .	427
20.23	Property Element.classList . . . . .	430
20.24	Method document.write() . . . . .	432
<b>21.</b>	<b>DOM Event . . . . .</b>	<b>435</b>
21.1	Pengertian Event . . . . .	435
21.2	Event Handler dari Atribut HTML . . . . .	436
21.3	Event Handler dari Property Element . . . . .	442
21.4	Event Handler dari Method Element . . . . .	446
21.5	Event Object . . . . .	451
21.6	Event Propagation . . . . .	462
21.7	Menghentikan Event Default . . . . .	467
21.8	Mouse Events . . . . .	469
<b>22.</b>	<b>Form Processing . . . . .</b>	<b>477</b>
22.1	Form Element: Node . . . . .	478
22.2	Form Element: Property . . . . .	483
22.3	Form Element: Method . . . . .	485
22.4	Form Element: Event . . . . .	487

## CONTENTS

22.5	Input Element: Property . . . . .	489
22.6	Input Element: Event . . . . .	495
22.7	Menjalankan Event Element Lain . . . . .	499
22.8	Keyboard Event . . . . .	502
22.9	Input Element: Type Password . . . . .	510
22.10	Textarea Element . . . . .	512
22.11	Input Element: Type Checkbox . . . . .	516
22.12	Input Element: Type Radio . . . . .	520
22.13	Select Element . . . . .	524
22.14	Case Study: Membuat Dropdown Dinamis . . . . .	530
22.15	Form Validation . . . . .	541
22.16	Case Study: Validasi Berbagai Element Form . . . . .	555
<b>23.</b>	<b>BOM (Browser Object Model) . . . . .</b>	<b>563</b>
23.1	Location Object . . . . .	563
23.2	History Object . . . . .	572
23.3	Navigator Object . . . . .	574
23.4	Screen Object . . . . .	580
<b>24.</b>	<b>Window Object . . . . .</b>	<b>582</b>
24.1	Load Event . . . . .	582
24.2	Method Window.alert() . . . . .	586
24.3	Method Window.confirm() . . . . .	587
24.4	Method Window.prompt() . . . . .	590
24.5	Method Window.print() . . . . .	591
24.6	Method Window.setTimeout() . . . . .	592
24.7	Method Window.clearTimeOut() . . . . .	595
24.8	Method Window.setInterval() . . . . .	596
24.9	Method Window.clearInterval() . . . . .	599
24.10	Method Window.open() . . . . .	600
24.11	Method Window.close() . . . . .	606
24.12	Case Study: Mengubah Konten Window . . . . .	607
24.13	Case Study: Mengirim Teks ke Jendela Lain . . . . .	610
<b>25.</b>	<b>AJAX . . . . .</b>	<b>613</b>
25.1	Pengertian AJAX . . . . .	613
25.2	Konsep Penggunaan AJAX . . . . .	614
25.3	XMLHttpRequest Object . . . . .	615
25.4	Menampilkan File Teks - Synchronous AJAX . . . . .	616
25.5	Menampilkan File Teks - Asynchronous AJAX . . . . .	620
25.6	Menampilkan File PHP dengan AJAX . . . . .	621
25.7	Mengirim Data ke File PHP - Metode GET . . . . .	623
25.8	Mengirim Data ke File PHP - Metode POST . . . . .	624
25.9	Case Study: Mengambil Data dari Database MySQL . . . . .	626
<b>26.</b>	<b>Case Study: SlideShow . . . . .</b>	<b>635</b>

## CONTENTS

26.1	Mempersiapkan kode HTML dan CSS . . . . .	636
26.2	Merancang Tombol Next dan Prev . . . . .	638
26.3	Merancang Tombol Start slideshow dan Stop slideshow . . . . .	640
<b>Penutup: JavaScript Uncover . . . . .</b>	<b>642</b>	
jQuery . . . . .	642	
HTML5 API (Application Program Interface) . . . . .	643	
Node.js . . . . .	644	

# Ucapan Terimakasih

Dalam kesempatan ini saya ingin mengucapkan terimakasih kepada Allah S.W.T karena dengan karuniaNya saya masih diberi kesempatan dan kesehatan untuk bisa menulis buku keempat DuniaIlkom: **JavaScript Uncover**.

Selanjutnya kepada keluarga saya yang terus memberi motivasi dan dukungan tiada henti untuk terus mengembangkan DuniaIlkom.

Terakhir kepada rekan-rekan pembaca dan pengunjung setia DuniaIlkom. Terutama bagi yang telah memberikan donasi untuk membeli buku saya sebelumnya: **HTML Uncover**, **CSS Uncover** dan **PHP Uncover**. Karena dari *feedback* dan dukungan rekan-rekan lah saya bisa lanjut menulis eBook **JavaScript Uncover** ini. Terimakasih :)

Padang Panjang, 2016

Penulis

Andre Pratama

[www.duniaIlkom.com](http://www.duniaIlkom.com)

# Tentang Penulis

## Andre Pratama



Saat ini memilih karir sebagai praktisi dan penulis di duniailkom.com. Menamatkan kuliah S1 Ilmu Komputer di Universitas Sumatera Utara pada tahun 2010. Sempat terjun ke dunia kerja melalui ODP (*Officer Development Program*) di Bank BUMN terbesar di Indonesia. Di sela-sela kesibukan, mendirikan situs duniailkom.com pada tahun 2012.

Karena kecintaan akan menulis dan programming, akhirnya memutuskan untuk keluar dari dunia perbankan dan fokus mengelola duniailkom di akhir tahun 2014. Berusaha untuk menjadikan duniailkom sebagai salah satu media belajar programming dan ilmu komputer terbaik di Indonesia.

Berdomisili di kota Padang Panjang, Sumatera Barat, Andre bisa dihubungi melalui email duniailkom di [duniailkom@gmail.com](mailto:duniailkom@gmail.com), facebook: [facebook.com/belajar.duniailkom](https://facebook.com/belajar.duniailkom)<sup>1</sup>, dan twitter: [twitter.com/duniailkom](https://twitter.com/duniailkom)<sup>2</sup>.

---

<sup>1</sup><https://www.facebook.com/belajar.duniailkom>

<sup>2</sup><https://twitter.com/duniailkom>

# Kata Pengantar

JavaScript merupakan bagian dari 5 materi dasar web programming, yakni: HTML, CSS, PHP, MySQL dan JavaScript.

Bersama-sama dengan HTML dan CSS, ketiganya berbagi peran masing-masing. HTML digunakan untuk membuat struktur dan isi dari halaman web (*content*). CSS untuk mempercantik tampilan website (*design*). Sedangkan Javascript berfungsi menangani interaksi (*behavior*). Peran ketiganya nyaris tidak tergantikan oleh bahasa pemrograman lain.

HTML, CSS dan JavaScript sama-sama termasuk ke dalam kelompok “*client side programming language*”, yakni bahasa pemrograman yang dijalankan di sisi client (web browser).

Walaupun begitu, HTML dan CSS belum bisa dikategorikan sebagai “bahasa pemrograman”. Keduanya hanyalah “bahasa kode” dengan aturan-aturan tertentu. Di dalam JavaScript-lah kita akan mempelajari sisi programmingnya.

PHP juga merupakan bahasa pemrograman web, tapi berada di dalam server, sehingga disebut sebagai “*server side programming language*”.

Perkembangan JavaScript yang sangat pesat akhir-akhir ini melahirkan banyak penerapan lain dari JavaScript. Sebagai contoh, Node.js adalah penggunaan JavaScript di sisi server. Dalam buku ini kita hanya fokus membahas penggunaan JavaScript di sisi client (di dalam web browser).

JavaScript hampir selalu digunakan dalam website modern. Mulai dari menampilkan jam, membuat slider, validasi form, animasi, hingga membuat game berbasis web. Fitur baru di HTML5 juga membutuhkan JavaScript, seperti <canvas> yang pernah saya bahas di buku **HTML Uncover**.

Dalam buku **JavaScript Uncover** ini saya akan mengajak anda untuk mempelajari JavaScript. Kita akan mulai dari cara menginput kode JavaScript ke halaman HTML, bagaimana aturan penulisan (*syntax*) JavaScript, mengenal struktur DOM (Document Object Model), mengulas tentang *event*, membuat fitur pengecekan form (form validation), hingga AJAX.

Sebagian besar pembahasan ini dilengkapi dengan contoh studi kasus penerapannya, seperti kalkulator sederhana, slider, form interaktif, dll.

Harus saya akui bahwa JavaScript “sedikit” lebih rumit jika dibandingkan dengan HTML dan CSS. Mudah-mudahan anda bisa mengikuti seluruh materi yang ada. Jika terdapat pembahasan yang kurang jelas, silahkan tanya-tanya melalui email ke duniailkom@gmail.com.

*Semoga buku **JavaScript Uncover** ini bisa menjadi buku pengantar terbaik dalam langkah anda menjadi seorang web programmer. Sampai jumpa di bab terakhir :)*

## **Terimakasih untuk tidak memperbanyak / mengedarkan / mencopy eBook ini**

Menulis sebuah buku hingga ratusan halaman butuh waktu yang tidak sebentar. Belum lagi saya harus berjuang mempelajari referensi yang kebanyakan dalam bahasa inggris. Ini saya lakukan agar pembaca bisa mendapatkan materi yang detail, update, dan berkualitas.

Saya menyadari kekurangan sebuah ebook adalah mudah dicopy-paste dan disebarluaskan. Tapi dengan eBook, harga buku bisa ditekan. Selain tidak perlu mencetak, eBook DuniaIlkom ini bisa di dapat dengan murah, termasuk bagi teman-teman di daerah yang ongkos kirimnya lumayan mahal (jika berbentuk buku fisik).

Atas dasar itulah saya mohon kerjasamanya dari rekan-rekan semua **untuk tidak memperbanyak, menggandakan, atau mengupload ulang buku ini di forum, situs dan media lain**.

Saya juga berharap rekan-rekan tidak memposting materi apapun yang ada di dalam buku ini. Jika ingin sebagai bahan artikel untuk postingan blog/situs, silahkan ambil materi yang ada di website duniaIlkom (jangan yang dari buku).

Apabila rekan-rekan memperoleh buku ini **bukan** dari DuniaIlkom, saya mohon bantuan donasinya untuk membeli versi asli. Donasi pembelian buku ini adalah sumber mata pencarian saya untuk menafkahi keluarga. Lisensi atau hak guna buku ini hanya untuk 1 orang, yakni yang telah membeli langsung ke **duniaIlkom@gmail.com**.

Dengan kualitas yang ditawarkan, harga buku ini cukup terjangkau. Buku ini saya buat dengan waktu yang tidak sebentar, hingga berbulan-bulan, dan kadang sampai tengah malam. Bantuan donasi dari rekan-rekan yang membeli buku secara resmi sangat saya hargai, selain mendapat ilmu yang berkah, ini juga bisa menjadi penyemangat saya untuk terus berkarya dan menghadirkan ebook-ebook programming berkualitas lainnya.

Untuk yang membeli dari DuniaIlkom, saya ucapan banyak terimakasih :)

### **Anda diperbolehkan untuk:**

- Mencetak eBook ini untuk keperluan pribadi dan dibaca sendiri.
- Mencopy eBook ini ke laptop/smartphone/laptop milik sendiri.
- Membuat ringkasan buku untuk digunakan sebagai bahan ajar (bukan keseluruan isi buku).

### **Anda tidak dibolehkan untuk:**

- Mencetak eBook ini untuk dibaca oleh orang lain, walaupun gratis.
- Mencopy eBook ini untuk dijual ulang, maupun dibagikan kepada orang lain dengan gratis.
- Membeli buku ini untuk dibaca bersama-sama (lisensi buku ini hanya untuk 1 orang).
- Mengambil sebagian atau seluruh isi buku untuk di publish ke blog, situs, artikel, dan media publik lain.
- Membagikan eBook ini kepada murid/siswa/mahasiswa (jika digunakan untuk bahan pengajaran).

# Asumsi / Pengetahuan Dasar

Agar bisa mengikuti pembahasan dalam buku **JavaScript Uncover** ini dengan maksimal, saya berasumsi anda sudah memiliki pengetahuan dasar seputar website dan internet, seperti apa itu link dan bagaimana cara menggunakan web browser.

Saya juga mengharapkan anda sudah paham tentang HTML. Termasuk cara menggunakan tag-tag HTML, apa atribut HTML, fungsi dari tag `<a>`, tag `<h1>`, tag `<table>`, hingga cara menjalankan file HTML di web browser.

Pemahaman tentang CSS dan PHP juga sangat membantu, karena terdapat beberapa materi yang merupakan gabungan kode CSS dan PHP dengan JavaScript.

Dalam buku ini, saya akan langsung masuk kepada materi **JavaScript** tanpa membahas ulang HTML, CSS, dan PHP. Jika anda perlu tutorial singkat tentang materi ini, bisa mengunjungi situs duniaIlkom: [Tutorial HTML Dasar<sup>3</sup>](#), [Tutorial CSS Dasar<sup>4</sup>](#), dan [Tutorial PHP Dasar<sup>5</sup>](#).

Contoh-contoh kode program dalam buku ini mungkin terlihat ‘apa-adanya’ tanpa warna, animasi, atau efek tampilan yang menarik. Ini semata-mata agar kode program yang ditulis lebih singkat dan simple. Untuk membuatnya lebih menarik, anda bisa menambahkan kode HTML dan CSS sendiri.

Bagi rekan-rekan yang sudah membaca eBook **HTML Uncover**, **CSS Uncover**, dan **PHP Uncover**, buku **JavaScript Uncover** ini merupakan lanjutan paling ‘pas’ sebagai langkah berikutnya untuk menguasai teknologi pembuatan website.

---

<sup>3</sup><http://www.duniaIlkom.com/tutorial-belajar-html-dasar-untuk-pemula/>

<sup>4</sup><http://www.duniaIlkom.com/tutorial-belajar-css-dasar/>

<sup>5</sup><http://www.duniaIlkom.com/tutorial-belajar-php-dasar-untuk-pemula/>

# Contoh Kode Program

Seluruh contoh kode program yang ada di buku **JavaScript Uncover** bisa di download dari folder sharing *Google Drive* yang saya kirim pada saat pembelian: **belajar\_javascript.zip**.

Sebagaimana yang sudah kita pahami, halaman web lengkap diawali dengan kode HTML seperti `<!DOCTYPE html>` serta tag-tag HTML lain seperti `<head>`, `<title>` dan `<body>`. Namun agar menghemat tempat, dalam buku ini kode pembuka HTML tersebut tidak selalu saya tulis. Di dalam file **belajar\_javascript.zip** saya kembali melengkapi tag-tag ini.

File kode program saya kelompokkan ke dalam folder sesuai dengan penomoran bab: `bab_06_variable`, `bab_07_tipe_data`, `bab_08_operator`, dst. Nama file juga akan disesuaikan berdasarkan nomor urut dan judul pembahasan.

Penomoran baris pada contoh kode program dalam buku ini (*line numbering*) berguna untuk memudahkan pembahasan. Jika anda ingin men-copy kode ini langsung dari eBook pdf, gunakan tombol **ALT + drag** agar nomor-nomor ini tidak ikut terseleksi.

Alternatif yang lebih saya sarankan adalah dengan mengetik ulang seluruh kode program. Tujuannya agar anda lebih cepat paham sekaligus bisa menghafal fungsi dari kode-kode tersebut.



Jika anda mencoba menulis ulang kode program yang ada di buku, dan ternyata tidak jalan, besar kemungkinan terdapat penulisan yang salah.

Di dalam programming, 1 karakter saja yang kurang (apakah itu berupa titik, tanda koma, atau tanda '>'), kode program tidak akan jalan sempurna. Jika ini yang terjadi, silahkan anda samakan dengan file-file dalam folder **belajar\_javascript.zip** ini.

# 1. Berkenalan Dengan JavaScript

JavaScript adalah bahasa pemrograman untuk HTML. Dalam bab pembuka buku JavaScript Uncover ini kita akan berkenalan dengan JavaScript, membahas peranan JavaScript dalam pengembangan web, mengenal apa itu *client side programming language*, serta melihat sekilas apa yang bisa dilakukan dengan JavaScript.

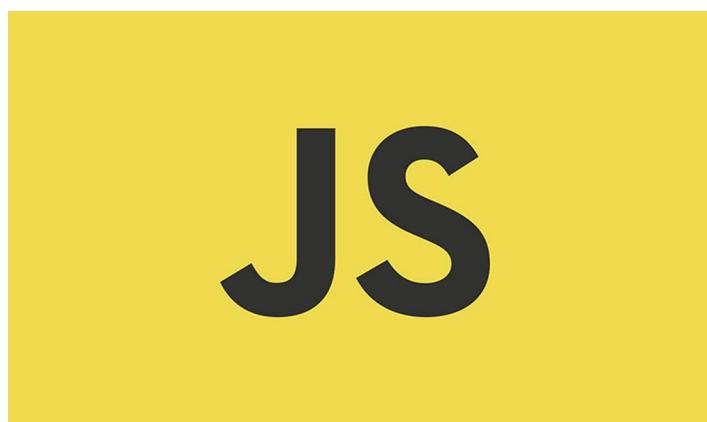
## 1.1 Pengertian JavaScript

Dalam pengertian sederhana, JavaScript adalah bahasa pemrograman web yang digunakan untuk memanipulasi element HTML dan membuat interaksi.

Sebagai contoh, apa yang terjadi ketika sebuah tombol di klik? Bagaimana membuat gambar muncul bergantian secara otomatis (slider), atau bagaimana cara mengubah warna sebuah kotak dari merah ke biru setelah halaman tampil selama 1 menit? Inilah yang bisa kita lakukan dengan JavaScript.

JavaScript menambahkan aspek “pemrograman” ke dalam HTML dan CSS. Misalnya jika sebuah tag `<h1>` di klik, tampilkan isi seluruh artikel yang terdiri dari 10 tag `<p>`, atau jika sebuah gambar di klik, kita bisa mengubah class CSS-nya dari `.normal` menjadi `.warning`.

JavaScript yang akan kita pelajari dalam buku ini termasuk kelompok bahasa pemrograman web berbasis client (*client side programming language*). Artinya, JavaScript di proses di dalam web browser, sama seperti kode HTML dan CSS. Ini berbeda dengan bahasa pemrograman PHP yang diproses di server (*server side programming language*).



Gambar: Unofficial JavaScript logo (wikipedia.org)

Mari kita simak pengertian JavaScript dari [wikipedia](#)<sup>1</sup>:

“JavaScript is a *high-level, dynamic, untyped, and interpreted* programming language. It has been standardized in the ECMAScript language specification. Along-side HTML and CSS, it is one of the three core technologies of World Wide Web content production.”

Terjemahan bebasnya:

“JavaScript adalah bahasa pemrograman yang memiliki ciri-ciri: *tingkat tinggi (high-level), dinamis, tidak bertipe* dan diproses secara *interpreted*. JavaScript menggunakan standar spesifikasi ECMAScript. Bersama-sama dengan HTML dan CSS, JavaScript menjadi salah satu teknologi inti dari pembuatan konten halaman web (World Wide Web).”

Mari kita bahas pengertian yang “sangat teknis” ini.

JavaScript disebut sebagai **bahasa pemrograman tingkat tinggi** atau *high-level* karena kode programnya sudah mirip dengan bahasa inggris sehari-hari.

Dalam bahasa pemrograman *high-level*, kita tidak akan dipusingkan dengan pengaturan dasar seperti alokasi *memory*, *register*, *garbage collection*, dan hal teknis lain yang umumnya ada di dalam bahasa pemrograman tingkat rendah (seperti *bahasa assembly*).

Hampir semua bahasa pemrograman modern sudah termasuk ke dalam high level programming language, seperti PHP, Pascal, C, C++, dan JAVA.

JavaScript memiliki fitur **dinamis, tidak bertipe** dan diproses secara **interpreted**. JavaScript mirip dengan bahasa PHP dimana kita tidak perlu menetapkan sebuah variabel harus bertipe *integer*, *float*, maupun *string*. Setiap variabel di dalam JavaScript bisa diisi dengan tipe data apa saja dan kapan saja sepanjang kode program (bersifat dinamis).

Ini berbeda dengan bahasa pemrograman seperti Pascal, C++ maupun JAVA yang setiap variabelnya hanya bisa diisi dengan tipe data yang sudah ditetapkan, atau dikenal sebagai *Typed Programming Language*. Dalam bahasa pemrograman jenis ini, jika sebuah variabel sudah ditetapkan bertipe integer (angka bulat), maka sepanjang kode program tidak bisa diisi dengan tipe data string (teks) maupun float (angka desimal).

JavaScript menggunakan standar spesifikasi **ECMAScript**, maksudnya yang membuat dan mengembangkan JavaScript adalah **ECMA**. ECMA merupakan singkatan dari *European Computer Manufacturers Association*, yakni sebuah lembaga standarisasi eropa khusus komputer. Jika diibaratkan, ini mirip seperti SNI kalau di Indonesia (Standar Nasional Indonesia). Lebih jauh tentang ECMA dan ECMAScript akan kita pelajari saat membahas tentang sejarah JavaScript pada bab selanjutnya.

Bersama-sama dengan **HTML** dan **CSS**, **JavaScript** menjadi teknologi inti dari pembuatan konten halaman web (World Wide Web). Ketiga teknologi ini memiliki peran masing-masing. **HTML** digunakan untuk membuat struktur dan isi dari halaman web (*content*). **CSS** untuk

---

<sup>1</sup><https://en.wikipedia.org/wiki/JavaScript>

mempercantik tampilan website (*design*). Sedangkan **Javascript** berfungsi menangani interaksi (*behavior*). Sebutan kerennya: “HTML for content, CSS for presentation and JavaScript for behavior”.

Jika pengertian dari wikipedia ini terasa membingungkan, jangan khawatir. Anda cukup pahami bahwa JavaScript adalah *bahasa pemrograman web yang digunakan untuk ‘memprogram’ HTML*. Dengan JavaScript, kita akan membuat halaman website menjadi lebih interaktif. Inilah yang akan saya bahas sepanjang buku ini.

## 1.2 Samakah JavaScript dengan JAVA?

Nama **JavaScript** memang sering membuat bingung programmer pemula. Ini dikarenakan terdapat bahasa pemrograman populer lain yang bernama **JAVA**. Tidak sedikit yang beranggapan bahwa JavaScript adalah versi “ringan” dari **JAVA**, atau JavaScript adalah bahasa pemrograman **JAVA** yang digunakan khusus membuat web.

**JavaScript sepenuhnya berbeda dari JAVA**. Penamaan yang mirip ini tidak lepas dari sejarah dan ide marketing yang dibuat oleh **Netscape Communications**, selaku perusahaan yang pertama kali mengembangkan JavaScript.

Sekitar tahun 1990an, Netscape (yang saat itu sedang bersaing dengan Microsoft) berusaha mendapatkan pangsa pasar web browser, yakni persaingan antara web browser **Netscape Navigator** dengan **Internet Explorer**. Era ini dikenal juga dengan era “**browser war**”.

Supaya lebih “menjual”, Netscape mengubah bahasa pemrograman yang dikembangkannya dari “**LiveScript**” menjadi “**JavaScript**” dengan harapan bisa mengikuti kepopuleran bahasa pemrograman **JAVA** (yang saat itu sedang booming di kalangan programmer).

Walaupun sepenuhnya berbeda, banyak syntax dan aturan penulisan JavaScript mirip dengan **JAVA**, ini karena keduanya sama-sama terinspirasi dari format penulisan bahasa C dan C++. Bahasa pemrograman **PHP** juga merupakan turunan dari C dan C++. Jika sebelumnya anda sudah mempelajari **PHP**, aturan penulisan kode program di JavaScript akan terasa sangat mirip.



ILLUSTRATED BY SEGUETECHNOLOGIES

Gambar: Java vs JavaScript = Ham vs Hamster ([segue.com](http://segue.com))

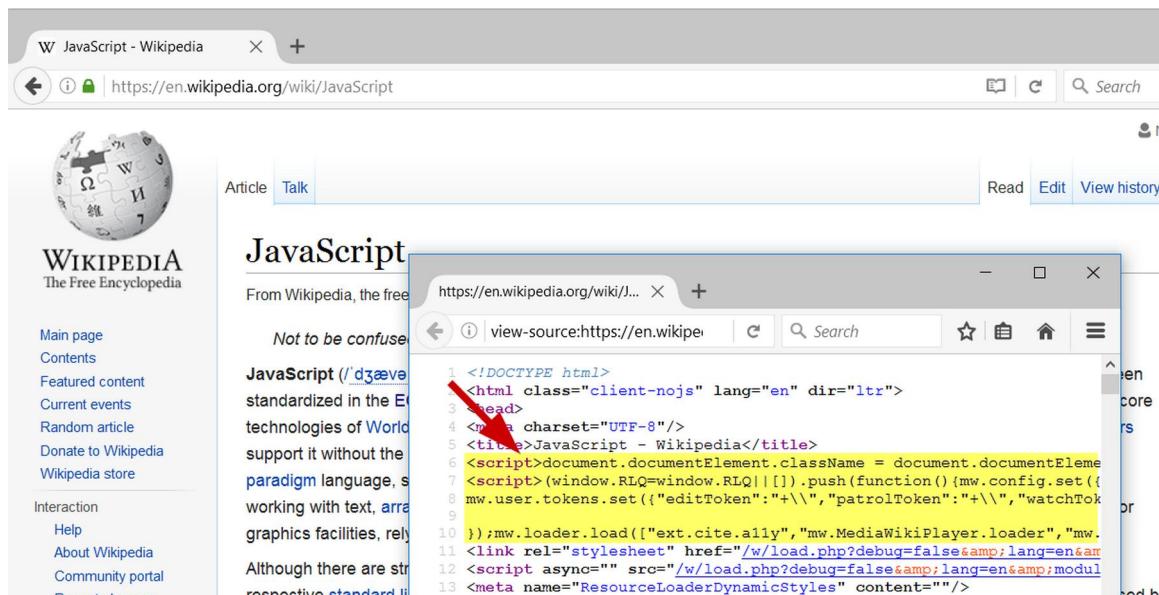
## 1.3 JavaScript Sebagai Client Side Programming Language

Sebagaimana yang telah disinggung sebelumnya, JavaScript termasuk ke dalam bahasa pemrograman web yang diproses di sisi client, atau dikenal dengan istilah **Client Side Programming Language**.

Ini artinya, untuk menjalankan JavaScript kita hanya butuh 2 aplikasi, yakni **text editor** dan **web browser**. Text editor digunakan membuat kode JavaScript, dan kode JavaScript tersebut bisa langsung diakses dari web browser. Caranya sama persis seperti menjalankan file HTML dan CSS.

Ini berbeda dengan bahasa pemrograman **Server Side** seperti PHP. Untuk menjalankan kode program PHP kita harus menggunakan aplikasi seperti **Apache web server** (yang merupakan bagian dari **XAMPP**).

Efek lain dari JavaScript sebagai *client side programming language* adalah, kita bisa melihat kode yang digunakan dari sebuah website (sama seperti HTML dan CSS). Silahkan buka website apa saja, klik kanan lalu pilih **View Page Source**. Anda bisa melihat kode JavaScript yang digunakan (yang jumlahnya bisa mencapai ratusan baris).



Gambar: Kode javascript di situs wikipedia.org

Seiring dengan perkembangan JavaScript yang sangat pesat, saat ini JavaScript tidak hanya berjalan di client, tapi juga bisa berjalan di server (seperti PHP). Contoh penerapan JavaScript di server ini adalah **node.js**<sup>2</sup>. Perusahaan besar seperti **Paypal**<sup>3</sup> dan **LinkedIn**<sup>4</sup>. Juga sudah menggunakan **node.js**.

Karena cakupan JavaScript sudah sangat luas, dalam buku *JavaScript Uncover* ini saya mem-

<sup>2</sup><https://nodejs.org>

<sup>3</sup><https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/>

<sup>4</sup><http://venturebeat.com/2011/08/16/linkedin-node/>

batasi pembahasan JavaScript sebagai client side programming, lebih sempit lagi yakni cara penggunaan JavaScript di dalam halaman web. Kita tidak akan mempelajari penggunaan JavaScript di sisi server.

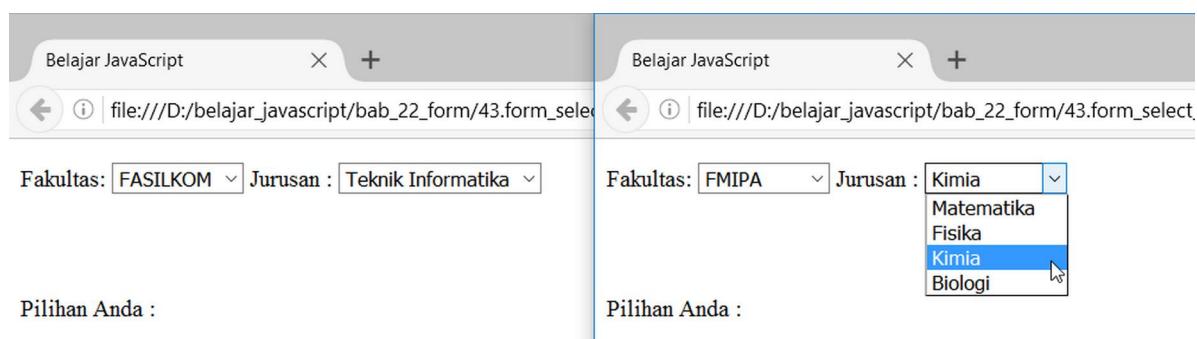
## 1.4 Contoh Penggunaan JavaScript

Setelah berkenalan dengan JavaScript, ada baiknya kita langsung ‘test drive’, seperti apa sih fungsi dari JavaScript ini?

- i** Silahkan anda download file `belajar_javascript.zip` yang disertakan di folder sharing Google Drive, kemudian buka folder `bab_01_berkenalan_dengan_javascript`. Di dalam folder ini terdapat 4 file HTML yang sudah berisi kode JavaScript.



Gambar: Membuat efek animasi pertukaran warna teks



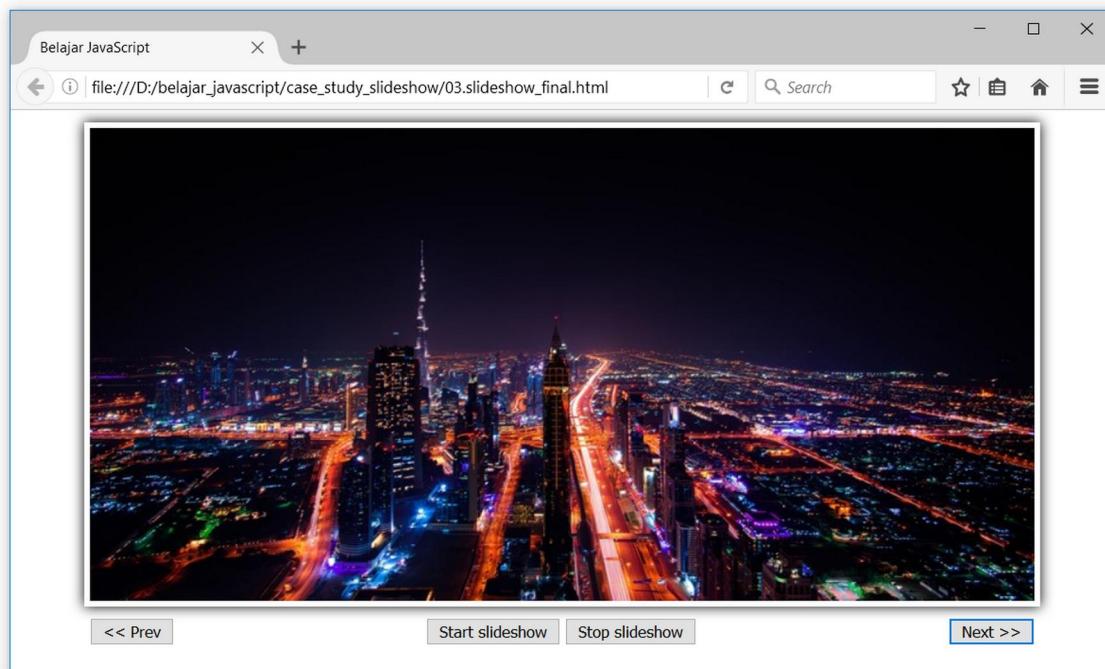
Gambar: Membuat dropdown dinamis (akan bertukar data saat pilihan diganti)

The screenshot shows a registration form titled "Form Pendaftaran". The form includes fields for Username, Password, Re-enter Password, Email, and a checkbox for accepting terms and conditions. Validation messages are displayed in red boxes next to the error fields:

- Username: "tiara[123]" - "Hanya bisa diisi karakter alfanumerik"
- Ulangi Password: "\*\*\*\*\*" - "Password tidak sama"
- Email: "tiara@" - "Format email harus sesuai"
- Checkbox message: "Syarat dan ketentuan harus di setujui"

A blue button labeled "Kirim Data" is at the bottom left, with a cursor pointing at it.

Gambar: Membuat validasi form



Gambar: Membuat slideshow

Seluruh kode program yang digunakan untuk membuat tampilan diatas akan kita bahas sepanjang buku, terutama di bab-bab akhir nanti.

## 1.5 JavaScript dan DOM

Sebenarnya, apa yang akan kita pelajari dalam buku ini terdiri dari 2 kelompok besar: **JavaScript** dan **DOM** (Document Object Model). **JavaScript** adalah bahasa pemrograman, sedangkan **DOM** merupakan objek HTML yang akan kita manipulasi, seperti teks, gambar, form, tombol, title bar web browser, event, dll.

Bagi yang baru pertama kali belajar JavaScript, keduanya tampak seperti satu kesatuan. Untuk apa belajar JavaScript jika tidak berhubungan dengan HTML? Tapi itulah kenyataannya.

Bahasa pemrograman **JavaScript** di kembangkan oleh **ECMA**, sedangkan **DOM** dikembangkan oleh **W3C** (organisasi yang juga membuat standar HTML dan CSS). Bisa dibilang, JavaScript sepenuhnya terpisah dari HTML.

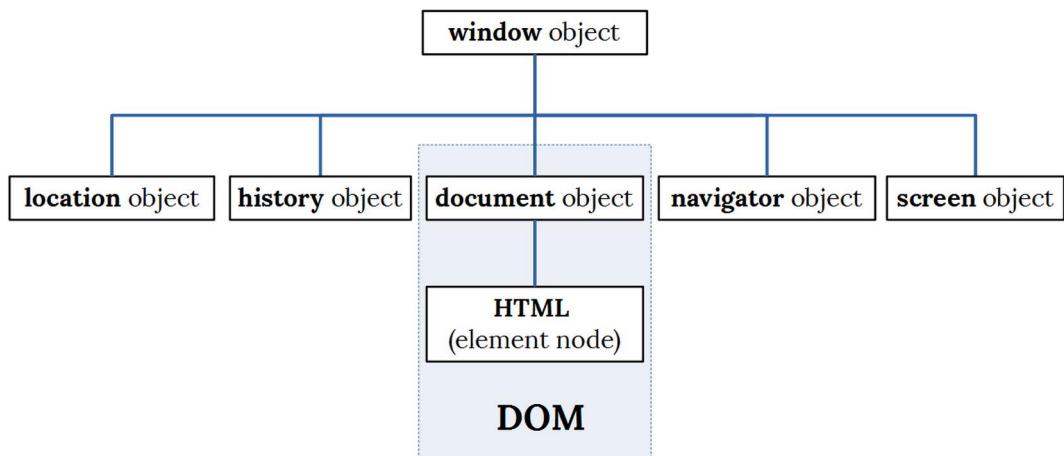
Pada bagian awal buku saya akan fokus ke JavaScript, yakni sisi programmingnya. Disini kita akan membahas dasar-dasar pemrograman JavaScript seperti variabel, tipe data, operator, kondisi if, perulangan, hingga object.

Dengan jujur saya katakan, pembahasan bagian ini akan terasa cukup membosankan, karena kita belum bisa melihat hasil akhir yang sebenarnya di web browser (hanya berupa teks). Bahkan bisa juga dibilang, anda boleh “melupakan” HTML untuk setengah buku ini, kita akan fokus ke sisi programming JavaScript.

Jika anda sudah pernah mempelajari bahasa pemrograman lain seperti PHP atau C++, materi dasar programming ini bukanlah hal baru. Namun JavaScript tetap memiliki perbedaan yang perlu dipelajari.

Setelah mempelajari JavaScript, barulah kita masuk ke **DOM**. Disinilah JavaScript digunakan untuk mengubah total tampilan halaman web. Materi ini sangat menarik dan kita akan melakukan banyak praktek latihan.

Jadi, jika anda merasa jemu dengan pembahasan dari bab 2 hingga 18, tahan dulu! Paksaan untuk terus mempelajarinya. Dengan pemahaman JavaScript yang cukup, kita memiliki pondasi yang kuat untuk memanipulasi objek HTML yang nantinya diakses lewat DOM mulai dari bab 19 hingga akhir buku.



Gambar: Diagram DOM yang merupakan bagian dari BOM (Browser Object Model)

---

Dalam bab pertama buku **JavaScript Uncover** ini kita telah membahas pengertian JavaScript, yakni sebuah bahasa pemrograman berbasis client yang digunakan untuk memanipulasi halaman web. Selain itu kita juga telah melihat beberapa contoh yang bisa dibuat dari JavaScript.

Agar lebih akrab dengan JavaScript, dalam bab selanjutnya saya akan mengajak anda mempelajari tentang sejarah dan perkembangan JavaScript. Bagaimana JavaScript dikembangkan pertama kali hingga kenapa ada istilah **ECMAScript**.

## 2. Sejarah dan Perkembangan JavaScript

Sejarah dari sebuah teknologi selalu menarik untuk diikuti, mulai dari berdirinya sebuah perusahaan, pengembangan berbagai fitur, munculnya pesaing, trik bisnis yang digunakan, hingga kemunduran perusahaan tersebut.

Jika berbicara tentang JavaScript, perusahaan yang dimaksud adalah **Netscape Communications**. Disinilah JavaScript “lahir” dan dibesarkan.

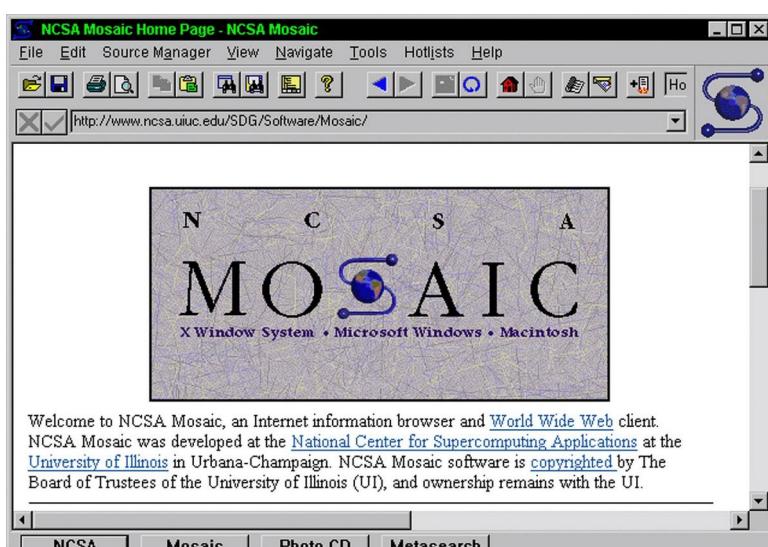
Perkembangan JavaScript juga diiringi sebuah era yang dinamakan “*browser war*”. Perang web browser antara **Internet Explorer** buatan Microsoft dengan **Netscape Navigator** buatan Netscape Communication.



Bab ini lebih ke pengetahuan umum. Dan seperti yang anda lihat, saya menulis sejarah JavaScript dengan cukup detail. Anda boleh melewati bab ini jika sudah tidak sabar ingin menulis kode JavaScript.

### 2.1 Kemunculan Web Browser Netscape Navigator

Sejak HTML dikembangkan pertama kali oleh Tim Bernes-Lee pada tahun 1991, muncul berbagai aplikasi untuk menampilkan konten HTML, atau yang kita kenal sebagai *web browser*. Saat itu, kebanyakan web browser masih berbasis teks yang mirip seperti DOS, jauh dari tampilan grafis yang kita gunakan seperti saat ini.



Gambar: Tampilan web browser NCSA Mosaic, sumber: mystatesman.com

Di tahun 1993, *National Center for Supercomputing Applications (NCSA)*, sebuah unit dari University of Illinois, mengembangkan **NCSA Mosaic**, web browser grafis pertama yang cukup populer. NCSA Mosaic ini menjadi salah satu aplikasi yang revolusioner dalam sejarah perkembangan World Wide Web.

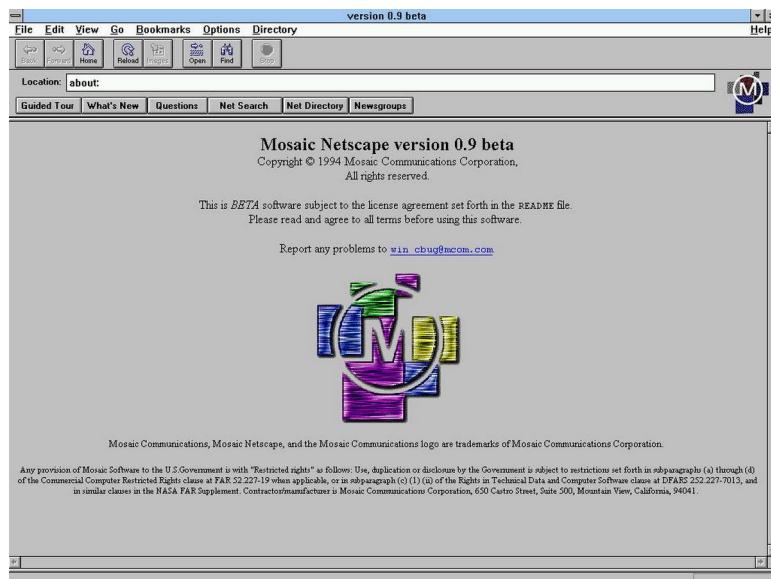
Setahun setelahnya, pada 1994 berdiri sebuah perusahaan bernama **Mosaic Communications**. Perusahaan ini bermarkas di Mountain View, California yang beranggotakan banyak programmer dari NCSA Mosaic, termasuk sang pendiri **Marc Andreessen** yang dulunya pemimpin proyek **NCSA Mosaic**.

Walaupun memakai nama “mosaic”, Mosaic Communications bukanlah bagian dari NCSA Mosaic. Perusahaan ini didirikan untuk mengembangkan web browser komersil, yang kelak diberi nama **Mosaic Netscape**.

Programmer internal perusahaan menggunakan codename “**Mozilla**” untuk menyebut web browser Mosaic Netscape, istilah ini merupakan singkatan dari “*Mosaic killer*”. Suatu indikasi dan tekad untuk mengalahkan NCSA Mosaic dan menjadi web browser nomor 1 di dunia.

Versi awal dari web browser ini, yakni **Mosaic Netscape 0.9** dirilis akhir 1994. Dalam waktu yang relatif singkat, yakni 4 bulan saja, Mosaic Netscape sudah menguasai 3/4 pangsa pasar web browser.

Untuk menghindari masalah merk dagang dengan NCSA, web browser ini berubah nama menjadi **Netscape Navigator**, Perusahaan Mosaic Communications juga berubah nama menjadi **Netscape Communications**.



Gambar: Tampilan web browser **Mosaic Netscape 0.9**, sumber: netscape.exp-soft.de

## 2.2 Kelahiran JavaScript

**Netscape Communications** menyadari bahwa web seharusnya lebih dinamis. Marc Andreessen percaya bahwa HTML butuh sebuah bahasa pemrograman tambahan untuk menyatukan berbagai komponen web seperti gambar, plugin, maupun link.

Bahasa pemrograman ini haruslah sederhana dan bisa dipelajari dalam waktu singkat, terutama bagi web designer yang tidak terlalu berpengalaman dengan coding. Bahasa baru ini juga harus bisa ditulis secara langsung di web browser, seperti layaknya HTML.



Gambar: Marc Andreessen, pendiri dari Netscape Communications

Pada tahun **1995**, Netscape merekrut **Brendan Eich**, seorang programmer yang saat itu bekerja di **MicroUnity Systems Engineering**. Brendan Eich diminta mengimplementasikan bahasa pemrograman “**Scheme**” ke dalam web browser Netscape. Saat itu bahasa pemrograman Scheme cukup populer dan banyak digunakan oleh kalangan akademisi.

Pada waktu yang bersamaan, Netscape Communications membuat kerjasama dengan **Sun Microsystems** dengan tujuan mengimplementasikan bahasa pemrograman **JAVA** ke dalam Netscape Navigator. Ini diperlukan agar bisa berkompetisi dengan web browser **Internet Explorer** buatan Microsoft. Netscape dan Sun ingin menambahkan Java ke dalam web browser.

Sampai disini, terdapat 2 bahasa pemrograman yang akan dimasukkan ke dalam Netscape: JAVA atau bahasa pemrograman baru dari Brendan Eich. Tidak mau kalah cepat, Brendan Eich membuat prototype bahasa baru ini dalam 10 hari. Pada **Mei 1995**, bahasa pemrograman “**Mocha**” lahir. Bahasa pemrograman inilah yang diputuskan untuk digunakan Netscape. Nama *Mocha* sendiri dipilih oleh Marc Andreessen.

Bahasa pemograman **Mocha** dirilis pertama kali ke dalam versi beta **Netscape Navigator 2.0** di bulan September 1995, tetapi dengan menggunakan nama baru: **LiveScript**. Kata ‘live’ dari LiveScript bisa bermakna bahwa bahasa ini akan menambah efek dinamis ke dalam HTML.

Umur dari **LiveScript** ternyata tidak lama. 3 bulan kemudian, tepatnya Desember 1995, hadir Netscape Navigator 2.0 beta 3 dengan sebuah bahasa baru: **JavaScript**. Sebenarnya ini bukanlah bahasa pemrograman baru, tapi perubahan nama dari LiveScript.

Nama **JavaScript** dipilih agar Netscape bisa ‘nompang tenar’ dari bahasa pemrograman JAVA milik Sun Microsystems, yang pada masa itu sangat populer di kalangan programmer. Netscape mendapat izin resmi menggunakan awalan “Java” dari Sun Microsystems.

Pilihan nama **JavaScript** inilah yang menjadi sumber kebingungan programmer hingga saat ini. Banyak yang menyangka JavaScript merupakan versi sederhana dari JAVA, atau implementasi bahasa JAVA ke web browser.



Gambar: Brendan Eich, creator dari bahasa pemrograman Mocha a.k.a LiveScript a.k.a JavaScript

## 2.3 JavaScript sebagai Bahasa Pemrograman Web

Dibalik itu semua, **JavaScript** segera mendapat tempat sebagai bahasa pemrograman yang membuat website menjadi lebih hidup. Programmer web bisa mengubah sebuah gambar ketika di klik oleh pengunjung. Proses validasi form bisa dilakukan di sebelum form dikirim ke web server.

JavaScript juga relatif mudah dipelajari, terutama jika dibandingkan dengan bahasa JAVA yang cukup kompleks. Sifatnya yang *typeless programming language*, berjalan di web browser (tidak butuh *compiler*), mudah di copy-paste dari satu halaman ke halaman lain membuat JavaScript menjadi sahabat web designer dan programmer pemula.

Di sisi lain, programmer professional ada yang menganggap JavaScript sebagai bahasa pemrograman ‘sederhana’, terutama karena tidak tersedianya aplikasi editor yang mumpuni, banyaknya bug, masalah kamanan, hingga kode program yang mudah di-copy paste.

Kekurangan JavaScript ini diperparah dengan masalah kompatibilitas, terutama dengan web browser saingan Netscape: **Internet Explorer**.

## 2.4 JScript dan VBScript dari Microsoft

Melihat kesuksesan JavaScript, Microsoft tidak mau ketinggalan dan segera membuat bahasa serupa untuk Internet Explorer. Tentu saja mereka tidak bisa menggunakan nama JavaScript karena masalah hak paten dengan Sun Microsystems.

Microsoft melakukan ‘*reverse-engineered*’ JavaScript, yakni membuat bahasa pemrograman baru dengan fitur mirip JavaScript (tapi tidak mencontek kode program aslinya). Bahasa baru ini diberi nama **JScript** yang dirilis pada **Internet Explorer 3.0** di Augustus 1996.

Seakan tidak puas dengan JScript, Microsoft juga mengembangkan bahasa pemrograman **VB-Script** (*Visual Basic Scripting Edition*). Bahasa VBScript sebenarnya tidak hanya untuk web programming, tapi juga digunakan dalam aplikasi lain yang dikembangkan Microsoft.

Seiring dengan meningkatnya popularitas Internet Explorer, Microsoft menambahkan berbagai fitur baru ke dalam JScript. Netscape pun tidak mau kalah dan terus mengembangkan JavaScript.

Era ini dikenal juga dengan “**the first browser war**”, perang web browser pertama antara **Internet Explorer** buatan Microsoft vs **Netscape Navigator** buatan Netscape Communications. Kedua perusahaan saling berlomba berebut pangsa pasar web browser.

Korban dari persaingan ini adalah web programmer. Masing-masing web browser memiliki fitur dan perbedaan, sehingga perlu usaha extra untuk membuat kode program yang tampil sempurna baik di IE maupun Netscape. Agar tidak terlalu pusing, banyak programmer memilih salah satu saja dan menampilkan logo: “*best viewed in Netscape*” atau “*best viewed in Internet Explorer*”.

## 2.5 Standarisasi JavaScript = ECMAScript

Supaya permasalahan **JScript vs JavaScript** tidak berlarut-larut, Netscape berinisiatif mengajukan sebuah standar mengenai bahasa pemrograman client side ini. Tujuannya agar JScript dan JavaScript mengikuti sebuah aturan yang baku. Harapannya web programmer tidak lagi dipusingkan terkait perbedaan implementasi JavaScript dari IE dengan Netscape.

Netscape memilih badan standarisasi **ECMA International** sebagai pihak yang akan mengembangkan bahasa pemrograman ini. Karena nama “JavaScript” terikat *trademark* dengan Sun, harus dipilih nama baru, yang akhirnya dipilih nama **ECMAScript** (gabungan dari ECMA dan JavaScript).

Untuk membuat standar **ECMAScript**, dibentuklah **Technical Committee 39** (TC39) yang beranggotakan programmer dari berbagai perusahaan internet pada saat itu, seperti Netscape, Sun, Microsoft, Borland, NOMBAS serta beberapa perusahaan lain yang tertarik dengan perkembangan JavaScript.



Pada awal 2013, anggota dari Technical Committee 39 ECMA berkisar antara 15 sampai 25 orang dari berbagai perusahaan teknologi, termasuk Microsoft, Mozilla, dan Google.

Standar pertama dari ECMAScript adalah **ECMA-262 standard** yang dikeluarkan pada bulan Juni 1997. Standar ini berisi aturan dan panduan yang harus diikuti oleh siapapun yang ingin mengimplementasikan bahasa pemrograman seperti JavaScript.

Sebagai contoh, jika anda berencana membuat sebuah web browser baru, ada baiknya mengikuti standar ECMAScript agar web browser tersebut mendukung penuh kode program JavaScript yang digunakan di web browser lain.

**Ecma International<sup>a</sup>** merupakan badan standarisasi khusus teknologi komputer dan telekomunikasi. Kalau di Indonesia bisa disamakan dengan SNI (Standar Nasional Indonesia).

Pada awalnya ECMA berasal dari singkatan *European Computer Manufacturers Association*, yakni badan standarisasi komputer untuk wilayah eropa. Karena cakupannya tidak lagi hanya di Eropa, pada tahun 1994 berubah nama menjadi **ECMA International**.

Standar lain yang juga dibuat di ECMA termasuk bahasa pemrograman C++, C# dan Office Open XML.

Ecma International ini hanya salah satu dari badan standarisasi, jika anda sering mengikuti perkembangan teknologi, mungkin sering mendengar istilah IEEE dan ISO, keduanya juga merupakan badan standarisasi internasional.

<sup>a</sup><http://www.ecma-international.org>

## 2.6 ECMAScript 2

Sebagaimana nasib standarisasi **HTML** dari **W3C**, standarisasi **ECMAScript** juga jauh dari harapan. Microsoft dan Netscape memang mendukung standar ini, namun mereka tetap menambahkan fitur non standar di web browser masing-masing.

Web programmer pun tetap dibuat bingung. Sebagian ada yang bersusah payah melakukan usaha extra membuat kode program yang bisa jalan di kedua web browser, tapi kebanyakan memilih salah satu web browser.

Walaupun begitu, ECMA terus mengembangkan **ECMAScript** agar bisa mengikuti fitur-fitur baru yang ada di Internet Explorer maupun Netscape Navigator. **ECMAScript 2** dirilis pada bulan Juni 1998. Tidak ada penambahan fitur baru, namun lebih agar seragam dengan standar ISO/IEC 16262 (dimana standar ini juga di daftarkan).

### EMCAScript atau JavaScript?

Sampai disini mungkin anda bingung, kenapa yang digunakan adalah **ECMAScript 2**, bukan **JavaScript 2**?

**ECMAScript** adalah sebuah standar bahasa pemrograman komputer, dimana **JavaScript** merupakan salah satu implementasi dari **ECMAScript**. **JavaScript** tidak bisa dijadikan standar karena masalah merk “**JAVA**” yang merupakan trademark SUN Micosystem (sekarang sudah diakuisisi **Oracle**).

Secara resmi, hanya Netscape dan SUN (Oracle) saja yang boleh menggunakan istilah **JavaScript**. Akan tetapi karena nama **JavaScript** sudah terlanjur populer, istilah ini tetap dipakai sampai sekarang. **EMCAScript** digunakan hanya saat merujuk ke versi dari **JavaScript**.

Dengan demikian, tidak ada namanya **JavaScript 2**, **JavaScript 3**, dst. Yang ada adalah **ECMAScript 2**, **ECMAScript 3**, dst. Dimana kita tetap menyebutnya sebagai **JavaScript** (memang sedikit membingungkan...)

## 2.7 ECMAScript 3

ECMAScript terus dikembangkan hingga mencapai versi 3 pada tahun 1999. Berita baiknya, hampir semua web browser saat itu, terutama Microsoft **Internet Explorer 5.5** dan **Netscape 6** telah mendukung **ECMAScript-262 versi 3**.

ECMAScript 3 menambahkan beberapa fitur baru seperti *regular expression*, penambahan fungsi string (*concat, match, replace, slice, split*), control statements baru (do-while), try/catch exception handling, penanganan error yang lebih baik, format output angka dan perbaikan lainnya.

Berita buruknya, masing-masing web browser menerapkan standar dengan sedikit berbeda, sehingga masih terdapat kemungkinan tidak kompitable.

Dilain pihak, era browser war akan segera berakhir. Dengan teknik marketing dan memanfaatkan “monopoli” sistem operasi Windows buatannya, perlahan tapi pasti Netspace Navigator mulai kehilangan pangsa pasar.

Internet Explorer dibundle dengan gratis di setiap sistem operasi Windows, sehingga tidak lagi ada alasan membeli Netspace Navigator. Ditambah dengan beberapa masalah teknis, Netspace Navigator tinggal menunggu waktu untuk menjadi sejarah.



Pada masa itu, Netspace Navigator merupakan web browser berbayar. Untuk dapat menggunakannya, anda harus membeli lisensi dengan harga \$49 (sekitar Rp. 600.000).

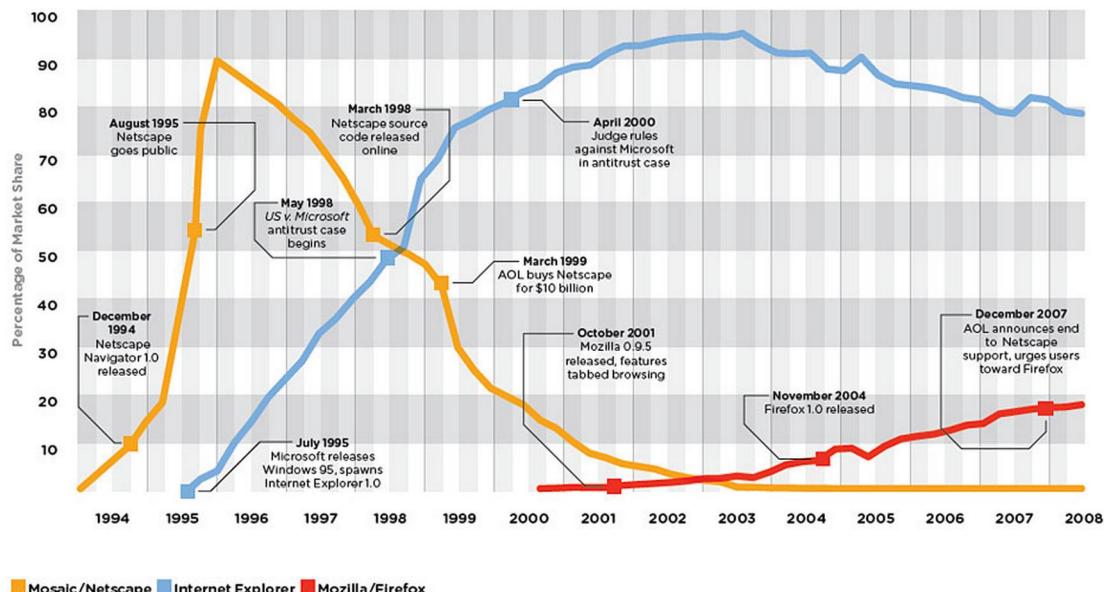
## 2.8 ECMAScript 4 dan Akhir Browser War Pertama

Penerus standar JavaScript, yakni **ECMAScript 4** mengalami nasib yang sama seperti **xHTML 2.0** dan **PHP 6**: pengembangan berhenti di tengah jalan.

Ini disebabkan perbedaan pendapat antar anggota komite TC39, terutama mengenai fitur apa yang harus ada di ECMAScript 4. Proses “perseteruan” berlangsung cukup lama, memakan waktu hingga 10 tahun (sampai dengan 2009). Selama jangka waktu tersebut, tidak ada versi baru dari ECMAScript.

Di masa ini tidak ada kemajuan berarti di dunia web programming. xHTML 2.0 yang dikembangkan oleh W3C juga mengalami jalan buntu.

Di pasar web browser, Internet Explorer menjadi sangat dominan, menguasai lebih dari 80% - 90% market share dari tahun 2001 hingga 2009. Netspace Navigator bisa dibilang sudah punah pada tahun 2004. Web browser Opera hadir sebagai alternatif, tapi tidak bisa berbuat banyak.



Gambar: Grafik penggunaan web browser IE vs Netscape Navigator, sumber: plyojump.com

## 2.9 AJAX dan JavaScript Library

Sekitar tahun 2005, kelompok Open Source dan komunitas programmer mencoba membuat sebuah revolusi untuk JavaScript. Ini diawali oleh **Jesse James Garrett** yang menulis paper tentang kemungkinan penggabungan JavaScript di client dengan pemrograman di server. Teknologi ini kemudian dikenal dengan istilah **AJAX**, singkatan dari (*Asynchronous JavaScript and XML*).

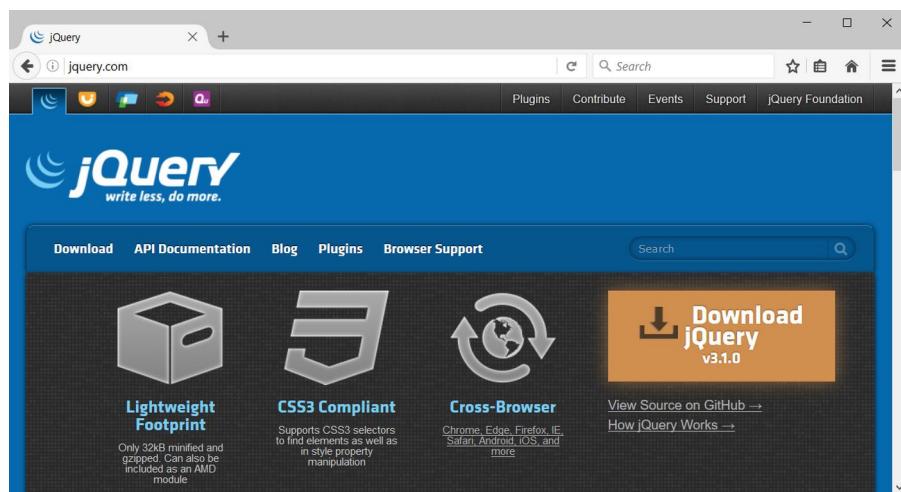
AJAX memungkinkan sebuah halaman web berkomunikasi langsung dengan server tanpa harus di-reload. Komunikasi antara web browser dengan web server berlangsung di latar belakang (background) secara *asynchronous*, hasilnya website menjadi lebih dinamis.

Sebagai contoh, halaman registrasi bisa langsung mengecek apakah username yang diinput sudah ada di database atau belum. Ini dapat dilakukan sesaat setelah user berpindah dari kolom input username ke kolom dibawahnya. Tanpa menggunakan AJAX, proses pengecekan baru berlangsung saat user selesai mengisi form dan men-klik tombol submit (karena pengecekan harus dilakukan ke database yang ada di server).

Salah satu web pertama yang menggunakan teknologi AJAX adalah **Gmail** dan **Google Map**. Yang segera diikuti oleh berbagai aplikasi web lain.

Selain AJAX, berkembang juga berbagai komunitas dan library JavaScript seperti **Prototype**, **jQuery**, **Dojo Toolkit**, dan **MooTools**.

Masa-masa ini bisa dibilang awal kebangkitan JavaScript. Dengan menggunakan library seperti **jQuery**, perbedaan implementasi ECMAScript dari berbagai web browser bisa diatasi dengan mudah. **jQuery** menyediakan '*abstraction layer*' agar web programmer bisa berfokus kepada fitur yang ingin dicapai. Programmer yang sebelumnya "anti" dengan JavaScript (karena susahnya mengatasi perbedaan fitur web browser), mulai melirik library seperti **jQuery**.



Gambar: Tampilan website jQuery di jquery.com

## 2.10 ECMAScript 5

Agar pengembangan terus berjalan, komite TC39 sepakat untuk melompati ECMAScript 4 dan lanjut ke **ECMAScript 5**. Beberapa fitur yang tadinya ingin dirancang untuk ECMAScript 4, dialihkan ke ECMAScript 5. **ECMAScript 5** resmi dirilis pada Desember 2009. Diantara fitur yang baru adalah: *strict mode*, *accessors*, dan **JSON**.

Pada bulan juni 2011, **ECMAScript 5.1** juga dirilis untuk menyamakan dengan standar ISO/IEC 16262:2011.

## 2.11 Reinkarnasi Netscape: Mozilla Firefox

Kalah telak dari Internet Explorer, pada awal tahun 1998 Netscape melakukan langkah yang cukup mengejutkan: Netscape Navigator akan digratiskan dan kode program (*source code*) web browser Netscape Navigator dirilis ke public sebagai proyek Open Source. Langkah kedua ini kelak akan mengubah sejarah web browser. Untuk mengelola kode program tersebut didirikan sebuah organisasi non profit: **Mozilla**.

Di tahun yang sama, Netscape Communcation diakuisisi oleh **America Online (AOL)**. Sejak di bawah AOL, web browser Netscape terus dikembangkan namun tetap tidak berhasil men-galahkan dominasi Internet Explorer. Versi terakhir **Netscape Navigator 9** dirilis pada awal tahun 1997. Setelah itu Netscape Navigator resmi dihentikan.

Di pihak lain, komunitas Mozilla merilis web browser baru: **Mozilla Firefox** pada akhir tahun 2004. Web browser ini mendapat respon yang sangat baik, sanggup mengurangi dominasi Internet Explorer. Firefox segera menjadi web browser nomor 2 terbanyak setelah IE.



Pada awalnya Mozilla ingin menggunakan nama **Phoenix**, untuk merefleksikan web browser baru yang hadir dari ‘bangkai’ Netscape Navigator. Nama ini akhirnya batal karena masalah trademark dengan perusahaan **Phoenix Technologies** yang sudah hadir lebih dulu.

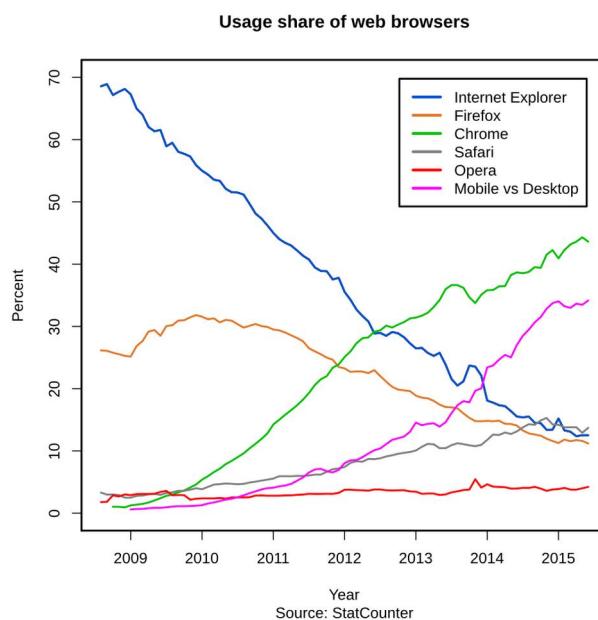
## 2.12 Browser War Kedua

Setelah perang web browser pertama berakhir dengan kekalahan telak Netscape, perang kedua segera mulai dengan dirilisnya **Mozilla Firefox**.

Dengan cepat Firefox menjadi web browser favorit yang sepertinya akan segera menggusur IE. Firefox membawa banyak fitur baru, seperti *tabbed browsing*, *session restore*, dan *spell-checker*. Firefox pun mendukung penuh berbagai standar terbaru ECMAScript maupun W3C (untuk HTML dan CSS).

Puncaknya di tahun 2010 Firefox menguasai sekitar 30% pasar web browser, walaupun IE masih tetap dominan tapi setiap tahun mengalami tren penurunan. Tidak lama lagi sepertinya Firefox bisa menjadi web browser paling populer menggantikan IE.

Namun harapan ini pupus karena muncul web browser baru dari raja mesin pencari: **Google Chrome** yang dirilis pada Desember 2008. Didukung dengan promosi gencar, nama besar Google, fitur menawan, dan eksekusi yang cepat, membuat Google Chrome segera menjadi web browser paling banyak digunakan hingga saat ini, mengalahkan IE, Firefox, dan Opera.



Gambar: Grafik peta “browser war kedua” antara tahun 2008 - 2016, sumber: wikipedia.org

## 2.13 ECMAScript 6

ECMAScript 6 atau ES6 atau ECMAScript 2015 dirilis pada bulan Juni 2015. Cukup banyak penambahan baru pada versi ini, sebagian besar merupakan fitur lanjutan untuk membuat aplikasi yang memiliki kompleksitas tinggi, seperti penggunaan JavaScript di server menggunakan **node.js**.

Diantara fitur tersebut adalah: iterator baru, *python-style generator*, *arrow function*, *binary data*, *typed arrays*, **collections** (*maps*, *sets* and *weak maps*), **promises** untuk membuat *asynchronous*

*programming*, penambahan function untuk tipe data *number* dan *math*, *reflection*, serta *proxies* (*metaprogramming* untuk virtual objects dan wrappers).

Mulai dari **ECMAScript 6** dan selanjutnya, penamaan ECMAScript akan menggunakan nama tahun saat standar tersebut dirilis, seperti **ECMAScript 2015**, **ECMAScript 2016**, dst. Banyak perdebatan mengenai pilihan nama ini, sehingga masih sering disebut sebagai ECMAScript 6.

## 2.14 ECMAScript 7

ECMAScript 7 atau nama resminya: **ECMAScript 2016**, diselesaikan pada Juni 2016. Fitur baru termasuk *exponentiation operator* (\*\*) dan **Array.prototype.includes**.

## 2.15 Versi ECMAScript Berapa yang Harus Saya Pelajari?

Kisah panjang dari JavaScript yang saya tulis sebelumnya diperlukan untuk menjawab pertanyaan ini. Dari sekian banyak versi ECMAScript, tentunya paling pas jika kita langsung belajar ECMAScript 7, karena inilah versi terakhir dan versi paling baru.

Akan tetapi **kompatibilitas** menjadi masalah utama. JavaScript (atau ECMAScript) harus dijalankan dari web browser pengunjung website. Tentunya tidak semua pengunjung situs kita menggunakan web browser terbaru yang bisa menjalankan ECMAScript 7.

ECMAScript 6 pun belum bisa dibilang “aman”, karena baru dirilis tahun lalu. Ditambah lagi web browser mobile seperti di smartphone android yang belum tentu update.

ECMAScript 5 menjadi versi JavaScript yang paling stabil dan boleh dibilang sudah di dukung penuh mayoritas web browser saat ini (kecuali IE jadul seperti IE6 bawaan Windows XP dan IE7). Karena itulah dalam buku ini kita lebih banyak membahas **ECMAScript 5**.

Walaupun menggunakan **ECMAScript 5**, dasar JavaScript yang ada di buku ini tetap valid untuk versi 6 dan versi 7. Dalam beberapa bahasan, saya akan membuat catatan jika terdapat perbedaan di versi ES yang lebih baru, atau ada jika ada fitur yang cukup penting di ECMAScript versi terbaru.

Fitur tambahan yang ada di **ECMAScript 6** dan **ECMAScript 7** juga lebih banyak ke materi lanjutan yang terlalu kompleks jika dimasukkan ke buku JavaScript untuk pemula. ES6 dan ES7 lebih cocok jika anda berniat mempelajari JavaScript sebagai bahasa pemrograman server menggunakan **Node.js**.



Jika anda mendapati buku dengan judul **ECMAScript 6** atau **ECMAScript 7**, buku seperti ini biasanya lebih ke pembahasan fitur terbaru di ES6 dan ES7, bukan membahas JavaScript dari awal.

## 2.16 Mengenal ECMAScript Engine (JavaScript Engine)

**JavaScript Engine** adalah mekanisme internal yang dimiliki oleh web browser untuk menjalankan kode JavaScript. JavaScript Engine dapat disamakan dengan compiler dalam bahasa pemrograman lain, yakni algoritma yang digunakan untuk menjalankan JavaScript. Semakin cepat sebuah web browser menjalankan JavaScript akan semakin baik.

Biasanya disetiap rilis baru web browser seperti Google Chrome, Internet Explorer, maupun Mozilla Firefox, juga diikuti rilis terbaru JavaScript Engine yang menawarkan kecepatan lebih baik.

**V8** adalah nama JavaScript Engine untuk Google Chrome, **SpiderMonkey** untuk Mozilla Firefox, dan **Chakra** untuk Internet Explorer. Daftar lengkap tentang JavaScript engine ini dapat dilihat di [List of ECMAScript engines<sup>1</sup>](#).

Umumnya kita tidak akan berurusan dengan JavaScript Engine ini, hanya sekedar pengetahuan yang mungkin akan anda temui dalam bahasan tentang JavaScript.

## 2.17 Perkembangan JavaScript Saat Ini

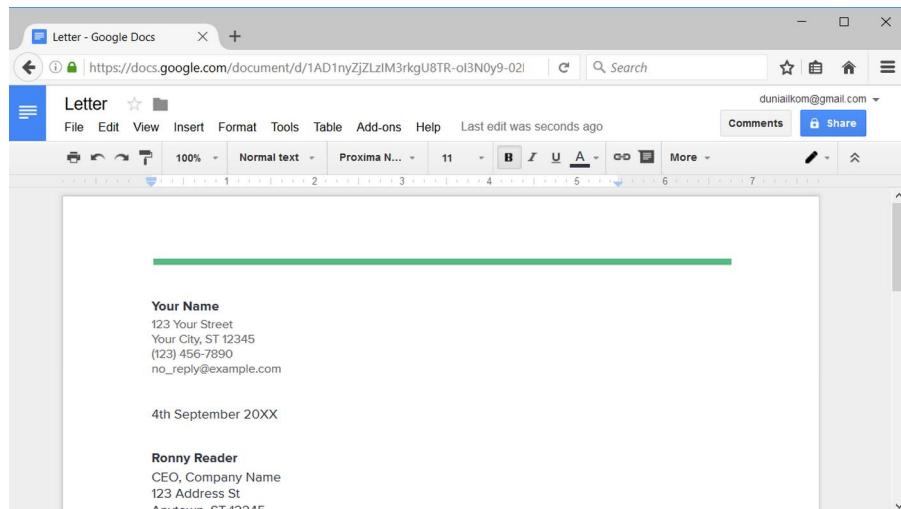
Antara 2-3 tahun belakangan, JavaScript berkembang dengan sangat pesat, terutama sejak dirilisnya **Node.js**. Dengan menggunakan Node.js, JavaScript bisa berjalan sebagai bahasa pemrograman server. Fitur utamanya seperti *non-blocking I/O model*, dimana banyak pemrosesan bisa berjalan bersamaan, tanpa menunggu proses lain selesai.

Selain itu juga banyak bermunculan framework JavaScript seperti **AngularJS** yang dikembangkan oleh Google serta **ReactJS** yang dikelola oleh FaceBook. Framework seperti ini digunakan untuk membuat website yang tidak berbentuk “website”, tetapi menyerupai aplikasi desktop yang dikenal sebagai **Single-page Application (SPA)**.

Contoh dari **Single-page Application** ini seperti aplikasi Google: Gmail, GDrive, Google Doc, dll. Di website tersebut, halamannya akan tetap sama, tidak di reload seperti layaknya sebuah website.

---

<sup>1</sup>[http://en.wikipedia.org/wiki/List\\_of\\_ECMAScript\\_engines](http://en.wikipedia.org/wiki/List_of_ECMAScript_engines)



Gambar: Tampilan halaman Google Doc, sebuah *Single-page Application*

Teknologi JavaScript ini juga memunculkan stack server baru. Jika selama ini yang populer adalah **LAMP** (Linux, Apache, Mysql dan PHP), sekarang dikenal juga **MEAN stack** (**MongoDB** database, **Express.js** web application server framework, **Angular.js**, dan **Node.js** runtime environment).

Salah satu rekan duniaIlkom ada yang sudah bekerja sebagai web developer di salah satu perusahaan. Bocorannya, 80% dari yang dikerjakan berhubungan dengan JavaScript. Hal ini menunjukkan besarnya dominasi JavaScript di pemrograman web sekarang ini.

Namun perlu juga dipahami bahwa walaupun materi di eBook JavaScript Uncover sudah lumayan rumit, ini barulah dasar dari JavaScript. Jika anda serius ingin mempelajari JavaScript lebih jauh lagi, bisa lanjut ke library seperti jQuery, framework seperti AngularJS maupun ReactJS, atau ke server menggunakan Node.js.

---

Dalam bab ini anda telah membaca cerita panjang perjalanan bahasa pemrograman **JavaScript**, mulai dari perubahan nama dari **Mocha** ke **LiveScript**, hingga menjadi **JavaScript**. Kita juga melihat perjalanan standar ECMAScript yang diikuti kemunculan berbagai library dan framework **JavaScript**.

Walaupun tidak berhubungan dengan coding, pengetahuan ini menjadi modal yang baik. Terutama jika anda membaca tutorial yang membahas fitur terbaru **ECMAScript 6** dan **7**, yang tidak lain adalah **JavaScript** juga.

Berikutnya kita akan langsung masuk praktik menjalankan kode program **JavaScript**.

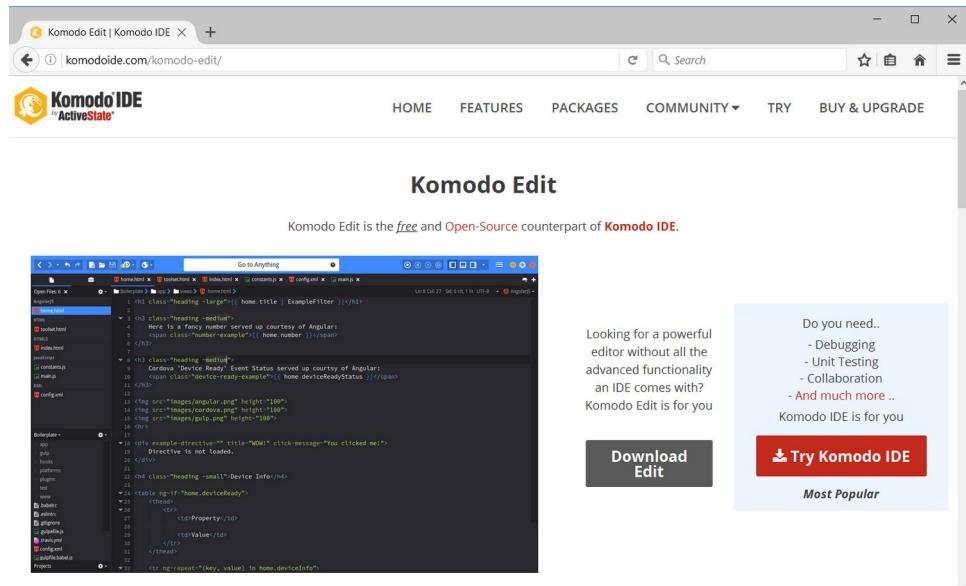
# 3. Menjalankan Kode Program JavaScript

Dalam bab ini kita akan mempelajari cara menjalankan kode program JavaScript. Dimulai dari mempersiapkan aplikasi teks editor dan web browser, membahas 3 metode input JavaScript, mengenal tab console dari web developer tools, hingga menampilkan pesan untuk web browser yang tidak mendukung JavaScript.

## 3.1 Text Editor dan Web Browser

Sama seperti HTML, aplikasi yang dibutuhkan untuk menulis dan menjalankan kode JavaScript hanya 2, yakni **text editor** dan **web browser**. Saya yakin kedua aplikasi ini sudah tersedia di komputer anda (karena syarat utama sebelum mempelajari JavaScript, sudah pernah membuat kode HTML).

Untuk teks editor, pilihan saya sama seperti di buku Uncover sebelumnya: [Notepad++<sup>1</sup>](#) atau [Komodo Edit<sup>2</sup>](#). Ini bukan sebuah keharusan, anda boleh menggunakan text editor apapun yang dirasa nyaman. Alternatif lain bisa coba: **Sublime Text** (berbayar), **Bracket**, **Atom**, **Aptana Studio**, **Eclipse** atau **Net Beans**. Untuk pengguna Mac, bisa mencoba **Text Wrangler**.



Gambar: Tampilan website untuk download Komodo Edit

Dalam buku ini saya lebih memilih **Komodo Edit** daripada **Notepad++**. Karena seperti yang akan kita pelajari nanti, **syntax** (aturan penulisan) JavaScript jauh lebih rumit daripada HTML

<sup>1</sup><http://notepad-plus-plus.org>

<sup>2</sup><http://komodoide.com/komodo-edit/>

maupun CSS. JavaScript bersifat *case sensitif* (huruf kecil dan huruf besar dianggap berbeda). Berbagai function di JavaScript sangat panjang dan tidak boleh ada salah ketik. Sebagai contoh `document.getElementById` berbeda dengan `document.getElementException` (berbeda hanya pada huruf ‘i’ kecil).

**Komodo Edit** menawarkan banyak fitur tambahan dibandingkan Notepad++, seperti *code completion* yang lebih powerfull serta **inline error correction**. Dengan *inline error correction*, pada saat kita mengetik kode program, Komodo Edit akan langsung memberi warning atau peringatan jika terjadi kesalahan, seperti lupa menambahkan tanda “;”, atau menggunakan *function* yang belum terdefinisi. Fitur seperti ini sangat bermanfaat, walaupun kadang juga bisa menggannggu.

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <script>
10    alert("Hello World");
11  </script>
12 </body>
13 </html>
```

Gambar: Fitur *inline error correction* pada Komodo Edit

Untuk web browser, saya menggunakan **Mozilla Firefox** versi 48.0.1 yang diupdate pada September 2016 (saat buku ini ditulis). Anda boleh menggunakan web browser lain seperti **Google Chrome**, **Opera**, bahkan **Internet Explorer** maupun **Microsoft Edge** (bawaan Windows 10), asalkan web browser tersebut update. Karena dalam beberapa pembahasan kita akan menggunakan fitur ECMAScript 6 dan 7.

## 3.2 Hello World dari JavaScript

Setelah aplikasi text editor dan web browser tersedia, saatnya mencoba membuat teks “Hello World” dengan JavaScript. Membuat teks “Hello World” bisa dibilang sebagai sebuah ‘ritual’ yang hampir selalu dilakukan saat mempelajari bahasa pemrograman baru.



Sekedar pengetahuan, kebiasaan “Hello World” berawal sejak tahun 1972 oleh **Brian Kernighan**, seorang ilmuwan komputer yang bekerja di Bell Labs. Hal ini tertulis di dalam buku “[The C Programming Language](#)<sup>3</sup>” karya Brian Kernighan dan Dennis Ritchie di tahun 1978. Karya ini dinobatkan sebagai buku terbaik dan paling berpengaruh dalam perkembangan bahasa pemrograman komputer.

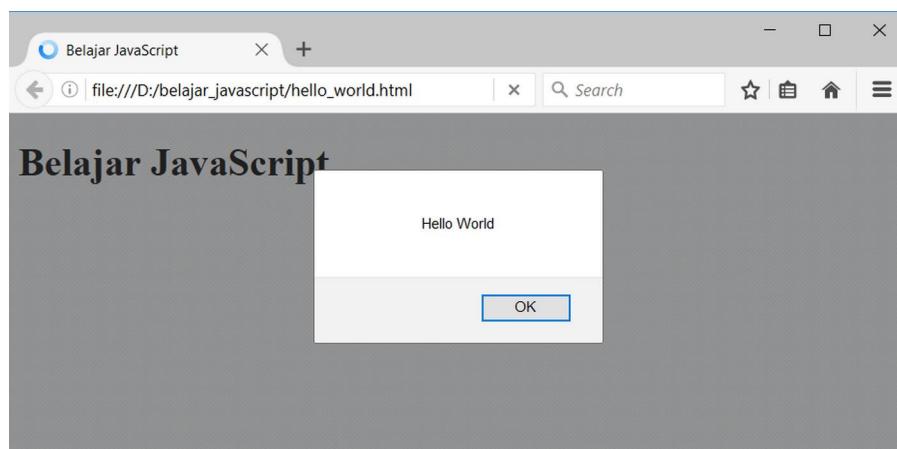
Baik, mari langsung praktek. Siapkan sebuah folder untuk menyimpan file latihan kita. Sebagai contoh, saya membuat folder “belajar\_javascript” di drive D. Anda boleh membuat folder yang sama maupun menggunakan folder lain.

<sup>3</sup>[https://en.wikipedia.org/wiki/The\\_C\\_Programming\\_Language](https://en.wikipedia.org/wiki/The_C_Programming_Language)

Buka teks editor, lalu ketikkan kode program berikut:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <script>
10     alert("Hello World");
11   </script>
12 </body>
13 </html>
```

Save sebagai **hello\_world.html**, lalu jalankan di web browser:



Gambar: Hello World dari JavaScript

Inilah hasil dari kode program diatas. Sebuah halaman HTML dengan jendela popup yang menampilkan tulisan “Hello World”. Mari kita bahas beberapa aspek dari kode program ini.

**Pertama**, 90% kode program terdiri dari HTML, yakni tag `<head>`, `<meta>`, `<title>`, `<body>`, dan `<h1>`. Kode JavaScript sendiri disisipkan ke dalam tag `<script>`.

**Kedua**, satu-satunya perintah JavaScript yang kita jalankan adalah `alert("Hello World")`; Kode ini berfungsi menampilkan jendela *popup* dengan teks “Hello World”. Perintah `alert()` merupakan sebuah *function* di dalam JavaScript. Anda bisa mengubah kode tersebut menjadi `alert("Selamat Siang")`; dan akan tampil jendela popup dengan teks “Selamat Siang”.

**Ketiga**, jika diperhatikan nama file tetap menggunakan extension `.html`, yakni `hello_world.html`. Sama seperti CSS, apabila kode JavaScript disisipkan ke dalam HTML, kita tetap meyimpannya sebagai `*.html`. Terkecuali jika kode JavaScript ini berada di sebuah file khusus (external JavaScript) maka akhiran filenya adalah `*.js`. Ini akan kita pelajari sesaat lagi.

### 3.3 Cara Menginput JavaScript ke dalam HTML

Contoh praktek “Hello World” yang kita jalankan sebelumnya menempatkan JavaScript langsung ke dalam kode HTML, atau dikenal sebagai **Internal JavaScript**. Selain cara tersebut, terdapat 2 metode lain untuk menginput kode JavaScript, yakni: **Inline** dan **External**. Mari kita bahas.

#### Inline JavaScript

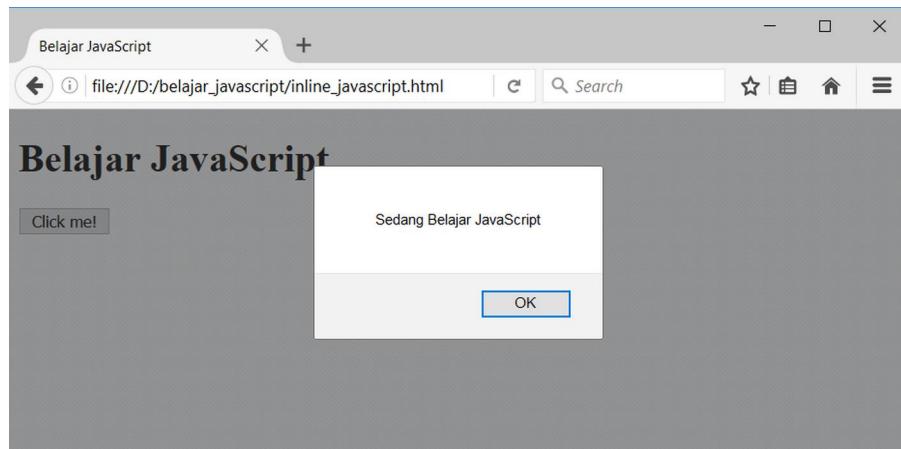
Inline JavaScript artinya menempatkan kode JavaScript secara langsung ke dalam atribut tag HTML. Berikut contoh dari inline JavaScript:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <button onclick="alert('Sedang Belajar JavaScript');">
10     Click me!
11   </button>
12 </body>
13 </html>
```

Disini saya memiliki sebuah tag `<button>` yang digunakan untuk membuat tombol. Perhatikan terdapat atribut `onclick="alert('Sedang Belajar JavaScript');"`, ini adalah kode JavaScript.

**Onclick** merupakan atribut khusus yang dikenal sebagai **event**. Event nantinya akan kita bahas dalam bab tersendiri. Namun sebagai gambaran, `onclick` berfungsi untuk menjalankan kode JavaScript hanya pada saat tombol tersebut di-click. Hampir semua element HTML bisa ditambahkan atribut **event** seperti ini.

Silahkan anda jalankan kode program diatas, kemudian klik tombol **Click me!**.



Gambar: Function alert() berjalan saat tombol ‘Click me!’ ditekan

Metode inline JavaScript sangat praktis, namun memiliki beberapa kelemahan:

1. Kode JavaScript “bercampur” dengan HTML. Akan memakan waktu lama untuk mencari kode seperti ini diantara ratusan baris HTML.
2. Kode JavaScript hanya bisa ditulis dalam 1 baris, sehingga tidak cocok untuk kode yang panjang.
3. Kode JavaScript tidak bisa digunakan di tempat lain (*code reuse*). Misalnya jika saya memiliki 5 halaman, terpaksa dalam setiap halaman ditulis kode JavaScript yang sama berulang kali.



Karena bisa ditulis secara instant dan singkat, anda akan sering menemukan inline JavaScript dalam artikel atau tutorial di internet. Untuk web yang sebenarnya, external JavaScript lebih disarankan (akan kita bahas sesaat lagi).

## 3.4 Internal JavaScript

Metode **internal JavaScript** sudah kita coba pada saat membuat program “Hello World”. Disini kode JavaScript ditulis di dalam tag <script>. Tag <script> merupakan tag khusus untuk menginput kode script, dimana salah satunya kode script itu adalah JavaScript.

Sebelum era HTML5 seperti saat ini, penulisan tag <script> ditambahkan atribut **language** dan/atau **type**, seperti contoh berikut:

```
<script type="text/javascript">
  //... kode javascript disini
</script>

<script language="javascript">
  //... kode javascript disini
</script>

<script type="text/javascript" language="javascript">
  //... kode javascript disini
</script>
```

Sekarang, ketiga cara penulisan diatas tidak lagi direkomendasikan. Kita cukup menulis tag `<script>` tanpa atribut apapun:

```
<script>
  //... kode JavaScript disini
</script>
```

Atribut `type` dan `language` merupakan ‘warisan’ dari era browser war. Ketika itu kode script yang tersedia tidak hanya JavaScript saja, juga ada **JScript** dan **VBScript**. Saat ini semua web browser manjadikan JavaScript sebagai bahasa default, sehingga atribut `type` dan `language` tidak perlu dituliskan.

Pembahasan yang menarik adalah, dimanakah sebaiknya kode JavaScript ini diletakkan? Apakah di bagian `<head>`, di tengah tag `<body>`, atau di akhir? Lokasi penempatan ini dipengaruhi oleh 2 aspek: **performa** dan **eksekusi**.

Biasanya, kode JavaScript ditempatkan di bagian atas, yakni di dalam tag `<head>`, seperti contoh berikut:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta charset="utf-8">
5    <title> Belajar JavaScript </title>
6    <script>
7      // kode javascript disini
8    </script>
9  </head>
10 <body>
11   <h1>Belajar JavaScript</h1>
12   <button> Click me! </button>
13 </body>
14 </html>
```

Menempatkan kode JavaScript di bagian atas seperti ini banyak ditemukan. Namun berkaitan dengan masalah **performa**, beberapa developer web menyarankan meletakkan JavaScript dibagian bawah tag `<body>`, yakni sebelum tag penutup `</body>`, sebagaimana yang dijelaskan dari sebuah artikel di Yahoo Developer Network: [Best Practices for Speeding Up Your Web Site<sup>4</sup>](#), seperti contoh berikut:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <button> Click me! </button>
10  <script>
11    // kode javascript disini
12  </script>
13 </body>
14 </html>
```

Untuk internal JavaScript, peningkatan performa tidak akan begitu terasa. Posisi JavaScript di bagian bawah seperti ini lebih ditujukan untuk **external JavaScript**.

Jika dikaitkan dengan proses **eksekusi**, posisi kode JavaScript bisa mempengaruhi alur proses dari web browser. Mari kita masuk ke contoh praktek:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <script>
7     alert("Hello 1");
8   </script>
9 </head>
10 <body>
11   <h1>Belajar JavaScript</h1>
12   <script>
13     alert("Hello 2");
14   </script>
15   <h1>Belajar Web Programming</h1>
16   <script>
17     alert("Hello 3");
```

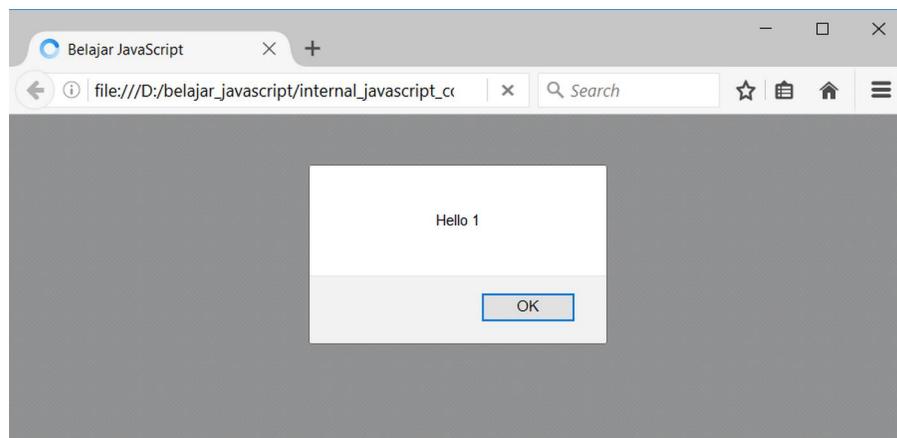
---

<sup>4</sup>[https://developer.yahoo.com/performance/rules.html#js\\_bottom](https://developer.yahoo.com/performance/rules.html#js_bottom)

```
18  </script>
19 </body>
20 </html>
```

Disini saya memiliki 3 buah tag `<script>`: di bagian `<head>`, awal tag `<body>`, dan akhir tag `</body>`. Masing-masing kode JavaScript ini menjalankan fungsi `alert()` seperti yang telah kita coba sebelumnya. Di dalam tag `<body>`, saya menempatkan 2 buah tag `<h1>` diantara kode JavaScript.

Mari kita coba:

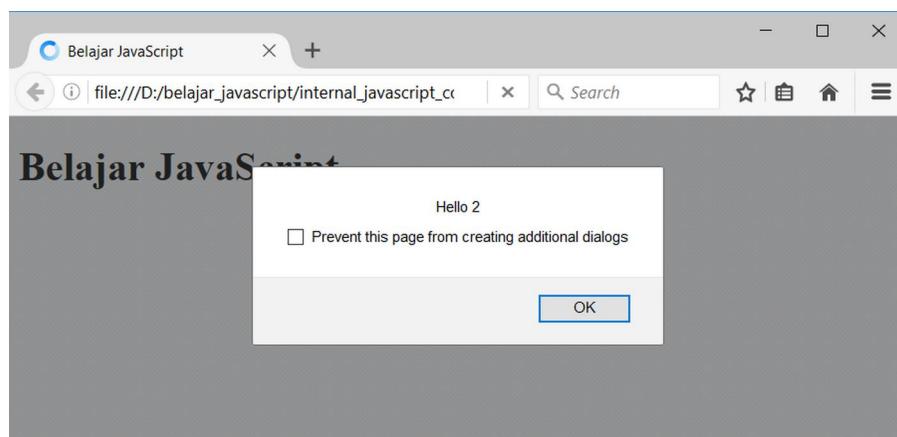


Gambar: Alert "Hello 1" yang berasal dari kode JavaScript di bagian head

Pada saat dijalankan, langsung tampil jendela popup "Hello 1". Ini berasal dari kode JavaScript yang berada di bagian `<head>`.

Sebelum anda klik tombol **OK**, perhatikan halaman web kita, *tidak tampil teks apapun!* Ini karena perintah `alert()` akan menghentikan eksekusi kode program yang dilakukan web browser. Artinya, saat ini yang diproses oleh web browser hanya sampai bagian `<head>` saja, belum masuk ke tag `<body>`. Karena itulah anda tidak melihat teks apapun di halaman web.

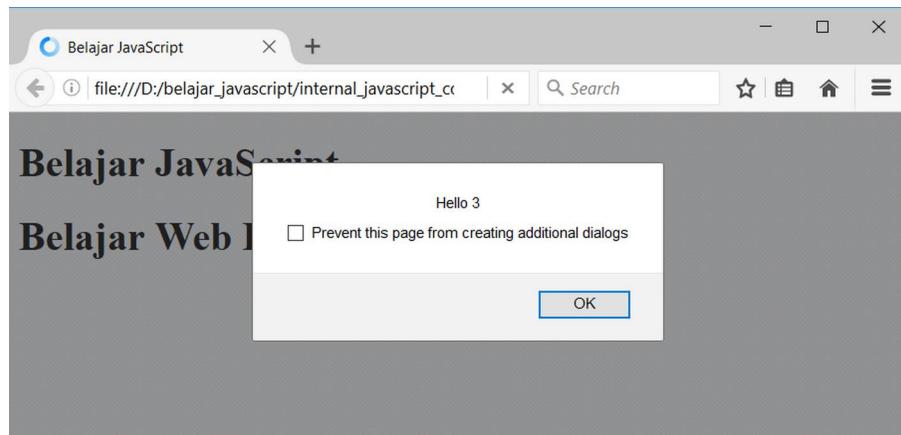
Silahkan klik tombol **OK**:



Gambar: Alert "Hello 2" yang berasal dari kode JavaScript di awal tag body

Sekarang tampil alert: "Hello 2" yang berasal dari kode JavaScript kedua. Untuk sementara, eksekusi halaman kembali terhenti. Di dalam web browser terlihat teks "Belajar JavaScript" yang berasal dari tag `<h1>Belajar JavaScript</h1>`. Teks ini tampil karena tag `<h1>` saya tulis sebelum `alert("Hello 2")`.

Tambahan checkbox "*prevent this page from creating additional dialogs*" tampil karena proteksi dari web browser. Untuk website yang sebenarnya (yang ada di internet), akan sangat mengejaskan jika mendapatkan website yang menampilkan jendela alert berulang kali. Web browser memberi kita pilihan untuk tidak melihat jendela alert berikutnya. Untuk kali ini, abaikan checkbox tersebut dan klik tombol **OK**:



Gambar: Alert "Hello 3" yang berasal dari kode JavaScript di akhir tag body

Alert kali ini berasal dari kode JavaScript terakhir yang berada sebelum tag penutup `</body>`. Perhatikan sekarang header `<h1>Belajar Web Programming</h1>` sudah terlihat.

Sampai disini, kode HTML sebenarnya belum selesai di proses, anda bisa lihat tanda loading di sudut kiri atas web browser yang masih berputar. Hanya ketika tombol **OK** di klik, sisa kode HTML segera di eksekusi dan halaman web selesai diproses. Inilah yang dijalankan web browser dalam menampilkan sebuah halaman web, yakni secara berurutan dari atas ke bawah, mulai dari baris pertama hingga baris terakhir.

Alur proses tersebut bisa mempengaruhi peletakan kode JavaScript. Jika terdapat kode yang **harus** dijalankan terlebih dahulu (seperti library *jQuery*), tempatkan di dalam tag `<head>`, agar bisa langsung dieksekusi.

Dengan menggunakan metode **internal JavaScript**, kode program kita sudah sedikit rapi. Seluruh kode JavaScript diletakkan di dalam tag `<script>`. Namun kode ini masih berada di halaman yang sama dengan HTML, akibatnya kita tidak bisa menggunakan kode JavaScript tersebut di halaman lain.

## 3.5 External JavaScript

Metode input JavaScript yang paling disarankan adalah **external JavaScript**. Disini kita membuat file khusus yang hanya berisi kode program JavaScript. File ini nantinya "dipanggil" oleh halaman HTML yang membutuhkan.

Sebagai contoh praktek, silahkan buat sebuah file baru yang isinya adalah sebagai berikut:

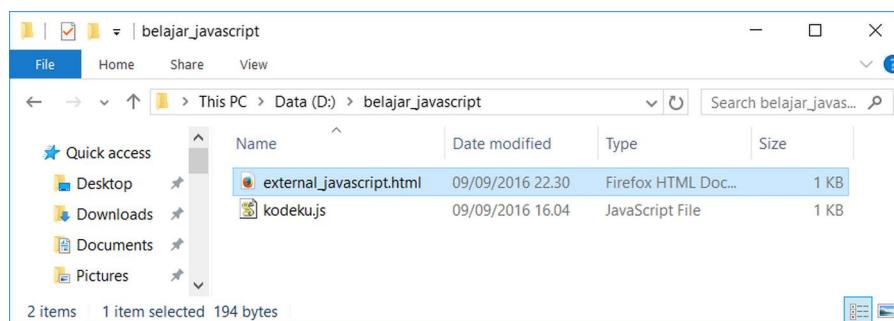
```
1 alert("Hello dari External JavaScript");
```

Yup, hanya satu baris saja kerena kita belum membahas kode JavaScript lain. Save file diatas sebagai kodeku.js. Nama file ini tidak harus kodeku.js, anda bisa menggunakan nama lain selama memiliki extension \*.js.

Untuk memasukkan file JavaScript kodeku.js ke dalam HTML, kita menggunakan atribut **src** dari tag <script>. Silahkan buat file HTML dengan kode berikut:

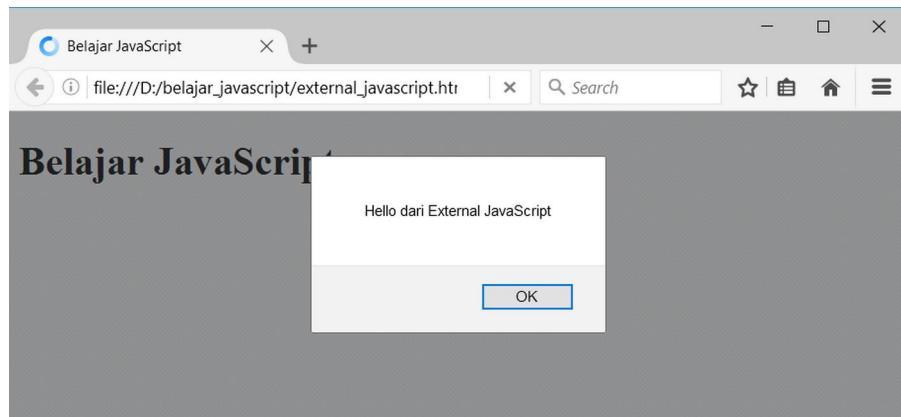
```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <script src="kodeku.js"></script>
10 </body>
11 </html>
```

File HTML diatas saya simpan dengan nama external\_javascript.html. Sehingga di folder belajar\_javascript terdapat 2 buah file:



Gambar: File belajar\_javascript dan kodeku.js berada dalam folder yang sama

Mari kita jalankan:



Gambar: Tampilan alert() yang berasal dari external JavaScript

Hasilnya sebuah jendela popup dengan teks "Hello dari External JavaScript". Ini artinya kode JavaScript yang berada di `kodeku.js` telah berhasil dieksekusi.

Sama seperti atribut `src` pada tag `<img>`, file JavaScript juga bisa menggunakan **alamat relatif** maupun **alamat absolute**. Kita bisa saja menginput file JavaScript yang berada di server lain, seperti contoh berikut:

```
<script src="https://code.jquery.com/jquery-3.1.0.js"></script>
```

Teknik seperti ini sering digunakan untuk mengakses file JavaScript yang berada di **CDN (Content Delivery Network)**.

Untuk menginput file JavaScript external, pasangan tag penutup `</script>` tetap harus ditulis walaupun diantara tag `<script>` dan `</script>` tidak berisi kode apapun. Kita tidak bisa menulis *self closing tag* seperti berikut:

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

Diantara tag `<script>` dan `</script>` juga tidak boleh terdapat kode internal JavaScript lain:

```
<script src="kodeku.js">
  alert("Ini tidak akan berjalan");
</script>
```

Jika butuh memanggil lebih dari 1 file JavaScript external, bisa dengan menulis tag `<script>` beberapa kali:

```
<script src="kodeku1.js"></script>
<script src="kodeku2.js"></script>
<script src="kodeku3.js"></script>
```

Yang perlu diperhatikan, file JavaScript ini di eksekusi secara berurutan mulai dari yang paling atas. File kodeku1.js di proses pertama kali, kemudian diikuti oleh kodeku2.js dan terakhir kodeku3.js.

Kita perlu mengatur posisi file jika terdapat kode JavaScript yang bergantung kepada file lain. Misalkan anda membuat program yang memerlukan library jQuery, pemanggilan library ini harus ditulis paling awal. Kalau tidak, kode program tidak akan berjalan.

Dengan memisahkan kode program JavaScript ke dalam file khusus, kita bisa melalukan **code reuse**, yakni menggunakan 1 file JavaScript oleh banyak halaman HTML. Halaman tersebut cukup memanggil file ini menggunakan tag <script>.

Fitur **cache** dari web browser juga bisa mempercepat pengaksesan website dengan cara menyimpan file JavaScript di dalam cache. Ketika mengunjungi halaman HTML yang memiliki file JavaScript external, web browser akan mendownload file-file ini dan disimpan di cache. Saat pindah ke halaman lain yang menggunakan file external JavaScript, web browser cukup mengambilnya di cache, tidak perlu mendownload ulang seluruh file ini (selama file external JavaScript tersebut sama persis).

Dibalik keunggulannya, external JavaScript juga memiliki satu kendala. Spesifikasi **protocol HTTP/1.1** menyatakan bahwa web browser **harus berhenti memproses HTML** pada saat mendownload file external JavaScript. Situasinya mirip seperti menulis fungsi alert() di bagian <head>. Sebagaimana yang telah kita lihat, fungsi alert() akan menahan web browser untuk memproses kode HTML sampai tombol **OK** di klik.

Sebagai contoh, misalkan saya memiliki kode HTML berikut:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8">
5      <title> Belajar JavaScript </title>
6      <script src="kodeku1.js"></script>
7      <script src="kodeku2.js"></script>
8      <script src="kodeku3.js"></script>
9      <script src="kodeku4.js"></script>
10 </head>
11 <body>
12     <h1>Belajar JavaScript</h1>
13 </body>
14 </html>
```

Kode program akan diproses secara berurutan dari baris paling atas ke bawah. Pertama kali web browser akan memproses kode HTML baris 1-4, kemudian masuk ke tag <script> di baris ke 5. Disini web browser berhenti sesaat untuk mendownload file kodeku1.js. Selama proses download ini, web browser dipaksa berhenti dan tidak bisa melakukan proses apapun.

Setelah kodeku1.js selesai di download, isinya akan segera di jalankan, kemudian lanjut ke baris 6 untuk mendownload kodeku2.js, kembali berhenti sesaat, dan demikian seterusnya hingga akhir kode program di baris 12.

Untuk website yang kompleks, file external JavaScript bukan hanya 4, tapi bisa 10 file atau lebih. Jika seluruh file ini diletakkan di bagian `<head>`, bisa saja halaman web tampil “kosong” selama beberapa detik menunggu web browser selesai mendownload file-file tersebut. Situasi ini dikenal dengan istilah **Render-Blocking JavaScript**<sup>5</sup>.

Solusi dari masalah ini adalah dengan memanggil file external JavaScript dari bagian bawah tag `<body>`. Dengan demikian, kita memberi kesempatan web browser untuk memproses kode HTML terlebih dahulu, baru kemudian mendownload file JavaScript. Efeknya, pengujung web bisa langsung melihat tampilan web selama proses ini, tidak hanya halaman kosong.



**External JavaScript** merupakan metode input yang paling disarankan. Tapi seperti yang sudah kita coba, butuh langkah tambahan untuk membuat file khusus JavaScript. Agar lebih sederhana, sebagian besar contoh kode program dalam buku ini akan menggunakan **internal JavaScript**.

External JavaScript hanya akan saya gunakan pada studi kasus yang cukup rumit atau terdapat lebih dari 1 halaman HTML yang butuh JavaScript. Sekali lagi, ini semata-mata hanya untuk mempersingkat penulisan.

## 3.6 Atribut `async` dan `defer`

Render-Blocking JavaScript menjadi sebuah masalah ketika kita ingin menggunakan **External JavaScript**. Ini karena web browser harus membagi waktu antara mendownload file JavaScript dengan memproses kode HTML.

Untungnya **HTML5** hadir sebagai penyelamat. Dengan atribut `async` dan `defer`, kita bisa mengatur kapan dan bagaimana file external JavaScript diproses. Kedua atribut ini memungkinkan penulisan tag `<script>` tidak harus di bawah tag `<body>`.

### Atribut `async`

Pada situasi “normal”, web browser langsung mendownload dan menjalankan file external JavaScript pada saat itu juga. Inilah yang bisa membuat jeda waktu beberapa saat ketika web browser sedang memproses JavaScript.

Jika atribut `async` ditambahkan ke dalam tag `<script>`, file JavaScript akan diproses pada saat yang bersamaan dengan kode HTML (secara simultan). Dengan kata lain, web browser tidak “terkunci” untuk menjalankan kode JavaScript. Metode ini juga dikenal dengan istilah **Asynchronous JavaScript**.

Berikut contoh penulisannya:

```
<script src="kodeku1.js" async></script>
<script src="kodeku2.js" async></script>
```

<sup>5</sup><https://developers.google.com/speed/docs/insights/BlockingJS>

Dengan penambahan atribut **async**, halaman web bisa tampil dengan lebih cepat, tanpa harus menunggu JavaScript selesai diproses. Artinya tidak mengalami *Render-Blocking JavaScript*.

Efek samping dari penggunaan atribut ini, bisa saja file kodeku2.js dieksekusi sebelum kodeku1.js. Atau jika kode JavaScript tersebut digunakan untuk menambah element baru, pengguna bisa melihat gambar dan tulisan yang bergeser saat JavaScript sedang diproses.

Contoh praktik **async** cukup susah dilakukan secara offline, karena kita mengakses halaman web di komputer sendiri tanpa ada jeda di jaringan (terlalu cepat untuk bisa melihat proses **async**). Penggunaannya baru bisa terlihat pada web online dengan banyak file JavaScript external dari berbagai server.

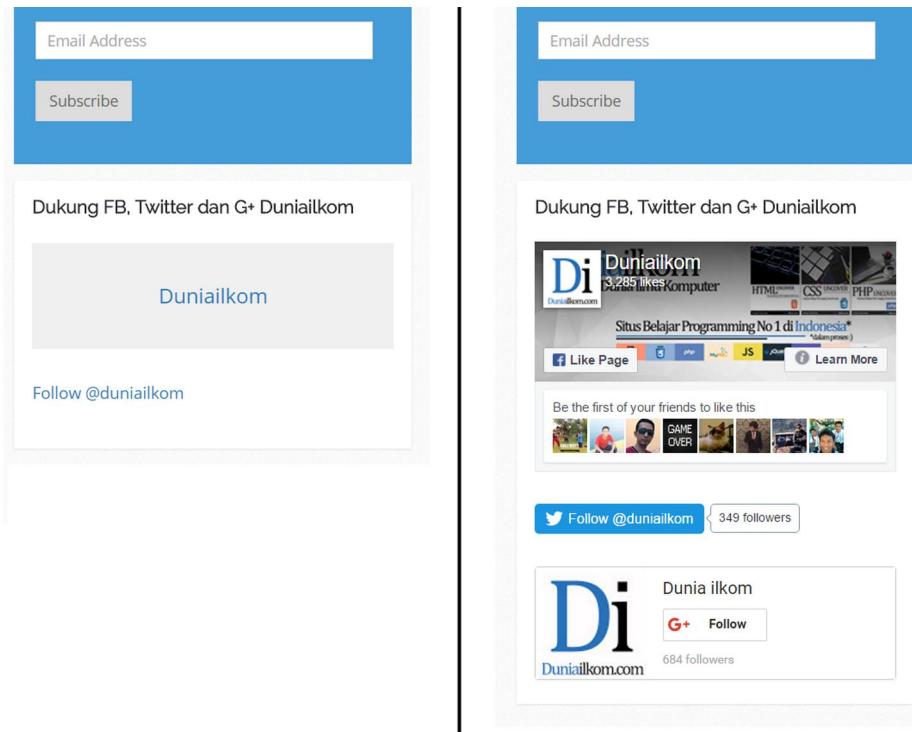
Situs **duniaIlkom** menggunakan cukup banyak file JavaScript external. Saya memerlukan file JavaScript ini untuk menampilkan iklan *Google adsense*, *Facebook page like*, *G+ badge*, hingga tombol *follow twitter*. Ini semua ditampilkan menggunakan JavaScript external dan diproses secara **Asynchronous**.

Berikut kode yang saya gunakan untuk menampilkan fitur tersebut:

```
<script async src="https://apis.google.com/js/platform.js" defer>
</script>
<script async src="//pagead2.googlesyndication.com/pagead/js/adsbygoogle.js">
</script>
<script async src="//platform.twitter.com/widgets.js" charset="utf-8">
</script>
```

Seperti yang terlihat, semuanya menggunakan tambahan atribut **async**. Seperti apa efeknya?

Di sidebar kanan situs **duniaIlkom**, saya menempatkan 3 layanan social media. Apabila anda memiliki koneksi internet yang cukup lambat (eh..), akan terlihat jeda saat halaman tampil pertama kali hingga seluruh kode JavaScript selesai dijalankan:



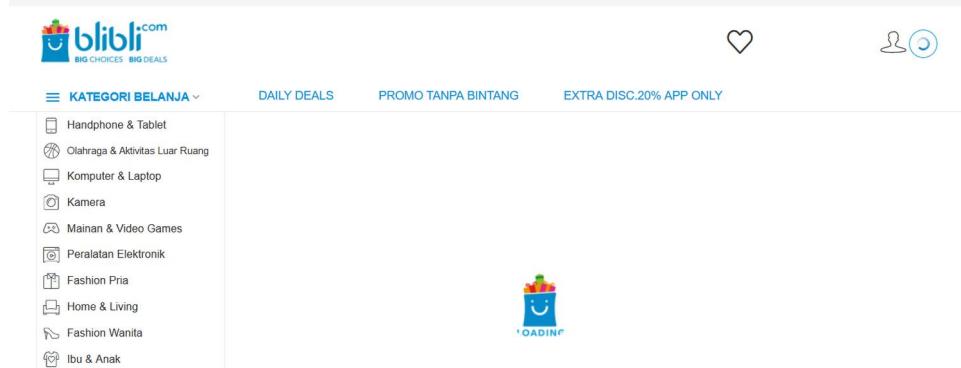
Gambar: Tampilan sidebar duniailkom pada saat proses loading (kiri) dan ketika selesai loading (kanan)

Bagian kiri menampilkan sidebar pada saat halaman duniailkom sedang loading, dan bagian kanan setelah file JavaScript selesai diproses. Sebenarnya, kode HTML dan JavaScript di proses secara bersamaan (**asynchronous**). Tapi karena file JavaScript berada di server lain, butuh proses download yang lebih lama. Oleh karena itulah seolah-olah JavaScript baru diproses setelah kode HTML.

Tanpa atribut **async** dan jika file JavaScript ini saya tempatkan di bagian `<head>`, web browser harus memproses seluruh file JavaScript terlebih dahulu, baru kemudian giliran HTML (tidak bisa bersamaan). Proses ini akan butuh waktu lama karena kode JavaScript tersebut harus diambil dari 3 server yang berbeda, yakni dari server facebook, google, dan twitter.

Dengan tambahan atribut **async**, kode HTML tetap diproses sembari mendownload file JavaScript. Dengan kata lain, web browser tidak masuk ke dalam **Render-Blocking JavaScript**.

Pada beberapa kasus, anda mungkin tidak ingin halaman web yang “setengah jadi” ini tampil, apalagi jika kode JavaScript tersebut sangat penting. Sebagai contoh, silahkan akses situs blibli.com. Jika koneksi anda cukup lambat, situs tersebut akan menampilkan proses loading selama menunggu semua kode program selesai diproses (yang bisa memakan waktu beberapa detik).



Gambar: Animasi proses loading di situs blibli.com

Ini dilakukan supaya pengunjung langsung mendapat halaman full, tidak setengah-setengah. Tetapi kita harus sabar menunggu hingga proses loading ini selesai.

## 3.7 Atribut defer

Atribut **defer** digunakan untuk mengatur kapan file JavaScript dijalankan. Dengan atribut ini, file JavaScript baru di download dan dieksekusi setelah seluruh kode HTML selesai diproses. Tanpa **defer**, file JavaScript akan dijalankan langsung pada posisinya saat ini.

Berikut contoh penulisan atribut **defer**:

```
<script src="kodeku1.js" defer></script>
<script src="kodeku2.js" defer></script>
```



Efek dari atribut **async** dan **defer** mungkin terdengar sama. Perbedaan mendasar adalah, **async** digunakan untuk mengatur *cara eksekusi kode JavaScript*, sedangkan **defer** untuk mengatur *kapan file JavaScript tersebut di download dan diproses*.

Sebagai contoh praktik, saya akan membuat kode JavaScript yang harus dijalankan setelah HTML selesai diproses. Dimana kode JavaScript ini butuh mengakses hasil HTML tersebut. Silahkan buat file HTML baru dengan kode program berikut:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8">
5      <title> Belajar JavaScript </title>
6  </head>
7  <body>
8      <h1>Belajar JavaScript</h1>
9      <button id="tombol">Click Me!</button>
10     <p id="hasil"></p>
```

```

11 <script src="kodeku2.js"></script>
12 </body>
13 </html>

```

Halaman ini terdiri dari sebuah header `<h1>`, satu tombol dari tag `<button>`, dan paragraf kosong dari tag `<p>`. Di bagian akhir terdapat tag `<script>` untuk memanggil file `kodeku2.js`. Tentu saja saya harus menyiapkan file ini.

Silahkan buat file `kodeku2.js` yang berisi kode program berikut:

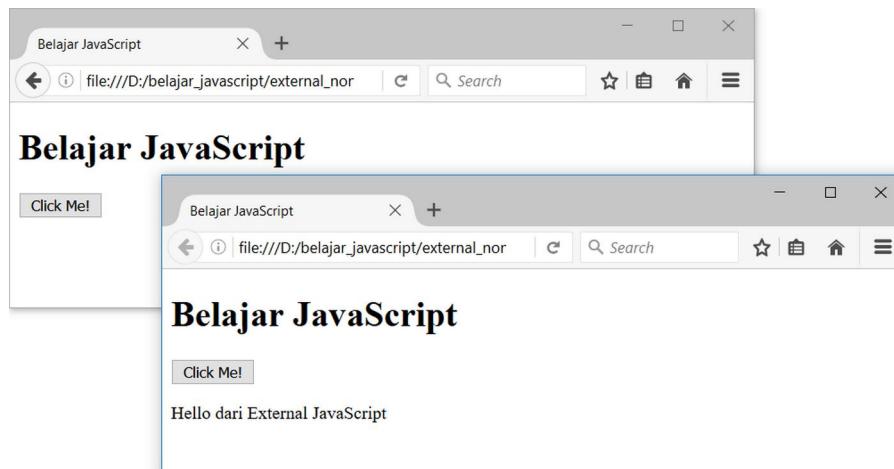
```

1 var tombol = document.getElementById("tombol");
2 var hasil = document.getElementById("hasil");
3
4 function hello(){
5   hasil.innerHTML = "Hello dari External JavaScript";
6 }
7
8 tombol.addEventListener("click", hello);

```

Untuk saat ini, kode program diatas tidak perlu anda pahami dulu karena kita bahas sepanjang buku ini. Sebagai gambaran, kode diatas berfungsi untuk menampilkan pesan "Hello dari External JavaScript" ketika tombol "Click Me!" di-klik, tetapi tidak menggunakan jendela popup seperti efek dari fungsi `alert()`.

Silahkan jalankan file HTML tersebut dan jika tidak ada masalah, teks "Hello dari External JavaScript" akan tampil di bawah tombol "Click Me!".



Gambar: Pesan "Hello dari External JavaScript" tampil ketika tombol "Click Me!" di-klik

Agar bisa berjalan, kode JavaScript harus dieksekusi **setelah** HTML selesai di proses, atau lebih tepatnya setelah tag `<button id="tombol">` dan `<p id="hasil">` di proses oleh web browser. Ini karena di dalam `kodeku2.js` saya harus mengakses kedua element ini.

Sebagai bukti, silahkan pindahkan baris `<script src="kodeku2.js"></script>` ke bagian `<head>`, seperti berikut:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <script src="kodeku2.js"></script>
7 </head>
8 <body>
9   <h1>Belajar JavaScript</h1>
10  <button id="tombol">Click Me!</button>
11  <p id="hasil"></p>
12 </body>
13 </html>
```

Saat kode program dijalankan, teks "Hello dari External JavaScript" tidak akan tampil, yang artinya ada sesuatu yang salah.

Dengan menempatkan kodeku2.js di bagian `<head>`, artinya kode JavaScript akan diproses **sebelum** tag `<body>`. Padahal supaya bisa bekerja, kodeku2.js harus menunggu tag `<body>` di proses terlebih dahulu.

Dalam situasi seperti inilah kita bisa menambahkan atribut **defer** ke dalam tag `<script>`. Efeknya, web browser akan menunda menjalankan JavaScript hingga HTML selesai diproses seluruhnya. Berikut modifikasi dari kode program kita:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <script src="kodeku2.js" defer></script>
7 </head>
8 <body>
9   <h1>Belajar JavaScript</h1>
10  <button id="tombol">Click Me!</button>
11  <p id="hasil"></p>
12 </body>
13 </html>
```

Sekarang, tombol "Click Me!" sudah kembali berjalan sebagaimana mestinya.

Tambahan atribut **async** dan **defer** dari HTML5 membawa perubahan terkait posisi terbaik peletakan kode JavaScript. Standar saat ini adalah menempatkan kode JavaScript di bagian `<head>` dengan tambahan atribut **async**. Alasannya, web browser bisa langsung mengeksekusi file JavaScript pada saat yang bersamaan dengan proses kode HTML, sehingga website dapat ditampilkan dengan lebih cepat (tidak mengalami **Render-Blocking JavaScript**).

Untuk kode JavaScript yang tidak terlalu penting (dan bisa menunggu), tambahkan atribut **defer**. Metode ini dibahas di forum stackoverflow.com: [Where is the best place to put <script> tags in HTML markup?](http://stackoverflow.com/questions/436411/where-is-the-best-place-to-put-script-tags-in-html-markup)<sup>6</sup>

Atribut **async** dan **defer** sebenarnya juga memiliki kelemahan. Kedua atribut ini bagian dari HTML5, yang artinya hanya bisa berjalan di web browser yang relatif baru. Web browser “lawas” seperti Internet Explorer 8 ke bawah belum mendukungnya.

Tapi karena penggunaan IE jadul ini sudah semakin sedikit, anda boleh mengabaikannya. Kecuali jika anda merancang website yang mayoritas pengunjungnya masih pakai Windows XP (web browser bawaan adalah IE 6).



Pembahasan tentang atribut **async** dan **defer** mungkin sedikit rumit, terutama jika anda belum pernah membuat kode JavaScript sebelum ini. Jika kurang paham, silahkan kembali setelah mempelajari bab-bab berikutnya.

Sebagai tambahan, atribut **async** dan **defer** hanya berlaku untuk **external JavaScript**. Untuk internal JavaScript, atribut ini akan diabaikan dan posisi terbaik tetap di bagian bawah tag `<body>`.

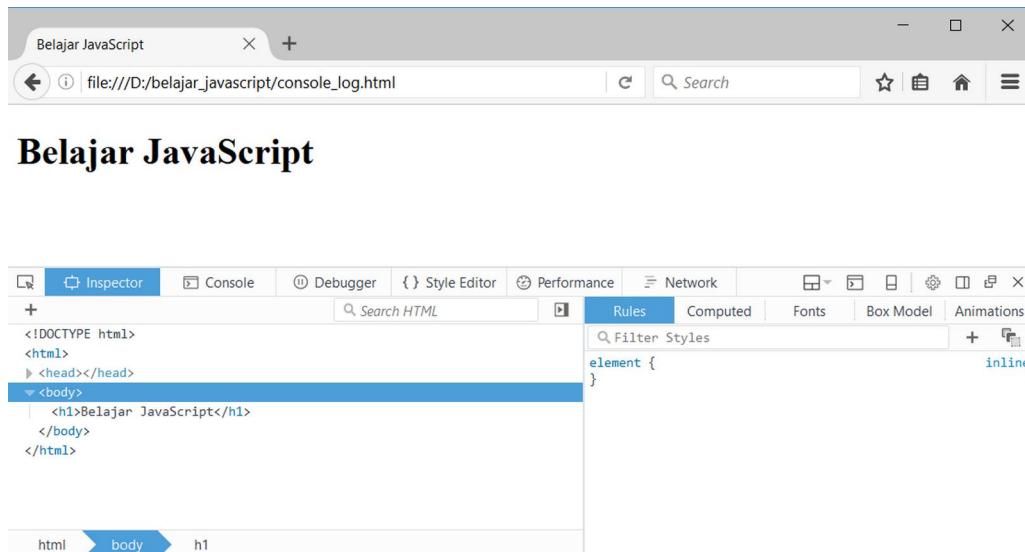
## 3.8 Web Developer Tools

Berbeda dengan mayoritas bahasa pemrograman lain, secara default kita tidak bisa melihat pesan error dari JavaScript. Padahal ini sangat penting selama pembuatan kode program. Tidak ada yang lebih membuat pusing dari program yang tidak berjalan, namun tidak tahu salahnya dimana.

Untuk menampilkan pesan error JavaScript, kita bisa menggunakan menu **Web developer tools** bawaan web browser. Setiap web browser modern memiliki tools seperti ini. Cara paling cepat dengan menekan kombinasi tombol **CRTL + SHIFT + I**. Berikut tampilannya di Mozilla Firefox:

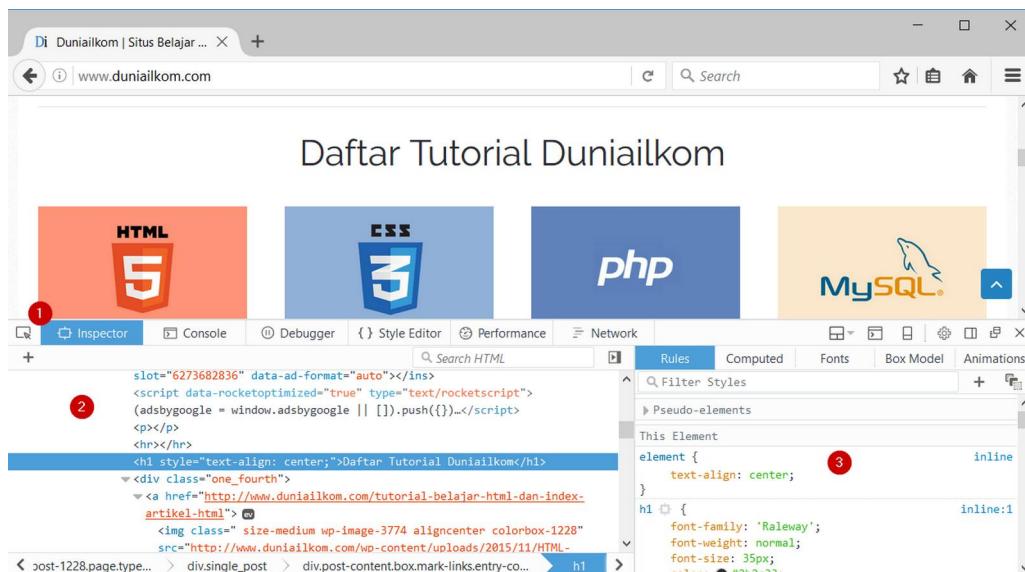
---

<sup>6</sup><http://stackoverflow.com/questions/436411/where-is-the-best-place-to-put-script-tags-in-html-markup>



Gambar: Tampilan jendela Web Developer Tools pada Mozilla Firefox

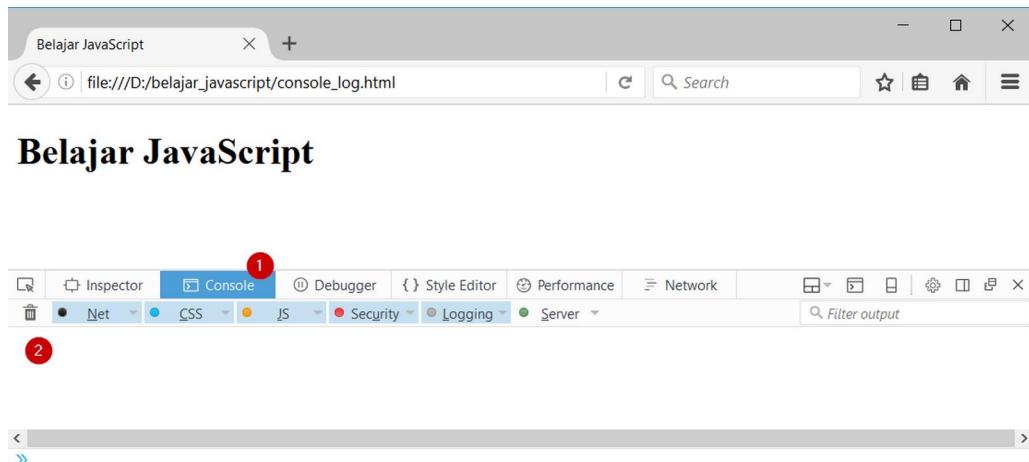
Jendela **Web developer tools** sangat berguna dalam proses pembuatan web, tidak hanya untuk JavaScript saja tapi juga untuk merancang kode HTML dan CSS. Disini terdapat beberapa tab, seperti **Inspector**, **Console**, **Debugger**, dll. Tab yang akan sering kita akses adalah **Inspector** dan **Console**.



Gambar: Tampilan tab Inspector dari Web Developer Tools

Tab **Inspector** (1) bisa digunakan untuk menelusuri seluruh kode HTML yang terdapat di dalam halaman web (2), di sisi kiri kita bisa melihat kode CSS yang digunakan oleh tag HTML tersebut (3). Jika anda sering mengedit kode CSS, tab **Inspector** ini sangat bermanfaat untuk melihat dan menjalankan kode CSS tanpa perlu mengubah file asli.

Tab yang sering kita akses selama membuat kode program JavaScript adalah tab **Console**, yang berada di sebelah kanan tab **Inspector** :



Gambar: Tampilan tab Console dari Web Developer Tools

Tab **Console** (1) digunakan untuk menampilkan pesan error JavaScript (apabila ada). Pesan error ini akan tampil di bagian bawah(2).

Mari langung kita praktekkan, silahkan ketik kode program berikut:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <script>
10    alert("Hello World");
11   </script>
12 </body>
13 </html>
```

Jika anda menggunakan **Komodo Edit**, sebenarnya aplikasi text editor ini akan langsung menginfokan ada sesuatu yang salah:



```

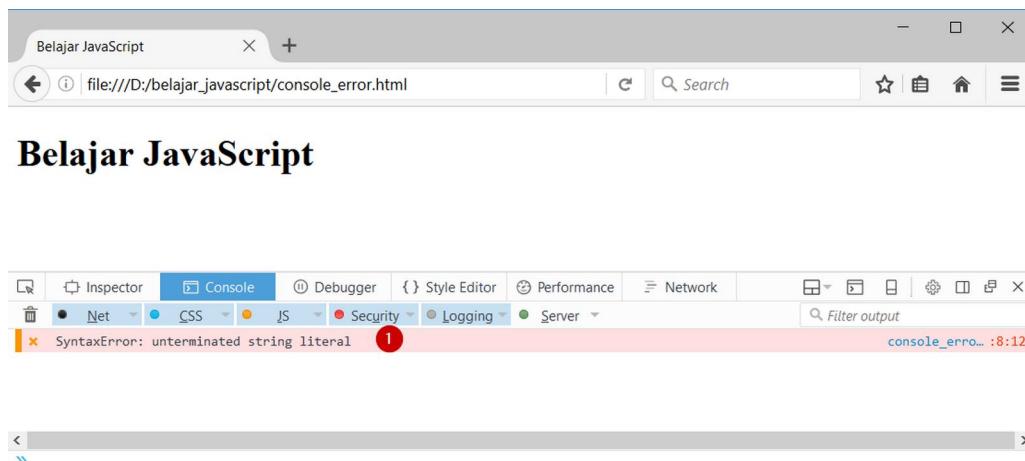
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <script>           JavaScript: Unclosed string.
10  alert("Hello World);
11 </script>
12 </body>
13 </html>

```

Gambar: Komodo Edit “protes” karena ada kode program yang salah

Tapi mari kita abaikan untuk saat ini dan jalankan kode program diatas. Seperti biasa, web browser tidak menampilkan pesan error apapun. Tapi kita sadar bahwa ada sesuatu yang salah karena jendela popup “Hello World” tidak tampil.

Baik, mari akses tab **Console** dari **Web Developer Tools**:



Gambar: Tab Console menampilkan error JavaScript

Tampak satu baris kalimat: *SyntaxError: unterminated string literal*. Inilah pesan kesalahan dari kode program kita. Saya lupa menutup string “Hello World” dengan tanda kutip. Yang tertulis adalah:

```

<script>
  alert("Hello World);
</script>

```

Dimana seharusnya ditulis:

```

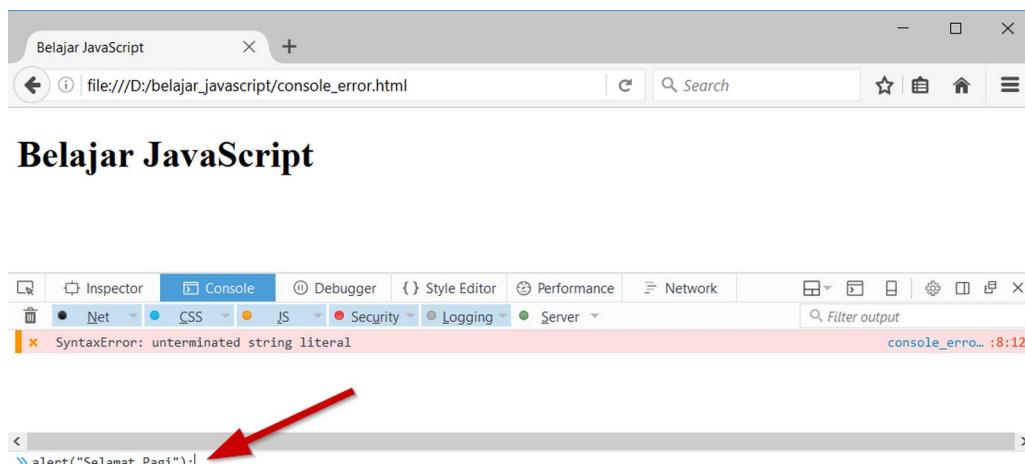
<script>
  alert("Hello World");
</script>

```

Pesan error seperti ini sangat sangat berguna. Apabila kode yang anda buat tidak berjalan sebagaimana mestinya, hal pertama yang harus dilakukan adalah memeriksa tab **Console** ini.

Selain menampilkan pesan error, di dalam tab **Console** kita juga bisa menjalankan kode program JavaScript secara langsung, tanpa harus menulisnya di dalam file HTML. Silahkan ketik kode berikut di baris paling bawah, kemudian tekan **Enter**:

```
alert("Selamat Pagi");
```



Gambar: Mengetik langsung kode program JavaScript dari tab Console

Akan tampil jendela popup seperti contoh-contoh kita sebelumnya. Kode JavaScript apapun bisa langsung anda jalankan disini.

### 3.9 Function `console.log()`

Tidak seperti PHP yang memiliki perintah “echo”, di JavaScript kita tidak memiliki cara menampilkan output kode program dengan cepat. Metode yang paling praktis adalah dengan fungsi `alert()`, sebagaimana yang telah kita gunakan sejauh ini.

Meskipun cepat, fungsi `alert()` hanya bisa menampilkan 1 output pada setiap pemanggilan. Selain itu fungsi ini juga men-block eksekusi JavaScript sampai tombol **OK** di klik.

Alternatif lain yang lebih rapi dan elegan adalah menggunakan fungsi `console.log()`. Fungsi ini berguna untuk menampilkan hasil kode program ke tab **Console**.

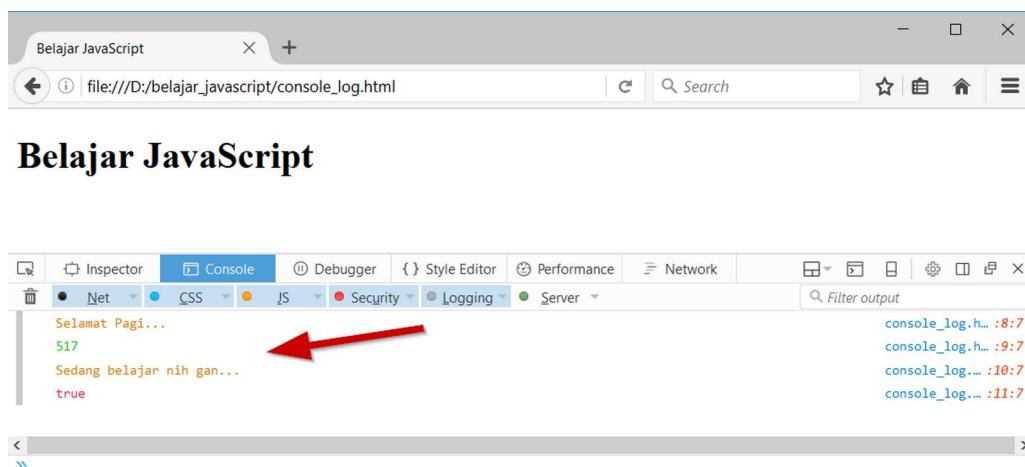
Mari kita coba, silahkan jalankan kode program berikut ini:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <script>
10    console.log("Selamat Pagi...");
11    console.log(99*4+121);
12    console.log("Sedang belajar nih gan...");
13    console.log(10>9);
14  </script>
15 </body>
16 </html>

```

Jika tab **Console** belum terbuka, silahkan akses dan akan terlihat hasilnya:



Gambar: Hasil kode JavaScript tampil di tab Console

Disini, saya menampilkan 4 buah perintah dengan fungsi `console.log()`. Jauh lebih elegan dibandingkan fungsi `alert()`. Tapi perlu diingat, fungsi ini tidak akan menampilkan apa-apa ke dalam halaman web. Kita hanya bisa melihatnya dari tab **Console**.



Sepanjang pembahasan dalam bab-bab berikutnya, saya akan banyak menggunakan fungsi `console.log()` ini. Setidaknya sampai kita masuk ke pembahasan tentang DOM (Document Object Model).

### 3.10 Tag `<noscript>`

Salah satu kelemahan (sekaligus keunggulan) dari JavaScript adalah, pengunjung web bisa mematikan JavaScript yang ada di web browser mereka. Selain itu juga terdapat web browser

yang tidak bisa memproses JavaScript sama sekali, misalnya web browser berbasis teks dan audio untuk teman kita penyandang disabilitas.

Atau pernahkan anda mengunjungi website yang tidak bisa klik kanan? Efek seperti ini didapat dengan menggunakan JavaScript. Jika JavaScript dimatikan, klik kanan akan kembali aktif.

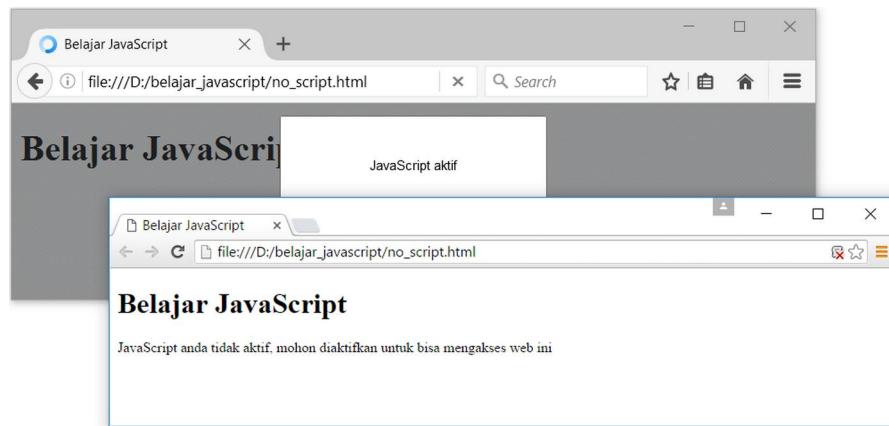
Inilah salah satu alasan untuk tidak sepenuhnya bergantung kepada JavaScript. Seseorang bisa dengan mudah mematikan JavaScript di dalam web browser mereka. Dalam kondisi ideal, website seharusnya tetap bisa diakses meskipun tanpa JavaScript. Fitur keamanan seperti validasi form akan ditangani oleh server (menggunakan PHP).

Tapi jika JavaScript sangat penting di dalam website anda (tidak bisa tidak), tag `<noscript>` bisa menjadi penolong. Tag ini digunakan untuk menampilkan teks keterangan yang hanya bisa terlihat pada web browser yang tidak memiliki JavaScript (atau JavaScriptnya dimatikan).

Mari kita coba, silahkan jalankan kode program berikut:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <script>
10    alert("JavaScript aktif");
11   </script>
12   <noscript>
13     JavaScript anda tidak aktif,
14     mohon diaktifkan untuk bisa mengakses web ini.
15   </noscript>
16 </body>
17 </html>
```

Ketika anda buka di web browser yang JavaScriptnya aktif, akan tampil jendela popup: “JavaScript aktif”, yang berasal dari fungsi `alert()`. Tapi jika JavaScript dimatikan, yang tampil adalah teks: “JavaScript anda tidak aktif, mohon diaktifkan untuk bisa mengakses web ini”.



Gambar: Halaman yang sama diakses dengan JavaScript aktif (atas) dan tidak aktif (bawah)



Jika ingin mematikan fitur JavaScript di web browser, bisa mengikuti panduannya di halaman ini: [How to Disable JavaScript](#)<sup>7</sup>.

---

Dalam bab ini kita telah membahas cara menyisipkan kode program JavaScript ke dalam file HTML. Meskipun **external JavaScript** sangat disarankan, dalam pembahasan sepanjang buku ini saya akan banyak menggunakan **internal JavaScript**. Ini semata-mata agar kode program lebih mudah dipelajari dan lebih singkat. Untuk studi kasus yang cukup rumit atau membutuhkan beberapa halaman, saya akan menggunakan **external JavaScript**.

Bab berikutnya kita akan membahas detail tentang aturan dasar penulisan kode program JavaScript.

---

<sup>7</sup><http://www.wikihow.com/Disable-JavaScript>

# 4. Aturan Dasar, Variabel dan Konstanta

Secara umum, aturan dasar penulisan kode program JavaScript mirip seperti bahasa pemrograman lain. Dalam bab ini saya akan membahas poin-poin penting terkait hal ini, yakni tentang **statement**, **case sensitivity**, **whitespace**, dan **baris komentar** di dalam JavaScript.

## 4.1 Statement

**Statement** adalah sebutan untuk sebuah baris perintah JavaScript. Walaupun saya menggunakan kata “baris”, bisa saja sebuah statement butuh beberapa baris. Atau dalam 1 baris bisa terdiri dari beberapa statement. Setiap statement diakhiri dengan tanda titik koma (semi colon): ‘ ; ’.

Berikut contoh penulisan statement JavaScript:

```
1 <script>
2   alert("Hello World");
3   console.log("Belajar JavaScript");
4   var a = 12; var b = "aku"; var c= 3.14;
5   d = 13 * 5 + 9 - 0.14;
6   let foo = document.getElementById("bar");
7   var elementsBox =
8   document.getElementsByClassName("box");
9 </script>
```

Pada baris ke-4, saya menulis 3 statement dalam 1 baris. Sedangkan di baris ke-7, satu statement dipecah menjadi 2 baris.

Sebenarnya, tanda titik koma untuk mengakhiri statement JavaScript ini adalah *opsional*. Artinya, boleh tidak ditulis sepanjang statement tersebut harus berada dalam baris baru (1 statement, 1 baris).

Sebagai contoh, kode program berikut tidak akan menimbulkan error dan berjalan sebagaimana mestinya:

```
1 <script>
2   alert("Hello World")
3   console.log("Belajar JavaScript")
4   var a = 12
5   var b = "aku"
6   var c= 3.14
7   d = 13 * 5 + 9 - 0.14
8   let foo = document.getElementById("bar")
9 </script>
```

Namun menulis statement seperti ini **sangat sangat tidak disarankan**. Sebaiknya kita tetap menambahkan tanda titik koma untuk mengakhiri setiap statement di dalam JavaScript.

## 4.2 Case Sensitivity

JavaScript termasuk bahasa pemrograman yang bersifat **case sensitif**, artinya **huruf besar dan huruf kecil dianggap berbeda**. Salah menulis huruf sangat sering terjadi, apalagi fungsi-fungsi bawaan JavaScript lumayan panjang, misalnya: `document.getElementById()` dan `document.getElementsByClassName()`. Penulisan fungsi seperti ini harus sama persis, tidak boleh ada salah ketik huruf besar atau kecil.

Hal yang sama juga berlaku untuk penulisan variabel dan keyword. Variabel `nama`, `Nama`, dan `NAMA` merupakan 3 variabel berbeda. Sedangkan untuk penulisan keyword `while`, harus ditulis sebagai ‘`while`’, bukan ‘`While`’ atau ‘`WHILE`’.

## 4.3 Whitespace

**Whitespace** berarti karakter “kosong” seperti spasi, tab, atau baris baru (new line). Secara umum di dalam JavaScript *whitespace* akan diabaikan. Anda bisa membuat seluruh kode program JavaScript di dalam 1 baris panjang maupun memecahnya menjadi beberapa baris.

Kedua contoh ini diproses sama dalam JavaScript:

```
1 <script>
2 var a="Selamat Pagi"; console.log(a); var b = "Good Morning"; alert(b);
3 </script>
```

```

1 <script>
2   var a="Selamat Pagi";
3   console.log(a);
4   var b = "Good Morning";
5   alert(b);
6 </script>

```

Contoh kedua tentu lebih mudah dibaca daripada contoh pertama.

**Indenting** adalah istilah yang digunakan untuk menambahkan spasi atau tab diawal baris kode program. Tujuannya agar kode program lebih mudah dibaca terutama jika kode program tersebut sudah mencapai puluhan atau ratusan baris kode program.

Berikut contoh penggunaan *indenting* dalam JavaScript:

```

1 <script>
2   var pesan = document.getElementById("pesan");
3   var username = document.getElementById("username");
4   var submit = document.forms[0].submit;
5   var pattern = /^[A-Za-z0-9]{6,}$/;
6
7   submit.onclick = function () {
8     if (pattern.test(username.value)) {
9       pesan.innerHTML = "Username sesuai";
10      pesan.className= "betul";
11    }
12  };
13 </script>

```

Jika tanpa indenting, kode program diatas sangat susah untuk dibaca. Kita tidak tahu baris mana yang menjadi awal dan akhir dari sebuah blok program.

Tidak ada aturan khusus penggunaann indenting ini. Sebagian programmer menggunakan tab, dan ada juga yang menggunakan spasi. Untuk saya pribadi, lebih suka menggunakan 2 buah spasi untuk membuat indenting, sebagaimana yang bisa anda lihat sepanjang buku ini.

```

▼ 41   submit.onclick = function () {
▼ 42     if (pattern.test(username.value)) {
43       pesan.innerHTML = "Username sesuai";
44       pesan.className = "betul";
45     }
▼ 46     else {
47       pesan.innerHTML = "Username minimal 6 digit huruf atau angka!";
48       pesan.className = "salah";
49     }
50     return false;
51   };

```

Gambar: Indenting menggunakan spasi sebanyak 2 kali (garis merah)

Di dalam kode program, spasi biasanya juga akan diabaikan. Anda bisa menggunakan spasi untuk memisahkan variabel, statement maupun operator:

```
1 <script>
2   var hasil=4*3;
3   var hasil = 4 * 3;
4 </script>
```

Disini kode program kedua tampak lebih rapi daripada yang pertama.



Penambahan karakter spasi atau tab untuk indenting ini memang akan memperbesar ukuran file JavaScript. Tapi keuntungannya jauh lebih banyak. Kode program kita menjadi lebih mudah dibaca.

Jika anda mengejar performa dan ingin memaksimalkan ukuran file JavaScript, tersedia aplikasi *javascript minifier* atau *javascript compressor*. Aplikasi ini bisa mengubah kode JavaScript menjadi lebih kecil dengan cara membuang semua spasi, komentar dan mengubah nama variabel menjadi 1 atau 2 huruf. Saya akan membahas cara penggunannya nanti. Oleh karena itu jangan takut dengan penambahan beberapa karakter spasi di dalam kode program.

## 4.4 Baris Komentar

**Baris komentar** atau **comment** adalah sebutan untuk kode program yang tidak akan di eksekusi oleh JavaScript. Jika tidak akan diproses, untuk apa ditulis?

Setidaknya terdapat 2 fungsi dari comment:

1. Membuat dokumentasi atau penjelasan tentang kode program yang ada.
2. Menonaktifkan beberapa baris kode program, misalnya jika anda sedang mencoba metode lain tapi tidak ingin menghapus kode program yang sudah ada saat ini.

Untuk membuat komentar, JavaScript menyediakan 2 cara penulisan:

1. Komentar singkat (1 baris), menggunakan karakter // (2 kali forward slash)
2. Komentar panjang, menggunakan tanda /\* sebagai awal komentar dan tanda \*/ sebagai akhir komentar.

Berikut contoh penulisannya:

```

1 <script>
2   // ini adalah komentar dalam 1 baris
3   /* Baris ini juga merupakan komentar */ // ini juga komentar
4
5   /* Komentar ini
6     mencakup beberapa
7     baris */
8
9  /*
10  * Beberapa programmer
11  * menambahkan tanda bintang
12  * agar penulisan komentar
13  * lebih rapi, termasuk editor Komodo Edit
14  */
15 </script>
```

Berikut contoh penggunaannya di dalam kode program JavaScript:

```

1 <script>
2   // Ambil setiap element, simpan ke variabel
3   var pesan = document.getElementById("pesan");
4   var username = document.getElementById("username");
5
6   // Buat regex untuk inputan minimal 6 karakter (digit dan angka)
7   var pattern = /^[A-Za-z0-9]{6,}$/;
8
9   /* Buat fungsi untuk mengecek username.
10    Fungsi ini dibuat menggunakan regular expression */
11
12 submit.onclick = function () {
13   if (pattern.test(username.value)) {
14     pesan.innerHTML = "Username sesuai";
15     pesan.className= "betul"; // Artinya username sudah sesuai
16   }
17 };
18 </script>
```

Dalam contoh diatas saya menggunakan komentar untuk membuat dokumentasi penjelasan tentang cara kerja kode program. Jika anda membuat web bersama tim, komentar seperti ini sangat penting ditambahkan, supaya rekan kerja lain mudah mempelajarinya.

Atau jikapun bekerja sendiri, komentar penting ditulis sebagai “pingingat”. Meskipun saat ini anda paham, anda yakin dalam 1 atau 2 bulan lagi anda akan lupa cara kerja sebagian besar kode program tersebut. Dokumentasi seperti ini bisa membantu kita. Tapi tentu saja tidak semua baris harus ditambahkan komentar. Hanya untuk bagian yang dirasa cukup rumit.

Selain sebagai dokumentasi, komentar juga sering digunakan untuk menghentikan sementara baris kode program. Bisa jadi anda ingin mencari metode lain, atau agar fokus ke bagian yang sedang dikerjakan. Berikut contoh penggunaannya:

```
1 <script>
2 // var pattern = /^[A-Z]{6,}$/;
3 // var pattern = /^[A-Za-z]$/;
4 // var pattern = /^[a-zA-Z0-9]{6,}$/;
5 var pattern = /^[A-Za-z0-9]{6,}$/;
6
7 submit.onclick = function () {
8     if (pattern.test(username.value)) {
9         alert("Username Sesuai");
10    }
11 };
12 </script>
```

Disini saya ingin membuat **regular expression**, tapi karena belum yakin saya mencoba beberapa pola. Jika ternyata tidak berjalan, saya bisa dengan mudah menggunakan kode program yang lama.



Regular expression sendiri akan kita bahas dalam bab khusus. Fokus dari pembahasan ini hanya pada penggunaan komentar untuk menghentikan beberapa baris kode program.

---

Pembahasan tentang aturan dasar penulisan JavaScript memang tidak terlalu banyak. Ini merupakan persiapan untuk memasuki bab selanjutnya yang cukup panjang, yakni mengenai Variabel dan Konstanta di dalam JavaScript.

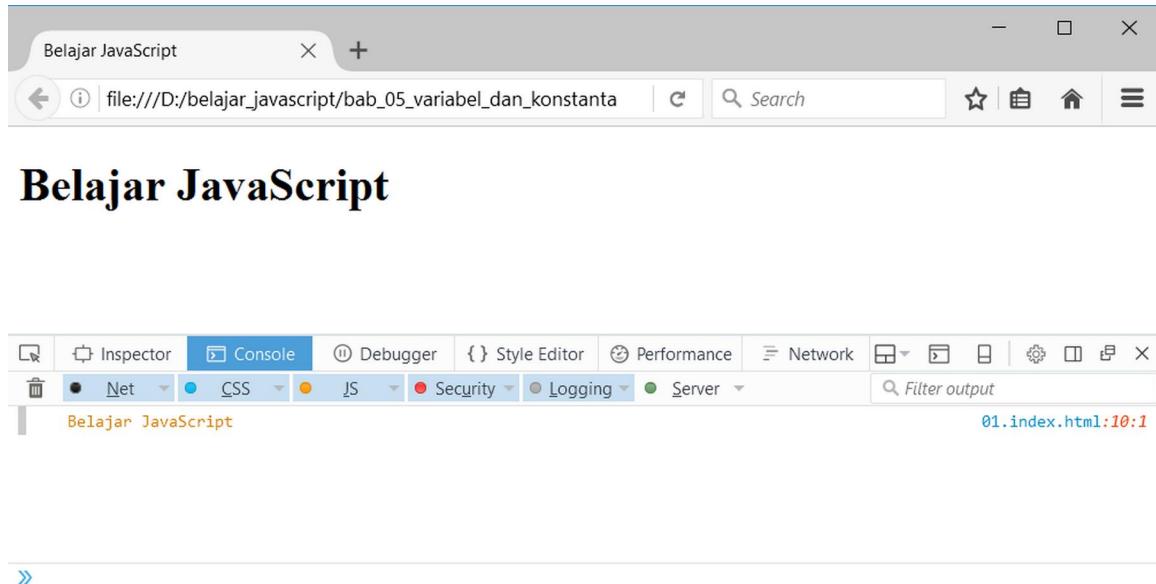
# 5. Variabel dan Konstanta

Variabel, konstanta, dan tipe data merupakan inti dari apa yang kita kerjakan dalam setiap bahasa pemrograman, termasuk JavaScript. Kode program itu sendiri pada dasarnya digunakan untuk mengolah data masuk (*input*), dan menghasilkan data baru (*output*).

Agar dapat diproses, data disimpan ke dalam variabel dan konstanta. Dalam bab ini kita akan membahas lebih dalam tentang ciri khas penulisan variabel dan konstanta JavaScript.

## Function console.log() dan Tab Console dari Web Developer Tools

Mulai dari bab ini dan selanjutnya, saya banyak menggunakan fungsi `console.log()` untuk menampilkan hasil kode program JavaScript. Untuk melihat hasil dari fungsi `console.log()` ini, bisa dengan membuka tab **Console** dari **Web Developer Tools**. Caranya sudah kita praktekkan pada bab ke 3 sewaktu mempelajari cara menjalankan kode program JavaScript.



Gambar: Cara melihat hasil fungsi `console.log()` dari tab **Console**

Agar menghemat tempat, saya tidak akan menulis lengkap kode HTML, tapi hanya bagian `<script>` saja. Jika anda ingin menjalankan kode program yang dibahas, tempatkan kode tersebut ke dalam sebuah file HTML.

Berikut contoh kode HTML yang bisa digunakan:

index.html

---

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <script>
10    // kode JavaScript disini
11    // kode JavaScript disini
12    // kode JavaScript disini
13  </script>
14 </body>
15 </html>
```

---

Posisi tag `<script>` bisa dimana saja, tidak harus di bawah tag `<body>`. Ini karena kita belum mengakses element HTML (setidaknya sampai masuk ke pembahasan tentang **DOM**: *Document Object Model*).

## JavaScript yang “Membosankan”

Dalam bab-bab awal ini, materi JavaScript yang saya bahas mungkin juga agak membosankan. Karena kita belum bisa melakukan “apa-apa” di website, kebanyakan kita hanya mengolah data teks dan menampilkannya menggunakan perintah `console.log()`. Tapi sebenarnya inilah yang disebut bahasa pemrograman JavaScript (**ECMAScript**).

Untuk bisa memanipulasi halaman web, seperti membuat animasi, validasi form atau membuat slideshow, kita harus mengkombinasikan **JavaScript** dengan **DOM** (*Document Object Model*). Secara teknis, DOM merupakan bagian yang terpisah dari JavaScript. Di dalam DOM inilah nantinya pembahasan jadi sangat menarik.

Sebelum bisa menggunakan DOM, anda harus paham JavaScript terlebih dahulu. Jadi, “paksakan” diri untuk mempelajari dasar-dasar JavaScript. Semuanya akan terbayar saat kita masuk ke materi tentang DOM di pertengahan buku nanti.



Jika sebelumnya anda sudah mempelajari bahasa pemrograman lain seperti **PHP**, materi dasar ini juga terasa sangat mirip, namun terdapat bahasan dan aturan penulisan JavaScript yang berbeda dari PHP, oleh karena itu anda juga sebaiknya tetap mempelajari materi dasar ini.

## 5.1 Variabel dalam JavaScript

Secara sederhana, variabel adalah “*penampung*” dari sebuah data. Disebut variabel karena data yang kita simpan bisa berubah-ubah sepanjang kode program (isinya tidak tetap). Agar bisa diakses, setiap variabel harus memiliki “**nama**”.

### Membuat Variabel dengan Perintah “var”

Untuk membuat variabel di dalam JavaScript, kita menggunakan kata kunci atau *keyword* **var**. Sebagai contoh, dalam kode program berikut ini saya membuat sebuah variabel bernama “**angka**”:

```
var angka;
```

Proses pembuatan variabel dikenal juga sebagai **deklarasi variabel**, atau **pendeklarasian variabel**.

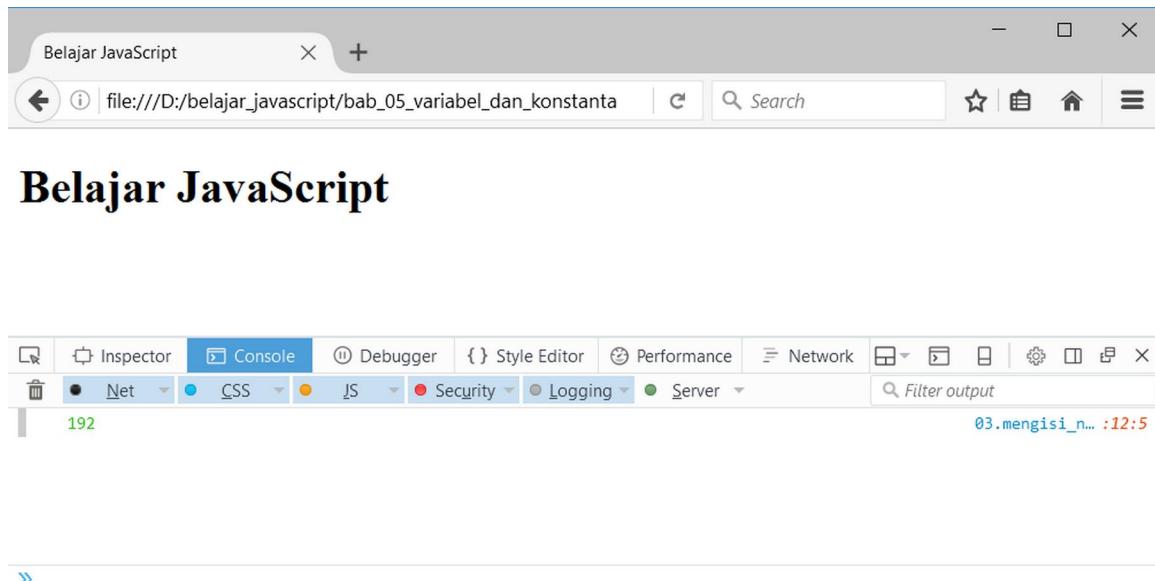
### Mengisi Nilai Variabel

Di dalam JavaScript, sebuah variabel bisa diisi dengan data apa saja, apakah itu teks, angka, atau data lain yang lebih kompleks seperti object.

Untuk memberikan nilai ke dalam variabel, digunakan **operator asignment**, yakni tanda sama dengan “**=**”. Berikut contoh penggunaannya:

```
1 var angka;  
2 angka = 192;  
3 console.log(angka); // 192
```

Dibaris pertama, saya membuat sebuah variabel bernama **angka**. Di baris kedua, variabel ini diberi nilai **192**. Selanjutnya variabel angka ditampilkan menggunakan fungsi `console.log()`. Jika anda menjalankan kode program tersebut, akan tampil angka **192** di tab **Console - Web Developer Tools**.



Gambar: Mengisi dan menampilkan nilai variabel “angka”



Apabila baru pertama kali belajar bahasa pemrograman, anda mungkin sedikit bingung dengan perintah `angka = 192;`. Operasi **asignment** atau memberikan nilai ke sebuah variabel dibaca dari kanan ke kiri. Artinya, 192 “dimasukkan” sebagai nilai ke variabel angka.

Proses pembuatan (*deklarasi*) dan pengisian nilai ke sebuah variabel bisa juga dilakukan dalam satu baris, seperti contoh berikut:

```
1 var angka = 192;
2 console.log(angka); // 192
```

Disini variabel **angka** saya deklarasikan dan langsung diinput nilai **192**, proses seperti ini dikenal juga dengan istilah **inisialisasi variabel**, yakni variabel di deklarasikan dan langsung diberikan nilai awal.

Dalam 1 baris, kita juga bisa melakukan proses deklarasi dan inisialisasi untuk beberapa variabel sekaligus. Antar variabel ini dipisahkan dengan tanda koma, seperti contoh berikut:

```
1 var a, b;
2 a = "Selamat Pagi";
3 b = 2000;
4
5 console.log(a); // Selamat Pagi
6 console.log(b); // 2000
7
8 var c = "Selamat Malam", d = 99;
9
10 console.log(c); // Selamat Malam
11 console.log(d); // 99
```

Disini saya membuat 4 variabel: **a**, **b**, **c**, dan **d**. Variabel **a** berisi teks "Selamat Pagi", variabel **b** berisi angka 2000, variabel **c** berisi teks "Selamat Malam", dan variabel **d** berisi angka 99.

Seperti yang terlihat, untuk menginput nilai teks, saya menggunakan tanda kutip. Nilai teks di dalam JavaScript dikenal sebagai **string**, atau lebih tepatnya **tipe data string**. Untuk nilai angka, tidak perlu ditulis dengan tanda kutip. Di dalam JavaScript nilai angka dikenal sebagai **tipe data number**. Lebih jauh tentang tipe data akan saya bahas dalam bab selanjutnya.

## JavaScript sebagai Typeless Programming Language

JavaScript termasuk ke dalam bahasa pemrograman **Typeless Programming Language**, yakni kelompok bahasa pemrograman yang variabelnya bisa diisi dengan tipe data apa saja tanpa harus dideklarasikan terlebih dahulu.

Sebagai bandingan, di dalam bahasa pemrograman **PASCAL**, **C**, **C++** maupun **JAVA**, setiap variabel hanya bisa diisi dengan data yang telah ditentukan. Apabila variabel di set untuk menampung nilai text, kita tidak bisa mengisinya dengan angka. Bahasa pemrograman jenis ini dikenal sebagai **Typed Programming Language**.

Di dalam JavaScript, kita bisa mengisi data variabel dengan nilai apa saja, dan mengubahnya sepanjang kode program, seperti contoh berikut:

```
1 var foo;
2 foo = "Selamat Pagi";
3 console.log(foo); // "Selamat Pagi"
4
5 foo = 1234.56;
6 console.log(foo); // 1234.56
7
8 foo = "Selamat Malam";
9 console.log(foo); // Selamat Malam
10
11 foo = false;
```

Pada awal program, saya mendeklarasikan sebuah variabel bernama **foo**. Di baris ke-2, variabel ini diinput teks "Selamat Pagi", dengan demikian, variabel **foo** berisi data **string**. Di baris ke-5, nilainya diubah menjadi angka 1234.56 yang merupakan tipe data **number**.

Di baris ke 8, saya mengubah kembali isi variabel **foo** menjadi string "Selamat Malam", kemudian mengubahnya dengan nilai **false** yang merupakan tipe data **boolean**.

Inilah maksud dari nilai variabel yang bisa diubah-ubah sepanjang kode program. Setiap kali nilai baru diisi ke dalam variabel **foo**, nilai yang lama otomatis tertimpa, termasuk jika nilai ini berasal dari variabel lain, seperti contoh berikut:

```
1 var foo, bar;
2
3 foo = "Hello World";
4 console.log(foo); // Hello World
5
6 bar = foo;
7 console.log(bar); // Hello World
8
9 var baz = bar;
10 console.log(baz); // Hello World
11
12 foo = "Hello Indonesia";
13 console.log(foo); // Hello Indonesia
14 console.log(bar); // Hello World
15 console.log(baz); // Hello World
```

Disini saya mendeklarasikan 2 buah variabel: **foo** dan **bar**. Variabel **foo** selanjutnya diisi string "Hello World".

Pada baris ke-6, saya membuat perintah **bar** = **foo**. Artinya, isi dari variabel **foo** di copy ke variabel **bar**. Sekarang, baik variabel **foo** maupun **bar** sama-sama berisi "Hello World".

Di baris ke-9, saya membuat variabel baru, **baz** yang langsung diisi dengan nilai dari variabel **bar**. Hasilnya, ketiga variabel, yakni **foo**, **bar**, dan **baz** berisi string yang sama: "Hello World".

Yang juga patut di perhatikan, dalam setiap operasi ini nilai **string** "Hello World" di copy dari satu variabel ke variabel lain. Jika saya mengubah nilai satu variabel, seperti pada baris ke 12, yang berubah hanya variabel itu saja, dimana **foo** sekarang berisi string "Hello Indonesia". Sedangkan untuk variabel **bar** dan **baz** tetap berisi "Hello World".

Konsep ini berbeda jika yang dicopy adalah **object** (akan kita bahas dalam bab berikutnya).

Apabila anda sering mengikuti tutorial programming dari situs berbahasa inggris, nama variabel **foo**, **bar**, dan **baz** sering digunakan. Ketiganya dikenal sebagai **dummy variabel**, yakni variabel yang fungsinya hanya sebagai contoh. Mirip seperti teks "*“Lorem Ipsum dolor sit amet”*" dalam bidang design, atau kalau kita di Indonesia sering bercerita tentang si "*fulan*".

Tujuan menggunakan nama yang tidak bermakna seperti ini adalah agar kita fokus ke penjelasan program.

Istilah lain untuk variabel **foo**, **bar**, dan **baz** adalah *placeholder names* atau *metasyntactic variables*. Nama variabel ini [dipopulerkan sejak tahun 1964<sup>a</sup>](#) oleh peneliti di MIT dengan bahasa pemrograman **LIPS**. Aplikasi media player **foobar2000<sup>b</sup>** juga berasal dari nama variabel ini.

<sup>a</sup><http://stackoverflow.com/questions/4868904/what-is-the-origin-of-foo-and-bar>

<sup>b</sup><http://www.foobar2000.org>

## 5.2 Aturan Penamaan Variabel (Identifier)

Nama variabel di dalam JavaScript harus mengikuti aturan dari *identifier*. **Identifier** adalah sebutan untuk **nama dari sesuatu** di dalam sebuah bahasa pemrograman (tidak hanya JavaScript saja).

Seperti yang sudah dipraktekkan, kita bisa memberi nama apa saja untuk variabel, apakah itu **angka**, **foo**, **bar**, **andi**, atau **username**. Selain variabel, kita juga bebas untuk membuat nama **konstanta**, **function**, maupun **object**. Semua inilah yang termasuk kedalam kelompok **identifier**.

**Identifier** di dalam JavaScript memiliki aturan sebagai berikut:

- Bisa terdiri dari huruf, angka, garis bawah “ \_ ” (*underscore*), dan tanda dollar “ \$ ” (*dollar sign*). Selain itu, dianggap sebagai karakter ilegal (tidak boleh digunakan).
- Karakter pertama dari *identifier* tidak boleh berupa angka. Angka hanya bisa digunakan sebagai karakter kedua dan seterusnya.
- Bersifat **case sensitive**, dimana huruf besar dan kecil dianggap berbeda.
- Harus selain dari **reserved keyword**, yakni kata khusus yang berfungsi sebagai perintah di dalam pemrograman JavaScript, seperti **var**, **while**, **function**, dll.

Karena *variabel* juga termasuk ke dalam *identifier*, nama variabel harus mengikuti aturan-aturan diatas.

Berikut contoh penulisan variabel yang salah:

```
1 var 9naga;           // diawali dengan angka
2 var satu-satu;      // terdapat karakter "-"
3 var satu satu;     // terdapat spasi
4 var satu%lima;     // terdapat karakter "%"
5 var continue;       // merupakan reserved keyword
```

Berikut contoh penulisan variabel yang benar:

```
1 var aa123;
2 var belajar_bahasa_javascript;
3 var jumlahTotal;
4 var $box;
5 var _begin;
```

Khusus 2 contoh terakhir sebaiknya tidak digunakan (walaupun penulisan seperti itu tidak salah). Variabel dengan karakter awal tanda dollar sering dipakai oleh library JavaScript seperti **jQuery**, sehingga ada kemungkinan bisa menimpa variabel lain.

Variabel yang diawali dengan karakter *underscore* umumnya digunakan sebagai variabel khusus atau internal variabel. Jika anda tidak bermaksud membuat sebuah kelompok variabel khusus, hindari penulisan seperti ini.

Nama variabel yang digunakan sebaiknya juga “bermakna”. Daripada menggunakan variabel seperti `a`, `b`, atau `c`, akan jauh lebih baik memilih nama seperti `username`, `totalBarang`, `jumlahPeserta`, dst.

Untuk **reserved keyword**, terdapat sekitar seratusan kata yang tidak boleh digunakan karena bagian dari internal JavaScript itu sendiri, seperti `var`, `boolean`, `string`, `number`, `null`, dll. Umumnya keyword ini menggunakan bahasa Inggris. Jika anda memilih membuat variabel menggunakan bahasa Indonesia, bisa dipastikan tidak akan bentrok dengan *reserved keyword*.

Daftar lengkap *reserved keyword* dalam JavaScript bisa dilihat dari link berikut: [JavaScript Reserved Words<sup>1</sup>](#).

## 5.3 Gaya Penulisan Variabel: CamelCase

Di dalam JavaScript, terdapat kebiasaan unik untuk menulis nama variabel, yakni menggunakan gaya *CamelCase*.

**CamelCase** adalah cara penulisan variabel dimana jika sebuah variabel terdiri dari beberapa kata, huruf pertama dari kata kedua dan seterusnya diubah menjadi huruf besar, seperti: `banyakAnggota`, `totalBiaya`, `mainBox`, atau `jumlahKlikSatuHari`. Jika variabel tersebut hanya terdiri dari 1 kata, ditulis dengan huruf kecil semua.

Anda tidak dipaksa mengikuti kebiasaan ini, namun mayoritas programmer dan tutorial terkait JavaScript menggunakan gaya penulisan **CamelCase**.

Jika anda pernah belajar CSS dan PHP, dapat diperhatikan bahwa setiap bahasa pemrograman punya kebiasaan masing-masing.

Di CSS kita menggunakan cara penulisan selector yang dipisah dengan tanda “ - ”, seperti `main-box`, `left-sidebar`, dan `single-post`. Di PHP kita mengenal **Snake Case**, yakni menggunakan huruf kecil dan tanda underscore sebagai pemisah variabel, seperti `jumlah_barang`, `nama_dosen`, dan `alamat_siswa`. Di JavaScript menggunakan **CamelCase**.

Ini semua merupakan keunikan dari masing-masing bahasa pemrograman, dan bukanlah sebuah keharusan (lebih ke kebiasaan). Semoga anda tidak “terindimidas” dengan aturan seperti ini. Terlebih karena di web programming, satu halaman bisa terdapat kode **HTML**, **CSS**, **PHP**, **MySQL** dan **JavaScript** sekaligus, yang masing-masingnya memiliki gaya penulisan sendiri-sendiri.

## 5.4 Membuat Variabel Tanpa Perintah var

Di dalam JavaScript, sebenarnya kita bisa membuat variabel tanpa perintah `var`, yakni dengan cara langsung mengisi variabel tersebut (tanpa di deklarasikan), seperti contoh berikut:

<sup>1</sup>[http://www.w3schools.com/js/js\\_reserved.asp](http://www.w3schools.com/js/js_reserved.asp)

```
1 angka = 123456;
2 console.log(angka); // 123456
3
4 nama = "Andi";
5 console.log(nama); // Andi
```

Disini saya tidak menulis `var angka = 123456`, tapi langsung `angka = 123456`. Kode program diatas berjalan sukses di web browser dan tidak tampil error apapun. Jadi kenapa harus repot-repot menambahkan perintah `var`?

Cara penulisan seperti ini **tidak disarankan**. Salah satu alasannya bisa mendatangkan bug untuk web browser tertentu seperti IE. Secara teknis, juga ada perbedaan mendasar berkaitan dengan variabel scope (akan kita bahas pada bab tentang *function*). Sedapat mungkin selalu menambahkan perintah `var` saat membuat variabel.

## 5.5 Mengenal Strict Mode

Menulis variabel tanpa perintah `var` seperti yang kita coba sebelum ini, dianggap sebagai kebiasaan yang buruk (*bad practice*), atau jika menggunakan istilah lain: **deprecated**, yang artinya sebaiknya tidak digunakan.

Salah satu kelemahan (sekaligus keunggulan) dari JavaScript adalah, web browser kadang tidak menampilkan error dimana seharusnya terdapat error. Istilah programmingnya: “**silent error**”.

Disatu sisi ini seperti membantu kita, tapi bisa menjadi bug yang tidak disangka-sangka. Contoh kasusnya seperti berikut:

```
1 var hargaBarang = 12000;
2 var ongkosKirim = 5000;
3 var totalBiaya = hargaBarang + ongkosKirim;
4
5 var pajak = 10/100 * totalBiaya;
6
7 totalbiaya = totalBiaya + pajak;
8 console.log(totalBiaya);
```

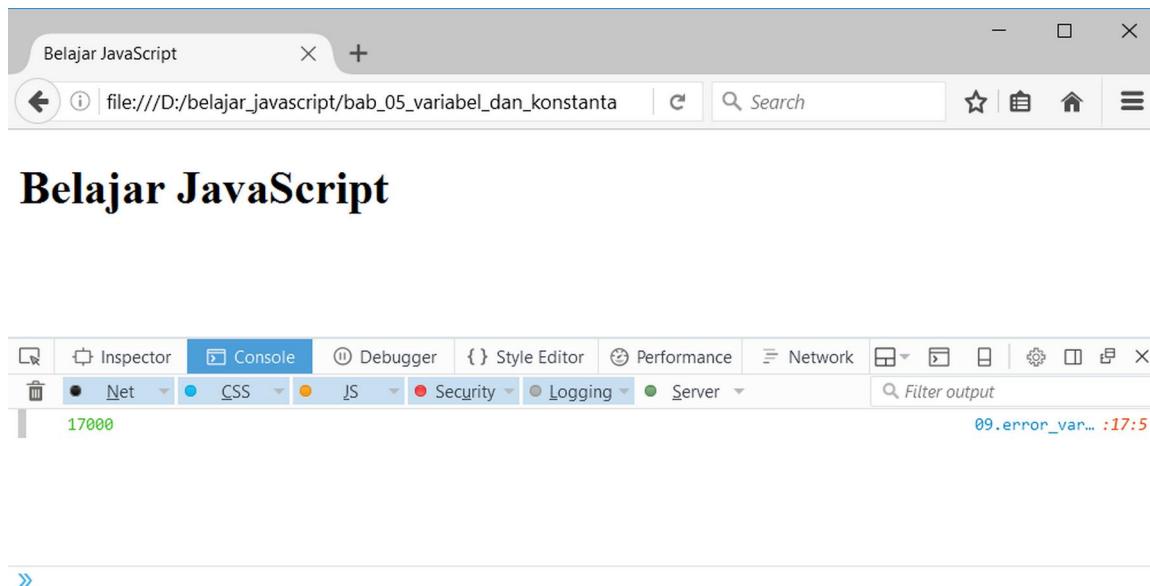
Kita belum masuk ke pembahasan tentang operator, namun kode program diatas cukup sederhana.

Di awal program saya membuat 2 buah variabel: `hargaBarang` dan `ongkosKirim`. Selanjutnya saya menghitung `totalBiaya` dengan menjumlahkan kedua variabel ini. Hasilnya akan disimpan ke dalam variabel `totalBiaya` yang berjumlah **17000**.

Ternyata total biaya ini belum final, karena masih ada pajak yang mesti ditambahkan. Saya menghitung pajak sebesar 10% dari total biaya, yang disimpan kedalam variabel `pajak`. Nilai pajak adalah  $10/100 * 17000 = 1700$ .

Terakhir saya menjumlahkan variabel `totalBiaya + pajak`, yang kembali disimpan kedalam variabel `totalBiaya`. Hasilnya  $17000 + 1700 = 18700$ .

Agar lebih yakin, mari jalankan kode program diatas:



Gambar: Variabel `totalBiaya` berisi nilai 17000!

Ternyata hasilnya adalah **17000!**, bukan **18700**. Apakah ada yang salah? JavaScript sendiri tidak mengeluarkan error apapun.

Letak kesalahan dari kode program diatas adalah, saya salah menulis perintah `totalbiaya = totalBiaya + pajak`. Seharusnya `totalBiaya = totalBiaya + pajak`, perhatikan perbedaan penulisan variabel `totalbiaya` dengan `totalBiaya`. Di dalam JavaScript, variabel bersifat case sensitif, artinya `totalbiaya` tidak sama dengan `totalBiaya`.

Pada baris ke-7, kode tersebut memerintahkan JavaScript untuk membuat sebuah variabel baru: `totalbiaya`. Variabel inilah yang bernilai **18700**, sedangkan variabel `totalBiaya` tetap **17000**.

Anda bisa bayangkan seandainya kesalahan seperti ini terjadi di aplikasi yang sudah rilis. Perusahaan akan merugi karena total biaya lebih murah dari yang seharusnya.

Salah satu cara untuk menghindari error seperti ini adalah dengan memaksa JavaScript menampilkan pesan kesalahan jika sebuah variabel dibuat tanpa perintah `var`, sebuah kebiasaan yang sebenarnya sangat baik.

Namun hal ini tidak bisa dijadikan fitur wajib di **ECMAScript** karena sudah banyak kode program JavaScript yang terlanjur beredar (dimana variabel banyak dibuat tanpa perintah `var`). Bisa-bisa jutaan situs web berhenti karena ada aturan baru yang ditambahkan tiba-tiba ke dalam JavaScript.

Untuk “memaksa programmer” menghentikan kebiasaan ini (tapi tidak mewajibkan), JavaScript memiliki mode pilihan dengan aturan yang lebih ketat, yakni **Strict Mode**. Fitur ini hadir di **ECMAScript 5**.

**Strict Mode** akan membuat web browser menampilkan error dimana sebelumnya hanya ada “silent error”. Salah satunya ketika membuat variabel tanpa perintah `var`.

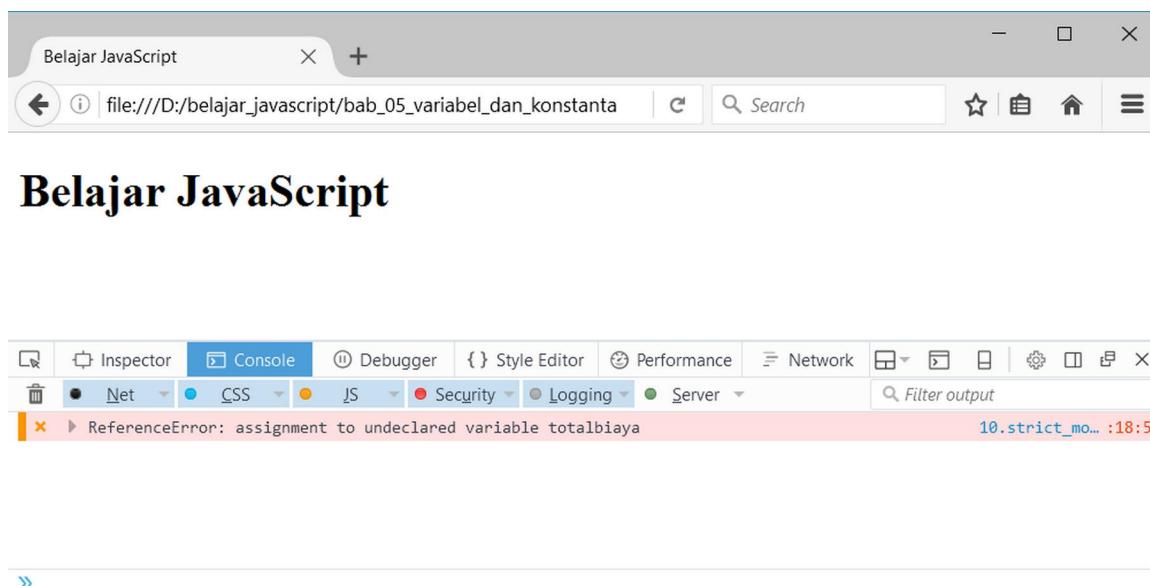
Untuk masuk ke dalam **Strict Mode**, tambahkan string "use strict"; di baris pertama kode JavaScript, seperti contoh berikut:

```

1 "use strict";
2
3 var hargaBarang = 12000;
4 var ongkosKirim = 5000;
5 var totalBiaya = hargaBarang + ongkosKirim;
6
7 var pajak = 10/100 * totalBiaya;
8
9 totalbiaya = totalBiaya + pajak;
10 console.log(totalBiaya);

```

Sekarang, jika anda menjalankan kode program diatas, hasilnya adalah: *ReferenceError: assignment to undeclared variable totalbiaya*. Artinya terdapat sebuah variabel totalbiaya yang tidak dideklarasikan, tapi langsung berisi nilai.



Gambar: Pesan error yang ditampilkan karena Strict Mode

**Strict mode** memaksa JavaScript menampilkan error pada kode program yang seharusnya bisa berjalan “normal”. Tujuannya, meminimalisir kemungkinan bug karena penulisan yang salah, typo, dan berbagai hal lain. **Strict mode** sepenuhnya opsional dan mungkin tidak bisa selalu anda gunakan, terutama jika terdapat kode JavaScript pendahulu yang terlalu rumit untuk diubah semuanya.

Teks "use strict"; harus ditempatkan di baris paling awal kode JavaScript, atau dibaris paling awal dari sebuah function. Untuk web browser yang belum mendukung ECMAScript 5, string "use strict"; akan diabaikan.

## Error Strict Mode di Komodo Edit

Jika anda menggunakan text editor **Komodo Edit**, akan keluar pesan error pada saat mengetik "use strict";. Pesan errornya adalah *JavaScript: Use the function form of "use strict"*.

```

5   <title> Belajar Javascript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <script>      JavaScript: Use the function form of "use strict".
10  "use strict";
11
12  var hargaBarang = 12000;
13  var ongkosKirim = 5000;
14  var totalBiaya = hargaBarang + ongkosKirim;

```

Gambar: Inline error correction Komodo Edit

Error terjadi karena **Komodo Edit** tidak menyarankan menggunakan "use strict"; secara global seperti ini. Alasannya, *strict mode* akan aktif untuk seluruh kode JavaScript, yang bisa jadi ada kode program lain yang belum mendukung aturan dari *strict mode*.

Kita disarankan menjalankan **strict mode** ke dalam sebuah *anonym function* (fungsi yang tidak memiliki nama), yang ditulis seperti berikut ini:

```
(function () {
  "use strict";
  // Kode Program JavaScript disini...
}());
```

Tujuannya, agar bisa memisahkan kode program apa saja yang ingin dijalankan di dalam **Strict mode** dan mana yang tidak. Untuk kode program yang ingin dijalankan di dalam **Strict mode**, tempatkan ke dalam function ini. Selain itu, kode program di luar function bisa dijalankan dengan cara biasa (dikenal dengan istilah **sloppy mode**).

Berikut contoh kode program totalBiaya sebelumnya jika menggunakan *anonym function*:

```

1 (function () {
2   "use strict"; // seluruh kode program dijalankan dalam strict mode
3
4   var hargaBarang = 12000;
5   var ongkosKirim = 5000;
6   var totalBiaya = hargaBarang + ongkosKirim;
7
8   var pajak = 10/100 * totalBiaya;
9
10  totalBiaya = totalBiaya + pajak;
11  console.log(totalBiaya);
12 }());
13
```

```
14 // diluar ini dijalankan tanpa strict mode (sloppy mode)
15 a = 15000;
16 console.log(a);
```

Sekali lagi, Strict Mode lebih ke saran penggunaan, dan bukan sebuah keharusan.

## 5.6 Membuat Variabel dengan Perintah let

EcmaScript 6 membawa fitur baru ke dalam JavaScript, yakni menggunakan perintah **let** untuk membuat variabel (sebagai alternatif dari **var**). Cara penggunaannya sama persis dengan **var**, berikut contohnya:

```
1 let a = "Selamat Pagi";
2 let b, c;
3 b = 9999.9999;
4 c = true;
5
6 console.log(a); // Selamat Pagi
7 console.log(b); // 9999.9999
8 console.log(c); // true
```

Perbedaan mendasar dari **var** dan **let** adalah terkait dengan variabel scope, yakni di bagian mana sebuah variabel masih bisa diakses. Penjelasan mengenai variabel scope akan saya bahas pada bab tentang function.

Untuk saat ini anda cukup memahami bahwa di ES 6, deklarasi variabel bisa menggunakan kata kunci **let**, dan **var** juga tetap didukung (bukan sebagai pengganti). Karena masih relatif baru, perintah **let** ini lebih banyak digunakan pada kode program JavaScript di server seperti **NodeJS**. Sebab, ada kemungkinan web browser belum mendukung **ECMAScript 6**.

## 5.7 Konstanta dalam JavaScript

**Konstanta** dapat dikatakan sebagai variabel yang tidak bisa diubah sepanjang kode program. Setelah konstanta ditulis dan diberi nilai awal, isi konstanta tersebut tidak bisa ditukar dengan nilai lain.

**Konstanta** sebenarnya fitur wajib yang selalu ada di setiap bahasa pemrograman. Namun di JavaScript sendiri, konstanta baru hadir pada **ECMAScript 6**. Untuk membuat konstanta, kita menggunakan perintah **const**. Berikut contohnya:

```

1 const PI = 3.14;
2 const SALAM_PAGI = "Selamat Pagi";
3
4 console.log(PI);           // 3.14
5 console.log(SALAM_PAGI);   // Selamat Pagi

```

Disini saya membuat 2 buah konstanta, yakni PI dan SALAM\_PAGI. Keduanya diberikan nilai awal 3.14 dan string "Selamat Pagi".

Aturan nama dari konstanta sama seperti variabel, karena nama konstanta ini juga termasuk ke dalam **identifier**.

Berbeda dengan variabel yang menggunakan **Camel Case**, konstanta biasa ditulis menggunakan huruf besar dan garis bawah (underscore) sebagai pemisah kata. Tujuannya agar mudah dibedakan dengan variabel. Aturan ini hanya kebiasaan programmer JavaScript. Tentu saja anda bisa membuat konstanta dengan huruf kecil:

```

1 const pi = 3.14;
2 const salam_pagi = "Selamat Pagi";
3
4 console.log(pi);           // 3.14
5 console.log(salam_pagi);   // Selamat Pagi

```

Sesuai dengan namanya, nilai dari **konstanta** tidak bisa diubah sepanjang kode program. Jika kita mencoba menimpa nilai konstanta ini, JavaScript akan error:

```

1 const PI = 3.14;
2 PI = 4.14;    // TypeError: invalid assignment to const `PI`
3
4 const SALAM_PAGI;
5 SALAM_PAGI = "Selamat Pagi"; // SyntaxError: missing = in const declaration

```

Contoh pada baris ke 5 mungkin sedikit aneh. Di baris ke 4, konstanta SALAM\_PAGI belum diisi nilai apapun, jadi seharusnya bisa diisi dengan string "Selamat Pagi". Error terjadi karena ketika sebuah konstanta di deklarasikan tanpa nilai awal, JavaScript otomatis memberikan nilai "undefined", yakni tipe data khusus yang berarti tidak terdefenisi.

Artinya, di baris ke 5 saya mencoba mengubah nilai "undefined" ini dengan string "Selamat Pagi". Tentu saja tidak bisa karena nilai konstanta SALAM\_PAGI sudah berisi nilai "undefined". Hal ini juga menunjukkan, untuk membuat konstanta kita harus memberikan nilai awal pada saat pendeklarasian.

Konstanta pas digunakan untuk nilai yang selalu tetap seperti konstanta matematika PI, kecepatan cahaya, tanggal kemerdekaan RI, dst.

Berdasarkan pengalaman saya, konstanta relatif jarang terpakai. Kita lebih banyak menggunakan variabel sepanjang pembuatan kode program.

## 5.8 Identifier dan Literal

Istilah **Identifier** dan **Literal** lebih ke arah teori konsep programming, tapi cukup penting untuk dipahami. Terutama jika anda ingin mengikuti pembahasan tentang JavaScript dari tutorial teknis seperti [developer.mozilla.org](https://developer.mozilla.org)<sup>2</sup> atau forum diskusi [stackoverflow.com](http://stackoverflow.com)<sup>3</sup>.

**Identifier** sudah kita singgung sebelumnya, yakni sebutan untuk nama dari “sesuatu” dalam JavaScript. Contoh identifier adalah *variabel*, *konstanta*, *nama fungsi (function)* dan *nama object*.

**Literal** adalah sebutan untuk nilai yang kita input ke dalam JavaScript. Umumnya literal ini disimpan ke dalam **identifier**, seperti *variabel*.

Perhatikan contoh berikut:

```
1 var hargaBarang = 12000;
2 var namaLengkap = "Rudi Siswoyo";
3 const PI = 3.14;
```

Variabel `hargaBarang`, `namaLengkap` dan konstanta `PI` adalah **identifier**. Sedangkan `12000`, `"Rudi Siswoyo"`, dan `3.14` adalah **literal**. Jika data itu berupa angka, dikenal dengan **numeric literal**, seperti `3.14`, `2000`, `0.001`. Sedangkan jika berupa teks (*string*) disebut sebagai **string literal**, seperti `"Rudi Siswoyo"`, `"DuniaIlkom"`, `"9 Naga"`.

Istilah **identifier** dan **literal** ini akan saya gunakan dalam beberapa pembasan nantinya.

---

Dalam bab ini kita telah membahas mengenai **variabel** dan **konstanta** JavaScript. Termasuk di dalamnya cara penulisan variabel dengan keyword `var` dan `let`, serta membahas pengertian **identifier** dan **literal**.

Berikutnya, kita akan mempelajari apa saja yang bisa diinput ke dalam variabel dan konstanta ini, yakni tentang Tipe Data di dalam JavaScript.

---

<sup>2</sup><https://developer.mozilla.org>

<sup>3</sup><http://stackoverflow.com>

# 6. Tipe Data JavaScript

Variabel dan konstanta yang kita pelajari dalam bab sebelum ini digunakan untuk menampung data. Data itu sendiri terdiri dari beragam jenis, mulai dari angka, teks, hingga data yang kompleks seperti *array* dan *object*.

Agar memudahkan pemrosesan, JavaScript membedakan data menurut sifatnya atau dikenal sebagai tipe data. Secara garis besar, tipe data dalam JavaScript terdiri dari 2 kelompok, yakni tipe data primitif (primitive type), dan tipe data object.

## 6.1 Perbedaan Tipe Data Primitive dan Object

Tipe data primitif disebut demikian karena tipe data ini “sederhana” dan hanya terdiri dari 1 nilai. Di dalam JavaScript terdapat 6 tipe data primitif:

1. Number
2. String
3. Boolean
4. Null
5. Undefined
6. Symbol\*

Selain dari tipe data ini, adalah *object*. **Object** bisa disebut sebagai tipe data “khusus” yang prilaku dan isinya bermacam-macam.

Adapun tipe data object bawaan JavaScript adalah:

1. Array
2. Date
3. RegExp
4. Map dan WeakMap\*
5. Set dan WeakSet\*

Untuk tipe data **Object**, dalam bab ini saya hanya membahas **object array**. Tipe data **Object**, **Date** dan **RegExp** akan dibahas dalam bab tersendiri karena butuh penjelasan yang cukup panjang, termasuk cara membuat object bentukan sendiri.



\* Tipe data **symbol**, **map**, **weakmap**, **set** dan **weakset** adalah tipe data baru dalam **ECMAScript 6**. Tipe data ini tidak akan saya bahas karena termasuk materi lanjutan yang cukup kompleks untuk pemula.

## 6.2 Tipe Data Number

Tipe data **number** adalah tipe data yang berisikan angka, baik itu angka bulat seperti 1, 5, 1000, -99, maupun angka pecahan seperti 1.55, 3.14, 0.0009.

JavaScript merupakan bahasa pemrograman yang cukup unik, karena tidak membedakan tipe data angka bulat dengan angka pecahan. Di dalam JavaScript, angka bulat dan pecahan digabung ke dalam **tipe data number**.



Umumnya dalam bahasa pemrograman lain kita akan memakai tipe data **integer** untuk angka bulat, dan tipe data **float** atau **real** untuk angka pecahan.

Berikut contoh cara mengisi variabel dengan tipe data number di dalam JavaScript:

```
1 var foo = 100;
2 var bar = -5000;
3 var baz = 0.66634;
4
5 console.log(foo); // 100
6 console.log(bar); // -5000
7 console.log(baz); // 0.66634
```

Jika anda baru belajar bahasa pemrograman, untuk membuat nilai pecahan digunakan tanda titik sebagai pemisah bilangan desimal, bukan koma seperti yang biasa kita gunakan sehari-hari.

## Penulisan Scientific Notation

Selain angka “normal”, JavaScript juga mendukung penulisan **scientific notation**, yakni penulisan seperti 3e3, atau 0.4e-3. Huruf e diantara angka ini menandakan pangkat sepuluh. Sebagai contoh 3e<sup>3</sup> artinya  $3 \times 10^3 = 3000$ , sedangkan 0.4e<sup>-3</sup> artinya  $0.4 \times 10^{-3} = 0.0004$ .

Berikut contoh penggunaannya:

```
1 var foo = 3e3;
2 var bar = 0.4e-3;
3 var baz = -9.353e6;
4
5 console.log(foo); // 3000
6 console.log(bar); // 0.0004
7 console.log(baz); // -9353000
```

## Bilangan Desimal, Biner, Oktal, dan Heksadesimal

Angka yang kita gunakan sehari-hari menggunakan basis 10, atau dalam matematika dikenal sebagai bilangan **desimal**. Bilangan desimal adalah angka yang disusun dari 10 digit karakter, yakni angka 0, 1, 2, 3 sampai 9. Selain bilangan desimal, dikenal juga bilangan **biner** (basis 2), **oktal** (basis 8), dan **heksadesimal** (basis 16).

Bilangan **biner** adalah angka yang digitnya hanya 2 buah, yakni 0 dan 1. Sebagai contoh, angka 18 desimal jika dikonversi ke dalam bentuk angka biner menjadi 10010.

Bilangan **oktal** adalah angka yang digitnya hanya 8 buah, yakni 0, 1, 2, .. 7. Sedangkan bilangan **heksadesimal** adalah angka yang digitnya terdiri 16 digit, yakni angka 0 sampai 9, kemudian ditambah dengan huruf A, B, C, D, E, dan F.

JavaScript mendukung penulisan bilangan *biner*, *oktal* dan *heksadesimal*. Berikut contoh penggunaannya:

```
1 var foo;  
2  
3 foo = 999;           // angka desimal  
4 console.log(foo);   // 999  
5  
6 foo = 0b111100111; // angka biner  
7 console.log(foo);   // 999  
8  
9 foo = 01747;        // angka oktal  
10 console.log(foo);  // 999  
11  
12 foo = 0o1747;      // angka oktal  
13 console.log(foo);  // 999  
14  
15 foo = 0x3E7;       // angka heksadesimal  
16 console.log(foo);  // 999
```

Untuk membuat bilangan **biner**, diawali dengan angka **0**, kemudian diikuti dengan huruf **b**, contohnya 0b111100111, yang artinya saya ingin membuat angka biner 1111100111.

Untuk membuat bilangan **oktal**, diawali dengan angka **0**, atau **0o** (angka nol dan dengan huruf ‘o’ kecil) seperti 01747 atau 0o1747. Ini artinya saya ingin membuat bilangan oktal dengan angka 1747.

Untuk membuat bilangan **heksadesimal**, diawali dengan angka **0** dan huruf ‘x’, seperti 0x3E7, yang artinya angka 3E7 dalam format *heksadesimal*.

Sebagaimana yang terlihat, walaupun saya mengisi variabel foo dengan bilangan *biner*, *oktal* dan *heksadesimal*, ketika di tampilkan angka-angka ini dikonversi otomatis oleh JavaScript ke bentuk desimal. Dengan kata lain, ketiga sistem bilangan ini hanya diproses secara internal oleh JavaScript.

Dalam contoh diatas juga terlihat bahwa angka **999 desimal = 1111100111 biner = 1747 oktal = 3E7 heksadesimal**.



Teori dasar seputar bilangan **biner**, **oktal** dan **heksadesimal** tidak akan saya bahas karena terlalu teknis. Selain itu, kita jarang menggunakan ketiga bilangan ini di dalam web programming.

## Nan dan Infinity

Selain “angka”, JavaScript memiliki 2 nilai yang termasuk tipe data number, yakni **NaN** dan **Infinity**. Kedua nilai ini hadir untuk menampung hasil matematika yang “tidak umum”, seperti contoh berikut:

```
1 var foo;  
2  
3 foo = 9 * "a";  
4 console.log(foo); // NaN  
5  
6 foo = 9 / "a";  
7 console.log(foo); // NaN  
8  
9 foo = 5 / 0;  
10 console.log(foo); // Infinity  
11  
12 foo = -5 / 0;  
13 console.log(foo); // -Infinity
```

Disini saya menggunakan operator matematika seperti perkalian dan pembagian untuk mendapatkan hasil **NaN** dan **Infinity**.

**NaN** adalah singkatan dari **Not a Number** yang berarti “bukan sebuah angka”. Dalam contoh diatas, saya mencoba mengalikan angka 9 dengan huruf “a”, atau membagi angka 9 dengan “a”. Tentu saja operasi seperti ini tidak bisa dilakukan. JavaScript mengatasinya dengan menghasilkan nilai **NaN**.

Berikut beberapa operasi lain yang juga akan menghasilkan nilai **NaN**:

- Pembagian 0 dengan 0.
- Pembagian *infinity* dengan *infinity*.
- Akar kuadrat dari nilai negatif.
- Operasi aritmatika dengan nilai yang bukan angka (dan tidak bisa dikonversi menjadi angka).

**Infinity** atau dalam istilah matematika indonesia dikenal dengan “tak hingga”, didapat dari operasi matematika yang tidak umum, misalnya membagi sebuah angka dengan nol.

Di dalam JavaScript **Infinity** terdiri dari dua jenis: **positif infinity**, dan **negatif infinity**. Pada contoh kode program sebelumnya saya mendapatkan nilai **infinity** karena mencoba membagi angka 5 dengan 0. Jika menggunakan -5, hasilnya adalah **-infinity**.

Berikut beberapa operasi yang juga akan menghasilkan nilai infinity :

- Jika sebuah angka melewati nilai maksimum yang bisa ditampung di dalam JavaScript (disebut juga dengan istilah: *overflow*).
- Jika sebuah angka lebih besar dari angka negatif yang bisa ditampung (*negative overflow*).
- Melakukan operasi aritmatika dengan infinity (misal:  $a = 1 + \text{infinity}$ ).
- Melakukan operasi pembagian dengan nilai 0 (*division by zero*).

Secara umum, jika anda mendapatkan hasil **NaN** atau **Infinity**, besar kemungkinan ada sesuatu yang salah dalam kode program yang dibuat, kecuali jika anda memang membuat program perhitungan matematis rumit yang bisa menghasilkan nilai-nilai ini.

## Kesalahan Pembulatan Tipe Data Number

Salah satu konsep dasar yang harus selalu diingat dalam programming adalah keterbatasan komputer dalam menyimpan dan melakukan perhitungan angka. Untuk angka bulat, tidak ada masalah. Tapi untuk nilai pecahan, terdapat kemungkinan perhitungan yang salah karena keterbatasan memory komputer.

Sebagai contoh, silahkan anda tebak hasil dari kode program berikut:

```
1 var foo;  
2  
3 foo = 0.1 + 0.2;  
4 console.log(foo);
```

Hasilnya 0,3 bukan? Tidak untuk JavaScript, hasilnya adalah: **0.3000000000000004**.

Kesalahan seperti ini bukan bug atau error dari JavaScript, tapi karena keterbatasan perhitungan komputer secara umum. JavaScript menggunakan format angka **IEEE-764 double-precision floating-point**. Konsekuensi dari hal ini, bisa terjadi kesalahan pembulatan seperti diatas. Dalam bahasa pemrograman lain, mungkin juga akan didapat hasil yang sama.

Jadi bagaimana untuk menghindari hal ini? Kita bisa menggunakan fungsi pembulatan seperti **Math.round()** yang akan saya bahas nantinya. Atau bisa juga mengkonversi angka pecahan ini menjadi angka bulat sebagai berikut:

```
1 var foo;  
2  
3 foo = (0.1 * 10) + (0.2 * 10);  
4 console.log(foo / 10); // 0.3
```

Disini saya mengalikan angka `0.1` dan `0.2` dengan angka `10`. Setelah proses penambahan selesai, hasilnya dibagi kembali dengan `10`.

Dalam penggunaan sehari-hari, kita tidak perlu khawatir dengan kesalahan pembulatan seperti ini. Namun jika anda membuat aplikasi yang melibatkan banyak perhitungan dalam bentuk pecahan, silahkan periksa setiap kemungkinan yang bisa terjadi akibat *rounding error* (kesalahan pembulatan).

## 6.3 Tipe Data String

Tipe data **string** adalah sebutan untuk data yang berisi teks, seperti `"hello"`, `"nama kamu"`, `"andi"` atau `"i"`. JavaScript mendukung pembuatan string menggunakan **tanda kutip satu** (`'`) maupun **tanda kutip dua** (`"`). Berikut contohnya:

```
1 var foo;  
2  
3 foo = "Hello World";  
4 console.log(foo); // Hello World  
5  
6 foo = 'Sedang belajar JavaScript';  
7 console.log(foo); // Sedang belajar JavaScript  
8  
9 foo = "199";  
10 console.log(foo); // 199 (string)
```

Contoh terakhir cukup menarik. Dengan menambahkan tanda kutip, variabel `foo` berisi **string** `199`, bukan tipe data **number** `199`. Jika yang dimaksud adalah angka `199`, hapus tanda kutip dari string tersebut.

Kita juga bisa mencampur penggunaan tanda kutip untuk membuat **string**, seperti contoh berikut:

```
1 var foo;  
2  
3 foo = "Hari Jum'at";  
4 console.log(foo);  
5 // Hari Jum'at  
6  
7 foo = 'Dia berkata: "aku sedang belajar JavaScript"';  
8 console.log(foo);  
9 // Dia berkata: "aku sedang belajar JavaScript"  
10  
11 foo = "Let's go!";  
12 console.log(foo);  
13 // Let's go!
```

Syarat dari penulisan seperti ini adalah, tanda kutip yang digunakan sebagai penanda string tidak boleh muncul di dalam string. Jika saya menggunakan tanda kutip dua untuk membuat string, maka yang boleh muncul hanya tanda kutip satu. Begitu juga sebaliknya.

Saya tidak bisa membuat string berikut:

```
1 var foo;  
2  
3 foo = 'Hari Jum'at';  
4 console.log(foo);  
5 // error !  
6  
7 foo = "Dia berkata: "aku sedang belajar JavaScript"";  
8 console.log(foo);  
9 // error !
```

Jika ditulis seperti ini, JavaScript “bingung” untuk menentukan awal dan akhir dari string. Tapi bagaimana jika saya tetap ingin membuat tanda kutip tersebut di dalam string? kita bisa menggunakan **escape character**.

## Escape Character

**Escape character** adalah cara memasukkan karakter khusus ke dalam string. Untuk membuat *escape character*, digunakan tanda **backslash** yakni karakter (\).

Sebagai contoh, jika saya ingin menampilkan tanda kutip ke dalam string yang juga dibuat dengan tanda kutip tersebut, caranya adalah sebagai berikut:

```
1 var foo;  
2  
3 foo = 'Hari Jum\'at';  
4 console.log(foo);  
5 // Hari Jum'at  
6  
7 foo = "Dia berkata: \"aku sedang belajar JavaScript\"";  
8 console.log(foo);  
9 // Dia berkata: "aku sedang belajar JavaScript"
```

Tanda \' artinya saya menginformasikan kepada JavaScript bahwa tanda kutip tersebut bukan sebagai penanda akhir string, tapi hanya sebagai karakter “biasa”.

Selain men-*escape* tanda kutip, *backslash* juga digunakan untuk menulis karakter khusus yang tidak bisa diketik, seperti karakter **enter** dan **tab**.

Berikut karakter escape lain yang bisa digunakan di dalam JavaScript:

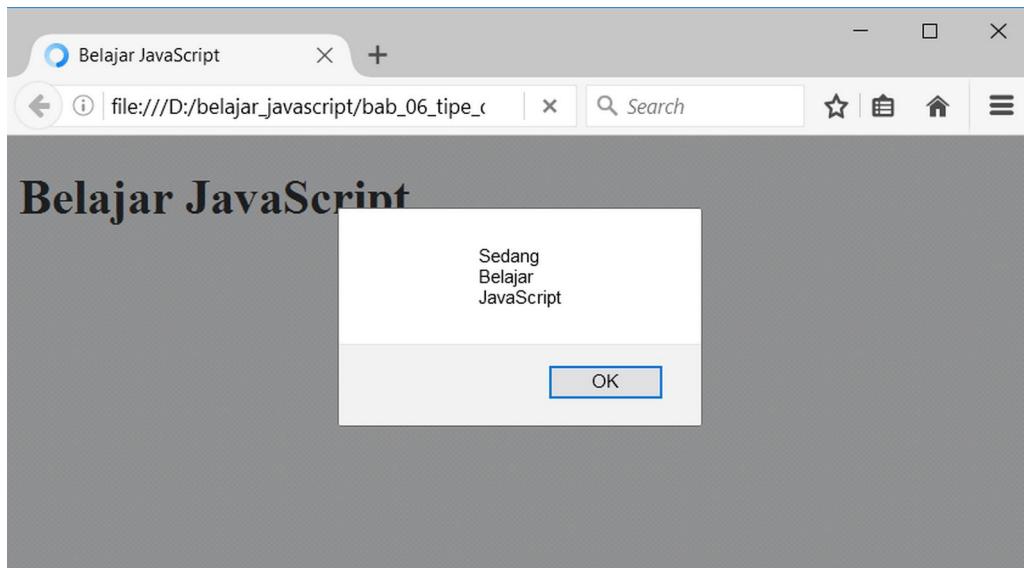
- \0: Karakter NUL
- \b: Backspace
- \t: Horizontal tab
- \n: Newline
- \v: Vertical tab
- \f: Form feed
- \r: Carriage return
- \": Tanda kutip dua (double quote)
- \': Tanda kutip satu (apostrophe atau single quote)
- \\: Garis miring backslash
- \xXX: Karakter Latin-1 dengan menggunakan dua digit heksa desimal XX
- \uXXXX: Karakter Unicode dengan menggunakan empat digit heksa XXXX

Selain menulis tanda kutip, *escape character* yang cukup sering digunakan adalah \n, fungsinya untuk membuat karakter ‘enter’ atau pindah baris.

Sebagai contoh, saya ingin menampilkan pesan `alert()` yang terdiri dari 3 baris. Berikut cara penulisannya:

```
1 var foo = "Sedang\nBelajar\nJavaScript";  
2 alert(foo);
```

Saya mengganti tempat spasi dengan karakter espace \n. Ketika kode ini dijalankan, hasilnya akan menampilkan jendela alert dengan teks "Sedang Belajar JavaScript" dipecah ke dalam 3 baris. Tanpa karakter \n, kita tidak akan bisa membuat tampilan seperti ini.



Gambar: Membuat alert() dengan 3 baris teks

Jika ingin menulis karakter backslash di dalam teks, kita juga harus menulisnya sebanyak 2 kali:

```
1 var foo = "http:\\\\www.duniaIlkom.com";
2 console.log(foo); // http:\\www.duniaIlkom.com
```

Ini diperlukan karena backslash merupakan karakter khusus di dalam JavaScript (yang digunakan sebagai *escape character*), jadi kita harus menulisnya 2 kali.

Bagaimana jika saya menulis *escape character* untuk huruf yang tidak spesial? JavaScript akan mengabaikannya, seperti contoh berikut:

```
1 var foo = "\H\e\1\1\o";
2 console.log(foo); // Hello
```

Karena H tidak bermakna apa-apa, JavaScript menampilkan huruf H biasa.

## Menulis Karakter Unicode

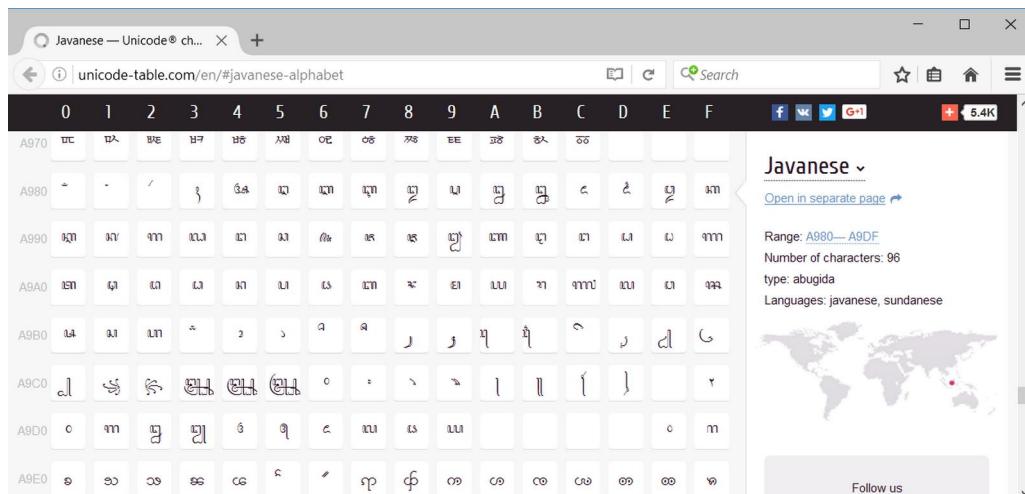
Dari daftar *escape character*, dua poin terakhir, yakni \xxx dan \uXXXX digunakan untuk menulis karakter **unicode** ke dalam string. Apa maksud karakter unicode ini?

**Unicode** adalah sistem pengkodean (*character set*) yang digunakan untuk menampilkan sebuah karakter ke dalam komputer. Contoh character set yang lebih umum adalah **ASCII** (*American Standard Code for Information Interchange*).

Di dalam ASCII, huruf “A” disimpan sebagai bit 1000001 atau 41 dalam bilangan **heksadesimal**. Sistem character set ASCII mendukung 256 karakter, atau 8 bit. Karakter sebanyak 256 ini sudah lebih dari cukup untuk menampung abjad latin dan angka. Tapi tidak untuk aksara lain seperti huruf jepang, arab, rusia, dsb.

Agar bisa mengakomodir karakter lain diluar huruf ASCII, dikembangkanlah **Unicode**. Secara teori **Unicode** sanggup menampung 1 juta lebih karakter, walaupun yang didaftarkan baru sekitar 128.237<sup>1</sup>.

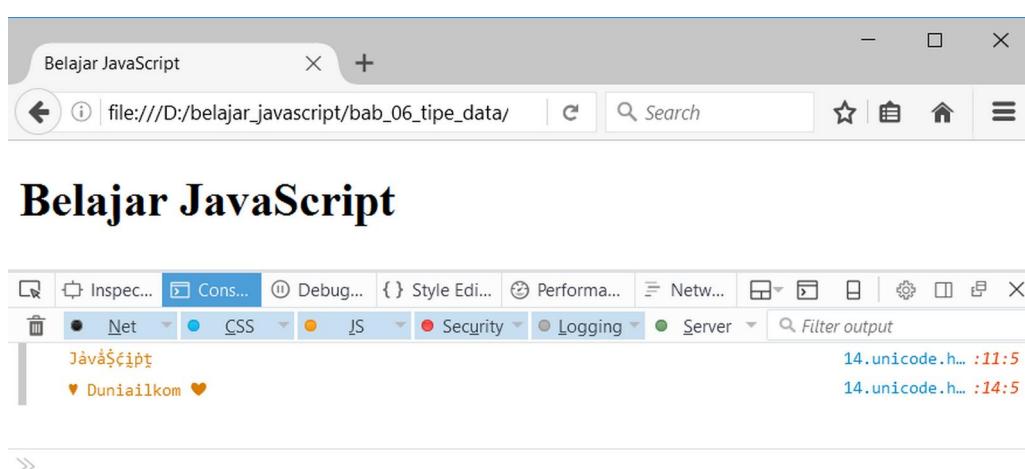
Daftar lengkap dari karakter **Unicode** ini bisa dilihat di [unicode-table.com](http://unicode-table.com)<sup>2</sup>. Karakter yang bisa ditulis sangat beragam, mulai dari huruf latin, jepang, korea, arab, hingga huruf jawa kuno dan emoji (icon).



Gambar: Karakter huruf jawa yang bisa ditulis menggunakan **unicode** ([unicode-table.com](http://unicode-table.com))

Berikut contoh penulisan karakter **Unicode** di dalam JavaScript:

```
1 var foo = "J\u1EA3v\u1EB3\u1E68\u1E09\u1E2D\u1E57\u1E71";
2 console.log(foo);
3
4 var bar = "\u2665 DuniaIlkom \u2764";
5 console.log(bar);
```



Gambar: Cara penulisan karakter **unicode** di dalam JavaScript

<sup>1</sup><https://en.wikipedia.org/wiki/Unicode>

<sup>2</sup><http://unicode-table.com>

Teks \u2665 artinya tampilkan karakter **Unicode** dengan nomor urut 2665. Teks \u1E71 artinya tampilkan karakter Unicode dengan nomor urut 1E71. Nomor urut karakter ini bisa anda lihat di [unicode-table.com](http://unicode-table.com).

## Template String

**Template String** (atau dikenal juga dengan sebutan *string interpolation*) merupakan fitur baru di ECMAScript 6. Dengan **template string**, kita bisa menampilkan nilai variabel saat berada di dalam string. Fitur ini mirip seperti penggunaan tanda kutip dua di bahasa pemrograman PHP.

Sebagai contoh, di PHP kita bisa melakukan hal berikut:

```
1 $foo = "Indonesia";
2 $bar = "Selamat Pagi $foo";
3 echo $bar; // Selamat Pagi Indonesia
```

Didalam string untuk variabel \$bar, saya menulis "Selamat Pagi \$foo". Oleh PHP, tanda *dollar* akan diproses untuk menampilkan nilai yang ada di dalam variabel \$foo. Disini \$foo merupakan perintah khusus (bukan bagian dari string). Cara penulisan ini hanya bisa dilakukan jika string dibuat menggunakan tanda kutip dua.

Dengan menggunakan fitur *template string*, saya bisa melakukan hal yang sama di JavaScript. Berikut contoh penggunaannya:

```
1 var foo = "Indonesia";
2 var bar = `Selamat Pagi ${foo}`;
3 console.log(bar); // Selamat Pagi Indonesia
```

Terlihat saya menggunakan karakter **backtick** untuk membuat string yang akan diinput ke dalam variabel bar1. Karakter backtick ini cukup jarang dipakai dan saya tidak akan heran jika anda susah menemukannya di keyboard. Karakter ini berada di sebelah kiri angka 1 (diatas tombol tab).

Agar variabel foo bisa diproses, string ini ditulis menggunakan **tanda dollar** dan **harus berada di dalam kurung kurawal**, seperti \${foo}. Inilah yang disebut fitur *template string* dari JavaScript.

Sebelum ada fitur *template string*, kita harus menggunakan operator ‘ + ’ untuk mendapatkan hasil yang sama:

```
1 var foo = "Indonesia";
2 var bar = "Selamat Pagi " + foo;
3 console.log(bar); // Selamat Pagi Indonesia
```

Pembahasan tentang operator ‘ + ’ ini akan saya bahas dalam bab selanjutnya.

## 6.4 Tipe Data Boolean

Tipe data **Boolean** adalah tipe data yang hanya mempunyai dua nilai, yakni **true** (benar) atau **false** (salah). Tipe data boolean sering digunakan untuk membuat alur logika program. Struktur logika seperti **if**, **else**, **while**, dan **do while**, membutuhkan nilai boolean sebagai ‘pengontrol’ alur program. Struktur logika ini akan saya bahas dalam bab terpisah.

Berikut contoh pembuatan tipe data **boolean** di dalam JavaScript:

```
1 var foo = true;
2 var bar = false;
3
4 console.log(foo); // true
5 console.log(bar); // false
```

Sama seperti penulisan *keyword* lain di dalam JavaScript, nilai **boolean** ini harus ditulis dalam huruf kecil, tidak boleh dengan huruf besar. Kode berikut akan menghasilkan error:

```
1 var foo = TRUE;
2 var bar = False;
3
4 console.log(foo); // ReferenceError: TRUE is not defined
5 console.log(bar); // ReferenceError: False is not defined
```

Tipe data boolean juga bisa didapat dari hasil operasi perbandingan. Misalkan apakah variabel a sama dengan b, atau apakah a lebih besar dari b. Ini juga akan kita pelajari nanti.

## 6.5 Tipe Data Null dan Undefined

Menyebut **null** dan **undefined** sebagai tipe data terasa kurang pas, karena kedua tipe data ini menyatakan “data yang tidak ada”.

Dalam teorinya, tipe data **null** hanya berisi 1 nilai, yakni **null**. Begitu juga dengan tipe data **undefined** yang berisi 1 nilai, yakni **undefined**. Keduanya mirip, tapi dengan beberapa perbedaan.

**Null** adalah keadaan dimana data itu “kosong”. Umumnya nilai **null** diinput dengan sengaja oleh kita, programmer yang membuat kode program. Berikut contoh cara memberikan tipe data **null** kedalam sebuah variabel:

```
1 var foo = null;
2 console.log(foo); // null
```

Bagi programmer pemula, mungkin ada akan bertanya: “*Untuk apa membuat data yang isinya tidak ada?*”.

Ada beberapa kasus dimana kita hanya bisa menggunakan nilai **null**. Misalnya untuk *function* yang butuh input 2 buah variabel, tapi kita hanya punya 1 variabel. Variabel kedua ini bisa diinput sebagai **null** (tidak semua function bisa diinput dengan nilai **null**).

Disisi lain, **undefined** menyatakan data yang tidak terdefinisi. Tapi berbeda dengan **null**, biasanya nilai **undefined** dihasilkan dari JavaScript itu sendiri, dan sebagian besar karena kesalahan program.

Contohnya, ketika kita mendefenisikan sebuah variabel tapi tidak memberikan nilai apapun. Variabel ini akan berisi data **undefined**:

```
1 var foo;  
2 console.log(foo); // undefined
```

Sama seperti **null**, kita juga bisa menginput manual nilai null ke dalam sebuah variabel:

```
1 var foo = "Belajar JavaScript";  
2 console.log(foo); // Belajar JavaScript  
3  
4 foo = undefined;  
5 console.log(foo); // undefined
```

Dalam kode program diatas, menginput nilai **undefined** ke dalam variabel yang sudah ada, dan ini bisa disamakan dengan menghapus variabel tersebut.

## 6.6 Operator **typeof**

Pembahasan tentang operator akan dibahas dalam bab berikutnya, tapi karena operator **typeof** sangat erat dengan pembahasan tentang tipe data, saya akan bahas disini.

Operator **typeof** digunakan untuk melihat tipe data dari sebuah variabel. Apakah tipe datanya **number**, **string**, **boolean**, **undefined**, atau sebuah **object**.

Berikut contoh penggunaan operator **typeof**:

```
1 var foo;
2 console.log (typeof foo); // undefined
3
4 foo = 199;
5 console.log (typeof foo); // number
6
7 foo = 0.0003;
8 console.log (typeof foo); // number
9
10 foo = "Belajar JavaScript";
11 console.log (typeof foo); // string
12
13 foo = `Hello Indonesia`;
14 console.log (typeof foo); // string
15
16 foo = false;
17 console.log (typeof foo); // boolean
18
19 foo = null;
20 console.log (typeof foo); // object
```

Sepanjang contoh kode program diatas, saya mengubah nilai variabel `foo` dengan berbagai data, mulai dari angka bulat, angka desimal, string, boolean, hingga null. Yang agak aneh, anda bisa melihat nilai null dianggap sebagai *object*. Ini memang sebuah pengecualian di dalam JavaScript.

Operator `typeof` sering digunakan untuk proses *debugging* (pencarian kesalahan). Jika kode program yang anda tulis tidak berjalan sebagaimana mestinya, bisa di cek apakah tipe data yang digunakan sudah benar atau salah. Misalkan apakah yang seharusnya ditulis adalah angka 5, atau string “5”.

## 6.7 Tipe Data Array

**Array** (dalam bahasa indonesia disebut juga sebagai **lariik**) adalah tipe data yang berisi kumpulan tipe data lain. Jika anda ingin membuat banyak data yang masuk dalam satu kelompok (seperti kumpulan nama siswa) akan lebih praktis menggunakan array dari pada membuat variabel tersebut satu per satu.

Misalkan saya ingin menginput 5 data nama siswa untuk diproses. Saya bisa menulis sebagai berikut:

```

1 var siswa0 = "Andri";
2 var siswa1 = "Joko";
3 var siswa2 = "Sukma";
4 var siswa3 = "Rina";
5 var siswa4 = "Sari";
6
7 console.log(siswa0); // Andri
8 console.log(siswa1); // Joko
9 console.log(siswa2); // Sukma
10 console.log(siswa3); // Rina
11 console.log(siswa4); // Sari

```

Kode program diatas tidak salah, namun kurang praktis. Jika saya memiliki 1000 siswa, saya harus membuat 1000 variabel.

Dengan menggunakan array, penulisannya jauh lebih sederhana:

```

1 var siswa = ["Andri", "Joko", "Sukma", "Rina", "Sari"];
2
3 console.log(siswa[0]); // Andri
4 console.log(siswa[1]); // Joko
5 console.log(siswa[2]); // Sukma
6 console.log(siswa[3]); // Rina
7 console.log(siswa[4]); // Sari

```

Sekarang, variabel `siswa` adalah **array** yang berisi nama-nama siswa.

Di dalam JavaScript, untuk membuat array kita cukup menggunakan tanda kurung siku ( [ ) dan ( ] ), yang didalamnya ditulis anggota dari array tersebut (dipisah dengan tanda koma). Anggota array ini dikenal juga dengan sebutan **element array**.

## Mengakses Element Array

Untuk mengakses element array, kita menuliskan **index** atau nomor urut dari element tersebut. Penomoran index array dimulai dari 0, bukan 1. Jika ingin mengakses element ke 2, kita menggunakan nomor index 1. Jika ingin mengakses element pertama, menggunakan index 0. Nomor index ini ditulis di dalam kurung siku.

Sebagai contoh, saya ingin menampilkan siswa “Rina” yang berada di index ke 3, bisa menggunakan perintah berikut:

```
console.log(siswa[3]);
```

Anggota dari array juga bisa bercampur dengan tipe data lain, tidak harus 1 tipe data saja:

```
1 var foo = [1, 2, 3, 4, 5];
2 console.log(foo);
3 // Array [ 1, 2, 3, 4, 5 ]
4
5 var bar = ["a", "b", "c"];
6 console.log(bar);
7 // Array [ "a", "b", "c" ]
8
9 var baz = ["aku", 999, true, 0.085, null];
10 console.log(baz);
11 // Array [ "aku", 999, true, 0.085, null ]
12
13 console.log(typeof baz); // object
```

Disini saya membuat 3 buah **array** dengan berbagai element, yakni variabel `foo`, `bar`, dan `baz`. Terlihat juga bahwa fungsi `console.log()` bisa digunakan untuk menampilkan seluruh element array dalam 1 kali pemanggilan.

Perintah terakhir cukup unik, dimana saya mencoba mengecek tipe data dari array `baz`. Hasilnya adalah **object**. Yup, di dalam JavaScript, array termasuk ke dalam kelompok objek yang memiliki fitur khusus.

## Mengubah Element Array

Jika kita mengakses element array dengan menulis nomor indexnya, cara yang sama juga bisa digunakan untuk mengubah isi dari element array tersebut.

Perhatikan contoh berikut:

```
1 var foo = ["andi", "santi", "joko"];
2
3 foo[0] = "alex";
4 console.log(foo); // Array [ "alex", "santi", "joko" ]
5
6 foo[3] = "rika";
7 console.log(foo); // Array [ "alex", "santi", "joko", "rika" ]
```

Diawal kode program, saya membuat array `foo` yang berisi 3 nama siswa. Di baris berikutnya, terdapat perintah `foo[0] = "alex"`. Ini artinya saya ingin mengganti isi dari element array `foo` di posisi index 0 dengan string baru: `"alex"`. Hasilnya, element pertama dari array `foo` sudah berganti dari `"andi"` menjadi `"alex"`.

Berikutnya, terdapat perintah `foo[3] = "rika"`. Artinya, saya ingin menginput string `"rika"` ke dalam array `foo` di nomor index 3. Akan tetapi, array `foo` hanya terdiri dari 3 element saja (dimana element terakhir memiliki index 2). Untuk situasi seperti ini, perintah tersebut akan menambahkan 1 element baru dalam variabel `foo` (dengan nomor index 3).

## Array 2 Dimensi

Sebuah array tidak hanya bisa diisi dengan tipe primitif saja (*number*, *string*, dan *boolean*), tapi juga bisa diisi dengan tipe data yang lebih kompleks, seperti *object* maupun *array* lain.

Jika element dari array berupa array lain, ini dikenal juga dengan sebutan **array 2 dimensi**. Berikut contoh penggunaannya:

```
1 var koordinat = [[2,5],[9,5],[3,5]];
2
3 console.log(koordinat[0][0]); // 2
4 console.log(koordinat[0][1]); // 5
5
6 console.log(koordinat[1][0]); // 9
7 console.log(koordinat[1][1]); // 5
8
9 console.log(koordinat[2][0]); // 3
10 console.log(koordinat[2][1]); // 5
```

Kali ini saya memiliki variabel *koordinat* yang merupakan array 2 dimensi. Disebut seperti itu karena element dari array *koordinat* adalah array lain.

Array *koordinat* memiliki 3 element, dimana ketiga element ini masing-masingnya array dengan 2 element.

Untuk mengakses array 2D, kita menulis 2 buah index, seperti: *koordinat[1][0]*. Index pertama digunakan untuk mencari posisi element array pertama, dan index kedua untuk mencari posisi array di dalam array (array terdalam).

Agar lebih memahami array 2 dimensi ini, silahkan anda pelajari kode program berikut:

```
1 var foo = [
2     ["sedang", "belajar", "javascript"],
3     ["selamat", "pagi", "dunia"],
4     ["a", "b", "c", "d", "e"]
5 ];
6
7 console.log(foo[1][2]); // "dunia"
8 console.log(foo[2][2]); // "c"
9
10 foo[1][1] = "malam";
11 console.log(foo[1]); // Array [ "selamat", "malam", "dunia" ]
12
13 foo[2][5] = "f";
14 foo[2][6] = "g";
15 console.log(foo[2]); // Array [ "a", "b", "c", "d", "e", "f", "g" ]
```

Pertanyaan pertama. *Berapakah banyak element dari array foo?*

Array `foo` hanya terdiri dari 3 element. Namun masing-masing element ini adalah array, yang juga berisi element lain.

Untuk mengakses string "dunia", saya menggunakan perintah `foo[1][2]`. String "dunia" berada di element kedua (index ke-1) dari array `foo`, dan di element ini, string "dunia" ada di urutan ketiga (index ke-2).

Cara akses ini juga bisa digunakan untuk mengubah atau menambah element baru ke dalam array. Perintah `foo[2][5] = "f"` artinya saya ingin mengganti element ke-5 untuk array kedua dari `foo` dengan string "f". Jika posisi ini tidak ditemukan, string "f" akan ditambahkan ke dalam array.

Tipe data array merupakan salah satu tipe data terpenting di dalam JavaScript. Ketika kita masuk ke pembahasan tentang **DOM**, mayoritas element di dalam web browser diakses menggunakan array.

---

Dalam bab ini kita telah membahas tentang tipe-tipe data di dalam JavaScript, terutama tipe data primitif: **number**, **string**, **boolean**, **null**, dan **undefined**. Serta satu tipe data object, yakni **array**. Tipe data object lain akan dipelajari dalam bab tersendiri.

Berikutnya, kita akan masuk ke bahasan tentang operator di dalam bahasa pemrograman JavaScript.

# 7. Operator JavaScript

**Operator** digunakan untuk memproses data yang sudah diinput ke dalam JavaScript. Berbagai tipe data yang kita pelajari dalam bab sebelumnya memiliki operator khusus, sesuai dengan ciri khasnya masing-masing.

Tipe data **number** biasanya digunakan bersamaan dengan **operator aritmatika**. Tipe data **string** biasanya digunakan dengan **operator string concatenation** (penyambungan string), dst.

Dalam bab ini kita akan membahas tentang Operator dalam JavaScript, yakni:

- Operator Aritmatika
- Operator Increment dan Decrement
- Operator Perbandingan
- Operator Logika
- Operator String
- Operator Bitwise
- Operator Assignment
- Operator Spread

## 7.1 Operator Aritmatika

**Operator aritmatika** adalah operator yang biasa kita gunakan untuk melakukan perhitungan dengan angka, yakni penambahan, pengurangan, perkalian, dan pembagian. Selain itu di dalam JavaScript juga terdapat operator *modulus* (sisa hasil bagi).

Tabel berikut merangkum operator aritmatika di dalam JavaScript:

Nama Operator	Operator	Contoh	Hasil
positif / plus	+	+a	nilai positif dari a
negatif / minus	-	-a	nilai negatif dari a
penambahan	+	a + b	total dari a dan b
pengurangan	-	a - b	selisih dari a dan b
perkalian	*	a * b	hasil kali dari a dan b
div/pembagian	/	a / b	hasil bagi dari a dan b
modulus/sisa hasil bagi	%	a % b	sisa pembagian a bagi b

Berikut contoh penggunaan operator aritmatika di dalam JavaScript:

```
1 var foo;  
2  
3 foo = +100;  
4 console.log(foo);      // 100  
5  
6 foo = -22;  
7 console.log(foo);      // -22  
8  
9 foo = 30 + 5;  
10 console.log(foo);     // 35  
11  
12 foo = 3.33 + 9.02;  
13 console.log(foo);     // 12.35  
14  
15 foo = 9 * 7;  
16 console.log(foo);     // 63  
17  
18 foo = 6 + 8 / 2 + 6;  
19 console.log(foo);     // 16  
20  
21 foo = 30 % 7;  
22 console.log(foo);     // 2
```

Saya yakin anda bisa langsung paham cara penggunaan sebagian besar operator ini. Khusus untuk operator modulus ‘%’ mungkin butuh sedikit penjelasan.

Operator **modulus** digunakan untuk mencari sisa hasil bagi. Sebagai contoh,  $10 \% 3 = 1$ , karena  $3 * 3 = 9$  (sisa 1). Contoh lain,  $30 \% 7 = 2$ , karena  $7 * 3 = 28$  (sisa 2).

Meskipun terdengar asing, operator modulus (sering disingkat dengan **mod**) cukup sering digunakan di dalam pemrograman, misalnya untuk membedakan antara angka bulat dengan ganjil. Jika sebuah angka di mod-kan dengan 2 hasilnya 0, maka itu adalah angka bulat. Jika tidak nol, pasti itu angka ganjil.

Urutan ‘kekuatan’ operator juga pengaruh di sini. Operator perkalian dan pembagian lebih kuat daripada operator penambahan dan pengurangan. Untuk menimpa urutan ini, kita bisa menggunakan tanda kurung. Berikut contohnya:

```
1 var foo;  
2  
3 foo = 4 + 6 / 5 - 3 * 2 + 3;  
4 console.log(foo);      // 2.2  
5  
6 foo = (4 + 6) / (5 - 3) * 2 + 3;  
7 console.log(foo);      // 13
```

Angka dan operator yang digunakan dalam kedua contoh diatas sama persis, tapi hasilnya berbeda. Ini terjadi karena di contoh pertama yang dijalankan adalah  $(4 + (6 / 5)) - (3 * 2)$

`+ 3`), dimana operator pembagian dan perkalian akan diproses dulu sebelum penambahan dan pengurangan.

Walaupun tidak harus, penggunaan tanda kurung sangat disarankan untuk menghindari hasil yang salah dan agar kode program lebih mudah dibaca, seperti contoh berikut:

```
1 var foo;  
2  
3 foo = 10 / 5 + 4.5;  
4 console.log(foo);           // 6.5  
5  
6 foo = (10 / 5) + 4.5;  
7 console.log(foo);           // 6.5
```

Disini tanda kurung sebenarnya tidak diperlukan, karena operasi perkalian lebih kuat daripada penambahan. Tetapi kode program kita lebih mudah dibaca dan tidak ambigu jika ditambahkan tanda kurung.

## 7.2 Operator Increment dan Decrement

Dalam beberapa tutorial atau materi di internet, **operator increment** dan **decrement** ini kadang dimasukkan juga ke dalam kelompok operator aritmatika.

**Operator increment** adalah sebutan untuk penulisan seperti `i++`. Perintah ini digunakan untuk menaikkan nilai variabel `i` sebanyak 1 angka. Perintah `i++` adalah penulisan singkat dari `i = i + 1`.

Sedangkan **operator decrement** adalah sebutan untuk penulisan seperti `i--`. Fungsinya untuk menurunkan nilai variabel `i` sebesar 1 angka. Perintah `i--` merupakan penulisan singkat dari `i = i - 1`.

Posisi peletakan tanda `++` dan `--` bisa diawal maupun diakhir variabel, seperti `++i` maupun `i++`. Keduanya berfungsi sama tapi dengan cara eksekusi yang sedikit berbeda. Tabel berikut merangkum perbedaan ini:

Nama	Contoh	Hasil
pre-increment	<code>++a</code>	tambah nilai <code>a</code> dengan 1, lalu kirim nilai <code>a</code>
post-increment	<code>a++</code>	kirim nilai <code>a</code> , lalu tambah nilai <code>a</code> dengan 1
pre-decrement	<code>--a</code>	kurangi nilai <code>a</code> dengan 1, lalu kirim nilai <code>a</code>
post-decrement	<code>a--</code>	kirim nilai <code>a</code> , lalu kurangi nilai <code>a</code> dengan 1

Berikut contoh praktek dari perbedaan **pre** dan **post** increment/decrement:

```
1 var foo;  
2  
3 foo = 7;  
4 console.log(++foo);      // 8  
5 console.log(foo);        // 8  
6  
7 foo = 7;  
8 console.log(foo++);      // 7  
9 console.log(foo);        // 8  
10  
11 foo = 7;  
12 console.log(--foo);      // 6  
13 console.log(foo);        // 6  
14  
15 foo = 7;  
16 console.log(foo--);      // 7  
17 console.log(foo);        // 6
```

**Pre-increment**, yakni `++foo` artinya naikkan nilai variabel `foo` sebanyak 1 angka, kemudian tampilkan angka yang sudah dinaikkan ini. Dengan demikian, perintah `console.log(++foo)` dengan `console.log(foo)` akan menghasilkan nilai yang sama, karena variabel `foo` sama-sama berisi angka 8 (angka setelah di-increment-kan).

**Post-increment** sedikit berbeda. Perintah `foo++` akan menampilkan nilai `foo` saat ini, baru kemudian menaikkan nilainya. Kata kunci disini adalah **tampilkan dulu, baru ditambahkan**. Pada saat perintah `console.log(foo++)` diproses, akan tampil angka 7. Inilah efek dari “tampilkan dulu”. Sesaat setelah perintah ini dijalankan, variabel `foo` sudah naik menjadi 8, yang merupakan hasil dari `console.log(foo)`.

Begitu pula efek yang terjadi dengan **pre-decrement** dan **post-decrement**. Operator *increment* dan *decrement* ini sering digunakan untuk struktur kondisi perulangan.

## 7.3 Operator Perbandingan

Operator **perbandingan** berfungsi untuk memeriksa bagaimana hubungan sebuah nilai dengan nilai lain, apakah sama besar, tidak sama, lebih besar atau lebih kecil.

Sebagai contoh, apakah `7 < 6` ? atau apakah variabel `b >` dari variabel `a`? Hasil akhir dari perbandingan ini berupa tipe data **boolean**: `true` (benar) atau `false` (salah).

Berikut tabel operator perbandingan di dalam JavaScript:

Nama Operator	Operator	Contoh	Hasil
Sama dengan	<code>==</code>	<code>a == b</code>	TRUE jika nilai a sama dengan b
Identik dengan	<code>===</code>	<code>a === b</code>	TRUE jika nilai a sama dengan b, dan memiliki tipe data yang sama
Tidak sama dengan	<code>!=</code>	<code>a != b</code>	TRUE jika nilai a tidak sama dengan b
Tidak identik dengan	<code>!==</code>	<code>a !== b</code>	TRUE jika nilai a tidak sama dengan b, dan memiliki tipe data yang tidak sama
Kurang dari	<code>&lt;</code>	<code>a &lt; b</code>	TRUE jika nilai a kurang dari b
Lebih dari	<code>&gt;</code>	<code>a &gt; b</code>	TRUE jika nilai a lebih dari b
Kurang dari atau sama dengan	<code>&lt;=</code>	<code>a &lt;= b</code>	TRUE jika nilai a kurang dari atau sama dengan b
Lebih dari atau sama dengan	<code>&gt;=</code>	<code>a &gt;= b</code>	TRUE jika nilai a lebih dari atau sama dengan b

Berikut prakteknya dalam kode program JavaScript:

```

1 var foo;
2
3 foo = 9 < 12;
4 console.log(foo);      // true
5
6 foo = 5 < 5;
7 console.log(foo);      // false
8
9 foo = 5 <= 5;
10 console.log(foo);     // true
11
12 foo = 8 != 12;
13 console.log(foo);     // true
14
15 foo = 15 == 12;
16 console.log(foo);     // false
17
18 foo = 15 === 15;
19 console.log(foo);     // true
20
21 foo = 0.3 === 3e-1;
22 console.log(foo);     // true

```

Sebagian besar hasil dari kode diatas sudah bisa kita pahami, misalnya hasil dari `9 < 12` adalah `true`, karena angka 9 tentu lebih kecil dari 12. Hasil dari `5 <= 5` juga true, karena 5 adalah lebih kecil atau sama dengan 5.

Untuk menyatakan tidak sama, JavaScript menyediakan operator `!=`, dimana `8 != 12` adalah `true`, dan `8 != 8` hasilnya `false`.

Operator perbandingan juga bisa digunakan untuk tipe data string, seperti contoh berikut:

```
1 var foo;  
2  
3 foo = 'a' < 'b';  
4 console.log(foo);      // true  
5  
6 foo = 'a' < 'A';  
7 console.log(foo);      // false  
8  
9 foo = 'reva' < 'revika';  
10 console.log(foo);     // true  
11  
12 foo = false < true;  
13 console.log(foo);     // true  
14  
15 foo = 'joni' == 12;  
16 console.log(foo);     // false
```

Jika yang dibandingkan berupa data **string**, JavaScript menggunakan nomor urut karakter ASCII untuk menentukan huruf mana yang lebih besar. Sebagai contoh, huruf “a” memiliki kode ASCII 97 desimal, sedangkan “b” berada di urutan 98. Dengan demikian operasi 'a' < 'b' akan menghasilkan **true** karena ‘a’ dianggap lebih kecil dari ‘b’.

Begitu juga dengan operasi 'a' < 'A' ,hasilnya adalah **false**. Karena huruf ‘A’ berada di nomor urut 65 di daftar karakter ASCII, yang lebih kecil dari ‘a’. Daftar urutan karakter ASCII ini bisa anda lihat di [ascii-code.com](http://www.ascii-code.com)<sup>1</sup>.

Apabila yang dibandingkan adalah string kata atau kalimat, perbandingan dilakukan huruf per huruf. Operasi 'reva' < 'revika' akan dibandingkan dari karakter pertama, antara ‘r’ dengan ‘r’, ‘e’ dengan ‘e’, ‘v’ dengan ‘v’, hingga ‘a’ dengan ‘i’. Sampai disini, huruf ‘a’ memiliki nilai ASCII yang lebih kecil, sehingga hasil dari 'reva' < 'revika' adalah **true**.

Yang cukup unik, nilai boolean **false** dianggap lebih kecil daripada **true**, sehingga **false** < **true**, hasilnya adalah **true**.

## Perbedaan antara == dengan ===

Di dalam JavaScript, operator “**sama dengan**” ada 2 jenis, yakni **==**, dan **====**. Operator dengan 3 buah sama dengan dikenal juga sebagai “**identik dengan**”, dan hanya menghasilkan **true** jika nilai yang dibandingkan memiliki tipe data yang sama.

Berikut contoh perbedaannya:

---

<sup>1</sup><http://www.ascii-code.com/>

```
1 var foo;  
2  
3 foo = 9 == '9';  
4 console.log(foo);      // true  
5  
6 foo = 9 === '9';  
7 console.log(foo);      // false  
8  
9 foo = true == 1;  
10 console.log(foo);     // true  
11  
12 foo = true === 1;  
13 console.log(foo);     // false  
14  
15 foo = false == 0;  
16 console.log(foo);     // true  
17  
18 foo = false === 0;  
19 console.log(foo);     // false
```

Untuk operator “sama dengan” ( == ), jika terdapat operasi yang melibatkan dua tipe data berbeda, JavaScript akan mencoba menkonversi tipe data tersebut ke tipe data lain, kemudian baru dibandingkan.

Dalam contoh diatas, operator == akan menghasilkan **true** untuk perbandingan `9 == '9'`, `true == 1`, dan `false == 0`.

Ini terjadi karena tipe data string '9', dianggap sama dengan tipe data number '9'. Begitu juga angka 1 dianggap sama dengan **true**, dan angka 0 dianggap sama dengan **false**.

Sedangkan operator “identik dengan” ( === ) akan mengecek apakah apakah tipe data yang dibandingkan tersebut juga berbeda tipe data. Jika iya, hasilnya langsung **false**. Operator ini hanya akan menghasilkan **true** jika kedua data bernilai sama dan memiliki tipe data yang sama.

Secara umum, menggunakan operator “identik dengan” akan lebih aman dibandingkan operator “sama dengan”. Kita bisa mengeliminasi kemungkinan error atau bug yang terjadi disebabkan konversi tipe data saat operasi perbandingan.

Operator perbandingan banyak digunakan untuk membuat percabangan kode program, seperti struktur **if else**. Kita akan membahas struktur percabangan ini dalam bab selanjutnya.

## Falsy and Truthy Value

Operator perbandingan == dan === cukup sering digunakan dalam pemrograman, misalnya untuk mengecek apakah password yang diinput sudah sesuai dengan yang ada di sistem atau tidak, atau apakah sebuah fungsi menghasilkan nilai **false** atau **tidak**.

Yang perlu menjadi perhatian, di dalam JavaScript sebuah tipe data akan berubah menjadi tipe data lain tergantung operator yang digunakan. Untuk operator perbandingan, tipe data ini akan dikonversi menjadi **boolean**, apakah itu menjadi boolean **true**, atau boolean **false**.

Nilai yang hasil konversinya menjadi *false* dikenal dengan istilah **falsy value**, sedangkan yang akan dikonversi menjadi *true* disebut sebagai **truthy value**.

Berikut daftar falsy value di dalam JavaScript:

- **false**
- **null**
- **undefined**
- **0**
- **NaN**
- **''** (string kosong)
- **""** (string kosong)

Selain nilai-nilai diatas, akan di konversi menjadi **truthy value**, termasuk:

- **true**
- **{ }** (object kosong)
- **[ ]** (array kosong)
- **42** (sembarang angka, termasuk angka negatif dan pecahan, selain angka 0)
- **"foo"** (sembarang string, selama bukan string kosong)
- **infinity** (termasuk -infinity)

Di dalam operasi perbandingan, nilai-nilai ini sering menjadi tidak terduga, seperti contoh berikut:

```
1 var foo;  
2  
3 foo = '' == '0';  
4 console.log(foo);      // false  
5  
6 foo = 0 == '' ;  
7 console.log(foo);      // true  
8  
9 foo = 0 == '0';  
10 console.log(foo);     // true  
11  
12 foo = false == 'false';  
13 console.log(foo);     // false  
14  
15 foo = false == '0';  
16 console.log(foo);     // true  
17  
18 foo = false == undefined;  
19 console.log(foo);     // false
```

```

20
21 foo = false == null ;
22 console.log(foo);      // false
23
24 foo = null == undefined ;
25 console.log(foo);      // true
26
27 foo = ' \t\r\n ' == 0 ;
28 console.log(foo);      // true

```

Untuk menghindari hasil yang ‘aneh’ seperti diatas, disarankan untuk selalu menggunakan operator indetik dengan ( === ) daripada operator sama dengan ( == ). Jika anda menggunakan teks editor **Komodo Edit**, semua kode diatas akan diberikan warning supaya kita menukaranya dengan operator ( === ).

## 7.4 Operator Logika

Operator logika digunakan untuk membandingkan 2 kondisi logika, yakni logika benar (**true**) dan logika salah (**false**). Nilai yang dibandingkan harus bertipe boolean. Jika tidak, JavaScript akan mengkonversinya secara otomatis berdasarkan **falsy** dan **truthy value**.

Tabel berikut merangkum jenis-jenis operator logika dalam JavaScript:

Nama Operator	Operator	Contoh	Hasil
and	&&	a && b	true, jika a dan b sama-sama bernilai true
or		a    b	true, jika salah satu dari a atau b bernilai true
not	!	! a	true, jika a bernilai false.



Jika anda pernah belajar bahasa pemrograman lain, di JavaScript tidak terdapat operator XOR, yakni dimana hasilnya **true** jika salah satu dari a atau b bernilai **true**, tapi tidak keduanya.

Berikut contoh penggunaan operator logika di dalam JavaScript:

```

1 var foo;
2
3 foo = true && false;
4 console.log(foo);      // false
5
6 foo = true || false;
7 console.log(foo);      // true
8
9 foo = !false;

```

```
10 console.log(foo);      // true
11
12 foo = true || true && false;
13 console.log(foo);      // true
14
15 foo = false && true || true;
16 console.log(foo);      // true
```

Di dalam JavaScript, kita harus menggunakan operator (`&&`) untuk logika **and**, dan (`||`) untuk logika **or**. JavaScript akan error jika saya membuat `var foo = true and false`, karena “**and**” dan “**or**” bukanlah sebuah operator sebagaimana layaknya di bahasa pemrograman PHP.

Operator `&&` memiliki kekuatan yang lebih tinggi daripada `||`. Sebagai contoh, `var foo = true || true && false` akan dijalankan sebagai `var foo = true || (true && false)`, sehingga hasilnya **true**. Begitu juga untuk operasi `var foo = false && true || true` akan dijalankan sebagai `var foo = (false && true) || true`, dimana hasilnya juga **true**.

## Konsep Short-circuit Evaluation

Ketika memproses operasi logika yang cukup panjang, JavaScript menggunakan konsep **short-circuit evaluation**. Maksudnya, jika dengan memeriksa 1 nilai saja hasil operasi tersebut sudah diketahui, nilai - nilai lain tidak akan di periksa.

Sebagai contoh, perhatikan kode program berikut:

```
1 var foo;
2
3 foo = true || false || true;
4 console.log(foo);      // true
5
6 foo = false && true && true;
7 console.log(foo);      // false
```

Untuk operasi logika, JavaScript menjalankan perintah tersebut dari kiri ke kanan, kecuali jika terdapat operator `&&` dan `||` dalam 1 operasi (operator `&&` akan dijalankan terlebih dahulu).

Pada contoh pertama, JavaScript mulai memproses dari **true**, kemudian ditemukan operator `||`. Sampai disini, JavaScript “paham” bahwa apapun nilai disisi kanan operator `||`, hasilnya sudah pasti **true**. Sehingga program akan berhenti sampai disini dan variabel `foo` akan bernilai **true**.

Hal yang sama juga berlaku untuk operasi kedua. Ketika JavaScript melihat `“false && ”`, proses akan langsung berhenti karena sudah bisa ipastikan hasilnya adalah **false**. Karena, untuk operator `&&`, jika salah satu saja nilainya terdapat **false**, hasilnya sudah pasti **false**.

Jika nilai pertama **true**, operator `&&` belum bisa menyimpulkan hasil akhirnya, dan harus memeriksa nilai setelah operator `&&`.

Fitur **short-circuit evaluation** ini cukup penting dipahami karena sering digunakan dalam kode program lanjutan, misalnya dalam contoh berikut:

```
1 var bar = false;
2 var foo = bar && alert("Hello Indonesia");
```

Disini, fungsi `alert()` baru akan berjalan jika variabel `bar` bernilai `true`. Jika variabel hasil bernilai `false`, efek *short-circuit* akan memutus kode program dan `alert()` tidak akan diproses.

Jika saya mengubahnya menjadi seperti ini:

```
1 var bar = true;
2 var foo = bar && alert("Hello Indonesia");
```

Sekarang akan tampil text box "Hello Indonesia". Ini terjadi karena JavaScript mendeteksi variabel `bar` bernilai `true`, sedangkan operatornya adalah `&&`. Short-circuit tidak terjadi karena JavaScript harus memeriksa nilai berikutnya.

Fitur **short-circuit evaluation** untuk operator logika ini juga sering digunakan ketika membuat trik khusus, dimana tipe data yang dibandingkan bukan boolean.

## Operasi Logika non Boolean

Salah satu fitur unik di dalam JavaScript adalah, jika operasi logika dijalankan untuk data non boolean, hasil akhir dari operasi ini juga bukan berupa boolean, tapi nilai terakhir dari pemrosesan tersebut.

Perhatikan hasil dari operasi-operasi berikut:

```
1 var foo = "DuniaIlkom" || "JavaScript";
2 console.log(foo);           // DuniaIlkom
3
4 var foo = "DuniaIlkom" && "JavaScript";
5 console.log(foo);           // JavaScript
6
7 var foo = true || "JavaScript";
8 console.log(foo);           // true
9
10 var foo = false || "JavaScript";
11 console.log(foo);           // JavaScript
12
13 var foo = "JavaScript" && false;
14 console.log(foo);           // false
15
16 var foo = false && "JavaScript";
17 console.log(foo);           // false
18
19 var foo = false || false && true || "JavaScript";
20 console.log(foo);           // JavaScript
21
22 var foo = true || false && true || "JavaScript";
23 console.log(foo);           // true
```

Hasil yang didapat memang agak “aneh”. Jika saya menjalankan kode program ini di PHP, hasilnya hanya *boolean*, apakah itu **true** atau **false**. Tapi di JavaScript, hasilnya bisa berupa string. Mari kita bahas satu persatu.



Konsep yang akan saya jelaskan mungkin sedikit rumit, tapi cukup penting untuk dipahami. Saya tidak akan heran jika anda harus mengulang membaca setiap paragraf beberapa kali untuk bisa paham. Selain itu, anda juga harus memahami bahwa operasi logika di proses dari kiri ke kanan, dan menggunakan prinsip **short-circuit evaluation**.

Operasi pertama adalah:

```
var foo = "DuniaIlkom" || "JavaScript";
```

Hasilnya berupa string "DuniaIlkom".

Ketika baris tersebut di proses, JavaScript melihat nilai pertama berupa string "DuniaIlkom", ini dipahami sebagai nilai **true** (lihat kembali konsep *falsy* dan *truthy value*). Setelah itu ditemukan operator **||**. Sesuai dengan *prinsip short-circuit*, proses akan berhenti karena apapun nilai setelah **||**, tidak akan mempengaruhi hasil akhir dari operator “or”. Hasil akhirnya pasti **true**.

Karena proses berhenti sampai disini, nilai **true** ini seharusnya dikembalikan ke dalam variabel **foo** (**foo** akan berisi boolean **true**). Akan tetapi karena tipe data asal adalah string "DuniaIlkom", nilai string inilah yang dikembalikan ke dalam **foo**, bukan **true**.

Mari lanjut ke operasi kedua, yakni:

```
var foo = "DuniaIlkom" && "JavaScript";
```

Jika dibandingkan dengan contoh pertama, yang berubah hanya operator dari **||** menjadi **&&**.

Saat baris ini diproses, JavaScript melihat nilai pertama adalah string "DuniaIlkom", ini dipahami sebagai nilai **true**. Setelah itu terdapat operator **&&**. Sampai disini, konsep *short-circuit* tidak bisa dijalankan. Jika nilai berikutnya **false**, hasil dari operasi ini adalah **false**, jika **true**, hasil akhirnya adalah **true**. JavaScript harus melihat nilai kedua.

Ternyata nilai kedua adalah string "JavaScript", yang artinya juga **true**. Kembali, nilai **true** seharusnya yang diisi ke dalam variabel **foo**. Namun karena tipe data asal adalah string "JavaScript", nilai ini yang dikembalikan ke dalam **foo**, bukan **true**.

Tunggu dulu, kenapa yang dikembalikan "JavaScript", bukannya "DuniaIlkom"? Bukankah ada 2 nilai yang dibandingkan? keduanya juga sama-sama bernilai **true**?

Konsep lain dari operasi seperti ini adalah, JavaScript akan mengembalikan nilai **dari posisi terakhir yang diperiksa**. Karena yang terakhir diperiksa terakhir adalah "JavaScript", hasil akhirnya berupa string "JavaScript", bukan "DuniaIlkom".

Lanjut, pada baris berikutnya terdapat operasi:

```
var foo = true || "JavaScript";
```

Jika anda sudah memahami penjelasan sebelumnya saya yakin anda bisa paham kenapa hasilnya berupa boolean `true`. Yup, karena yang diproses adalah `true` dan ketemu operator `||`, konsep *short-circuit* akan berjalan, dan nilai `true` dikembalikan ke variabel `foo`.

Operasi selanjutnya juga sangat mirip:

```
var foo = false || "JavaScript";
```

Tapi karena yang diproses pertama kali adalah `false` dan ketemu operator `||`, konsep *short-circuit* tidak bisa dijalankan. JavaScript harus melihat nilai berikutnya, yakni string `"JavaScript"`. Hasil akhir dari proses ini adalah `true`, yang tipe data aslinya berupa string `"JavaScript"`.

Berikutnya terdapat operasi:

```
var foo = "JavaScript" && false;
```

Apakah *short-circuit* bisa dijalankan? **Tidak!** sebab nilai pertama yang diperoleh adalah string `"JavaScript"` (`true`), sehingga harus dilihat apa nilai berikutnya. Ternyata nilai kedua adalah boolean `false`, nilai inilah yang dikembalikan ke variabel `foo`.

Operasi selanjutnya juga sangat mirip:

```
var foo = false && "JavaScript";
```

Namun karena nilai `false` bertemu dengan operator `&&`, proses akan langsung berhenti dan mengembalikan nilai `false` (terjadi *short-circuit*).

Dua operasi terakhir melibatkan 3 operator logika, yang pertama adalah:

```
var foo = false || false && true || "JavaScript";
```

Sama seperti sebelumnya, kita harus menjalankan kode ini dari kiri ke kanan untuk memeriksa apakah terdapat *short-circuit*. Yang juga tidak boleh dilupakan, operator `&&` memiliki kekuatan yang lebih tinggi, sehingga akan diproses terlebih dahulu.

Dengan demikian, operasi ini dijalankan sebagai berikut:

```
var foo = false || (false && true) || "JavaScript";
```

Yang akan menjadi:

```
var foo = false || false || "JavaScript";
```

Sekarang, operasi akan diproses dari kiri ke kanan. Pertanyaan berikutnya, apakah bisa terjadi *short-circuit*? **Tidak!** JavaScript terpaksa menjalankan seluruh operator dari kiri ke kanan, sehingga yang terakhir diperiksa adalah string `"JavaScript"`.

Operasi baris terakhir adalah:

```
var foo = true || false && true || "JavaScript";
```

Dapatkah anda menjelaskan kenapa hasil akhirnya adalah `true`? Yup, disini terjadi *short-circuit*. Konsep *operasi logika non boolean* di JavaScript ini sangat unik dan jarang ditemukan dalam bahasa pemrograman lain.

## 7.5 Operator String

Dalam JavaScript terdapat 1 operator yang digunakan untuk **penyambungan string** (*string concatenation*). Operator ini menggunakan karakter tambah (`+`). Jika data yang akan disambung bukan bertipe string, akan dikonversi menjadi string secara otomatis.

Berikut contoh penggunaan operator penyambungan string di dalam JavaScript:

```
1 var foo, bar, baz;
2
3 foo = "Sedang " + "Belajar " + "JavaScript";
4 console.log(foo);           // Sedang Belajar JavaScript
5
6 foo = "Belajar " + "ECMAScript " + 7;
7 console.log(foo);           // Belajar ECMAScript 7
8
9 foo = 'Selamat ';
10 bar = "Malam ";
11 baz = `Indonesia`;
12 hasil = foo + bar + baz;
13 console.log(hasil);        // Selamat Malam Indonesia
```

Dalam contoh ini saya menyambung berbagai string, mulai dari yang dibuat langsung (*string literal*), juga string yang berada di dalam variabel.

Perhatikan juga bahwa tidak ada perbedaan string yang dibuat menggunakan tanda kutip satu (`'`), kutip dua (`"`), maupun dengan backtick, seperti yang terlihat pada contoh terakhir.

Operator `+` harus digunakan jika kita ingin mengambung string dengan sebuah variabel, seperti contoh berikut:

```
1 var foo, bar, baz;
2 var siswa = ["Andri", "Joko", "Sukma", "Rina", "Sari"];
3
4 foo = siswa[1] + " mendapat penghargaan sebagai siswa teladan";
5 console.log(foo);
6 // Joko mendapat penghargaan sebagai siswa teladan
7
8 bar = siswa[3] + ' dan ' + siswa[4] + ' adalah teman akrab';
9 console.log(bar);
10 // Rina dan Sari adalah teman akrab
11
12 baz = "Dalam ujian kemaren, " + siswa[0] + ' mendapat nilai ' + 4*20;
13 console.log(baz);
14 // Dalam ujian kemaren, Andri mendapat nilai 80
```

Kali ini saya memiliki array `siswa` yang berisi 5 element. Untuk membuat string yang di dalamnya terdapat element array, kita bisa menggunakan operator plus (+) untuk ‘keluar’ dan ‘masuk’ dari string.

Sebagai contoh, perhatikan kode berikut:

```
bar = siswa[3] + ' dan ' + siswa[4] + ' adalah teman akrab';
```

Untuk bisa menampilkan isi element array dari `siswa` ke-4 (dengan index = 3), saya merangkai variabel `bar` dengan menulis `siswa[3]`. Ini merupakan perintah JavaScript untuk mengakses element array.

Perintah `siswa[3]` harus ditulis di luar string, karena jika ditulis dari dalam string akan menjadi '`siswa[3]`', bukan `Rina` yang ada di array `siswa`.

Kemudian saya menyambung `siswa[3]` dengan string ' dan '. Karena ini merupakan string, jadi harus disambung menggunakan operator +. Setelah itu saya ‘keluar’ lagi dari string dan mengakses `siswa[4]`. Terakhir kembali disambung dengan string ' adalah teman akrab'.

Hasil kode program diatas adalah: `Rina dan Sari adalah teman akrab`.

Teknik penyambungan seperti ini **sangat sangat sering dilakukan**. Kuncinya, anda harus paham bagaimana cara merangkai string tersebut. Jika itu adalah kode JavaScript (seperti variabel), harus ditulis di luar string.

Saya tidak bisa menulis seperti ini:

```
bar = 'siswa[3] dan siswa[4] adalah teman akrab';
```

Karena `siswa[3]` dan `siswa[4]` akan diproses sebagai bagian dari string, bukan kode JavaScript untuk mengakses element array `siswa`.

Dengan fitur *template string* dari ECMAScript 6, kita bisa menulis semuanya di dalam string, seperti contoh berikut:

```
1 var foo, bar, baz;
2 var siswa = ["Andri", "Joko", "Sukma", "Rina", "Sari"];
3
4 foo = `${siswa[1]} mendapat penghargaan sebagai siswa teladan`;
5 console.log(foo);
6 // Joko mendapat penghargaan sebagai siswa teladan
7
8 bar = `${siswa[3]} dan ${siswa[4]} adalah teman akrab`;
9 console.log(bar);
10 // Rina dan Sari adalah teman akrab
11
12 baz = `Dalam ujian kemaren ${siswa[0]} mendapat nilai + {4*20}`;
13 console.log(baz);
14 // Dalam ujian kemaren, Andri mendapat nilai 80
```

Disini saya membuat string menggunakan karakter *backtick*, kemudian menggunakan tanda \$ dan kurung kurawal untuk mengakses element array siswa secara langsung. Jika anda sudah mempelajari PHP, penulisan seperti ini terasa sangat familiar.

## Penyambungan String atau Penambahan Angka

Di dalam JavaScript, operator penyambungan string dan penambahan angka (aritmatika) sama-sama menggunakan tanda tambah ( + ). Karena sifat JavaScript yang bisa mengubah tipe data secara otomatis, ini bisa menjadi efek yang ‘menjebak’.

Perhatikan kode program berikut:

```
1 var foo;
2
3 foo = 10 + 10 + 9;
4 console.log(foo); // 29
5
6 foo = '10' + 10 + 9;
7 console.log(foo); // 10109
8
9 foo = 10 + '10' + 9;
10 console.log(foo); // 10109
11
12 foo = 10 + 10 + '9';
13 console.log(foo); // 209
```

Kali ini saya membuat beberapa contoh kasus. Pada contoh pertama, `foo = 10 + 10 + 9` akan menghasilkan nilai 29. Tidak ada yang aneh karena seluruh nilai yang ditambahkan bertipe **number**.

Di contoh kedua, saya membuat `foo = '10' + 10 + 9`. Perhatikan bahwa ‘angka’ pertama bukanlah tipe data number, tapi sebuah string yang isinya karakter angka 10. Hasil dari kode

ini adalah 10109. Artinya, JavaScript mengkonversi number 10 dan 9 menjadi string. Efek yang sama juga terjadi untuk `foo = 10 + '10' + 9`.

Contoh terakhir cukup unik: `foo = 10 + 10 + '9'`. Hasilnya adalah 209. Ini terjadi karena yang menjadi string terletak di akhir. JavaScript akan menambahkan terlebih dahulu  $10 + 10 = 20$ , kemudian baru digabung dengan string 9 menjadi 209.

Jika anda mengecek tipe data `foo` untuk setiap contoh diatas, akan terlihat bahwa semuanya menjadi string, kecuali contoh pertama.

Prilaku operator tambah (`+`) yang bisa mengganti tipe data menjadi string ini cukup penting dipahami, karena bisa jadi yang kita ingin menjumlahkan angka, tapi hasilnya malah penggabungan string.

## 7.6 Operator Bitwise

**Operator bitwise** adalah operator khusus untuk menangani operasi logika bilangan biner. Bilangan biner atau *binary* adalah jenis bilangan yang hanya terdiri dari 2 karakter angka, yakni 0 dan 1. Contohnya 1100 dan 1110 0110.

Operator **bitwise** sendiri sangat jarang digunakan dalam JavaScript dan cukup rumit. Selain itu anda perlu memahami konsep dasar tentang bilangan biner terlebih dahulu.



Penjelasan tentang operator bitwise ini boleh dilewati jika kurang paham. Hingga akhir buku nanti, saya tidak akan menggunakan operator ini. Operator **bitwise** tetap saya bahas agar melengkapi materi tentang operator di JavaScript.

JavaScript menyediakan 6 operator **bitwise**, sebagaimana yang bisa dilihat dari tabel berikut:

Nama	Operator	Contoh	Hasil
and	&	a & b	1, jika kedua bit bernilai 1.
or (inclusive or)		a   b	1, jika salah satu bit bernilai 1.
xor (exclusive or)	^	a ^ b	1, jika salah satu bit bernilai 1, tapi bukan keduanya
not (negasi)	~	~ a	bit 0 menjadi 1, dan bit 1 menjadi 0.
shift left	<<	a << b	menggeser sebanyak b bit ke kiri (setiap 1 kali pergeseran = kelipatan 2)
shift right	>>	a >> b	menggeser sebanyak b bit ke kanan (setiap 1 kali pergeseran = bagi 2)

Berikut contoh penggunaannya:

```
1 var a = 0b10101010; // desimal 170
2 var b = 0b11011110; // desimal 222
3
4 console.log(a);      // 170
5 console.log(b);      // 222
6
7 var foo;
8
9 foo = a & b;
10 console.log(foo); // 138 desimal = 1000 1010 biner
11
12 foo = a | b;
13 console.log(foo); // 254 desimal = 1111 1110 biner
14
15 foo = a ^ b;
16 console.log(foo); // 116 desimal = 0111 0100 biner
17
18 foo = ~a;
19 console.log(foo); // -171 desimal
20
21 foo = a >> 2;
22 console.log(foo); // 42 desimal = 0010 1010 biner
23
24 foo = a << 3;
25 console.log(foo); // 1360 desimal = 0101 0101 0000 biner
```

Di awal program, saya mendefenisikan 2 buah variabel: a dan b. Kedua variabel ini diisi angka biner. Variabel a berisi 10101010, sedangkan variabel b berisi 11011110. Cara penulisan angka biner telah kita bahas dalam bab sebelumnya.

Saya mengisi kedua variabel ini dengan angka biner karena seluruh operator bitwise beroperasi dalam **bit**, yakni sebutan untuk setiap angka 1 dan 0 dari bilangan biner.

Operasi **and** (`&`) akan menghasilkan angka 1 jika nilai bit dari kedua variabel bernilai 1. Jika tidak, akan menjadi nol. Berikut cara perhitungan dari `a & b`:

```
a =      10101010
b =      11011110
-----
a & b =  10001010  = 138 (desimal)
```

Operasi **or** (`|`) akan menghasilkan angka 1 jika salah satu nilai bit dari kedua variabel bernilai 1. Jika tidak, akan menjadi nol. Berikut perhitungan dari `a | b`:

```
a =      10101010
b =      11011110
-----
a | b = 11111110 = 254 (desimal)
```

Operasi **xor** (^) akan menghasilkan angka 1 jika salah satu dari kedua variabel bernilai 1, tapi tidak jika keduanya sama-sama 1 atau sama-sama 0. Berikut perhitungan dari a ^ b:

```
a =      10101010
b =      11011110
-----
a ^ b = 01110100 = 116 (desimal)
```

Operasi **not** (~) akan membalikkan nilai bit dari 0 menjadi 1, dan dari 1 menjadi nol. Namun perhitungan operasi not ini menjadi rumit karena menggunakan metode yang dinamakan “**two’s complement**”. Penjelasan tentang “**two’s complement**” dapat anda baca lebih lanjut di [wikipedia](#)<sup>2</sup>.

Berikut perhitungan dari ~a:

```
a      = 000000000000000000000000000010101010 (32 bit)
-----
~a      = 11111111111111111111111111110101010 (32 bit negative)
Flip & -1 = 000000000000000000000000000010101010 - 1
~a      = 000000000000000000000000000010101011 = -171 (desimal)
```

Operator **shift right** (>>) akan menggeser seluruh bit yang ada ke kanan sebanyak beberapa tempat. Jika ditulis a >> 2, artinya bit-bit penyusun variabel a akan digeser 2 kali ke kanan. Berikut proses yang terjadi:

```
$a      = 10101010 = 170(desimal)
$a >> 1 = 01010101 = 85 (desimal)
$a >> 2 = 00101010 = 42 (desimal)
```

Karena digeser ke kanan, otomatis bit paling kanan akan ‘hilang’, dan disisi kiri diisi dengan angka 0. Setiap penggeseran 1 tempat ke kanan akan membagi 2 nilai asal. Dalam contoh, hasilnya adalah  $170/2 = 85$ , dan  $85/2 = 42$  (dibulatkan).

Operator **shift left** (<<) akan menggeser seluruh bit yang ada ke kanan sebanyak beberapa tempat. Jika ditulis a << 3, artinya bit-bit penyusun variabel a akan digeser 3 kali ke kiri. Berikut proses yang terjadi:

---

<sup>2</sup>[http://en.wikipedia.org/wiki/Two%27s\\_complement](http://en.wikipedia.org/wiki/Two%27s_complement)

```
$a      = 10101010    = 170  (desimal)
$a << 1 = 101010100   = 340  (desimal)
$a << 2 = 1010101000  = 680  (desimal)
$a << 3 = 10101010000 = 1360 (desimal)
```

Karena digeser ke kiri, bit paling kiri akan pindah ke tempat yang lebih tinggi, kemudian disisi kanan diisi dengan angka 0. Setiap penggeseran 1 tempat ke kanan akan mengali 2 nilai asal. Dalam contoh, hasilnya adalah  $170 \times 2 = 340$ ,  $340 \times 2 = 680$ ,  $680 \times 2 = 1360$ .

## 7.7 Operator Assignment

**Operator assignment** adalah operator yang digunakan untuk memasukkan nilai ke dalam sebuah variabel. Kita sudah berkali-kali menggunakan operator ini, yakni tanda sama dengan (`=`).

Berikut contoh penggunaan operator assignment:

```
1 var a = 12;
2 var b = "Belajar";
3
4 a = a + 10;
5 a = a + a + 5;
6
7 b = b + " JavaScript";
8 b = b + " dari buku JavaScript Uncover";
9
10 console.log(a); // 49
11 console.log(b); // Belajar JavaScript dari buku JavaScript Uncover
```

Dalam 2 baris pertama, saya mendefenisikan variabel `a` dan `b` yang langsung diisi dengan nilai awal 12 dan "Belajar".

Selanjutnya saya membuat perintah `a = a + 10`, ini artinya variabel `a` ditambah 10, baru kemudian disimpan kembali ke dalam variabel `a`, hasilnya variabel `a` sekarang berisi angka  $12 + 10 = 22$ .

Konsep seperti ini sangat penting dipahami, karena kita akan sering membuat kode program yang diisi dengan variabel itu sendiri. Pada perintah `a = a + 10`, yang diproses terlebih dahulu adalah operasi di sisi kanan tanda sama dengan. Dengan kata lain, bisa ditulis seperti ini: `a = (a + 10)`.

Dibaris selanjutnya saya membuat `a = a + a + 5`. Dapatkah anda menebak berapa hasil akhir dari variabel `a`? Yup, 49. Yang dijalankan adalah seperti ini: `a = (a + a + 5)`, menjadi `a = (22 + 22 + 5)`, yang akhirnya `a = 49`.

Contoh berikutnya saya melakukan operasi *assignment* untuk tipe data string (penyambungan string). Konsep dasarnya sama seperti variabel `a`, dimana sisi kanan tanda sama dengan akan diproses terlebih dahulu.

Operasi `b = b + " JavaScript"`, diproses sebagai `b = "Belajar" + " JavaScript"`, menjadi `b = "Belajar JavaScript"`.

Dibaris terakhir, saya menambahkan lagi `b = b + " dari buku JavaScript Uncover"`, yang diproses menjadi `b = ("Belajar JavaScript" + " dari buku JavaScript Uncover")`. Hasil akhirnya variabel `b` berisi string "Belajar JavaScript dari buku JavaScript Uncover".

## Operator Gabungan Assignment

Perintah asignment seperti `a = a + 10`, atau `b = b + " JavaScript"` sangat sering digunakan. Karena itu pula tersedia penulisan singkat dari perintah ini, yang bisa disebut sebagai *operator gabungan assignment*.

Sebagai contoh, `a = a + 10` dapat ditulis menjadi `a += 10`, dan `b = b + " JavaScript"` ditulis sebagai `b += " JavaScript"`. Tabel berikut merangkum seluruh operator gabungan asignment dalam JavaScript:

Nama Operator	Operator	Contoh	Hasil
plus-equals	<code>+=</code>	<code>b += 10</code>	<code>b = b + 10</code>
minus-equals	<code>-=</code>	<code>b -= 10</code>	<code>b = b - 10</code>
divide>equals	<code>/=</code>	<code>b /= 10</code>	<code>b = b / 10</code>
multiply>equals	<code>*=</code>	<code>b *= 10</code>	<code>b = b * 10</code>
modulus>equals	<code>%=</code>	<code>b %= 10</code>	<code>b = b % 10</code>
bitwise-xor>equals	<code>^=</code>	<code>b ^= 10</code>	<code>b = b ^ 10</code>
bitwise-and>equals	<code>&amp;=</code>	<code>b &amp;= 10</code>	<code>b = b &amp; 10</code>
bitwise-or>equals	<code> =</code>	<code>b  = 10</code>	<code>b = b   10</code>
left-shift>equals	<code>&lt;&lt;=</code>	<code>b &lt;&lt;= 10</code>	<code>b = b &lt;&lt; 10</code>
right-shift>equals	<code>&gt;&gt;=</code>	<code>b &gt;&gt;= 10</code>	<code>b = b &gt;&gt; 10</code>

Berikut contoh penggunaannya:

```

1 var a = 12;
2 var b = "Belajar";
3
4 a += 10;
5 a += a + 5;
6 console.log(a); // 49
7
8 b += " JavaScript";
9 b += " dari buku JavaScript Uncover";
10 console.log(b); // Belajar JavaScript dari buku JavaScript Uncover
11
12 a /= 7;

```

```
13 console.log(a); // 7
14
15 a -= 5;
16 console.log(a); // 2
```

Operator gabungan asignment ini murni untuk mempersingkat penulisan (bukan sebuah keharusan). Menggunakan cara yang normal seperti `a = a + 5` juga dibolehkan.

## 7.8 Operator Spread

**Operator spread**, atau **spread operator**, merupakan operator baru di **ECMAScript 6**. Operator ini digunakan untuk berbagai keperluan yang berhubungan dengan *array*, salah satunya untuk menggabungkan array.

*Spread operator* menggunakan tanda titik tiga kali (...), kemudian diikuti dengan nama variabel. Berikut contoh penggunaannya:

```
1  var nilai1 = ["a", "b", "c", "d"];
2  console.log(nilai1);
3  // Array [ "a", "b", "c", "d" ]
4
5  var nilai2 = [1, 2, 3, 4];
6  console.log(nilai2);
7  // Array [ 1, 2, 3, 4 ]
8
9  var nilai3 = [...nilai1, "e", "f"];
10 console.log(nilai3);
11 // Array [ "a", "b", "c", "d", "e", "f" ]
12
13 var nilai4 = [0, ...nilai2, 5, 6];
14 console.log(nilai4);
15 // Array [ 0, 1, 2, 3, 4, 5, 6 ]
16
17 var nilai5 = [...nilai2, ...nilai3];
18 console.log(nilai5);
19 // Array [ 1, 2, 3, 4, "a", "b", "c", "d", "e", "f" ]
```

Di awal kode program, saya membuat 2 buah array, yakni array `nilai1` yang berisi 4 karakter huruf, dan array `nilai2` yang berisi 4 angka.

Perhatikan proses pengisian array `nilai3`, saya menulis `var nilai3 = [...nilai1, "e", "f"]`. Tanda `...nilai1` adalah penulisan dari *spread operator*, yang tujuannya mengakses seluruh element dari array `nilai1`. Dengan kata lain, saya ingin membuat array `nilai3` yang isinya terdiri dari seluruh array `nilai1` ditambah 2 karakter lagi, yakni `"e"` dan `"f"`.

Proses pembuatan array `nilai4` juga kurang lebih sama, dimana saya menulis `var nilai4 = [0, ...nilai2, 5, 6]`. Artinya variabel array `nilai4` terdiri dari angka 0, seluruh element dari `array2`, dan ditambah 2 angka baru, yakni 5 dan 6.

Terakhir untuk array `nilai5`, elementnya berasal dari array `nilai2` dan `nilai3`. Ini didapat dari penggunaan operator spread `var nilai5 = [...nilai2, ...nilai3]`.

The screenshot shows a browser window titled "Belajar JavaScript". The address bar shows the file path: "file:///D:/belajar\_javascript/bab\_07\_operator/20.spi". The browser's toolbar includes icons for back, forward, search, and other navigation functions. Below the toolbar is a tab bar with "Console" selected. The main area of the browser displays the output of several JavaScript code snippets. The snippets are as follows:

```

Array [ "a", "b", "c", "d" ]
Array [ 1, 2, 3, 4 ]
Array [ "a", "b", "c", "d", "e", "f" ]
Array [ 0, 1, 2, 3, 4, 5, 6 ]
Array [ 1, 2, 3, 4, "a", "b", "c", "d", "e", "f" ]

```

Each line of output is preceded by the word "Array" and followed by a series of elements enclosed in brackets. To the right of each line, there is a timestamp: "20.spread\_op... :11:3", "20.spread\_op... :14:3", "20.spread\_op... :17:3", "20.spread\_op... :20:3", and "20.spread\_op... :23:3".

Gambar: Hasil *spread operator* dalam penggabungan array

Selain untuk menggabungkan array, *spread operator* ini juga bisa digunakan untuk berbagai hal lain, misalnya di dalam *function*. Kita akan membahasnya pada bab khusus tentang *function*.

## 7.9 Urutan Operator dalam JavaScript

Dari sekian banyak operator di dalam JavaScript, terdapat aturan tentang operator mana yang akan didahulukan. Misalnya untuk operator aritmatika, operasi perkalian akan didahulukan daripada operasi penambahan.

Tabel berikut merangkum urutan operator di dalam JavaScript. Operator paling atas memiliki kekuatan paling tinggi.

Operator type	Individual operators
member	<code>.</code> <code>[]</code>
call / create instance	<code>()</code> <code>new</code>
negation/increment	<code>!</code> <code>~</code> <code>-</code> <code>++</code> <code>--</code>
multiply/divide	<code>*</code> <code>/</code> <code>%</code>
addition/subtraction	<code>+</code> <code>-</code>
bitwise shift	<code>&lt;&lt;</code> <code>&gt;&gt;</code> <code>&gt;&gt;&gt;</code>
relational	<code>&lt;</code> <code>&lt;=</code> <code>&gt;</code> <code>&gt;=</code> <code>in</code> <code>instanceof</code>
equality	<code>==</code> <code>!=</code> <code>====</code> <code>!==</code>

Operator type	Individual operators
bitwise-and	&
bitwise-xor	^
bitwise-or	
logical-and	&&
logical-or	
conditional	? :
assignment	= += -= *= /= %= <<= >>= >>>= &= ^=  =
comma	,

Beberapa operator dari tabel diatas belum kita bahas, seperti operator `new` yang digunakan untuk membuat objek, dan operator conditional (`? :`). Kita akan membahasnya dalam bab lain.

Urutan operator ini cukup penting dipahami, misalnya kenapa kode program `foo = true || true && false` akan menghasilkan `true`, bukan `false`. Ini karena operator `&&` memiliki urutan prioritas lebih tinggi daripada operator `||`.

Untuk kasus seperti ini, kita bisa menggunakan tanda kurung untuk ‘mendahulukan’ sebuah operasi. Contohnya `foo = (true || true) && false` akan menghasilkan `false`.

---

Sepanjang bab ini kita telah mempelajari berbagai operator di dalam JavaScript. Selain operator **bitwise**, semua operator ini sangat sering digunakan di dalam kode program.

Berikutnya, kita akan masuk ke struktur logika dan perulangan di dalam JavaScript.

# 8. Struktur Logika dan Perulangan

Dalam bab ini kita akan masuk ke struktur logika dan perulangan bahasa pemrograman JavaScript. Di sini akan dibahas tentang struktur logika **if**, **if else**, dan **switch**. Kemudian akan dilanjutkan dengan perulangan **for**, **while**, dan **do while**.

Konsep tentang tipe data dan operator yang kita pelajari sebelumnya sangat berguna untuk memahami struktur control ini.

## 8.1 Struktur Logika IF

Struktur logika **if** digunakan untuk membuat percabangan kode program yang bergantung apakah sebuah kondisi bisa dipenuhi atau tidak. Jika kondisi dipenuhi (bernilai *true*), jalankan kode program.

Berikut format dasar penulisan struktur logika IF:

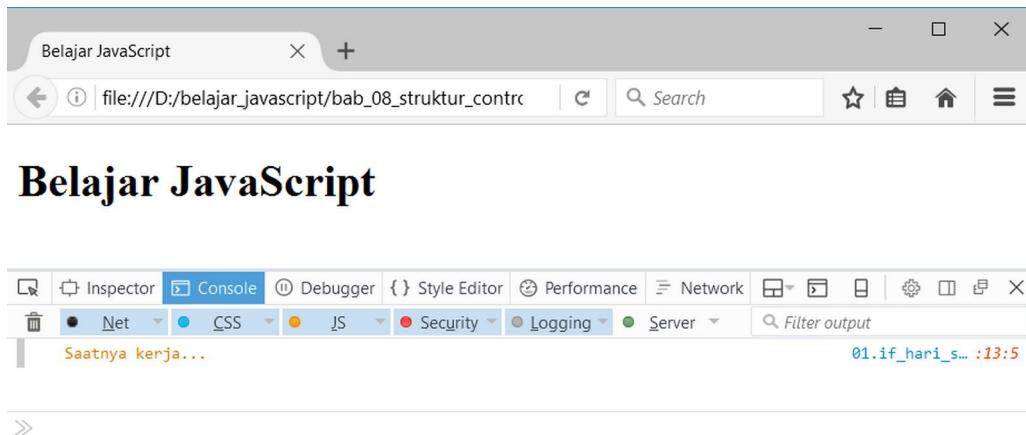
```
if (condition) {  
    statement1  
}
```

**Condition** adalah kondisi penentu. Condition ini biasanya berupa operasi perbandingan, yang akan menghasilkan nilai *true* atau *false*. Misalnya apakah  $6 > 7$ ? atau apakah variabel *user* berisi string "admin"?

Pada dasarnya, semua kode program yang bisa menghasilkan *true* atau *false*, bisa diletakkan di dalam condition. Jika condition ini dipenuhi (menghasilkan nilai *true*), jalankan kode program yang ada diantara kurung kurawal. Tanda kurung kurawal ini menandakan *block* kode program dari struktur if.

Mari kita lihat contoh prakteknya:

```
1 var hari = "senin";  
2  
3 if (hari === "senin"){  
4     console.log("Saatnya kerja...");  
5 }
```



Gambar: String “Saatnya kerja...” tampil karena kondisi terpenuhi

Dibaris pertama, saya membuat variabel `hari` yang diisi string "senin". Kode program `if (hari === "senin")` merupakan *condition* yang akan di periksa, yakni apakah variabel `hari` berisi string "senin"? Benar (hasilnya: *true*), karena itu kode program `console.log("Saatnya kerja...")` akan dieksekusi.

Mari kita coba sekali lagi:

```

1 var hari = "minggu";
2
3 if (hari === "senin"){
4   console.log("Saatnya kerja...");
```

Disini saya mengganti isi variabel `hari` dengan string "minggu". Apakah variabel `hari === "senin"`? Tidak, karena itu baris perintah `console.log("Saatnya kerja...")` tidak akan dijalankan.

Tanda kurung kurawal yang menandakan awal dan akhir block `if` sebenarnya boleh tidak ditulis, dengan syarat kode program yang dijalankan hanya 1 baris saja. Contoh saya sebelum ini hanya terdiri 1 baris, sehingga bisa ditulis sebagai berikut:

```

1 var hari = "senin";
2
3 if (hari === "senin")
4   console.log("Saatnya kerja...");
```

Namun penulisan seperti ini sangat tidak disarankan, karena jika kita lupa akan aturan ini dan menambahkan baris kedua, kode tersebut tidak akan menjadi bagian dari `if`, tapi berada di luar `if`:

```

1 var hari = "selasa";
2
3 if (hari === "senin")
4   console.log("Saatnya kerja..."); 
5   console.log("Besok hari selasa"); // ini berada di luar if

```

Ketika kode diatas dijalankan, akan tampil teks "Besok hari selasa". Padahal yang saya inginkan, string tersebut hanya tampil jika kondisi `if (hari === "senin")` dipenuhi.

Jika block if sudah lebih dari 1 baris, harus ditambahkan tanda kurung kurawal untuk mendakan awal dan akhir blok if:

```

1 var hari = "selasa";
2
3 if (hari === "senin") {
4   console.log("Saatnya kerja..."); 
5   console.log("Besok hari selasa");
6 }

```

Pada prakteknya, kondisi if sering digunakan untuk pengecekan validasi, seperti contoh berikut:

```

1 var user = "admin";
2
3 if (user === "admin"){
4   console.log("Selamat datang admin..."); 
5 }

```

Disini saya mengecek isi variabel `user`. Jika berisi string "admin", perintah `console.log("Selamat datang admin...")` akan dijalankan. Isi dari variabel `user` bisa saja berasal dari form HTML, atau dari database (diambil menggunakan bahasa pemrograman PHP).

Dalam pemrosesan form, kita juga perlu mengecek apakah nilai yang diinput user sudah sesuai atau tidak, seperti contoh berikut:

```

1 var username = "alex";
2
3 if (typeof username === "string"){
4   console.log("Username valid..."); 
5 }

```

Perhatikan bagian *condition*, saya ingin memeriksa apakah `typeof username === "string"`? Kondisi ini hanya akan bernilai *true* jika variabel `username` bertipe string. Karena bisa saja ada yang menginput nilai selain string:

```
1 var username = 22.3;
2
3 if (typeof username === "string"){
4   console.log("Username valid...");
5 }
```

Kode `console.log("Username valid...")` tidak akan dijalankan karena variabel `username` bertipe *number*, bukan *string*.

## 8.2 Struktur Logika IF ELSE

Kondisi logika if yang kita bahas sebelum ini hanya membuat 1 percabangan, yakni untuk kondisi bernilai *true*. Tapi bagaimana membuat struktur logika jika kondisi *false*? Kita bisa menggunakan percabangan `if else`.

Berikut format dasarnya:

```
if (condition) {
  statement1
}
else {
  statement2
}
```

Sekarang, jika *condition* dipenuhi (menghasilkan nilai *true*), baris *statement1* akan dijalankan. Namun jika tidak terpenuhi (*condition* menghasilkan nilai *false*), yang akan dijalankan adalah *statement2*.

Mari kita lihat contoh prakteknya:

```
1 var hari = "selasa";
2
3 if (hari === "senin") {
4   console.log("Saatnya kerja...");
5 }
6 else {
7   console.log("Bukan hari senin...");
8 }
```

Disini saya memiliki variabel `hari` yang diisi dengan string `"selasa"`. Jika kondisi `if (hari === "senin")` terpenuhi, akan dijalankan perintah `console.log("Saatnya kerja...")`. Namun jika variabel `hari` ternyata bukan berisi string `"senin"`, yang akan dijalankan adalah `console.log("Bukan hari senin...")`.

Contoh lain sebagai berikut:

```

1 var username = 22.3;
2
3 if (typeof username === "string"){
4   console.log("Username valid...");
5 }
6 else {
7   console.log("Validasi gagal...");
8 }

```

Saya ingin memeriksa apakah variabel `username` bertipe data string atau bukan. Jika iya, jalankan kode program `console.log("Username valid...")`. Jika tidak, validasi dianggap gagal dan jalankan perintah `console.log("Validasi gagal...")`.

## 8.3 Struktur Logika IF ELSE IF

Untuk kasus yang lebih rumit, kita bisa melakukan pengecekan kondisi lebih dari sekali. Caranya, dengan merangkai logika `if else` dengan `if` lain. Berikut format dasarnya:

```

if (condition1) {
  statement1
}
else if (condition2) {
  statement2
}
else {
  statement3
}

```

Disini terdapat 2 kali pengecekan kondisi. Jika `condition1` dipenuhi, jalankan `statement1`. Kalau tidak, cek apakah `condition2` dipenuhi. Jika dipenuhi, jalankan `statement2`. Jika tidak juga, jalankan `statement3`. JavaScript tidak membatasi berapa banyak `else if` yang boleh ditulis, bisa 2, 3 atau bahkan 100 kondisi.

Berikut contoh prakteknya:

```

1 var username = 22.3;
2
3 if (typeof username === "string") {
4   console.log("Username string");
5 }
6 else if (typeof username === "number") {
7   console.log("Username angka");
8 }
9 else {
10   console.log("Username bukan string dan angka");
11 }

```

Dalam kode program diatas, saya ingin memeriksa apa jenis tipe data dari variabel *username*. Jika bertipe *string*, maka yang tampil adalah `console.log("Username string")`, jika tidak, periksa lagi apakah tipe data *number*? Jika iya jalankan `console.log("Username angka")`. Jika tidak juga, jalankan `console.log("Username bukan string dan angka")`.

## 8.4 Nested IF ELSE

**Nested if else**, dikenal juga dengan *struktur if bersarang*. Disini kita membuat percabangan kode program yang cukup kompleks, dimana di dalam if, terdapat if lain.

Berikut format dasarnya:

```
if (condition1) {  
    statement1  
    if (condition1.1) {  
        statement1.1  
    }  
}  
else if (condition2) {  
    statement2  
    if (condition2.1) {  
        statement2.1  
    }  
}  
else {  
    statement3  
}
```

Untuk percabangan yang cukup banyak seperti ini, kita perlu memperhatikan awal dan akhir dari setiap blok if. Tidak jarang anda akan bingung dimana awal dari sebuah kondisi dan akhir kondisi tersebut.

Sebagai contoh praktik, silahkan anda pelajari kode program berikut:

```
1 var foo = 7;  
2 console.log("nilai foo = "+ foo);  
3  
4 if (typeof foo === "number") {  
5     console.log("foo bertipe number");  
6     if (foo >= 10) {  
7         console.log("nilai foo besar dari 10");  
8     }  
9     else {  
10        console.log("nilai foo kecil dari 10");  
11    }  
12 }
```

```

12 }
13 else if (typeof foo === "string") {
14   console.log("foo bertipe string");
15 }
16 else {
17   console.log("foo bukan bertipe number maupun string");
18 }

```



Gambar: Hasil nested loop

Kode program diatas tampak cukup ‘njelimet’. Disini saya ingin memeriksa nilai dari variabel **foo**. Struktur logikanya seperti ini:

- Apakah **foo** bertipe *number*? Jika iya, tampilkan "foo bertipe number". Kemudian, hanya jika **foo** bertipe *number*, apakah nilainya besar atau sama dengan 10? Jika iya, tampilkan "nilai foo besar dari 10", jika tidak tampilkan "nilai foo kecil dari 10".
- Jika **foo** bukan bertipe *number*, apakah bertipe *string*? Jika iya, tampilkan "foo bertipe *string*".
- Jika **foo** bukan bertipe *number* maupun *string*, maka tampilkan "foo bukan bertipe *number* maupun *string*".

Agar bisa memahami alur kode program diatas, anda harus bisa menentukan awal dan akhir dari setiap block if. Silahkan tukar nilai dari variabel **foo** menjadi 5, "belajar", maupun true, hasil kode program akan berubah tergantung tipe data dari **foo**.

Sebagai alternatif, kode program diatas bisa di konversi menjadi kondisi if “biasa” tanpa menggunakan *nested if*. Idenya, terdapat 4 kondisi yang harus diperiksa:

- Jika **foo** bertipe *number* **dan** kecil dari 10, tampilkan "foo bertipe *number*", dan "nilai foo kecil dari 10".
- Jika **foo** bertipe *number* **dan** besar atau sama dengan 10, tampilkan "foo bertipe *number*", dan "nilai foo besar dari 10".

- Jika `foo` bertipe `string`, tampilkan "foo bertipe string".
- Jika `foo` bukan bertipe `number` maupun `string`, tampilkan "foo bukan bertipe number maupun string".

Silahkan anda coba rancang kode programnya. Disini kita perlu menggunakan operator `&&` untuk menggabungkan 2 kondisi logika.

Baik, berikut kode program yang saya gunakan:

```

1 var foo = "duniailkom";
2 console.log("nilai foo = "+ foo);
3
4 if ((typeof foo === "number") && (foo >= 10)) {
5   console.log("foo bertipe number");
6   console.log("nilai foo besar dari 10");
7 }
8 else if ((typeof foo === "number") && (foo < 10)) {
9   console.log("foo bertipe number");
10  console.log("nilai foo kecil dari 10");
11 }
12 else if (typeof foo === "string") {
13   console.log("foo bertipe string");
14 }
15 else {
16   console.log("foo bukan bertipe number maupun string");
17 }
```



Gambar: Alternatif dari nested if

Perubahan mendasar, saya menggabung kondisi `if ((typeof foo === "number") && (foo >= 10))`. Ini artinya, kedua kondisi ini harus bisa dipenuhi agar block `if` bisa dieksekusi. Jika tidak, akan lanjut ke `else if` berikutnya.

Kemampuan merangkai logika seperti ini sangat penting untuk bisa dipahami. Hampir setiap kode program butuh berbagai kondisi `if` untuk menyelesaikan masalah.

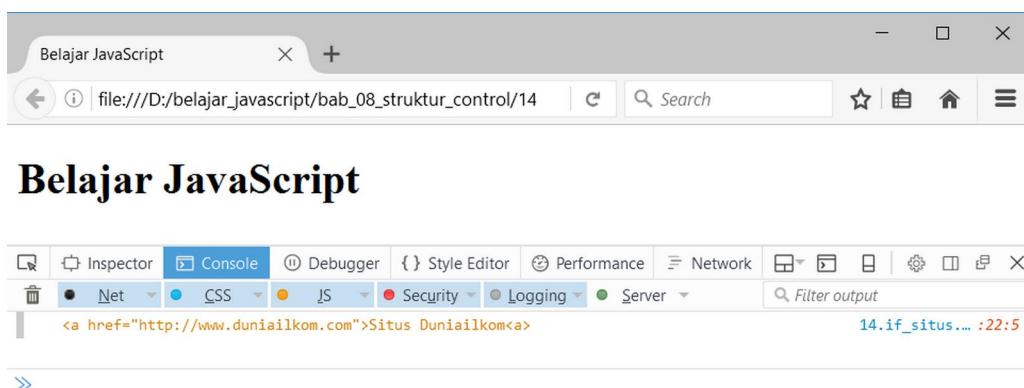
Sebagai latihan, saya ingin membuat kondisi if yang akan menampilkan 4 link ke situs *google*, *facebook*, *twitter* dan *duniaikom*. Jika string berisi "google", tampilkan <a href="http://www.google.com"> Situs Google <a>, jika string berisi "facebook", tampilkan <a href="http://www.facebook.com"> Situs Facebook <a>, dst.

Disini ada 5 kemungkinan, 4 untuk setiap situs, serta 1 alternatif "Situs tidak terdaftar" jika string berisi nilai lain. Silahkan anda coba buat kode programnya.

Baik, berikut kode yang saya gunakan:

```

1 var situs = "duniaikom";
2
3 if (situs === "google") {
4     console.log('<a href="http://www.google.com">Situs Google<a>');
5 }
6 else if (situs === "facebook") {
7     console.log('<a href="http://www.facebook.com">Situs Facebook<a>');
8 }
9 else if (situs === "twitter") {
10    console.log('<a href="http://www.twitter.com">Situs Twitter<a>');
11 }
12 else if (situs === "duniaikom") {
13    console.log('<a href="http://www.duniaikom.com">Situs Duniaikom<a>');
14 }
15 else {
16    console.log("Situs tidak terdaftar");
17 }
```



Gambar: Hasil link situs duniaikom

Disini saya merangkai 4 percabangan if else, masing-masing untuk setiap situs. Perhatikan juga bahwa fungsi `console.log` akan menampilkan hasil dalam bentuk string, bukan link asli. Tag <a> di dalam HTML digunakan untuk membuat link, dan baru berfungsi jika kita menampilkannya ke dalam halaman web. Ini akan kita gunakan saat masuk ke bab tentang **DOM** (*Document Object Model*).

## 8.5 Struktur Logika SWITCH

Struktur logika **switch** kurang lebih mirip seperti **if else**, tapi lebih efisien jika kondisi yang diperiksa ada banyak.

Berikut format penulisan **switch** dalam JavaScript:

```
switch (variable) {  
  
    case (condition1) :  
        statement1;  
        break;  
    case (condition2) :  
        statement2;  
        break;  
    case (condition3) :  
        statement3;  
        break;  
    case (condition4) :  
        statement4;  
        break;  
    default:  
        statement5;  
}
```

Dibaris pertama struktur *switch*, kita menentukan variabel yang akan diperiksa. Pengecekan **kondisi** dari variabel ini dilakukan dalam setiap **case**.

Berikut contoh prakteknya:

```
1 var situs = "twitter";  
2  
3 switch (situs) {  
4     case "google":  
5         console.log('<a href="http://www.google.com">Situs Google<a>');  
6         break;  
7     case "facebook":  
8         console.log('<a href="http://www.facebook.com">Situs Facebook<a>');  
9         break;  
10    case "twitter":  
11        console.log('<a href="http://www.twitter.com">Situs Twitter<a>');  
12        break;  
13    case "duniailkom":  
14        console.log('<a href="http://www.duniailkom.com">Situs DuniaIlkom<a>');  
15        break;  
16    default:
```

```
17     console.log("Situs tidak terdaftar");
18 }
```



Gambar: Hasil tampilan case kode program ‘situs’

Tujuan dari kode program diatas sama persis seperti latihan kita sebelumnya, yakni menampilkan link situs dengan memeriksa isi dari variabel `situs`. Dalam struktur `switch`, kondisi dari variabel `situs` ditulis di dalam `case`.

Jika seluruh `case` telah diperiksa dan tidak ada yang cocok, bagian `default` akan dijalankan. Ini mirip seperti `else` dalam struktur `if else`.

Perintah `break` digunakan untuk keluar dari `switch` begitu kondisi yang cocok sudah ditemukan. Mari kita coba menghapus `break` ini dan jalankan kode programnya:

```
1 var situs = "twitter";
2
3 switch (situs) {
4   case "google":
5     console.log('<a href="http://www.google.com">Situs Google<a>');
6   case "facebook":
7     console.log('<a href="http://www.facebook.com">Situs Facebook<a>');
8   case "twitter":
9     console.log('<a href="http://www.twitter.com">Situs Twitter<a>');
10  case "duniaIlkom":
11    console.log('<a href="http://www.duniaIlkom.com">Situs DuniaIlkom<a>');
12  default:
13    console.log("Situs tidak terdaftar");
14 }
```

Hasilnya adalah:

```
<a href="http://www.twitter.com">Situs Twitter<a>
<a href="http://www.duniailkom.com">Situs DuniaIlkom<a>
Situs tidak terdaftar
```

Ini terjadi karena saat logika *case* cocok dengan "twitter", seluruh perintah setelah *case* ini akan dijalankan, termasuk case "duniailkom", dan *default*.

Jika anda menukar isi string **situs** dengan "google", akan tampil seluruh link, termasuk case *default*. Karena itulah, perintah **break** untuk setiap case sangat penting ditulis. Teks editor seperti **Komodo Edit** akan komplain ketika kita menulis *case* tanpa ditutup perintah **break**.

Fitur lain dari struktur *case* adalah, kita bisa menggabungkan beberapa kondisi, seperti contoh berikut:

```
1  var nilai = 6;
2
3  switch (nilai) {
4      case 1:
5      case 2:
6      case 3:
7      case 4:
8      case 5:
9          console.log('Selama ini ngapain aja bro?');
10         break;
11     case 6:
12     case 7:
13     case 8:
14         console.log('Belajar lebih giat lagi!');
15         break;
16     case 9:
17     case 10:
18         console.log('Pertahankan!');
19         break;
20     default:
21         console.log("Masukkan angka 1 - 10");
22 }
```



Gambar: Hasil kondisi switch untuk kode program “nilai”

Kali ini saya membuat kode program yang memeriksa variabel **nilai**. Variabel nilai bisa berisi angka 1 sampai 10, kemudian menampilkan hasil dari setiap *case*.

Perhatikan cara penulisan *case*. Untuk case 1 sampai 4, saya tidak menulis perintah apapun. Barulah pada case 5 terdapat perintah `console.log('Selama ini ngapain aja bro?')` dan `break`. Artinya, jika variabel angka bernilai 1 - 5, perintah yang dijalankan adalah `console.log('Selama ini ngapain aja bro?')`.

Kode program diatas sama maksudnya jika ditulis seperti ini:

```

1 var nilai = 9;
2
3 switch (nilai) {
4     case 1: console.log('Selama ini ngapain aja bro?'); break;
5     case 2: console.log('Selama ini ngapain aja bro?'); break;
6     case 3: console.log('Selama ini ngapain aja bro?'); break;
7     case 4: console.log('Selama ini ngapain aja bro?'); break;
8     case 5: console.log('Selama ini ngapain aja bro?'); break;
9     case 6: console.log('Belajar lebih giat lagi!'); break;
10    case 7: console.log('Belajar lebih giat lagi!'); break;
11    case 8: console.log('Belajar lebih giat lagi!'); break;
12    case 9: console.log('Pertahankan!'); break;
13    case 10: console.log('Pertahankan!'); break;
14    default: console.log("Masukkan angka 1 - 10");
15 }
```

Saya menulis setiap *case* ke samping, yang tampak lebih rapi. Penulisan seperti ini sering dijumpai jika kode program *case* tidak terlalu panjang.

Secara garis besar, setiap kode program yang dibuat dengan *switch* bisa dikonversi menjadi *if else*. Tapi sebaliknya belum tentu. Struktur *switch* hanya cocok untuk kondisi sederhana seperti mengecek apakah nilai variabel sama dengan string tertentu.

Jika kondisi yang diperiksa sudah gabungan, misalnya menggunakan operator `&&` atau `||`, kita tidak bisa lagi menggunakan `switch`. Ini pula yang menjadi alasan struktur `if` lebih banyak digunakan daripada `switch`.

## 8.6 Operator Conditional

JavaScript memiliki 1 **operator conditional** yang mirip seperti struktur `if else`. Berikut format dasar dari operator conditional ini:

```
result = (condition) ? statement_true : statement_false
```

`result` adalah variabel penampung hasil kondisi. Jika `condition` bernilai *true*, akan dijalankan `statement_true`, yang hasil akhirnya dikirim ke variabel `result`. Jika `condition` bernilai *false*, akan dijalankan `statement_false` yang hasilnya juga dikirim ke variabel `result`.

Agar lebih mudah dipahami, perhatikan kode program berikut:

```
1 var jumlah_barang = 1000;
2 var total;
3
4 if (jumlah_barang > 500) {
5     total = jumlah_barang * 100;
6 }
7 else {
8     total = jumlah_barang * 150;
9 }
10
11 console.log(total); // 100000
```

Saya membuat sebuah percabangan kode program menggunakan `if else`. Jika jumlah barang lebih besar dari 500, harganya 100 per barang. Selain itu, harganya 150 per barang. Prinsip seperti ini sering kita dapatkan untuk aplikasi pemesanan barang, dimana jika barang dipesan dalam jumlah besar, harga per satuatannya bisa lebih murah.

Disini saya mengisi variabel `jumlah_barang` dengan nilai 1000, sehingga variabel `total` akan berisi  $1000 * 100 = 100.000$ .

Jika menggunakan **operator conditional**, kode program diatas bisa diubah menjadi seperti ini:

```

1 var jumlah_barang = 501;
2 var total;
3
4 total = jumlah_barang > 500? jumlah_barang * 100 : jumlah_barang * 150;
5
6 console.log(total); // 100000

```

Penulisannya jauh lebih singkat namun strukturnya sama persis seperti if else sebelumnya. Baris kode program:

```
total = jumlah_barang > 500? jumlah_barang * 100 : jumlah_barang * 150;
```

Bisa dibaca: apakah `jumlah_barang > 500?` Jika iya (*true*), jalankan `jumlah_barang * 100`, hasilnya dikirim ke variabel `total`. Jika tidak (*false*), jalankan `jumlah_barang * 150`, hasilnya dikirim ke variabel `total`.

Sebagai contoh lain, bisakah anda menjelaskan kode program berikut?

```

1 var user = "admin";
2
3 var hak_akses = user === "admin"? true : false ;
4
5 if (hak_akses) {
6   console.log("Selamat datang admin...");
7 }

```

Pertama, saya membuat variabel `user` yang diisi dengan string "admin". Selanjutnya saya membuat *operator conditional*:

```
var hak_akses = user === "admin"? true : false ;
```

Ini bisa dibaca: apakah variabel `user` berisi string "admin"? Jika iya (*true*), kirim nilai *true* ke dalam variabel `hak_akses`. Jika tidak (*false*), kirim nilai *false* ke dalam variabel `hak_akses`.

Sampai disini, variabel `hak_akses` hanya memiliki 2 kemungkinan, apakah *true* atau *false*. Saya bisa menggunakan nilai ini untuk kondisi selanjutnya. Kode `if (hak_akses)` akan dijalankan jika `hak_akses` berisi nilai *true*.

**Operator conditional** mempersingkat penulisan `if else`, namun seperti yang anda lihat, perlu dipahami dengan seksama.

## 8.7 Perulangan FOR

Perulangan atau dalam bahasa Inggris disebut sebagai **loop**, adalah struktur kode program yang digunakan untuk mengulang beberapa baris perintah. Dengan menggunakan perulangan, sangat memudahkan kita menjalankan perintah yang sama secara terus menerus.

Di dalam JavaScript terdapat 3 buah struktur perulangan, yakni **for**, **while**, dan **do while**. Kita akan mulai dari perulangan **for** terlebih dahulu.

Berikut format dasar perulangan **for** dalam JavaScript:

```
for (start; condition; increment) {  
    statement;  
}
```

- **Start** diisi dengan kondisi awal dari perulangan. Biasanya kita mendefenisikan sebuah *variabel counter* yang berfungsi mengontrol perulangan. Sebagai contoh, saya bisa menulis `var i = 1`. Variabel `i` disini berperan sebagai *variabel counter*.
- **Condition** biasanya diisi dengan *operasi perbandingan*, selama operasi ini menghasilkan nilai *true*, perulangan akan dijalankan terus. Sebagai contoh jika saya menulis `i < 10`, artinya perulangan akan terus dijalankan selama nilai variabel `i` kurang dari 10.
- **Increment** diisi dengan instruksi untuk menaikkan nilai *variabel counter*. Dalam setiap perulangan, baris ini akan dieksekusi. Sebagai contoh, saya bisa membuat `i++`, yang artinya nilai variabel `i` akan naik 1 angka dalam setiap perulangan.

Baik, mari kita lihat contoh prakteknya:

```
1 for (var i = 1; i < 10; i++) {  
2     console.log("Hello Indonesia");  
3 }
```

Disini saya membuat perulangan sebanyak **9 kali**. Dalam setiap perulangan ini jalankan kode program `console.log("Hello Indonesia")`.

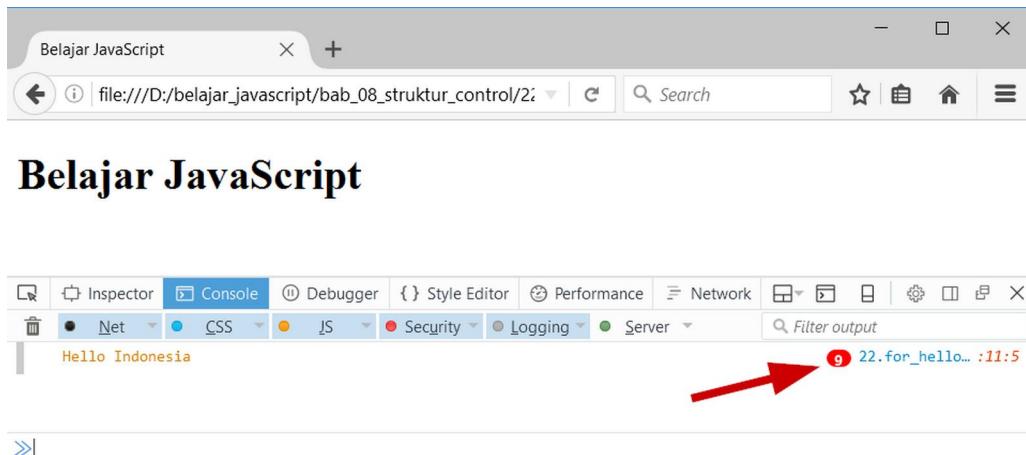
*Darimana datangnya 9 kali perulangan?*

Pertama, perhatikan bagian **start**, disini saya membuat `var i = 1`. Variabel `i` berperan sebagai *variabel counter*. Inilah yang mengontrol berapa banyak perulangan yang akan dijalankan. Karena saya menulis `var i = 1`, artinya perulangan mulai dari 1.

Bagian kedua, terdapat kondisi `i < 10`, artinya selama nilai *variabel counter* `i` kurang dari 10, jalankan perulangan.

Bagian ketiga terdapat `i++`, yang artinya dalam setiap perulangan nilai variabel `i` akan naik sebanyak 1 angka. Ingat, `i++` sama artinya dengan `i = i + 1`.

Dengan demikian, nilai variabel `i` akan mulai dari 1, kemudian 2, 3, 4, 5, 6, 7, 8 dan 9. Total terdapat 9 kali perulangan. Berikut tampilannya di tab console:



Gambar: Hasil perulangan for

Kenapa hanya 1 baris? Ternyata tab console mengelompokkan hasil perulangan ini dan hanya menampilkan angka 9 di sudut kiri string "Hello Indonesia". Terlihat angka 9 yang mendudukkan baris ini seharusnya ditampilkan sebanyak 9 kali.

Tapi tunggu dulu, bukankah di kode program tertulis `i < 10`? Kenapa hanya sampai 9? Bukan 10?

Operator perbandingan yang saya gunakan adalah *kurang dari* (`<`), artinya ketika *variabel counter* naik menjadi 10, kondisi `10 < 10` akan menghasilkan *false*. Jika saya ingin membuat perulangan sebanyak 10 kali, kondisi ini bisa diganti menjadi `i <= 10`:

```
1 for (var i = 1; i <= 10; i++) {
2   console.log("Hello Indonesia");
3 }
```

Sekarang, baris "Hello Indonesia" akan tampil sebanyak 10 kali. Memahami perbedaan antara kondisi `i < 10` dengan `i <= 10` terkesan sepele, tapi sangat mempengaruhi hasil akhir perulangan.

Variabel counter `i` juga bisa kita akses di dalam perulangan, seperti contoh berikut:

```
1 for (var i = 1; i <= 10; i++) {
2   console.log("Hello Indonesia " + i);
3 }
```

Kali ini *block* perulangan berisi `console.log("Hello Indonesia " + i)`, artinya variabel counter `i` akan ditambahkan ke dalam akhir string, dengan tampilan akhir sebagai berikut:

The screenshot shows a browser window titled "Belajar JavaScript". Below the title bar is a toolbar with various developer tools: Inspector, Console, Debugger, Style Editor, Performance, Network, and others. The "Console" tab is selected. In the main content area, there is a list of log entries. Each entry consists of a timestamp and a message. The messages are: "Hello Indonesia 1", "Hello Indonesia 2", "Hello Indonesia 3", "Hello Indonesia 4", "Hello Indonesia 5", "Hello Indonesia 6", "Hello Indonesia 7", "Hello Indonesia 8", "Hello Indonesia 9", and "Hello Indonesia 10". To the right of each message is a timestamp: "24.for\_hello... :11:5".

Gambar: Tampilan variabel counter i yang menaik dari 1 hingga 10

Terlihat di akhir string "Hello Indonesia " telah ditambahkan dengan nilai variabel i yang menaik dari 1 hingga 10.

Bagaimana jika saya ingin variabel i ini ditulis menurun, yakni dari 10, 9, 8, dst hingga 1?

Caranya adalah dengan mengubah nilai start awal perulangan, kondisi, dan counter, seperti contoh berikut:

```
1 for (var i = 10; i >= 1; i--) {
2   console.log("Hello Indonesia " + i);
3 }
```

The screenshot shows a browser window titled "Belajar JavaScript". Below the title bar is a toolbar with various developer tools: Inspector, Console, Debugger, Style Editor, Performance, Network, and others. The "Console" tab is selected. In the main content area, there is a list of log entries. Each entry consists of a timestamp and a message. The messages are: "Hello Indonesia 10", "Hello Indonesia 9", "Hello Indonesia 8", "Hello Indonesia 7", "Hello Indonesia 6", "Hello Indonesia 5", "Hello Indonesia 4", "Hello Indonesia 3", "Hello Indonesia 2", and "Hello Indonesia 1". To the right of each message is a timestamp: "25.for\_hello... :11:5".

Gambar: Perulangan i menurun, dari 10 ke 1

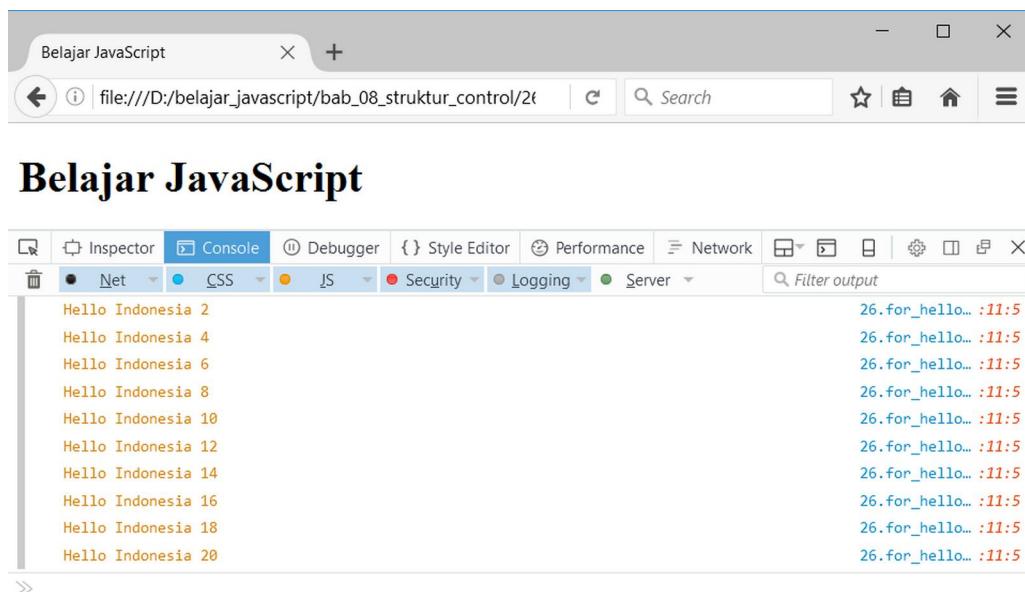
Kali ini, saya mulai dari `var i = 10`, dengan kondisi `i >= 1` dan counter `i--`. Artinya, perulangan mulai dari 10, dalam setiap perulangan nilai `i` diturunkan sebanyak 1 angka, selama nilai `i` lebih besar atau sama dengan 1.

Selanjutnya, bagaimana membuat perulangan yang ‘lompat’? Misalnya nilai `i` dari 2 menjadi 4, kemudian 8, 10, dan seterusnya dengan kenaikan 2? Caranya adalah dengan mengubah baris `counter`.

Selama ini saya menggunakan `counter i++` atau `i--`, ini sama dengan `i = i + 1` dan `i = i - 1`. Bagaimana jika saya ubah menjadi `i = i + 2`? Berikut prakteknya:

```
1 for (var i = 2; i <= 20; i = i + 2) {
2   console.log("Hello Indonesia " + i);
3 }
```

Baris perintah perulangan `for (var i = 2; i <= 20; i = i + 2)`, bisa dibaca: Mulai dari `i = 2`, kemudian naikkan nilai `i` sebanyak 2 angka, selama kurang atau sama dengan 20. Berikut hasilnya:



Gambar: Perulangan dengan nilai `i` kelipatan 2

Seperti yang terlihat, nilai `i` naik mulai dari 2, 4, 8, dst hingga 20.

Sebagai latihan, bisakah anda membuat perulangan `for` yang hasilnya berupa perkalian:

```
100 * 5 = 500
95 * 5 = 475
90 * 5 = 450
...
0 * 5 = 5
```

Disini kita hitung mundur dari 100 ke 1, dengan kelipatan 5.

Berikut kode program yang saya gunakan:

```

1 for (var i = 100; i >= 0; i = i - 5) {
2   console.log( i + " * 5 = " + i*5);
3 }
```

Saya sarankan anda untuk mencoba latihan berbagai soal lain, agar konsep perulangan for ini bisa lebih dipahami.

Dalam contoh kode program yang kita gunakan sejauh ini, saya terus membuat lengkap 3 bagian dari perulangan for, yakni **start**, **condition**, dan **increment**. Sebenarnya, bagian ini boleh tidak ditulis, dengan catatan kita memindahkannya ke tempat lain.

Sebagai contoh, perhatikan kode berikut:

```

1 var i = 1;
2
3 for ( ; i <= 10; i++) {
4   console.log("Hello JavaScript " + i);
5 }
```

Disini saya membuat kondisi **start** diluar struktur **for**. Jika ditulis seperti ini, kita tetap harus menyisakan tanda titik koma di posisi **start** dari perulangan for.

Bahkan bagian **counter** juga bisa dipindahkan ke dalam perulangan:

```

1 var i = 1;
2
3 for ( ; i <= 10; ) {
4   console.log("Hello JavaScript " + i);
5   i++;
6 }
```

Penulisan seperti ini boleh dan valid, tapi memang jarang digunakan. Untuk yang tidak memahami kode diatas, bisa saja mengatakan bahwa ini salah, padahal memang dibolehkan di dalam JavaScript.

## 8.8 Perulangan Bersarang (Nested Loop)

**Perulangan bersarang** atau dalam bahasa inggris dikenal dengan *nested loop* merupakan sebuahan untuk *perulangan di dalam perulangan*. Struktur seperti ini diperlukan untuk kode program yang cukup kompleks. Karena kompleksitasnya inilah sering soal-soal latihan programming melibatkan *nested loop*.

Berikut format dasar dari perulangan bersarang dalam JavaScript:

```
for (start1; condition1; increment1) {  
    statement1;  
    for (start2; condition2; increment2) {  
        statement2;  
    }  
}
```

Tampak ada dua buah perulangan, perulangan luar (*outer loop*) dan perulangan dalam (*inner loop*). Setiap kali perulangan luar naik 1 angka, perulangan dalam akan dijalankan seluruhnya.

Misalkan perulangan luar saya jalankan sebanyak 5 kali dan perulangan dalam 3 kali. Secara total, baris *statement2* akan dijalankan sebanyak 15 kali, didapat dari  $5 * 3$ .

Mari kita masuk ke dalam contoh program:

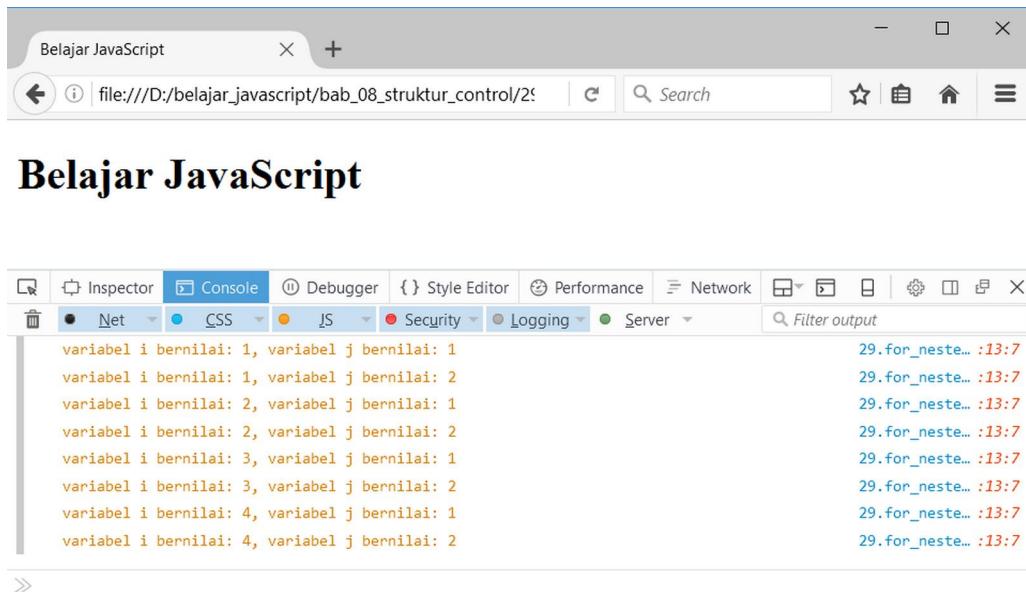
```
1 var i,j;  
2 for (i = 1; i <= 4; i++) {  
3     for (j = 1; j <= 2; j++) {  
4         console.log("variabel i bernilai: "+ i +", variabel j bernilai: "+j);  
5     }  
6 }
```

Karena kita punya 2 buah perulangan, saya juga harus menggunakan 2 buah variabel counter, yakni *i* dan *j*.

Perulangan luar dijalankan dari *i* = 1 sampai dengan *i*  $\leq$  4, artinya akan terjadi 4 kali perulangan. Untuk perulangan dalam dijalankan dari *j* = 1, hingga *j*  $\leq$  2, yang artinya terdapat 2 kali perulangan.

Untuk setiap perulangan *i*, perulangan *j* akan dijalankan 2 kali, sehingga total baris *console.log()* akan dijalankan sebanyak  $4 * 2 = 8$  kali.

Berikut tampilannya:



Gambar: Tampilan hasil perulangan bersarang (nested loop)

Agar bisa memahami apa yang terjadi, silahkan anda pelajari dari hasil tersebut. Nested loop ini relatif jarang digunakan, tapi bukan tidak mungkin anda akan menemui masalah yang hanya bisa diselesaikan menggunakan nested loop.

## 8.9 Infinity Loop

**Infinity loop** adalah perulangan yang berjalan terus menerus dan tidak akan pernah selesai. Ini biasanya terjadi karena kesalahan dari kita, programmer yang membuat kode program.

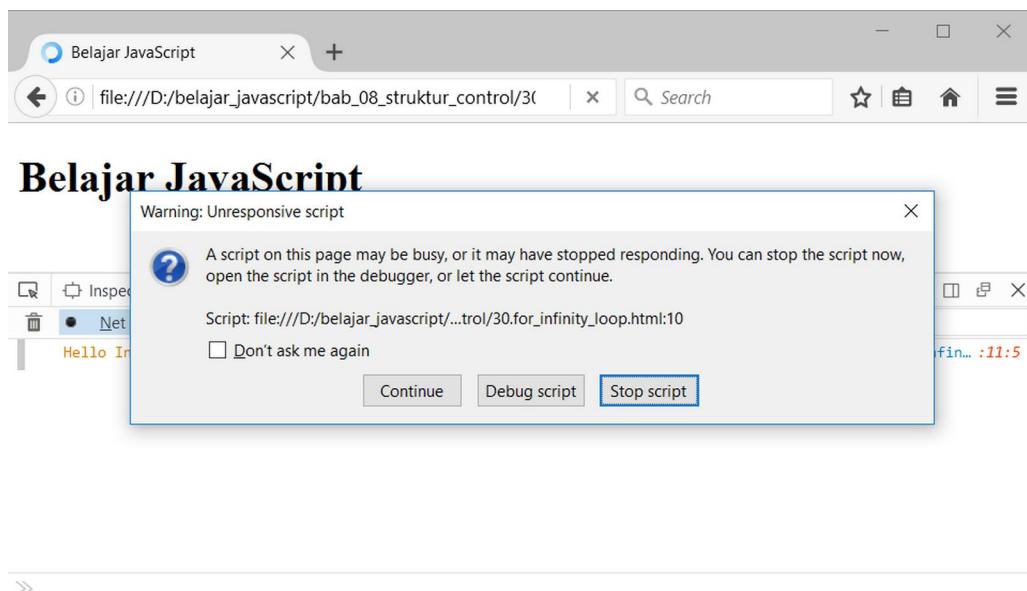
Sebagai contoh, perhatikan kode berikut, dapatkah anda menebak letak kesalahannya?

```
1 for (var i = 20; i > 10; i++) {  
2     console.log("Hello Indonesia");  
3 }
```

Disini saya memiliki perulangan **for** yang dimulai dari  $i = 20$ , dengan kenaikan  $i++$ . Perulangan akan berhenti saat  $i > 10$ . Lalu dimana letak salahnya?

Perhatikan kondisi akhir  $i > 10$  dan nilai awal  $i = 20$ . Artinya, kondisi  $i > 10$  akan selalu benar, sedangkan nilai  $i$  tidak akan pernah dibawah 10, karena awalnya saja sudah 20 dan *increment* malah ditambah lagi  $i++$ .

Jika anda menjalankan kode diatas, web browser bisa menjadi *not responding*. Kode tersebut akan memaksa processor terus bekerja tanpa tau kapan selesai. Untungnya, web browser modern akan menampilkan jendela pemberitahuan bahwa ada sesuatu yang salah dari kode JavaScript tersebut, dan kita diberikan pilihan untuk menghentikannya secara paksa.



Gambar: Web browser Mozilla Firefox menginfokan ada sesuatu yang ‘salah’

Dengan men-klik tombol “Stop Script”, kode JavaScript akan dipaksa berhenti mengeksekusi *infinity loop*.

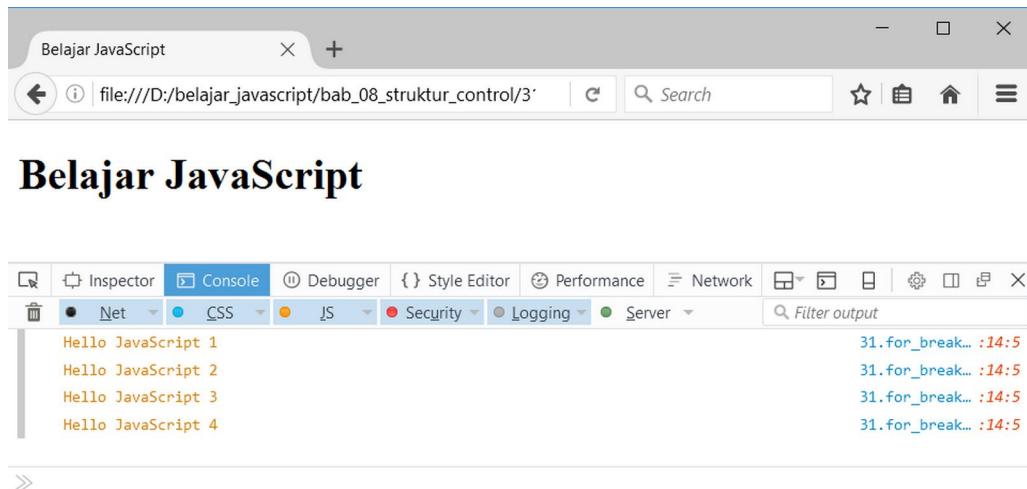
## 8.10 Perintah Break dan Continue

Perintah **break** dan **continue** digunakan untuk menghentikan sebuah perulangan sebelum waktunya, atau dengan kata lain perulangan berhenti secara *prematur*.

Berikut contoh penggunaan perintah **break** untuk menghentikan perulangan for:

```
1 for (var i = 1; i <= 10; i++) {  
2     if (i === 5) {  
3         break;  
4     }  
5     console.log("Hello JavaScript " + i);  
6 }
```

Program diatas akan membuat perulangan dari  $i = 1$  sampai  $i = 10$ , dimana seharusnya kana tampil string "Hello JavaScript" sebanyak 10 kali. Namun di dalam perulangan saya membuat sebuah kondisi if. Ketika variabel  $i$  berisi nilai 5, jalankan perintah **break**. Bagaimana hasilnya?



Gambar: Hasil perulangan dengan perintah break pada i === 5

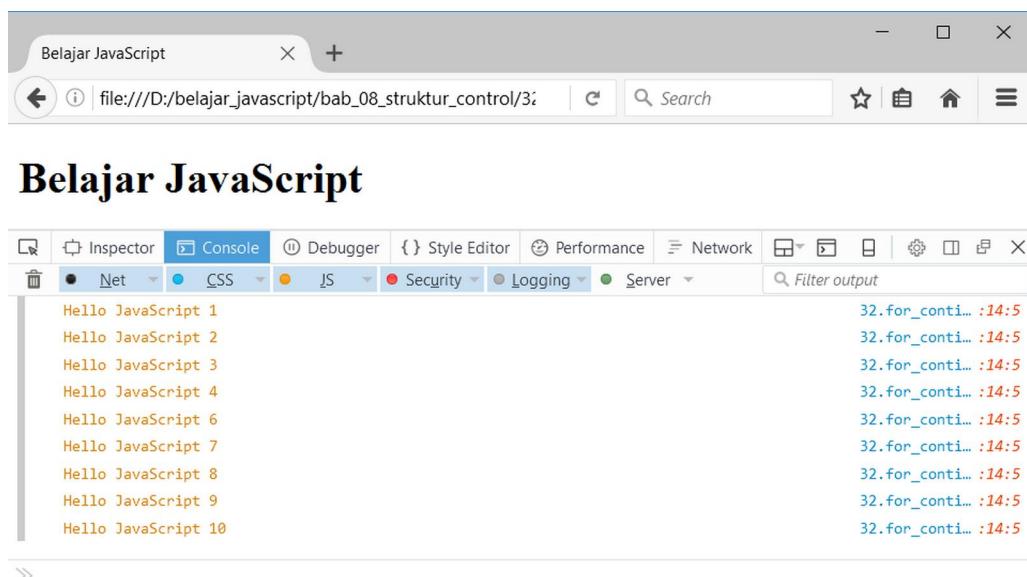
Perulangan kita berhenti ketika kondisi `if (i === 5)` dipenuhi, yakni pada saat perulangan ke lima.

Bagaimana jika diganti dengan perintah **continue**?

```

1 for (var i = 1; i <= 10; i++) {
2   if (i === 5) {
3     continue;
4   }
5   console.log("Hello JavaScript " + i);
6 }

```



Gambar: Hasil perulangan dengan perintah continue pada i === 5

Kelihatannya perulangan tetap dijalankan hingga 10. Tetapi perhatikan perulangan ke 5. Pada saat *variabel counter* `i` berisi angka 5, perintah **continue** akan dijalankan.

Perintah ini memaksa perulangan untuk berhenti memproses perulangan saat ini, dan langsung lanjut ke perulangan berikutnya, yakni dimana nilai i berisi angka 6. Perintah continue bisa digunakan untuk melewati 1 kali perulangan.

## 8.11 Perulangan WHILE

Perulangan kedua yang akan kita bahas adalah **while**. Perulangan ini mirip seperti perulangan **for**, tapi dengan memecah bagian **start**, **condition**, dan **increment**.

Berikut format dasar perulangan **while** JavaScript:

```
start;
while (condition) {
    statement;
    increment;
}
```

Sekarang, bagian awal **start** berada sebelum *loop*, serta bagian **increment** berada di dalam *loop*.

Mari kita lihat contohnya:

```
1 var i = 1;
2 while (i <= 10){
3     console.log("Hello Indonesia");
4     i++;
5 }
```

Disini saya membuat perulangan **while** sebanyak 10 kali, dimulai dari  $i = 1$ , sampai dengan  $i \leq 10$ . Setiap perulangan di-*increment* sebanyak 1 angka ( $i++$ ). Hasilnya teks "Hello Indonesia" tampil sebanyak 10 kali.

Jika anda sudah memahami struktur perulangan **for**, saya rasa tidak terlalu sulit memahami perulangan **while** ini.

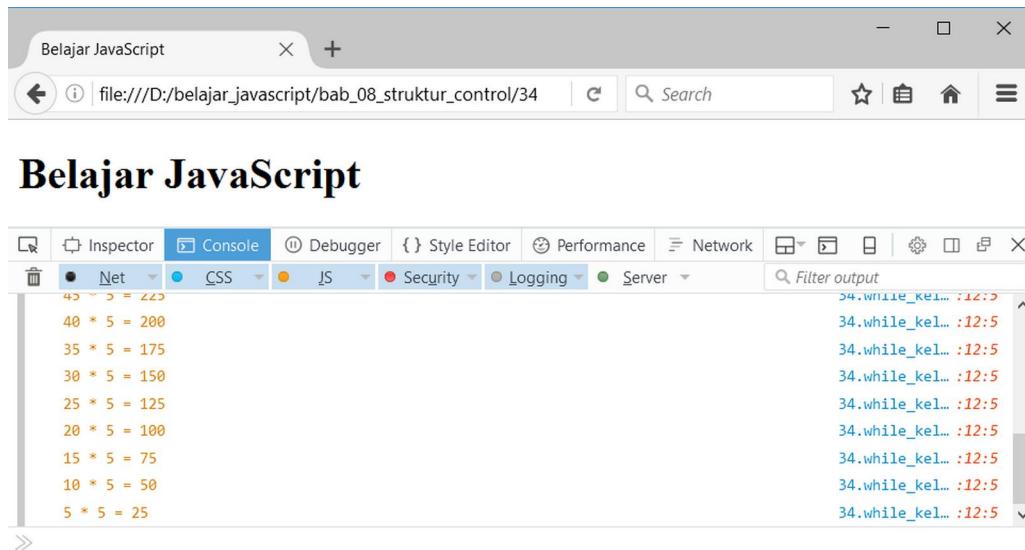
Baik, bagaimana dengan kode berikut?

```
1 var i = 100;
2 while (i > 0){
3     console.log( i + " * 5 = " + i*5);
4     i = i - 5;
5 }
```

Kali ini perulangan dimulai dari  $i = 100$ , hingga  $i > 0$ . Sampai disini kita bisa menebak kalau ini merupakan perulangan menurun. Di akhir *block* **while** terlihat decrement  $i = i - 5$ , yang berarti looping ini akan menurun dari 100 dengan tiap penurunan sebanyak 5 angka. Variabel i akan turun dari 100, ke 95, 90, 85, dst hingga 5.

Kenapa 5? Bukannya hingga 0? Lihat kembali kondisi akhir perulangan ini:  $i > 0$ , artinya variabel i tidak akan pernah mencapai nol. Beda jika saya ubah menjadi  $i \geq 0$ .

Kemudian apa yang dijalankan dalam setiap perulangan? `console.log( i + " * 5 = " + i*5)`. Bisa anda lihat bahwa saya ingin membuat sebuah perkalian menurun, dimana nilai i dikali 5. Berikut hasil akhirnya:



Gambar: Perulangan while dengan perkalian lima

Perintah **break** dan **continue** juga tetap berlaku di dalam while:

```

1 var i = 1;
2
3 while (i <= 10){
4     if (i === 5) {
5         break;
6     }
7     console.log("Hello JavaScript " + i);
8     i = i + 1;
9 }

```

Dalam program ini perulangan akan berhenti saat *variabel counter* i berisi angka 5.

Pada prinsipnya, perulangan **while** cocok digunakan untuk situasi dimana kita tidak tau berapa banyak perulangan yang mesti dijalankan. Konsepnya seperti ini:

```
1 foo = true;
2 while (foo) {
3     statement;
4     if (condition) {
5         foo = false;
6     }
7 }
```

Disini saya memiliki variabel `foo` yang berfungsi sebagai penentu perulangan. Variabel `foo` ini biasa disebut sebagai **flag** (bahasa inggris: bendera), karena jalan atau tidaknya perulangan bergantung kepada `foo` apakah berisi *true* atau *false*.

Pada awal kode program, `foo` diisi *true*. Perulangan langsung berjalan karena saya tidak membuat kondisi apapun, tapi langsung menulis `while (foo)`. Sekilas tampak akan terjadi *infinity loop*, karena kondisi `while (true)` akan selalu terpenuhi.

Akan tetapi dalam kode program terdapat kondisi **if**:

```
1 if (condition) {
2     foo = false;
3 }
```

Jika **condition** untuk **if** ini terpenuhi, kita ubah nilai **flag** `foo` menjadi *false*, sehingga perulangan akan berhenti.

Saat ini kita belum bisa mencoba prakteknya, karena butuh materi lanjutan. Tapi anda bisa bayangkan perulangan seperti ini untuk proses menampilkan data yang digenerate secara dinamis.

Misalnya kita membuat program yang mengambil suatu data dari situs facebook (menggunakan **API - Application Programming Interface**). Kita tidak tau berapa banyak baris tabel yang ingin diambil. Dalam situasi seperti ini, perulangan **while** bisa digunakan, dan tidak bisa menggunakan perulangan **for**.

Secara umum, setiap perulangan **for** bisa dikonversi menjadi perulangan **while**, tapi tidak sebaliknya. Sehingga perulangan **while** lebih fleksibel daripada **for**.

Tapi perulangan **for** terkesan lebih rapi dan mudah dibaca, ini karena bagian **start**, **condition**, dan **increment** terletak di satu baris. Jika anda sudah bisa menentukan berapa banyak perulangan yang harus dilakukan, perulangan **for** lebih mudah ditulis daripada **while**.

## 8.12 Perulangan DO WHILE

Perulangan **do while** sangat mirip seperti perulangan **while**. Bedanya, dalam perulangan **do while**, kondisi di cek diakhir block perulangan.

Berikut format dasar perulangan **do while** JavaScript:

```

start;
do {
    statement;
    increment;
} while (condition)

```

Terlihat pengecekan **condition** dilakukan di akhir. Mari kita lihat contoh penggunaannya:

```

1 var i = 1;
2 do {
3     console.log("Hello Indonesia");
4     i++;
5 } while (i <= 10);

```

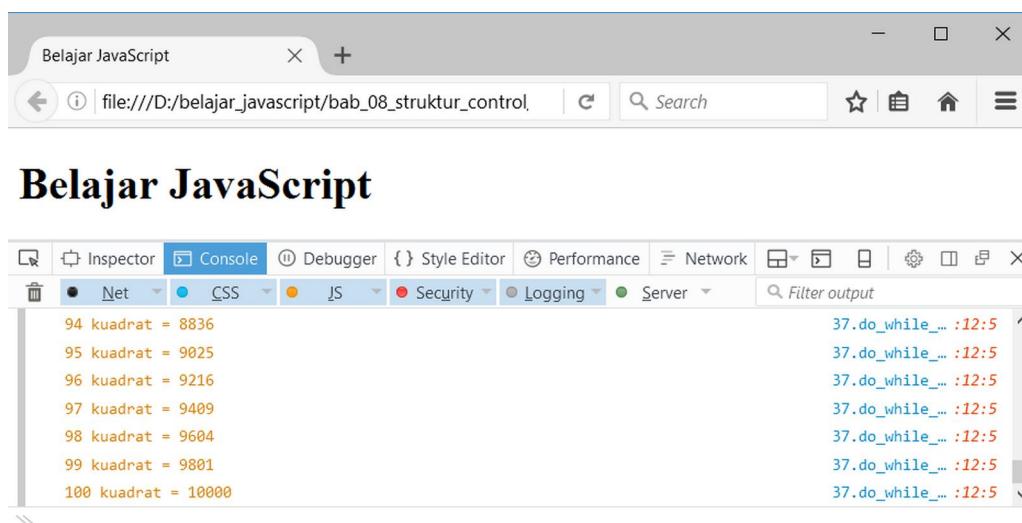
Disini saya ingin menampilkan teks "Hello Indonesia" sebanyak 10 kali. Mulai dari  $i = 1$ , hingga  $i \leq 10$ , dengan increment  $i++$ . Tidak masalah.

Berikut contoh kedua:

```

1 var i = 1;
2 do {
3     console.log(i + " kuadrat = " + i*i);
4     i = i + 1;
5 } while (i <= 100);

```



Gambar: Perkalian kuadrat menggunakan do while

Kode program diatas akan menampilkan perkalian kuadrat dari setiap angka mulai dari 1 hingga 100. Juga tidak ada hal yang baru disini, contoh kita mirip seperti perulangan **while**, dengan memindahkan bagian kondisi di akhir perulangan.

Konsekuensi dari pengecekan yang dilakukan di akhir adalah, block perulangan setidaknya akan diproses 1 kali, walaupun kondisi tersebut sudah tidak dipenuhi sejak awal.

Berikut contoh kasusnya:

```

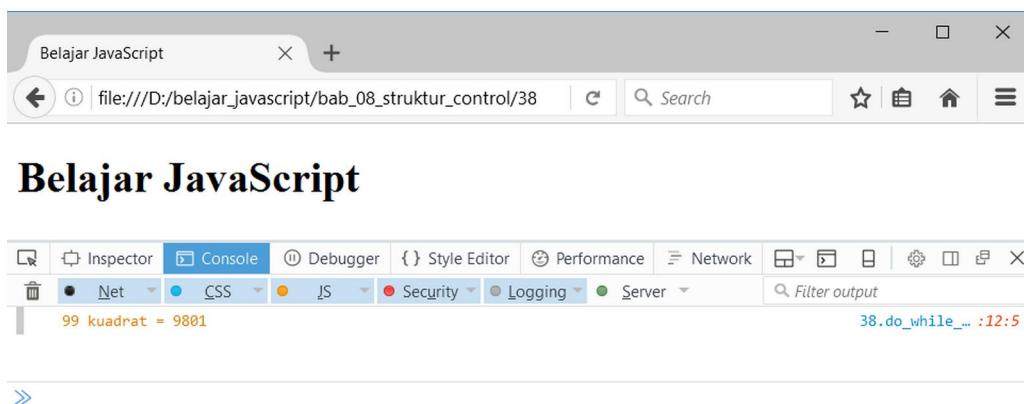
1 var i = 99;
2 do {
3     console.log(i + " kuadrat = " + i*i);
4     i = i + 1;
5 } while (i <= 10);

```

Kode ini merupakan modifikasi dari contoh sebelumnya. Kondisi awal dimulai dari `var i = 99`, manampilkan nilai `i * i`, increment dengan `i = i + 1`. Tidak ada masalah.

Tapi perhatikan kondisi `while (i <= 10)`, ini artinya perulangan dijalankan selama nilai variabel `i` kurang atau sama dengan 10. Padahal saya membuat kondisi awal `i = 99`. Bisa disimpulkan bahwa kondisi perulangan ini sudah tidak dipenuhi sejak awal, `99 <= 10` menghasilkan `false`.

Mari kita jalankan:



Gambar: Hasil dari perulangan `do while`, tetap ditampilkan walaupun kondisi awal tidak dipenuhi

Terlihat tampak baris `99 kuadrat = 9801`. Karena pengecekan kondisi dilakukan diakhir, block `do while` tetap dijalankan minimal 1 kali.

Dalam praktiknya, saya jarang menemukan aplikasi dari `do while`. Kita akan banyak menggunakan perulangan `for` dan `while`.

## 8.13 Menampilkan Element Array Dengan Perulangan

Salah satu penggunaan terbanyak dari perulangan adalah untuk menampilkan element array. Mari kita lihat cara penggunaannya:

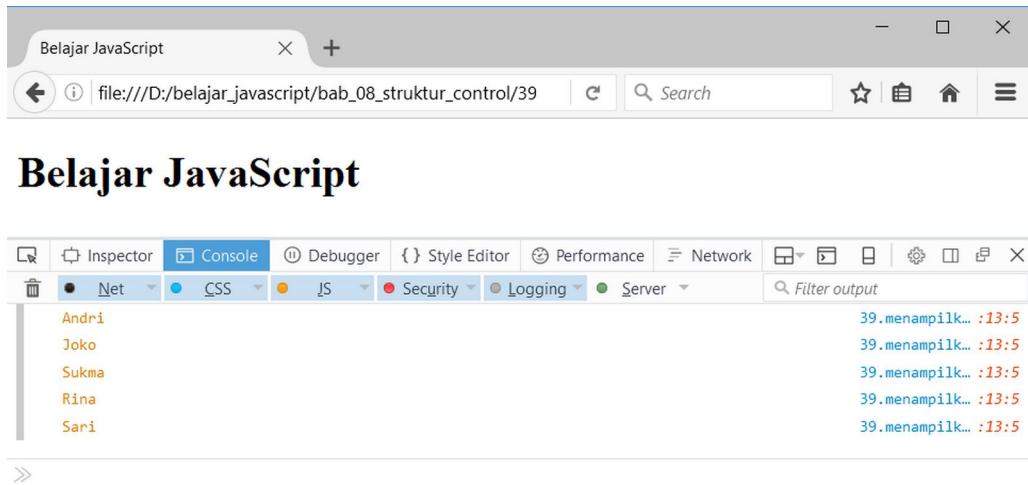
```

1 var siswa = ["Andri", "Joko", "Sukma", "Rina", "Sari"];
2
3 for (var i = 0; i <= 4; i++){
4     console.log(siswa[i]);
5 }

```

Saya memiliki array `siswa` dengan 5 element. Untuk menampilkan seluruh element array, saya bisa menggunakan perulangan `for`, dimana nilai awal mulai dari 0 (ingat, index element array mulai dari 0, bukan 1). Perulangan terus dilakukan hingga `i <= 4`, yakni sampai index ke 4.

Perintah `console.log(siswa[i])` akan menampilkan element array mulai dari `siswa[0], siswa[1]`, hingga `siswa [4]`.



Gambar: Seluruh element array siswa ditampilkan

Kekurangan dari cara ini adalah, kita harus menulis manual jumlah element dalam array. Saya membuat kondisi akhir perulangan `i <= 4`, karena telah menghitung isi dari array `siswa` yang memiliki 5 element (dimana index terakhir adalah 4). Akan jauh lebih fleksibel jika kita bisa mendapatkan jumlah element array ini secara otomatis.

Untungnya, di dalam JavaScript setiap array memiliki *property length* yang berisi total jumlah element dari suatu array. Pembahasan tentang apa itu *property* akan kita bahas dalam bab tersendiri. Untuk saat ini, anda bisa menganggap bahwa *property* ini sebuah variabel khusus yang sudah berisi nilai.

Untuk mengambil jumlah element dari array `siswa`, saya bisa menulis seperti ini:

```
var jumlah_siswa = siswa.length;
```

Variabel `jumlah_siswa` otomatis berisi angka 5, yakni sebanyak total element dalam array tersebut.

Baik, mari kita tulis ulang kode sebelumnya:

```

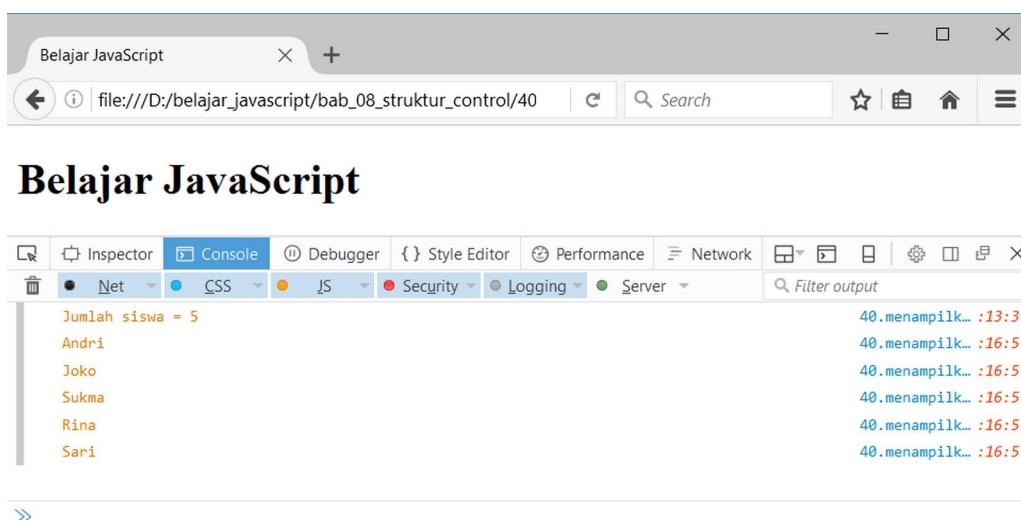
1 var siswa = ["Andri", "Joko", "Sukma", "Rina", "Sari"];
2 var jumlah_siswa = siswa.length;
3
4 console.log ("Jumlah siswa = " + jumlah_siswa);
5
6 for (var i = 0; i < jumlah_siswa; i++){
7   console.log(siswa[i]);
8 }

```

Di baris kedua, saya mengambil jumlah element array siswa menggunakan perintah `siswa.length`. Dengan kode ini, tidak peduli berapa banyak element array siswa, kita bisa mendapatkannya dengan mudah. Variabel `jumlah_siswa` ini bisa digunakan untuk menentukan kondisi akhir perulangan, yakni hingga `i < jumlah_siswa`.

Saya menggunakan tanda “`<`” karena index terakhir dari element array adalah jumlah element - 1. Sebagai contoh, variabel `jumlah_siswa` berisi angka 5, karena di dalam array siswa terdapat 5 element. Namun index terakhir dari element array siswa ini adalah 4, bukan 5. Karena penomoran index mulai dari 0, bukan 1.

Berikut hasilnya:



Gambar: Tampilan seluruh element dari array siswa

Konsep seperti ini sangat sangat sering kita gunakan, terutama di dalam JavaScript. Struktur DOM yang nantinya kita pelajari, semuanya berupa array (atau ‘mirip array’). Untuk menampilkannya, bisa menggunakan perulangan array seperti ini.

## 8.14 Perulangan FOR OF

Masih berkaitan dengan array, dalam ECMAScript 6, terdapat sebuah perulangan baru, yakni perulangan `for of` yang digunakan khusus digunakan untuk menampilkan element array. Jika anda sudah pernah belajar PHP, perulangan `for of` ini mirip seperti perulangan `foreach` PHP.

Berikut contoh penggunaannya:

```
1 var siswa = ["Andri", "Joko", "Sukma", "Rina", "Sari"];
2
3 for (var i of siswa){
4     console.log(i);
5 }
```

Kita cukup membuat sebuah variabel yang akan menampung element array, dalam contoh diatas saya menggunakan variabel `i`. Seperti yang terlihat, penulisannya sangat singkat dan kita tidak perlu mencari tau berapa jumlah element array. Perulangan `for of` otomatis akan berhenti setelah seluruh element array selesai diproses.

Tapi berbeda dengan perulangan `foreach` PHP, disini kita tidak memiliki cara untuk mengakses nilai index dari array tersebut. Jika anda butuh mengakses index array, terpaksa menggunakan perulangan `for` biasa.

---

Dalam bab ini kita telah membahas tentang struktur kondisi `if`, `else`, `if else`, `switch` serta perulangan `for`, `while`, dan `do while`. Semuanya merupakan materi penting yang wajib dipahami. Pada bab selanjutnya, kita akan masuk ke dalam `function`.

# 9. Function

Untuk membuat sebuah program besar, sebaiknya dipecah menjadi sub program yang lebih kecil, lalu secara bersama-sama membangun aplikasi akhir. Dalam pemrograman JavaScript, sub program ini dikenal dengan istilah **function**.

Function di dalam JavaScript juga cukup unik, karena bisa disimpan ke dalam variabel, atau dijadikan sebagai *argument*. Istilah programmingnya, function di JavaScript disebut adalah sebagai **First-class citizen**.

Selain itu setiap variabel yang ada di dalam function memiliki *scope* atau ruang lingkup yang terbatas. Dalam bab ini kita akan membahas tentang *function* JavaScript, serta bagaimana cara penggunaannya.

## 9.1 Pengertian Function

**Function** atau dalam bahasa indonesia disebut sebagai **fungsi**, adalah kumpulan kode program yang dirancang untuk menyelesaikan sebuah tugas tertentu, dan merupakan bagian dari program utama. *Function* diperlukan untuk memecah alur program yang besar menjadi beberapa program kecil agar mudah di kelola.

Setiap *function* punya tugas dan fungsi masing-masing. Bisa saja di dalam sebuah function terdapat function lain, tergantung kompleksitas masalah yang ingin dipecahkan.

JavaScript sendiri memiliki ratusan function bawaan yang bisa digunakan, seperti fungsi `alert()`. Selain itu, kita juga bisa membuat fungsi sendiri.



Dalam bab ini saya akan fokus kepada cara membuat fungsi sendiri (*user defined function*). Fungsi-fungsi bawaan JavaScript akan dibahas di bab tersendiri.

Berikut format dasar pembuatan *function* di dalam JavaScript:

```
function function_name (argument1, argument2, ...) {  
    statement;  
    statement;  
    return value;  
}
```

Setelah *keyword* `function`, diikuti dengan *function\_name*. **Function\_name** ini merupakan nama dari fungsi tersebut. Penulisan nama fungsi mengikuti aturan dari **identifier**, atau dengan kata lain mengikuti aturan pembuatan *variabel*, yakni tidak boleh diawali angka, tidak boleh mengandung spasi, dst.

Setiap function bisa memiliki 1 atau beberapa *argument*. **Argument** adalah variabel yang berfungsi sebagai nilai *input* ke dalam function. **Argument** ditulis di dalam tanda kurung setelah `function_name`.

**Block function** ditandai dengan tanda kurung kurawal. Di sinilah kode program JavaScript yang membentuk function ditulis. Perintah **return** digunakan sebagai *output* function. Penulisan *argument* dan perintah *return* bersifat opsional dan boleh tidak ditulis.

Jika anda belum pernah belajar bahasa pemrograman, penjelasan saya diatas mungkin susah untuk dipahami. Tapi jangan khawatir, saya akan jelaskan langkah per langkah menggunakan contoh kode program.

## 9.2 Membuat dan Memanggil Function

Sebagai praktek pertama, saya akan merancang function `pagi()` yang menampilkan kata "Selamat Pagi" dalam 4 bahasa:

```
1 function pagi(){
2   console.log("Selamat Pagi");
3   console.log("Good Morning");
4   console.log("Ohayou Gozaimasu");
5   console.log("Buenos Dias");
6 }
```

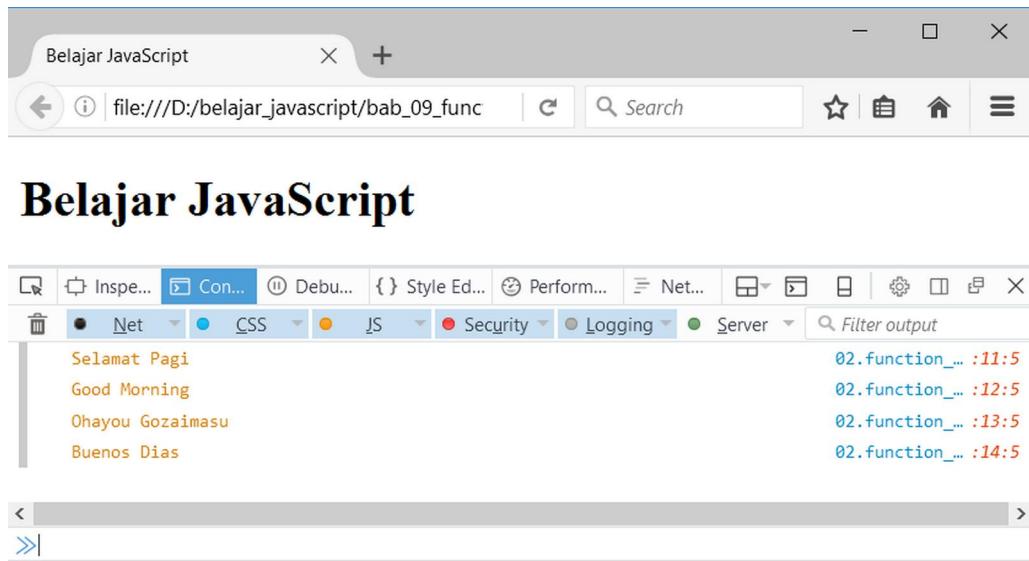
Fungsi ini terdiri dari 4 baris kode program JavaScript, didalamnya saya menampilkan 4 buah string menggunakan perintah `console.log()`.

Fungsi diatas tidak memiliki *argument*, namun tetap harus ditulis dengan tanda kurung kosong: `pagi()`. Inilah yang membedakan sebuah fungsi dengan *identifier* lain seperti variabel. Jika anda menemukan saya menulis sebuah kata diikuti dengan tanda kurung kosong, artinya itu sebuah *function*.

Menulis fungsi seperti ini disebut juga dengan **function declaration** atau pendeklarasian fungsi.

Jika anda jalankan kode program diatas, tidak akan tampil apa-apa. Karena kita harus "memanggil" fungsi ini terlebih dahulu. Bagaimana caranya? Cukup dengan menulis nama fungsi tersebut, seperti contoh berikut:

```
1 function pagi(){
2   console.log("Selamat Pagi");
3   console.log("Good Morning");
4   console.log("Ohayou Gozaimasu");
5   console.log("Buenos Dias");
6 }
7
8 pagi();
```



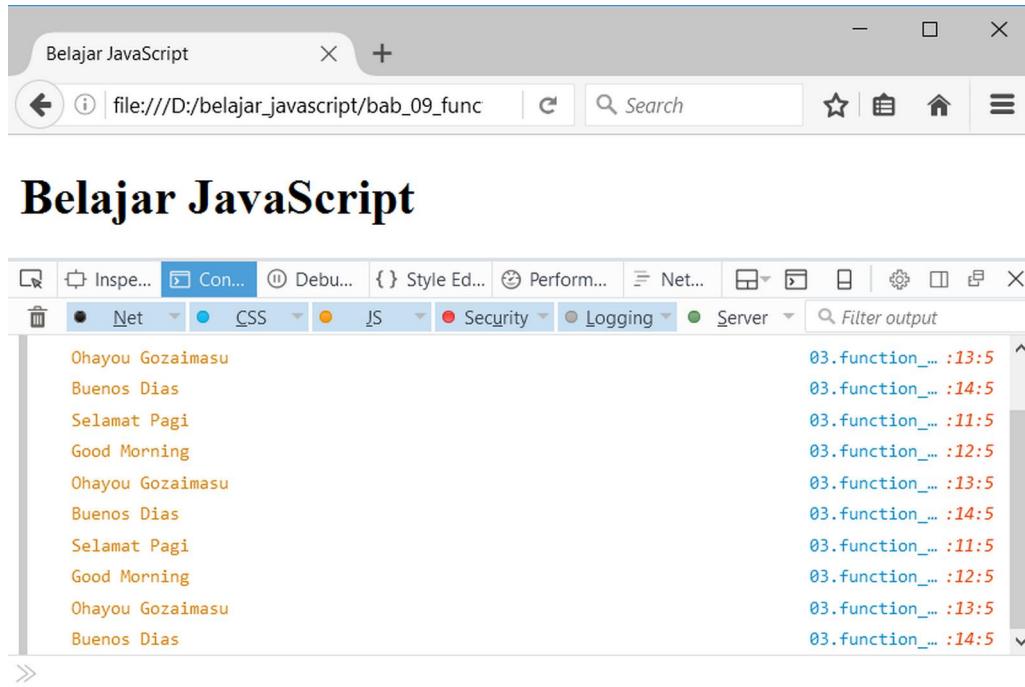
Gambar: Salam dengan Bahasa Indonesia, Inggris, Jepang, dan Spanyol

Setelah mendeklarasikan function pagi(), saya menjalankannya dengan cara menulis nama fungsi tersebut: pagi();.

Menjalankan fungsi dengan cara seperti ini dikenal dengan istilah **memanggil fungsi** (*calling a function*). Istilah lain adalah *running*, *executing*, *invoking*, atau *dispatching a function*.

Bagaimana jika kita coba memanggil fungsi ini beberapa kali?

```
1 function pagi(){
2     console.log("Selamat Pagi");
3     console.log("Good Morning");
4     console.log("Ohayou Gozaimasu");
5     console.log("Buenos Dias");
6 }
7
8 pagi();
9 pagi();
10 pagi();
```



Gambar: Pemanggilan fungsi pagi() beberapa kali

Karena ada 4 salam dalam 1 kali pemanggilan, ketika fungsi pagi() dipanggil sebanyak 3 kali, akan tampil 12 salam. Inilah kepraktisan menggunakan fungsi, kita bisa dengan mudah menjalankan kode program yang sama berulang kali, dan bisa dipanggil kapan saja.

## 9.3 Mengembalikan Nilai Function

Fungsi pagi() yang kita rancang sebelum ini sangat praktis untuk menampilkan 4 salam dalam setiap kali pemanggilan. Tapi bagaimana jika saya ingin salam ini tidak langsung tampil, tapi ditampung dulu ke dalam variabel (untuk nantinya diproses lagi)? Kita bisa menggunakan keyword **return**. Perintah **return** akan mengembalikan suatu nilai ketika fungsi ini dipanggil.

Mari kita lihat contoh penggunaannya:

```
1 function pagi(){
2     return "Selamat Pagi";
3 }
```

Kali ini saya mengubah fungsi pagi() menjadi hanya 1 baris: `return "Selamat Pagi"`. Dengan demikian, saya bisa menjalankannya sebagai berikut:

```
1 function pagi(){
2   return "Selamat Pagi";
3 }
4
5 var salam = pagi();
6 console.log(salam); // "Selamat Pagi"
```

Perhatikan cara pemanggilan fungsi pagi(), kali ini saya menyambungnya menggunakan *operator assignment*. Dengan kata lain, hasil dari fungsi pagi(), akan disimpan ke dalam variabel salam. Setelah itu, apa isi dari variabel salam ini? Adalah string "Selamat Pagi".

Selain menampungnya ke dalam variabel, saya juga bisa menulis seperti ini:

```
1 function pagi(){
2   return "Selamat Pagi";
3 }
4
5 console.log(pagi());
```

Bisakah apa menebak apa yang ditampilkan oleh perintah console.log(pagi())? String "Selamat Pagi". Ini berasal dari perintah return "Selamat Pagi" yang langsung ditampilkan oleh console.log().

Selain mengembalikan nilai, efek lain dari perintah return adalah, akan langsung menghentikan function tersebut, seperti contoh berikut:

```
1 function pagi(){
2   return "Selamat Pagi";
3   console.log ("Good Morning"); // tidak akan pernah di eksekusi
4 }
5
6 console.log(pagi()); // "Selamat Pagi"
```

Saya menempatkan 1 baris lagi setelah perintah return, yakni console.log ("Good Morning"). Baris ini tidak akan pernah dijalankan, karena pada saat kode program memproses return, function pagi() dianggap sudah selesai.

Dalam contoh sebelumnya saya mengembalikan nilai string yang langsung ditulis setelah return. Kita juga bisa mengembalikan nilai dalam bentuk variabel:

```

1 function pagi(){
2   var salam = "Selamat Pagi";
3   return salam;
4 }
5
6 console.log(pagi()); // "Selamat Pagi"

```

Contoh ini sangat mirip dengan sebelumnya. Hanya saja string "Selamat Pagi" kali ini saya tempatkan ke dalam variabel `salam`. Selanjutnya, variabel `salam` inilah yang `return`..

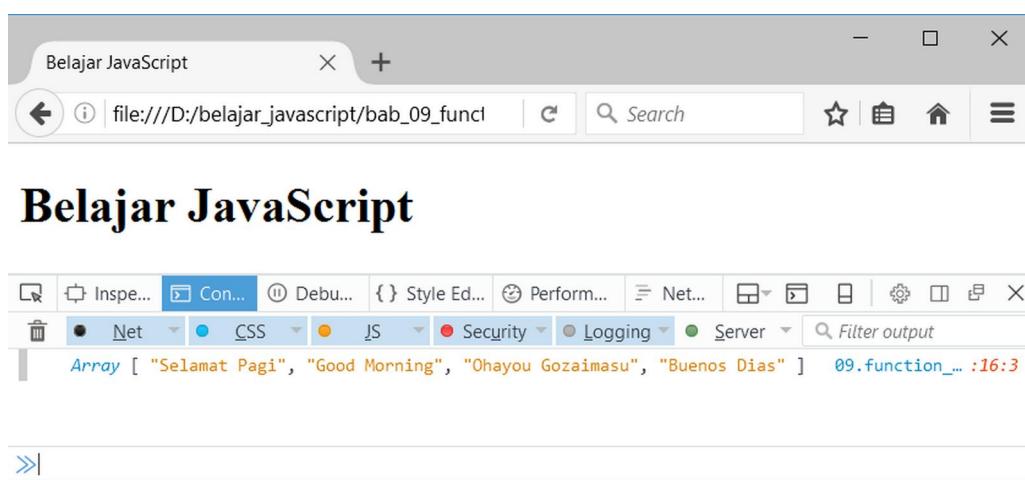
## Mengembalikan Array

Salah satu aturan dari perintah `return` adalah, kita hanya boleh mengembalikan 1 nilai saja, apakah itu 1 string, maupun 1 variabel. Jadi, bagaimana jika nilai yang dikembalikan ada banyak? Kita bisa memanfaatkan **array**:

```

1 function pagi(){
2   var salamPagi = ["Selamat Pagi", "Good Morning",
3                     "Ohayou Gozaimasu", "Buenos Dias" ];
4   return salamPagi;
5 }
6
7 var salam = pagi();
8 console.log(salam);
9 // Array [ "Selamat Pagi", "Good Morning", "Ohayou Gozaimasu", "Buenos Dias" ]

```



Gambar: Array yang berisi ucapan selamat pagi, berasal dari `function pagi()`

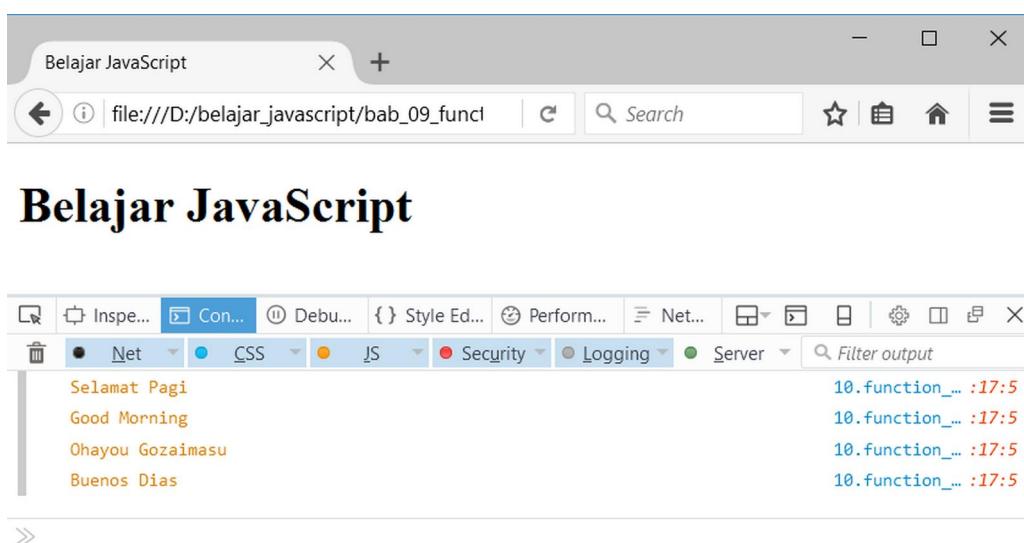
Saya membuat variabel `salamPagi` di dalam `function pagi()`, variabel ini selanjutnya diisi array yang terdiri dari 4 buah string. Perintah `return salamPagi` akan mengembalikan array `salamPagi`.

Ketika saya memanggil fungsi ini dengan `var salam = pagi()`, isi dari array `salamPagi` akan pindah ke variabel `salam`. Dengan trik ini, kita bisa mengembalikan banyak nilai menggunakan perintah `return`.

Sebagai latihan, bisakah anda membuat perulangan untuk menampilkan seluruh element array `salam`?

Berikut kode programnya:

```
1 function pagi(){
2     var salamPagi = ["Selamat Pagi", "Good Morning",
3                     "Ohayou Gozaimasu", "Buenos Dias" ];
4     return salamPagi;
5 }
6
7 var salam = pagi();
8 for (var value of salam) {
9     console.log(value);
10 }
```



Gambar: Menampilkan array dari function pagi()

Disini saya menggunakan perulangan `for of` untuk menampilkan array `salam`. Sebagai latihan lanjuta, bisakah anda membuatnya dengan perulangan `for` biasa ?

## 9.4 Argument Function

Fitur berikutnya dari function adalah **argument**, yakni mengirim satu atau beberapa nilai ke dalam function untuk diproses. Berikut contoh penggunaannya:

```

1 function pagi(siapa){
2   return "Selamat Pagi " + siapa;
3 }
4
5 var salam = pagi("Jakarta");
6 console.log(salam); // Selamat Pagi Jakarta

```

Perhatikan cara pendefenisian fungsi pagi(). Saya menulisnya dengan `function pagi(siapa)`. `siapa` disini merupakan sebuah *argument*, yang sebenarnya tidak lain merupakan variabel. Apa isi dari variabel ini? Nantinya ditentukan pada saat pemanggilan fungsi.

Di dalam block fungsi, kita bisa menggunakan variabel *argument* dengan bebas, mirip seperti cara penggunaan variabel biasa. Dalam contoh diatas, saya menyambungnya dengan string "Selamat Pagi ". Nilai ini akan menjadi nilai **return** dari function.

Pada saat fungsi pagi() dipanggil, saya menulis `pagi("Jakarta")`. String "Jakarta" inilah yang menjadi input ke fungsi pagi(). Dengan kata lain, saya mengisi variabel `siapa` di fungsi pagi() dengan string "Jakarta". Hasil akhir dari pemanggilan fungsi `pagi("Jakarta")`, adalah "Selamat Pagi Jakarta".

Dengan menyediakan argument yang bisa "diisi", kita bisa menampilkan berbagai string, tergantung cara pemanggilan fungsi pagi(), seperti contoh berikut:

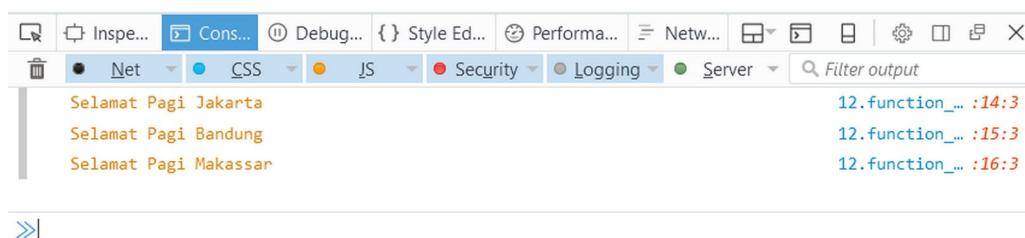
```

1 function pagi(siapa){
2   return "Selamat Pagi " + siapa;
3 }
4
5 console.log(pagi("Jakarta")); // Selamat Pagi Jakarta
6 console.log(pagi("Bandung")); // Selamat Pagi Bandung
7 console.log(pagi("Makassar")); // Selamat Pagi Makassar

```



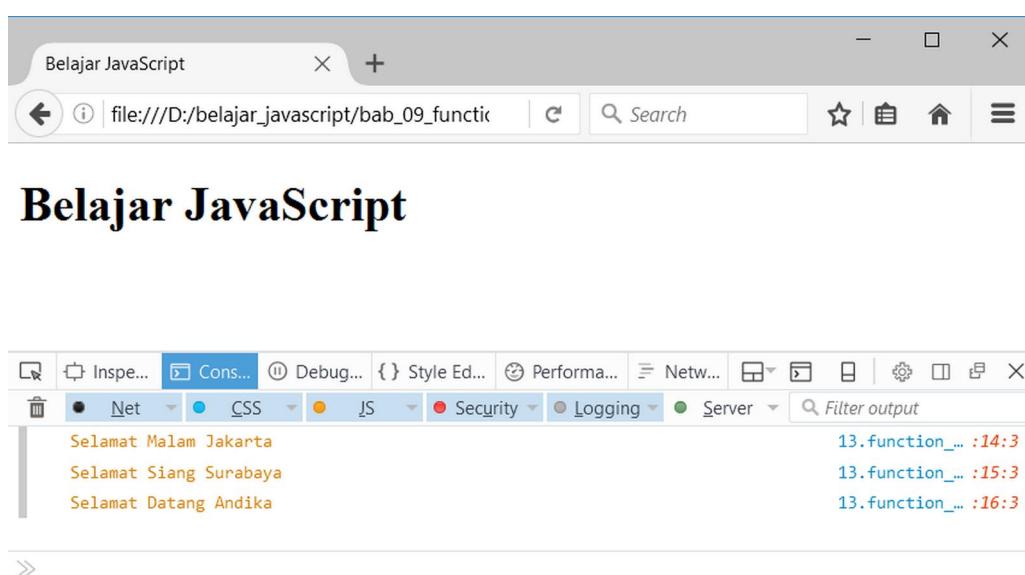
## Belajar JavaScript



Gambar: Function pagi() dengan berbagai argument

JavaScript tidak membatasi berapa banyak argument yang bisa dikirim, selama nilai tersebut sesuai dengan banyaknya input yang diberikan. Saya bisa modifikasi fungsi pagi() menjadi sebagai berikut:

```
1 function salam(kapan, siapa){  
2     return "Selamat " + kapan + " " + siapa;  
3 }  
4  
5 console.log(salam("Malam", "Jakarta")); // Selamat Malam Jakarta  
6 console.log(salam("Siang", "Surabaya")); // Selamat Siang Surabaya  
7 console.log(salam("Datang", "Andika")); // Selamat Datang Andika
```



Gambar: Function pagi() dengan 2 argument

Kali ini saya menambah argument fungsi pagi() menjadi 2 buah, yang disimpan ke dalam variabel kapan dan siapa. Anda bebas memilih nama argument ini, selama mengikuti aturan penulisan identifier. Argument function mirip seperti variabel, tapi kita tidak perlu menulis keyword var.

Sebagai contoh lain, saya akan membuat fungsi pencari rata-rata. Fungsi ini bernama rata2(), dan membutuhkan 4 argumen, hasil akhir dari fungsi ini berupa rata-rata dari ke-4 argumen. Silahkan anda coba merancang fungsi ini.

Baik, berikut kode program yang saya gunakan:

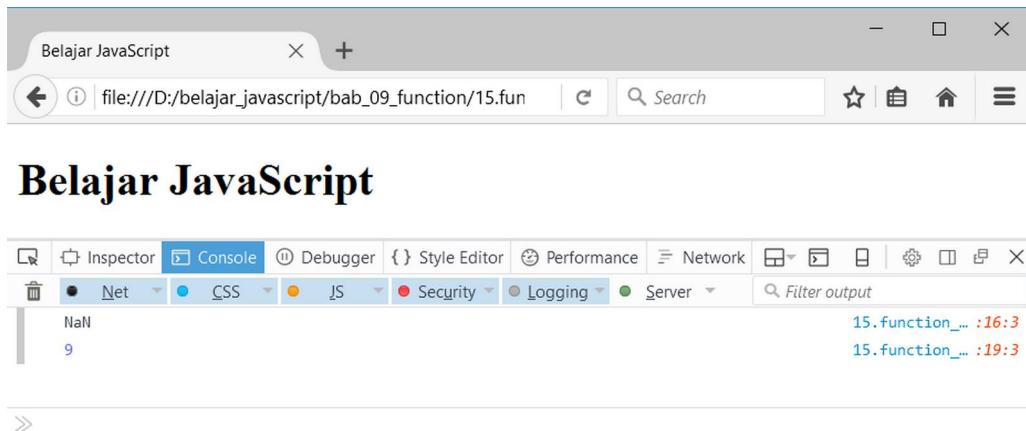
```
1 function rata2(a, b, c, d){  
2     var hasil = (a + b + c + d)/4;  
3     return hasil;  
4 }  
5  
6 var nilai1 = rata2(8, 5, 2, 4);  
7 console.log(nilai1);    // 4.7514  
8  
9 var nilai2 = rata2(90, 78, 95, 76);  
10 console.log(nilai2);   // 84.7514  
11  
12 var nilai3 = rata2(nilai1, 55, 60, nilai2);  
13 console.log(nilai3);   // 51.125
```

Silahkan anda pelajari sebentar kode program diatas. Disini fungsi `rata2()` memiliki 4 argumen: `a`, `b`, `c`, dan `d`. Setiap pemanggilan fungsi `rata2()`, harus menyertakan ke-4 argumen ini. Di dalam fungsi `rata2()`, saya menjumlahkan ke-4nya, kemudian dibagi 4.

Untuk contoh pemanggilan fungsi yang terakhir, dapat dilihat bahwa kita bisa menginput nilai argument yang berasal dari variabel lain, dalam hal ini saya mengambil angka `nilai1` dan `nilai2` sebagai input untuk `var nilai3= rata2(nilai1, 55, 60, nilai2)`.

Dalam contoh tentang fungsi `rata2()` diatas, saya membuatnya dengan 4 argument. Sehingga pada saat fungsi ini dipanggil, kita harus menginput 4 angka. Tapi bagaimana jika angka ini kurang atau malah berlebih? Mari kita coba:

```
1 function rata2(a, b, c, d){  
2     var hasil = (a + b + c + d)/4;  
3     return hasil;  
4 }  
5  
6 var nilai1 = rata2(8, 5, 2);  
7 console.log(nilai1);    // NaN (tidak error!)  
8  
9 var nilai2 = rata2(9, 9, 9, 9, 1, 1);  
10 console.log(nilai2);   // 9 (tidak error!)
```



Gambar: Fungsi rata2 dipanggil dengan jumlah argument yang kurang dan berlebih

Pertama, untuk nilai1 saya memanggil fungsi `rata2()` dengan 3 argumen, yang artinya kurang 1 angka. Bagaimana hasilnya? `NaN`. Ini terjadi karena saat argument ke-4 tidak ditulis, JavaScript memberikan nilai `undefined` kepada variabel `d`. Operasi  $(8 + 5 + 2 + \text{undefined}) / 2$  hasilnya adalah `NaN`.

Kedua, untuk nilai2 saya memanggil fungsi `rata2()` dengan 6 argument. Sekarang jumlah argument malah berlebih. Hasilnya? `9`. Artinya, argument ke 5 dan ke 6 akan diabaikan.

Yang patut diperhatikan, dalam kedua kasus ini JavaScript tidak mengeluarkan error apapun. Walaupun kita menulis nilai argument yang tidak sesuai dengan yang dibutuhkan oleh function tersebut.

## Argumen vs Parameter

Jika anda mempelajari buku teks algoritma dan pemrograman, terdapat istilah **argument** dan **parameter**. Keduanya hampir sama, tetapi sedikit berbeda.

Parameter adalah sebutan untuk inputan fungsi pada saat pendefenisian fungsi tersebut. Variabel `a`, `b`, `c`, dan `d` dari fungsi `rata2()` disebut sebagai **parameter**.

Sedangkan argument adalah sebutan untuk inputan fungsi pada saat pemanggilan fungsi tersebut. Angka `8`, `5`, `2` dan `4` saat pemanggilan fungsi `rata2(8, 5, 2, 4)` disebut sebagai **argument**.

Dalam penggunaan sehari-hari, kedua istilah ini sering dipertukarkan. Di dalam pembahasan tentang JavaScript, istilah **argument** lebih banyak dipakai untuk menyebut keduanya.

## 9.5 Default Argument

**Default argument** adalah mempersiapkan nilai awal untuk argument. Dengan tujuan, jika argument ini tidak diinput, nilai awal akan digunakan. Kenapa harus diberikan nilai awal? karena jika sebuah argument tidak diisi, JavaScript akan memberikan nilai `undefined` (seperti contoh kita sebelum ini).

Berikut contoh penggunaan *default argument*:

```
1 function rata2(a = 10, b = 10, c = 10, d = 10){  
2     var hasil = (a + b + c + d)/4;  
3     return hasil;  
4 }  
5  
6 var nilai1 = rata2();  
7 console.log(nilai1);    // 10  
8  
9 var nilai2 = rata2(20);  
10 console.log(nilai2);   // 12.5  
11  
12 var nilai3 = rata2(20, 5, 30);  
13 console.log(nilai3);   // 16.25
```

Pada saat pembuatan fungsi `rata2()`, saya menulisnya sebagai berikut:

```
function rata2(a = 10, b = 10, c = 10, d = 10)
```

Artinya, jika fungsi `rata2()` dipanggil dengan jumlah argument yang tidak cukup (kurang dari 4 nilai), angka disamping tanda sama dengan akan digunakan. Dengan demikian, jika saya memanggil fungsi `rata2` tanpa menuliskan argument apapun, hasilnya adalah 10. Ini didapat dari  $(10 + 10 + 10 + 10) / 4$ .

Bagaimana jika dipanggil dengan 1 argumen saja? Seperti `var nilai2 = rata2(20)?` Disini, nilai 20 akan diinput ke dalam variabel `a`, sedangkan untuk argumen `b, c`, dan `d` akan menggunakan nilai default, yakni 10.

Begitu juga ketika fungsi ini dipanggil dengan 3 argumen: `var nilai3 = rata2(20, 5, 30)`. Nilai rata-rata yang dihitung adalah  $(20 + 5 + 30 + 10)/4$ . Angka 10 terakhir berasal dari nilai default argumen `d`.

Fitur *default argument* seperti ini baru tersedia di **ECMAScript 6**. Untuk versi JavaScript sebelumnya, harus diakali dengan trik khusus, yakni menggunakan prinsip *short circuit* dari operasi perbandingan.

Syarat lain dari *default argument* adalah argument dengan nilai default harus diletakkan sebagai argument terakhir. Misalkan jika saya ingin hanya 2 argument saja yang memiliki nilai default, maka itu adalah untuk argument `c` dan `d`:

```
1 function rata2(a, b, c = 10, d = 10){  
2     var hasil = (a + b + c + d)/4;  
3     return hasil;  
4 }  
5  
6 var nilai1 = rata2(20);  
7 console.log(nilai1);    // NaN  
8  
9 var nilai2 = rata2(20, 40);  
10 console.log(nilai2);   // 20  
11  
12 var nilai3 = rata2(5, 5, 5, 5);  
13 console.log(nilai3);   // 5
```

Pemanggilan pertama menghasilkan nilai `Nan` karena fungsi `rata2()` butuh minimal 2 argumen (2 argumen terakhir bersifat *opsional*).

Bagaimana jika default argument ini ditempatkan di awal?

```
1 function rata2(a = 10, b = 10, c, d){  
2     var hasil = (a + b + c + d)/4;  
3     return hasil;  
4 }  
5  
6 var nilai1 = rata2(20);  
7 console.log(nilai1);    // NaN  
8  
9 var nilai2 = rata2(20, 40);  
10 console.log(nilai2);   // NaN  
11  
12 var nilai3 = rata2(5, 5, 5, 5);  
13 console.log(nilai3);   // 5
```

Dari sisi logika program, meletakkan *default argument* diawal tidak akan berfungsi, karena mau tidak mau kita harus menulis seluruh argument. Kalau tidak, hasilnya adalah `Nan`, karena argument yang tidak memiliki nilai *default* akan menjadi `undefined`.

Tapi dengan trik khusus, ini bisa dilakukan:

```
1 function rata2(a = 10, b = 10, c, d){  
2     var hasil = (a + b + c + d)/4;  
3     return hasil;  
4 }  
5  
6 var nilai3 = rata2(undefined, undefined, 5, 5);  
7 console.log(nilai3);    // 7.5
```

Hal yang unik adalah, saya menuliskan nilai `undefined` pada saat pemanggilan fungsi. Dengan ditulis seperti ini, argument `a` dan `b` akan mengambil nilai *default*. Kode program diatas menghasilkan nilai 7.5 yang berasal dari  $(10+10+5+5)/4$ . Inilah sedikit fungsi dari nilai `undefined` di dalam pemrograman JavaScript.

## 9.6 Arguments Object

Materi tentang **object** akan kita pelajari pada bab selanjutnya, tapi karena sangat berhubungan dengan function, pembahasan tentang *arguments object* akan saya bahas disini.

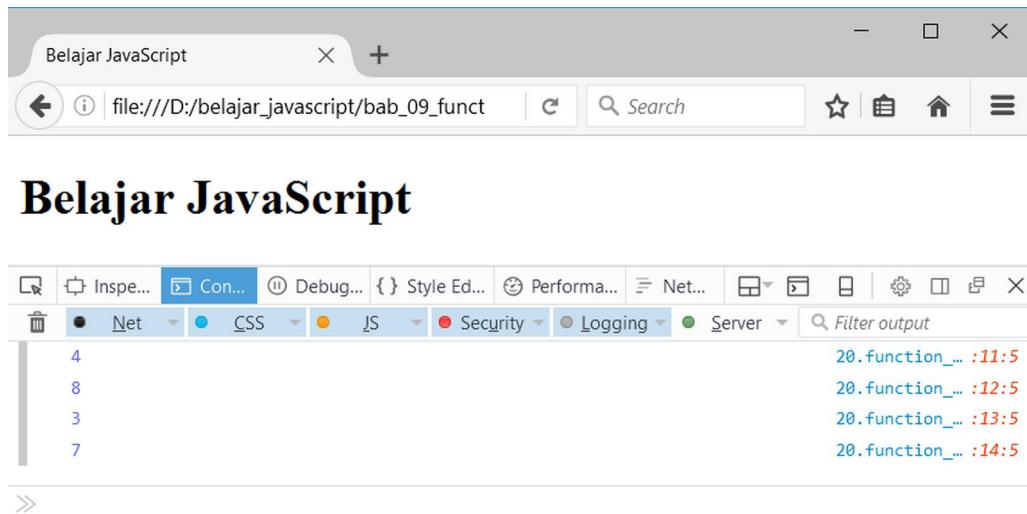
**Arguments object** adalah sebutan untuk ‘object’ yang menampung seluruh argument pada saat pemanggilan function. *Arguments object* sebenarnya lebih mirip seperti **array**. Oleh karena itu boleh dibilang bahwa kita sedang membahas tentang array khusus bernama **arguments**.

Mari lihat contoh penggunaannya:

```
1 function angka(){  
2     console.log(arguments[0]);  
3     console.log(arguments[1]);  
4     console.log(arguments[2]);  
5     console.log(arguments[3]);  
6 }  
7  
8 angka(4, 8, 3, 7);
```

Saya mendefenisikan fungsi `angka()` tanpa membuat tempat untuk argument. Di dalam fungsi ini terdapat empat perintah `console.log()`, yang masing-masingnya menampilkan array `arguments` mulai dari index 0 hingga 4.

Kemudian saya memanggil fungsi `angka()` dengan 4 buah argument: `angka(4, 8, 3, 7)`. Bagaimana hasilnya?



Gambar: Argument dari sebuah fungsi ditampilkan menggunakan object “arguments”

Seperti yang terlihat, setiap *argument* yang saya tulis bisa diakses menggunakan *array arguments*, dimana argument pertama disimpan ke dalam `arguments[0]`, argument kedua ke dalam `arguments[1]`, dst. *Array arguments* ini tersedia di dalam setiap function JavaScript dan bisa diakses dari dalam function tersebut.

Karena mirip array, kita bisa mencari tau berapa banyak element yang ada di dalam *arguments object*. Yang sebenarnya juga untuk mencari tau seberapa banyak jumlah argument pada saat pemanggilan fungsi. Caranya adalah dengan mengakses *property length*:

```

1  function angka(){
2    var totalArg = arguments.length;
3    return totalArg;
4  }
5
6  var a = angka(1, 9);
7  console.log(a); // 2
8
9  var b = angka(4, 8, 3, 7);
10 console.log(b); // 4
11
12 var c = angka(3, 9, 1, 4, 6, 9, 0);
13 console.log(c); // 7

```

Disini saya mendefenisikan fungsi `angka()` hanya untuk mengetahui berapa banyak argument yang ditulis pada saat pemanggilan fungsi tersebut. Perintah `arguments.length` berisi informasi terkait hal ini.

Berbekal *array argument* dan `arguments.length`, kita bisa membuat fungsi `rata2()` yang bisa menerima berapapun jumlah argument. Untuk menghitung rata-rata, saya bisa menggunakan sebuah perulangan. Berikut kode programnya:

```

1 function rata2(){
2   var totalArg = arguments.length;
3   var hasil = 0;
4   for (var i = 0; i < totalArg; i++){
5     hasil = hasil + arguments[i];
6   }
7   return hasil/totalArg;
8 }
9
10 var a = rata2(1, 9);
11 console.log(a); // 5
12
13 var b = rata2(4, 8, 3, 7);
14 console.log(b); // 5.5
15
16 var c = rata2(14, 34, 17, 55, 98, 22, 26);
17 console.log(c); // 38

```

Agar mudah membaca kode program tersebut, silahkan lompati pendefinisian fungsi `rata2()` untuk sementara, dan lihat bagaimana fungsi ini dipanggil.

Ketika saya memanggil fungsi `rata2()` dengan 2 argument, yakni `rata2(1, 9)`, hasilnya adalah 5. Nilai ini didapat dari argument pertama (1) ditambah argument kedua (9), lalu dibagi dengan jumlah argument yang ada (2). Perhitungannya menjadi  $(1 + 9)/2 = 5$ .

Bagaimana dengan 4 dan 7 argument? Perhitungannya juga sama, dimana nilai setiap argument dijumlahkan, lalu dibagi dengan berapa banyak argument tersebut.

Untuk merancang fungsi seperti ini, saya menggunakan **arguments object**. Karena fungsi ini harus fleksibel dan bisa menerima barapa pun banyak argument yang ada. Mari kita bahas cara kerja fungsi `rata2()` ini:

```

1 function rata2(){
2   var totalArg = arguments.length;
3   var hasil = 0;
4   for (var i = 0; i < totalArg; i++){
5     hasil = hasil + arguments[i];
6   }
7   return hasil/totalArg;
8 }

```

Satu hal yang pasti, saya tidak perlu menulis argument, karena kita akan mengaksesnya dari array `arguments` nanti.

Baris pertama dari fungsi `rata2()` berisi perintah `var totalArg = arguments.length`. Ini digunakan untuk mengambil informasi berapa banyak argument yang ada ketika fungsi ini dipanggil.

Selanjutnya, saya membuat perulangan:

```
1 for (var i = 0; i < totalArg; i++){
2   hasil = hasil + arguments[i];
3 }
```

Fungsi dari perulangan ini adalah menjumlahkan setiap argument yang ada. Berapa kali perulangan berlangsung? Tergantung dari jumlah argument, dan informasi ini sudah kita dapatkan dari variabel `totalArg`.

Setelah perulangan selesai, variabel `hasil` akan berisi nilai total dari seluruh argument. Untuk mencari nilai rata-ratanya, saya tinggal membagi isi variabel `hasil` dengan `totalArg`. Nilai ini dikembalikan menggunakan perintah `return hasil/totalArg`.

Fitur yang disediakan oleh `arguments object` sangat praktis untuk membuat fungsi yang fleksibel, seperti contoh kita kali ini. Sebagai alternatif, hal yang sama bisa didapat menggunakan `spread operator`.

## 9.7 Spread Operator untuk Argument

**Spread operator** (...) merupakan fitur baru di **ECMAScript 6**. Fungsi pertamanya sudah kita bahas di bab tentang operator, yakni untuk menggabungkan array. Fungsi lain dari operator ini adalah menggantikan peran `arguments object`.

Penggunaannya sangat mirip seperti `arguments object`, seperti contoh berikut:

```
1 function angka(...arg){
2   console.log(arg[0]); // 4
3   console.log(arg[1]); // 8
4   console.log(arg[2]); // 3
5   console.log(arg[3]); // 7
6 }
7
8 angka(4, 8, 3, 7);
```

Perhatikan penulisan argument pada saat pendefenisian fungsi `angka()`, saya menulis `angka(...arg)`. Ini bisa dibaca: Jika fungsi `angka()` dipanggil dengan argument, masukkan seluruh argument tersebut ke dalam variabel `arg`.

Variabel `arg` akan menjadi array yang berisi seluruh argument ketika pemanggilan fungsi. Kurang lebih mirip seperti `arguments object` yang kita pelajari sebelum ini.

Nama variabel `arg` hanya sebagai contoh, anda bebas menggantinya dengan nama lain:

```
1 function angka(...foo){  
2   console.log(foo[0]); // 4  
3   console.log(foo[1]); // 8  
4   console.log(foo[2]); // 3  
5   console.log(foo[3]); // 7  
6 }  
7  
8 angka(4, 8, 3, 7);
```

Selain itu, kita juga bisa membuat argument biasa yang disambung dengan *spread operator*, seperti contoh berikut:

```
1 function angka(a, b, ...sisa){  
2   console.log(a); // 4  
3   console.log(b); // 8  
4   console.log(sisa); // Array [ 3, 7 ]  
5 }  
6  
7 angka(4, 8, 3, 7);
```

Pendefinisian function `angka(a, b, ...sisa)` bisa dibaca: Jika fungsi `angka()` dipanggil oleh lebih dari 3 argument, maka argument pertama dan kedua masuk ke variabel `a` dan `b`, sisanya disimpan kedalam array `sisa`.

Sebagai latihan, bisakah anda membuat fungsi `rata2()` menggunakan *spread operator*? Konsepnya sama seperti contoh pada *arguments object*. Selain itu untuk perulangan jumlah element, gunakan perulangan `for of`. Silahkan anda rancang sebentar.

Baik, berikut kode yang saya gunakan:

```
1 function rata2(...semuaArg){  
2   var totalArg = semuaArg.length;  
3   var hasil = 0;  
4   for (var i of semuaArg){  
5     hasil = hasil + i;  
6   }  
7   return hasil/totalArg;  
8 }  
9  
10 var a = rata2(1, 9);  
11 console.log(a); // 5  
12  
13 var b = rata2(4, 8, 3, 7);  
14 console.log(b); // 5.5  
15  
16 var c = rata2(14, 34, 17, 55, 98, 22, 26);  
17 console.log(c); // 38
```

Nyaris tidak berbeda dengan contoh *arguments object* sebelumnya.

Dengan menggunakan fitur *arguments object* maupun *spread operator*, kita bisa merancang function yang sangat fleksibel.

## 9.8 Variable Scope

**Variable scope** adalah istilah tentang sejauh mana sebuah variabel masih dapat diakses. Di dalam JavaScript terdapat **global variable**, yakni variabel yang bisa diakses darimana saja. Selain itu ada **local variable**, yakni variabel yang hanya bisa diakses di dalam ruang lingkup terbatas, seperti di dalam sebuah *function*.

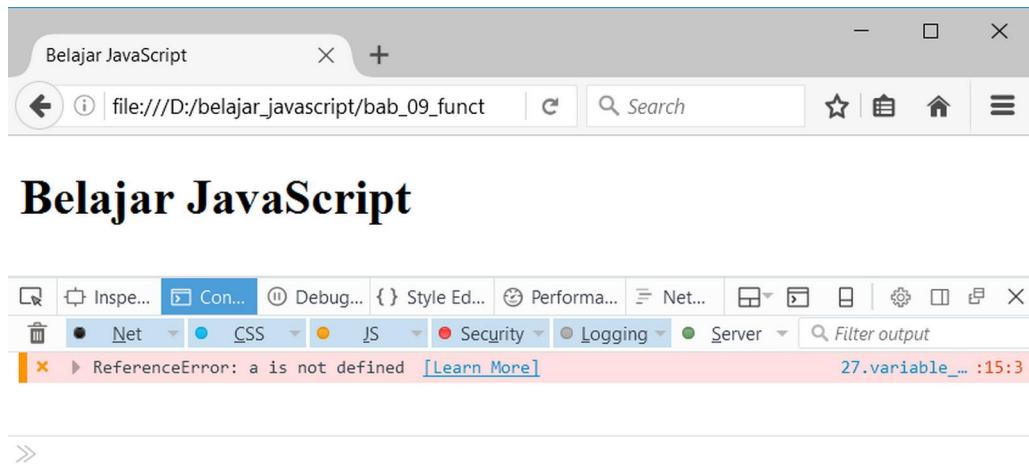
Agar lebih jelas, mari kita bahas menggunakan contoh kode program. Perhatikan kode berikut:

```
1 var a = "Belajar JavaScript";
2
3 function foo(){
4     console.log(a);
5 }
6
7 foo(); // Belajar JavaScript
```

Saya membuat sebuah variabel *a* yang diisi string "Belajar JavaScript". Di dalam JavaScript, variabel ini merupakan **global variable**, karena di definisikan tidak di dalam scope apapun. Variabel *a* ini bisa diakses dari mana saja, termasuk dari dalam *function* seperti contoh tersebut.

Sekarang, bagaimana jika pendefenisian variabel *a* ini kita pindahkan ke dalam *function*?

```
1 function foo(){
2     var a = "Belajar JavaScript";
3 }
4
5 foo();
6 console.log(a); // ReferenceError: a is not defined
```



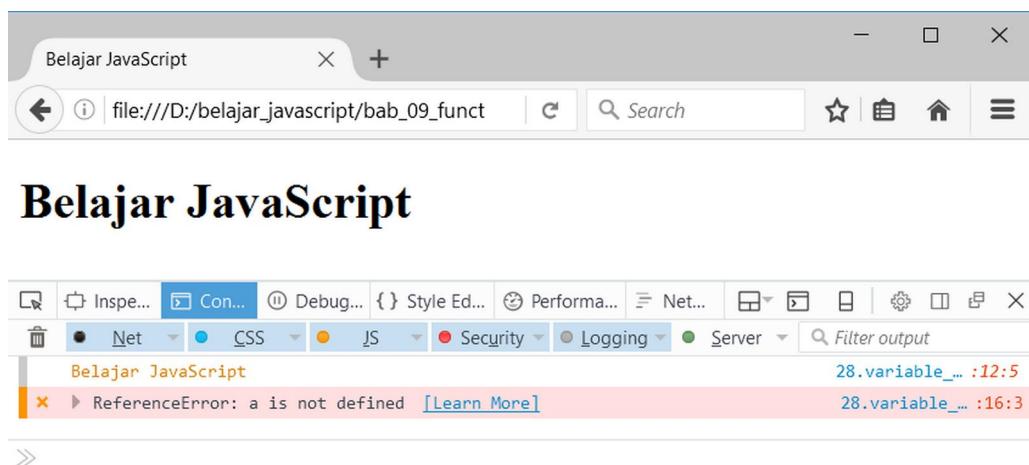
Gambar: Pesan error karena variabel a tidak terdefenisi

Hasilnya akan tampil pesan error. Ini terjadi karena ketika sebuah variabel di buat di dalam function, variabel itu hanya “ada” di dalam function, atau di sebut sebagai **local variable**. Ketika diakses dari luar function, variabel a dianggap tidak ada.

Sebagai buktinya, mari kita akses variabel a dari dalam function itu sendiri:

```

1 function foo(){
2   var a = "Belajar JavaScript";
3   console.log(a); // Belajar JavaScript
4 }
5
6 foo();
7 console.log(a); // ReferenceError: a is not defined
  
```



Gambar: Variabel a sukses diakses dari dalam function

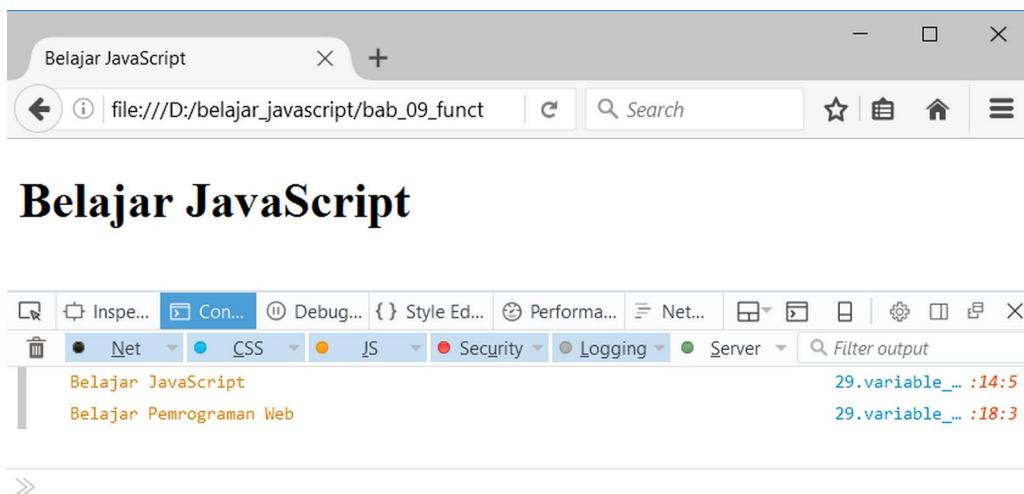
Walaupun masih terdapat error, tapi itu karena saya tetap mencoba mengakses variabel a dari luar function. Sedangkan ketika diakses dari dalam function, isi variabel a berhasil ditampilkan.

Selanjutnya, dapatkah anda menjelaskan apa yang terjadi di dalam kode program berikut:

```

1 var a = "Belajar Pemrograman Web";
2
3 function foo(){
4     var a = "Belajar JavaScript";
5     console.log(a); // Belajar JavaScript
6 }
7
8 foo();
9 console.log(a); // Belajar Pemrograman Web

```



Gambar: Global variable vs local variable

Sekilas, hanya ada 1 variabel `a`, tapi sebenarnya terdapat 2 jenis variabel `a` yang saling terpisah, yang kebetulan bernama sama.

Di awal kode program, saya membuat `var a = "Belajar Pemrograman Web"`. Ini adalah **global variabel a**. Ketika fungsi `foo()` di jalankan, di dalamnya saya buat variabel `a` lagi dengan perintah `var a = "Belajar JavaScript"`. Variabel `a` disini bukanlah **global variabel a**, tapi **local variabel a** yang hanya ada di dalam fungsi ini.

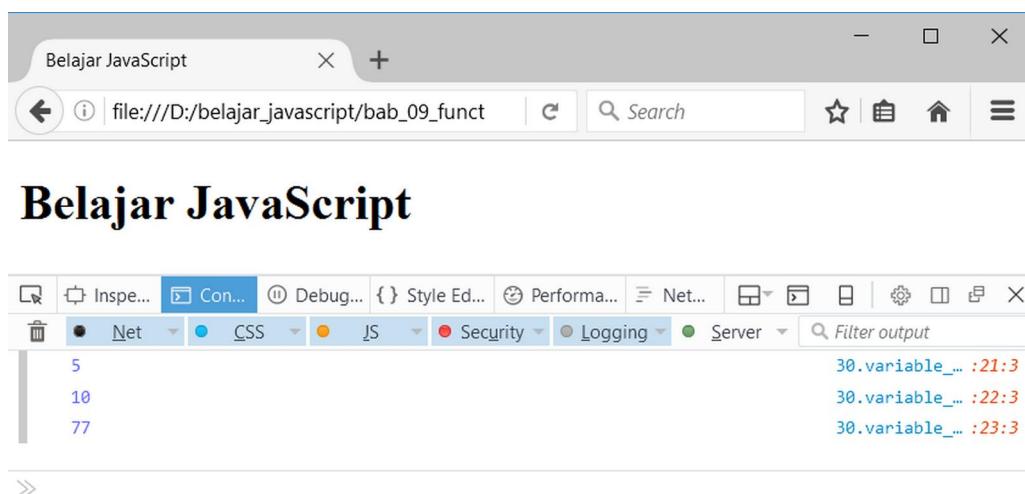
Meskipun di dalam function `foo()` variabel `a` sudah saya tukar nilainya, di luar fungsi `foo()`, isi variabel `a` masih "`Belajar Pemrograman Web`", bukan "`Belajar JavaScript`". Dengan kata lain, di dalam function `foo()`, yang berlaku adalah variabel `a` kepunyaan function itu sendiri, bukan **global variabel a**.

Prinsip pemisahan antara **global variable** dan **local variable** ini juga berlaku untuk *argument*, seperti contoh berikut:

```

1 function foo(a,b){
2     a = 21;
3     b = 56;
4     return (a + b);
5 }
6
7 var a = 5;
8 var b = 10;
9
10 var c = foo(a,b);
11
12 console.log(a); // 5
13 console.log(b); // 10
14 console.log(c); // 77

```



Gambar: Local scope juga berlaku untuk argument

Agar mudah memahami kode program yang melibatkan *function*, kita harus lompati dulu pendefenisian fungsi tersebut. Karena sebuah fungsi baru akan berjalan ketika dipanggil.

Mari kita lompati sejenak pendefenisian function `foo()`. Pertama kali saya membuat 2 buah variabel: `a` dan `b`. Kedua variabel ini diisi dengan angka 5 dan 10. Selanjutnya keduanya saya input sebagai argument untuk pemanggilan fungsi `foo()` menggunakan baris perintah `var c = foo(a,b)`.

Selanjutnya, apa yang dilakukan oleh fungsi `foo()`? Mari kita bahas. Nilai yang dikirim dari pemanggilan `foo()`, akan disimpan ke dalam argument `a` dan `b`, karena saya menulis `function foo(a,b)` ketika pembuatan fungsi ini. Dengan kata lain, function `foo()` akan menerima 2 argument.

Sampai disini, di dalam function `foo()` terdapat 2 buah variabel: `a` dan `b`, yang didefinisikan sebagai *argument*. Variabel `a` dan `b` ini merupakan variabel baru, yakni variabel **local scope** yang berbeda dengan variabel `a` dan `b` di luar function. Kasusnya sama seperti contoh kita sebelum ini.

Sebagai pembuktian, di dalam function `foo()`, saya mengubah nilai variabel **a** menjadi 21, dan **b** menjadi 56, kemudian mengembalikan nilai totalnya. Apakah ini mengubah nilai variabel **a** dan **b** yang ada di luar function?

Hasil pemanggilan `console.log()` memperlihatkan bahwa variabel **a** dan **b** yang ada diluar function, tetap bernilai 5 dan 10, bukan 21 dan 56. Contoh ini kembali mempertegas pemisahan **global variable** dengan **local variable**.

Kode program tersebut, sama artinya jika ditulis seperti ini:

```
1 function foo(){
2     var a = 21;
3     var b = 56;
4     return (a + b);
5 }
6
7 var a = 5;
8 var b = 10;
9
10 var c = foo();
11
12 console.log(a); // 5
13 console.log(b); // 10
14 console.log(c); // 77
```

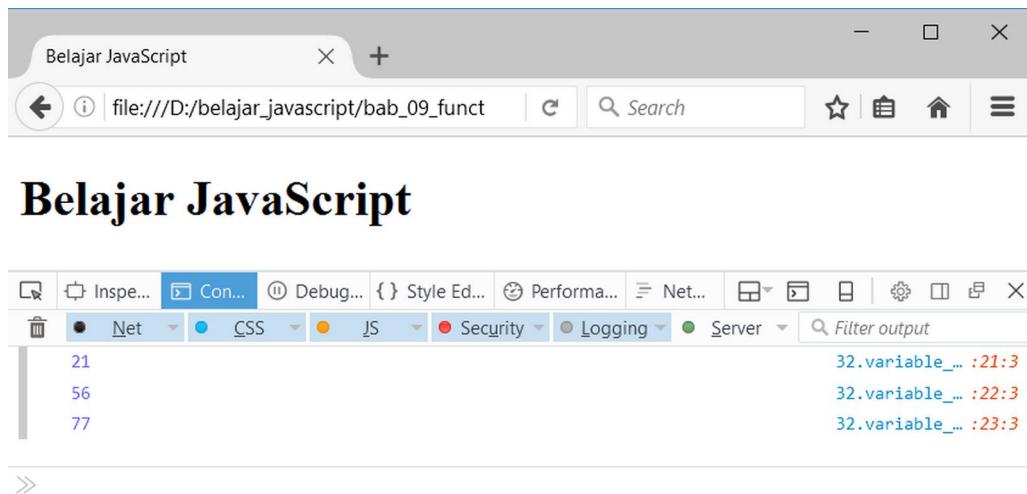
Disini, saya mendefenisikan function `foo()` tanpa argument, namun variabel **a** dan **b** dibuat menggunakan keyword `var a = 21`, dan `var b = 56`. Keduanya akan menjadi **local variable** yang sepenuhnya terpisah dari global variable **a** dan **b** yang di defenisikan di luar function.

Dalam prakteknya, saya tidak menyarankan anda menggunakan variabel yang bernama sama, baik di global maupun di local. Jika ingin membuat variabel **a** dan **b** di *global scope*, akan lebih baik jika saya menggunakan variabel **c** dan **d** sebagai argument function `foo()`, kode program kita akan jauh lebih mudah dipahami.

Semoga anda bisa memahami penjelasan saya sebelum ini. Tapi, bagaimana dengan kode program berikut?

```
1 function foo(){
2     a = 21;
3     b = 56;
4     return (a + b);
5 }
6
7 var a = 5;
8 var b = 10;
9
10 var c = foo();
11
```

```
12 console.log(a); // 21
13 console.log(b); // 56
14 console.log(c); // 77
```



Gambar: Local scope tidak berlaku?

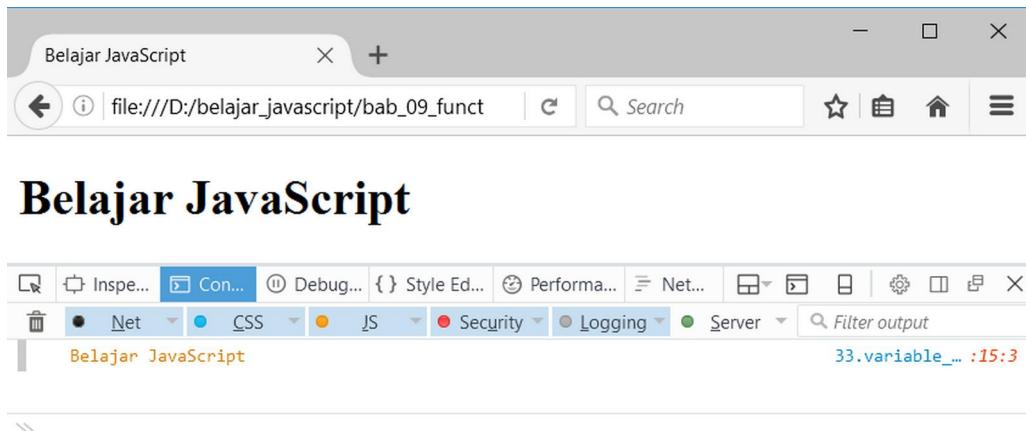
Yang pertama, dapatkah anda melihat perbedaan kode program ini dengan sebelumnya? Jika tidak, kenapa hasilnya variabel **a** menjadi 21, dan variabel **b** menjadi 56? Apakah *local scope* tidak berlaku disini?

Perbedaannya adalah, di dalam function `foo()`, saya mendefenisikan variabel **a** dan **b** tanpa perintah `var`.

Ini merupakan perbedaan mendasar jika sebuah variabel di defenisikan **tanpa var**. Ketika ditulis seperti ini, variabel **a** akan berefek ke *global scope*, walaupun ditulis dari dalam function.

Contoh berikut akan memperjelas konsep ini:

```
1 function foo(){
2   a = "Belajar JavaScript";
3 }
4
5 foo();
6 console.log(a); // "Belajar JavaScript"
```



Gambar: Membuat global variabel dari dalam function

Jika anda masih ingat, kode program diatas sudah pernah kita jalankan di awal pembahasan tentang *variable scope*. Ketika itu saya menulis `var a = "Belajar JavaScript"` dari dalam function `foo()`. Dan hasilnya adalah *ReferenceError: a is not defined*, karena variabel `a` hanya terdefenisi di dalam function.

Namun kali ini, saya tidak menggunakan perintah `var` untuk mendefenisikan variabel `a`. Hasilnya? Variabel `a` ini menjadi **global variable!**

Sepintas ini adalah cara mendefenisikan *global variabel* dari dalam function. Tetapi seperti pembahasan dan contoh kasus di bab 4 tentang variabel, pembuatan sebuah variabel tanpa perintah `var` tidak disarankan.

Jadi bagaimana membuat *global variable* dari dalam function? Caranya sebagai berikut:

```

1 var a;
2
3 function foo(){
4     a = "Belajar JavaScript";
5 }
6
7 foo();
8 console.log(a); // "Belajar JavaScript"

```

Disini, variabel `a` kita deklarasikan terlebih dahulu di *global scope*, tapi tanpa nilai. Cukup `var a` saja. Ketika variabel `a` ini diubah dari dalam function, yang diubah merupakan variabel `a` global, bukan variabel `a` secara local.

Kenapa harus dibuat seperti ini? Apa salahnya jika tanpa deklarasi `var a`? Penjelasannya saya rasa terlalu teknis, tapi bisa dibilang kalau sebuah variabel dibuat tanpa `var`, bisa menyebabkan bug untuk beberapa kasus. Jika anda tertarik, bisa membaca penjelasannya kesini: [stackover-flow.com](http://stackoverflow.com)<sup>1</sup>.

Secara umum, membuat global variabel dari dalam function bukan ide yang bagus. Sedapat mungkin agar selalu membuat variabel local. Setidaknya untuk menghindari bentrok dengan variabel global lain yang mungkin saja menggunakan nama sama.

<sup>1</sup><http://stackoverflow.com/questions/5786851/define-global-variable-in-a-javascript-function>

## 9.9 Perbedaan Cara Membuat Variabel var vs let

Dalam bab 4 yang membahas materi variabel dan konstanta, saya pernah menyungguh bahwa di ECMAScript 6 tersedia keyword `let` yang juga digunakan untuk mendeklarasikan variabel.

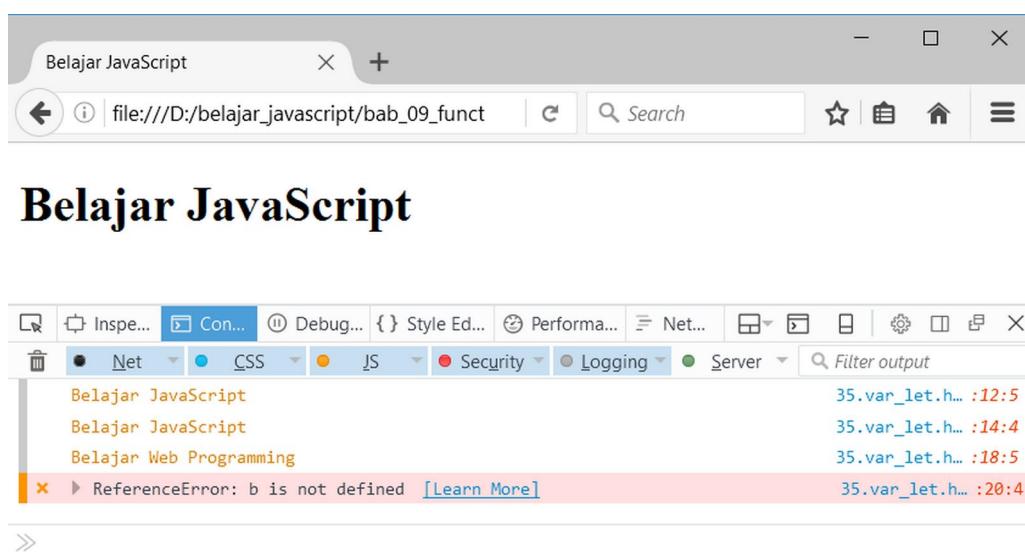
Pada materi tersebut saya belum bisa menjelaskan tentang perbedaan keduanya, karena butuh pemahaman tentang *variable scope* terlebih dahulu.

Perbedaan antara `var` dan `let` ada di *variable scope*. Jika kita membuat variabel dengan `var`, variabel tersebut *dibatasi di ruang function*. Dengan kata lain, yang bisa membatasi scope variabel yang di definisikan dengan keyword `var` hanyalah sebuah *function*.

Dilain pihak, variabel yang dibuat menggunakan `let` memiliki scope yang lebih kecil, yakni *dibatasi oleh tanda kurung kurawal* yang biasanya digunakan untuk block kode program. Selain itu juga dibatasi oleh *function* seperti layaknya variabel yang dibuat dengan `var`.

Berikut contoh perbedaannya:

```
1  {
2    var a = "Belajar JavaScript";
3    console.log(a);      // Belajar JavaScript
4  }
5  console.log(a);      // Belajar JavaScript
6
7  {
8    let b = "Belajar Web Programming";
9    console.log(b);      // Belajar Web Programming
10 }
11 console.log(b);      // ReferenceError: b is not defined
```



Gambar: Variabel `let` tidak bisa diakses diluar block kode program

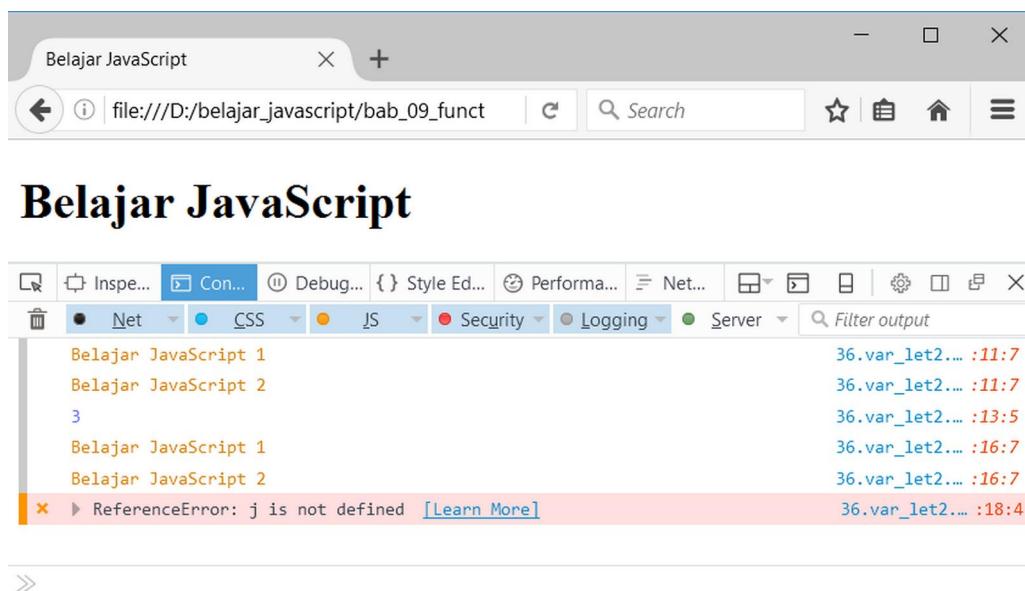
Dapat anda lihat bahwa variabel `b` tidak terdefinisi jika diakses dari luar block kode program. Dimana block kode program ini ditandai dengan tanda kurung kurawal.

Tanda block kode program ini biasa digunakan oleh alur logika seperti `if` `else`, atau perulangan `for`, seperti contoh berikut:

```

1  for (var i = 1; i < 3; i++) {
2      console.log("Belajar JavaScript " + i);
3  }
4  console.log(i); // 3
5
6  for (let j = 1; j < 3; j++) {
7      console.log("Belajar JavaScript " + j);
8  }
9  console.log(j); // i is not defined

```



Gambar: Variabel counter `let j` tidak bisa diakses dari luar perulangan

Kali ini saya membuat *variabel counter* `j` menggunakan `let`. Dan seperti yang sudah diperkirakan, variabel `j` tidak bisa diakses dari luar perulangan.

Sekilas terlihat membuat variabel dengan `let` lebih merepotkan, karena kita tidak bisa diakses nilainya dari luar scope. Tapi ini sebenarnya sebuah keuntungan. Dengan membatasi ruang lingkup sebuah variabel, kita terhindar dari masalah bentrok antar variabel. Terutama jika kode program sudah cukup banyak dan melibatkan ratusan variabel.

Contoh kasusnya seperti ini:

```
1 var i = 10000;
2
3 for (var i = 1; i < 3; i++) {
4   console.log("Belajar JavaScript " + i);
5 }
6 console.log("Harga barang = " + i); // Harga barang = 3
```

Sebelum perulangan, saya mengisi variabel `i` dengan nilai 10000. Bisa jadi ini berasal dari hasil perhitungan kode program diatasnya.

Kemudian saya membuat perulangan, yang kebetulan juga menggunakan variabel `i`. Efeknya, setelah perulangan selesai, nilai dari variabel `i` akan tertimpa.

Jika menggunakan `let`, kode program kita akan lebih aman:

```
1 var i = 10000;
2
3 for (let i = 1; i < 3; i++) {
4   console.log("Belajar JavaScript " + i);
5 }
6 console.log("Harga barang = " + i); // Harga barang = 10000
```

Sekarang, walaupun saya tidak sengaja menggunakan nama variabel yang sama untuk perulangan, kedua variabel ini berbeda scope, sehingga tidak berpengaruh satu sama lain.

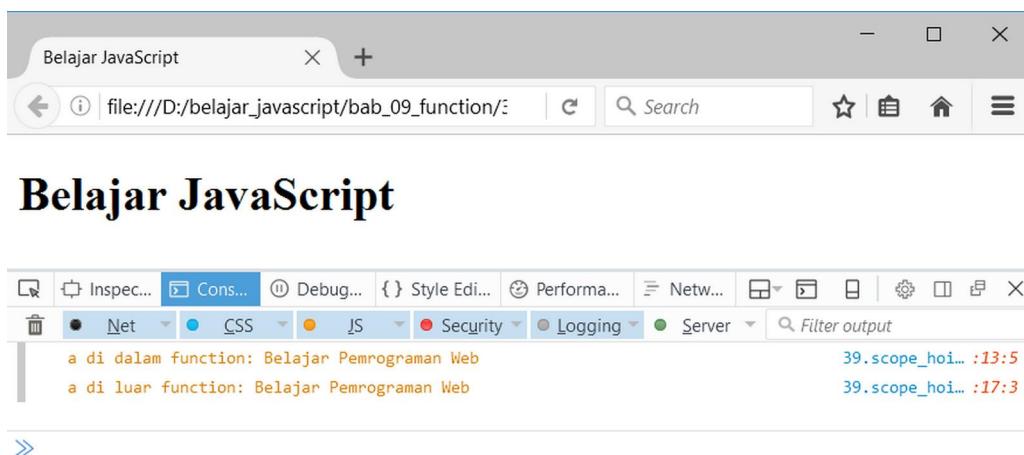
Kesimpulannya, semakin kecil ruang lingkup sebuah variabel, semakin bagus, agar kita terhindar dari kesalahan seperti diatas. Jika anda membuat kode program secara kolaborasi dengan tim, kemungkinan variabel yang bentrok akan lebih besar. Menggunakan `let` sebagai pengganti `var` bisa memperkecil terjadinya masalah ini.

## 9.10 JavaScript Hoisting

**JavaScript hoisting** adalah hal yang unik dari JavaScript, yang jika tidak diketahui bisa menjadi bug dan mendatangkan error. **JavaScript hoisting** berkaitan dengan cara JavaScript mengeksekusi kode program, dimana JavaScript akan “mengangkat” pendefinisian variabel dan function ke baris paling atas kode program.

Agar lebih mudah, saya akan membahasnya secara bertahap. Silahkan anda pelajari sebentar kode program berikut ini:

```
1 var a = "Belajar Pemrograman Web";
2
3 function foo(){
4     console.log("a di dalam function: " + a);
5 }
6
7 foo();
8 console.log("a di luar function: " + a);
```

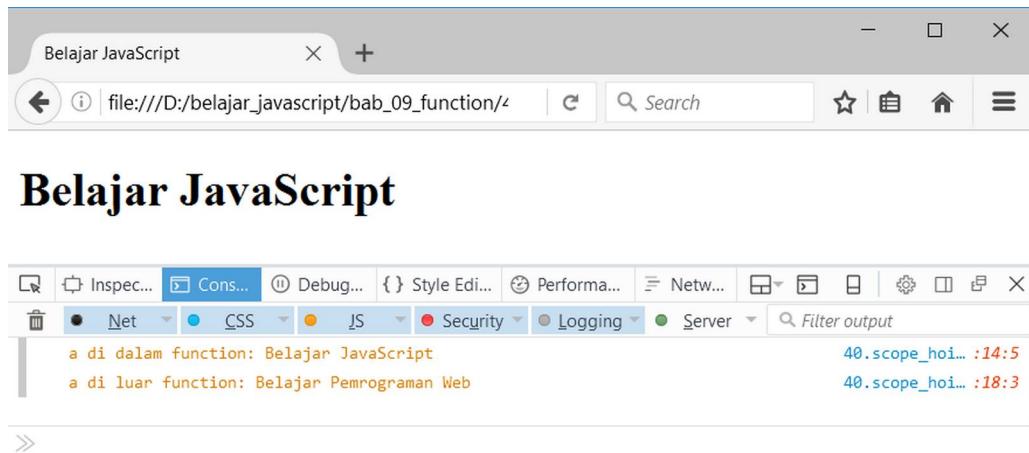


Gambar: Hasil variabel a di dalam dan di luar function

Di awal program saya membuat variabel a yang berisi string "Belajar Pemrograman Web". Karena di definisikan di luar function, variabel a akan menjadi *global variable*, artinya bisa diakses di dalam maupun di luar function. Tidak ada masalah.

Mari lanjut ke contoh kedua:

```
1 var a = "Belajar Pemrograman Web";
2
3 function foo(){
4     var a = "Belajar JavaScript";
5     console.log("a di dalam function: " + a);
6 }
7
8 foo();
9 console.log("a di luar function: " + a);
```



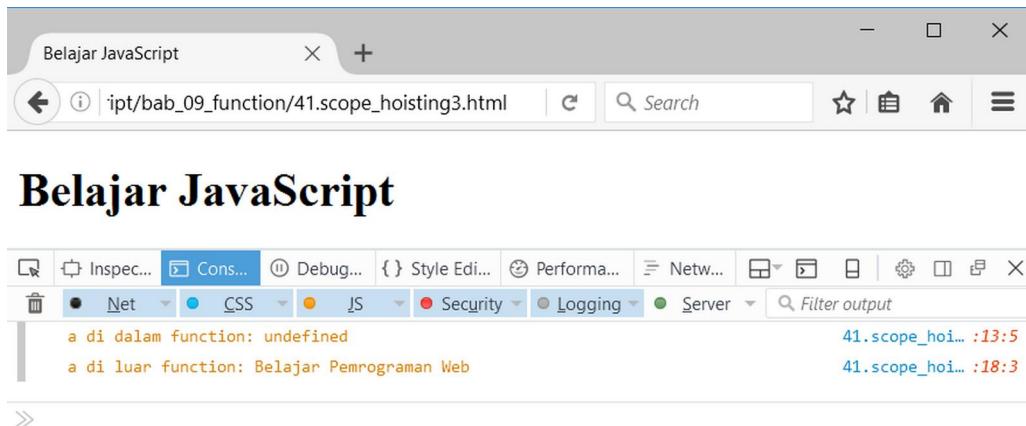
Gambar: Variabel a di definisikan ulang dari dalam function

Disini saya menambahkan perintah `var a = "Belajar JavaScript"` di dalam function `foo()`. Jika anda mengikuti pembahasan tentang *variable scope*, ini artinya saya membuat variabel a baru di dalam function, yang sepenuhnya berbeda dengan variabel a yang ada diluar function.

Dengan kata lain, variabel a di dalam function berisi string "Belajar JavaScript", ini merupakan **local variable**. Sedangkan variabel a diluar function berisi "Belajar Pemrograman Web", ini merupakan **global variable**.

Hasil tampilan kode program diatas masih sesuai dengan prediksi. Tapi bagaimana dengan kode program berikut?

```
1 var a = "Belajar Pemrograman Web";
2
3 function foo(){
4     console.log("a di dalam function: " + a); // undefined ???
5     var a = "Belajar JavaScript";
6 }
7
8 foo();
9 console.log("a di luar function: " + a);
```



Gambar: Variabel a menjadi *undefined* ??

Disini saya mengubah posisi penulisan variabel a di dalam function foo(), letaknya saya tukar antara baris `console.log()` dengan `var a = "Belajar JavaScript"`. Pertanyaannya, kenapa variabel a di `console.log()` tidak terdefinisi?

Di dalam function foo(), pada baris pertama saya langsung mengakses variabel a. Tapi karena belum ada instruksi pembuatan *local variable* a, maka seharusnya a disini berisi *global variable* a, yakni "Belajar Pemrograman Web", tapi ternyata isinya *undefined*, apa yang terjadi?

Kondisi inilah yang dinamakan **JavaScript hoisting**. Dimana ketika sebuah variable di definisikan, variabel tersebut akan "dikerek" (bahasa inggris: *hoist*) ke atas.

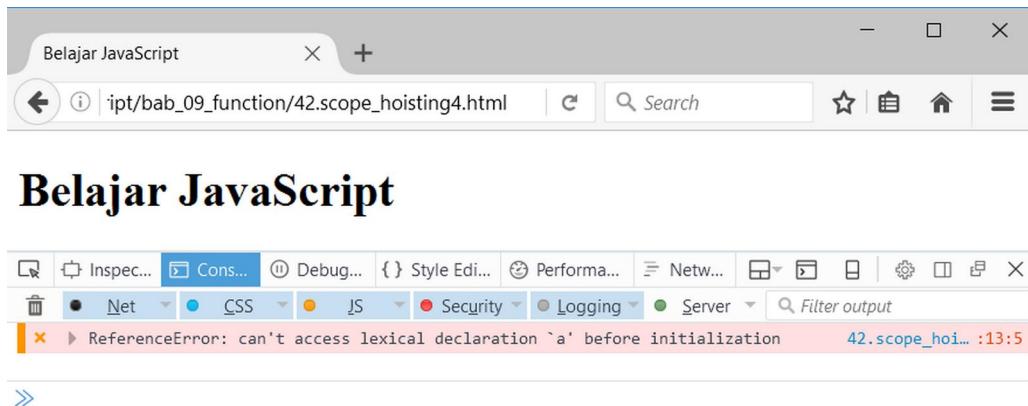
Yang akan diproses oleh JavaScript di dalam function foo() adalah seperti ini:

```
1 function foo(){
2   var a;
3   console.log("a di dalam function: " + a); // undefined ???
4   a = "Belajar JavaScript";
5 }
```

Jika kodenya ditulis seperti ini, kita bisa memahami kenapa variabel a menjadi *undefined*.

Situasi diatas sering menjadi *bug* dan membuat bingung, karena JavaScript tidak mengeluarkan error apapun. Jika saya ganti pendefenisian variabel menggunakan `let`, hasilnya akan error:

```
1 var a = "Belajar Pemrograman Web";
2
3 function foo(){
4   console.log("a di dalam function: " + a); // Error !!!
5   let a = "Belajar JavaScript";
6 }
7
8 foo();
9 console.log("a di luar function: " + a);
```



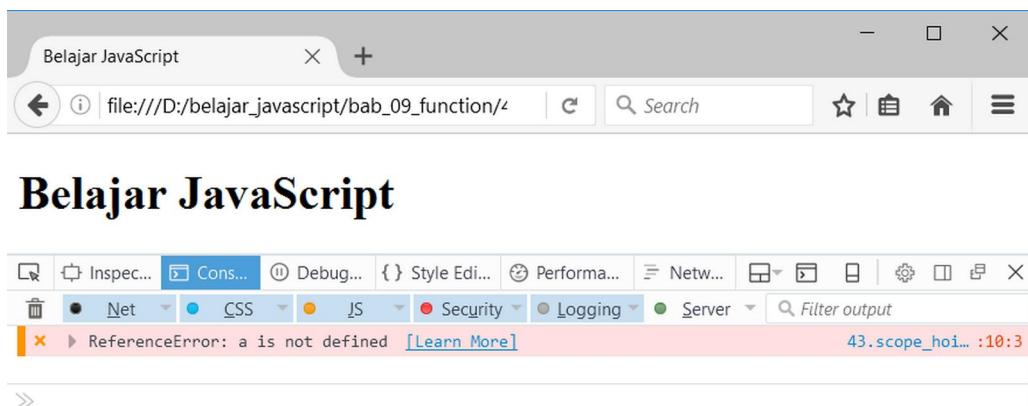
Gambar: JavaScript error karena saya mengakses sebuah variabel yang belum di deklarasikan

Error disini sangat bagus bagi kita sebagai programmer, karena bisa mendapat informasi bahwa ada sesuatu yang salah. Jika menggunakan `var` memang tidak tampil error, tapi bisa sangat berbahaya karena kita tidak tau ada yang salah di kode program.

Ini juga yang menjadi alasan banyak tutorial atau buku JavaScript terbaru yang menyarankan menggunakan `let` daripada `var` dalam pendefenisian variabel JavaScript. Namun memang perintah `let` baru tersedia di ECMAScript 6.

Efek yang sama juga berlaku di luar function. Jika kita mengakses sebuah variabel yang belum terdefenisi sama sekali, akan menjadi error:

```
1 console.log(a); // ReferenceError: a is not defined
```



Gambar: ReferenceError: a is not defined, karena variabel a tidak terdefensi

Tapi kalau terdapat defenisi variabel di bagian manapun (walau ratusan baris di bawahnya), hasilnya akan berbeda:

```
1 console.log(a); // Tidak error lagi, tapi undefined
2 var a = "Hadir bro...";
```



Gambar: Tidak error lagi, tapi variabel a sekarang menjadi undefined

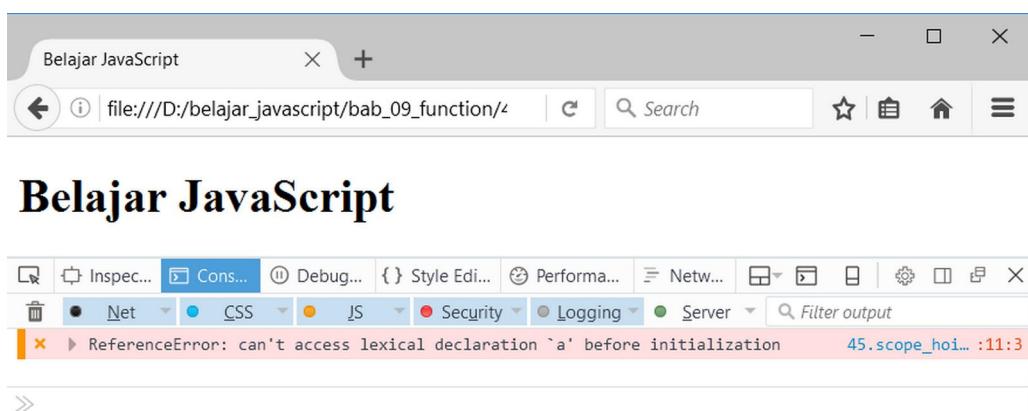
Fitur **hoisting** ini sendiri hanya “mengangkat” pendeklarasian variabel saja, bukan defenisinya (variabel a belum berisi nilai apapun). Kode program diatas akan di proses oleh JavaScript sebagai berikut:

```
1 var a;
2 console.log(a); // Tidak error lagi, tapi undefined
3 a = "Hadir bro...";
```

Variabel a sendiri baru akan terisi di posisinya saat di defenisikan.

Jika variabel a di deklarasikan menggunakan keyword `let`, hasilnya akan error, bukan undefined:

```
1 // ReferenceError: can't access lexical declaration `a` before initialization
2 console.log(a);
3 let a = "Hadir bro...";
```



Gambar: Efek hoisting akan menjadi error jika menggunakan `let`

Cara paling efektif untuk menghindari efek JavaScript **hoisting** ini adalah, selalu mendefenisikan variabel di awal kode program.

## 9.11 Function: First-class Citizen

Hal yang cukup unik dan jarang ditemui dalam bahasa pemrograman lain adalah di dalam JavaScript, **function dianggap sebagai tipe data**. Artinya, function bisa disimpan ke dalam variabel, serta bisa digunakan sebagai argumen sebagaimana layaknya tipe data biasa.

Istilah programmingnya, function di dalam JavaScript disebut sebagai **First-class Citizen**<sup>2</sup>.

Mari kita bahas menggunakan contoh kode program. Silahkan anda pelajari kode program berikut ini:

```
1 function rata2(a,b) {  
2     return (a + b)/2;  
3 }  
4  
5 console.log(rata2(4,8)); // 6
```

Saya mendefenisikan fungsi `rata2()` dengan 2 argumen: `a` dan `b`. Hasil akhir dari fungsi ini adalah rata-rata dari kedua nilai tersebut. Caranya, argumen `a` ditambah dengan `b`, kemudian dibagi dua. Jika saya memanggil `rata2(4,8)`, hasilnya 6. Tidak ada hal yang baru disini.

Sekarang, saya bisa mengisi sebuah variabel dengan fungsi `rata2()`:

```
1 var hitung = function rata2(a,b) {  
2     return (a + b)/2;  
3 };  
4  
5 console.log(hitung(4,8)); // 6
```

Perhatikan baris pertama kode program ini. Saya mendefenisikan sebuah variabel `hitung` yang diisi dengan defenisi function `rata2()`. Yang diisi adalah pendefenisian fungsi, bukan hasil pemanggilan fungsi. Jika anda terbiasa dengan bahasa pemrograman seperti PHP, mengisi sebuah variabel dengan function terasa sangat aneh.

Hasilnya, variabel `hitung` seolah-olah menjadi function `rata2()`. Saya bisa memanggil function `hitung(4,8)`, yang sama artinya dengan `rata2(4,8)`.

Dalam istilah pemrograman JavaScript, fitur ini dikenal sebagai **Function Expressions**. Sedangkan mendefenisikan fungsi secara normal (seperti mayoritas fungsi yang kita buat dalam bab ini), disebut sebagai **Function Declarations**. *Function Expression* nantinya banyak digunakan untuk membuat **JavaScript Object**.

Kode program diatas bisa juga saya tulis ulang sebagai berikut:

---

<sup>2</sup>[https://en.wikipedia.org/wiki/First-class\\_citizen](https://en.wikipedia.org/wiki/First-class_citizen)

```

1 var hitung = function rata2(a,b) { return (a + b)/2; };
2
3 console.log(hitung(4,8)); // 6

```

Kode yang digunakan tidak berbeda dari sebelumnya, hanya saja pendefenisian fungsi `rata()` saya buat dalam 1 baris agar terlihat seperti cara kita mengisi tipe data “normal”. Karena di sini `function rata2()` berperan sebagai pengganti “tipe data”, baris ini juga harus ditutup dengan tanda titik koma.

Jika diperhatikan lagi, fungsi diatas memiliki 2 nama, yakni `hitung()` dan `rata2()`. Namun ketika kita memanggil fungsi `rata2()`, hasilnya akan error:

```

1 var hitung = function rata2(a,b) { return (a + b)/2; };
2
3 console.log(hitung(4,8)); // 6
4 console.log(rata2(4,8)); // ReferenceError: rata2 is not defined

```

Karena itu, JavaScript membolehkan kita menghapus nama `rata2()`, menjadi seperti berikut:

```

1 var hitung = function (a,b) { return (a + b)/2; };
2
3 console.log(hitung(4,8)); // 6

```

Cara penulisan fungsi “tanpa nama” seperti ini dikenal sebagai **Anonymous Functions**. Di dalam library JavaScript seperti `jQuery`, cara penulisan ini sering dilakukan.

Fitur kedua dari function sebagai *first-class citizen* adalah, kita bisa mengirim function sebagai argumen. Konsep ini sedikit sulit dijelaskan, dan akan saya bahas secara bertahap.

Silahkan anda pelajari kode program berikut:

```

1 function rata2(a,b) {
2   return (a + b)/2;
3 }
4
5 function tambah(c,d) {
6   return c + d;
7 }
8
9 var hasil = tambah(6, rata2(4,8));
10 console.log(hasil); // 12

```

Saya membuat 1 lagi fungsi, yakni `tambah()`. Isi dari fungsi `tambah()` berupa penambahan 2 variabel. Ketika saya memanggil `tambah(6, rata2(4,8))`, 2 fungsi ini akan dijalankan.

Oleh karena di dalam pemanggilan ini terdapat fungsi `rata2(4,8)`, maka ini dulu yang dihitung. Hasil dari `rata2(4,8) = 6`, nilai 6 ini dikembalikan ke dalam perintah `tambah(6, rata2(4,8))`, sehingga menjadi `tambah(6, 6)`, hasilnya adalah 12.

Alur kode program diatas merupakan alur “normal”, namun terkesan sedikit rumit karena saya memanggil function dari dalam argument. Ini juga bukan yang dimaksud sebagai “mengirim function sebagai argumen”, karena yang dikirim tetap hasil dari pemanggilan function `rata2(4,8)`, bukan function itu sendiri.

Yang saya maksud “mengirim function sebagai argumen” adalah sebagai berikut:

```
1 function rata2(a,b) {  
2     return (a + b)/2;  
3 }  
4  
5 function tambah(c,d) {  
6     return c + d(4,8);  
7 }  
8  
9 var hasil = tambah(6, rata2);  
10 console.log(hasil); // 12
```

Perhatikan cara pemanggilan fungsi `tambah()`, saya menulis:

```
var hasil = tambah(6, rata2);
```

Argument kedua dari fungsi ini adalah sebuah **function!** Mari kita bahas.

Saat saya memanggil fungsi `tambah(6, rata2)`, nilai 6 dan `rata2` akan dikirim ke argument `c` dan `d` (perhatikan kembali pendefenisian fungsi `tambah()`). Oleh karena untuk argument `d` yang dikirim berupa sebuah function, cara penulisannya tanpa menggunakan tanda kurung, tapi cukup nama fungsi itu saja: `rata2`.

Di dalam fungsi `tambah()`, variabel `d` berprilaku sebagaimana layaknya fungsi `rata2()`. Saya menulis `d(4,8)`, artinya sama dengan `rata2(4,8)`. Hanya saja kali ini diwakilkan oleh variabel `d`.

Hasil dari `d(4,8)` adalah 6, dengan demikian yang dijalankan dari fungsi `tambah(6, rata2)` adalah `tambah(6, 6)`, hasilnya 12.

Jika anda masih kurang paham, mari kita coba contoh selanjutnya:

```
1 function foo(apo) {  
2     alert(apo);  
3 }  
4  
5 function salam(bar) {  
6     bar("Selamat Malam");  
7 }  
8  
9 salam(foo);
```

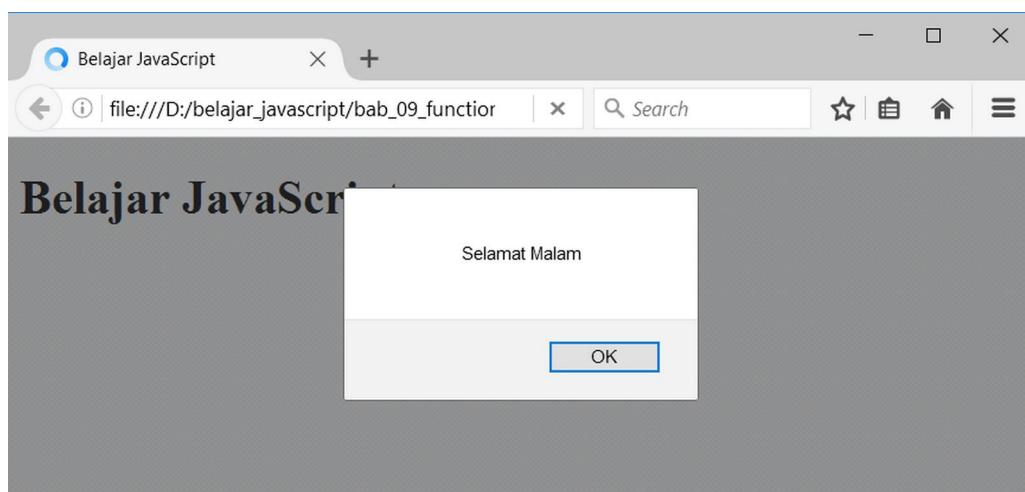
Saya akan mulai dari pemanggilan fungsi `salam(foo)`. Ketika melihat kode ini, Anda mungkin beranggapan bahwa argument dari fungsi `salam` adalah sebuah variabel bernama `foo`. Dari kode program terlihat tidak ada variabel `foo`, yang ada adalah sebuah fungsi `foo()`.

Oleh karena itu bisa diambil kesimpulan, perintah `salam(foo)` sedang mengirim function `foo()` sebagai argument. Baik, kita masuk ke defenisi function `salam()`.

Fungsi `salam()` di defenisikan menerima 1 argumen, yang ditampung oleh variabel `bar`. Pada saat fungsi `salam(foo)` dijalankan, fungsi `foo` akan diisi kedalam variabel `bar`. Di dalam fungsi `salam()`, variabel `bar` sama artinya dengan fungsi `foo()`. Dengan demikian, `bar("Selamat Malam")` sama artinya dengan `foo("Selamat Malam")`.

Apa yang dijalankan dari `foo("Selamat Malam")`? Ternyata fungsi `foo()` menerima 1 argumen, yang disimpan ke dalam variabel apa. Hasil fungsi ini adalah `alert(apa)`. Sehingga jika dipanggil `foo("Selamat Malam")`, akan dijalankan `alert("Selamat Malam")`.

Hasilnya, di web browser tampil jendela alert, dengan string "Selamat Malam".



Gambar: Hasil dari pemanggilan fungsi `salam(foo)`

Mudah-mudahan anda bisa memahami penjelasan tentang “mengirim function sebagai argumen” di JavaScript. Jika masih kurang paham, silahkan baca beberapa kali secara perlahan. Saya sendiri butuh waktu cukup lama mencerna alur program logika seperti ini.

## 9.12 Arrow Notation

**Arrow Notation** juga merupakan fitur baru di **ECMAScript 6**. Ini merupakan alternatif penulisan dari *function expression*. Disebut sebagai *arrow notation*, karena kita menggunakan tanda panah (*arrow*)  $\Rightarrow$  untuk membuat function.

Berikut contoh penggunaannya:

```

1 var pagi = function () { return "Selamat Pagi"; };
2
3 var pagiJuga = () => { return "Selamat Pagi"; };
4
5 console.log( pagi() );           // Selamat Pagi
6 console.log( pagiJuga() );      // Selamat Pagi

```

Saya membuat 2 buah variabel, yakni `pagi` dan `pagiJuga`. Kedua variabel ini digunakan untuk menampung *function expression*, yang berarti keduanya akan menjadi sebuah *function*.

Untuk variabel `pagi`, saya mengisinya menggunakan *function expression* seperti yang sudah kita pelajari sebelumnya. Disini saya menggunakan *anonymous functions*, dimana pada saat pendefenisian fungsi, fungsi ini tidak bernama, tapi langsung dibuat tanda kurung (sebagai tempat untuk argument).

Untuk variabel `pagiJuga`, saya menggunakan **arrow notation**. Perhatikan cara penulisannya, kita tidak perlu menulis keyword *function*, tapi cukup tanda kurung (), kemudian diikuti tanda panah =>. Setelah itu, kurung kurawal menandakan dimulainya *block function* seperti fungsi reguler.

Terlihat kedua fungsi ini identik. Saya bisa menjalankannya dengan perintah `pagi()` dan `pagiJuga()`. Keduanya akan mengembalikan string "Selamat Pagi".

Bagaimana jika menggunakan argument? Berikut contohnya:

```

1 var pagi = function(siapa) { return "Selamat Pagi "+ siapa; };
2
3 var pagiJuga = (siapa) => { return "Selamat Pagi "+ siapa; };
4
5 console.log( pagi("Indonesia") );           // Selamat Pagi Indonesia
6 console.log( pagiJuga("Jakarta") );         // Selamat Pagi Jakarta

```

Kedua fungsi ini juga identik. Untuk tempat argument, kita tinggal mengisinya diantara tanda kurung sebelum tanda panah. Di dalam *block function*, argument ini bisa diakses seperti biasa.

Berikut contoh lainnya:

```

1 var total = function(a,b,c) { return a+b+c; };
2
3 var totalJuga = (a,b,c) => { return a+b+c; };
4
5 console.log( total(4,5,6) );           // 15
6 console.log( totalJuga(7,8,9) );       // 24

```

Disini saya membuat fungsi `total` dan `totalJuga`. Keduanya menerima 3 argument, dan mengembalikan jumlah total dari ketiga argument tersebut. Terlihat, menggunakan *arrow notation* sangat mirip seperti pembuatan fungsi biasa.

Mungkin ada sempat kepikiran, buat apa repot-repot menulis function menggunakan *function expression* maupun *arrow notation*? Kenapa kita tidak menulis function seperti biasa? Toh ujung-ujungnya juga dipanggil menggunakan nama function.

Penulisan *function expression* akan banyak dipakai jika kita sudah masuk ke **JavaScript Object**, dan inilah yang akan kita bahas pada bab selanjutnya.

---

Dalam bab ini kita telah membahas panjang lebar tentang **function** di dalam JavaScript. Mulai dari cara pembuatannya, mengenal *argument*, *variable scope*, hingga melihat function sebagai *First-class citizen* dan penulisan *arrow notation*.

Berikutnya kita akan masuk ke pembahasan yang tidak kalah penting, yakni tentang **JavaScript Object**.

# 10. JavaScript Object

JavaScript merupakan bahasa pemrograman berbasis object. Sejak awal pembahasan buku ini sebenarnya kita telah menggunakan berbagai konsep terkait object. Fungsi `alert()` merupakan *method* dari **window object**. Fungsi `console.log()` merupakan cara pemanggilan *method* `log()` dari **console object**. Array pun termasuk ke dalam **object**.

Dalam bab ini kita akan mempelajari tentang **Object** di dalam JavaScript, membahas pengertian *property* dan *method*, serta bagaimana cara pembuatannya.

## 10.1 Apa itu Object?

Di dalam JavaScript, **object** sangat mirip seperti array, yakni sebuah tipe data bentukan yang terdiri dari kumpulan tipe data lain. Object bisa diisi data *string*, *number*, *boolean*, atau bahkan object lain. Berbeda dengan array, object bisa memiliki function sendiri.

**Object** juga bisa disebut sebagai **container**, yakni wadah yang digunakan untuk menampung berbagai data.

Dalam bahasa pemrograman PHP terdapat konsep *associative array*, yaitu array yang key-nya bisa diisi string (tidak hanya angka saja sebagaimana layaknya array biasa). Konsep *associative array* PHP ini sangat mirip seperti object di dalam JavaScript.

## 10.2 Format Dasar Object

Jika sebelumnya anda sudah pernah belajar bahasa pemrograman berbasis object seperti **JAVA** atau **OOP PHP**, object di JavaScript terasa cukup aneh. Hingga **ECMAScript 5**, JavaScript disebut sebagai *classless programming language*. Dimana untuk membuat object, kita tidak perlu membuat class terlebih dahulu.



Agar anda bisa memahami object di JavaScript, silahkan “lupakan” sejenak teori Object dari bahasa pemrograman lain.

JavaScript menggunakan konsep **prototypical inheritance** untuk menerapkan konsep pemrograman berbasis object. Saya akan membahas *prototypical inheritance* ini dalam bab berikutnya. Saat ini anda cukup memahami bahwa untuk membuat object di JavaScript, caranya adalah dengan langsung menulis object tersebut (tidak perlu membuat *class*).

Seperti yang saya singgung sebelumnya, object di dalam JavaScript sangat mirip seperti array, dan cara pembuatannya pun hampir sama.

Berikut format dasar pembuatan object di JavaScript:

```
var namaObject = {
    property1 : "isi_property1",
    property2 : "isi_property2",
    property3 : "isi_property3",

    method1 : function (){
        "isi method 1";
    },

    method2 : function (){
        "isi method 2";
    }
}
```

Pada baris pertama, saya membuat variabel **namaObject**, variabel inilah yang akan diisi dengan data object. Aturan nama object sama seperti *identifier* lain, yakni tidak boleh diawali dengan angka, tidak boleh mengandung spasi, dst.

Block kode program untuk pendefenisian object diawali tanda kurung kurawal. Disinilah ‘isi’ dari object kita tulis.

Dalam pendefenisian object, terdapat istilah **property** dan **method**. **Property** adalah sebutan untuk variabel yang berada di dalam object. Sedangkan **method** adalah function yang ditempatkan ke dalam object.

Baik *property* maupun *method* diberi nilai menggunakan tanda titik dua, bukan tanda sama dengan sebagaimana layaknya pengisian variabel biasa (operasi *assignment*). Diantara *property* yang satu dengan yang lain, dipisah menggunakan tanda koma.

Sebenarnya, isi object ini (*property* dan *method*) adalah opsional, sehingga boleh tidak ditulis. Berikut contohnya:

```
1 var foo = {};
2 console.log(typeof foo); // object
```

Yup, dengan menulis tanda kurung kurawal kosong, kita sudah bisa membuat object JavaScript. Hasil dari operator **typeof** menyatakan bahwa variabel **foo** adalah object. Tapi tentu saja agar bisa bermanfaat, kita harus mengisi object ini dengan *property* dan *method*.

## 10.3 Object Property

**Property** adalah sebutan untuk variabel yang diletakkan ke dalam object. Berikut contoh pembuatannya:

```
1 var mahasiswa = {  
2     nama: "Andi",  
3     jurusan: "Teknik Informatika",  
4     ipk : 3.67,  
5     semester: 4  
6 };
```

Disini saya membuat variabel **mahasiswa** yang akan diisi dengan data-data mahasiswa. Object **mahasiswa** terdiri dari 4 property: **nama**, **jurusan**, **ipk**, dan **semester**. Setiap property diisi dengan berbagai data. Diantara property satu dengan yang lain dipisah dengan tanda koma.

Tanda titik koma diakhir penulisan object juga sebaiknya tetap ditulis (setelah kurung kurawal penutup). Karena proses pembuatan object sama seperti pengisian variabel biasa, jadi butuh diakhiri dengan tanda titik koma.

Penulisan memanjang kebawah seperti diatas bukan sebuah keharusan. Kita juga bisa membuatnya dalam 1 baris saja:

```
1 var mahasiswa = { nama: "Andi", jurusan: "Teknik Informatika",  
2                     ipk : 3.67, semester: 4 };
```

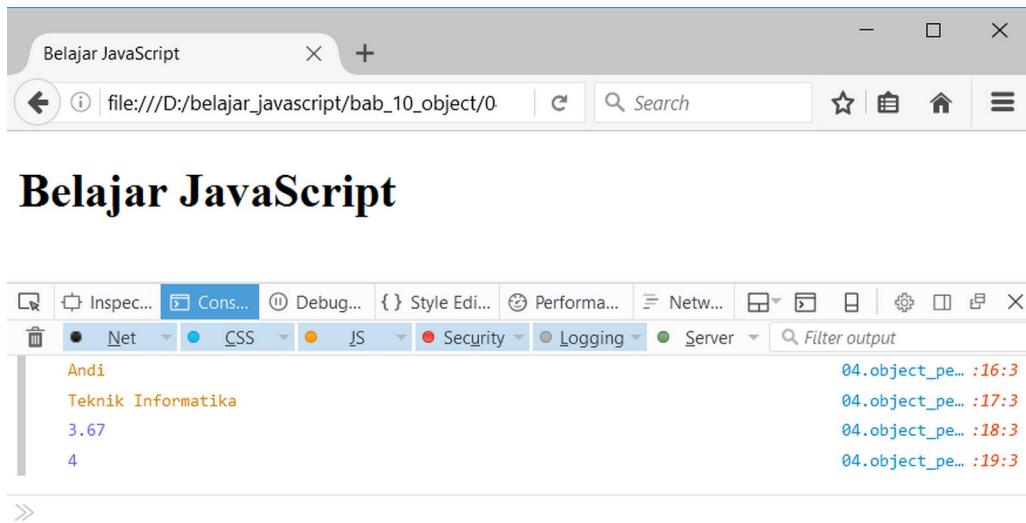
Setelah mengisi object **mahasiswa** dengan 4 property, bagaimana cara mengakses nilainya?

Terdapat 2 cara pengaksesan property (*property accessors*): menggunakan tanda titik (**dot notation**) atau menggunakan tanda kurung siku (**bracket notation**). Format dasarnya sebagai berikut:

```
object.property  
object["property"]
```

Mari kita akan lihat praktik menggunakan *dot notation* terlebih dahulu:

```
1 var mahasiswa = { nama: "Andi",  
2                     jurusan: "Teknik Informatika",  
3                     ipk : 3.67,  
4                     semester: 4 };  
5  
6 console.log( mahasiswa.nama );           // Andi  
7 console.log( mahasiswa.jurusan );        // Teknik Informatika  
8 console.log( mahasiswa.ipk );            // 3.67  
9 console.log( mahasiswa.semester );       // 4
```



Gambar: Cara pengaksesan property object JavaScript

Untuk mengakses property `nama` dari object `mahasiswa`, ditulis sebagai `mahasiswa.nama`. Pengaksesan property menggunakan dot notation merupakan cara yang paling umum dan paling sering digunakan di dalam JavaScript.

Cara kedua adalah menggunakan *bracket notation*:

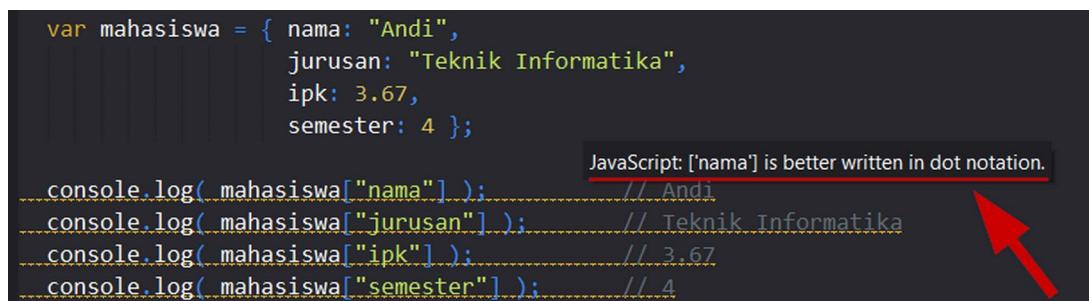
```

1 var mahasiswa = { nama: "Andi",
2                     jurusan: "Teknik Informatika",
3                     ipk: 3.67,
4                     semester: 4 };
5
6 console.log( mahasiswa["nama"] );           // Andi
7 console.log( mahasiswa["jurusan"] );         // Teknik Informatika
8 console.log( mahasiswa["ipk"] );             // 3.67
9 console.log( mahasiswa["semester"] );         // 4

```

Kali ini untuk mengakses property `nama` dari object `mahasiswa`, ditulis dengan `mahasiswa["nama"]`. Tanda kutip perlu ditambahkan karena pada dasarnya nama property dianggap sebagai string.

Pengaksesan property menggunakan *bracket notation* tidak terlalu sering digunakan. Umumnya *bracket notation* kita pakai dalam mengakses element array. Teks editor Komodo Edit juga akan menyarankan kita menggunakan dot notation untuk mengakses property sebuah object.

Gambar: Saran Komodo Edit untuk menggunakan *dot notation* daripada *bracket notation*

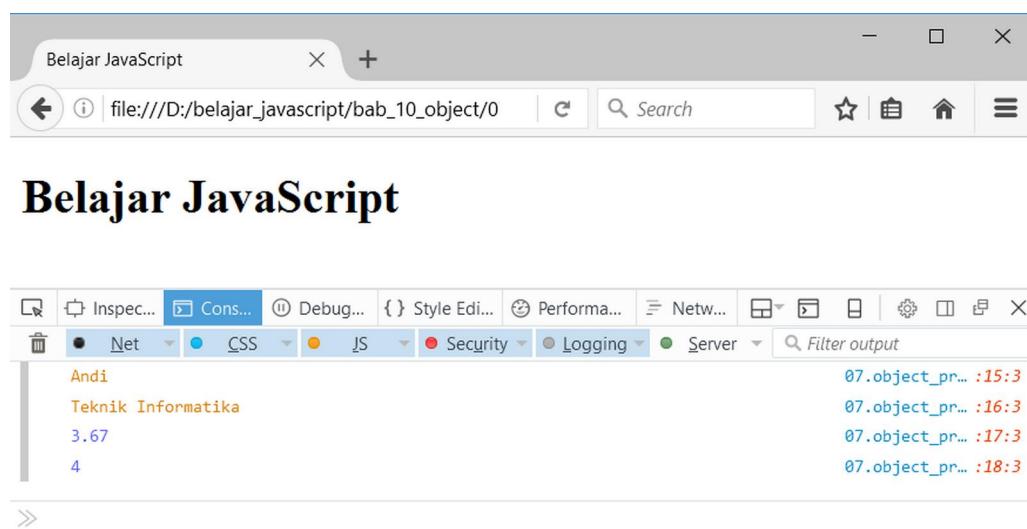
Akan tetapi, penulisan dengan *bracket notation* memiliki kelebihan lain. Kita bisa mengakses property yang namanya tidak umum, seperti mengandung spasi atau karakter khusus seperti angka (akan saya bahas sesaat lagi).

Mengakses property dengan *bracket notation* harus ditulis dengan tanda kutip. Ini karena pada dasarnya nama property object diproses JavaScript sebagai sebuah string. Object mahasiswa bisa juga saya tulis sebagai berikut:

```
1 var mahasiswa = { "nama": "Andi",
2                     "jurusan": "Teknik Informatika",
3                     "ipk": 3.67,
4                     "semester": 4 };
```

Perhatikan kali ini saya menuliskan nama property dengan menggunakan tanda kutip: "nama": "Andi". Penulisan property seperti ini tetap bisa diakses menggunakan *dot notation* maupun *bracket notation*:

```
1 var mahasiswa = { "nama": "Andi",
2                     "jurusan": "Teknik Informatika",
3                     "ipk": 3.67,
4                     "semester": 4 };
5
6 console.log( mahasiswa.nama );           // Andi
7 console.log( mahasiswa.jurusan );        // Teknik Informatika
8 console.log( mahasiswa["ipk"] );         // 3.67
9 console.log( mahasiswa["semester"] );     // 4
```



Gambar: Pengaksesan property bisa dilakukan dengan dot notation maupun bracket notation

Hal yang cukup unik adalah, kita bisa membuat nama property yang tidak memenuhi syarat *identifier*. Misalnya property yang mengandung spasi seperti contoh berikut:

```

1 var mahasiswa = { "nama lengkap": "Andi Wibowo",
2                     "jurusan kuliah": "Teknik Informatika",
3                     "nilai ipk": 3.67,
4                     "semester terakhir": 4 };

```

Syaratnya, property tersebut harus ditulis menggunakan tanda kutip (sebagai string).

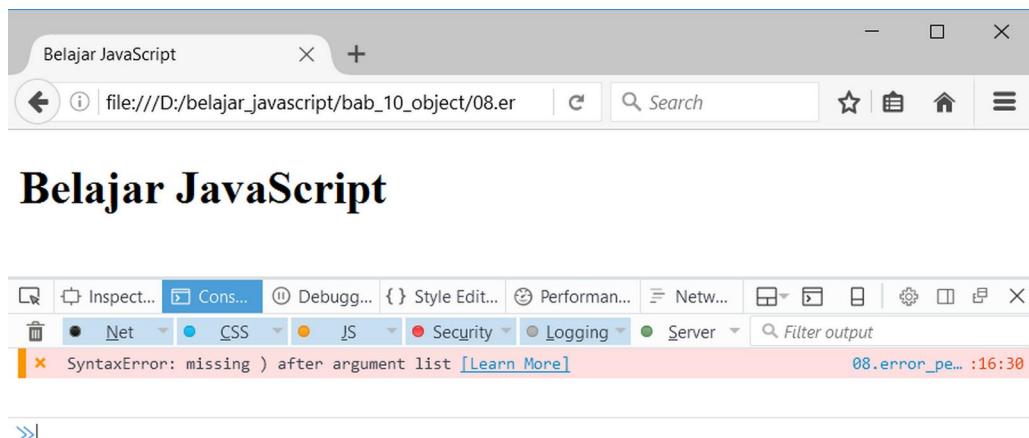
Penulisan seperti ini memang tidak disarankan tapi bisa. Mari kita coba akses:

```

1 var mahasiswa = { "nama lengkap": "Andi Wibowo",
2                     "jurusan kuliah": "Teknik Informatika",
3                     "nilai ipk": 3.67,
4                     "semester terakhir": 4 };
5
6 // SyntaxError: missing ) after argument list
7 console.log( mahasiswa.nama lengkap );

```

Hasilnya terdapat error! Kenapa bisa seperti ini?



Gambar: Error saat mengakses object property

Inilah salah satu kelemahan penggunaan *dot notation* untuk mengakses object property. Untuk property yang tidak umum (seperti memiliki spasi), kita hanya bisa menggunakan *bracket notation*:

```

1 var mahasiswa = { "nama lengkap": "Andi Wibowo",
2                     "jurusan kuliah": "Teknik Informatika",
3                     "nilai ipk": 3.67,
4                     "semester terakhir": 4 };
5
6 console.log( mahasiswa["nama lengkap"] );           // Andi Wibowo
7 console.log( mahasiswa["jurusan kuliah"] );         // Teknik Informatika
8 console.log( mahasiswa["nilai ipk"]);                // 3.67
9 console.log( mahasiswa["semester terakhir"]);        // 4

```

Object property juga bisa ditulis menggunakan angka, yang sebenarnya juga tidak memenuhi syarat sebuah *identifier*:

```

1 var absensi = { 1: "Andi",
2                 2: "Alex",
3                 3: "Joni",
4                 4: "Siska" };
5
6 console.log( absensi[1] );           // Andi
7 console.log( absensi[2]);           // Alex
8 console.log( absensi["3"]);          // Joni
9 console.log( absensi["4"]);          // Siska

```

Sama seperti sebelumnya, property angka tetap dianggap sebagai string oleh JavaScript. Object **absensi** diatas bisa saya tulis ulang sebagai berikut:

```

1 var absensi = { "1": "Andi",
2                 "2": "Alex",
3                 "3": "Joni",
4                 "4": "Siska" };
5
6 console.log( absensi[1] );           // Andi
7 console.log( absensi[2]);           // Alex
8 console.log( absensi["3"]);          // Joni
9 console.log( absensi["4"]);          // Siska

```

Dapat dilihat, meskipun property ditulis sebagai string, pada saat diakses saya bisa langsung menggunakan angka (sama seperti cara pengaksesan element array). Pengaksesan property **absensi[1]**, bisa juga ditulis dengan **absensi["1"]**.

Property yang ditulis menggunakan angka seperti ini juga tidak bisa diakses menggunakan *dot notation*. *Dot notation* hanya bisa digunakan untuk mengakses property yang memenuhi syarat *identifier*, yakni property yang tidak boleh mengandung spasi dan tidak boleh berupa angka.

```

1 var absensi = { "1": "Andi",
2                 "2": "Alex",
3                 "3": "Joni",
4                 "4": "Siska" };
5
6 console.log( absensi[1] );           // Andi
7 console.log( absensi["1"] );          // Andi
8 console.log( absensi.1 );            // SyntaxError: missing ) after argument list

```

Error diatas terjadi karena saya mencoba mengakses property yang berupa angka menggunakan *dot notation*.

## 10.4 Menambah Object Property

Selain membuat langsung property pada saat pendefenisian object, kita juga bisa menambahkannya setelah object tersebut dibuat. Caranya mirip seperti mengisi array.

Mari kita lihat kembali cara membuat object mahasiswa:

```
1 var mahasiswa = { nama: "Andi",
2                     jurusan: "Teknik Informatika",
3                     ipk : 3.67,
4                     semester: 4
5                 };
```

Object diatas bisa juga saya buat seperti ini:

```
1 var mahasiswa = {};
2
3 mahasiswa.nama = "Andi";
4 mahasiswa.jurusan = "Teknik Informatika";
5 mahasiswa.ipk = 3.67;
6 mahasiswa.semester = 4;
```

Saya membuat terlebih dahulu sebuah object kosong, yakni `mahasiswa`. Pengisian property dilakukan setelah pendefenisian object. Caranya seperti seperti mengisi variabel, hanya saja kita juga menulis lengkap nama object beserta nama property yang ingin ditambahkan.

Dengan menggunakan *bracket notation*, kita juga bisa menambahkan property `mahasiswa`:

```
1 var mahasiswa = {};
2
3 mahasiswa["nama lengkap"] = "Andi";
4 mahasiswa["jurusan kuliah"] = "Teknik Informatika";
5 mahasiswa["nilai ipk"] = 3.67;
6 mahasiswa["semester terakhir"] = 4;
```

Khusus penambahan property dengan *bracket notation*, kita bisa membuat nama property yang memiliki spasi (yang akan error jika ditulis menggunakan *dot notation*).

Tentu saja kita juga bisa membuat sebagian property pada saat pendefenisian, dan sebagian lagi setelahnya:

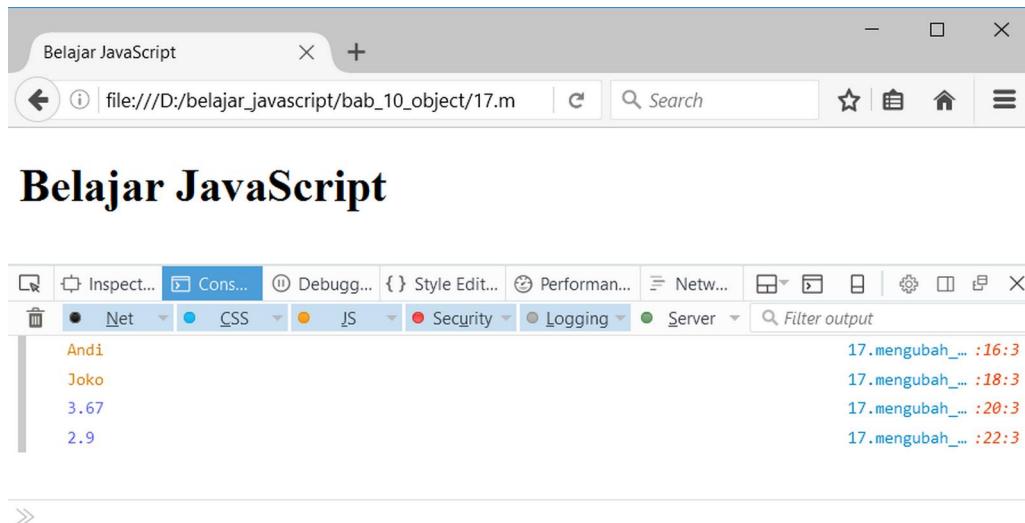
```
1 var mahasiswa = { nama: "Andi",
2                     jurusan: "Teknik Informatika" };
3
4 mahasiswa.ipk = 3.67;
5 mahasiswa.semester = 4;
6
7 console.log( mahasiswa.nama );           // Andi
8 console.log( mahasiswa.jurusan );        // Teknik Informatika
9 console.log( mahasiswa.ipk );            // 3.67
10 console.log( mahasiswa.semester );       // 4
```

## 10.5 Mengubah Nilai Object Property

Untuk mengubah nilai property dari sebuah object, bisa dengan cara menimpa property tersebut, contohnya sebagai berikut:

```
1 var mahasiswa = { nama: "Andi",
2                     jurusan: "Teknik Informatika",
3                     ipk : 3.67,
4                     semester: 4 };
5
6 console.log( mahasiswa.nama );           // Andi
7 mahasiswa.nama = "Joko";
8 console.log( mahasiswa.nama );           // Joko
9
10 console.log( mahasiswa["ipk"] );         // 3.67
11 mahasiswa["ipk"] = 2.9;
12 console.log( mahasiswa["ipk"] );          // 2.9
```

Setelah pendefenisian object `mahasiswa`, saya mengubah nilai property `nama` menjadi "Joko", dan mengubah nilai `ipk` menjadi 2.9. Terlihat bahwa kita bisa menggunakan *dot notation* maupun *bracket notation*.



Gambar: Mengubah property mahasiswa

## 10.6 Object Method

Selain *property*, di dalam object bisa terdapat **method**, yakni sebutan untuk function. Penulisannya sangat mirip seperti function biasa yang kita bahas pada bab sebelumnya, yakni menggunakan *function expressions*.

Sebagai contoh object, kali saya membuat object **mobil**:

```

1 var mobil = { merk: "Toyota Avanza",
2                 tipe: "MPV",
3                 harga: 200000000,
4                 warna: "merah" };
5
6 console.log(mobil.merk);    // Toyota Avanza
7 console.log(mobil.tipe);    // MPV
8 console.log(mobil.harga);   // 200000000
9 console.log(mobil.warna);   // merah

```

Saya membuat variabel **mobil** yang akan diisi object dengan 4 property: **merk**, **tipe**, **harga**, dan **warna**. Cara pembuatan property seperti ini sudah kita pelajari sebelumnya.

Sekarang, saya ingin menambahkan sebuah method **hidupkan()**:

```

1 var mobil = { merk: "Toyota Avanza",
2             tipe: "MPV",
3             harga: 200000000,
4             warna: "merah",
5             hidupkan: function () { return "Mesin Dihidupkan"; }
6           };
7
8 console.log( mobil.hidupkan() ); // Mesin Dihidupkan

```

Perhatikan cara pembuatan method `hidupkan()`, saya menggunakan cara penulisan *function expressions*, dimana *anonymous function* diinput ke dalam variabel **hidupkan**.

Untuk menjalankan method, caranya adalah dengan menulis nama object, tanda titik, lalu nama method yang diikuti tanda kurung (yang nantinya sebagai tempat untuk *argument*).

Bagaimana jika menggunakan *bracket notation*? Walaupun tidak umum dipakai, tapi kita bisa menjalankan method menggunakan cara tersebut:

```

1 var mobil = { merk: "Toyota Avanza",
2             tipe: "MPV",
3             harga: 200000000,
4             warna: "merah",
5             hidupkan: function () { return "Mesin Dihidupkan"; }
6           };
7
8 console.log( mobil["hidupkan"]() ); // Mesin Dihidupkan

```

Kurang lebih sama seperti *dot notation*, tapi kali ini kita menggantinya dengan *bracket notation*.

Selanjutnya, bagaimana cara menuliskan argument untuk method ini? Caranya sama seperti pendefenisian function biasa:

```

1 var mobil = { merk: "Toyota Avanza",
2             tipe: "MPV",
3             harga: 200000000,
4             warna: "merah",
5             pergi: function (tempat) { return "Pergi ke "+ tempat; }
6           };
7
8 console.log( mobil.pergi("Medan") ); // Pergi ke Medan
9 console.log( mobil["pergi"]("Amboon") ); // Pergi ke Amboon

```

Kali ini saya membuat method `pergi()` yang memiliki 1 argument: `tempat`. Argument `tempat` digunakan untuk membuat nilai kembalian: `return "Pergi ke "+ tempat`.

Selain di defenisikan di dalam object, method juga bisa diinput setelah pendefinisian object, seperti contoh berikut:

```
1 var mobil = { merk: "Toyota Avanza",
2             tipe: "MPV",
3             harga: 200000000,
4             warna: "merah" };
5
6 mobil.hidupkan = function () { return "Mesin Dihidupkan"; };
7 mobil.pergi = function (tempat) { return "Pergi ke "+ tempat; };
8
9 console.log( mobil.hidupkan() );           // Mesin Dihidupkan
10 console.log( mobil.pergi("Bali") );        // Pergi ke Bali
```

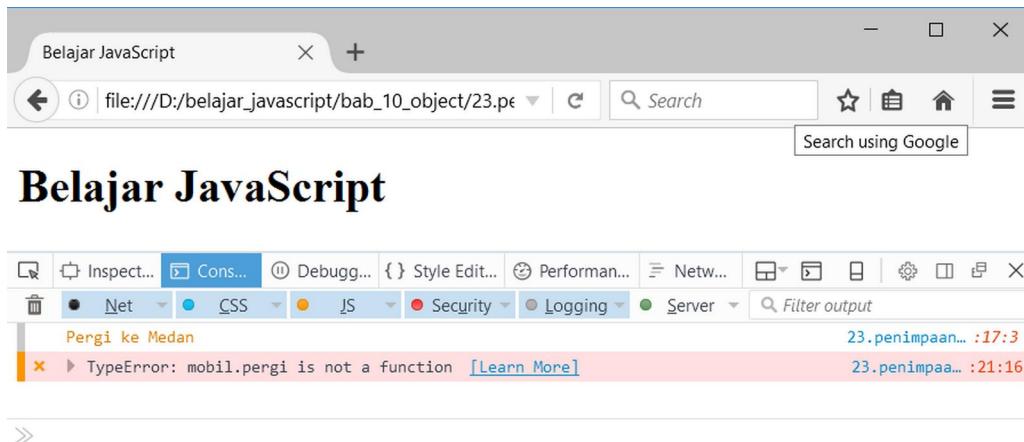
Cara pembuatannya sama seperti *object property*, dimana anonymous function kita input ke variabel yang menjadi bagian dari object **mobil**.

## 10.7 Mengubah Object Method

Menambahkan property maupun method ke dalam object JavaScript sangat mudah. Seperti yang sudah kita coba, caranya cukup dengan memberikan nilai baru ke property atau method tersebut.

Tapi, proses ini meyimpan bahaya yang bisa mengintai:

```
1 var mobil = { merk: "Toyota Avanza",
2             tipe: "MPV",
3             harga: 200000000,
4             warna: "merah",
5             pergi: function (tempat) { return "Pergi ke "+ tempat; }
6           };
7
8 console.log( mobil.pergi("Medan") );        // Pergi ke Medan
9
10 mobil.pergi = "Methodnya sudah tertimpa..";
11
12 console.log( mobil.pergi("Medan") );        // error!
13 // TypeError: mobil.pergi is not a function
```



Gambar: Method pergi() ditimpak dengan nilai baru

Dalam contoh diatas, saya mendefenisikan sebuah method pergi(). Kemduian, dengan tidak sengaja (eh..), saya menimpanya dengan perintah:

```
1 mobil.pergi = "Methodnya sudah tertimpak..";
```

Karena perintah ini, method pergi() sudah tidak bisa diakses lagi, karena saya sudah menimpa isinya dengan sebuah string. JavaScript membolehkan hal ini terjadi dan tidak menampilkan pesan error apapun. Error diatas baru tampil ketika saya mencoba mengakses method pergi() yang sudah berubah menjadi property pergi.

Jadi, kita harus berhati-hati dalam menulis *property* dan *method* object JavaScript, karena sangat mudah tertimpak dengan nilai lain.

## 10.8 Nested Object

**Nested object** adalah sebutan untuk object di dalam object. Sebagai mana kita melihat array 2 dimensi (array di dalam array), *nested object* juga seperti itu. Caranya adalah dengan mengisi nilai property dengan sebuah object lain. Struktur seperti ini diperlukan untuk membuat data yang kompleks.

Berikut contoh pembuatan **nested object** JavaScript:

```
1 var mahasiswa = { nama: "Andi",
2                     jurusan: "Teknik Informatika",
3                     ipk : {
4                         1: 3.1,
5                         2: 3.6,
6                         3: 2.7
7                     },
8                     semester: 4 };
9
10 console.log(mahasiswa.ipk[1]); // 3.1
11 console.log(mahasiswa.ipk[3]); // 2.7
```

Perhatikan property ipk, isinya berupa object lain. Disini saya memecah nilai ipk untuk semester 1, semester 2, dan semester 3.

Saya menggunakan cara pengaksesan *bracket notation* karena object ipk ini dibuat dengan angka (tidak valid untuk *identifier*). Tapi jika property tersebut berisi nama property yang valid, kita bisa menggunakan *dot notation*:

```
1 var mahasiswa = { nama: "Andi",
2                     jurusan: "Teknik Informatika",
3                     ipk : {
4                         semester1: 3.1,
5                         semester2: 3.6,
6                         semester3: 2.7
7                     },
8                     semester: 4 };
9
10 console.log(mahasiswa.ipk.semester1); // 3.1
11 console.log(mahasiswa.ipk.semester3); // 2.7
```

Penulisan `mahasiswa.ipk.semester1`, artinya saya ingin mengakses nilai property `semester1` yang berada di dalam object `ipk`, dimana object `ipk` ini juga merupakan bagian dari object `mahasiswa`.

**Nested object** nantinya akan sering kita jumpai pada saat masuk ke materi tentang DOM (Document Object Model). Berikut salah satu contohnya:

```
var header = window.document.getElementById("header");
```

Artinya saya mengakses sebuah method `getElementById()`, yang terdapat di dalam object `document`, dimana object `document` ini juga bagian dari object `window`.

## 10.9 Object Reference

**Object reference** adalah materi yang membahas bagaimana prilaku sebuah object ketika dipindahkan ke variabel lain. Ini pula yang nantinya membedakan hasil operasi perbandingan, ketika yang dibandingkan itu berupa object.

Sebelum kita membahas *object reference*, silahkan lihat sejenak kode program berikut:

```
1 var mahasiswa = "Andi";
2 var mahasiswaBaru = mahasiswa;
3
4 console.log( mahasiswa );           // Andi
5 console.log( mahasiswaBaru );       // Andi
6
7 mahasiswaBaru = "Joko";
8
9 console.log( mahasiswa );           // Andi
10 console.log( mahasiswaBaru );        // Joko
11
12 mahasiswa = "Vino";
13
14 console.log( mahasiswa );           // Vino
15 console.log( mahasiswaBaru );        // Joko
```

Saya memiliki variabel `mahasiswa` yang diisi dengan string "Andi". Kemudian terdapat operasi `var mahasiswaBaru = mahasiswa`. Perintah ini berarti saya sedang mengirim nilai variabel `mahasiswa` ke `mahasiswaBaru`. Seperti yang dapat ditebak, kedua variabel ini akan sama-sama berisi string "Andi".

Kata kunci disini adalah, **nilai** dari variabel `mahasiswa` dikirim ke `mahasiswaBaru`. Dengan demikian, ketika isi variabel `mahasiswaBaru` saya ubah menjadi "Joko", variabel `mahasiswa` tetap berisi string "Andi", karena kedua variabel ini saling terpisah dan memiliki nilai masing-masing.

Begitu pula ketika saya mengubah nilai variabel `mahasiswa` menjadi "Vino", dimana yang berubah hanya nilai dari variabel `mahasiswa` saja. Variabel `mahasiswaBaru` tetap berisi string "Joko".

Dalam istilah bahasa pemrograman, operasi `var mahasiswaBaru = mahasiswa` disebut sebagai "**assignment by value**", dimana yang dikirim ke variabel `mahasiswaBaru` adalah nilai (*value*) dari variabel `mahasiswa`. Di dalam JavaScript, seluruh tipe data primitive (*number*, *string*, dan *boolean*) menggunakan prinsip "*assignment by value*".

Khusus untuk object, hasilnya akan berbeda:

```
1 var mahasiswa = { nama: "Andi",
                     jurusan: "Teknik Informatika" };
2
3 var mahasiswaBaru = mahasiswa;
4
5 console.log( mahasiswa.nama );           // Andi
6 console.log( mahasiswaBaru.nama );       // Andi
7
8 mahasiswa.nama = "Joko";
9 console.log( mahasiswa.nama );           // Joko
10 console.log( mahasiswaBaru.nama );        // Joko
11
12
```

```
13 mahasiswaBaru.jurusan = "Ekonomi Manajemen";
14 console.log( mahasiswa.jurusan );           // Ekonomi Manajemen
15 console.log( mahasiswaBaru.jurusan );       // Ekonomi Manajemen
```

Kali ini saya memiliki variabel `mahasiswa`, yang diisi dengan object. Di dalam object terdapat 2 property: `nama` dan `jurusan`. Untuk object `mahasiswa`, saya mengisi property `nama`: "Andi", dan property `jurusan`: "Teknik Informatika".

Baris berikutnya, saya juga menjalankan perintah `var mahasiswaBaru = mahasiswa`. Tapi kali ini yang dikirim ke variabel `mahasiswaBaru` bukanlah nilai, tapi **referensi (reference)**. Operasi ini dikenal dengan istilah: "**assignment by reference**".

Artinya, sekarang variabel `mahasiswaBaru` dan `mahasiswa` merujuk ke object yang sama. Ketika saya mengubah isi property `nama` dari object `mahasiswa` menjadi "Joko", `mahasiswaBaru.nama` juga berubah menjadi "Joko". Begitu pula saat saya mengubah property `mahasiswaBaru.jurusan` = "Ekonomi Manajemen", isi dari property `jurusan` di object `mahasiswa` juga berubah.

**Assignment by reference** terjadi ketika kita menjalankan operasi *asignment* untuk tipe data object, seperti perintah `var mahasiswaBaru = mahasiswa` pada contoh diatas.

Dalam perintah tersebut yang dikirim dari variabel `mahasiswa` ke variabel `mahasiswaBaru` adalah alamat memory dari object `mahasiswa`. Alamat memory inilah yang dikenal dengan istilah **reference**.

Misalkan alamat memory ini berada di 100AAE, isinya berupa variabel `nama` dan `jurusan` yang membangun object `mahasiswa`. Alamat 100AAE ini juga yang akan diakses oleh variabel `mahasiswaBaru`, dengan demikian ketika salah satunya mengubah isi alamat memory 100AAE, akan mempengaruhi variabel lain.



Alamat memory / reference hanya digunakan secara internal oleh JavaScript dan kita tidak bisa mengaksesnya.

Efek dari references juga terjadi ke operasi perbandingan. Saat kita membandingkan object menggunakan operator `==` dan `===`, hasilnya hanya bisa `true` apabila kedua object merujuk ke referensi yang sama, seperti kode program berikut:

```
1 var mahasiswa = { nama: "Andi",
                     jurusan: "Teknik Informatika" };
2
3
4 var mahasiswaBaru = mahasiswa;
5
6 console.log(mahasiswa == mahasiswaBaru);    // true
7 console.log(mahasiswa === mahasiswaBaru);     // true
```

Jika yang dibandingkan berisi object yang sama persis tapi tidak berupa reference, hasilnya akan `false`:

```
1 var mahasiswa = { nama: "Andi",
2                     jurusan: "Teknik Informatika" };
3
4 var mahasiswaBaru = { nama: "Andi",
5                     jurusan: "Teknik Informatika" };
6
7 console.log(mahasiswa == mahasiswaBaru);    // false
8 console.log(mahasiswa === mahasiswaBaru);   // false
```

Dalam bahasa pemrograman lain, prinsip *assignment by reference* biasanya juga dipakai untuk tipe data object, sedangkan untuk tipe data yang sederhana seperti string, number, dan boolean yang berlaku adalah *assignment by value*.

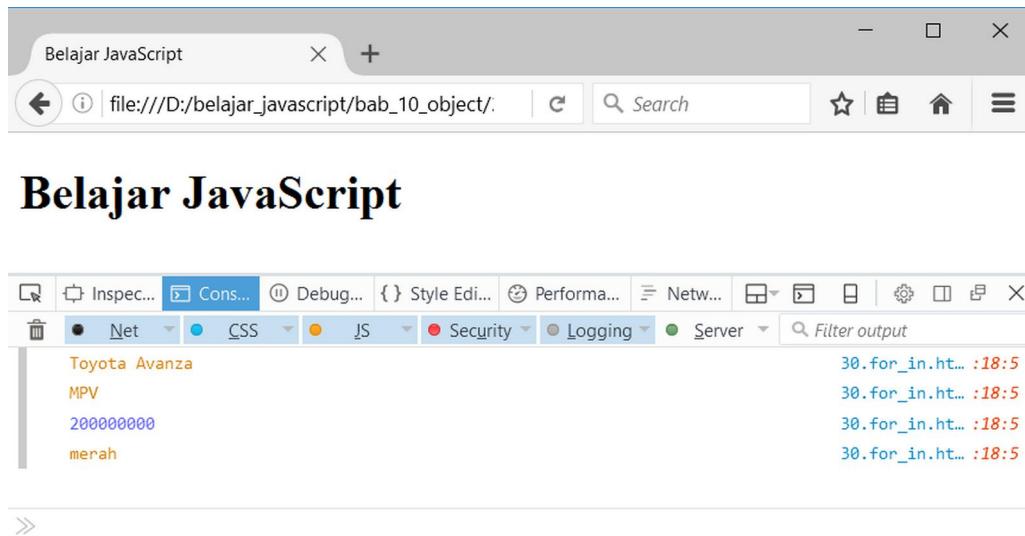
## 10.10 Perulangan FOR IN

JavaScript memiliki sebuah perulangan khusus yang bisa digunakan untuk menampilkan seluruh isi object (*property* dan *method*). Perulangan tersebut adalah **for in**.

Cara penggunaannya mirip seperti perulangan **for of**, bedanya yang menjadi variabel *counter* adalah nama dari property tersebut, bukan nilainya.

Berikut contoh penggunaan perulangan **for in** JavaScript:

```
1 var mobil = { merk: "Toyota Avanza",
2               tipe: "MPV",
3               harga: 200000000,
4               warna: "merah" };
5
6 var prop;
7 for (prop in mobil) {
8   console.log( mobil[prop] );
9 }
```



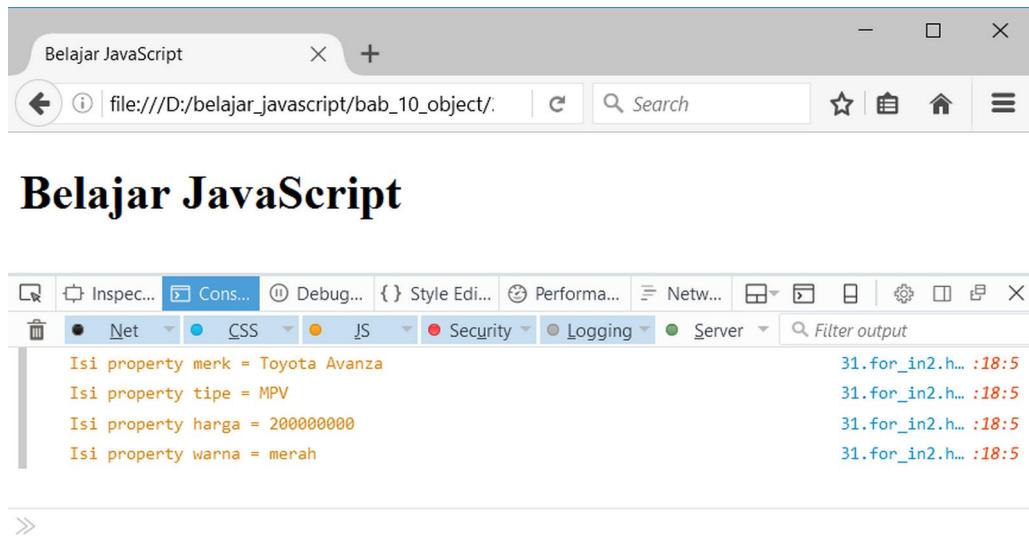
Gambar: Tampilan isi property dari object mobil

Kali ini saya memiliki object `mobil` yang diisi dengan 4 property. Setelah pendefenisian object, saya medeklarasikan variabel `prop` (singkatan dari property). Variabel ini akan digunakan di dalam perulangan `for in`.

Di dalam perulangan `for in`, variabel `prop` akan berisi nama property dari object yang sedang di proses (object `mobil`), karana itulah pemanggilannya ditulis sebagai `mobil[prop]`.

Dengan sedikit modifikasi, kita bisa melihat isi dari variabel `prop` di dalam perulangan:

```
1 var mobil = { merk: "Toyota Avanza",
2             tipe: "MPV",
3             harga: 200000000,
4             warna: "merah" };
5
6 var prop;
7 for (prop in mobil) {
8   console.log('Isi property '+ prop + ' = ' + mobil[prop]);
9 }
```

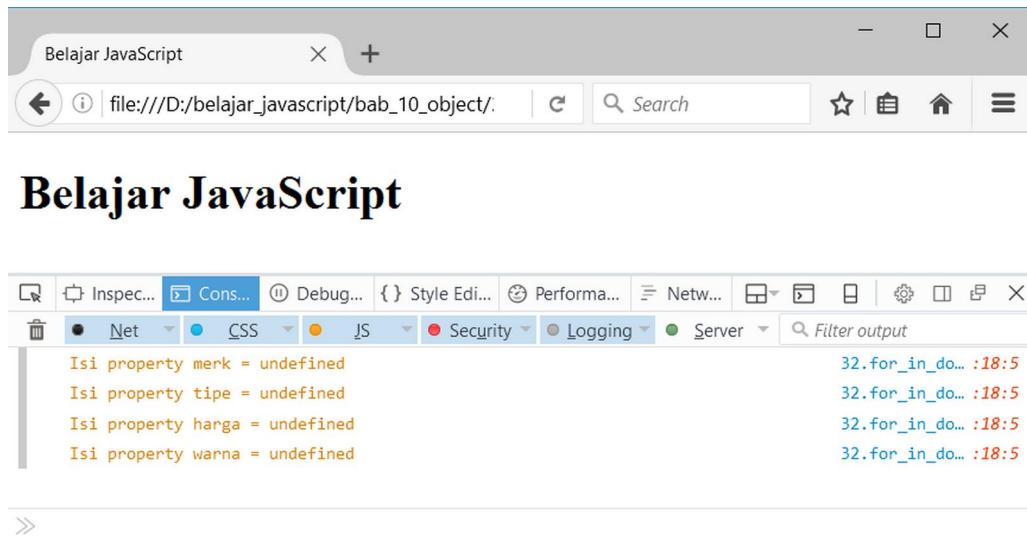


Gambar: Isi dari variabel prop dan pemanggilan mobil[prop]

Karena prop merupakan sebuah variabel yang isinya selalu berubah, kita tidak bisa menggunakan penulisan *dot notation* untuk perulangan **for in**, tapi harus menggunakan *bracket notation*.

Berikut percobaannya:

```
1 var mobil = { merk: "Toyota Avanza",
2                 tipe: "MPV",
3                 harga: 200000000,
4                 warna: "merah" };
5
6 var prop;
7 for (prop in mobil) {
8   console.log('Isi property ' + prop + ' = ' + mobil[prop]);
9 }
```

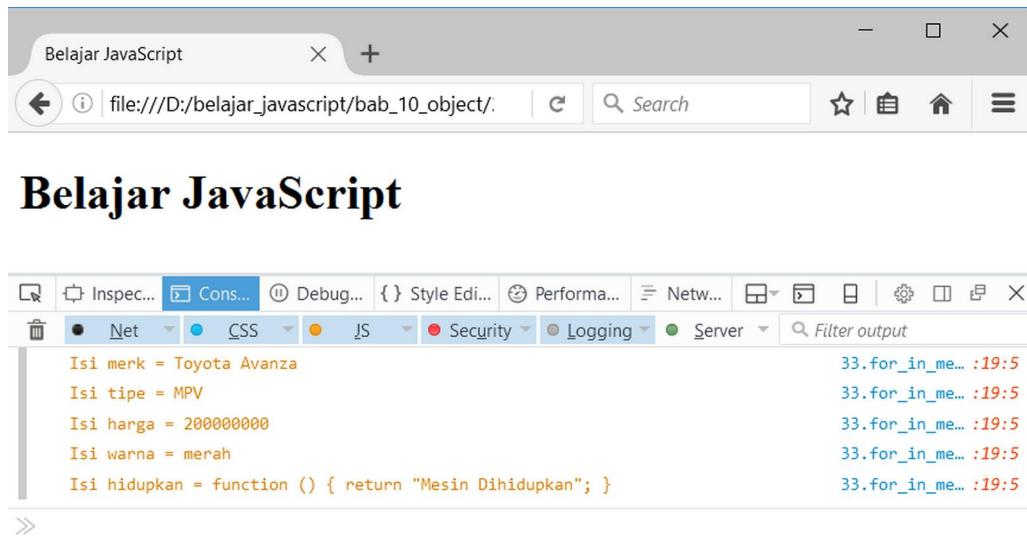


Gambar: Hasil undefined karena penggunaan dot notation dalam perulangan for in

Hasil *undefined* di dapat karena di dalam perulangan, kode kita mencoba mengakses property prop dari object mobil, bukan variabel prop yang ada di perulangan **for in**. Karena property prop ini memang tidak ada, hasilnya adalah *undefined*.

Sekarang, bagaimana jika di dalam object tersebut terdapat *method*? Mari kita coba:

```
1 var mobil = { merk: "Toyota Avanza",
2             tipe: "MPV",
3             harga: 200000000,
4             warna: "merah",
5             hidupkan: function () { return "Mesin Dihidupkan"; }
6 };
7
8 var prop;
9 for (prop in mobil) {
10   console.log('Isi ' + prop + ' = ' + mobil[prop]);
11 }
```



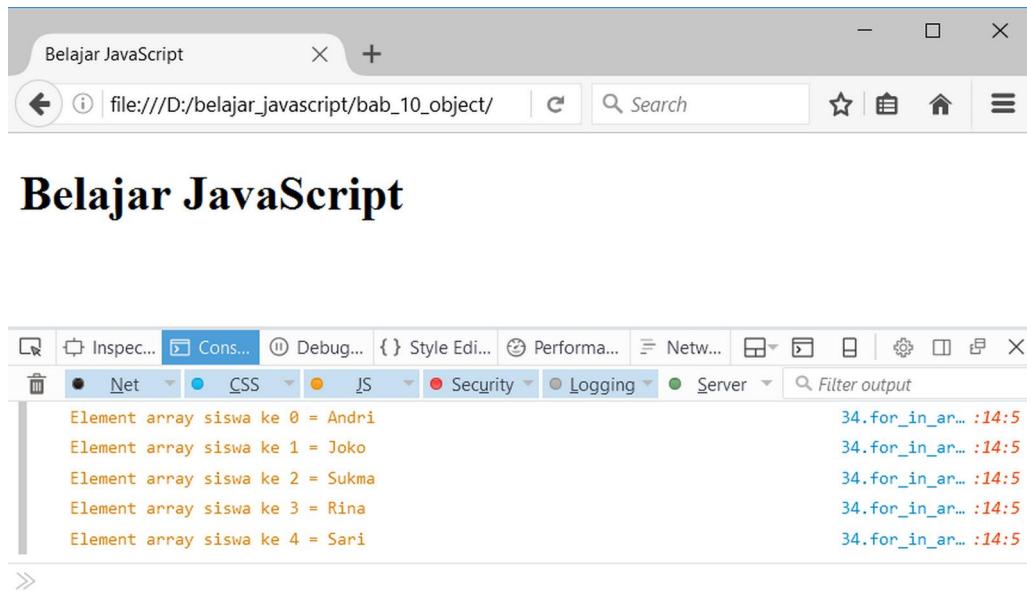
Gambar: Hasil tampilan perulangan for in untuk object yang memiliki method

Hasilnya, yang ditampilkan oleh perulangan **for in** adalah isi pendefenisian method tersebut.

## Perulangan for in Untuk Array

Jika anda masih ingat, didalam JavaScript array adalah object. Oleh karena itu, kita juga bisa menggunakan perulangan **for in** untuk menampilkan array:

```
1 var siswa = ["Andri", "Joko", "Sukma", "Rina", "Sari"];
2
3 var key;
4 for (key in siswa) {
5   console.log('Element array siswa ke ' + key + ' = ' + siswa[key]);
6 }
```



Gambar: Menampilkan isi element array menggunakan perulangan for in

Cara penggunaannya sama persis seperti sebelumnya. Khusus untuk array, di dalam perulangan **for in** yang diakses adalah **key element**, bukan **value element** seperti perulangan **for of**.

Jadi, yang mana sebaiknya di pakai? Perulangan **for in** atau **for of**?

Jika yang ingin ditampilkan adalah object, pilihan kita cuma 1: perulangan **for in**. Perulangan **for of** hanya bisa digunakan untuk array (atau yang menyerupai array).

Untuk object yang cukup kompleks, perulangan **for in** ternyata memiliki masalah, dimana ia akan menampilkan property yang berasal dari *prototype* object induknya. Kita harus menambahkan sebuah kode program lain agar **for in** tidak mendatangkan masalah.

Ini juga ‘disarankan’ oleh teks editor **Komodo Edit**. Setiap kali kita menulis perulangan **for in**, akan tampil pesan: *The body of a for in should be wrapped in an if statement to filter unwanted properties from the prototype.*

```

10 var mobil = { merk: "Toyota Avanza",
11     tipe: "MPV",
12     harga: 200000000,
13     warna: "merah",
14     hidupkan: function () { return "Mesin Dihidupkan"; }
15 };
16
17 var prop;           JavaScript: The body of a for in should be wrapped in an if statement to filter unwanted
18 for (prop in mobil) properties from the prototype.
19     console.log('Isi ' + prop + ' = ' + mobil[prop]);
20 }
21 </script>

```

Gambar: Saran dari Komodo Edit untuk perulangan for in

Penjelasan terkait alasan ini cukup kompleks. Singkatnya, untuk object yang rumit perulangan **for in** ini akan menampilkan property lain yang sebenarnya bukan kepunyaan object yang saat ini sedang kita akses.

Cara yang disarankan adalah menggunakan sebuah kondisi **if**:

```
1 for(var prop in mobil) {  
2     if(mobil.hasOwnProperty(prop)) {  
3         ...  
4     }  
5 }
```

Saya tidak akan bahas hal ini lebih lanjut karena lumayan rumit dan kurang diperlukan dalam tahap-tahap awal belajar JavaScript. Jika anda tertarik, bisa lanjut kesini: [What ‘body of a for in should be wrapped in an if statement’ mean?](#)<sup>1</sup>.

Dalam artikel tersebut juga disarankan untuk tidak menggunakan perulangan **for in** dalam menampilkan array (walaupun bisa, seperti contoh kita sebelumnya). Jika ingin menampilkan isi array, silahkan pakai perluangan **for** biasa atau **for of**.

---

Dalam bab ini kita telah membahas salah satu tipe data paling penting di dalam JavaScript, yakni Object. Tapi JavaScript masih memiliki pembahasan yang tak kalah penting, yakni tentang konsep pemrograman object, atau istilah kerennya “*Object Oriented Programming*” (OOP).

Apa itu OOP? Apa bedanya dengan object yang sudah kita pelajari disini? Inilah bahasan kita pada bab selanjutnya.

---

<sup>1</sup> [stack overflow.com/questions/1963102/what-does-the-jslint-error-body-of-a-for-in-should-be-wrapped-in-an-if-statement](https://stackoverflow.com/questions/1963102/what-does-the-jslint-error-body-of-a-for-in-should-be-wrapped-in-an-if-statement)

# 11. Object Oriented Programming JavaScript

Apa yang akan kita bahas dalam bab ini masih berhubungan dengan object, kali ini kita akan fokus ke penerapan JavaScript sebagai bahasa pemrograman berbasis object atau dalam bahasa inggris dikenal dengan istilah *Object Oriented Programming (OOP)*.

Loh, bukannya kita sudah membahas panjang lebar tentang object?

Betul, dalam bab sebelumnya kita telah mempelajari tentang object di dalam JavaScript. Tapi bukan sebagai OOP. Apa yang kita bahas baru sebatas penggunaan object sebagai **tipe data**.

OOP adalah sebuah *paradigma*, atau cara berfikir dalam membuat aplikasi. Untuk benar-benar bisa menerapkan pemrograman berbasis object ke dalam suatu bahasa pemrograman, butuh waktu yang tidak sebentar dan perlu pemahaman yang mendalam, termasuk konsep seperti *design pattern*.

Saya memutuskan untuk tidak terlalu banyak membahas materi OOP JavaScript, karena cakupannya sangat luas. Selain itu juga kurang cocok dimasukkan ke buku JavaScript level pemula. Materi OOP lebih pas dipelajari di tahap lanjutan nanti.

Disini saya akan membahas OOP JavaScript secara garis besar saja. Cukup sebagai modal bagi kita untuk masuk ke bab selanjutnya. Tapi seperti yang akan anda lihat nantinya, materi dalam bab ini bisa jadi menjadi bahasan paling rumit di dalam JavaScript.

## 11.1 Pengertian OOP

Konsep *Pemrograman Berbasis Object (PBO)*, atau dalam bahasa inggris dikenal sebagai *Object Oriented Programming (OOP)* di dasarkan pada object dalam kehidupan kita sehari-hari.

Misalkan object **gelasKaca** bisa memiliki property: `tinggiGelas`, `volumeGelas`, dan `bentukGelas`. Object **gelasKaca** juga bisa memiliki method: `diisi()`, `dibersihkan()` dan `dilemparkan()`.

Contoh lain seperti object **mobil** yang bisa memiliki property: `merk`, `warna`, dan `jenisMobil`. Dimana methodnya adalah: `dihidupkan()`, `dimatikan()`, dan `jalanKeJakarta()`.

Jika di konversi ke dalam pemrograman OOP, setiap *property* dan *method* yang diperlukan oleh suatu tipe data, akan melekat ke **object-nya**. Mari kita bahas apa maksud dari kalimat ini.

Misalkan saya memiliki function `potongString()`. Sesuai dengan namanya, fungsi ini berguna untuk memotong string. Cara penulisannya adalah sebagai berikut:

```
1 var kalimat = "Hello Indonesia";
2
3 var hasil = potongString(kalimat,6,14);
```

Variabel **hasil** akan berisi string "Indonesia", karena saya memotong string tersebut mulai dari index karakter ke-6 hingga ke-14. Ini adalah cara penulisan kode program menggunakan metode **prosedural** (berbasiskan function).

Jika menggunakan **object**, saya bisa memasukkan fungsi `potongString()` menjadi method dari object **String**, sehingga cara pemanggilannya sebagai berikut:

```
1 var kalimat = new String("Hello Indonesia");
2
3 var hasil = kalimat.potongString(6,14);
```

Untuk sementara, anda bisa abaikan keyword "**new**". Yang jelas, variabel **kalimat** akan berisi sebuah object string yang berisi "**Hello Indonesia**". Di dalam object string ini terdapat method `potongString()`, yang dipanggil menggunakan *dot notation* sebagaimana kita memanggil method biasa.

Inilah contoh *peng-objek-an* sebuah kode program, dimana sebelumnya saya menggunakan *function*, diubah ke dalam bentuk *object*.

Dari penjelasan diatas, terlihat bahwa "lawan" dari OOP adalah pemrograman prosedural yang berbasiskan function. Tapi bukan berarti bahwa pemrograman prosedural itu tidak bagus.

Untuk membuat program sederhana dalam waktu cepat, *pemrograman prosedural* merupakan pilihan yang paling pas. *Pemrograman berbasis object* baru terasa manfaatnya ketika kita membuat aplikasi yang besar, terdiri dari berbagai komponen, dan dikerjakan oleh banyak orang.

JavaScript memang sebuah bahasa pemrograman berbasis object, tapi tidak benar-benar menyerapkan konsep OOP secara penuh. Sebagai contoh, di dalam bahasa pemrograman full OOP seperti JAVA, untuk menjalankan kode program yang paling sederhana sekali pun, kita harus merancang class dan membuat object.

## 11.2 Pengertian Class dan Object

Dalam teori OOP, **Class** adalah cetak biru dari sebuah **object**, atau dalam bahasa sederhananya, **class** merupakan kelompok umum dari sesuatu. **Object** adalah "sesuatu" tersebut. **Class** dan **object** merupakan materi paling dasar dari pemrograman berbasis object.

Jika **mobil** adalah *class*, maka **mobilAndi** merupakan *object* dari *class* mobil. Jika **binatang** adalah *class*, maka **sapi** merupakan *object* dari *class* binatang. Jika **Gelas** adalah *class*, maka **gelasKaca** adalah *object* dari *class* gelas.

Terlihat disini bahwa **mobilAndi** menjadi bagian dari kelompok **mobil**. Dan bisa jadi nanti akan ada **mobilJoko** dan **mobilAnton** yang semuanya tergabung ke dalam kelompok **mobil** (yang kita sebut sebagai *class* **mobil**).

Uniknya, sebelum **ECMAScript 6**, di dalam JavaScript tidak ada yang namanya **class**. Karena itulah JavaScript disebut juga sebagai *classless programing language* (bahasa pemrograman tanpa class), padahal JavaScript sendiri merupakan sebuah bahasa pemrograman berbasis object.

Ini dimungkinkan karena Object di JavaScript dibuat menggunakan mekanisme yang dinamakan **prototype** (akan kita bahas nantinya). Di **ECMAScript 6**, fitur **class** sudah ditambahkan, sehingga kita bisa menggunakannya untuk membuat object.

## 11.3 Membuat Class dan Object

Untuk membuat class di dalam JavaScript, dalam **ECMAScript 6** bisa menggunakan keyword **class**:

```
1 class Mobil {  
2   // isi class disini  
3 }
```

Disini saya membuat class **Mobil** yang belum berisi apa-apa. Salah satu kebiasaan programmer JavaScript adalah, nama **class** ditulis dengan awalan huruf besar, bukan dengan huruf kecil sebagaimana layaknya variabel. Walaupun tentu saja kita juga bisa menggunakan awalan huruf kecil, seperti:

```
1 class mobil {  
2   // isi class disini  
3 }
```

Jika class ini terdiri dari 2 kosa kata atau lebih, tetap menggunakan gaya penulisan camel case:

```
1 class MobilBaruBeli {  
2   // isi class disini  
3 }
```

Dengan penulisan seperti ini, kita bisa membedakan apakah sebuah *identifier* itu berisi **class** (diawali huruf besar), **variabel** (diawali huruf kecil), atau **konstanta** (huruf besar semua).

Dalam konsep OOP, **class** hanya berperan sebagai cetak biru (*blue print*) dari **object**. Di dalam object lah pemrosesan dilakukan. Untuk membuat object dari suatu class, digunakan keyword **new**, kemudian diikuti dengan nama class tersebut.

Berikut contohnya:

```
1 class Mobil {  
2 }  
3  
4 var mobilAndi = new Mobil();  
5 var mobilJoko = new Mobil();
```

Kode program diatas artinya saya membuat 2 buah *object* dari **class Mobil**. Object pertama disimpan kedalam variabel `mobilAndi`, dan object kedua ke dalam variabel `mobilJoko`.

Proses pembuatan object dari class ini dikenal dengan sebutan **instantiation**. Bahasa Inggris *instantiation* sendiri berarti: *membuat sesuatu yang berwujud dari yang abstrak*. Dalam contoh ini, class **Mobil** merupakan suatu hal yang abstract, wujudnya ada di object `mobilAndi` dan `mobilJoko`.

JavaScript juga memiliki operator **instanceof** yang bisa digunakan untuk mengecek apakah suatu object merupakan *instance* dari sebuah class.

```
1 class Mobil {  
2 }  
3  
4 var mobilAndi = new Mobil();  
5 var mobilJoko = new Mobil();  
6  
7 console.log( mobilAndi instanceof Mobil ); // true  
8 console.log( mobilJoko instanceof Mobil ); // true  
9 console.log( mobilJoko instanceof String ); // false
```

Disini `mobilAndi instanceof Mobil` dan `mobilJoko instanceof Mobil` akan menghasilkan `true`, karena object `mobilAndi` memang berasal dari class **Mobil**.

### Contoh Class Yang “Aneh”

Saat pertama kali belajar pemrograman berbasis object, saya selalu diajarkan class yang “aneh” dan kurang jelas fungsinya seperti class *mobil*, atau class *binatang*. Berbagai buku pun menggunakan contoh seperti ini.

Salah satu alasannya adalah, konsep OOP cukup kompleks jika langsung diajarkan menggunakan object yang sebenarnya.

Untuk sementara waktu anda sebaiknya “pasrah” menerima contoh object seperti ini, dimana saya akan menggunakan contoh class **Mobil**. Setelah paham gambaran dasar dari OOP, barulah kita akan masuk ke pembuatan class yang sebenarnya.

## 11.4 Class Sebagai “Object Induk”

Pertanyaan yang cukup mendasar ketika belajar OOP adalah, untuk apa kita repot-repot membuat *class*? Kenapa tidak langsung membuat *object* saja?

*Class* berperan sebagai wadah yang menyediakan semua hal yang dibutuhkan oleh object. Sebagai contoh, perhatikan kode program berikut:

```

1 var mobilAndi = { merk: "Daihatsu Xenia",
2                     tipe: "MPV",
3                     harga: 150000000};
4
5 var mobilJoko = { merk: "Toyota Camry",
6                     tipe: "Sedan",
7                     harga: 350000000};

```

Disini saya membuat 2 buah object: `mobilAndi` dan `mobilJoko`. Keduanya memiliki 3 property yang sama: `merk`, `tipe`, dan `harga`, dengan isi data yang berlainan.

Sekarang, mari kita tambahkan method:

```

1 var mobilAndi = { merk: "Daihatsu Xenia",
2                     tipe: "MPV",
3                     harga: 150000000,
4                     hidupkan: function () { return "Mesin Dihidupkan"; },
5                     pergi: function (tempat) { return "Pergi ke "+ tempat; }
6                 };
7
8 var mobilJoko = { merk: "Toyota Camry",
9                     tipe: "Sedan",
10                    harga: 350000000,
11                    hidupkan: function () { return "Mesin Dihidupkan"; },
12                    pergi: function (tempat) { return "Pergi ke "+ tempat; }
13                };

```

Kali ini setiap object memiliki 2 method tambahan: `hidupkan()` dan `pergi()`.

Dapat dilihat bahwa kedua object ini sebenarnya masuk ke golongan “mobil”. Jika saya membuat object baru dengan nama `mobilAlex`, tentunya anda bisa menebak bahwa object `mobilAlex` ini seharusnya juga memiliki property dan method yang sama.

Bagaimana jika saya butuh membuat 100 object mobil? Akan lebih efisien jika terdapat sebuah “object induk” bernama `Mobil`.

Ketika sebuah object dinyatakan sebagai bagian dari “Mobil”, otomatis ia sudah memiliki 3 property: `merk`, `tipe`, dan `harga`. Serta memiliki 2 buah method: `hidupkan()` dan `pergi()`.

Untuk membuat object `mobilTono`, `mobilRudi`, atau `mobilAnton`, saya cukup memasukkannya sebagai anggota dari “object induk” `Mobil`. Dengan demikian saya bisa langsung mengakses `mobilRudi.merk`, atau `mobilAnton.hidupkan()`.

“Object induk” inilah yang disebut sebagai **class**. Ketika object dibuat dari suatu class, object tersebut otomatis berisi seluruh property dan method yang melekat kepada asal class-nya, yakni `Mobil`. Selain itu, saya juga bisa membuat kode program yang nantinya hanya bisa menerima input berupa object dari class `Mobil`.

Mari kita lihat kembali cara pembuatan class `Mobil` yang sudah dibahas sebelumnya:

```

1 class Mobil {
2 }
3
4 var mobilAndi = new Mobil();
5 var mobilJoko = new Mobil();

```

Saya bisa menambahkan “sesuatu” di dalam class **Mobil** ini, entah itu berupa property atau method. Dengan demikian, object `mobilAndi` dan `mobilJoko` otomatis juga memiliki property dan method yang sama. Termasuk seluruh object lain yang nantinya dibuat menggunakan perintah `new Mobil()`.

Mari kita pelajari cara menambahkan property dan method ke dalam class ini.

## 11.5 Class Property

Untuk bisa menambahkan property ke dalam **Class**, kita harus menggunakan method khusus bernama **constructor()**. Method *constructor* ini otomatis akan berjalan saat proses pembuatan object (saat proses *instantiation*).

Penulisannya sebagai berikut:

```

1 class Mobil {
2   constructor(){
3     // isi constructor disini
4   }
5 }

```

Di dalam JavaScript, semua property untuk **Class** harus ditulis di dalam method `constructor()`. Bagaimana cara menulis property untuk class ini? Kita tidak bisa langsung membuat property seperti biasa, tapi harus menggunakan object khusus bernama **this**:

```

1 class Mobil {
2   constructor() {
3     this.merk = "Daihatsu Xenia";
4     this.tipe = "MPV";
5     this.harga = "150000000";
6   }
7 }

```

**This** adalah object khusus sebagai pengganti object yang nantinya di buat dari class **Mobil**. Penjelasan tentang **this** memang sedikit rumit, untuk sementara mari kita anggap bahwa dalam membuat property class, harus menggunakan perintah `this.namaProperty`.

Setelah ditulis seperti ini, setiap object yang dibuat dari class **Mobil** secara otomatis punya 3 property tersebut. Mari kita coba:

```

1 class Mobil {
2   constructor() {
3     this.merk = "Daihatsu Xenia";
4     this.tipe = "MPV";
5     this.harga = "150000000";
6   }
7 }
8
9 var mobilAndi = new Mobil();
10 console.log ( mobilAndi.merk );    // Daihatsu Xenia
11
12 var mobilJoko = new Mobil();
13 console.log ( mobilJoko.tipe );    // MPV
14
15 var mobilAlex = new Mobil();
16 console.log ( mobilAlex.merk );    // Daihatsu Xenia

```

Di dalam class Mobil saya membuat 3 property: `this.merk`, `this.tipe`, dan `this.harga`. Ketika class Mobil dijadikan object, property ini sudah langsung ‘melekat’ ke setiap object.

Disini anda bisa melihat bahwa class Mobil berperan sebagai “cetakan” untuk object `mobilAndi`, `mobilJoko`, dan `mobilAlex`. Setiap object ini memiliki property yang sama, yang berasal dari class Mobil.

## 11.6 Pengertian this

Mari kita bahas sedikit tentang fungsi dari keyword `this` yang digunakan saat pembuatan *class property*.

Ketika object `mobilAndi` dibuat dengan perintah:

```
var mobilAndi = new Mobil()
```

JavaScript akan menjalankan method `constructor()` dari class `Mobil`. Jika di dalamnya terdapat keyword `this`, itu akan diganti menjadi `mobilAndi`. Artinya yang dijalankan secara internal oleh JavaScript adalah sebagai berikut:

```

1 class Mobil {
2   constructor() {
3     mobilAndi.merk = "Daihatsu Xenia";
4     mobilAndi.tipe = "MPV";
5     mobilAndi.harga = "150000000";
6   }
7 }

```

Karena itulah kita bisa mengakses `mobilAndi.merk`, `mobilAndi.tipe`, dan `mobilAndi.harga`.

Bagaimana dengan object lain? misalnya:

```
var mobilJoko = new Mobil();
```

Yang terjadi juga sama. JavaScript akan mengganti `this` dengan `mobilJoko`:

```
1 class Mobil {  
2     constructor() {  
3         mobilJoko.merk = "Daihatsu Xenia";  
4         mobilJoko.tipe = "MPV";  
5         mobilJoko.harga = "150000000";  
6     }  
7 }
```

Terlihat, inilah fungsi dari penulisan `this` di dalam pembuatan class. `This` berfungsi sebagai pengganti object yang nantinya dibuat dari class `Mobil`.

Peran `this` tidak terbatas di dalam `constructor` saja, tapi juga bisa digunakan di luar `constructor`, selama masih di dalam class.

## 11.7 Argument Constructor

Dalam penjelasan sebelum ini, kita sudah membahas proses pembuatan object dari class `Mobil`. Akan tetapi, objek `Mobil` yang dihasilkan identik satu sama lain. Baik `mobilJoko.merk` dan `mobilAlex.merk` sama-sama berisi "Daihatsu Xenia".

Sekarang, bagaimana caranya agar setiap object punya isi property yang berbeda? Solusinya, kita bisa menambahkan argument ke dalam `constructor`. Argumen ini diinput pada saat proses pembuatan object.

Sebagai contoh, untuk `mobilAndi`, saya bisa menulis sebagai berikut:

```
var mobilAndi = new Mobil("Daihatsu Xenia", "MPV", 150000000);
```

Sedangkan untuk `mobilJoko` saya bisa menulis seperti ini:

```
var mobilJoko = new Mobil("Toyota Camry", "Sedan", 350000000);
```

Dengan demikian, ketika `mobilJoko.merk` diakses, hasilnya adalah "Toyota Camry", sedangkan ketika `mobilAndi.merk` diakses, hasilnya adalah "Daihatsu Xenia". Setiap object dari class `Mobil` akan punya property yang sama, tapi dengan nilai yang berbeda.

Karena kita mengirim "sesuatu" saat class `Mobil` dipanggil, maka kita butuh "menangkapnya" di class `constructor()`. Berikut kode program penambahan argument `constructor` untuk class `Mobil`:

```

1 class Mobil {
2     constructor(merkArg, tipeArg, hargaArg) {
3         this.merk = merkArg;
4         this.tipe = tipeArg;
5         this.harga = hargaArg;
6     }
7 }
8
9 var mobilAndi = new Mobil("Daihatsu Xenia", "MPV", 150000000);
10 console.log ( mobilAndi.merk );      // Daihatsu Xenia
11 console.log ( mobilAndi.tipe );      // MPV
12
13 var mobilJoko = new Mobil("Toyota Camry", "Sedan", 350000000);
14 console.log ( mobilJoko.merk );      // Toyota Camry
15 console.log ( mobilJoko.tipe );      // Sedan

```

Saya menambahkan 3 argument ke dalam method `constructor()`, dimana masing-masingnya digunakan untuk menampung inputan saat pembuatan class **Mobil**.

Ketika object `mobilAndi` dibuat dengan perintah:

```
var mobilAndi = new Mobil("Daihatsu Xenia", "MPV", 150000000);
```

String "Daihatsu Xenia" akan dikirim ke variabel `merkArg`, string "MPV" akan dikirim ke variabel `tipeArg`, dan number 150000000 akan dikirim ke variabel `hargaArg`.

Selanjutnya, argumen `merkArg`, `tipeArg` dan `hargaArg` diinput ke dalam property `this.merk`, `this.tipe`, dan `this.harga`. Hasilnya, ketika kita mengakses `mobilAndi.merk`, yang tampil adalah "Daihatsu Xenia", sesuai nilai yang diinput pada saat pembuatan object `mobilAndi`.

Begitu juga halnya dengan object `mobilJoko`:

```
var mobilJoko = new Mobil("Toyota Camry", "Sedan", 350000000);
```

Ketiga argumen akan dipindahkan ke property `this.merk`, `this.tipe` dan `this.harga`. Sehingga bisa diakses dari `mobilJoko.merk`, `mobilJoko.tipe` dan `mobilJoko.harga`.

Saya sengaja membuat argument dengan nama `merkArg`, `tipeArg` dan `hargaArg`, agar mudah dibedakan. Biasanya argumen seperti ini cukup dibuat sebagai berikut:

```

1 class Mobil {
2     constructor(merk, tipe, harga) {
3         this.merk = merk;
4         this.tipe = tipe;
5         this.harga = harga;
6     }
7 }
```

Variabel `merk`, `tipe` dan `harga` yang terdapat di argument, digunakan untuk menampung nilai inputan saat proses pembuatan object. Sedangkan `this.merk` merupakan **Class** property yang nilainya berasal dari argument.

## 11.8 Class Method

Kita telah berhasil membuat property di dalam class. Selanjutnya adalah membuat method agar bisa mengakses `mobilAndi.hidupkan()` dan `mobilAndi.pergi("Jakarta")`.

Untuk menambahkan method ke dalam class, kita tidak perlu menulisnya di dalam `constructor()`, tapi cukup di dalam class:

```

1 class Mobil {
2     constructor(merk, tipe, harga) {
3         this.merk = merk;
4         this.tipe = tipe;
5         this.harga = harga;
6     }
7
8     hidupkan(){
9         return "Mesin Dihidupkan";
10    }
11 }
12
13 var mobilAndi = new Mobil("Daihatsu Xenia","MPV",150000000);
14 console.log ( mobilAndi.merk );           // Daihatsu Xenia
15 console.log ( mobilAndi.hidupkan() );    // Mesin Dihidupkan
16
17 var mobilJoko = new Mobil("Toyota Camry","Sedan",350000000);
18 console.log ( mobilJoko.merk );          // Toyota Camry
19 console.log ( mobilJoko.hidupkan() );    // Mesin Dihidupkan
```

Setelah pembuatan `constructor`, saya menulis sebuah method `hidupkan()`. Isinya hanya mengembalikan string "Mesin Dihidupkan".

Sama seperti class property, setiap object yang diturunkan dari class **Mobil**, otomatis punya method `hidupkan()`. Perhatikan juga bahwa pembuatan method di dalam class tidak menggunakan keyword *function* maupun *function expression*, tapi langsung ditulis nama functionnya.

Baik, mari kita buat lebih menarik. Di dalam method `hidupkan()`, saya ingin mengakses sebuah class property:

```

1 class Mobil {
2   constructor(merk, tipe, harga) {
3     this.merk = merk;
4     this.tipe = tipe;
5     this.harga = harga;
6   }
7
8   hidupkan(){
9     return "Mesin "+this.merk+" Dihidupkan";
10  }
11 }
12
13 var mobilAndi = new Mobil("Daihatsu Xenia","MPV",150000000);
14 console.log ( mobilAndi.merk );           // Daihatsu Xenia
15 console.log ( mobilAndi.hidupkan() );    // Mesin Daihatsu Xenia Dihidupkan
16
17 var mobilJoko = new Mobil("Toyota Camry","Sedan",350000000);
18 console.log ( mobilJoko.merk );          // Toyota Camry
19 console.log ( mobilJoko.hidupkan() );    // Mesin Toyota Camry Dihidupkan

```

Hasil dari pemanggilan method `hidupkan()` adalah string: `Mesin "Mesin "+this.merk+" Dihidupkan"`. Artinya, ketika method `hidupkan()` dipanggil, hasilnya akan berbeda pada setiap object. Isi dari variabel `this.merk` diambil dari property object.

Di dalam object `mobilAndi`, variabel `this.merk` akan menjadi `"Daihatsu Xenia"`, oleh karena itu hasil pemanggilan method `mobilAndi.hidupkan()` adalah `Mesin Daihatsu Xenia Dihidupkan`. Bagaimana dengan method yang memiliki argumen? Caranya sama seperti pembuatan method biasa:

```

1 class Mobil {
2   constructor(merk, tipe, harga) {
3     this.merk = merk;
4     this.tipe = tipe;
5     this.harga = harga;
6   }
7
8   hidupkan(){
9     return "Mesin "+ this.merk +" Dihidupkan";
10  }
11
12  pergi(tempat) {
13    return "Pergi ke "+ tempat +" dengan "+ this.merk;
14  }

```

```

15 }
16
17 var mobilAndi = new Mobil("Daihatsu Xenia", "MPV", 150000000);
18 console.log ( mobilAndi.merk ); // Daihatsu Xenia
19 console.log ( mobilAndi.pergi("Jakarta") );
20 // Pergi ke Jakarta dengan Daihatsu Xenia
21
22 var mobilJoko = new Mobil("Toyota Camry", "Sedan", 350000000);
23 console.log ( mobilJoko.merk ); // Toyota Camry
24 console.log ( mobilJoko.pergi("Bali") );
25 // Pergi ke Bali dengan Toyota Camry

```

Disini saya menambahkan method `pergi()` ke dalam class **Mobil**, dimana method ini membutuhkan 1 argumen, yakni tempat. Method `pergi()` mengembalikan nilai "Pergi ke "+ tempat +" dengan "+ `this.merk`.

Artinya, ketika saya menjalankan `mobilAndi.pergi("Jakarta")`, variabel tempat akan diganti menjadi string "Jakarta", sedangkan variabel `this.merk` akan diambil dari constructor(), dimana dalam contoh ini berisi string `Daihatsu Xenia`. Hasilnya adalah text "Pergi ke Jakarta dengan Daihatsu Xenia".

Ketika method `pergi()` dipanggil dari object `mobilJoko`, hasilnya juga akan berbeda.

Satu hal yang pasti, setiap object yang dibuat dari class **Mobil**, akan memiliki 3 property: `merk`, `tipe`, harga serta 2 method: `hidupkan()`, dan `pergi()`.

## 11.9 Javascript: Prototype Based Language

Sebelum kehadiran fitur class di ECMAScript 6, di dalam JavaScript tidak dikenal yang namanya **class**. Tapi JavaScript tetap menerapkan konsep OOP.

OOP di dalam JavaScript dibuat melalui mekanisme yang dinamakan **prototype**. Karena itulah JavaScript disebut sebagai **Prototype Based Language**.

**Prototype based language** berbeda dengan konsep OOP di bahasa pemrograman lain seperti JAVA, C++ maupun PHP. JavaScript bisa dibilang sebagai satu-satunya bahasa pemrograman populer yang menggunakan konsep ini. Bahasa pemrograman lain menggunakan konsep "OOP classic" yang menggunakan **class** (mirip seperti materi yang kita pelajari sebelumnya).

Ini pula yang membuat programmer pemula susah untuk memahami OOP di JavaScript, termasuk saya (karena sudah terlanjur klop dengan "OOP classic"). Ibarat terbiasa menyentir mobil di sebelah kanan, sekarang harus belajar cara nyetir di sebelah kiri, dan menjalankan pedal gas dengan tangan (karena aturannya harus seperti itu).

Diskusi tentang kenapa JavaScript menggunakan *prototype based language*, dan bukannya "OOP yang biasa", cukup sering terjadi di forum-forum. Jika tertarik, anda bisa baca-baca kesini: [What does it mean that Javascript is a prototype based language?](#)<sup>1</sup> atau [prototype based vs. class based inheritance](#)<sup>2</sup>.

<sup>1</sup><http://stackoverflow.com/questions/186244/what-does-it-mean-that-javascript-is-a-prototype-based-language>

<sup>2</sup><http://stackoverflow.com/questions/816071/prototype-based-vs-class-based-inheritance>

Mayoritas “protes” karena susahnya dalam memahami OOP di JavaScript. Fitur standar OOP seperti konsep property yang bisa di set sebagai *private*, *protected* dan *public* juga tidak ada di JavaScript.

Mungkin karena hal inilah akhirnya *Technical Committee 39 (TC39)* dari ECMA, yakni komite yang membuat standar ECMAScript, menambahkan fitur **class** di ECMAScript 6, sebagaimana yang sudah sudah kita pelajari sebelum ini.

Namun karena ECMAScript 6 sendiri terbilang cukup baru, masih banyak tutorial dan library yang menggunakan **prototype** dalam pembuatan object JavaScript. Karena itulah kita juga akan mempelajarinya.

## 11.10 Membuat Object dengan Constructor Functions

Untuk membuat “class” tanpa keyword **class** (semoga anda paham apa yang saya maksud), JavaScript menyediakan **constructor functions**. **Constructor functions** adalah function khusus yang digunakan untuk membuat “kelompok object”, yang tidak lain adalah sebutan untuk **class** di dalam OOP biasa.

**Constructor functions** sebenarnya tidak berbeda dengan function biasa. Sebagai contoh, untuk membuat class **Mobil**, penulisannya adalah sebagai berikut:

```
1 function Mobil (merk, tipe, harga) {  
2     this.merk = merk;  
3     this.tipe = tipe;  
4     this.harga = harga;  
5 }  
6  
7 var mobilAndi = new Mobil("Daihatsu Xenia", "MPV", 150000000);  
8 var mobilJoko = new Mobil("Toyota Camry", "Sedan", 350000000);  
9  
10 console.log( mobilAndi.merk ); // Daihatsu Xenia  
11 console.log( mobilJoko.merk ); // Toyota Camry  
12  
13 console.log( mobilAndi instanceof Mobil); // true  
14 console.log( mobilJoko instanceof Mobil); // true
```

Saya membuat function **Mobil** dengan 3 argument: `merk`, `tipe`, dan `harga`. Di dalam function **Mobil**, ketiga argumen diinput ke 3 property: `this.merk`, `this.tipe`, dan `this.harga`. Ketiganya membentuk sebuah “kelompok object” yang bernama **Mobil** (ingat, kita tidak menggunakan istilah “class”).

Keyword `this`-lah yang membedakan sebuah function “biasa” dengan **constructor functions**. Cara penulisannya sama persis seperti *method constructor* yang kita gunakan ketika membuat class **Mobil** di ECMAScript 6.

Untuk membuat object `mobilAndi` dan `mobilJoko`, caranya juga sama, yakni menggunakan keyword `new`. Perhatikan bahwa operator `instanceof` akan menghasilkan nilai `true`. Artinya, object `mobilAndi` dan `mobilJoko` adalah anggota (*instance of*) dari “kelompok object” **Mobil**.

Sekarang, bagaimana cara membuat *method*? Kita tinggal menambahkan *function expression* ke dalam *constructor functions*, seperti contoh berikut:

```

1 function Mobil (merk, tipe, harga) {
2     this.merk = merk;
3     this.tipe = tipe;
4     this.harga = harga;
5     this.hidupkan = function(){
6         return "Mesin Dihidupkan";
7     };
8 }
9
10 var mobilAndi = new Mobil("Daihatsu Xenia","MPV",150000000);
11 var mobilJoko = new Mobil("Toyota Camry", "Sedan",350000000);
12
13 console.log( mobilAndi.hidupkan() ); // Mesin Dihidupkan
14 console.log( mobilJoko.hidupkan() ); // Mesin Dihidupkan

```

Sangat mirip seperti cara pembuatan method biasa ke dalam object. Bedanya, kita menggunakan `this.hidupkan`.

Bagaimana jika method tersebut menggunakan argument? Berikut contoh penulisannya:

```

1 function Mobil (merk, tipe, harga) {
2     this.merk = merk;
3     this.tipe = tipe;
4     this.harga = harga;
5
6     this.hidupkan = function(){
7         return "Mesin Dihidupkan";
8     };
9
10    this.pergi = function(tempat) {
11        return "Pergi ke "+ tempat +" dengan "+ this.merk;
12    };
13 }
14
15 var mobilAndi = new Mobil("Daihatsu Xenia","MPV",150000000);
16 var mobilJoko = new Mobil("Toyota Camry", "Sedan",350000000);
17
18 console.log( mobilAndi.hidupkan() ); // Mesin Dihidupkan
19 console.log( mobilJoko.hidupkan() ); // Mesin Dihidupkan
20

```

```
21 console.log( mobilAndi.pergi("Padang") );
22 // Pergi ke Padang dengan Daihatsu Xenia
23
24 console.log( mobilJoko.pergi("Makassar") );
25 // Pergi ke Makassar dengan Toyota Camry
```

Saya menambahkan method `pergi()` dengan 1 argument: `tempat`. Dapat terlihat bahwa kita juga bisa menggunakan variabel `this` di dalam method.

## 11.11 Membuat Class Method Dengan Object Prototype

Selain membuat “kelompok object” menggunakan *constructor functions*, kita juga bisa membuatnya menggunakan object khusus yang bernama **prototype**. Dengan menggunakan **prototype**, property dan method bisa ditambahkan kapan saja.

Berikut contoh penggunaannya:

```
1 function Mobil (merk, tipe, harga) {
2     this.merk = merk;
3     this.tipe = tipe;
4     this.harga = harga;
5 }
6
7 Mobil.prototype.jumlahRoda = 4;
8
9 Mobil.prototype.hidupkan = function(){
10         return "Mesin Dihidupkan";
11     };
12
13 Mobil.prototype.pergi = function(tempat) {
14         return "Pergi ke "+ tempat +" dengan "+ this.merk;
15     };
16
17 var mobilAndi = new Mobil("Daihatsu Xenia","MPV",150000000);
18 var mobilJoko = new Mobil("Toyota Camry","Sedan",350000000);
19
20 console.log( mobilAndi.jumlahRoda ); // 4
21 console.log( mobilJoko.jumlahRoda ); // 4
22
23 console.log( mobilAndi.pergi("Padang") );
24 // Pergi ke Padang dengan Daihatsu Xenia
25
26 console.log( mobilJoko.pergi("Makassar") );
27 // Pergi ke Makassar dengan Toyota Camry
```

Karena kelompok object **Mobil** memiliki *argument*, maka saya harus membuatnya menggunakan *constructor functions*. Selanjutnya, saya menambahkan 1 property **jumlahRoda**. Perhatikan cara penulisannya:

```
Mobil.prototype.jumlahRoda= 4;
```

Inilah cara menambahkan property menggunakan **prototype object**. Karena perintah ini, seluruh object **Mobil** akan memiliki property **jumlahRoda**.

Bagaimana dengan method? Caranya juga sama:

```
1 Mobil.prototype.hidupkan = function(){
2     return "Mesin Dihidupkan";
3 };
4
5 Mobil.prototype.pergi = function(tempat) {
6     return "Pergi ke "+ tempat +" dengan "+ this.merk;
7 };
```

Hasilnya, seluruh object **Mobil** akan memiliki method **hidupkan()** dan **pergi()**.

Jika anda sudah paham cara pembuatan object menggunakan keyword **class**, tidak akan merasa kesulitan beralih menggunakan **constructor functions** maupun **object prototype**. Perbedaan-nya mirip seperti **function declaration** dengan **function expression** (mudah-mudahan anda masih ingat pengertian kedua istilah ini)

Bahasan tentang OOP JavaScript sebenarnya belum selesai sampai disini, misalnya nanti ada lagi istilah **closure**, cara membuat **inheritance**, **encapsulation**, dll. Penjelasan terkait materi ini lebih pas sebagai bahan buku JavaScript lanjutan. Juga agar anda tidak makin pusing dengan pemrograman berbasis object di JavaScript.

## 11.12 Object Bawaan JavaScript

Dalam bab ini dan bab sebelumnya, kita telah membahas cara pembuatan object sebagai tipe data dan object sebagai bagian dari class. Keduanya merupakan object yang kita buat sendiri.

Selain itu, juga terdapat object bawaan JavaScript (*JavaScript Native Object*), yang bisa kita gunakan dalam merancang kode program. Object bawaan ini memiliki banyak method dan property, yang akan saya bahas secara bertahap mulai bab berikutnya.



Jika di dalam bahasa pemrograman PHP terdapat *function* bawaan seperti `var_dump()`, `substr()`, dan `pow()`. Di dalam JavaScript mayoritas function ini berbentuk *method*, dan harus diakses menggunakan *Object*.

Berikut Object bawaan JavaScript yang akan kita bahas nantinya:

- Number
- Math
- String
- Array
- Boolean
- RegExp
- Function
- Date

Selain daftar diatas, masih terdapat object bawaan JavaScript lain. Sengaja tidak saya tulis karena tidak akan kita bahas dalam buku ini. Jika tertarik, anda bisa melihat daftar lengkapnya di sini: [Standard built-in objects JavaScript<sup>3</sup>](#).

## Object atau Tipe Data Primitif?

Tunggu dulu, dari daftar diatas, terdapat object **Number**, **String**, dan **Boolean**. Bukankah ini tipe data primitif? Lalu ada juga **Function**, kenapa ini semua termasuk disebut **object**? Inilah satu lagi fitur unik di dalam JavaScript.

Jika saya meminta anda untuk membuat angka 52, lalu angka ini disimpan ke dalam variabel foo, tentu kode programnya adalah sebagai berikut:

```
var foo = 52;
```

Ini adalah cara membuat tipe data primitif number, menggunakan **number literal**. Tapi terdapat cara lain:

```
var foo = new Number(52);
```

Cara kedua ini menggunakan apa yang disebut sebagai **Object Constructor**. Perintah `new Number()` digunakan untuk membuat object **Number**.

Ini juga berlaku untuk tipe data string. Jika menggunakan tipe data primitif, saya bisa menulis sebagai berikut:

```
var foo = "Belajar JavaScript";
```

Dengan **Object Constructor**, penulisannya seperti ini:

```
var foo = new String("Belajar JavaScript");
```

Variabel foo pertama berisi string "Belajar JavaScript" sebagai **tipe data primitif** (string literal), sedangkan variabel foo kedua berisi **Object string** "Belajar JavaScript".

Tipe data yang dibuat dari *object constructor* berprilaku sama seperti data biasa. Kita bisa menggunakan seluruh operator layaknya tipe data primitif:

---

<sup>3</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects)

```

1 var foo = new Number("52");
2
3 var hasil = foo + 25;
4 console.log( hasil );      // 77
5
6 var bar = new String("Belajar JavaScript");
7
8 var sambung = "Sedang " + bar;
9 console.log( sambung );    // Sedang Belajar JavaScript

```

Jadi, apa bedanya mengisi variabel dengan **tipe data primitif** dan **object constructor**?

Jika menggunakan *object constructor*, kita bisa mengakses puluhan method yang tersedia untuk object tersebut. Berikut contohnya:

```

1 var foo = new Number("52");
2 console.log( foo.toString(16) );      // 34
3 console.log( foo.toExponential(2));   // 5.20e+1
4
5 var bar = new String("Belajar JavaScript");
6
7 console.log( bar.toUpperCase() );     // BELAJAR JAVASCRIPT
8 console.log( bar.toLowerCase() );     // belajar javascript
9 console.log( bar.substr(3,9) );       // ajar Java

```

Seperti yang terlihat, dengan menggunakan *object constructor* kita bisa mengakses berbagai method untuk object foo dan bar. Method `toString()` dan `toExponential()` “melekat” ke **Object Number**. Sedangkan method `toUpperCase()`, `toLowerCase()`, dan `substr()` merupakan kepunyaan dari **Object String**.

Begitu juga halnya dengan object lain, punya method dan juga property yang bisa kita akses. Bahkan function juga memiliki *object constructor* sendiri, yang contoh penggunaannya adalah sebagai berikut:

```

1 var tambah = new Function('a', 'b', 'return a + b');
2
3 console.log( tambah(3,5) );
4 console.log( tambah.length ); // 2

```

Untuk membuat function tambah() menggunakan *object constructor*, argument pertama dan kedua diisi dengan argument untuk function tambah itu sendiri, yakni a dan b. Argument terakhir diisi string sebagai hasil akhir fungsi tambah, yakni `return a + b`.

Dengan menggunakan *object constructor* dalam membuat function, kita bisa mengakses property `length`, yang berisi banyaknya argument dari sebuah function. Karena fungsi tambah() memiliki 2 argumen, maka hasil dari `tambah.length` adalah 2.

Dari beberapa contoh ini terlihat bahwa membuat tipe data menggunakan *object constructor* lebih bagus daripada menggunakan tipe data primitif, karena kita bisa mengakses ratusan *method* dan *property* bawaan JavaScript. Tapi benarkah demikian?

## Selalu Pakai Tipe Data Primitif!

Sekali lagi, yang unik dari JavaScript, tipe data primitif ini ternyata juga bisa “meminjam” method dan property dari “kakaknya” yang berbentuk object. Berikut percobaannya:

```
1 var foo = 52;
2 console.log( foo.toString(16) );      // 34
3 console.log( foo.toExponential(2));   // 5.20e+1
4
5 var bar = "Belajar JavaScript";
6
7 console.log( bar.toUpperCase() );    // BELAJAR JAVASCRIPT
8 console.log( bar.toLowerCase() );    // belajar javascript
9 console.log( bar.substr(3,9) );       // ajar Java
```

Disini saya mengisi variabel `foo` dan `bar` menggunakan tipe data primitif, yakni `number` dan `string`. Ternyata kita tetap bisa mengakses method dan property yang seharusnya “kepunyaan” `object constructor`.

Secara internal, ketika kita memanggil *method* atau *property* dari tipe data primitif, JavaScript akan mengkonversinya menjadi Object untuk sementara, kemudian mengembalikannya kembali menjadi tipe data primitif.

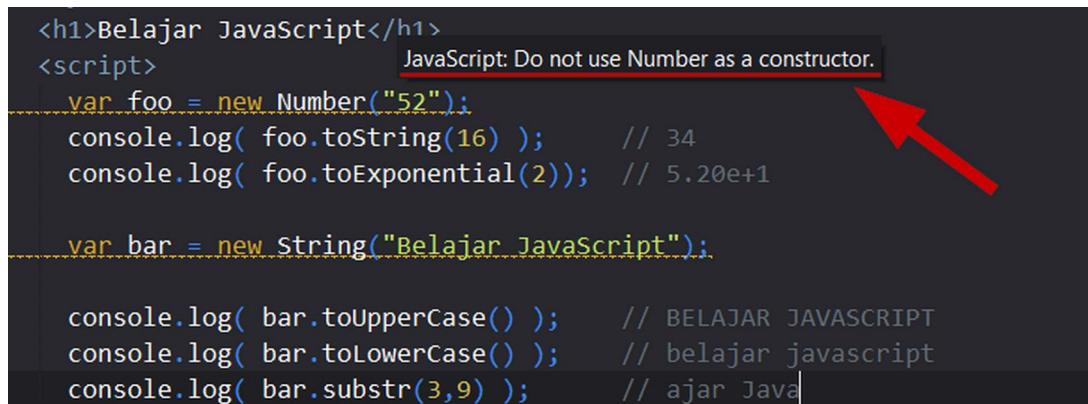
Proses ini dilakukan dengan sangat cepat sehingga tidak akan terdeteksi. Tapi inilah penjelasan kenapa kita bisa memanggil method dan property Object dari tipe data primitif.

Jadi jika hasilnya sama saja, mana yang lebih baik? Membuat variabel menggunakan **tipe data primitif** atau menggunakan **object constructor**?

Jawabannya: **lebih baik menggunakan tipe data primitif**. Karena `object constructor` butuh alokasi memory komputer yang lebih banyak dan waktu proses yang sedikit lebih lama.

Kita memang tidak akan merasakan perbedaannya karena mayoritas kode program yang ditulis hanya terdiri dari beberapa baris. Tapi untuk kode program yang kompleks dan terdiri dari ribuan baris, perbedaan performa akan cukup terasa.

Jika anda mencari referensi tentang *tipe data primitif vs object constructor*, hampir semuanya mengatakan untuk selalu pakai tipe data primitif. Teks editor **Komodo Edit** juga akan protes saat kita membuat tipe data dasar seperti `number` atau `string` dengan menggunakan `object constructor`.



```
<h1>Belajar JavaScript</h1>
<script>
    var foo = new Number("52");
    console.log( foo.toString(16) );      // 34
    console.log( foo.toExponential(2));   // 5.20e+1

    var bar = new String("Belajar JavaScript");

    console.log( bar.toUpperCase() );    // BELAJAR JAVASCRIPT
    console.log( bar.toLowerCase() );    // belajar javascript
    console.log( bar.substr(3,9) );      // ajar Java|
```

Gambar: Peringatan dari Komodo Edit agar tidak menggunakan object constructor

Kesimpulannya, untuk penggunaan sehari-hari tetap gunakan tipe data primitif. Penggunaan object constructor hanya diperlukan untuk kasus khusus, yang sangat jarang terjadi.

Akan tetapi saran ini hanya berlaku untuk tipe data yang memang tersedia bentuk primitifnya, seperti tipe data *number*, *string*, *boolean*, *function*, dan *array*. Untuk object JavaScript lain seperti **Math**, **Date** dan **RegExp**, satu-satunya cara adalah tetap menggunakan *object constructor*.

Terkait apa itu object **Math**, **Date** dan **RegExp** serta apa saja property dan method setiap Object ini, akan kita bahas secara bertahap mulai dari bab berikutnya.

---

Bahasan tentang OOP di JavaScript mungkin bisa menjadi materi tersulit di dalam buku ini (dan di dalam JavaScript secara keseluruhan). Tapi semoga anda bisa memahami materi yang saya berikan.

Jika merasa kurang paham atau ada yang bingung, silahkan ulangi lagi baca dari awal, sambil mencoba setiap kode program, termasuk object bentukan anda sendiri.

Berikutnya kita akan membahas apa saja property dan method yang ada di dalam object **Number** dan **Math** JavaScript.

# 12. Number Object

Pada akhir bab sebelumnya kita telah melihat terdapat berbagai property dan method yang “melekat” ke Object bawaan JavaScript. Kita akan mulai bahas dari object **Number**.



Jika anda butuh panduan teknis (referensi) tentang Object Number yang lebih detail, silahkan buka [Standard built-in objects: Number<sup>1</sup>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number) dari Mozilla Developer Network.

## 12.1 Cara Mengakses Property dan Method Object

Sebelum masuk ke apa saja property serta method yang disediakan oleh Object Number, mari kita lihat terlebih dahulu bagaimana cara mengaksesnya.

JavaScript menyimpan property dan method Object bawaan di 4 “tempat”:

- **Object property**
- **Object method**
- **Object instance property**
- **Object instance method**

**Object property** dan **Object method** adalah sebutan untuk “sesuatu” yang melekat langsung ke Object. Misalnya untuk mencari tau angka maksimum yang bisa disimpan JavaScript, bisa diakses dari property `Number.MAX_VALUE`. Sedangkan untuk mengecek apakah sebuah variabel berisi angka bulat atau tidak, bisa menggunakan method `Number.isInteger()`.

Perhatikan bahwa cara pemanggilan untuk *object property* dan *object method* adalah langsung dari Object-nya, yakni **Number**. Dalam konsep OOP, *object property* dan *object method* ini dikenal sebagai **static property** dan **static method**, yaitu property dan method yang melekat ke **Class**, bukan ke object hasil *instance class* tersebut.

Sedangkan **Object instance property** dan **Object instance method** adalah sebutan untuk “sesuatu” yang melekat ke *instance object*. Sebagai contoh, untuk memformat sebuah angka, kita bisa menggunakan method `toPrecision()` yang cara penggunaannya adalah sebagai berikut:

```
1 var foo = new Number(5.123456);
2
3 console.log( foo.toPrecision(3) ); // 5.12
```

Perhatikan bahwa method `toPrecision()` melekat ke variabel `foo`, yang merupakan hasil *instance* dari Object Number, bukan diakses langsung dari `Number`. Disini method `toPrecision()` termasuk ke dalam kelompok **Object instance method**.

Untuk **Object instance property**, contohnya seperti property `length` dari sebuah string:

---

<sup>1</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number)

```
1 var foo = new String("Selamat Belajar JavaScript");
2
3 console.log( foo.length );      // 26
```

Disini, property `length` melekat ke hasil *instance* dari Object `String`. Bukan ke object `String`-nya langsung.

Sebagai tambahan, seperti yang telah kita pelajari pada bab sebelumnya, seluruh property dan method bawaan JavaScript juga bisa diakses dari *tipe data primitif* object tersebut. Dan ini merupakan cara yang paling disarankan:

```
1 var foo = 5.123456;
2
3 console.log( foo.toPrecision(3) );      // 5.12
4
5 var foo = "Selamat Belajar JavaScript";
6
7 console.log( foo.length );            // 26
```

Agar mudah dibedakan, property dan method yang melekat ke Object-nya, akan saya tulis langsung dengan nama Object tersebut, misalnya:

- `Number.MAX_VALUE`
- `Number.MIN_VALUE`
- `Number.isInteger()`
- `Number.parseInt()`
- `Math.abs()`

Sedangkan untuk property dan method yang melekat ke hasil instance dari Object, ditulis dengan tambahan `prototype`:

- `String.prototype.length`
- `Number.prototype.toFixed()`
- `Number.prototype.toString()`

Cara penulisan seperti ini juga digunakan oleh [Mozilla Developer Network<sup>2</sup>](https://developer.mozilla.org/en-US/) (referensi yang saya pakai untuk membuat materi buku ini).

Yang juga perlu diketahui, tidak semua Object memiliki ke-4 cara penulisan ini. Contohnya, object `Number` tidak memiliki *object instance property*. Sedangkan object `Math` hanya memiliki *object property* dan *object method* saja, tapi tidak memiliki *object instance property* dan *object instance method*.

---

<sup>2</sup><https://developer.mozilla.org/en-US/>

## 12.2 Number Object Property

**Number** adalah object bawaan JavaScript yang berisi berbagai property dan method yang berkaitan dengan.. yup, angka. Mulai dari menformat angka, membulatkan angka, mengkonversi angka menjadi string, dll.

Kita akan mulai dari **object property** terlebih dahulu. Di dalam JavaScript terdapat 8 object property untuk **Number**:

- **Number.EPSILON**
- **Number.MAX\_SAFE\_INTEGER**
- **Number.MAX\_VALUE**
- **Number.MIN\_SAFE\_INTEGER**
- **Number.MIN\_VALUE**
- **Number.NaN**
- **Number.NEGATIVE\_INFINITY**
- **Number.POSITIVE\_INFINITY**

### 12.3 Property Number.EPSILON

Property **Number.EPSILON** berisi interval terkecil dari dua angka di dalam JavaScript. Berikut hasil pemanggilannya:

```
console.log( Number.EPSILON ); // 2.220446049250313e-16
```

### 12.4 Property Number.MAX\_SAFE\_INTEGER

Property **Number.MAX\_SAFE\_INTEGER** berisi nilai maksimum dari angka bulat (integer) di dalam JavaScript, yang nilainya adalah  $2^{53} - 1$ . Angka ini sesuai dengan standar IEEE-754 double precision.

```
console.log( Number.MAX_SAFE_INTEGER ); // 9007199254740991
```

### 12.5 Property Number.MAX\_VALUE

Property **Number.MAX\_VALUE** berisi angka tertinggi yang bisa ditampung di dalam JavaScript.

```
console.log( Number.MAX_VALUE ); // 1.7976931348623157e+308
```

### 12.6 Property Number.MIN\_SAFE\_INTEGER

Property **Number.MIN\_SAFE\_INTEGER** berisi nilai minimum dari sebuah angka bulat (integer) di dalam JavaScript. Dimana nilainya adalah  $-(2^{53} - 1)$ .

```
console.log( Number.MIN_SAFE_INTEGER ); // -9007199254740991
```

## 12.7 Property Number.MIN\_VALUE

Property Number.MIN\_VALUE berisi angka positif terkecil yang bisa ditampung JavaScript, yakni nilai pecahan yang sangat mendekati nol, tapi bukan nol.

```
console.log( Number.MIN_VALUE ); // 5e-324
```

## 12.8 Property Number.NaN

Property Number.NaN berisi nilai khusus yang menyatakan *not a number*. Ini adalah cara untuk membuat nilai NaN.

```
console.log( Number.NaN ); // NaN
```

## 12.9 Property Number.NEGATIVE\_INFINITY

Property Number.NEGATIVE\_INFINITY berisi nilai khusus yang menyatakan *-infinity*. Ini adalah cara untuk membuat nilai *-infinity*.

```
console.log( Number.NEGATIVE_INFINITY ); // -Infinity
```

## 12.10 Property Number.POSITIVE\_INFINITY

Property Number.POSITIVE\_INFINITY berisi nilai khusus yang menyatakan *infinity*. Ini adalah cara untuk membuat nilai *infinity*.

```
console.log( Number.POSITIVE_INFINITY ); // Infinity
```

Dari ke-8 Number object property ini, sebagian besar berisi konstanta dasar JavaScript dan kita jarang memerlukannya.

## 12.11 Number Object Method

Object Number memiliki 6 method:

- Number.isNaN()
- Number.isFinite()
- Number.isInteger()
- Number.isSafeInteger()
- Number.parseFloat()
- Number.parseInt()

Mari kita bahas.

## 12.12 Method Number.isNaN()

Method `Number.isNaN()` digunakan untuk mengecek apakah hasil operasi / suatu variabel berisi `NaN` atau bukan. Hasilnya `true` jika itu `NaN`, dan `false` jika bukan `NaN`:

```
1 var foo;  
2  
3 foo = 5 ;  
4 console.log( Number.isNaN(foo) ); // false  
5  
6 foo = 5/'a' ;  
7 console.log( Number.isNaN(foo) ); // true  
8  
9 foo = Number.NaN ;  
10 console.log( Number.isNaN(foo) ); // true
```

Method ini cukup penting karena di dalam JavaScript operasi perbandingan `NaN == NaN`, maupun `NaN === NaN` akan menghasilkan `false`. Satu-satunya cara untuk memastikan apakah isi sebuah variabel itu `NaN` atau bukan adalah menggunakan method `isNaN()`:

```
1 var foo;  
2  
3 foo = NaN ;  
4 console.log( Number.isNaN(foo) ); // true  
5  
6 foo = NaN == NaN;  
7 console.log( foo ); // false  
8  
9 foo = NaN === NaN;  
10 console.log( foo ); // false
```

## 12.13 Method Number.isFinite()

Method `Number.isFinite()` digunakan untuk mengecek apakah sebuah nilai/variabel berisi angka yang bisa dihitung. Fungsi ini mengembalikan boolean `true` jika nilai tersebut bisa dihitung (berupa angka biasa), dan mengembalikan nilai `false` jika berupa `infinity`, `NaN`, atau bukan tipe data `Number`.

```
1 var foo;  
2  
3 foo = 6 ;  
4 console.log( Number.isFinite(foo) ); // true  
5  
6 foo = 3.21456 ;  
7 console.log( Number.isFinite(foo) ); // true  
8  
9 foo = 'a';  
10 console.log( Number.isFinite(foo) ); // false  
11  
12 foo = 1/0 ;  
13 console.log( Number.isFinite(foo) ); // false  
14  
15 foo = Number.NEGATIVE_INFINITY ;  
16 console.log( Number.isFinite(foo) ); // false  
17  
18 foo = Number.NaN ;  
19 console.log( Number.isFinite(foo) ); // false
```

## 12.14 Method Number.isInteger() dan Number.isSafeInteger()

Kedua method ini digunakan untuk memeriksa apakah suatu nilai/variabel berisi angka integer. Jika berupa integer, hasilnya adalah **true**, jika tidak integer hasilnya **false**.

Bedanya, pada method `Number.isSafeInteger()`, nilai integer dibatasi sesuai standar IEEE-754 double precision, yakni nilai maksimum  $2^{53} - 1$ . Jika lebih dari itu, method `Number.isSafeInteger()` akan mengembalikan nilai **false**.

```
1 var foo;  
2  
3 foo = 6 ;  
4 console.log( Number.isInteger(foo) ); // true  
5 console.log( Number.isSafeInteger(foo) ); // true  
6  
7 foo = 3.21456 ;  
8 console.log( Number.isInteger(foo) ); // false  
9 console.log( Number.isSafeInteger(foo) ); // false  
10  
11 foo = 1/0 ;  
12 console.log( Number.isInteger(foo) ); // false  
13 console.log( Number.isSafeInteger(foo) ); // false  
14  
15 foo = 9007199254740992 ;
```

```

16 console.log( Number.isInteger(foo) );      // true
17 console.log( Number.isSafeInteger(foo) ); // false

```

Dalam variabel `foo` terakhir, nilainya adalah `9007199254740992`, ini sudah melebihi batas maksimum  $2^{53} - 1$ , karena itulah hasil dari `Number.isSafeInteger(foo)` akan mengembalikan nilai `false`.

## 12.15 Method Number.parseFloat()

Method `Number.parseFloat()` digunakan untuk mengkonversi nilai atau variabel ke bentuk angka *float*. *Float* adalah angka yang memiliki nilai pecahan, serta juga termasuk angka bulat (integer).

Method `Number.parseFloat()` biasanya dipakai untuk mengkonversi string menjadi `Number`. Jika string tersebut tidak bisa dikonversi menjadi number, method ini akan mengembalikan nilai `Nan`. Berikut contoh penggunaannya:

```

1 var foo = "1234";
2 console.log( typeof foo ); // string
3
4 foo = Number.parseFloat(foo);
5 console.log( foo ); // 1234
6 console.log( typeof foo ); // number
7
8 var bar = "-1234.5678";
9 console.log( typeof bar ); // string
10
11 bar = Number.parseFloat(bar);
12 console.log( bar ); // -1234.5678
13 console.log( typeof bar ); // number
14
15 console.log( Number.parseFloat("12.045 potong ayam") ); // 12.045
16 console.log( Number.parseFloat("Ada 12.045 potong ayam") ); // NaN

```

Di awal kode program terlihat bahwa variabel `foo` dan `bar` sebenarnya bertipe string. Setelah menggunakan method `Number.parseFloat()`, hasilnya menjadi tipe data `number`.

Contoh 2 baris terakhir cukup menarik. Apabila sebuah string diawali dengan angka, angka tersebut akan dicoba di konversi menjadi `number`, hingga ditemukan karakter yang bukan angka. Namun jika di awal string sudah bukan angka, method `Number.parseFloat()` akan mengembalikan nilai `NaN`.

## 12.16 Method Number.parseInt()

Method `Number.parseInt()` sangat mirip seperti `Number.parseFloat()`, bedanya method ini akan mengkonversi nilai atau variabel menjadi angka integer (angka bulat). Jika di dalam string

asal terdapat nilai pecahan, bagian pecahan akan dibuang. Jika tidak bisa dikonversi menjadi number, method ini mengembalikan nilai NaN.

Berikut contohnya penggunaannya:

```

1 var foo = "1234";
2 console.log( typeof foo );      // string
3
4 foo = Number.parseInt(foo) ;
5 console.log( foo );           // 1234
6 console.log( typeof foo );    // number
7
8 var bar = "-1234.5678";
9 console.log( typeof bar );    // string
10
11 bar = Number.parseInt(bar) ;
12 console.log( bar );          // -1234
13 console.log( typeof bar );   // number
14
15 console.log( Number.parseInt("12.045 potong ayam") ); // 12
16 console.log( Number.parseInt("Ada 12.045 potong ayam") ); // NaN

```

Terlihat bahwa string "-1234.5678" dikonversi menjadi number -1234 (nilai di belakang koma akan dibuang).

Method `Number.parseInt()` memiliki 1 lagi argumen kedua yang bersifat opsional. Argumen kedua ini bisa diisi dengan `radix`, yakni basis bilangan. Nilai default dari `radix` adalah 10, dimana method `Number.parseInt()` akan menggunakan bilangan basis 10 (bilangan desimal) jika argumen kedua dari method ini tidak ditulis.

Jika `radix` diisi angka 2, artinya kita ingin JavaScript memproses argument pertama sebagai bilangan biner. `Radix` 8 untuk bilangan oktadesimal, dan `radix` 16 untuk bilangan hexadesimal.

Berikut contoh penggunaannya:

```

1 console.log( Number.parseInt("10101101",2) ); // 173
2 console.log( Number.parseInt("255",8) );       // 173
3 console.log( Number.parseInt("AD",16) );       // 173
4 console.log( Number.parseInt("173",10) );      // 173
5 console.log( Number.parseInt("173 ikan",10) ); // 173

```

Kode `Number.parseInt("10101101",2)` artinya saya ingin memproses string "10101101" sebagai bilangan biner (`radix` = 2). Oleh JavaScript, hasil akhirnya dikonversi menjadi angka desimal 173.

Untuk `Number.parseInt("255",8)`, disini saya ingin memproses string "255" sebagai bilangan oktal. Dimana radixnya = 8. Begitu juga dengan `Number.parseInt("AD",16)` yang menggunakan radix 16 (bilangan hexadesimal).

JavaScript mendukung nilai radix 2 hingga 36, tp memang yang sering digunakan hanya radix 2, 8, 10, dan 16.

## 12.17 Number Instance Method

**Number instance method** adalah method JavaScript yang melekat ke object hasil instance dari Number. Number sendiri tidak memiliki instance property, tapi hanya instance method.

Terdapat 6 Number Instance Method di dalam JavaScript:

- `Number.prototype.toExponential()`
- `Number.prototype.toFixed()`
- `Number.prototype.toPrecision()`
- `Number.prototype.toString()`
- `Number.prototype.toLocaleString()`
- `Number.prototype.valueOf()`

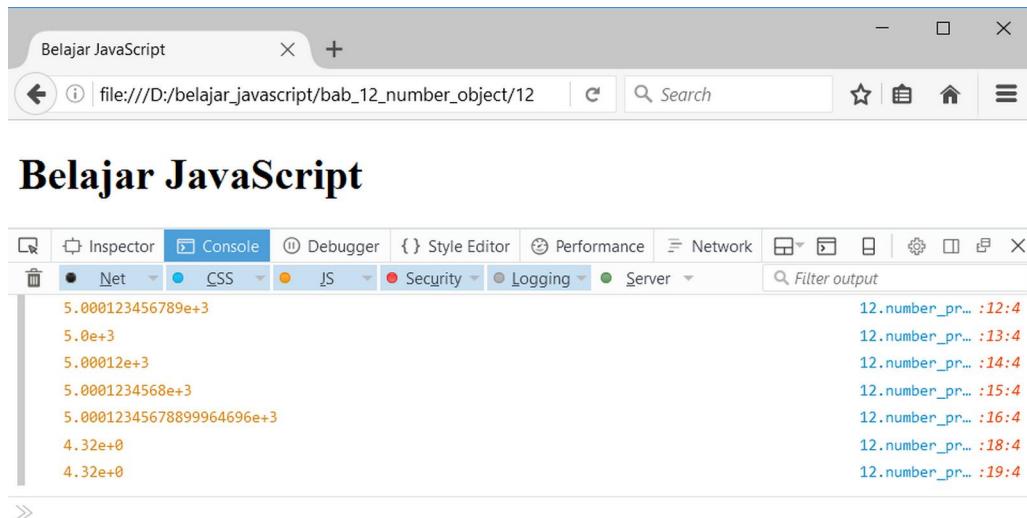
## 12.18 Method Number.prototype.toExponential()

Method `toExponential` digunakan untuk meformat tampilan angka menjadi *scientific notation*. Scientific Notation adalah bentuk tampilan angka dengan 1 digit sebelum tanda desimal, kemudian diikuti dengan tanda pangkat.

Misalnya angka 123.45, jika ditulis ke dalam bentuk scientific notation akan menjadi  $1.23 \times 10^2$ . Dalam bahasa pemrograman, pangkat sepuluh ini diganti dengan karakter e atau E. Sehingga  $1.23 \times 10^2$  ditulis menjadi 1.23e+2.

Method `toExponential` menyediakan 1 argumen opsional yang jika diinput akan menentukan ‘panjang’ digit untuk pecahan. Argumen ini bisa diisi dengan angka 0 – 20. Apabila tidak menggunakan argumen, method `toExponential` akan memakai panjang paling maksimal sesuai jumlah digit dalam variabel asal.

```
1 var foo = 5000.123456789;  
2  
3 console.log( foo.toExponential() );      // 5.000123456789e+3  
4 console.log( foo.toExponential(1) );      // 5.0e+3  
5 console.log( foo.toExponential(5) );      // 5.00012e+3  
6 console.log( foo.toExponential(10) );     // 5.0001234568e+3  
7 console.log( foo.toExponential(20) );     // 5.00012345678899964696e+3  
8  
9 console.log( 4.32345.toExponential(2) );  // 4.32e+0  
10 console.log( (4.32345).toExponential(2) ); // 4.32e+0
```



Gambar: Tampilan method `toExponential(5)` dengan berbagai nilai argumen

Perintah `foo.toExponential(5)` artinya, tampilkan variabel `foo` menggunakan format scientific notation dengan 5 digit pecahan (5 digit angka dibelakang tanda titik).

Untuk 2 contoh terakhir, dapat anda perhatikan bahwa *instance method* Number bisa dipanggil langsung dari sebuah angka (tidak perlu disimpan ke dalam sebuah variabel terlebih dahulu). Ini juga berlaku untuk *instance method* lainnya. Penggunaan tanda kurung direkomendasikan agar mudah membedakan karakter titik untuk angka desimal dengan dot notation untuk mengakses method.

## 12.19 Method Number.prototype.toFixed()

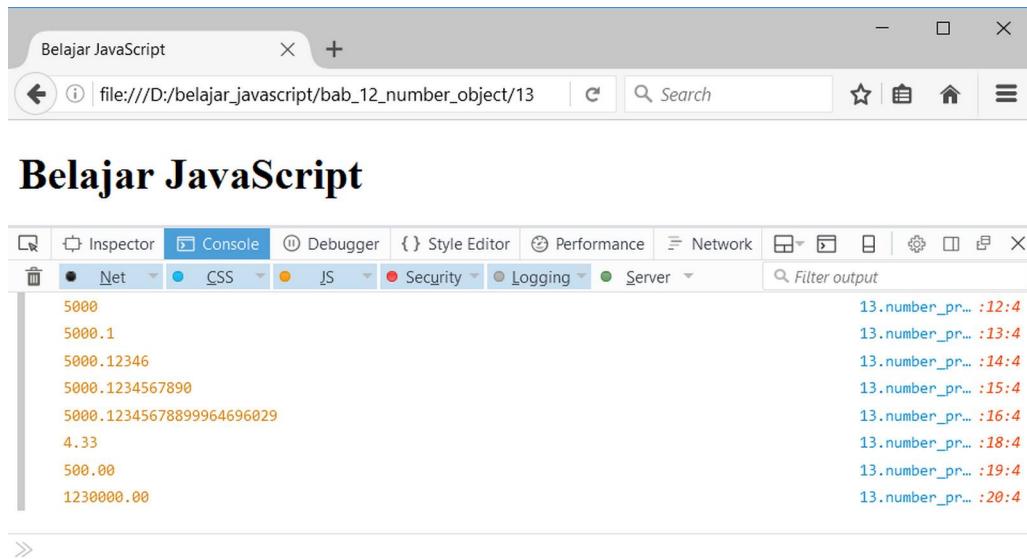
Method `toFixed` digunakan untuk membuat tampilan angka dengan jumlah digit desimal yang tetap. Method ini membutuhkan satu argumen opsional berupa jumlah digit setelah tanda desimal. Jika dipanggil tanpa argumen, akan dianggap 0 sehingga hasilnya tanpa angka pecahan.

Angka hasil pemanggilan method `toFixed` dibulatkan ke bilangan terdekat (0,5 akan menjadi 1). Jika angka yang ingin ditampilkan melebihi digit asal, sisa digit diisi angka 0. Berikut contoh penggunaannya:

```

1 var foo = 5000.123456789;
2
3 console.log( foo.toFixed() );      // 5000
4 console.log( foo.toFixed(1) );     // 5000.1
5 console.log( foo.toFixed(5) );     // 5000.12346
6 console.log( foo.toFixed(10) );    // 5000.1234567890
7 console.log( foo.toFixed(20) );    // 5000.12345678899964696029
8
9 console.log( (4.32745).toFixed(2) ); // 4.33
10 console.log( (500).toFixed(2) );    // 500.00
11 console.log( (1.23e+6).toFixed(2) ); // 1230000.00

```



Gambar: Hasil method `toFixed()` dengan berbagai nilai argumen

Disini saya memiliki variabel `foo` dengan nilai angka `5000.123456789`. Hasil pemanggilan method `foo.toFixed()` adalah `5000`. Karena jika method `toFixed()` dipanggil tanpa argument, artinya sama dengan `foo.toFixed(0)`.

Hasil pemanggilan `foo.toFixed(5)` adalah `5000.12346`, bukan `5000.12345`, sebab digit ke-5 dibulatkan keatas.

Pemanggilan `foo.toFixed(20)` tampak tidak sesuai, seharusnya hasil yang ditampilkan adalah `5000.1234567890000000000000`. Kemungkinan besar ini terjadi karena keterbatasan tipe data number di dalam JavaScript (rounding error).

Jika sebuah angka tidak memiliki digit desimal, namun dipanggil dengan method `toFixed(2)`, angka desimal ditambah dengan 00, seperti `(500).toFixed(2)` yang hasilnya menjadi `500.00`.

## 12.20 Method Number.prototype.toPrecision()

Method `toPrecision()` mirip seperti `toFixed()`, bedanya angka yang ditulis di dalam argumen merupakan total dari seluruh digit.

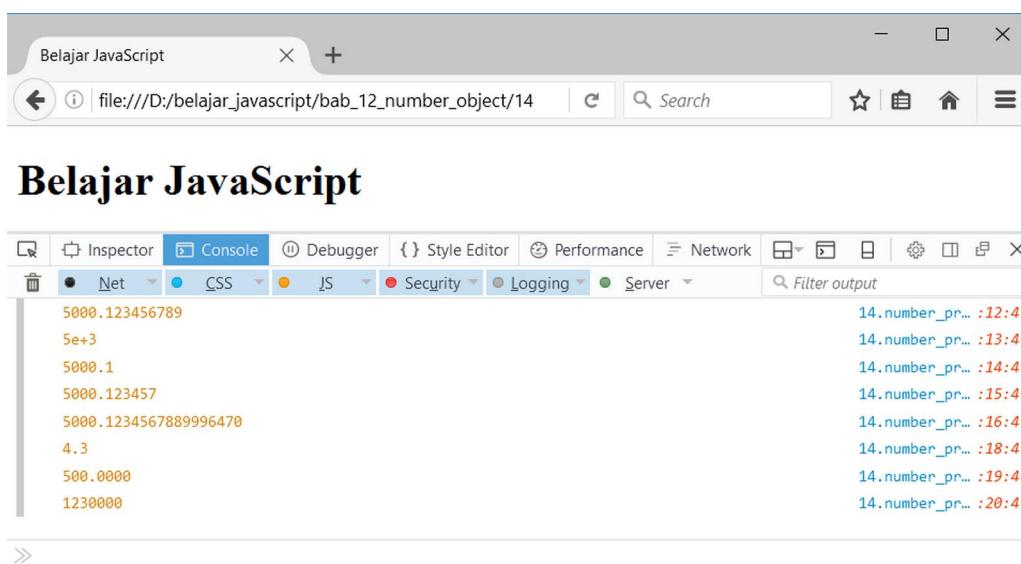
Sebagai contoh, `foo.toPrecision(3)` artinya tampilan variabel `foo` dengan 3 digit angka. Jika jumlah digit di variabel `foo` kurang dari 3, tambahkan angka 0 di bagian desimal. Jika jumlah digitnya lebih, akan dibulatkan dan jika perlu diformat ke bentuk scientific notation.

Berikut contoh penggunaannya:

```

1 var foo = 5000.123456789;
2
3 console.log( foo.toPrecision() );      // 5000.123456789
4 console.log( foo.toPrecision(1) );     // 5e+3
5 console.log( foo.toPrecision(5) );     // 5000.1
6 console.log( foo.toPrecision(10) );    // 5000.123457
7 console.log( foo.toPrecision(20) );    // 5000.1234567889996470
8
9 console.log( (4.32745).toPrecision(2) ); // 4.3
10 console.log( (500).toPrecision(7) );   // 500.0000
11 console.log( (1.23e+6).toPrecision(7) ); // 1230000

```



Gambar: Hasil method `foo.toPrecision()` dengan berbagai nilai argumen

Pemanggilan method `foo.toPrecision()` tanpa argumen tidak merubah tampilan apapun dari variabel `foo`.

Untuk `foo.toPrecision(1)`, artinya tampilkan nilai variabel `foo` dengan 1 digit angka. Namun karena `foo` berisi lebih dari 1 digit dan memiliki nilai desimal (pecahan), nilai pecahan akan dibulatkan terlebih dahulu, kemudian `foo` ditampilkan dengan bentuk scientific notation: `5e+3`.

Kenapa dalam bentuk scientific notation? Ini karena kita mensyaratkan `toPrecision(1)` sedangkan variabel asal `foo` berisi: 5000 (angka desimal sudah dibulatkan). Sehingga agar menjadi “satu angka”, digunakan scientific notation: `5e+3`.

Untuk `foo.toPrecision(5)`, artinya tampilkan nilai variabel `foo` dengan 5 angka. Karena angka bulat dari `foo` sudah mengambil tempat 4 digit: 5000, hanya ada sisa 1 tempat untuk nilai desimal, yang dibulatkan ke angka 1. Hasilnya: 5000.1.

Kesalahan pembulatan juga terjadi saat pemanggilan `foo.toPrecision(20)`. Alasannya juga sama seperti method `toFixed()`, yakni rounding error dari JavaScript.

Bagaimana jika digitnya kurang? ketika saya menjalankan `(500).toPrecision(7)`, artinya tampilkan angka 500 dengan 7 digit. Namun karena “500” baru mengambil 3 digit, sisanya ditambahkan angka 0 sebagai nilai pecahan: 500.0000 (total 7 digit).

Di contoh terakhir, terlihat bahwa method `toPrecision` juga mendukung penulisan scientific notation sebagai nilai asal: `(1.23e+6).toPrecision(7)`, dimana hasilnya adalah: `1230000`.

## 12.21 Method Number.prototype.toString()

Method `toString()` digunakan untuk mengkonversi tipe data number menjadi tipe data string. Method ini juga otomatis dijalankan ketika kita menampilkan tipe data number menggunakan perintah `alert()` maupun `console.log()`, hanya saja jika butuh mengkonversi tipe data number secara manual, bisa menggunakan method `toString()`.

Berikut contoh penggunaannya:

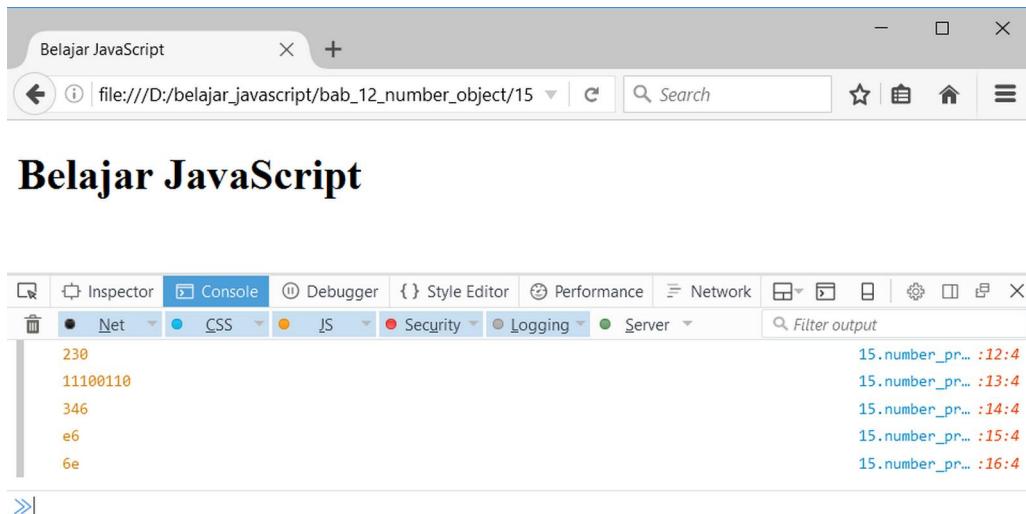
```
1 var foo = 230;
2 console.log( typeof foo ); // number
3
4 bar = foo.toString();
5 console.log( typeof bar ); // string
6 console.log( bar ); // 230
7
8 var foo1 = new Number(230);
9 console.log( typeof foo1 ); // object
10
11 bar1 = foo1.toString();
12 console.log( typeof bar1 ); // string
13 console.log( bar1 ); // 230
```

Disini saya membuat variabel `foo` dan `foo1`. Keduanya berisi tipe data primitif `number`, dan `object Number`. Ini terlihat dari hasil operator `typeof`. Setelah pemanggilan method `toString()`, hasilnya menjadi tipe data `string`.

Hampir setiap tipe data di dalam JavaScript memiliki method `toString()`, termasuk `boolean`, `array` bahkan `function`. Fungsinya sama, yakni mengkonversi tipe data tersebut menjadi `string`.

Khusus untuk tipe data `Number`, method `toString()` bisa diisi dengan 1 argument opsional, yakni `radix` untuk mengkonversi angka tersebut ke basis bilangan tertentu. Radix bisa diisi angka 2 hingga 36. Berikut contohnya:

```
1 var foo = 230;
2
3 console.log( foo.toString() ); // 230
4 console.log( foo.toString(2) ); // 11100110
5 console.log( foo.toString(8) ); // 346
6 console.log( foo.toString(16) ); // e6
7 console.log( foo.toString(36) ); // 6e
```



Gambar: Tampilan method `toString()` dengan berbagai radix

Saya menampilkan angka 230 desimal (basis 10), ke dalam bentuk biner (basis 2), oktal (basis 8), heksadesimal (basis 16) dan hexatrigesimalimal (basis 36). Jika method `toString()` dipanggil tanpa argumen, nilai default adalah 10 (bilangan desimal).

## 12.22 Method Number.prototype.toLocaleString()

Method `toLocaleString()` juga digunakan untuk mengkonversi tipe data number menjadi tipe data string, namun menggunakan format angka lokal yang ada di sistem (*locale*).

Berikut hasil yang saya dapatkan:

```

1 var foo = 1234500.346;
2
3 console.log( foo.toString() );           // 1234500.346
4 console.log( foo.toLocaleString() );    // 1.234.500,346

```

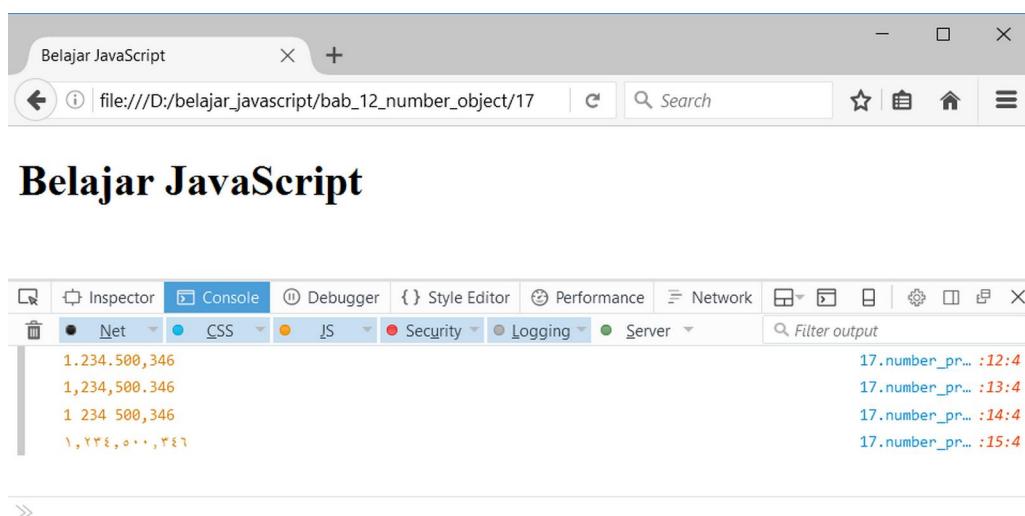
Terlihat hasil pemanggilan method `toLocaleString()` langsung di format dengan tanda titik sebagai pemisah ribuan, dan tanda koma sebagai pemisah pecahan. Ini karena komputer saya di set ke settingan Indonesia. Jika komputer yang saya gunakan di set ke settingan US (Amerika), yang tampil adalah 1,234,500.346 (karakter koma sebagai pemisah ribuan, dan titik sebagai pemisah pecahan).

Method `toLocaleString()` juga bisa diisi dengan argumen opsional yakni kode bahasa yang menentukan settingan local:

```

1 var foo = 1234500.346;
2
3 console.log( foo.toLocaleString('id-ID')); // 1.234.500,346
4 console.log( foo.toLocaleString('en-GB')); // 1,234,500.346
5 console.log( foo.toLocaleString('fr-FR')); // 1 234 500,346
6 console.log( foo.toLocaleString('ar-SA')); // ١٢٣٤٥٠٠٣٤٦

```



Gambar: Method `toLocaleString()` menampilkan angka dengan format sesuai bahasa dan negara

Kode bahasa (locale code) terdiri dari 4 karakter, 2 karakter pertama adalah nama bahasanya, sedangkan 2 karakter terakhir adalah lokasi/negara. Sebagai contoh 'id-ID' artinya Bahasa Indonesia (indonesian - id) di negara Indonesia (ID). Kode 'en-US' artinya bahasa inggris (english - en) di Amerika Serikat (US).

Kenapa harus menggunakan bahasa dan negara? Karena bisa jadi terdapat bahasa yang sama tapi dengan aturan dan dialek yang berbeda. Misalnya terdapat 'en-AU', yakni bahasa inggris di Australia, atau 'en-GB', yakni bahasa inggris di Negara Inggris (Great Britain). Kode-kode seperti ini umum ditemukan untuk aplikasi multibahasa.

Dalam kode diatas, saya menampilkan format angka dalam 4 bahasa: Indonesia (ID), Bahasa Inggris (US), Bahasa Perancis (France), dan bahasa Arab (Saudi Arabia). Seperti yang terlihat, method `toLocaleString()` menampilkan angka dalam format yang berbeda-beda.



Untuk daftar kode bahasa dan negara, bisa anda lihat di [Table of locales<sup>3</sup>](#).

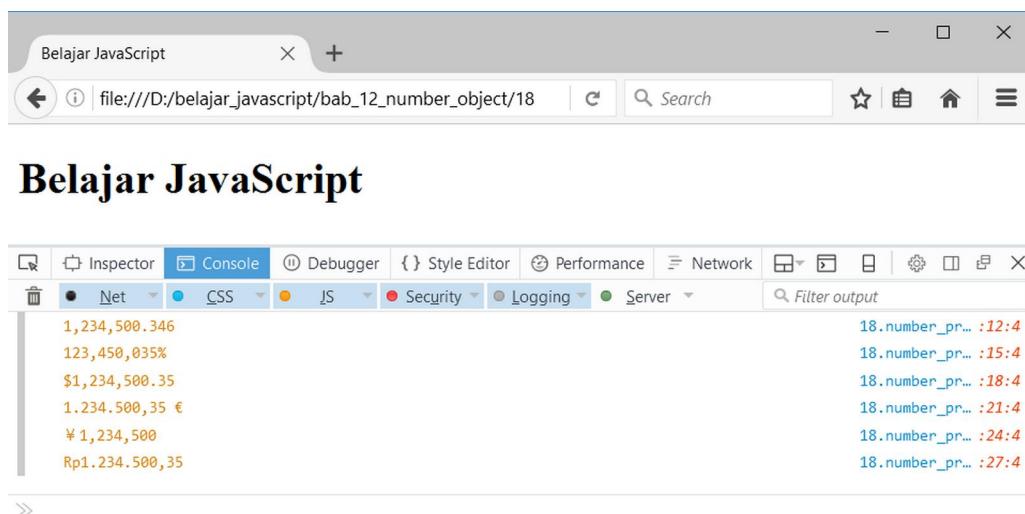
Method `toLocaleString()` juga memiliki argumen kedua berupa object dengan banyak konfigurasi. Saya tidak akan membahas semua konfigurasi ini, tapi ada beberapa yang cukup menarik, yakni pengaturan `style` dan `currency`. Berikut contoh penggunaannya:

<sup>3</sup>[https://docs.moodle.org/dev/Table\\_of\\_locales](https://docs.moodle.org/dev/Table_of_locales)

```

1 var foo = 1234500.346;
2
3 console.log( foo.toLocaleString('en-US', { style: 'decimal' } )); // 1,234,500.346
4
5 console.log( foo.toLocaleString('en-US', { style: 'percent' } )); // 123,450,035%
6
7 console.log( foo.toLocaleString('en-US', { style: 'currency', currency: 'USD' } )); // $1,234,500.35
8
9 console.log( foo.toLocaleString('de-DE', { style: 'currency', currency: 'EUR' } )); // 1.234.500,35 €
10
11 console.log( foo.toLocaleString('ja-JP', { style: 'currency', currency: 'JPY' } )); // 1,234,500
12
13 console.log( foo.toLocaleString('id-ID', { style: 'currency', currency: 'IDR' } )); // Rp1.234.500,35

```



Gambar: Setingen style dan currency pada method toLocaleString()

Pada 2 contoh pertama, saya menambahkan argument kedua untuk method `toLocaleString()` yang isinya berupa object dengan 1 property: `style`, yang ditulis dalam bentuk object: `{ style: 'decimal' }`. Penulisan seperti umum digunakan untuk method yang memiliki banyak konfigurasi.

Pengaturan `style` ada 3 macam: `decimal` (pilihan default), `percent`, dan `currency`. Jika kita memilih `style: 'currency'`, maka harus disertakan property kedua, yakni `currency`. Property `currency` digunakan untuk menginput kode mata uang yang ingin ditampilkan. Bisa anda perhatikan bahwa setiap `currency` memiliki cara penulisan masing-masing. Untuk mata uang euro, tanda euro di tampilkan di belakang.

-  Berbagai settingan lain terkait method `toLocaleString()` bisa anda lihat di [Number.prototype.toLocaleString\(\)](#)<sup>4</sup> dari Mozilla Developer Network.

## 12.23 Method Number.prototype.valueOf()

Method `valueOf()` umumnya digunakan secara internal oleh JavaScript, dan kita jarang mengaksesnya secara langsung. Method ini digunakan untuk mengkonversi Object Number menjadi tipe data primitif number.

```
1 var foo = new Number(230);
2 console.log( typeof foo ); // object
3
4 bar = foo.valueOf();
5 console.log( typeof bar ); // number
6 console.log( bar ); // 230
```

---

Dalam bab ini kita telah membahas berbagai property dan method dari tipe data **Number**. JavaScript masih memiliki 1 lagi object yang menyimpan berbagai property dan method yang berhubungan dengan angka, yakni object **Math**. Inilah yang akan kita pelajari dalam bab berikutnya.

---

<sup>4</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Number/toLocaleString](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toLocaleString)

# 13. Math Object

**Math** merupakan object JavaScript yang berisi berbagai property dan method untuk pemrosesan angka. Berbeda dengan object **Number**, seluruh property dan method untuk Math object, melekat ke objectnya langsung. Math object tidak memiliki *instance property* maupun *instance method*.



Jika anda butuh panduan teknis (referensi) tentang Object Math yang lebih detail, silahkan buka [Standard built-in objects: Math<sup>1</sup>](#) dari Mozilla Developer Network.

## 13.1 Math Object Property

Terdapat 8 property dari Math object, yang semuanya merupakan konstanta matematis:

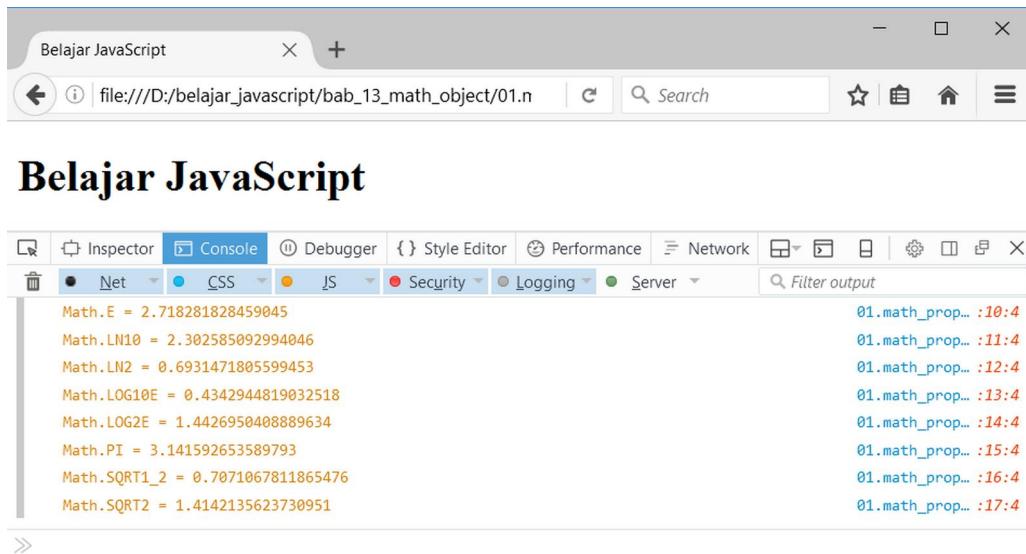
- **Math.E**: Berisi angka logaritma natural e, dengan nilai 2.718281828459045.
- **Math.LN10**: Berisi angka logaritma natural 10, dengan nilai 2.302585092994046.
- **Math.LN2**: Berisi angka logaritma natural 2, dengan nilai 0.6931471805599453.
- **Math.LOG10E**: Berisi angka logaritma natural e basis 10, dengan nilai 0.4342944819032518.
- **Math.LOG2E**: Berisi angka logaritma natural e basis 2, dengan nilai 1.4426950408889634.
- **Math.PI**: Berisi angka pi ( $\pi$ ) dengan nilai 3.141592653589793.
- **Math.SQRT1\_2**: Berisi angka 1 dibagi dengan akar kuadrat 2, dengan nilai 0.707106781186.
- **Math.SQRT2**: Berisi angka akar kuadrat dari 2, dengan nilai 1.4142135623730951.

Berikut hasil pemanggilannya:

```
1 console.log("Math.E = " + Math.E);           // 2.718281828459045
2 console.log("Math.LN10 = " + Math.LN10);      // 2.302585092994046
3 console.log("Math.LN2 = " + Math.LN2);         // 0.6931471805599453
4 console.log("Math.LOG10E = " + Math.LOG10E);    // 0.4342944819032518
5 console.log("Math.LOG2E = " + Math.LOG2E);      // 1.4426950408889634
6 console.log("Math.PI = " + Math.PI);            // 3.141592653589793
7 console.log("Math.SQRT1_2 = " + Math.SQRT1_2);  // 0.7071067811865476
8 console.log("Math.SQRT2 = " + Math.SQRT2);      // 1.4142135623730951
```

---

<sup>1</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Math](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math)



Gambar: Hasil pemanggilan Math property

Kita bisa menggunakan konstanta ini dalam perhitungan matematis, seperti contoh berikut:

```

1 var jari2 = 7;
2 var luasLingkaran = Math.PI * jari2 * jari2;
3 console.log( luasLingkaran ); // 153.93804002589985

```

Sebagai latihan tambahan, bisakah anda menformat tampilan hasil luasLingkaran dengan 2 posisi desimal? Silahkan pakai method bawaan tipe data **Number** yang sudah kita pelajari dalam bab sebelumnya.

Method apa yang bisa dipakai? kita bisa menggunakan method **toFixed()**:

```

1 var jari2 = 7;
2 var luasLingkaran = Math.PI * jari2 * jari2;
3 console.log( luasLingkaran.toFixed(2) ); // 153.94

```

## 13.2 Math Object Method

Math object memiliki banyak method, total terdapat 35 method:

- **Math.abs(x)**
- **Math.acos(x)**
- **Math.acosh(x)**
- **Math.asin(x)**
- **Math.asinh(x)**
- **Math.atan(x)**
- **Math.atanh(x)**
- **Math.atan2(y, x)**

- `Math.cbrt(x)`
- `Math.ceil(x)`
- `Math.clz32(x)`
- `Math.cos(x)`
- `Math.cosh(x)`
- `Math.exp(x)`
- `Math.expm1(x)`
- `Math.floor(x)`
- `Math.fround(x)`
- `Math.hypot([x[, y[, ...]]])`
- `Math.imul(x, y)`
- `Math.log(x)`
- `Math.log1p(x)`
- `Math.log10(x)`
- `Math.log2(x)`
- `Math.max([x[, y[, ...]]])`
- `Math.min([x[, y[, ...]]])`
- `Math.pow(x, y)`
- `Math.random()`
- `Math.round(x)`
- `Math.sign(x)`
- `Math.sin(x)`
- `Math.sinh(x)`
- `Math.sqrt(x)`
- `Math.tan(x)`
- `Math.tanh(x)`
- `Math.trunc(x)`

Saya tidak akan membahas semua method ini, karena seperti yang terlihat, sebagian besar terkait dengan rumus matematika. Kecuali anda sedang membuat aplikasi matematis atau yang melibatkan statistik, rumus-rumus ini sangat jarang dipakai.

Kita akan membahas beberapa yang penting saja. Jika anda berminat mempelajari semuanya, bisa mengunjungi [Standard built-in objects: Math<sup>2</sup>](#) dari Mozilla Developer Network.

### 13.3 Method `Math.ceil(x)`, `Math.floor(x)`, dan `Math.round(x)`

Ketiga method ini digunakan untuk membulatkan angka ke integer terdekat (angka bulat). Masing-masingnya menerima 1 argumen berupa angka yang ingin dibulatkan.

---

<sup>2</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Math](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math)

Method `Math.ceil()` digunakan untuk pembulatan ke atas, `Math.floor()` untuk pembulatan ke bawah, dan `Math.round()` untuk pembulatan ke bawah jika kurang dari 0,5 dan pembulatan ke atas jika lebih dari 0,5 (termasuk angka 0,5).

Berikut contoh penggunaannya:

```
1 var foo = 12.5;
2
3 console.log( Math.floor(foo) ); // 12
4 console.log( Math.ceil(foo) ); // 13
5 console.log( Math.round(foo) ); // 13
6
7 var bar = 12.4;
8
9 console.log( Math.floor(bar) ); // 12
10 console.log( Math.ceil(bar) ); // 13
11 console.log( Math.round(bar) ); // 12
```

Bagaimana jika kita ingin membulatkan angka bukan ke bentuk integer, tapi (misalnya) ke posisi 3 angka desimal? Ini bisa diakali dengan mengalikan nilai tersebut dengan 1000, jalankan method, lalu bagi kembali dengan 1000:

```
1 var foo = 12.1354;
2
3 console.log( Math.floor(foo) ); // 12
4 console.log( Math.ceil(foo) ); // 13
5 console.log( Math.round(foo) ); // 12
6
7 console.log( Math.floor(foo*1000)/1000 ); // 12.135
8 console.log( Math.ceil(foo*1000)/1000 ); // 12.136
9 console.log( Math.round(foo*1000)/1000 ); // 12.135
```

Jika anda butuh banyak pembulatan desimal seperti ini, bisa membuat sebuah function khusus seperti contoh di halaman ini: [PHP-Like rounding Method<sup>3</sup>](#). Atau khusus untuk pembulatan `Math.round()`, bisa juga menggunakan method `Number.prototype.toFixed()`.

## 13.4 Method Math.random()

Method `Math.random()` digunakan untuk membuat angka acak (random). Angka acak yang dihasilkan berada dalam rentang 0 - 1, tapi tidak termasuk angka 1 itu sendiri. Maksudnya angka acak yang dihasilkan adalah dari 0, 0.1, 0.2, 0.3 dst hingga 0.9.

Berikut contoh penggunaannya:

---

<sup>3</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Math/round#PHP-Like\\_rounding\\_Method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/round#PHP-Like_rounding_Method)

```

1 var foo = Math.random();
2 console.log( foo );           // 0.7656339439523763

```

Jika anda menjalankan kode diatas, hasilnya akan berbeda tergantung nilai acak yang digenerate oleh JavaScript.

Sekarang, bagaimana caranya membuat angka acak yang bulat? Bukan dalam bentuk pecahan seperti itu? Misalnya angka acak bulat 0, 1, 2 sampai 10? Kita tinggal mengalikan angka acak hasil pemanggilan `Math.random()` dengan 10, lalu dibulatkan menggunakan method `Math.round()`:

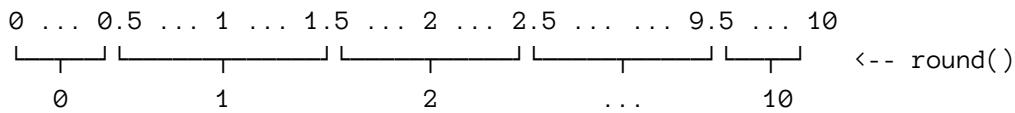
```

1 var foo = Math.random();
2 console.log( foo );           // 0.5092621312447921
3
4 foo = Math.round(foo * 10);
5 console.log( foo );           // 5

```

Dengan cara ini saya bisa membuat angka acak dari 0 - 10. Bagaimana untuk angka 0 - 100? tinggal kalikan saja hasil random dengan 100, lalu dibulatkan.

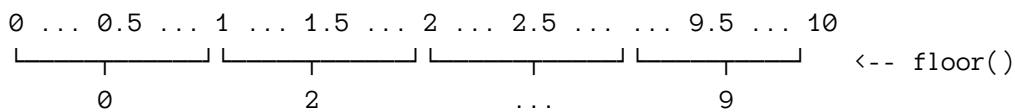
Sebenarnya trik membulatkan angka acak menggunakan method `Math.round()` memiliki 1 kelemahan, dimana sebaran angka yang dihasilkan kurang merata. Untuk bisa memahami apa yang saya maksud, perhatikan ilustrasi berikut:



Ilustrasi diatas adalah peluang kemunculan angka 0 sampai 10 jika menggunakan method `Math.round()`.

Permasalahannya ada di peluang kemunculan angka 0 dan 10. Kemungkinan kedua angka ini keluar hanya setengah kali angka-angka lain. Untuk angka 0, peluangnya ada antara 0.0 - 0.4, sedangkan untuk angka 1, peluangnya ada dari 0.5 - 1.4. Untuk angka 10, peluangnya lebih kecil lagi, karena hanya dari 9,5 - 9,9 (angka 10 itu sendiri tidak termasuk).

Solusinya, kita bisa mengganti method `Math.round()` menjadi method `Math.floor()`, dimana semua angka akan dibulatkan ke bawah. Sebaran peluangnya menjadi merata:



Sekarang, peluang muncul angka 0 sama besarnya dengan angka-angka lain. Tapi masalahnya, angka 10 tidak akan pernah muncul, karena nilai 9,9 selalu dibulatkan ke 9, bukan ke 10. Jalan keluarnya adalah dengan menambahkan nilai maksimum + 1 menjadi 11.

Berdasarkan analisis ini, metode pembuatan angka acak kita menjadi sebagai berikut:

```
1 var foo = Math.random();
2 console.log( foo );      // 0.39055944950626187
3
4 foo = foo * (10 + 1);
5 console.log( foo );      // 4.29615394456888
6
7 foo = Math.floor(foo);
8 console.log( foo );      // 4
```

Pertama, variabe `foo` diisi angka hasil `Math.random()`. Kedua, `foo` dikali dengan `(10 + 1)`. Ketiga, hasilnya dibulatkan ke bawah menggunakan method `Math.floor(foo)`. Ketiga langkah ini bisa digabung menjadi 1 kali pemanggilan:

```
1 var foo = Math.floor(Math.random() * (10 + 1));
2 console.log( foo );      // 8
```

Sekarang kita sudah mendapatkan solusi untuk membuat angka acak bulat.

Sebagai latihan, bisakah anda menghasilkan angka acak dari 50 - 99? Silahkan buka teks editor dan modifikasi “rumus” yang kita dapatkan sebelumnya.

Baik, berikut kode yang saya gunakan:

```
1 var foo = Math.floor(Math.random() * (50)) + 50;
2 console.log( foo );      // 68
```

Idenya, saya akan mengenerate angka acak dari 0 - 49 terlebih dahulu, menggunakan perintah: `Math.floor(Math.random() * (50))`. Setelah itu hasilnya ditambah 50. Dengan demikian, kita bisa mendapat angka acak dari 50 - 99.

Latihan kedua (dan ini mungkin akan sering anda buat nantinya), adalah menggenerate angka acak sebanyak 10 buah, yang nilainya disimpan ke dalam sebuah array. Angka acak ini saya batasi dari 1 - 6.

Berikut kode yang saya gunakan:

```
1 var foo = [];
2
3 for (var i = 0; i < 10; i++){
4   foo[i] = 1 + Math.floor( Math.random() * 6 );
5 }
6
7 console.log( foo );
8 // array acak berisi 10 element angka antara 1 - 6
9 // contoh hasil: Array [ 2, 4, 2, 3, 2, 5, 1, 2, 1, 6 ]
```



Gambar: Hasil array foo yang menampung 10 angka acak

Seperti yang bisa anda duga, untuk membuat 10 angka acak kita akan menggunakan perulangan (looping). Pada awal kode program, saya menyiapkan variabel `foo` yang diisi dengan array kosong. Variabel inilah yang nantinya akan diisi di dalam perulangan.

Saya menggunakan perulangan `for` yang dijalankan sebanyak 10 kali, mulai dari variabel counter `i` bernilai 0, hingga 9. Dalam setiap perulangan, jalankan perintah: `foo[i] = 1 + Math.floor(Math.random() * 6)`. Perintah ini akan menginput angka acak ke dalam array `foo`. Terakhir, saya menampilkan isi variabel `foo` menggunakan perintah `console.log()`.

Dalam prakteknya, angka random sering digunakan dalam membuat aplikasi games. Misalnya array yang berisi angka 1-6 ini bisa digunakan untuk membuat games yang melibatkan dadu, seperti monopoli, ular tangga, dll.

## 13.5 Method Math.max() dan Math.min()

Method `Math.max()` dan `Math.min()` digunakan untuk mencari nilai paling besar dan nilai paling kecil dari angka-angka yang diinput ke dalam argumennya.

Berikut contoh penggunaan method `Math.max()` dan `Math.min()`:

```

1 var foo = Math.max(45,90,12,55,19,75);
2 console.log( foo );      // 90
3
4 var bar = Math.min(45,90,12,55,19,75);
5 console.log( bar );      // 12

```

JavaScript tidak membatasi berapa jumlah argumen yang bisa diinput.

Dalam kebanyakan kasus, kita ingin mencari nilai maksimum dan minimum dari data yang tersimpan di dalam array. Sayangnya, kedua method tidak bisa diinput langsung dengan tipe data array. Kita harus menggunakan perulangan untuk mencari nilai maksimum dan minimum dari sebuah array.

Namun **spread operator** dari ECMAScript 6 muncul sebagai solusi. Kita bisa menggunakan spread operator untuk memasukkan seluruh element array ke dalam method `Math.max()` dan `Math.min()`. Berikut contoh penggunaannya:

```
1 var foo = [45,90,12,55,19,75];
2
3 console.log( Math.max(...foo) );           // 90
4 console.log( Math.min(...foo) );           // 12
```

Kita tinggal mengisi ...foo sebagai nilai argumen dari kedua method ini.

## 13.6 Method Math.abs()

Method `Math.abs()` cukup simple, method ini digunakan untuk menghasilkan nilai absolut dari argumentnya. Jika angka itu positif, tidak dilakukan perubahan apapun. Namun jika angka itu negatif, hasilnya adalah angka positif.

Berikut contoh penggunaannya:

```
1 var foo = 5;
2 console.log( Math.abs(foo) );           // 5
3
4 var bar = -5;
5 console.log( Math.abs(bar) );           // 5
```

## 13.7 Method Math.pow()

Method `Math.pow()` digunakan untuk pemangkatan suatu angka. Method ini membutuhkan 2 argumen. Argumen pertama adalah angka yang ingin dicari pangkatnya (base number) dan argumen kedua adalah nilai pangkatnya.

Berikut contoh penggunaan dari method `Math.pow()`:

```
1 console.log( Math.pow(5,2) );           // 25
2 console.log( Math.pow(2,8) );           // 256
3
4 console.log( Math.pow(49,1/2) );         // 7
5 console.log( Math.pow(6561,1/4) );       // 9
6
7 console.log( Math.pow(-49,1/2) );        // NaN
```

Perintah `Math.pow(5,2)` artinya sama dengan  $5^2$ , sedangkan `Math.pow(2,8)` artinya  $2^8$ .

Dengan sedikit teori matematika, kita juga bisa menggunakan method `Math.pow()` untuk mencari akar dari sebuah angka. Caranya, dengan memberikan nilai pecahan pada argumen kedua. Perintah `Math.pow(49,1/2)` artinya akar kuadrat dari 49. Sedangkan `Math.pow(6561,1/4)` artinya akar pangkat 4 dari 6561.

Jika angka yang dihasilkan tidak bisa dihitung, method `Math.pow()` akan mengembalikan nilai `NaN`. Dalam contoh baris terakhir, saya ingin mencari akar dari angka negatif, yang tentu saja tidak bisa dihitung.

## 13.8 Method Math.sqrt()

Method `Math.sqrt()` digunakan untuk akar kuadrat (*square*) dari suatu angka. Penggunaannya sangat mudah:

```
1 console.log( Math.sqrt(25) ); // 5
2 console.log( Math.sqrt(81) ); // 9
3
4 console.log( Math.sqrt(1) ); // 1
5 console.log( Math.sqrt(-1) ); // NaN
```

Jika akar kuadrat tersebut tidak valid, seperti akar kuadrat dari angka negatif, method ini akan menghasilkan nilai `NaN`.

## 13.9 Method Math.log(), Math.log10() dan Math.log2()

`Math` object memiliki beberapa method untuk menghitung logaritma. Tiga diantaranya adalah: `Math.log()`, `Math.log10()` dan `Math.log2()`.

Method `Math.log()` digunakan untuk mencari nilai logaritma natural ( $e$ ), bukan logaritma yang kita gunakan sehari-hari (basis 10). Untuk logaritma dengan basis desimal (basis 10), kita harus menggunakan `Math.log10()`. Sedangkan untuk `Math.log2()` merupakan method untuk menghitung nilai logaritma dari bilangan biner.

Berikut contoh penggunaan dari method `Math.log()`, `Math.log10()` dan `Math.log2()`:

```
1 console.log( Math.log(1) ); // 0
2 console.log( Math.log(10) ); // 2.302585092994046
3
4 console.log( Math.log10(1000) ); // 3
5 console.log( Math.log10(0.001) ); // -3
6
7 console.log( Math.log2(256) ); // 8
8 console.log( Math.log2(4294967296) ); // 32
```

## 13.10 Method Math.sin(), Math.cos() dan Math.tan()

Jika anda masih ingat masa-masa romantis belajar trigonometri, melihat sekilas method ini pasti sudah bisa menebak apa fungsinya. Ketiganya digunakan untuk mencari nilai *sinus*, *cosinus* dan *tangen*.

Akan tetapi argumen yang diinput adalah dalam bentuk radian, bukan derajat. Dimana  $360^\circ$  sama dengan  $2\pi$  radian. Atau dengan kata lain  $1$  derajat =  $(2\pi \text{ radian})/360$ .

Sebelum terlalu pusing dengan rumus-rumus ini, langsung saja kita lihat contohnya:

```
1 var derajat = 60;
2 var rad = derajat * ((2 * Math.PI) / 360);
3
4 console.log( Math.sin(rad) ); // 0.8660254037844386
5 console.log( Math.cos(rad) ); // 0.5000000000000001
6 console.log( Math.tan(rad) ); // 1.7320508075688767
```

Saya membuat variabel `foo` yang diisi dengan angka **60**, dimana maksudnya adalah 60 derajat. Sebelum diinput ke dalam method, saya menjalankan perintah `derajat * ((2 * Math.PI) / 360)` untuk mengkonversinya ke dalam satuan radian. Kemudian baru diinput ke dalam setiap method.

Selain method `Math.sin()`, `Math.cos()` dan `Math.tan()`. Object math masih memiliki method lain seputar trigonometri, seperti `Math.asin()`, `Math.acos()`, dan `Math.atan()`. Penggunaannya kurang lebih sama seperti yang kita bahas disini.

---

Sepanjang bab ini kita telah membahas berbagai property dan method dari **Math** object JavaScript. Seperti yang terlihat, hampir semuanya berhubungan dengan angka dan rumus matematika.

Sebagai tambahan, jika anda butuh “rumus” matematika yang lebih kompleks, misalnya untuk memproses matrix, bisa menggunakan library seperti [mathjs<sup>4</sup>](http://mathjs.org).

Berikutnya, kita akan membahas **String object** JavaScript.

---

<sup>4</sup><http://mathjs.org>

# 14. String Object

JavaScript menyediakan berbagai property dan method untuk pemrosesan tipe data **String**, yang semuanya disimpan ke dalam **String Object**. Dalam bab kali ini kita akan membahasnya dengan lebih detail.

Seperti yang sudah kita pelajari, **tipe data primitif string** maupun **String Object** sama-sama bisa digunakan untuk mengakses property dan method string. Berikut contohnya:

```
1 // membuat object string
2 var foo = new String("Belajar JavaScript");
3 console.log ( foo );      // String { "Belajar JavaScript", 19 more... }
4
5 console.log ( typeof foo );           // object
6 console.log ( foo.length );          // 18
7 console.log ( foo.toUpperCase() );    // BELAJAR JAVASCRIPT
8
9 // membuat string sebagai tipe data primitif (menggunakan string literal)
10 var bar = "Belajar JavaScript";
11 console.log ( bar );    // Belajar JavaScript
12
13 console.log ( typeof bar );         // string
14 console.log ( bar.length );        // 18
15 console.log ( bar.toUpperCase() ); // BELAJAR JAVASCRIPT
```

Disini saya membuat 2 variabel string. Variabel `foo` berisi **String Object** "Belajar JavaScript". Sedangkan variabel `bar` diisi **tipe data primitif string** "Belajar JavaScript". Baik variabel `foo` maupun `bar` bisa digunakan untuk memanggil property `length` dan method `toUpperCase()`.

Meskipun kedua cara pembuatan string ini bisa dipakai, cara yang disarankan adalah menggunakan **tipe data primitif string** atau disebut juga sebagai *string literal*, yakni string yang dibuat menggunakan tanda kutip.



Jika anda butuh panduan teknis (referensi) tentang **String Object** yang lebih detail, silahkan buka [Standard built-in objects: String<sup>1</sup>](#) dari Mozilla Developer Network.

## 14.1 String Object Method

Terdapat 3 method yang melekat langsung ke String Object:

---

<sup>1</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)

- `String.fromCharCode()`
- `String.fromCodePoint()`
- `String.raw()`

Method `String.raw()` tidak umum digunakan dan hanya di dukung oleh Mozilla Firefox, oleh karena itu tidak akan saya bahas.

## 14.2 Method `String.fromCharCode()` dan `String.fromCodePoint()`

Kedua method ini digunakan untuk membuat string berdasarkan kode Unicode. Perbedaannya, method `String.fromCodePoint()` mendukung penulisan karakter Unicode yang lebih baru.

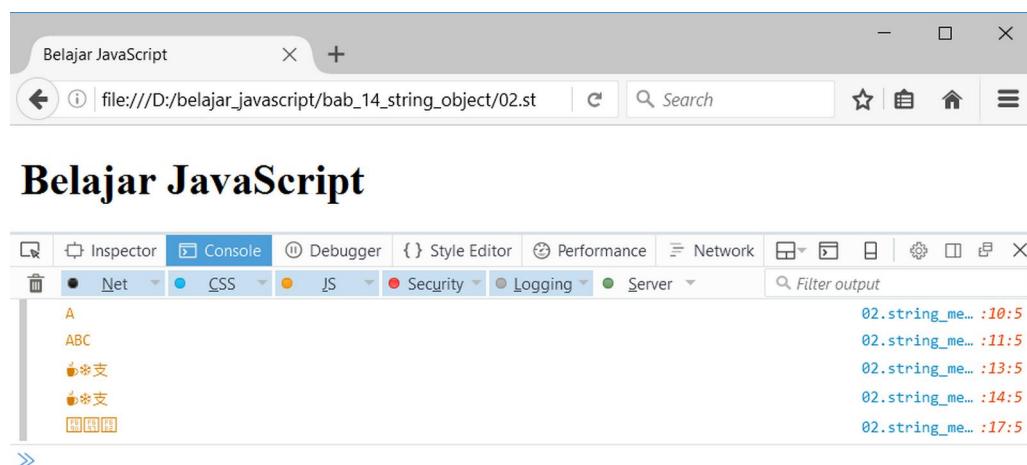
Seperti yang pernah kita singgung sewaktu membahas tipe data string, Unicode merupakan karakter set yang dipakai JavaScript, dimana jumlah karakternya masih terus ditambah. Masalahnya, karakter baru Unicode butuh cara pengaksesan yang sedikit berbeda, dan tidak bisa ditampilkan oleh method `String.fromCharCode()`.

Berikut contoh penggunaan method `String.fromCharCode()`:

```

1 console.log (String.fromCharCode(65) );           // A
2 console.log (String.fromCharCode(65, 66, 67) );     // ABC
3
4 console.log (String.fromCharCode(0x2615, 0x2744, 0x2F40) );
5 console.log (String.fromCharCode(9749, 10052, 12096) );
6
7 // karakter baru Unicode, tidak bisa ditampilkan
8 console.log (String.fromCharCode(128656, 128663, 128690) );

```



Gambar: Tampilan dari method `String.fromCharCode()`

Dalam daftar karakter Unicode, huruf A, B, dan C berada di nomor urut 65, 66, dan 67 (desimal). Oleh karena itulah perintah `String.fromCharCode(65, 66, 67)` akan menghasilkan string A, B, dan C.

Selain nomor urut desimal, kita juga bisa menggunakan nomor urut angka heksadesimal Unicode, seperti `String.fromCharCode(0x2615, 0x2744, 0x2F40)` (awalan `0x` digunakan untuk membuat nilai heksadesimal).

Kode ini menghasilkan tampilan yang sama dengan `String.fromCharCode(9749, 10052, 12096)` yang menggunakan nomor urut desimal.

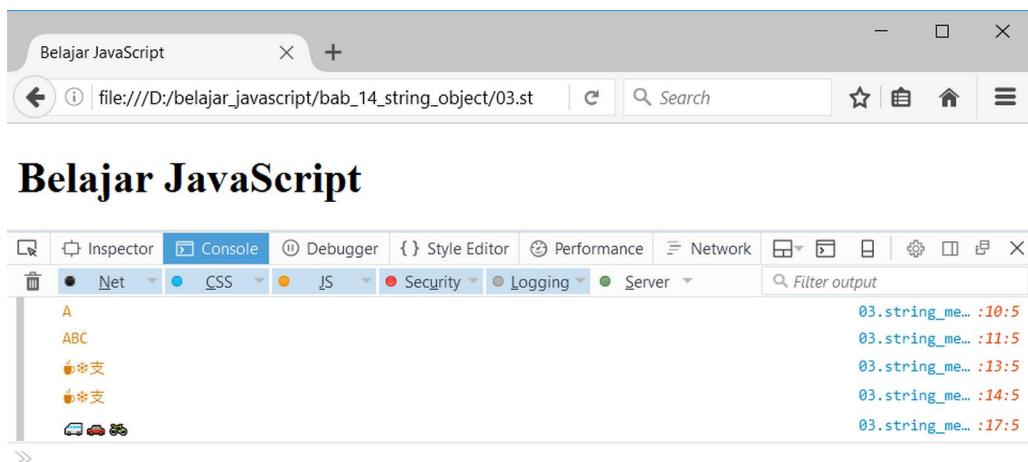
Di baris terakhir, saya menggunakan nomor urut karakter baru Unicode. Terlihat bahwa method `String.fromCharCode()` gagal menampilkan karakter tersebut (karakter ‘emoji’).

Mari kita coba dengan method `String.fromCodePoint()`:

```

1 console.log (String.fromCodePoint(65) );           // A
2 console.log (String.fromCodePoint(65, 66, 67) );    // ABC
3
4 console.log (String.fromCodePoint(0x2615, 0x2744, 0x2F40) );
5 console.log (String.fromCodePoint(9749, 10052, 12096) );
6
7 // karakter baru Unicode, sudah bisa ditampilkan
8 console.log (String.fromCodePoint(128656, 128663, 128690) );

```



Gambar: Tampilan dari method `String.fromCodePoint()`

Cara penggunaan method `String.fromCodePoint()` sama persis seperti `String.fromCharCode()`. Terlihat di baris terakhir karakter emoji dari Unicode sudah tampil.

Yang juga perlu diketahui, method `String.fromCodePoint()` merupakan bagian dari ECMAScript 6, sehingga mungkin ada beberapa web browser yang belum mendukungnya.



Daftar lengkap karakter Unicode bisa anda akses dari web [unicode-table.com](http://unicode-table.com)<sup>2</sup>.

<sup>2</sup><http://unicode-table.com>

## 14.3 String Instance Property

String instance property adalah property tipe data string yang “melekat” ke hasil instance dari **String object**. Di dalam JavaScript, hanya terdapat 1 string instance property:

- `String.prototype.length`

## 14.4 Property String.prototype.length

Property `length` digunakan untuk mengambil info panjang karakter dari sebuah string. Berikut contoh penggunaannya:

```
1 var foo = "Belajar JavaScript";
2 console.log ( foo.length );           // 18
3
4 var bar = "Learning Something Fun".length;
5 console.log ( bar );                // 22
6
7 console.log ( "DuniaIalkom".length ); // 10
```

Property `length` cukup sering dipakai, terutama dalam proses validasi form. Misalnya kita ingin memastikan isian form sudah sesuai syarat minimal 6 karakter, atau kode pegawai yang harus diisi 10 digit karakter (tidak boleh kurang atau lebih). Prakteknya akan kita bahas di bab tersendiri.

## 14.5 String Instance Method

String instance method adalah method yang “melekat” ke instance dari String object. Method ini cukup banyak, berikut daftar yang saya ambil dari **Mozilla Developer Network**:

- `String.prototype.charAt()`
- `String.prototype.charCodeAt()`
- `String.prototype.codePointAt()`
- `String.prototype.concat()`
- `String.prototype.includes()`
- `String.prototype.endsWith()`
- `String.prototype.indexOf()`
- `String.prototype.lastIndexOf()`
- `String.prototype.localeCompare()`
- `String.prototype.match()`
- `String.prototype.normalize()`

- `String.prototype.padEnd()`
- `String.prototype.padStart()`
- `String.prototype.quote()`
- `String.prototype.repeat()`
- `String.prototype.replace()`
- `String.prototype.search()`
- `String.prototype.slice()`
- `String.prototype.split()`
- `String.prototype.startsWith()`
- `String.prototype.substr()`
- `String.prototype.substring()`
- `String.prototype.toLocaleLowerCase()`
- `String.prototype.toLocaleUpperCase()`
- `String.prototype.toLowerCase()`
- `String.prototype.toSource()`
- `String.prototype.toString()`
- `String.prototype.toUpperCase()`
- `String.prototype.trim()`
- `String.prototype.trimLeft()`
- `String.prototype.trimRight()`
- `String.prototype.valueOf()`
- `String.prototype[@@iterator]()`

Kita akan membahas sebagian besar diantaranya.

## 14.6 Method `String.prototype.toLowerCase()` dan `String.prototype.toUpperCase()`

Kedua method ini digunakan untuk mengubah **case** (bentuk huruf) dari sebuah string. Method `toLowerCase()` digunakan untuk mengubah string menjadi huruf kecil, sedangkan method `toUpperCase()` untuk mengubah string menjadi huruf besar.

Berikut contoh penggunaannya:

```
1 var foo = "Belajar JavaScript";
2
3 console.log ( foo.toLowerCase() ); // belajar javascript
4 console.log ( foo.toUpperCase() ); // BELAJAR JAVASCRIPT
5 console.log ( foo ); // Belajar JavaScript
```

JavaScript juga menyediakan method yang hampir mirip: `String.prototype.toLocaleLowerCase()` dan `String.prototype.toLocaleUpperCase()`.

Tambahan kata “Locale” menegaskan bahwa method ini mengubah case karakter sesuai settingan bahasa lokal. Contoh bahasa yang membedakan huruf besar dan kecil adalah bahasa Turki. Bagi kebanyakan bahasa (termasuk bahasa Indonesia), efeknya sama seperti method `toLowerCase()` dan `toUpperCase()`:

```
1 var foo = "Belajar JavaScript";
2
3 console.log ( foo.toLocaleLowerCase() ); // belajar javascript
4 console.log ( foo.toLocaleUpperCase() ); // BELAJAR JAVASCRIPT
5 console.log ( foo ); // Belajar JavaScript
```

## 14.7 Method String.prototype.charAt()

Method `charAt()` berguna untuk menampilkan karakter yang berada di posisi tertentu dari sebuah string. Berikut contoh penggunaannya:

```
1 var foo = "Belajar JavaScript";
2
3 console.log ( foo.charAt(1) ); // e
4 console.log ( foo.charAt(8) ); // J
5 console.log ( 'DuniaIlkom'.charAt(4) ); // a
```

Perhitungan posisi untuk method `charAt()` dimulai dari 0. Dengan kata lain, karakter pertama berada di posisi 0, karakter kedua di posisi 1, dst. Kurang lebih mirip seperti penomoran index array.

Sebenarnya, object string JavaScript juga bisa diakses langsung seperti layaknya sebuah array:

```
1 var foo = "Belajar JavaScript";
2
3 console.log ( foo[1] ); // e
4 console.log ( foo[8] ); // J
5 console.log ( 'DuniaIlkom'[4] ); // a
```

Terlihat, hasilnya sama persis seperti menggunakan method `charAt()`. Namun berbeda dengan array, kita tidak bisa mengubah karakter ini:

```
1 var foo = "Belajar JavaScript";
2 foo[0] = 'S'; // tidak akan mengubah string foo
3 console.log ( foo ); // Belajar JavaScript
```

## 14.8 Method String.prototype.charCodeAt() dan String.prototype.codePointAt()

Kedua method ini berfungsi untuk menampilkan kode Unicode dari sebuah karakter. Bedanya, method ‘codePointAt()’ mendukung penomoran karakter Unicode yang baru.

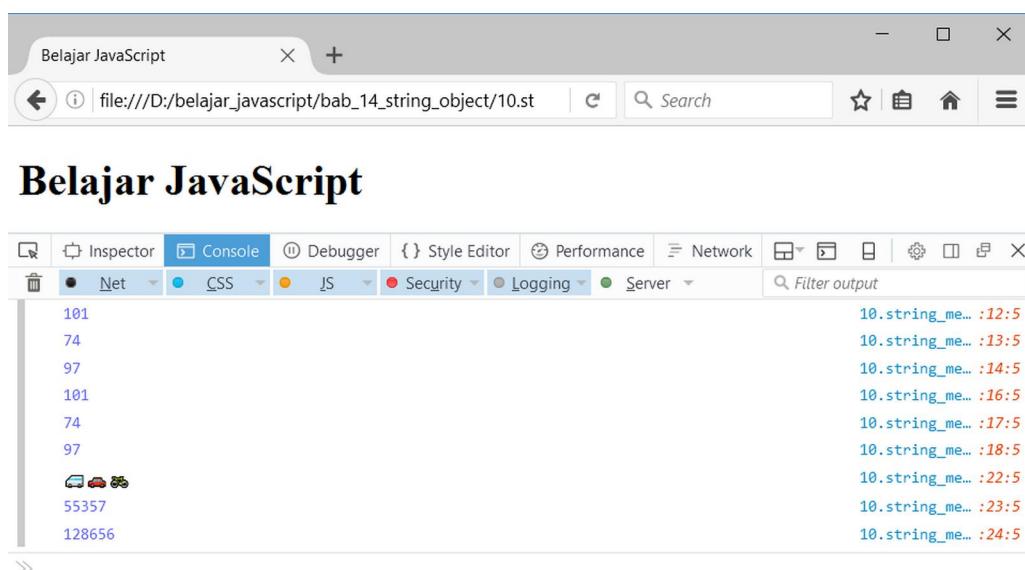
Method `charCodeAt()` dan `codePointAt()` adalah kebalikan dari method `String.fromCharCode()` dan `String.fromCodePoint()`.

Berikut contoh penggunaan kedua method ini:

```

1 var foo = "Belajar JavaScript";
2
3 console.log ( foo.charCodeAt(1) );           // 101
4 console.log ( foo.charCodeAt(8) );           // 74
5 console.log ( 'DuniaIlkom'.charCodeAt(4) ); // 97
6
7 console.log ( foo.codePointAt(1) );           // 101
8 console.log ( foo.codePointAt(8) );           // 74
9 console.log ( 'DuniaIlkom'.codePointAt(4) ); // 97
10
11 var bar = String.fromCodePoint(128656, 128663, 128690);
12
13 console.log ( bar );                      // emoji
14 console.log ( bar.charCodeAt(0) );          // 55357
15 console.log ( bar.codePointAt(0) );          // 128656

```



Gambar: Hasil pemanggilan method `charCodeAt()` dan `codePointAt()`

Perintah `foo.charCodeAt(1)` artinya, tampilkan no urut unicode untuk karakter yang terdapat di index 1 dari variabel `foo`. Variabel `foo` sendiri berisi string "Belajar JavaScript", sehingga

pada index 1 terdapat karakter ‘e’. Di dalam tabel Unicode, karakter ‘e’ berada di nomor urut 101.



Gambar: No urut karakter ‘e’, berada di 65 heksadesimal, atau 101 desimal

Begitu juga dengan `foo.charCodeAt(8)`, dimana karakter ‘J’ memiliki kode unicode 74. Dan `'DuniaIlkom'.charCodeAt(4)` dimana karakter ‘a’ memiliki kode unicode 97. Perhatikan juga bahwa method `codePointAt()` menghasilkan tampilan yang sama dengan method `charCodeAt()`, kecuali untuk variabel `bar`.

Variabel `bar` saya isi dengan string karakter emoji. String ini dibuat dari method `String.fromCharCode()`. Di karenakan karakter emoji ini merupakan karakter baru Unicode, method `charCodeAt()` tidak berjalan sempurna. Hasil yang tepat baru didapat menggunakan method `codePointAt()`.

Sama seperti method `String.fromCharCode()`, method `codePointAt()` juga bagian dari ECMAScript 6.

## 14.9 Method String.prototype.substr()

Method `substr()` digunakan untuk mengambil sebagian karakter string. Method ini bisa diisi dengan 2 argumen:

- **Argumen pertama** adalah nomor index awal pengambilan. Jika diisi angka positif, perhitungan index dimulai dari awal string (dari sebelah kiri). Jika diisi angka negatif, perhitungan karakter dimulai dari akhir string (dari sebelah kanan).
- **Argumen kedua** (opsional) menentukan berapa banyak karakter yang ingin diambil. Jumlah karakter dihitung dari index yang terdapat di argumen pertama. Jika argumen kedua ini tidak ditulis, artinya ambil seluruh karakter hingga akhir string.

Method `substr()` tidak mengubah string asal, tapi mengembalikan nilai string baru. Berikut contoh penggunaannya:

```
1 var foo = "Belajar JavaScript";
2
3 console.log ( foo.substr(3) );           // ajar JavaScript
4 console.log ( foo.substr(-3) );          // pt
5 console.log ( foo.substr(3, 4) );         // ajar
6 console.log ( foo.substr(-10, 4) );        // Java
7
8 console.log ( foo );                  // Belajar JavaScript
```

Perintah `foo.substr(3)` berarti ambil seluruh string `foo`, mulai dari index ke-3 (karakter ke-4) hingga akhir string. Perintah `foo.substr(-3)` artinya ambil 3 karakter terakhir dari string `foo`.

Perintah `foo.substr(3, 4)` berarti ambil 4 karakter dari string `foo`, mulai dari index ke-3. Sedangkan perintah `foo.substr(-10, 4)` artinya, ambil 4 karakter string `foo`, mulai dari karakter ke 10 dari akhir string.

Di akhir kode program, saya menampilkan isi variabel `foo` yang tampak masih utuh, tidak terpengaruh dari pemanggilan method `foo.substr()`.

## 14.10 Method String.prototype.substring()

Method `substring()` mirip seperti `substr()`, yakni sama-sama berfungsi untuk mengambil sebagian karakter string. Bedanya, dalam method `substring()` argumen pertama adalah index awal pengambilan dan argumen kedua sebagai index akhir pengambilan.

Method `substring()` bisa diisi dengan 2 argumen:

- **Argumen pertama** adalah nomor index awal pengambilan. Jika diisi angka positif, perhitungan index dimulai dari awal string. Jika diisi angka negatif, akan dianggap 0 dimana perhitungan juga tetap dihitung dari awal string (tidak dari akhir seperti method `substr()`).
- **Argumen kedua** (opsional) adalah nomor index akhir pengambilan. Jika tidak ditulis, artinya ambil seluruh string hingga akhir. Jika diisi angka positif, perhitungan index dimulai dari awal string. Jika diisi angka negatif, akan dianggap 0, artinya akhir string ada di awal.

Berikut contoh penggunaannya:

```
1 var foo = "Belajar JavaScript";
2
3 console.log ( foo.substring(3) );           // ajar JavaScript
4 console.log ( foo.substring(3, 4) );         // a
5 console.log ( foo.substring(3, 12) );        // ajar Java
6 console.log ( foo.substring(3, 0) );          // Bel
7 console.log ( foo.substring(-3) );           // Belajar JavaScript
8 console.log ( foo.substring(-3, 12) );        // Belajar Java
9 console.log ( foo.substring(-3, -1) );         // (kosong)
10
11 console.log ( foo );                      // Belajar JavaScript
```

Ketika method `substring()` diisi dengan argumen angka positif, hasilnya cukup mudah untuk dipahami. Mari kita bahas argumen dengan angka 0 dan negatif.

Untuk perintah `foo.substring(3, 0)`, artinya ambil string `foo` mulai dari index ke 3 hingga ke 0. Hasil perintah ini sama artinya dengan ambil 3 karakter pertama dari string `foo`.

Dalam perintah `foo.substring(-3)`, saya mengisinya index awal dengan angka negatif. Untuk hal seperti ini, JavaScript akan mengkonversi nilai negatif menjadi nol. Artinya perintah `foo.substring(-3)` sama dengan `foo.substring(0)`. Dimana kita ingin mengambil string `foo` mulai dari index pertama hingga akhir string.

Perintah `foo.substring(-3, 12)` artinya ambil string `foo` mulai dari index ke -3 (angka negatif dikonversi menjadi 0) hingga index ke 12. Sedangkan perintah `foo.substring(-3, -1)` tidak menampilkan apa-apa karena sama artinya dengan `foo.substring(0, 0)`.

Di akhir kode program saya menampilkan kembali isi variabel `foo` untuk memperlihatkan bahwa method `substring()` tidak mengubah string asal.

## 14.11 Method String.prototype.slice()

Method `slice()` juga digunakan untuk mengambil sebagian karakter string. Penggunaannya mirip seperti method `substr()`, tapi argumen kedua berupa index akhir pengambilan, bukan banyak karakter.

Method `slice()` bisa diisi dengan 2 argumen:

- **Argumen pertama** adalah nomor index awal pengambilan. Jika diisi angka positif, perhitungan index dimulai dari awal string (sebelah kiri). Jika diisi angka negatif, perhitungan karakter dimulai dari akhir string (sebelah kanan).
- **Argumen kedua** (opsional) adalah nomor index akhir pengambilan. Jika diisi angka positif, perhitungan index dimulai dari awal string (sebelah kiri). Jika diisi angka negatif, perhitungan karakter dimulai dari akhir string (sebelah kanan). Jika tidak ditulis, ambil seluruh string hingga karakter terakhir.

Berikut contoh penggunaan dari method `slice()`:

```
1 var foo = "Belajar JavaScript";
2
3 console.log ( foo.slice(3) );           // ajar JavaScript
4 console.log ( foo.slice(3, 4) );         // a
5 console.log ( foo.slice(3, 12) );        // ajar Java
6
7 console.log ( foo.slice(-3) );          // ipt
8 console.log ( foo.slice(-3, 12) );       // (kosong)
9 console.log ( foo.slice(-3, -1) );        // ip
10
11 console.log ( foo );                  // Belajar JavaScript
```

Method `foo.slice(3)` artinya ambil string `foo` mulai dari index ke-3 (karakter ke-4) hingga akhir string. Method `foo.slice(3, 4)` artinya ambil string `foo` mulai dari index ke-3 hingga index ke-4. Method `foo.slice(3, 12)` artinya ambil string `foo` mulai dari index ke-3 hingga index ke-12.

Jika argumen diinput angka negatif, perhitungan karakter dimulai dari akhir string. Method `foo.slice(-3)` artinya ambil 3 karakter terakhir dari string `foo`. Method `foo.slice(-3, 12)` artinya ambil 3 karakter terakhir dari string `foo`, hingga index ke 12. Hasilnya tidak tampil apa-apa, karena index ke 12 ada sebelum 3 karakter terakhir. Untuk method `foo.slice(-3, -1)` artinya ambil 3 karakter mulai dari akhir string `foo`, kecuali 1 karakter terakhir.

Ketiga method yang kita bahas secara berurutan: `substr()`, `substring()`, dan `slice()` sangat mirip satu sama lain. Perbedaannya hanya di argumen kedua masing-masing method.

## 14.12 Method String.prototype.split()

Method `split()` digunakan untuk memecah sebuah string menjadi array. Method ini memiliki 2 argumen:

- **Argumen pertama** (opsional) bisa diinput dengan karakter pembatas yang digunakan untuk memecah string. Jika tidak ditulis, string di konversi menjadi 1 elemen array. Argumen pertama ini juga bisa diisi dengan *regular expression*.
- **Argumen kedua** (opsional) adalah jumlah elemen array yang ingin diambil. Jika tidak ditulis seluruh string akan dikonversi menjadi array.

Mari kita lihat contohnya:

```
1 var foo = "Belajar";
2
3 console.log ( foo.split() );
4 // Array [ "Belajar" ]
5
6 console.log ( foo.split('') );
7 // Array [ "B", "e", "l", "a", "j", "a", "r" ]
```

Jika method `split()` dipanggil tanpa argumen, string akan dikonversi menjadi 1 element array. Dalam perintah `foo.split('')` saya mengisi argumen pertama dengan string kosong. Hasilnya, setiap karakter dari string `foo` akan dikonversi menjadi array (1 karakter untuk setiap element).

Mari kita lihat contoh selanjutnya:

```
1 var foo = "satu dua tiga empat lima";
2 console.log ( foo.split(' ') );
3 // Array [ "satu", "dua", "tiga", "empat", "lima" ]
4
5 var bar = "satu,dua,tiga,empat,lima";
6 console.log ( bar.split(',') );
7 // Array [ "satu", "dua", "tiga", "empat", "lima" ]
```

Disini saya membuat variabel `foo` berisi 5 kata yang masing-masingnya dipisah sebuah spasi. Dengan menginput spasi sebagai argumen pertama method `split()`, setiap kata akan menjadi element array.

Untuk variabel `bar`, setiap kata dipisah dengan tanda koma. Dengan menginput karakter ini sebagai argumen pertama, kita juga mendapatkan hasil yang sama.

Mari kita tambahkan argumen kedua:

```
1 var foo = "satu dua tiga empat lima";
2 console.log ( foo.split(' ',3) );
3 // Array [ "satu", "dua", "tiga" ]
4
5 var bar = "satu,dua,tiga,empat,lima";
6 console.log ( bar.split(',', 7) );
7 // Array [ "satu", "dua", "tiga", "empat", "lima" ]
```

Argumen kedua digunakan untuk membatasi jumlah element array. Perintah `foo.split(' ',3)` artinya, pecah variabel `foo` menjadi array yang dipisah dengan sebuah spasi, kemudian ambil 3 element pertama. Sisa element lain akan diabaikan.

Untuk perintah `bar.split(',', 7)`, jumlah element array tidak sampai 7 buah, sehingga seluruh string `bar` akan dikonversi menjadi array.

Fitur lain dari argumen pertama method `split()` adalah bisa menerima *regular expression*:

```
1 var foo = "satu,dua;tiga-empat+lima";
2 console.log ( foo.split(/\W/) );
3 // Array [ "satu", "dua", "tiga", "empat", "lima" ]
```

Materi tentang *regular expression* cukup luas dan akan saya bahas dalam 1 bab khusus. Di dalam JavaScript, regular expression disimpan ke dalam **RegExp Object**.

Perintah `foo.split(/\W/)` artinya, konversi string `foo` menjadi array dengan batasan setiap element berupa regexp "`\W`". Di dalam regular expression, pola "`\W`" akan cocok dengan seluruh karakter selain huruf dan angka. Sehingga karakter `,` `-` dan `+` akan menjadi karakter pembatas dari method `split()`.

## 14.13 Method String.prototype.trim()

Method `trim()` digunakan untuk menghapus karakter **whitespace** di awal dan akhir string. **Whitespace** adalah karakter “spasi”, termasuk tab, enter, dan *carriage return* (karakter penanda untuk pindah baris).

Berikut contoh penggunaannya:

```
1 var foo = "    JavaScript      ";
2 console.log ( foo.length );          // 18
3 console.log ( foo );                // "    JavaScript      "
4
5 foo = foo.trim();
6 console.log ( foo.length );          // 10
7 console.log ( foo );                // "JavaScript"
```

Disini saya menampilkan isi string `foo` sebelum dan sesudah pemanggilan method `trim()`. Hasil property `foo.length` menegaskan bahwa spasi di awal dan akhir string `foo` telah dihapus. Yakni dari 18 karakter menjadi 10 karakter. Inilah hasil dari perintah `foo = foo.trim()`.

Method `trim()` sering dipakai dalam proses validasi form. Kita bisa menggunakan untuk meminimalisir error akibat spasi yang tidak sengaja diinput oleh user. Contoh kasusnya seperti berikut ini:

```
1 var username = "admin ";
2 if (username === "admin") {
3   console.log("Welcome admin...");
4 }
5 else {
6   console.log("User tidak ditemukan");
7 }
8 // akan tampil: "User tidak ditemukan"
```

Pada baris keduan, saya membandingkan string "admin " dengan "admin". Hasilnya adalah: "User tidak ditemukan".

Ini terjadi karena variabel `foo` berisi string "admin" **dan sebuah spasi**. User yang tidak sadar telah menambah 1 karakter spasi akan bingung kenapa usernya salah. Method `trim()` bisa digunakan untuk mengatasi masalah ini:

```
1 var username = "admin ";
2 if (username.trim() === "admin") {
3   console.log("Welcome admin...");
4 }
5 else {
6   console.log("User tidak ditemukan");
7 }
8 // akan tampil: "Welcome admin..."
```

Sekarang, walaupun ada yang tidak sengaja menambah spasi di awal atau akhir string `admin`, operasi perbandingan tetap menghasilkan nilai `true`.

## 14.14 Method String.prototype.concat()

Method `concat()` digunakan untuk penyambungan string. Fungsinya mirip seperti operator '+'. String yang ingin disambung diinput sebagai argumen dari method `concat()`. Berikut contoh penggunaannya:

```
1 var foo = "Belajar";
2 var bar = foo.concat(" JavaScript");
3
4 console.log ( bar );
5 // Belajar JavaScript
6
7 console.log ( bar.concat(" di", " DuniaIkom", "... semangat!!" ) );
8 // Belajar JavaScript di DuniaIkom... semangat!!
```

Di awal kode program saya mendefenisikan variabel `foo = "Belajar"`. Hasil pemanggilan `foo.concat(" JavaScript")`, akan menampilkan "Belajar JavaScript". Begitu juga halnya di baris terakhir dimana saya menyambung string `bar` dengan 3 argumen dari method `concat()`.

Method `concat()` relatif jarang terpakai karena kita bisa menggunakan operator *string concatenation* atau operator penyambungan string menggunakan tanda '+'.

## 14.15 Method String.prototype.includes()

Method `includes()` digunakan untuk mengecek apakah sebuah string ada di dalam string asal atau tidak. Method ini memiliki 2 argumen:

- **Argumen pertama** adalah string yang ingin dicari.
- **Argumen kedua** (opsional) berupa index dimana pencarian akan dimulai. Jika argumen kedua ini tidak ditulis, pencarian akan dimulai dari awal string (nilai argumen kedua dianggap 0).

Proses pengecekan dilakukan secara **case sensitif**, dimana huruf kecil dan huruf besar dianggap berbeda. Berikut contoh penggunaan dari method `includes()` :

```
1 var foo = "Sedang belajar JavaScript dari buku JavaScript Uncover";
2 console.log ( foo.includes('Uncover') );           // true
3
4 console.log ( foo.includes('JavaScript') );        // true
5 console.log ( foo.includes('javascript') );        // false
6
7 console.log ( foo.includes('Sedang') );            // true
8 console.log ( foo.includes('Sedang',1) );          // false
9 console.log ( foo.includes('edang',1) );           // true
```

Disini saya mendefenisikan variabel `foo` dengan string “Sedang belajar JavaScript dari buku JavaScript Uncover”. Inilah string yang akan menjadi patokan kita.

Perintah `foo.includes('Uncover')` berarti apakah di dalam variabel `foo` terdapat string 'Uncover'? Seperti yang bisa kita lihat, di dalam variabel `foo` memang ada string 'Uncover', sehingga hasilnya **true**.

Pada contoh selanjutnya saya membandingkan perintah `foo.includes('JavaScript')` yang hasilnya **true** dengan `foo.includes('javascript')` yang hasilnya **false**. Perbedaan hasil ini terjadi karena method `includes()` bersifat **case sensitif**, string 'javascript' tidak sama dengan 'JavaScript'.

Di baris selanjutnya, hasil dari perintah `foo.includes('Sedang')` adalah **true**, karena di dalam string `foo` memang terdapat kata 'Sedang'. Akan tetapi pemanggilan `foo.includes('Sedang',1)` hasilnya **false**. Ini terjadi karena pencarian string mulai dari index 1, sedangkan huruf "S" dari kata "Sedang" berada di index 0.

Hasil perintah `foo.includes('edang',1)` menjadi **true** karena yang di index 0 hanya huruf "S", sehingga string 'edang' tetap ditemukan di dalam variabel `foo`.

## 14.16 Method String.prototype.startsWith() dan String.prototype.endsWith()

Kedua method ini mirip seperti `includes()`, yakni untuk mengecek apakah sebuah string ada di dalam string lain atau tidak. Bedanya, method `startsWith()` menambahkan syarat bahwa string yang dicari harus berada di awal, sedangkan method `endsWith()` string yang akan dicari berada di akhir.

Method `startsWith()` dan `endsWith()` memiliki 2 argumen yang sama persis seperti method `includes()`:

- **Argumen pertama** adalah string yang ingin dicari.
- **Argumen kedua** (opsional) berupa index dimana pencarian akan dimulai. Jika argumen kedua ini tidak ditulis, pencarian akan dimulai dari awal string (nilai argumen kedua dianggap 0).

Proses pengecekan dilakukan secara **case sensitif**, dimana huruf kecil dan huruf besar dianggap berbeda. Berikut contoh penggunaan dari method `startsWith()` dan `endsWith()` :

```

1 var foo = "Sedang belajar JavaScript dari buku JavaScript Uncover";
2 console.log ( foo.startsWith('Sedang') );      // true
3 console.log ( foo.endsWith('Uncover') );       // true
4
5 console.log ( foo.startsWith('sedang') );      // false
6 console.log ( foo.endsWith('uncover') );        // false
7
8 console.log ( foo.startsWith('belajar',7) );    // true
9 console.log ( foo.endsWith('belajar',14) );     // true

```

Menurut saya tidak perlu kita bahas lagi, karena penggunaannya sama seperti method `includes()`.

Khusus untuk 2 contoh terakhir, perintah `foo.startsWith('belajar',7)` hasilnya **true** karena kata 'belajar' dimulai dari index ke-7. Sedangkan `foo.endsWith('belajar',14)` hasilnya juga **true** karena kata 'belajar' berakhir di index ke-14.

## 14.17 Method String.prototype.repeat()

Fungsi dari method `repeat()` cukup simple, yakni mengulang string beberapa kali. Method ini membutuhkan 1 argumen yang menentukan banyak perulangan. Berikut contoh penggunaannya:

```

1 var foo = "semangat.. ";
2 console.log ( foo.repeat(1) );      // semangat..
3 console.log ( foo.repeat(3) );      // semangat.. semangat.. semangat..
4
5 console.log ( '=-'.repeat(10) );   // =====-

```

Perintah `foo.repeat(1)` artinya tampilkan isi variabel `foo` sebanyak 1 kali. Perintah `foo.repeat(3)` artinya tampilkan isi variabel `foo` sebanyak 3 kali. Di baris terakhir saya memanggil method `repeat(10)` langsung dari string literal.

## 14.18 Method String.prototype.toString() dan String.prototype.valueOf()

Method `toString()` dan `valueOf()` berfungsi untuk mengkonversi `String Object` menjadi tipe data primitif. Kedua method ini relatif jarang kita pakai dan hanya digunakan secara internal oleh JavaScript.

Berikut contoh penggunaan method `toString()` dan `valueOf()`:

```
1 var foo = new String ("Belajar JavaScript");
2 console.log ( typeof foo );    // object
3 console.log ( foo );
4 // String { "Belajar JavaScript", 19 more... }
5
6 bar = foo.toString();
7 console.log ( typeof bar );    // string
8 console.log ( bar );          // Belajar JavaScript
9
10 baz = foo.valueOf();
11 console.log ( typeof baz );   // string
12 console.log ( baz );         // Belajar JavaScript
```

Dalam kode program diatas, saya mengkonversi variabel `foo` yang asalnya `String object` menjadi tipe data primitif.

## 14.19 Method String.prototype.indexOf()

Method `indexOf()` berfungsi untuk mencari posisi index dari sebuah string. Sekilas method ini mirip seperti `includes()`, namun hasil akhirnya bukan hanya `true` atau `false`, tapi posisi dari string yang dicari.

Method `indexOf()` bisa diisi dengan 2 argumen:

- **Argumen pertama** adalah string yang ingin dicari.
- **Argumen kedua** (opsional) berupa index dimana pencarian akan dimulai. Jika argumen kedua ini tidak ditulis, pencarian akan dimulai dari awal string (nilai argumen kedua dianggap 0).

Jika method ini tidak menemukan string yang dicari, hasil akhirnya adalah `-1`. Pencarian dilakukan secara **case sensitif**, dimana huruf besar dan kecil akan dibedakan.

Berikut contoh penggunaan dari method `indexOf()`:

```
1 var foo = "Belajar JavaScript dari buku JavaScript Uncover";
2 console.log ( foo.indexOf('JavaScript') ); // 8
3 console.log ( foo.indexOf('buku') ); // 24
4 console.log ( foo.indexOf('Buku') ); // -1
```

Perintah pertama: `foo.indexOf('JavaScript')` hasilnya 8. Artinya string 'JavaScript' ditemukan di index ke-8 dari variabel `foo`. Begitu juga dengan perintah `foo.indexOf('buku')`, dimana string 'buku' ditemukan di posisi index ke 24.

Pada baris terakhir, perintah `foo.indexOf('Buku')` hasilnya -1. Artinya tidak ditemukan ada string 'Buku' di dalam variabel `foo`. Yang ada adalah string 'buku', bukan 'Buku'.

Mari kita coba menambahkan argumen kedua:

```
1 var foo = "Belajar JavaScript dari buku JavaScript Uncover";
2 console.log ( foo.indexOf('JavaScript') ); // 8
3 console.log ( foo.indexOf('JavaScript',9) ); // 29
4 console.log ( foo.indexOf('JavaScript',30) ); // -1
```

Di dalam string `foo`, terdapat 2 kali kata 'JavaScript', yakni di posisi index 8 dan 29. Ketika method `foo.indexOf('JavaScript')` dipanggil tanpa argumen, pencarian string dimulai dari awal (kiri). Jika ditemukan, langsung tampilkan hasilnya dan pencarian selesai.

Bagaimana cara untuk mencari kata 'JavaScript' berikutnya? Kita bisa menambahkan argumen kedua yang berisi posisi setelah kata 'JavaScript' pertama. Saya bisa mulai dari index ke-9. Dengan demikian, proses pencarian dimulai dari index ke-9. Perintah `foo.indexOf('JavaScript',9)` menghasilkan nilai 29. Artinya kata 'JavaScript' juga ditemukan di posisi index 29.

Di baris terakhir saya membuat proses pencarian mulai dari index ke 30. Hasilnya -1 karena tidak ditemukan lagi kata 'JavaScript' setelah index ke 30.

Sebagai contoh latihan, saya ingin membuat sebuah perulangan untuk mencari berapa kali sebuah kata muncul di dalam string asal. Silahkan anda pelajari sejenak kode program berikut:

```
1 var foo = "Belajar JavaScript dari buku JavaScript Uncover";
2 var count = 0;
3 var posisi = foo.indexOf('a');
4
5 while (posisi !== -1) {
6     count++;
7     posisi = foo.indexOf('a', posisi + 1);
8 }
9
10 console.log (count); // 7
```

Disini saya ingin menghitung berapa kali huruf 'a' muncul di dalam string `foo`. Kuncinya terletak di dalam perulangan. Program diatas akan terus dijalankan selama posisi `!== -1`.

Artinya, selama method `indexOf()` belum menghasilkan nilai `-1` (string tidak lagi ditemukan), perulangan terus berjalan.

Variabel counter dalam perulangan diatas adalah `posisi`, bukan `count`. Variabel `count` hanya sebagai penghitung berapa kali perulangan berjalan. Perintah `posisi = foo.indexOf('a'), posisi + 1` akan terus mencari posisi berikutnya dari string '`a`'. Jika tidak lagi ditemukan, variabel `posisi` menjadi `-1` dan perulangan berhenti.

## 14.20 Method String.prototype.lastIndexOf()

Method `lastIndexOf()` mirip seperti `indexOf()`, hanya saja proses pencarian di mulai dari akhir string, bukan dari awal string. Method ini juga bisa diisi dengan 2 argumen yang berfungsi sama sebagaimana method `indexOf()`.

Method `lastIndexOf()` bisa diisi dengan 2 argumen:

- **Argumen pertama** adalah string yang ingin dicari.
- **Argumen kedua (opsional)** berupa index dimana pencarian akan dimulai. Jika argumen kedua ini tidak ditulis, pencarian akan dimulai dari akhir string.

Jika method ini tidak menemukan string yang dicari, hasil akhirnya adalah `-1`. Pencarian dilakukan secara **case sensitif**, dimana huruf besar dan kecil akan dibedakan.

Berikut contoh penggunaannya:

```
1 var foo = "Belajar JavaScript dari buku JavaScript Uncover";
2 console.log ( foo.lastIndexOf('JavaScript') ); // 29
3 console.log ( foo.lastIndexOf('buku') ); // 24
4 console.log ( foo.lastIndexOf('Buku') ); // -1
```

Hasil pemanggilan `foo.lastIndexOf('JavaScript')` adalah `29`, artinya method ini mendapatkan string '`JavaScript`' berada di index ke `29`.

Berikut contoh yang menggunakan argumen:

```
1 var foo = "Belajar JavaScript dari buku JavaScript Uncover";
2 console.log ( foo.lastIndexOf('JavaScript') ); // 29
3 console.log ( foo.lastIndexOf('JavaScript',9) ); // 8
4 console.log ( foo.lastIndexOf('JavaScript',7) ); // -1
```

Saya rasa anda sudah bisa paham maksud dari kode program ini, dimana proses pencarian string '`JavaScript`' dimulai dari akhir (dari sebelah kanan).

Sebagai latihan, bisakah anda membuat program pencarian total jumlah string menggunakan method `lastIndexOf()`? Yakni seperti yang kita coba pada pembahasan method `indexOf()`. Silahkan modifikasi kode program kita sebelumnya.

Baik, berikut kode yang saya gunakan:

```

1 var foo = "Belajar JavaScript dari buku JavaScript Uncover";
2 var count = 0;
3 var posisi = foo.lastIndexOf('a');
4
5 while (posisi !== -1) {
6     count++;
7     posisi = foo.lastIndexOf('a', posisi - 1);
8 }
9
10 console.log (count); // 7

```

Selain mengubah method `indexOf()` menjadi `lastIndexOf()`, perubahan lain ada di argumen kedua method `lastIndexOf()`. Karena pencarian string dimulai dari akhir, maka kita menggunakan perhitungan `posisi - 1`. Dimana posisi pencarian akan mundur 1 karakter saat string ditemukan.

## 14.21 Method String.prototype.search()

Method `search()` berfungsi sebagaimana method `indexOf()`, yakni untuk mencari lokasi index dari sebuah string. Bedanya method `search()` mencari berdasarkan **regular expression** yang diinput sebagai argumen pertama method ini.

Jika hasilnya tidak ditemukan, method `search()` akan mengembalikan nilai `-1`. Berikut contoh penggunaannya:

```

1 var foo = "Belajar JavaScript dari buku JavaScript Uncover";
2 console.log ( foo.search('JavaScript') ); // 8
3 console.log ( foo.search('buku') ); // 24
4 console.log ( foo.search('Buku') ); // -1

```

Kode program diatas menghasilkan nilai yang sama seperti contoh pada method `indexOf()`.

Loh bukannya method `search()` hanya bisa diinput dengan *regular expression*? Betul, tapi jika yang diinput berupa string, JavaScript akan mengkonversinya menjadi *regular expression*. Kode diatas sama artinya dengan:

```

1 var foo = "Belajar JavaScript dari buku JavaScript Uncover";
2 console.log ( foo.search(/JavaScript/) ); // 8
3 console.log ( foo.search(/buku/) ); // 24
4 console.log ( foo.search(/Buku/) ); // -1

```

Mengenai apa itu **regular expression**, akan kita bahas dalam bab selanjutnya. Secara singkat, *regular expresion* adalah kumpulan karakter yang diproses untuk pengenalan pola (*pattern matching*). Dalam JavaScript, *regular expression* dibuat menggunakan tanda *forward slash* '/'.

Menggunakan *regular expression*, proses pencarian string menjadi sangat fleksibel. Sebagai contoh, proses pencarian bisa kita set sebagai *case insensitive*, yakni huruf besar dan kecil dianggap sama (tidak dibedakan):

```
1 var foo = "Belajar JavaScript dari buku JavaScript Uncover";
2 console.log ( foo.search('Buku') );           // -1
3 console.log ( foo.search(/Buku/i) );          // 24
```

Di baris kedua, saya menambahkan karakter **i** setelah penulisan *regular expression*. Dengan demikian, string 'Buku' juga akan cocok dengan 'buku'. Tanpa penambahan ini, method 'search()' bersifat *case sensitif*, dimana huruf besar dan kecil dianggap berbeda.

*Regular expression* juga bisa digunakan untuk mencari pola tertentu yang tidak bisa kita buat menggunakan pencarian string biasa, seperti contoh berikut:

```
1 var foo = "Belajar JavaScript dari buku JavaScript Uncover";
2 console.log ( foo.search(..ri/) );           // 12
3
4 var bar = "Ayo cari lagi";
5 console.log ( bar.search(..ri/) );          // 4
```

Perintah `foo.search(..ri/)` artinya, cari di dalam string `foo`, apakah ada sebuah kata yang terdiri dari 4 karakter, dimana kata tersebut diakhiri dengan `ri`. Ini akan cocok dengan kata `dari`, `cari`, `kiri`, `suri`, dst.

Di dalam variabel `foo`, kata `dari` berada di index ke-12. Sedangkan di variabel `bar`, kata `cari` berada di index ke 4. Inilah salah satu fitur dari *regular expression*.

## 14.22 String.prototype.match()

Method `match()` juga digunakan untuk pencarian *regular expression*. Tapi hasil akhir dari method ini berupa array. Pola *regular expression* juga diinput sebagai argumen pertama method ini. Jika tidak terdapat pola yang cocok, method `match()` akan mengembalikan nilai `null`.

Berikut contohnya:

```
1 var foo = "Belajar JavaScript dari buku JavaScript Uncover";
2 console.log ( foo.match('JavaScript') ); // Array [ "JavaScript" ]
3 console.log ( foo.match('a') );          // Array [ "a" ]
4 console.log ( foo.match('Tidak ada') ); // null
```

Disini saya mencoba memanggil method `match()` menggunakan string biasa (bukan *regular expression*). Terlihat array yang dihasilkan hanya 1 element.

Mari kita coba menggunakan *regular expression*:

```
1 var foo = "1 jam sama dengan 60 menit. Juga sama dengan 3600 detik";
2
3 console.log ( foo.match(/\d+/g) );
4 // Array [ "1", "60", "3600" ]
5
6 console.log ( foo.match(/\w*e\w*/g) );
7 // Array [ "dengan", "menit", "dengan", "detik" ]
```

Sekarang, hasil akhir dari method `match()` bukan lagi 1 element array, tapi ada banyak. Setiap element merupakan pola yang sesuai dengan regular expression.

Perintah `foo.match(/\d+/g)` artinya, cari seluruh angka yang terdapat di variabel `foo`, kemudian simpan tiap angka ini ke dalam array. Perintah `foo.match(/\w*e\w*/g)` artinya cari suatu kata di dalam variabel `foo` yang memiliki setidaknya 1 huruf `e`. Kata yang cocok dengan pola ini adalah: "dengan", "menit" dan "detik".

## 14.23 Method String.prototype.replace()

Method `replace()` digunakan untuk proses pergantian string dengan string lain. Method ini harus dipanggil dengan 2 argumen:

- **Argumen pertama** berupa pola *regular expression* yang ingin dicari.
- **Argumen kedua** adalah string pengganti seandainya ditemukan pola yang cocok.

Berikut contoh penggunaannya:

```
1 var foo = "Belajar JavaScript dari buku JavaScript Uncover";
2
3 console.log ( foo.replace("JavaScript","PHP") );
4 // Belajar PHP dari buku JavaScript Uncover
```

Disini saya mengisi argumen pertama dengan string "JavaScript". String ini akan dikonversi menjadi regular expression /JavaScript/. Artinya jika di dalam string `foo` terdapat kata "JavaScript", ganti kata tersebut menjadi "PHP".

Hasil dari perintah `foo.replace("JavaScript","PHP")` adalah "Belajar PHP dari buku JavaScript Uncover". Di dalam variabel `foo`, sebenarnya terdapat 2 kali kata "JavaScript". Namun yang diganti hanya kata pertama saja.

Bagaimana jika kita ingin mengganti seluruh kata "JavaScript" menjadi "PHP"? Caranya, gunakan *regular expression*:

```
1 var foo = "Belajar JavaScript dari buku JavaScript Uncover";
2
3 console.log ( foo.replace(/JavaScript/g,"PHP") );
4 // Belajar PHP dari buku JavaScript Uncover
5
6 console.log ( foo.replace(/a/g,"u") );
7 // Belajar JuvuScript dari buku JuvuScript Uncover
```

Perintah `foo.replace(/JavaScript/g,"PHP")` artinya ganti seluruh string yang cocok dengan pola `/JavaScript/` di dalam variabel `foo` menjadi string `"PHP"`.

Sedangkan perintah `foo.replace(/a/g,"u")` artinya cari seluruh huruf `a` dari variabel `foo`, kemudian ganti menjadi huruf `e`.

---

Dalam bab ini kita telah membahas sebagian besar property dan method **String Object** di dalam JavaScript. Selanjutnya, kita akan membahas **RexExp Object**, yakni tentang **Regular Expression**.

# 15. Regular Expression Object

**Regular expression** merupakan fitur pencocokan pola yang tersedia di hampir semua bahasa pemrograman. Di dalam JavaScript, regular expression disimpan ke dalam object tersendiri yakni **RegExp Object**.

Apabila anda sudah pernah menggunakan Regular Expression di bahasa pemrograman lain seperti PHP, regular expression di JavaScript terasa sangat mirip. Bahkan sebenarnya regular expression ini juga sama dengan yang dipakai di bahasa pemrograman lainnya.



Jika anda butuh panduan teknis (referensi) tentang **RegExp Object** yang lebih detail, silahkan buka [Standard built-in objects: RegExp<sup>1</sup>](#) dari Mozilla Developer Network.

## 15.1 Pengertian Regular Expression

**Regular Expression** atau sering disingkat sebagai **RegExp** atau **RE**, adalah suatu mekanisme pencocokan pola (*pattern matching*), yang dibuat menggunakan karakter-karakter khusus. Fungsinya sangat beragam, mulai dari memeriksa apakah sebuah inputan sudah sesuai atau belum (*test*), untuk membuat fitur pencarian (*search*), atau penggantian string (*replace*).

Contoh method yang menggunakan **RegExp** sudah kita lihat pada pembahasan tentang **String Object**. Diantaranya method `search()`, `match()`, dan `replace()`. Selain itu juga terdapat beberapa method yang khusus “melekat” ke **RegExp Object** JavaScript.

Penggunaan paling banyak dari **RegExp** adalah untuk proses validasi form. Sebagai contoh, bagaimana caranya kita memastikan seseorang sudah menginput alamat email dengan benar? Apakah yang diinput di kolom total pemesanan sudah berupa angka? Atau malah huruf? Bagaimana cara memastikan inputan password yang syaratnya harus dibuat dari 6 karakter dan mengandung minimal 1 angka dan 1 huruf besar? Ini semua bisa ditangani oleh **RegExp**.

## 15.2 Cara Membuat RegExp Object

Di dalam JavaScript, *regular expression* ditempatkan ke dalam object tersendiri, yakni **RegExp Object**. Sama seperti mayoritas object bawaan JavaScript lain, kita memiliki 2 cara penulisan: menggunakan **object constructor** atau cara langsung (**literal**).

Berikut contoh pembuatan **RegExp** di dalam JavaScript:

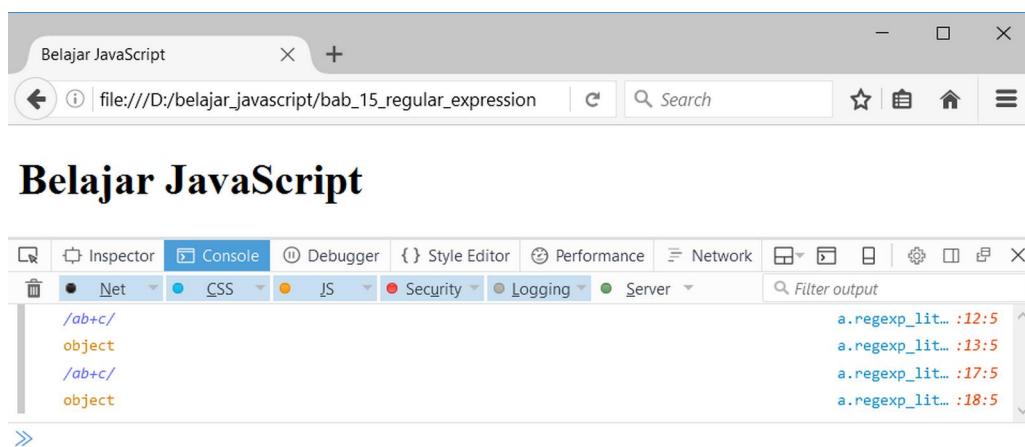
---

<sup>1</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/RegExp](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp)

```

1 var foo = /ab+c/;
2
3 console.log ( foo );           // /ab+c/
4 console.log ( typeof foo );   // object
5
6 var bar = new RegExp("ab+c");
7
8 console.log ( bar );         // /ab+c/
9 console.log ( typeof bar );  // object

```



Gambar: Cara pembuatan RegExp Object di dalam JavaScript

Menggunakan penulisan *literal*, kita tinggal membuat pola karakter diantara tanda *forward slash*: / dan /. Ini merupakan cara membuat regexp yang paling disarankan.

Menggunakan *object constructor*, perintahnya adalah new RegExp(). Dimana pola *regular expression* diinput sebagai argumen dari RexExp(). Cara ini tidak sering dipakai karena kurang efisien jika dibandingkan penulisan literal. Namun apabila pola *regular expression* diinput secara realtime (misalnya kode regexp diinput langsung dari form). Penulisan object constructor bisa dipakai.

## 15.3 RegExp Object Method

RexExp Object memiliki beberapa method dan property. Sebagian besar dari method ini merupakan fitur lanjutan yang relatif jarang dipakai. Kita hanya akan membahas 2 diantaranya: method test() dan exec().

## 15.4 Method RegExp.prototype.test()

Method test() digunakan untuk memeriksa apakah sebuah string lolos dari pola *regular expression* yang diinput. Jika lolos, hasilnya **true**. Jika tidak, hasilnya **false**.

Method ini mirip seperti includes() dari String Object. Bedanya, pada method test() pengecekan string menggunakan pola regular expression. Berikut contoh penggunaannya:

```

1 var foo = "Belajar JavaScript dari buku JavaScript Uncover";
2 var pola = /JavaScript/;
3
4 console.log ( pola.test(foo) );           // true
5 console.log ( /buku/.test(foo) );         // true
6 console.log ( /Buku/.test(foo) );          // false

```

Disini saya membuat 2 buah variabel: `foo` dan `pola`. Variabel `foo` berisi string, sedangkan variabel `pola` berisi pola regular expression `/JavaScript/`.

Perintah `pola.test(foo)` artinya saya ingin memeriksa apakah pola `/JavaScript/` terdapat di dalam string `foo` atau tidak. Pola `/JavaScript/` maksudnya sama dengan sebuah kata "JavaScript". Kata ini tentunya ada di dalam variabel `foo` dan hasil method `test()` adalah `true`.

Begitu juga dengan perintah `/buku/.test(foo)` yang artinya saya ingin memeriksa apakah kata "buku" ada di dalam string `foo`. Terlihat bahwa kita bisa langsung memanggil method `RexExp` dari penulisan literal, tanpa harus menyimpannya ke dalam variabel terlebih dahulu.

Perintah terakhir `/Buku/.test(foo)` hasilnya `false` karena perbedaan huruf besar di karakter "B". Artinya kata "Buku" tidak ditemukan (bersifat *case sensitif*). Lebih jauh tentang cara membuat pola `RegExp` akan kita bahas sesaat lagi.

## 15.5 Method `RegExp.prototype.exec()`

Method `exec()` berfungsi untuk mencari karakter atau kata yang cocok dengan pola regular expression, kemudian meyimpan hasilnya ke dalam array.

Berikut contoh penggunaan method `exec()`:

```

1 var foo = "1 jam sama dengan 60 menit, juga sama dengan 3600 detik";
2 var pola = /\d+/;
3
4 console.log ( pola.exec(foo) );           // Array [ "1" ]

```

Pola `\d+` artinya cari karakter angka dengan jumlah digit 1 atau lebih. Sehingga perintah `pola.exec(foo)` digunakan mencari karakter angka dari sting `foo`. Hasilnya adalah angka 1 yang menjadi element array. Ini merupakan fitur default dari method `exec()`, yakni ketika sebuah pola yang cocok sudah ketemu, method akan langsung berhenti.

Bagaimana cara mencari "semua" digit? Kita bisa menambahkan sebuah flag `g` atau penanda di dalam pola *regular expression*. Pola tersebut menjadi seperti ini:

```

1 var foo = "1 jam sama dengan 60 menit, juga sama dengan 3600 detik";
2 var pola = /\d+/g;
3
4 console.log ( pola.exec(foo) );           // Array [ "1" ]

```

Tapi, kenapa hasilnya tetap cuma 1 element array? Ternyata method `exec()` harus dipanggil beberapa kali. Pada setiap pemanggilan, method ini akan lanjut ke posisi berikutnya:

```

1 var foo = "1 jam sama dengan 60 menit, juga sama dengan 3600 detik";
2 var pola = /\d+/g;
3
4 console.log ( pola.exec(foo) );           // Array [ "1" ]
5 console.log ( pola.exec(foo) );           // Array [ "60" ]
6 console.log ( pola.exec(foo) );           // Array [ "3600" ]
7 console.log ( pola.exec(foo) );           // null

```

Setiap kali method `exec()` dipanggil, pencarian pola akan lanjut ke posisi berikutnya. Jika tidak ada lagi pola yang cocok, method ini akan mengembalikan nilai `null`.

Oleh karena fitur method `exec()` yang seperti ini, untuk mencari seluruh pola, kita harus menggunakan perulangan. Konsepnya, selama method `pola.exec(foo)` belum mengembalikan nilai `null`, lakukan terus pencarian pola.

Sebenarnya, method `exec()` tidak hanya mengembalikan array, tapi lebih mirip object:

```

1 var foo = "1 jam sama dengan 60 menit, juga sama dengan 3600 detik";
2 var pola = /\d+/g;
3
4 var hasil1 = pola.exec(foo);
5 console.log ( hasil1.index );           // 0
6 console.log ( hasil1[0] );              // 1
7
8 var hasil2 = pola.exec(foo);
9 console.log ( hasil2.index );           // 18
10 console.log ( hasil2[0] );              // 60
11
12 var hasil3 = pola.exec(foo);
13 console.log ( hasil3.index );           // 45
14 console.log ( hasil3[0] );              // 3600

```

Terlihat, setiap kali method `exec()` dipanggil, kita juga bisa mengakses property `index` yang berisi posisi index dimana pola itu ditemukan.

Jika anda hanya ingin mencari seluruh pola yang cocok (tanpa peduli posisinya), akan lebih sederhana menggunakan method `match()` dari **String object**:

```

1 var foo = "1 jam sama dengan 60 menit. Juga sama dengan 3600 detik";
2
3 console.log ( foo.match(/\d+/g) );
4 // Array [ "1", "60", "3600" ]

```

## 15.6 Pola Reguler Expression

Mempelajari pola *regular expression* bisa dibilang “gampang-gampang susah”. Karakter penyusun pola **regexp** tidak begitu banyak, tapi hasil dari pola yang ditulis cukup susah untuk dibaca. Agar mudah dipahami, kita akan banyak menggunakan contoh kode program.

## Pola RegExp Sebagai String

Mari kita mulai dari pola yang paling sederhana, yakni jika isi *regular expression* berbentuk string biasa yang terdiri dari sebuah kata atau beberapa karakter. Jika ditulis seperti ini, pola tersebut akan cocok selama di dalam string terdapat kata tersebut (di posisi mana saja). Berikut contohnya:

```

1 var foo = "Belajar JavaScript";
2
3 console.log ( /JavaScript/.test(foo) );    // true
4 console.log ( /Javascript/.test(foo) );    // false
5 console.log ( /Belajar/.test(foo) );        // true
6 console.log ( /belajar/.test(foo) );        // false
7 console.log ( /Java/.test(foo) );           // true
8 console.log ( /ajar/.test(foo) );           // true

```

Disini saya memiliki sebuah string "Belajar JavaScript" yang disimpan ke dalam variabel `foo`. Perhatikan penulisan pola regular expression, semuanya berisi kata-kata biasa. Method `test()` akan menghasilkan nilai `true` jika kata tersebut ada di dalam string `foo`.

## Regular Expression Flag

Dalam contoh sebelumnya, perintah `/Javascript/.test(foo)` dan `/belajar/.test(foo)` hasilnya adalah `false` karena method `test()` dijalankan secara **case sensitif**, dimana huruf besar dan kecil dianggap berbeda.

Bagaimana jika kita ingin proses pencocokan pola di lakukan secara **case insensitif**? Dimana huruf besar dan huruf kecil dianggap sama? Caranya adalah dengan menambahkan sebuah flag (penanda) karakter `i` persis setelah tanda *forward slash* penutup regular expression:

```

1 var foo = "Belajar JavaScript";
2
3 console.log ( /JavaScript/i.test(foo) );    // true
4 console.log ( /Javascript/i.test(foo) );    // true
5 console.log ( /JAVASCRIPT/i.test(foo) );    // true
6 console.log ( /jAvAsCRiPT/i.test(foo) );    // true

```

Dengan tambahan flag `i`, proses pencocokan pola akan dilakukan secara **case insensitif**. Selain flag `i`, terdapat juga beberapa flag lain dalam regular expression:

**g** = **global match**, digunakan untuk mencari **seluruh** string yang cocok dengan pola. Flag ini hanya bisa digunakan untuk method yang mendukung banyak pencarian sekaligus, seperti `match()` dan `exec()`. Jika flag ini tidak ditambahkan, regexp akan langsung berhenti di pola pertama.

**i** = **ignore case**, digunakan untuk membuat pencarian pola secara **case insensitif**, yakni huruf besar dan kecil dianggap sama. Jika flag ini tidak ditulis, proses pencarian dilakukan secara **case sensitif** dimana huruf besar dan kecil dianggap berbeda.

**m = multiline**, flag ini baru berfungsi jika kita membuat string asal yang terdiri dari beberapa kalimat / paragraf. Jika ditambahkan, pola regular expression akan cocok untuk setiap kalimat, bukan keseluruhan string. Efeknya digunakan untuk pola ^ dan \$ yang berfungsi sebagai tanda awal dan akhir string.

**u = unicode**, jika ditambahkan, pola regular expression akan dianggap sebagai code unicode.

Contoh penggunaan flag ini akan kita bahas kembali nantinya.

## Menandakan Awal dan Akhir Pola

Pola yang akan kita pelajari selanjutnya adalah untuk membedakan posisi dari karakter yang ingin dicari, apakah harus berada diawal string, diakhir string, atau keduanya. Untuk ini regular expression menyediakan 2 karakter khusus: ^ dan \$.

^ = Sebagai karakter penanda awal pola. Jika regular expression ditambahkan karakter ini, pola tersebut harus berada di awal string. Kalau tidak, hasilnya akan **false**.

\$ = Sebagai karakter penanda akhir pola. Jika regular expression ditambahkan karakter ini, pola tersebut harus berada di akhir string. Kalau tidak, hasilnya akan **false**.

Berikut contoh penggunaannya:

```

1 var foo = "Belajar JavaScript";
2
3 console.log ( /JavaScript/.test(foo) );           // true
4 console.log ( /^JavaScript/.test(foo) );          // false
5 console.log ( /JavaScript$/ .test(foo) );          // true
6 console.log ( /^Belajar/.test(foo) );              // true
7 console.log ( /^Belajar JavaScript$/ .test(foo) ); // true
8 console.log ( /Script$/ .test(foo) );              // true

```

Perintah /*JavaScript*/.*test*(*foo*) artinya, cek apakah pola "JavaScript" berada di awal dari string *foo*. Hasilnya menjadi **false** karena "JavaScript" bukan berada di awal, tapi di akhir string *foo*.

Perintah /*JavaScript*\$/.*test*(*foo*) artinya, cek apakah pola "JavaScript" berada di akhir dari string *foo*. Hasilnya **true** karena "JavaScript" memang berada di akhir dari string *foo*.

Kita juga bisa menggabungkan karakter penanda awal dan akhir string ini ke dalam 1 pola, seperti /*Belajar JavaScript*\$/.*test*(*foo*). Ini artinya, isi variabel *foo* harus persis "Belajar JavaScript" dan tidak boleh ada string lain baik di awal maupun di akhir string.

## Wildcard

Selanjutnya kita akan masuk ke karakter **wildcard**, yakni jika kita ingin mencari pola yang bisa diganti dengan karakter apa saja. Karakter *wildcard* ditulis dengan tanda titik ( . ).

. = Jika di dalam pola regular expression terdapat tanda titik, artinya karakter itu bisa digantikan dengan karakter apa saja.

Berikut contoh dari penggunaan dari pola *wildcard* ini:

```

1 var pola = /.b../;
2
3 console.log ( pola.test("abaa") );      // true
4 console.log ( pola.test("aba") );       // false
5 console.log ( pola.test("abbaa") );     // true
6 console.log ( pola.test("baab") );      // false
7 console.log ( pola.test("1b11") );      // true
8 console.log ( pola.test(" b ") );       // true

```

Pola `/.b../` akan menghasilkan **true** jika terdapat 1 karakter (apa saja), kemudian diikuti oleh huruf **b**, yang kemudian harus diikuti minimal 2 karakter apa saja. Contoh yang memenuhi syarat ini adalah string "abaa", "abbaa", "1b11", dan " b ". Khusus yang terakhir, karakter wildcard ini juga termasuk spasi.

Bagaimana jika saya tulis seperti ini? `/^ .b..$/`

Pola diatas hanya akan cocok jika keempat karakter tersebut ada di awal **dan** diakhir string. Maksudnya string tersebut harus terdiri dari 4 karakter, tidak boleh kurang maupun lebih:

```

1 var pola = /^ .b..$/;
2
3 console.log ( pola.test("abaa") );      // true
4 console.log ( pola.test("aba") );       // false
5 console.log ( pola.test("abbaa") );     // false
6 console.log ( pola.test("baab") );      // false
7 console.log ( pola.test("1b11") );      // true
8 console.log ( pola.test(" b ") );       // true

```

Terlihat bahwa string "abbaa" akan menghasilkan **false**, karena di akhir string harus ada 2 karakter setelah huruf b, tidak boleh lebih.

## Pola Character Set

Pola selanjutnya adalah **character set**, dimana kita bisa membuat syarat bahwa hanya karakter tertentu saja yang boleh ditulis. Ini dibuat menggunakan tanda kurung siku: [ dan ]. Hanya karakter yang ada di dalam tanda kurung ini saja yang akan memenuhi syarat.

Berikut contoh penggunaannya:

```

1 var pola = /[abcde].../;
2
3 console.log ( pola.test("abaa") );           // true
4 console.log ( pola.test("fba") );            // false
5 console.log ( pola.test("1dd") );            // false
6 console.log ( pola.test("add") );            // true
7 console.log ( pola.test(" dd") );            // false
8 console.log ( pola.test("belajar") );         // true

```

Pola `/[abcde].../` artinya, digit pertama hanya bisa diisi oleh salah satu dari huruf **a**, **b**, **c**, **d** atau **e**. Kemudian salah satu huruf tersebut diikuti setidaknya oleh 2 karakter lain (bebas mau karakter apa saja).

Dengan demikian, string yang memenuhi syarat ini adalah: "abaa", "add", dan "belajar". String "fba" menjadi **false** karena digit pertama diawali huruf f. String "1dd" juga **false** karena karakter pertama berupa angka, sedangkan string " dd" hasilnya **false** karena karakter pertama berisi spasi.

Pola `/[abcde].../` bisa juga ditulis menjadi: `/[a-e].../`.

Sebagai latihan, dapatkah anda memahami pola berikut: `/^ [a-e] [1-9] [R-Z] ./?` Disini saya menggabungkan beberapa karakter pola yang telah kita pelajari sebelumnya.

Pola diatas artinya, karakter pertama harus diawali dengan huruf a-e. Karakter kedua berupa angka 1-9. Karakter ketiga harus huruf R-Z, dan diikuti oleh 1 karakter apa saja (perhatikan ada tanda wildcard titik di akhir pola).

Seperti yang terlihat, walaupun kita baru membahas beberapa pola, *regular expression* terasa cukup rumit untuk dipahami.

Berikut hasil dari pola `/^ [a-e] [1-9] [R-Z] ./`:

```

1 var pola = /^ [a-e] [1-9] [R-Z] ./;
2
3 console.log ( pola.test("a9R") );           // false
4 console.log ( pola.test("a9Ra") );          // true
5 console.log ( pola.test("fa9Ra") );         // false
6 console.log ( pola.test("e9ZA") );          // true
7 console.log ( pola.test("e9ZAAAAA") );       // true

```

String "a9R" hasilnya **false** karena kurang 1 digit terakhir (wildcard). String "fa9Ra" juga **false** karena karakter pertama diawali oleh huruf f. String "e9ZAAAAA" hasilnya **true** karena di dalam pola ini kita tidak membatasi akhir string, jadi selama pola dipenuhi, tambahan karakter lain tidak akan mempengaruhi hasilnya.

## Pola Negasi Character Set

Pola negasi character set digunakan untuk menambahkan instruksi: **selain** ke dalam character set. Misalnya saya ingin digit pertama hanya boleh diisi **selain** huruf a - e. Untuk membuat pola seperti ini, kita menggunakan karakter ^.

Loh, bukannya tanda ^ berfungsi sebagai penanda awal string? Betul, karakter ^ memang juga digunakan untuk penanda awal string (dan tanda \$ sebagai penanda akhir string). Namun efeknya akan berbeda jika ditempatkan di dalam character set.

Contoh penggunaannya seperti ini: `/[^a-e].../`, Pola tersebut akan cocok dengan string yang didalamnya terdapat karakter **selain** huruf a-e, kemudian diikuti setidaknya 2 karakter lain (wildcard). Karakter ini tidak harus berada di awal string.

Berikut percobaannya:

```

1 var pola = /[^\u00e1-\u00e9].../;
2
3 console.log ( pola.test("abaa") );           // false
4 console.log ( pola.test("fba") );            // true
5 console.log ( pola.test("1dd") );            // true
6 console.log ( pola.test("ddd") );            // false
7 console.log ( pola.test(" dd") );            // true
8 console.log ( pola.test("belajar") );         // true

```

String "abaa" hasilnya **false** karena karakter pertama diawali huruf a. Begitu juga dengan string "ddd" yang juga menghasilkan **false**. String "belajar" hasilnya **true** karena cocok dengan teks "lajar".

Agar lebih paham, bisakah anda membedakan pola `/[^a-e].../` dengan pola `/^[\u00e1-\u00e9].../`?

Perhatikan untuk pola `/^[\u00e1-\u00e9].../` tanda ^ berada **di luar** character set. Pola tersebut artinya akan cocok dengan string yang **diawali** dengan digit a-e, kemudian diikuti setidaknya 2 karakter lain (wildcard).

Selain itu, pola `/^[\u00e1-\u00e9].../` juga harus berada di awal string, dan tidak boleh ada karakter lain sebelumnya. Tanda ^ disini sudah berfungsi sebagai penanda awal string, bukan lagi sebagai karakter negasi. Berikut hasilnya:

```

1 var pola = /^[\u00e1-\u00e9].../;
2
3 console.log ( pola.test("abaa") );           // true
4 console.log ( pola.test("fba") );            // false
5 console.log ( pola.test("1dd") );            // false
6 console.log ( pola.test("ddd") );            // true
7 console.log ( pola.test(" dd") );            // false
8 console.log ( pola.test("belajar") );         // true

```

Terlihat bahwa hasilnya berkebalikan dari pola `/^[\u00e1-\u00e9].../`

## Membatasi Jumlah Karakter

Regular expression juga mengizinkan kita membatasi jumlah karakter tertentu, misalnya huruf a harus ditulis 2 kali, atau digit angka harus tertulis sebanyak 8 - 12 kali (untuk nomor HP).

Karakter yang digunakan untuk membuat pola ini adalah tanda kurung kurawal { dan }. Jika di dalam tanda kurung ini hanya ada 1 angka, berarti digit tersebut harus sesuai dengan banyak angka. Pola /a{4}/ artinya huruf a harus ditulis sebanyak 4 kali. Berikut contohnya:

```
1 var pola = /A{2}1{3}/;
2
3 console.log ( pola.test("AA111") );           // true
4 console.log ( pola.test("zzAAA111zz") );      // true
5 console.log ( pola.test("A11") );             // false
6 console.log ( pola.test("AA11") );            // false
7 console.log ( pola.test("A1111") );           // false
```

Pola /A{2}1{3}/ akan cocok dengan string yang memiliki huruf A sebanyak 2 kali dan angka 1 sebanyak 3 kali. String "A11" menghasilkan **false** karena huruf A cuma ada 1 buah. Begitu juga dengan string "A1111". Sedangkan string "AA11" menjadi **false** karena angka 1 hanya ada 2 buah.

Selain membatasi jumlah karakter, kita juga bisa membatasi jangkauan jumlah karakter. Caranya dengan membuat 2 angka di dalam kurung kurawal. Angka pertama menentukan jumlah minimum, sedangkan angka kedua untuk menentukan nilai maksimum. Pola /a{2,4}/ artinya huruf a boleh ditulis 2 kali dan maksimal 4 kali.

Berikut contohnya:

```
1 var pola = /A{2}1{2,3}/;
2
3 console.log ( pola.test("AA111") );           // true
4 console.log ( pola.test("zzAAA111zz") );      // true
5 console.log ( pola.test("A11") );             // false
6 console.log ( pola.test("AA11") );            // true
7 console.log ( pola.test("A1111") );           // false
```

Pola /A{2}1{2,3}/ akan cocok dengan string yang memiliki huruf A sebanyak 2 kali dan angka 1 sebanyak 2 atau 3 kali. String "A11" menghasilkan **false** karena huruf A cuma ada 1 buah. Begitu juga dengan string "A1111". Sedangkan string "AA11" menjadi **true** karena angka 1 ada 2 buah dan sesuai dengan syarat minimum.

Jika digit kedua tidak ditulis itu artinya digit maksimal tidak dibatasi. Berikut contohnya:

```

1 var pola = /A{1,}1{3,}/;
2
3 console.log ( pola.test("AA111") );           // true
4 console.log ( pola.test("zzAAA111zz") );      // true
5 console.log ( pola.test("A11") );             // false
6 console.log ( pola.test("AA11") );            // false
7 console.log ( pola.test("A1111") );           // true

```

Pola `/A{1,}1{3,}/` akan cocok dengan string yang memiliki huruf A minimum 1 kali dan angka 1 minimum sebanyak 3 kali. String "A11" menghasilkan `false` karena angka 1 cuma ada 2 buah, begitu juga dengan string "AA11". Sedangkan string "A1111" menjadi `true` karena angka digit A dan angka 1 sudah memenuhi syarat minimum (lebih tidak masalah).

Sebagai latihan, bisakah anda mengartikan pola berikut? `/[A-Z]{2}[0-9]{1,4}[A-Z]{1,3}/`

Pola diatas terlihat panjang, tapi sebenarnya cukup sederhana. Artinya, dua karakter pertama harus diisi huruf kapital A-Z, 4 digit berikutnya harus diisi dengan angka 0-9 dengan jumlah digit minimal 1 dan maksimal 4. Kemudian diikuti oleh minimal 1 huruf dan maksimal 3 huruf A-Z.

Disini saya mencoba membuat pola regular expression untuk nomor polisi kendaraan bermotor di Indonesia. Berikut contoh prakteknya:

```

1 var pola = /[A-Z]{2}[0-9]{1,4}[A-Z]{1,3}/;
2
3 console.log ( pola.test("BM12345ARA") );      // false
4 console.log ( pola.test("BM1234ARA") );        // true
5 console.log ( pola.test("B99XYZ") );           // false
6 console.log ( pola.test("BA99XYZ") );          // true
7 console.log ( pola.test("Belajar BA99XYZ") );  // true

```

String "BM12345ARA" tidak lolos karena jumlah karakter angka melebihi batas maksimum, yakni 5 digit. String "B99XYZ" juga tidak lolos karena pada pola saya membuat huruf awal harus 2 karakter, padahal untuk nomor polisi sebenarnya boleh diawali 1 huruf. Silahkan anda perbaiki pola diatas supaya bisa meloloskan string ini.

Sebagai latihan kedua, bisakah anda membuat pola regular expression dengan syarat sebagai berikut:

- Dua digit pertama harus diisi huruf besar A-Z atau boleh juga berupa angka 0 - 9. Kalaupun lebih dari 2 digit, tidak masalah.
- Diikuti dengan 2 huruf z.
- Diakhiri dengan 1 karakter underscore : \_.
- Tidak boleh ada karakter lain sebelum dan sesudahnya.

Silahkan coba anda rancang pola regular expression berdasarkan syarat diatas.

Baik, berikut pola yang saya gunakan: `/^A-Z0-9]{2,}z{2}$_/`. Kenapa harus pakai tanda ^ dan \$? Karena di syarat tidak boleh ada karakter lain sebelum dan sesudah string. Berikut percobaannya:

```

1 var pola = /^[A-Z0-9]{2,}z{2}$_/;
2
3 console.log ( pola.test("A1zz_") );           // true
4 console.log ( pola.test("AABBCCzz_") );        // true
5 console.log ( pola.test("_AABBCCzz_") );       // false
6 console.log ( pola.test("ZZ0101zz_") );        // true
7 console.log ( pola.test("ZZ0101_zz_") );       // false
8 console.log ( pola.test("100101zz_") );        // true

```

String "\_AABBCCzz\_" hasilnya **false** karena terdapat 1 karakter lain sebelum string. Sedangkan string "ZZ0101\_zz\_" juga **false** karena terdapat 1 karakter underscore di antara pola (sebelum zz).

Selain menulis manual jumlah karakter, regular expression juga menyediakan beberapa karakter khusus untuk membatasi pola:

- \* (tanda bintang), sama fungsinya dengan {0,}. Artinya cocok dengan 0 atau lebih karakter (tidak dibatasi).

- + (tanda tambah), sama fungsinya dengan {1,}. Artinya cocok dengan 1 karakter atau lebih (tidak dibatasi).

- ? (tanda tanya), sama fungsinya dengan {0,1}. Artinya cocok dengan 0 atau 1 karakter (tidak boleh lebih).

Sebagai contoh, pola /ab\*c/ akan cocok dengan string yang diawali huruf a, kemudian diikuti tanpa atau beberapa huruf b, dan diakhiri dengan 1 huruf c. Contohnya seperti string "abc", "abbbbbbc", maupun "ac" akan menghasilkan **true**. Sedangkan string "aaaab" akan **false** karena tidak ada huruf c. Berikut prakteknya:

```

1 var pola = /ab*c/;
2
3 console.log ( pola.test("abc") );           // true
4 console.log ( pola.test("abbbbbbc") );        // true
5 console.log ( pola.test("ac") );             // true
6 console.log ( pola.test("aaaab") );          // false

```

Contoh lain, pola /ab+c/ akan cocok dengan string "abc" dan "abbbbbbc", namun akan menghasilkan **false** untuk string "ac" dan "aaaab". Pola /ab+c/ mensyaratkan huruf b harus ada minimal sekali:

```

1 var pola = /ab+c/;
2
3 console.log ( pola.test("abc") );           // true
4 console.log ( pola.test("abbbbbbc") );        // true
5 console.log ( pola.test("ac") );             // false
6 console.log ( pola.test("aaaab") );          // false

```

Bagaimana dengan pola /ab?c/ ? Pola ini artinya huruf b boleh ada maupun tidak, jika ada hanya boleh sekali (tidak bisa lebih). String "abc" dan "ac" akan menghasilkan **true**, sedangkan string "abbbbbbc" dan "aaaab" akan menghasilkan **false**:

```

1 var pola = /ab?c/;
2
3 console.log ( pola.test("abc") );           // true
4 console.log ( pola.test("abbbbbc") );        // false
5 console.log ( pola.test("ac") );             // true
6 console.log ( pola.test("aaaab") );          // false

```

## Pola Karakter Khusus

Untuk pola yang sering digunakan seperti digit angka 0-9, atau huruf abjad A-Z dan a-z, regular expression memiliki karakter khusus sebagai pengganti pola-pola ini. Berikut diantaranya:

\d: Cocok dengan seluruh digit angka, atau sama dengan [0-9].

\D: Cocok dengan seluruh digit **selain** angka, atau sama dengan [^0-9].

\w: Cocok dengan seluruh huruf alfabet dan angka (alfanumerik) serta karakter underscore: \_. Ini sama dengan pola [A-Za-z0-9\_].

\W: Cocok dengan seluruh huruf **selain** alfabet dan angka (alfanumerik) serta karakter underscore: \_. Ini sama dengan pola [^A-Za-z0-9\_].

\s: Cocok dengan karakter whitespace, seperti spasi, tab, form feed, atau line feed, termasuk karakter lain yang dianggap whitespace.

\S: Cocok dengan satu karakter **selain** whitespace.

\t: Cocok dengan satu karakter tab (*horizontal tab*).

\r: Cocok dengan satu karakter enter (*carriage return*).

\n: Cocok dengan satu karakter penanda pindah baris (*linefeed*).

\v: Cocok dengan satu karakter tab vertical (*vertical tab*).

\f: Cocok dengan satu karakter *form-feed*.

\ : Sebagai karakter ‘escape’.

Berikut contoh penggunaannya:

```

1 var pola = /\d\w\s/;
2
3 console.log ( pola.test("1v ") );           // true
4 console.log ( pola.test("3Ba") );            // false
5 console.log ( pola.test("9z ") );            // true
6 console.log ( pola.test("1Z") );             // false

```

Pola /\d\w\s/ akan cocok dengan string yang berisi 1 digit angka 0-9 (\d), diikuti 1 karakter alfanumerik (\w), dan diikuti 1 karakter whitespace (\s). String yang memenuhi syarat adalah "1v " dan "9z ". Sedangkan string "3Ba" dan "1Z" menghasilkan false karena karakter whitespace ketiga tidak ada.

Karakter backslash \ digunakan sebagai karakter **escape** untuk mencegah sebuah karakter dianggap sebagai karakter khusus. Misalnya bagaimana cara membuat karakter titik di dalam pola? Kita tidak bisa menulisnya langsung karena tanda titik punya makna khusus di dalam regular expression. Oleh karena itu harus ditulis sebagai \..

Sebagai contoh, dapatkan anda menerjemahkan pola /www\.....\..com/ ? Pola ini akan menghasilkan **true** untuk string yang diawali dengan karakter www, sebuah titik, diikuti oleh 4 karakter apa saja, sebuah titik, dan akhiran com. Berikut prakteknya:

```

1 var pola = /www\.....\..com/;
2
3 console.log ( pola.test("www.google.com") );      // false
4 console.log ( pola.test("www.abcd.com") );        // true
5 console.log ( pola.test("www.xyz1.com") );        // true
6 console.log ( pola.test("www.1234.com") );        // true

```

String "www.google.com" hasilnya **false** karena terdapat 5 karakter diantara tanda titik. Dimana syaratnya harus 4 karakter.

Bagaimana dengan pola berikut? /.+@.+\.+/ Ini artinya, mulai dengan 1 atau lebih karakter, diikuti dengan lambang @, disambung dengan 1 atau lebih karakter, sebuah tanda titik, dan diakhiri oleh 1 atau lebih karakter.

Pola ini merupakan versi sederhana untuk memeriksa sebuah alamat **email**. Berikut prakteknya:

```

1 var pola = /.+@.+\.+/";
2
3 console.log ( pola.test("aku@gmail.com") );        // true
4 console.log ( pola.test("hehe@co.cocok") );        // true
5 console.log ( pola.test("123@123.12") );          // true
6 console.log ( pola.test(" @ . ") );                // true
7 console.log ( pola.test("duniailkom@gmail.com") ); // true

```

Walaupun tampak tidak seperti alamat email, string " @ . " menghasilkan nilai **true** karena cocok dengan pola. Jika anda butuh pengecekan alamat email yang lebih tepat, pola regular expressionnya bisa sangat kompleks, seperti yang ada di [emailregex.com](http://emailregex.com)<sup>2</sup>.

## Pola Logika OR

Pola terakhir yang akan saya bahas (huff.. akhirnya..) adalah untuk membuat kondisi **OR**. Dengan fitur ini, kita bisa membuat pola yang jika tidak cocok, bisa menggunakan pola lain. Untuk membuat logika OR, digunakan karakter pipe: |.

Berikut contoh penggunaannya:

---

<sup>2</sup><http://emailregex.com>

```
1 var pola = /aku|dia|kami/;  
2  
3 console.log ( pola.test("aku di sini") ); // true  
4 console.log ( pola.test("dia di sana") ); // true  
5 console.log ( pola.test("kami belajar JavaScript") ); // true  
6 console.log ( pola.test("belajar JavaScript dengan dia") ); // true
```

Pola `/aku|dia|kami/` akan menghasilkan nilai `true` jika di dalam string terdapat setidaknya 1 kata dari 3 kemungkinan tersebut.

Sebagai latihan terakhir dari materi tentang regular expression, saya akan merevisi pola untuk memeriksa nomor polisi yang kita buat sebelumnya. Polanya adalah:

```
/^ [A-Za-z]{1,2} \s* \d{1,4} \s* [A-Za-z]{1,3} $/
```

Silahkan anda pelajari sejenak apa maksud dari setiap karakter diatas. Semuanya sudah kita bahas dari awal bab ini. Berikut hasil prakteknya:

```
1 var pola = /^ [A-Za-z]{1,2} \s* \d{1,4} \s* [A-Za-z]{1,3} $/;  
2  
3 console.log ( pola.test("B 1 RI") ); // true  
4 console.log ( pola.test("B1RI") ); // true  
5 console.log ( pola.test("DA 9999 XYZ ") ); // false  
6 console.log ( pola.test("DA 9999 XYZ") ); // true  
7 console.log ( pola.test("bk9he") ); // true  
8 console.log ( pola.test("zz 9YES") ); // true  
9 console.log ( pola.test("_zz9YES") ); // false
```

Sekarang, pola tersebut bisa menerima huruf besar maupun huruf kecil, dengan atau tanpa spasi, dan 1 atau 2 digit untuk huruf awal.

---

**Regular Expression** merupakan salah satu materi penting yang perlu anda ketahui dalam programming (tidak hanya JavaScript saja). Implementasi penggunannya sangat banyak, dimana salah satu yang terpenting adalah untuk validasi form, yakni pengecekan data yang diinput oleh user.

Berikutnya, kita akan mempelajari property dan method dari **Array Object** JavaScript.

# 16. Array Object

Array merupakan tipe data bentukan yang berisi banyak data. Di dalam JavaScript, array juga termasuk ke dalam **Object** yang memiliki berbagai property dan method. Dalam bab ini kita akan fokus membahas tentang **Array Object** JavaScript.

Array bisa ditulis menggunakan **object constructor** dengan perintah `new Array()`, maupun dengan penulisan **array literal** menggunakan tanda kurung siku `[ ]`. Berikut contohnya:

```
1 var foo = new Array("a", "b", "c", "d", "e");
2 console.log ( typeof foo );           // object
3 console.log ( foo );                // Array [ "a", "b", "c", "d", "e" ]
4
5 var bar = [ "a", "b", "c", "d", "e" ];
6 console.log ( typeof bar );         // object
7 console.log ( bar );               // Array [ "a", "b", "c", "d", "e" ]
```

Terlihat kedua cara pembuatan array ini tetap dianggap sebagai **object** oleh JavaScript. Namun karena lebih praktis dan efisien, cara pembuatan array yang lebih disarankan adalah menggunakan **array literal**, yakni membuat array menggunakan tanda kurung siku `[ ]`.



Jika anda butuh panduan teknis (referensi) tentang Array Object yang lebih detail, silahkan buka [Standard built-in objects: Array<sup>1</sup>](#) dari Mozilla Developer Network.

## 16.1 Array Object Method

**Array Object Method** adalah method yang melekat ke Array Object, bukan hasil instance-nya. Dari web Mozilla Developer Network terdapat 3 method dari Array Object:

- `Array.from()`
- `Array.isArray()`
- `Array.of()`

Kita hanya akan membahas method `Array.isArray()`, karena method yang lain cukup jarang dipakai.

## 16.2 Method Array.isArray()

Method `isArray()` digunakan untuk mengecek apakah isi dari suatu variabel berupa array atau tidak. Jika berupa array, hasil method ini adalah boolean `true`. Jika bukan array, hasilnya boolean `false`:

---

<sup>1</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

```
1 var a = new Array("a", "b", "c", "d", "e");
2 var b = ["a", "b", "c", "d", "e"];
3 var c = "aiueo";
4 var d = 123456;
5 var e = true;
6
7 console.log ( Array.isArray(a) );           // true
8 console.log ( Array.isArray(b) );           // true
9 console.log ( Array.isArray(c) );           // false
10 console.log ( Array.isArray(d) );          // false
11 console.log ( Array.isArray(e) );          // false
12 console.log ( Array.isArray([1,2,3]) );    // true
```

Saya membuat 5 variabel a, b, c, d dan e. Setiap variabel diisi dengan tipe data yang berbeda-beda. Terlihat, hanya variabel a dan b yang menghasilkan nilai **true**, karena hanya kedua variabel ini saja yang berisi array.

Di baris terakhir saya menginput langsung array literal sebagai argumen method `isArray()`, hasilnya juga **true**.

## 16.3 Array Instance Property

**Array instance property** adalah property yang melekat ke object hasil instance dari array. Di dalam JavaScript, hanya terdapat 1 array instance property: `Array.prototype.length`

## 16.4 Property `Array.prototype.length`

Property `length` berisi informasi mengenai jumlah element dari sebuah array. Berikut contoh penggunaannya:

```
1 var foo = ["a", "b", "c", "d", "e"];
2 console.log ( foo.length );           // 5
3
4 var bar = ["apel", "pisang", "anggur", "jambu"];
5 console.log ( bar.length );          // 4
6
7 console.log ( ["x", "y", "z"].length ); // 3
```

Property `length` ini sering digunakan dalam proses perulangan array, yakni sebagai penentu berapa banyak perulangan yang harus dilakukan, seperti contoh berikut:

```
1 var foo = ["apel", "pisang", "anggur", "jambu"];
2
3 for (var i=0; i < foo.length; i++) {
4   console.log( foo[i] ); // "apel", "pisang", "anggur", "jambu"
5 }
```

Perintah `console.log()` akan dijalankan sebanyak jumlah element array, yakni hingga kondisi `i < foo.length` bernilai **false**.

Yang juga perlu diingat, index terakhir dari sebuah array adalah jumlah property `length` - 1. Dikarenakan index array dimulai dari 0, bukan 1. Dalam contoh diatas, hasil perintah `foo.length` adalah 4, dimana index terakhir dari array `foo` adalah `foo[3]`.

Perulangan diatas berjalan seperti yang kita inginkan. Namun ada satu hal lagi yang bisa diperbaiki agar lebih efisien.

Kondisi `i < foo.length` akan terus diperiksa selama perulangan berlangsung. Artinya, JavaScript akan selalu mengecek nilai `foo.length` dalam setiap perulangan. Padahal nilai `foo.length` tidak akan pernah berubah (setidaknya untuk contoh kode diatas).

Supaya lebih efisien, sebaiknya nilai `foo.length` diambil **sebelum** perulangan, seperti contoh berikut:

```
1 var foo = ["apel", "pisang", "anggur", "jambu"];
2 var panjangArray = foo.length;
3
4 for (var i=0; i < panjangArray; i++) {
5   console.log( foo[i] ); // "apel", "pisang", "anggur", "jambu"
6 }
```

Disini, nilai `foo.length` saya ambil sebelum perulangan dan disimpan ke dalam variabel `panjangArray`. Variabel inilah yang akan di cek di dalam perulangan.

Kode programnya memang menjadi sedikit panjang, tapi lebih efisien. Sekarang JavaScript tidak perlu lagi mengecek kondisi `i < foo.length`, tapi cukup `i < panjangArray`. Cara ini tentunya tidak berlaku jika di dalam perulangan jumlah element array ikut diubah.

Property `length` array tidak hanya berisi informasi, tapi juga bisa diubah:

```
1 var foo = ["a", "b", "c", "d", "e"];
2 console.log ( foo.length ); // 5
3 console.log ( foo );
4 // Array [ "a", "b", "c", "d", "e" ]
5
6 foo.length = 3;
7 console.log ( foo.length ); // 3
8 console.log ( foo );
9 // Array [ "a", "b", "c" ]
```

Saya mendefenisikan array foo berisi 5 element. Kemudian mengubah nilai foo.length menjadi 3. Akibatnya, isi array foo juga berkurang menjadi 3 element.

Bagaimana jika property length ini malah ditambah?

```
1 var foo = [ "a", "b", "c", "d", "e" ];
2 console.log ( foo.length );           // 5
3 console.log ( foo );
4 // Array [ "a", "b", "c", "d", "e" ]
5
6 foo.length = 7;
7 console.log ( foo.length );           // 7
8 console.log ( foo );
9 // Array [ "a", "b", "c", "d", "e", <2 empty slots> ]
10
11 console.log ( foo[5] );  // undefined
12 console.log ( foo[6] );  // undefined
```

Disini saya menambah isi property foo.length menjadi 7. Akibatnya di dalam array foo akan hadir 2 element baru. Pada saat di cek, isi dari element baru ini berupa element kosong, atau undefined.

## 16.5 Array Instance Method

Tipe data array JavaScript memiliki banyak method, mengutip situs **Mozilla Developer Network**, terdapat setidaknya 30 array instance method:

- `Array.prototype.copyWithin()`
- `Array.prototype.fill()`
- `Array.prototype.pop()`
- `Array.prototype.push()`
- `Array.prototype.reverse()`
- `Array.prototype.shift()`
- `Array.prototype.sort()`
- `Array.prototype.splice()`
- `Array.prototype.unshift()`
- `Array.prototype.concat()`
- `Array.prototype.includes()`
- `Array.prototype.indexOf()`
- `Array.prototype.join()`
- `Array.prototype.lastIndexOf()`
- `Array.prototype.slice()`
- `Array.prototype.toSource()`

- `Array.prototype.toString()`
- `Array.prototype.toLocaleString()`
- `Array.prototype.entries()`
- `Array.prototype.every()`
- `Array.prototype.filter()`
- `Array.prototype.find()`
- `Array.prototype.findIndex()`
- `Array.prototype.forEach()`
- `Array.prototype.keys()`
- `Array.prototype.map()`
- `Array.prototype.reduce()`
- `Array.prototype.reduceRight()`
- `Array.prototype.some()`
- `Array.prototype.values()`

Seperti biasa, kita akan membahas sebagian besar diantara method-method ini, kecuali yang jarang digunakan atau cukup rumit untuk dibahas.

## 16.6 Method `Array.prototype.reverse()`

Method `reverse()` digunakan untuk membalik urutan element dari sebuah array. Berikut contoh penggunaannya:

```
1 var foo = ["a", "b", "c", "d", "e"];
2 console.log ( foo );    // Array [ "a", "b", "c", "d", "e" ]
3
4 var bar = foo.reverse();
5 console.log ( bar );    // Array [ "e", "d", "c", "b", "a" ]
6 console.log ( foo );    // Array [ "e", "d", "c", "b", "a" ]
```

Yang perlu diperhatikan, selain mengembalikan array yang sudah diubah, method `reverse()` juga merubah array asal. Dalam contoh diatas, urutan element di dalam array `foo` juga ikut berubah.

## 16.7 Method `Array.prototype.concat()`

Method `concat()` digunakan untuk proses penggabungan array. Argumen dari method ini bisa diisi dengan variabel array, maupun array literal (array yang biasanya kita buat menggunakan tanda kurung siku).

Berikut contoh penggunaannya:

```
1 var foo = ["a", "b", "c", "d"];
2 var bar = [1,2,3,4];
3
4 var fooBar = foo.concat(bar);
5 console.log( fooBar );
6 // Array [ "a", "b", "c", "d", 1, 2, 3, 4 ]
7
8 var barFoo = bar.concat(foo);
9 console.log( barFoo );
10 // Array [ 1, 2, 3, 4, "a", "b", "c", "d" ]
11
12 console.log( foo ); // Array [ "a", "b", "c", "d" ]
13 console.log( bar ); // Array [ 1, 2, 3, 4 ]
```

Saya membuat 2 buah array: `foo` dan `bar`. Perintah `var fooBar = foo.concat(bar)` artinya sambung array `foo` dengan array `bar`, kemudian hasilnya disimpan ke dalam variabel `fooBar`.

Terlihat isi dari variabel `fooBar` berupa gabungan kedua element array, dimulai dari isi array `foo`, kemudian diikuti dari element array `bar`. Jika pemanggilannya dibalik, yakni `bar.concat(foo)`, hasil urutan element akan dimulai dari variabel `bar`, baru diikuti variabel `foo`.

Di baris terakhir, saya menampilkan kembali isi dari variabel `foo` dan `bar`, terlihat kedua array tidak berubah. Artinya method `concat()` tidak mengubah variabel asal.

Bagaimana jika array yang ingin disambung merupakan array literal? Berikut contoh penggunaannya:

```
1 var foo = ["a", "b", "c", "d"];
2 var bar = foo.concat("e", "f", "g");
3
4 console.log( bar );
5 // Array [ "a", "b", "c", "d", "e", "f", "g" ]
```

Disini argument method `concat()` bukan lagi sebuah variabel, melainkan string. Perintah `var bar = foo.concat("e", "f", "g")` artinya, sambung element `"e"`, `"f"` dan `"g"` sebagai element berikutnya dari array `foo`, hasilnya disimpan ke dalam variabel `bar`.

Kita juga bisa menggabung variabel array dengan array literal untuk pemanggilan method `concat()`, seperti contoh berikut:

```
1 var foo = ["a", "b", "c", "d"];
2 var bar = [1,2,3,4];
3
4 var fooBar = foo.concat(bar, "e", "f");
5 console.log( fooBar );
6 // Array [ "a", "b", "c", "d", 1, 2, 3, 4, "e", "f" ]
```

Perintah `var fooBar = foo.concat(bar, "e", "f")` artinya: sambung array `foo` dengan array `bar`, diikuti dengan element `"e"` dan `"f"`, hasilnya kemudian disimpan ke dalam variabel `fooBar`.

## 16.8 Method Array.prototype.slice()

Method `slice()` digunakan untuk mengambil / men-copy sebagian element array. Method ini memiliki 2 argumen opsional, dengan ketentuan sebagai berikut:

- **Argumen pertama** (opsional) merupakan index awal pengambilan. Jika diisi angka positif, ambil element array mulai dari index tersebut. Jika diisi angka negatif, perhitungan di mulai dari akhir element.
- **Argumen kedua** (opsional) merupakan index akhir pengambilan. Jika diisi angka positif, ambil element array sampai index tersebut (tapi tidak termasuk index itu sendiri). Jika diisi angka negatif, perhitungan di mulai dari element terakhir. Apabila index ini tidak ditulis, ambil seluruh element hingga element terakhir.
- Jika method `slice()` dipanggil tanpa argument, seluruh element array akan diambil (di copy).

Berikut contoh penggunaannya:

```
1 var foo = [ "a", "b", "c", "d", "e", "f", "g" ];
2
3 console.log ( foo.slice() );
4 // Array [ "a", "b", "c", "d", "e", "f", "g" ]
5
6 console.log ( foo.slice(3) );
7 // Array [ "d", "e", "f", "g" ]
8
9 console.log ( foo.slice(3,5) );
10 // Array [ "d", "e" ]
11
12 console.log ( foo.slice(-5) );
13 // Array [ "c", "d", "e", "f", "g" ]
14
15 console.log ( foo.slice(-5,-2) );
16 // Array [ "c", "d", "e" ]
17
18 console.log ( foo );
19 // Array [ "a", "b", "c", "d", "e", "f", "g" ]
```

Di baris terakhir saya menampilkan kembali array `foo`, terlihat bahwa method `slice()` tidak mengubah array asal.

## 16.9 Method Array.prototype.splice()

Method `splice()` digunakan untuk menambah atau mengurangi element array. Method ini bisa diinput dengan beberapa argumen:

- **Argumen pertama** merupakan index awal penambahan / pengurangan. Jika diinput angka negatif, perhitungan dimulai dari element terakhir.
- **Argumen kedua** (opsional) bisa diinput dengan jumlah element yang akan dihapus. Jika diisi 0, artinya tidak ada element yang dihapus. Apabila tidak ditulis, seluruh sisa element akan dihapus (mulai dari index yang diinput pada argumen pertama).
- **Argument ketiga dan seterusnya** (opsional) bisa diinput dengan array baru yang ingin ditambahkan.

Berikut contoh penggunaan dari method `splice()` :

```

1 var foo = ["a", "b", "c", "d", "e", "f", "g"];
2 foo.splice(4);
3 console.log ( foo );
4 // Array [ "a", "b", "c", "d" ]
5
6 var bar = ["a", "b", "c", "d", "e", "f", "g"];
7 bar.splice(4,2);
8 console.log ( bar );
9 // Array [ "a", "b", "c", "d", "g" ]
10
11 var baz = ["a", "b", "c", "d", "e", "f", "g"];
12 baz.splice(4,2,"x","y","z");
13 console.log ( baz );
14 // Array [ "a", "b", "c", "d", "x", "y", "z", "g" ]
15
16 baz.splice(4,0,"1");
17 console.log ( baz );
18 // Array [ "a", "b", "c", "d", "1", "x", "y", "z", "g" ]

```

Perintah `foo.splice(4)` artinya awali perubahan dari element index ke-4. Karena argumen kedua tidak saya tulis, hasilnya adalah penghapusan element kelima hingga akhir array. Dengan kata lain `foo.splice(4)` artinya hapus semua element array `foo` kecuali 4 element pertama.

Perintah `bar.splice(4,2)` artinya awali perubahan dari element index ke-4, lalu hapus 2 element. Sisa element akan disambung ke element awal. Dari hasil `console.log ( bar )` terlihat bahwa huruf "e" dan "f" sudah tidak ada lagi.

Perintah `baz.splice(4,2,"x","y","z")` artinya awali perubahan dari element index ke-4, hapus 2 element, tambah element "x","y" dan "z" di posisi tersebut, kemudian sambung dengan sisa element array `baz`. Disini terjadi 2 operasi: menghapus element lama, dan menambahkan element baru.

Perintah `baz.splice(4,0,"1")` artinya awali perubahan dari element index ke-4, tambah string "1" di posisi tersebut, dan sambung dengan sisa element array `baz`. Argumen kedua saya input dengan nilai 0, artinya tidak ada penghapusan elemen lama, hanya terjadi penambahan 1 element baru.

Dari hasil percobaan, terlihat juga bahwa method `splice()` akan mengubah variabel array asal.

## 16.10 Method Array.prototype.join()

Method `join()` berfungsi untuk menggabungkan element array menjadi string. Method ini memiliki 1 argumen, yakni karakter pembatas sebagai pemisah antar element. Method `join()` merupakan kebalikan dari method `split()` dari String Object.

Berikut contoh penggunaan method `join()`:

```
1 var foo = ["a", "b", "c", "d", "e"];
2 console.log ( foo.join() );           // a,b,c,d,e
3 console.log ( foo.join("-") );        // a-b-c-d-e
4 console.log ( foo.join(" ") );        // a b c d e
5 console.log ( foo.join(" # ") );      // a # b # c # d # e
```

Jika method `join()` dipanggil tanpa argument, setiap element array akan disambung dengan tanda koma.

## 16.11 Method Array.prototype.push() dan Array.prototype.pop()

Method `push()` dan `pop()` digunakan untuk menambah dan mengurangi element array dari posisi terakhir. Dimana method `push()` untuk menambah element baru, sedangkan method `pop()` untuk mengeluarkan (dan menghapus) 1 element.

Berikut contoh penggunaan method `push()`:

```
1 var foo = ["a", "b", "c", "d", "e"];
2
3 foo.push("x");
4 console.log ( foo );
5 // Array [ "a", "b", "c", "d", "e", "x" ]
6
7 foo.push("y", "z");
8 console.log ( foo );
9 // Array [ "a", "b", "c", "d", "e", "x", "y", "z" ]
```

Perintah `foo.push("x")` artinya tambah string "x" di posisi terakhir array `foo`. Sedangkan perintah `foo.push("y", "z")` artinya tambahkan string "y" dan "z" di posisi terakhir array `foo`.

Berikut contoh penggunaan method `pop()`:

```
1 var foo = [ "a", "b", "c", "d", "e" ];
2 var hasil;
3
4 hasil = foo.pop();
5 console.log ( hasil );      // e
6 console.log ( foo );        // Array [ "a", "b", "c", "d" ]
7
8 hasil = foo.pop();
9 console.log ( hasil );      // d
10 console.log ( foo );       // Array [ "a", "b", "c" ]
```

Perintah `hasil = foo.pop()` artinya, ambil 1 element terakhir dari array `foo`, lalu simpan ke variabel `hasil`. Akibat perintah ini, element "`e`" dari array `foo` akan terhapus. Ketika saya menjalankan lagi perintah tersebut, giliran element "`d`" yang akan diambil. Karena element inilah yang sekarang berada di posisi terakhir array `foo`.

Dari hasil yang tampil, terlihat bahwa method `push()` dan `pop()` akan mengubah array asal `foo`.

Method `push()` dan `pop()` ini sering digunakan dalam pemrosesan array sebagai tumpukan (**stack**), atau dalam sistem antrian.

## 16.12 Method Array.prototype.unshift() dan Array.prototype.shift()

Method `unshift()` dan `shift()` sangat mirip seperti method `push()` dan `pop()`, bedanya element yang ditambah atau diambil akan diproses dari awal array.

Method `unshift()` digunakan untuk menambah element baru di awal array, sedangkan method `shift()` untuk mengambil 1 element dari awal array.

Berikut contoh penggunaan method `unshift()`:

```
1 var foo = [ "a", "b", "c", "d", "e" ];
2
3 foo.unshift("x");
4 console.log ( foo );
5 // Array [ "x", "a", "b", "c", "d", "e" ]
6
7 foo.unshift("y", "z");
8 console.log ( foo );
9 // Array [ "y", "z", "x", "a", "b", "c", "d", "e" ]
```

Perintah `foo.unshift("x")` artinya, tambahkan string "`x`" di posisi paling awal array `foo`. Sedangkan perintah `foo.unshift("y", "z")` artinya, tambahkan string "`y`" dan "`z`" di posisi paling awal array `foo`. Akibat penambahan ini, seluruh element array lain akan bergeser, menyesuaikan tempat dengan element yang baru ditambah ini.

Berikut contoh penggunaan method `shift()`:

```
1 var foo = ["a", "b", "c", "d", "e"];
2 var hasil;
3
4 hasil = foo.shift();
5 console.log ( hasil );      // a
6 console.log ( foo );        // Array [ "b", "c", "d", "e" ]
7
8 hasil = foo.shift();
9 console.log ( hasil );      // b
10 console.log ( foo );       // Array [ "c", "d", "e" ]
```

Perintah `hasil = foo.shift()` artinya, ambil element pertama dari array `foo`, lalu simpan ke dalam variabel `hasil`. Akibat perintah ini, element "a" dari array `foo` akan terhapus. Sisa element bergeser mengisi kekosongan yang ditinggalkan element "a".

Ketika perintah yang sama dijalankan lagi, giliran element "b" yang akan diambil. Karena element inilah yang berada di posisi pertama.

Method `unshift()` dan `shift()` ini sering digunakan dalam pemrosesan array sebagai tumpukan (**stack**), atau dalam sistem antrian.

## 16.13 Method Array.prototype.toString() dan Array.prototype.toLocaleString()

Kedua method ini digunakan untuk mengkonversi array menjadi string. Namun tidak seperti method `join()`, kita tidak bisa mengubah karakter pembatas antar element. Method `toString()` dan `toLocaleString()` menggabungkan element array dengan tanda koma.

Perbedaan antara `toString()` dan `toLocaleString()` adalah pada `toLocaleString()`, array akan diubah menurut settingan bahasa local yang digunakan web browser. Tapi dalam kebanyakan kasus (terutama yang menggunakan abjad latin), hasilnya sama saja dengan method `toString()`.

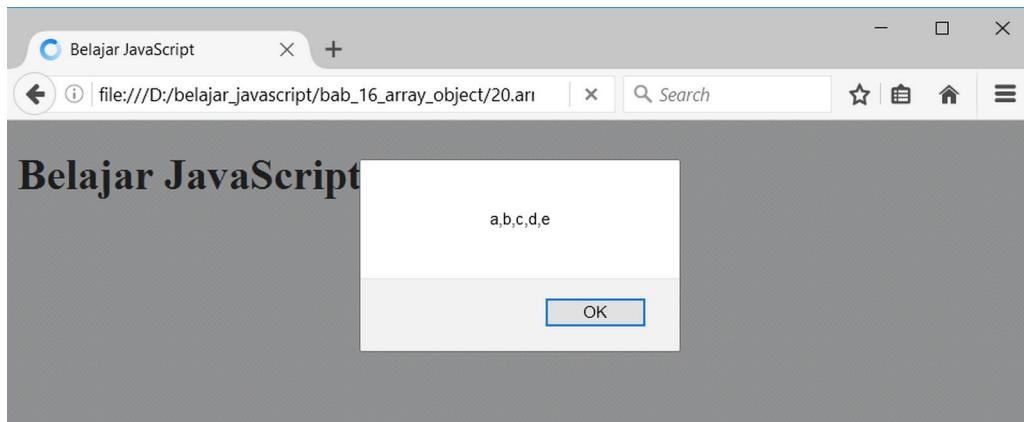
Berikut contoh penggunaan kedua method ini:

```
1 var foo = new Array("a", "b", "c", "d", "e");
2 console.log ( foo.toString() );           // a,b,c,d,e
3 console.log ( foo.toLocaleString() );      // a,b,c,d,e
4
5
6 var bar = ["a", "b", "c", "d", "e"];
7 console.log ( bar.toString() );           // a,b,c,d,e
8 console.log ( bar.toLocaleString() );      // a,b,c,d,e
```

Disini saya membuat array menggunakan object constructor dan array literal, keduanya menghasilkan tampilan yang sama saat dikonversi menjadi string menggunakan method `toString()` dan `toLocaleString()`.

Kedua method ini relatif jarang kita gunakan secara langsung. Biasanya method `toString()` otomatis dipanggil JavaScript ketika sebuah array “dipaksa” tampil sebagai string. Misalnya dalam perintah `alert()`:

```
1 var foo = ["a", "b", "c", "d", "e"];
2 alert(foo);
```



Gambar: Tampilan array foo di jendela alert

Disini saya ingin menampilkan array foo ke dalam jendela alert. Hasilnya, JavaScript akan menampilkan array foo berupa string yang sama persis seperti hasil pemanggilan `toString()`.

## 16.14 Method Array.prototype.includes()

Method `includes()` digunakan untuk mengecek apakah sebuah nilai ada di dalam array. Jika ada, method ini mengembalikan nilai `true`, namun jika nilai yang dicari tidak ditemukan, method ini mengembalikan nilai `false`.

Method `includes()` bisa diisi dengan 2 argument:

- **Argumen pertama** berupa element yang akan dicari.
- **Argumen kedua** (opsional) berupa nomor index awal pencarian. Jika tidak ditulis akan dianggap 0, yakni pencarian dimulai dari index paling awal.

Berikut contoh penggunaan method `includes()`:

```
1 var foo = [ "a", "b", "c", "d", "e" ];
2 console.log ( foo.includes("c") );           // true
3 console.log ( foo.includes("e") );           // true
4 console.log ( foo.includes("z") );           // false
5 console.log ( foo.includes("1") );           // false
6 console.log ( foo.includes("b", 1) );         // true
7 console.log ( foo.includes("b", 2) );         // false
```

Perintah `foo.includes("b", 2)` akan menghasilkan nilai `false` karena index pencarian dimulai dari 2, sedangkan huruf "b" berada di index ke 1.

## 16.15 Method Array.prototype.indexOf()

Method `indexOf()` mirip seperti method `includes()`, yakni sama-sama digunakan untuk mencari sebuah element di dalam array. Bedanya, hasil dari method `indexOf()` berupa posisi index dari element yang dicari. Apabila tidak ditemukan, method ini mengembalikan nilai `-1`.

Method `indexOf()` memiliki 2 argument:

- **Argumen pertama** diisi dengan element yang akan dicari.
- **Argumen kedua** (opsional) diisi dengan index awal pencarian. Jika tidak ditulis akan dianggap 0, yakni pencarian dimulai dari index paling awal.

Berikut contoh penggunaan method `indexOf()`:

```
1 var foo = [ "a", "b", "c", "d", "e" ];
2 console.log ( foo.indexOf("c") );           // 2
3 console.log ( foo.indexOf("e") );           // 4
4 console.log ( foo.indexOf("z") );           // -1
5 console.log ( foo.indexOf("1") );           // -1
6 console.log ( foo.indexOf("b", 1) );         // 1
7 console.log ( foo.indexOf("b", 2) );         // -1
```

Perintah `foo.indexOf("b", 2)` akan menghasilkan nilai `-1` karena index pencarian dimulai dari 2, sedangkan huruf "b" berada di index ke 1.

## 16.16 Method Array.prototype.forEach()

Dari semua method bawaan JavaScript yang telah kita pelajari hingga saat ini, seluruh argumen dari method tersebut berupa tipe data primitif, seperti string, angka, atau array. Sekarang, kita akan membahas method yang argumennya tidak lagi diisi dengan tipe data sederhana seperti itu, melainkan sebuah **function**.

Salah satu method yang argumennya berupa function adalah `forEach()`. Method `forEach()` sendiri berfungsi menjalankan sebuah function untuk setiap element array.

Pada bab tentang function, kita telah membahas bahwa function bisa diinput sebagai argumen. Dalam JavaScript, function yang diletakkan sebagai argument dikenal juga dengan istilah **Callback**.

Dikarenakan **method** pada dasarnya juga merupakan function, maka **Callback** ini adalah cara menginput function ke dalam function (mudah-mudahan anda tidak bingung dengan istilah ini).

Bagi saya pribadi, konsep ini memang sedikit rumit, karena itu kita akan membahasnya secara bertahap.

Method `forEach()` mirip seperti perulangan `for of`, yakni akan dijalankan sebanyak jumlah element dari array. Namun yang dijalankan disini adalah sebuah function. Berikut contoh dasar penggunannya:

```
1 var foo = ["a", "b", "c", "d", "e"];
2 foo.forEach( function() { console.log ("JavaScript"); } );
```

Disini saya membuat array `foo` dengan 5 element, kemudian memanggil method `forEach()`. Argument dari method `forEach()` adalah sebuah function, atau tepatnya *anonymous function*, karena function ini tidak memiliki nama. Isi dari function dalam contoh diatas berupa 1 baris perintah `console.log ("JavaScript")`.

Artinya, perintah `console.log ("JavaScript")` akan dijalankan untuk setiap element array `foo`. Karena array `foo` berisi 5 element, maka perintah `console.log()` juga dijalankan sebanyak 5 kali. Berikut hasilnya:



Gambar: Hasil perintah `console.log ("JavaScript")` sebanyak 5 kali

Seperti biasa, jika ada kode program yang isinya berulang, tab **Console** dari *Web Developer Tools* akan menggabung hasil tersebut. Angka 5 di kanan menandakan bahwa string "JavaScript" sebenarnya tampil 5 kali.

Function yang menjadi argument untuk method `forEach()` ini bisa sangat panjang, tergantung apa yang akan dijalankan di dalam function tersebut. Agar lebih mudah dibaca, saya bisa menulisnya jadi seperti ini:

```

1 var foo = ["a", "b", "c", "d", "e"];
2
3 foo.forEach(
4   function() {
5     console.log ("JavaScript");
6   }
7 );

```

Kode diatas sama persis seperti sebelumnya. Hanya saja saya memisahkan penulisan function ke baris tersendiri.

Ketika JavaScript menjalankan function **Callback**, method `forEach()` sebenarnya mengirim 3 argumen, yakni:

- **Argumen pertama** berupa nilai element yang sedang di proses (atau value array).
- **Argumen kedua** berupa index element yang sedang di proses (atau key array).
- **Argumen ketiga** berisi seluruh array asal.

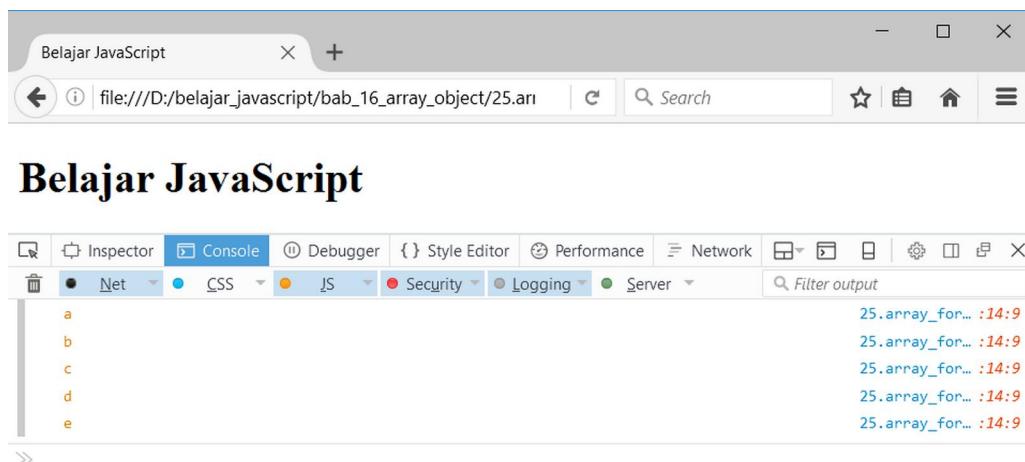
Ketiga argumen ini tidak harus ditulis, hanya jika dibutuhkan saja. Mari kita lihat cara penggunaan argumen ini:

```

1 var foo = ["a", "b", "c", "d", "e"];
2
3 foo.forEach(
4   function(element) {
5     console.log (element);
6   }
7 );

```

Disini, function **callback** saya input dengan 1 argumen: **element**. Selanjutnya, **element** ini ditampilkan menggunakan perintah `console.log (element)`. Berikut hasil kode program diatas:



Gambar: Isi element foo ditampilkan menggunakan method `forEach`

Seperti yang terlihat, argument `element` di dalam `callback` berisi element array yang saat ini sedang di proses. Dan nama argumen ini sebenarnya bisa menggunakan nama variabel apa saja, seperti berikut:

```

1 var foo = ["a", "b", "c", "d", "e"];
2
3 foo.forEach(
4   function(a) {
5     console.log (a);
6   }
7 );

```

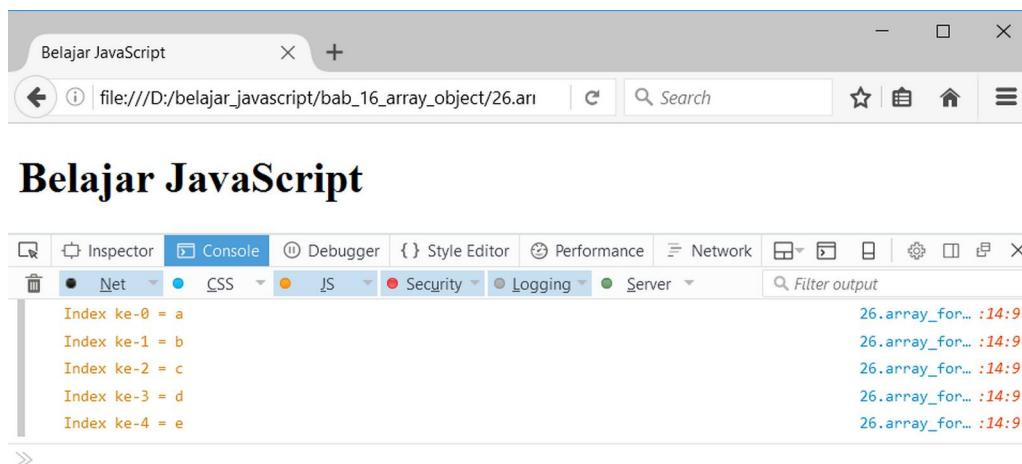
Kali ini saya membuat argumen `a` sebagai input untuk `callback`, dan tidak ada masalah. Yang terpenting adalah, argumen pertama berfungsi untuk mengakses element dari array `foo`.

Baik, mari kita tambah dengan argumen kedua:

```

1 var foo = ["a", "b", "c", "d", "e"];
2
3 foo.forEach(
4   function(element, index) {
5     console.log ("Index ke-" + index + " = " + element);
6   }
7 );

```



Gambar: Isi element dan index element `foo` ditampilkan menggunakan method `forEach`

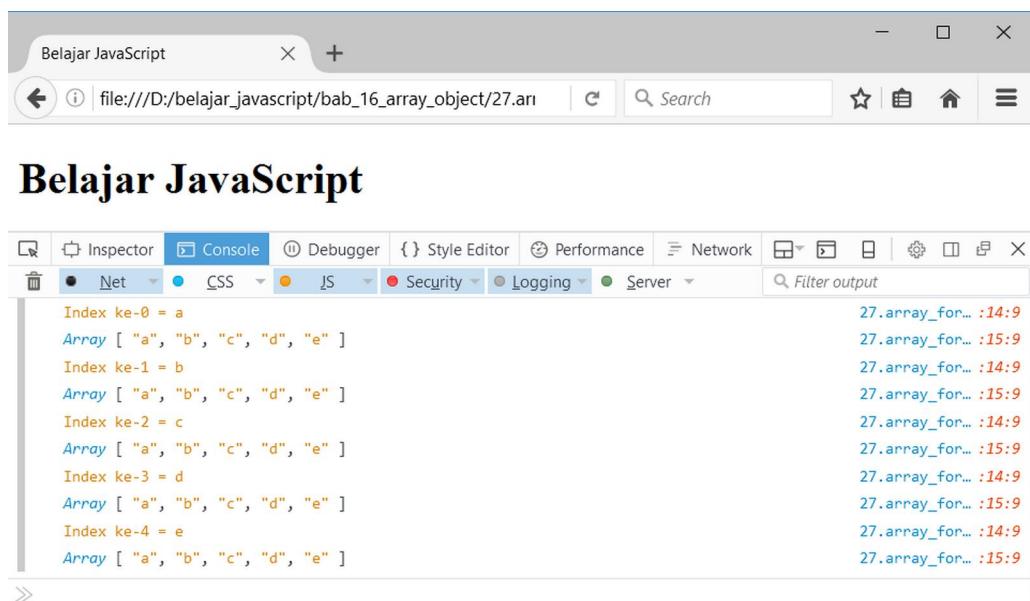
Argumen kedua dari `callback` diisi JavaScript dengan nomor index dari array asal (dalam contoh diatas, array `foo`). Argumen ini kemudian saya tampilkan menggunakan perintah `console.log()`. Sehingga kita bisa melihat index dan isi dari setiap element array `foo`.

Terakhir, mari kita tambahkan argumen ketiga:

```

1 var foo = ["a", "b", "c", "d", "e"];
2
3 foo.forEach(
4   function(element, index, array) {
5     console.log ("Index ke-" + index + " = " + element);
6     console.log (array);
7   }
8 );

```



Gambar: Isi element, index element, dan variabel asal foo ditampilkan menggunakan method forEach

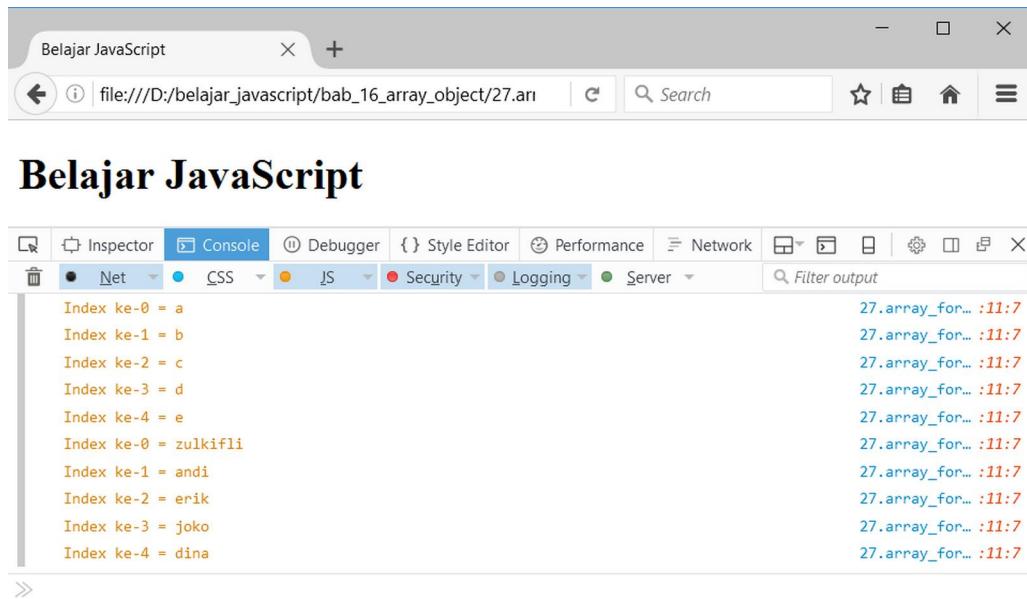
Argumen ketiga dari callback berisi seluruh data array dari variabel asal. Peranan argumen ketiga ini mungkin tidak sepenting argumen pertama dan kedua.

Dalam contoh diatas, saya membuat function **callback** secara langsung di dalam argumen **forEach()**. Function ini sebenarnya bisa dipisah menjadi fungsi tersendiri, seperti contoh berikut:

```

1 function tampil(element, index) {
2   console.log ("Index ke-" + index + " = " + element);
3 }
4
5 var foo = ["a", "b", "c", "d", "e"];
6 foo.forEach(tampil);
7
8 var bar = ["zulkifli", "andi", "erik", "joko", "dina"];
9 bar.forEach(tampil);

```



Gambar: Memisahkan function callback

Disini saya membuat sebuah function `tampil()`. Function ini nantinya dipanggil oleh method `forEach()`, dan berperan sebagai **callback**. Dengan memisahkan function **callback** keluar, function `tampil()` bisa dipakai oleh array lain.

Yang juga dapat dilihat, untuk memanggil function `tampil()` dari method `forEach()`, kita tidak perlu menambahkan argumen apa-apa, cukup nama fungsinya saja: `foo.forEach(tampil)`. JavaScript secara otomatis akan mengirim 3 argumen callback yang sudah kita bahas sebelum ini : **nilai element**, **index element**, dan **isi seluruh array**.

Konsep **callback** memang agak rumit, tapi **sangat penting** untuk dipahami. Jika anda masih kurang paham, saya sarankan untuk kembali mempelajari penjabaran diatas, karena sisanya method dari Array Object yang akan kita pelajari dalam bab ini, semuanya menggunakan **callback** function.

## 16.17 Method Array.prototype.map()

Method `map()` mirip seperti `forEach()`, dalam artian method ini juga menjalankan sebuah function **callback** sebagai argument. Bedanya, method `map()` akan mengembalikan sebuah array baru sebagai hasil callback.

Berikut contoh penggunaannya:

```
1 var foo = [8,4,7,9,3,2];
2
3 var bar = foo.map(
4   function (element, index, array) {
5     return element * 2;
6   }
7 );
8
9 console.log ( bar ); // Array [ 16, 8, 14, 18, 6, 4 ]
```

Di awal kode program, saya membuat array `foo` yang berisi 6 element angka. Kemudian saya menjalankan perintah `var bar = foo.map()`. Isi argumen dari method `foo.map()` berupa sebuah function **callback**.

Dalam function **callback**, setiap element dari array `foo` akan dikali 2, kemudian dikembalikan dengan perintah `return`, inilah maksud dari `return element * 2`. Hasil akhirnya, variabel `bar` akan berisi seluruh array `foo` yang telah dikali dua.

Bagaimana jika **callback** ini dikeluarkan? Tidak masalah!

```
1 function kaliDua(element) {
2   return element * 2;
3 }
4
5 var foo = [8,4,7,9,3,2];
6 var bar = foo.map( kaliDua );
7 console.log ( bar ); // Array [ 16, 8, 14, 18, 6, 4 ]
```

Dengan memisahkan penulisan fungsi **callback**, kode program menjadi lebih rapi. Disini saya membuat function `kaliDua()` yang isinya akan mengali seluruh element array dengan angka 2.



Jika anda kurang paham cara penulisan **callback** seperti ini, silahkan pelajari lagi pembahasannya pada penjelasan method `forEach()`.

Sebagai latihan, bisakah anda membuat method `map()` yang fungsinya mencari akar kuadrat dari setiap element array? Tipsnya, gunakan method `Math.sqrt()` sebagai fungsi bantu.

Baik, berikut kode program yang saya maksud:

```
1 function akarKuadrat(element){  
2   return Math.sqrt(element);  
3 }  
4  
5 var foo = [49,81,4,9,36,121];  
6 var bar = foo.map(akarKuadrat);  
7 console.log ( bar ); // Array [ 7, 9, 2, 3, 6, 11 ]
```

Hanya sedikit perubahan dari kode program kita sebelumnya.

Sebagai latihan kedua, bagaimana dengan membuat method `map()` yang akan menghitung pangkat 3 dari seluruh element array? Tipsnya, gunakan method `Math.pow()` sebagai fungsi bantu.

Berikut kode program yang saya gunakan:

```
1 function pangkatTiga (element){  
2   return Math.pow(element,3);  
3 }  
4  
5 var foo = [8,4,7,9,3,2];  
6 var bar = foo.map(pangkatTiga);  
7 console.log ( bar ); // Array [ 512, 64, 343, 729, 27, 8 ]
```

Sebagai latihan terakhir untuk method `map()`, bagaimana dengan sebuah kode program yang akan menfilter element array. Jika angka itu genap, nilainya tidak berubah. Tapi jika angka itu ganjil, ganti menjadi angka 0.

Tipsnya, disini kita harus menggunakan sebuah kondisi `if` untuk mengecek apakah element array yang diproses saat ini genap atau ganjil. Jika genap, `return element`. Jika ganjil, `return 0`.

Baik, berikut kota program yang saya gunakan:

```
1 function genap(element) {  
2   if (element % 2 === 0) {  
3     return element;  
4   }  
5   else {  
6     return 0;  
7   }  
8 }  
9  
10 var foo = [8,4,7,9,3,2];  
11 var bar = foo.map(genap);  
12 console.log ( bar ); //      Array [ 8, 4, 0, 0, 0, 2 ]
```

Silahkan anda pelajari sejenak kode program diatas. Sebagai hasil akhir, variabel `bar` akan berisi seluruh nomor genap dari array `foo`. Dimana angka ganjil akan digantikan dengan 0.

## 16.18 Method Array.prototype.filter()

Method `filter()` juga mirip seperti `map()`, yakni sama-sama menjalankan sebuah function *callback*. Bedanya pada method `filter()` hasil pemanggilan *callback* hanya bisa `true` atau `false`. Jika hasilnya `true`, pertahankan element array. Jika hasilnya `false`, hapus element tersebut.

Sebagai contoh, mari kita pakai studi kasus bilangan genap dari pembahasan method `map()` sebelum ini. Sekarang, hasil `return` dari function *callback* diganti menjadi `true` atau `false`:

```
1 function genap(element) {
2   if (element % 2 === 0) {
3     return true;
4   }
5   else {
6     return false;
7   }
8 }
9
10 var foo = [8,4,7,9,3,2];
11 var bar = foo.filter(genap);
12 console.log ( bar ); // Array [ 8, 4, 2 ]
```

Kode program diatas akan men-filter isi dari array `foo`. Jika di dalam array `foo` angkanya genap, pertahankan. Jika tidak genap (ganjil), hapus element tersebut.

Tentang element mana yang dipertahankan dan element mana yang mesti dihapus, diatur dari nilai kembalian *callback*. Kode `if (element % 2 === 0) { return true; }` Akan mempertahankan nilai genap. Selain itu, `return false`.

Sebagai latihan, bisakah anda men-filter element array dimana hanya element yang memiliki angka lebih besar atau sama dengan 30 saja yang dipertahankan, sisanya dihapus.

Berikut kode program yang saya maksud:

```
1 function besar30(element) {
2   if (element >= 30 ) {
3     return true;
4   }
5   else {
6     return false;
7   }
8 }
9
10 var foo = [23,60,44,12,34];
11 var bar = foo.filter(besar30);
12 console.log ( bar ); // Array [ 60, 44, 34 ]
```

Perintah `if (element >= 30) { return true; }` akan mempertahankan element mana saja yang dikirim ke variabel `bar`. Jika angka tersebut tidak lebih besar dari 30, maka `return false`. Hasil akhirnya, variabel `bar` hanya berisi element dengan nilai lebih besar dari 30.

## 16.19 Method Array.prototype.every()

Method `every()` digunakan untuk mengecek apakah setiap element array memenuhi syarat tertentu. "Syarat" ini harus kita buat sendiri menggunakan function `callback`.

Jika seluruh element lolos pemeriksaan callback (hasilnya `true` semua), maka method `every()` akan mengembalikan nilai `true`. Jika ada salah satu saja nilai element yang `false`, method `every()` akan mengembalikan nilai `false`.

Sebagai contoh kasus, saya memiliki sebuah array `foo`. Saya ingin memastikan seluruh element di dalamnya genap. Untuk contoh seperti inilah method `every()` bisa dipakai:

```
1  function genap(element) {
2      if (element % 2 === 0) {
3          return true;
4      }
5      else {
6          return false;
7      }
8  }
9
10 var foo = [6,8,10,12,16];
11 console.log ( foo.every(genap) );           // true
12
13 var bar = [3,8,10,12,16];
14 console.log ( bar.every(genap) );           // false
```

Array `bar` tidak lolos seleksi karena terdapat angka 3. Angka 3 ini akan menghasilkan nilai `false` dari callback, itulah sebabnya hasil dari `bar.every(genap)` juga `false`.

Sebagai contoh kedua, saya ingin mengecek apakah seluruh element array berisi angka yang besar dari 10. Silahkan anda coba rancang kode programnya.

Baik, berikut kode yang saya gunakan:

```
1 function besarDari10(element) {  
2     return element >= 10;  
3 }  
4  
5 var foo = [23,60,44,12,34];  
6 console.log ( foo.every(besarDari10) );           // true  
7  
8 var bar = [2,60,44,12,34];  
9 console.log ( bar.every(besarDari10) );           // false
```

Loh, kenapa isi function callback `besarDari10()` sangat singkat? Hanya `return element >= 10`?

Saya bisa menulis seperti ini karena hasil pengecekan `element >= 10` sudah otomatis menghasilkan `true` atau `false`, jadi kita tidak perlu membuat manual kode `return false` atau `return true`.

Bahkan kode pemeriksaan angka genap sebelumnya juga bisa saya tulis sebagai berikut:

```
1 function genap(element) {  
2     return (element % 2 === 0);  
3 }  
4  
5 var foo = [6,8,10,12,16];  
6 console.log ( foo.every(genap) );           // true  
7  
8 var bar = [3,8,10,12,16];  
9 console.log ( bar.every(genap) );           // false
```

Secara logika, kode program ini sama seperti pemeriksaan bilangan genap yang sebelumnya saya pakai (menggunakan `return true` dan `return false`). Hanya dipersingkat dengan sedikit analisis logika.

Sebagai latihan tambahan, trik seperti ini juga bisa digunakan untuk contoh bilangan genap ganjil pada pembahasan method `filter()`.

## 16.20 Method Array.prototype.some()

Method `some()` mirip seperti `every()`, tapi syaratnya terbalik. Method `some()` akan mengembalikan nilai `true` jika salah satu element saja memenuhi syarat. Dan akan mengembalikan nilai `false` hanya ketika seluruh element tidak memenuhi syarat.

Mari kita pakai contoh kasus bilangan genap sebelumnya:

```
1 function genap(element) {
2     return (element % 2 === 0);
3 }
4
5 var foo = [6,8,10,12,16];
6 console.log ( foo.some(genap) );           // true
7
8 var bar = [3,7,9,99,42];
9 console.log ( bar.some(genap) );           // true
10
11 var baz = [3,7,9,99,41];
12 console.log ( baz.some(genap) );           // false
```

Perintah `foo.some(genap)` dan `bar.some(genap)` menghasilkan `true` karena ada minimal 1 angka genap di dalam array. Sedangkan `baz.some(genap)` menghasilkan `false` karena tidak ada satupun memiliki angka genap.

Contoh kedua untuk memeriksa apakah bilangan besar dari 10:

```
1 function besarDari10(element) {
2     return element >= 10;
3 }
4
5 var foo = [23,60,44,12,34];
6 console.log ( foo.some(besarDari10) );       // true
7
8 var bar = [2,60,44,12,34];
9 console.log ( bar.some(besarDari10) );       // true
```

Baik array `foo` maupun `bar` sama-sama menghasilkan nilai `true`, karena di dalam kedua array ada minimal 1 angka yang besar dari 10.

## 16.21 Method Array.prototype.find() dan Array.prototype.findIndex()

Method `find()` dan `findIndex()` digunakan untuk mencari suatu nilai di dalam array berdasarkan syarat tertentu. Sama seperti method-method sebelumnya, syarat ini dibuat menggunakan `callback`.

Kedua method ini akan langsung berhenti dan mengembalikan nilai yang ditemukan pertama kali (jika element yang memenuhi syarat lebih dari 1). Method `find()` akan mengembalikan nilai element array tersebut, sedangkan method `findIndex()` akan mengembalikan `index` dimana element tersebut ditemukan.

Berikut contoh penggunaanya:

```

1 function genap(element) {
2   return (element % 2 === 0);
3 }
4
5 var foo = [5,7,14,12,16];
6 console.log ( foo.find(genap) );           // 14
7 console.log ( foo.findIndex(genap) );       // 2
8
9 var bar = [99,75,17,29,88];
10 console.log ( bar.find(genap) );          // 88
11 console.log ( bar.findIndex(genap) );       // 4

```

Disini saya menggunakan function **callback** `genap()`, yang akan mengembalikan nilai **true** jika angka yang diuji merupakan angka genap. Perintah `foo.find(genap)` artinya cari di dalam array `foo`, apakah ada angka yang memenuhi syarat *callback* `genap()` (menghasilkan **true**).

Proses pengujian dimulai dari index pertama hingga terakhir. Apakah  $5 \% 2 === 0$ ? **tidak**, lanjut ke element berikutnya. Apakah  $7 \% 2 === 0$ ? juga **tidak**. Apakah  $14 \% 2 === 0$ ? **benar**, function *callback* `genap()` akan mengembalikan nilai **true** dan method `find()` menghasilkan nilai **14**.

Begitu juga dengan perintah `foo.findIndex(genap)`. Proses pencarian akan berhenti di angka **14**, namun yang dikembalikan adalah index-nya, yakni index element `foo` ke **2**. Proses yang sama juga berlaku untuk array `bar`.

Sebagai contoh kedua, saya akan menggunakan **callback** `besarDari10()`:

```

1 function besarDari10(element) {
2   return element >= 10;
3 }
4
5 var foo = [15,7,14,12,16];
6 console.log ( foo.find(besarDari10) );      // 15
7 console.log ( foo.findIndex(besarDari10) );    // 0
8
9 var bar = [9,75,17,29,88];
10 console.log ( bar.find(besarDari10) );        // 75
11 console.log ( bar.findIndex(besarDari10) );     // 1
12
13 var baz = [9,3,4,5,6];
14 console.log ( baz.find(besarDari10) );         // undefined
15 console.log ( baz.findIndex(besarDari10) );      // -1

```

Prinsip kerjanya kurang lebih sama. Syarat kali ini adalah angka yang lebih besar dari **10**.

Untuk array `baz`, tidak ada element array yang nilainya lebih dari **10**. Oleh karena itu perintah `baz.find(besarDari10)` akan mengembalikan nilai `undefined`, dan perintah `baz.findIndex(besarDari10)` akan mengembalikan nilai **-1**.

## 16.22 Method Array.prototype.reduce() dan Array.prototype.reduceRight()

Kedua method ini digunakan untuk memproses total seluruh element array dan menghasilkan 1 nilai akhir. Sebagai contoh, saya ingin menambahkan semua isi array foo, atau ingin mengurangkan hasil pangkat dua dari seluruh isi array foo. Inilah yang bisa diproses menggunakan method `reduce()` dan `reduceRight()`.

Bedanya, pada method `reduce()` proses akan dimulai dari awal array, sedangkan method `reduceRight()` di proses dari akhir element array. Hasil kedua method ini hanya 1 nilai akhir. Proses untuk method `reduce()` dan `reduceRight()` melibatkan sebuah function **callback**, dan satu nilai awal (opsional).

Sebagai argumen ke dalam function **callback**, bisa diisi dengan 4 nilai:

- **Argumen pertama**: sebagai *accumulator*, atau variabel penampung total.
- **Argumen kedua**: nilai array yang saat ini sedang di proses.
- **Argumen ketiga** (opsional): index array yang saat ini sedang di proses.
- **Argumen keempat** (opsional): berisi seluruh element array.

Sedangkan untuk nilai awal (opsional) diinput setelah penulisan **callback**.

Penjelasan argumen diatas terasa cukup rumit, sehingga ada baiknya kita langsung lihat contoh kode program:

```
1 function tambah(total,angka) {  
2   return total + angka;  
3 }  
4  
5 var foo = [5,7,14,12,16];  
6 console.log ( foo.reduce(tambah) );           // 54  
7 console.log ( foo.reduce(tambah,10) );         // 64  
8  
9 console.log ( foo.reduceRight(tambah) );        // 54  
10 console.log ( foo.reduceRight(tambah,10) );      // 64
```

Dalam contoh ini, saya hanya menggunakan 2 buah argumen untuk function callback `tambah()`. Argumen pertama (`total`) sebagai penampung nilai akumulasi, dan argumen kedua (`angka`) sebagai penampung element array yang saat ini sedang diproses.

Ketika perintah `foo.reduce(tambah)` mulai diproses, nilai 5 akan masuk ke dalam argumen `total`, dan nilai 7 ke dalam argumen `angka`. Fungsi `tambah()` akan mengembalikan nilai  $5+7 = 12$ .

Selanjutnya nilai 12 ini akan dipindahkan ke dalam argumen `total`, dan argumen `angka` akan berisi element selanjutnya dari array `foo`, yakni 14. Fungsi `tambah()` akan mengembalikan nilai

$12+14 = 26$ . Proses seperti ini berlangsung hingga element terakhir dari array foo, dan hasilnya adalah 54. Ini didapat dari  $5 + 7 + 14 + 12 + 16 = 54$ .

Untuk perintah `foo.reduce(tambah,10)`, disini saya membuat angka 10 sebagai **nilai awal**. Akibatnya, pada saat diproses pertama kali, nilai 10 masuk ke argumen `total`, dan element pertama dari array foo (nilai 5) masuk ke dalam argumen `angka`. Proses penambahan yang sama juga akan berlangsung. Hasilnya adalah 64, yang didapat dari  $10 + 5 + 7 + 14 + 12 + 16 = 64$ .

Untuk method `reduceRight`, proses yang sama juga berlangsung. Hanya saja akan dimulai dari sisi kanan, atau dari element terakhir array foo. Dengan kata lain, perintah `foo.reduceRight(tambah)` akan memproses:  $16 + 12 + 14 + 7 + 5$ . Sedangkan perintah `foo.reduceRight(tambah,10)` akan memproses  $10 + 16 + 12 + 14 + 7 + 5$ .

Karena dalam teori matematika  $7 + 2$  sama saja dengan  $2 + 7$ , method `reduce()` dan `reduceRight()` akan menghasilkan nilai yang sama untuk operasi penambahan.

Sebagai latihan, bisakah anda membuat penjumlahan nilai kuadrat dari setiap element array foo? Tipsnya gunakan method `Math.pow()`.

Berikut contoh kode program yang saya gunakan:

```
1 function pangkat2(total,angka) {
2     return total + Math.pow(angka,2);
3 }
4
5 var foo = [5,7,14];
6 console.log ( foo.reduce(pangkat2) );           // 250
7 console.log ( foo.reduce(pangkat2,0) );          // 270
8
9 console.log ( foo.reduceRight(pangkat2) );       // 250
10 console.log ( foo.reduceRight(pangkat2,0) );      // 270
```

Pertanyaannya, kenapa hasil perintah `foo.reduce(pangkat2)` berbeda dengan hasil dari perintah `foo.reduce(pangkat2,0)`? Ini terjadi karena terdapat perbedaan perlakuan untuk element pertama variabel foo.

Pada saat perintah `foo.reduce(pangkat2)`, dijalankan pertama kali, element pertama dari array foo, yakni nilai 5 akan masuk ke argumen `total`, dan nilai 7 akan masuk ke argumen `angka`. Akibatnya, yang diproses adalah  $5 + 7^2 + 14^2 = 250$ .

Sedangkan untuk perintah `foo.reduce(pangkat2,0)`, pada saat pertama kali dijalankan, argumen `total` akan berisi 0, dan argument `angka` akan berisi nilai 5. Sehingga yang diproses adalah:  $0 + 5^2 + 7^2 + 14^2 = 270$ . Tentu saja inilah hasil akhir yang kita inginkan.

Karena juga proses penambahan, hasil dari `foo.reduce(pangkat2)` tidak berbeda dengan `foo.reduceRight(pangkat2)`.

Sekarang, bisakah anda menjelaskan perbedaan dari hasil berikut?

```
1 function bagi(total,angka) {  
2     return total / angka;  
3 }  
4  
5 var foo = [49,7,2];  
6 console.log ( foo.reduce(bagi) );           // 3.5  
7 console.log ( foo.reduceRight(bagi) );      // 0.0058309037900874635
```

Hasil dari `foo.reduce(bagi)` akan berbeda dengan `foo.reduceRight(bagi)`, kenapa?

Karena operasi pembagian akan menghasilkan nilai yang berbeda jika urutannya dibalik. Untuk perintah `foo.reduce(bagi)`, yang akan diproses adalah  $= 49/7/2 = 7/2 = 3.5$ . Sedangkan perintah `foo.reduceRight(bagi)`, yang akan diproses adalah  $= 2/7/49 = 0.28/49 = 0.0058$ .

Method `reduce()` cukup sering digunakan untuk mencari total jumlah element dari sebuah array. Penggunaannya terkesan rumit, tapi cukup mudah ditulis jika anda sudah paham proses kerja dari method ini.

## 16.23 Mehtod Array.prototype.sort()

Sesuai dengan namanya, method `sort()` digunakan untuk proses pengurutan element array. Method ini memiliki 1 argumen opsional yang bisa diisi dengan sebuah function `callback` yang berfungsi untuk mengatur proses pengurutan.

Jika function `callback` tidak disertakan, proses pengurutan akan menggunakan nomor urut kode Unicode.

Berikut contoh penggunaannya:

```
1 var foo = ["zulkifli","andi","erik","joko","dina"];  
2 foo.sort();  
3 console.log ( foo );  
4 // Array [ "andi", "dina", "erik", "joko", "zulkifli" ]
```

Variabel `foo` saya isi dengan 5 string yang tidak tersusun. Dengan memanggil method `sort()`, semua string ini akan diurutkan berdasarkan abjad. Juga terlihat bahwa method `sort()` akan mengubah array asal.

Bagaimana dengan nilai angka? Mari kita coba:

```
1 var foo = [3,5,2,8,1,31,22,44,33,11];  
2 foo.sort();  
3 console.log ( foo );  
4 // Array [ 1, 11, 2, 22, 3, 31, 33, 44, 5, 8 ]
```

Hasilnya tidak seperti yang kita harapkan. Angka 11 diletakkan sebelum angka 2. Kenapa ini bisa terjadi? Karena pada saat pengurutan, JavaScript akan mengkonversi setiap angka menjadi string, baru kemudian membandingkan nomor urut karakter unicode dari string-string ini. Kode unicode untuk string 11 lebih kecil dari string 2.

Untuk mengatasi masalah diatas, kita harus membuat sebuah function **callback** sebagai pengatur urutan.

Function **callback** untuk method `sort()` menerima 2 input argumen, yakni 2 nilai yang saat ini sedang dibandingkan, misalnya `a` dan `b`. Untuk menentukan nilai mana yang harus di dahulukan, dilihat dari nilai kembalian function:

- Jika kembalian function *callback* kurang dari 0, nilai `a` dianggap lebih kecil. `a` diurutkan sebelum `b`.
- Jika kembalian function *callback* sama dengan 0, nilai `a` dan `b` dianggap sama dan urutannya tidak diubah.
- Jika kembalian function *callback* besar dari 0, nilai `b` dianggap lebih kecil. `a` diurutkan setelah `b`.

Mari kita lihat contoh prakteknya:

```
1 function bandingkan(a, b) {  
2     if (a < b) {  
3         return -1; // a lebih kecil  
4     }  
5     if (a > b) {  
6         return 1; // b lebih kecil  
7     }  
8     return 0; // sama besar  
9 }  
10  
11 var foo = [3,5,2,8,1,31,22,44,33,11];  
12 foo.sort(bandingkan);  
13 console.log ( foo );  
14 // Array [ 1, 2, 3, 5, 8, 11, 22, 31, 33, 44 ]
```

Saya membuat sebuah function *callback* `bandingkan()`. Function ini memiliki 2 argumen, yakni `a` dan `b`. Kedua argumen akan diisi oleh setiap element dari array `foo`:

- Jika `a < b`, return `-1`.
- Jika `a > b`, return `1`.
- Selain keduanya (berarti `a` sama dengan `b`), return `0`.

Dengan bantuan fungsi *callback* ini, proses pengurutan angka sudah sesuai. Namun kita juga bisa membuat kode program yang lebih singkat, seperti berikut ini:

```
1 function bandingkan(a, b) {  
2     return a - b;  
3 }  
4  
5 var foo = [3,5,2,8,1,31,22,44,33,11];  
6 foo.sort(bandingkan);  
7 console.log ( foo );  
8 // Array [ 1, 2, 3, 5, 8, 11, 22, 31, 33, 44 ]
```

Sekarang isi dari function *callback* bandingkan() hanya 1 baris, yakni `return a - b;`. Satu baris ini sudah mewakili logika yang sama seperti kode sebelumnya, yakni jika a lebih kecil dari b, hasilnya sebuah angka negatif. Jika a lebih besar dari b, hasilnya angka positif, dan jika sama hasilnya 0.

Method `sort()` bisa digunakan untuk menyusun element array sesuai urutan. Namun seperti yang terlihat, untuk pengurutan nilai angka prosesnya membutuhkan sedikit bantuan.

---

Dalam bab ini kita telah membahas sebagian besar property dan method yang melekat ke **Array Object** JavaScript. Array sendiri nantinya banyak kita gunakan pada saat pembahasan tentang struktur **DOM** (Document Object Model).

Berikutnya, kita akan masuk ke **Date Object** JavaScript.

# 17. Date Object

Date object merupakan object khusus di dalam JavaScript untuk memproses tanggal (dan waktu). Object ini memiliki berbagai method yang bisa kita gunakan.

Walaupun terkesan sederhana, pemrosesan tanggal bisa menjadi kompleks karena adanya perbedaan waktu antara satu tempat dengan tempat lain. Masalah ini semakin rumit karena JavaScript dengan membagi 2 method untuk Date object: **Locale Time** dan **UTC Time**. Ini semua akan kita bahas dengan detail nantinya.

## 17.1 Cara Membuat Date Object

Untuk membuat Date Object, kita harus menggunakan cara penulisan *object constructor*, yakni menggunakan perintah `new`. Ini karena Date object tidak memiliki cara penulisan *literal*.

Berikut cara pembuatan *Date object*:

```
1 var foo = new Date();
2 console.log( foo );           // Date 2016-12-04T11:24:53.053Z
3 console.log( typeof foo );   // object
```

Jika *date constructor* dipanggil tanpa argumen seperti contoh diatas, yakni dengan perintah `new Date()`, tanggal yang disimpan adalah **tanggal sekarang**, atau lebih tepatnya tanggal ketika kode tersebut diproses.

Hasil dari perintah `console.log( foo )` berbentuk *string*: `Date 2016-12-04T11:24:53.053Z`. Ini adalah format tanggal **ISO** yang digunakan JavaScript.

Artinya, tanggal ketika saya menjalankan kode tersebut adalah: 04 - 12 - 2016 pukul 11:24:53 dan 053 milidetik. Betul, JavaScript menyimpan ketelitian hingga **milidetik** (*millisecond*). Dari tampilan hasil format ISO, antara tanggal dengan waktu dipisah dengan huruf T, dan diakhir terdapat tambahan huruf z.

Bagaimana jika saya ingin menampilkan tanggal dengan format yang lebih rapi? Misalnya 04/12/2016, atau 04 Desember 2016? JavaScript menyediakan berbagai method untuk keperluan ini. Sebelum sampai kesana, kita akan fokus dulu membahas **Date constructor**.

Seperti yang telah kita lihat, jika dipanggil tanpa argument, *Date constructor* akan menyimpan tanggal saat ini. Bagaimana cara membuat **tanggal lain**? Kita bisa menginput argumen tambahan ke dalam perintah `new Date()`.

Terdapat 3 cara untuk membuat tanggal tertentu ke dalam **Date constructor**:

- `new Date(year, month, day, hours, minutes, seconds, milliseconds)`.
- `new Date(dateString)`.

- `new Date(milliseconds)`.

Cara pertama adalah menginput 7 argumen yang masing-masingnya untuk men-set: tahun (*year*), bulan (*month*), hari (*day*), jam (*hours*), menit (*minutes*), detik (*seconds*), dan milidetik (*milliseconds*).

Sebagai contoh, berikut kode program untuk membuat *object Date* yang berisi tanggal 2 Desember 2016 pukul 9:30:15 lebih 125 milidetik:

```
1 var foo = new Date(2016,11,2,9,30,15,125);
2 console.log( foo );      // Date 2016-12-02T02:30:15.125Z
```

Jika anda teliti, ada yang salah dari cara saya menginput bagian bulan (*month*). Bukankah bulan desember itu bulan ke 12? kenapa di input 11?

Di dalam object **Date** JavaScript, bulan ditulis menggunakan index 0 - 11. Artinya bulan Januari = 0, Februari = 1, Maret = 2, dst hingga Desember = 11.

Baik, mari kita lihat hasilnya: `Date 2016-12-02T02:30:15.125Z`. Sekali lagi, apakah anda juga bisa mencari hal yang salah? Fokuskan pada penulisan jam.

Betul, disana yang tertulis adalah `02:30:15.125`. Kenapa menjadi jam 2? bukankah jam yang saya input adalah `9:30:15.125`? Perbedaan ini terjadi karena perintah `console.log()` menampilkan object Date dalam format waktu UTC.

UTC atau *Coordinated Universal Time* adalah standar waktu internasional. Secara umum, standar ini bisa disamakan dengan **GMT** atau *Greenwich Mean Time*. Seluruh waktu yang ada di dunia berpatokan kepada UTC dan **GMT**.

Berapakah jarak antara waktu UTC dengan waktu di Indonesia (**WIB**)? Yakni - 7 jam. Artinya, waktu di Indonesia lebih cepat 7 jam dari waktu UTC. Jika saat ini di Indonesia bagian barat (**WIB**) menunjukkan pukul 9:30 pagi, maka waktu UTC baru 2:30 dini hari.

Inilah yang menjadi alasan kenapa perintah `console.log()` menampilkan waktu `02:30:15.125`, padahal yang saya input adalah `9:30:15.125`, disini terdapat selisih waktu 7 jam.

Jadi bagaimana cara menampilkan tanggal sesuai dengan waktu WIB di Indonesia dan bukannya UTC? JavaScript menyediakan method khusus untuk keperluan ini, dan akan kita bahas sesaat lagi.

Kembali kepada constructor object **Date**, kita tidak harus menginput ketujuh argumen ini sekaligus, tapi bisa sebagian selama urutannya sesuai. Berikut kode program yang saya maksud:

```
1 var foo;  
2  
3 foo = new Date(2016);  
4 console.log( foo );      // Date 1970-01-01T00:00:02.016Z  
5  
6 foo = new Date(2016,11);  
7 console.log( foo );      // Date 2016-11-30T17:00:00.000Z  
8  
9 foo = new Date(2016,11,2);  
10 console.log( foo );     // Date 2016-12-01T17:00:00.000Z  
11  
12 foo = new Date(2016,11,2,9);  
13 console.log( foo );     // Date 2016-12-02T02:00:00.000Z  
14  
15 foo = new Date(2016,11,2,9,30);  
16 console.log( foo );     // Date 2016-12-02T02:30:00.000Z  
17  
18 foo = new Date(2016,11,2,9,30,15);  
19 console.log( foo );     // Date 2016-12-02T02:30:15.000Z
```

Disini juga terdapat beberapa hal yang agak aneh. Pertama, hasil dari `foo = new Date(2016)` adalah `Date 1970-01-01T00:00:02.016Z`. Artinya nilai 2016 masuk ke bagian milidetik, bukan tahun. Ini memang sudah bawaan JavaScript.

Kedua, `foo = new Date(2016,11)` hasilnya `Date 2016-11-30T17:00:00.000Z` Perhatikan bagian tanggal, yakni `2016-11-30`. Kenapa tanggalnya mundur? Ini terjadi karena perbedaan UTC dengan **WIB**, dimana tanggal `2016-12-01 00:00:00` mundur 7 jam menjadi `2016-11-30 17:00:00`.

Waktu yang mundur 7 jam ini juga tampak dari perintah berikutnya. Sekali lagi, ini terjadi karena perintah `console.log()` menampilkan **Date Object** menggunakan standar waktu UTC.

Cara kedua untuk membuat object **Date** adalah menggunakan perintah `new Date(dateString)`. Argumen `dateString` ini adalah sebuah string yang berbentuk tanggal. JavaScript mendukung berbagai string tanggal dan otomatis menkonversinya menjadi **Date**, seperti contoh berikut:

```
1 var foo;  
2  
3 foo = new Date("12/20/2016");  
4 console.log( foo );  
5 // Date 2016-12-19T17:00:00.000Z  
6  
7 foo = new Date("12/20/2016 9:30");  
8 console.log( foo );  
9 // Date 2016-12-20T02:30:00.000Z  
10  
11 foo = new Date("12 20 2016 9:30:15");  
12 console.log( foo );
```

```
13 // Date 2016-12-20T02:30:15.000Z
14
15 foo = new Date("20 Dec 2016 9:30:15");
16 console.log( foo );
17 // Date 2016-12-20T02:30:15.000Z
18
19 foo = new Date("2016-12-20");
20 console.log( foo );
21 // Date 2016-12-20T00:00:00.000Z
22
23 foo = new Date("December 20, 2016 9:30:15");
24 console.log( foo );
25 // Date 2016-12-20T02:30:15.000Z
```

Disini saya menginput setiap argumen dengan berbagai format penulisan string. Semuanya untuk tanggal 20-12-2016 9:30:15. Seperti yang bisa kita tebak, perintah `console.log()` menguranginya dengan 7 jam, hasilnya menjadi `Date 2016-12-20T02:30:15.000Z`.

Pertanyaannya adalah, string apa saja yang bisa dipahami oleh JavaScript? Secara teknis, format string ini mengikuti standar [IETF-compliant RFC 2822 timestamps<sup>1</sup>](#). Dimana beberapa format tersebut sama seperti yang saya gunakan diatas.

Saat membuat format `dateString`, anda juga harus hati-hati karena JavaScript bisa salah arti. Yakni apakah 12/1/2016 akan dianggap sebagai 12 Januari 2016, atau 1 Desember 2016. Untuk menghindari kejadian seperti ini, akan lebih aman jika kita menggunakan cara pembuatan `Date object` dengan 7 argumen (yang sudah kita pelajari sebelumnya) daripada menggunakan `dateString`.

Khusus untuk `dateString`, kita bisa menambahkan string **UTC** atau **GMT** di bagian akhir untuk menegaskan kepada JavaScript bahwa tanggal yang diinput adalah tanggal UTC, bukan waktu setempat (**WIB**):

```
1 var foo;
2
3 foo = new Date("12/20/2016 9:30:15");
4 console.log( foo );
5 // Date 2016-12-20T02:30:15.000Z
6
7 foo = new Date("12/20/2016 9:30:15 UTC");
8 console.log( foo );
9 // Date 2016-12-20T09:30:15.000Z
10
11 foo = new Date("12/20/2016 9:30:15 GMT");
12 console.log( foo );
13 // Date 2016-12-20T09:30:15.000Z
```

---

<sup>1</sup><https://tools.ietf.org/html/rfc2822#section-3.3>

Terlihat ketika saya menambahkan string UTC atau GMT, jam yang ditampilkan oleh perintah `console.log()` tetap 9:30:15. Tanpa tambahan string UTC atau GMT, waktu yang disimpan ke dalam object Date dikurangi 7 jam, menjadi 02:30:15.

Cara terakhir untuk membuat object **Date** adalah dengan menginput angka **milliseconds**, seperti contoh berikut:

```
1 var foo = new Date(1482201015000);
2 console.log( foo );      // Date 2016-12-20T02:30:15.000Z
```

Apa maksud dari angka 1482201015000? Ini merupakan total milidetik sejak tanggal **UNIX epoch time**<sup>2</sup>, yakni dari tanggal **1 January 1970**. Secara internal, nilai inilah yang disimpan oleh object **Date** JavaScript. Pada saat ditampilkan, nilai angka ini akan dikonversi menjadi string tanggal seperti tampilan diatas.

## 17.2 Method Getter UTC

**Method Getter UTC** adalah kelompok method yang menampilkan tanggal dalam format UTC. Diantaranya adalah sebagai berikut:

- `Date.prototype.toISOString()`
- `Date.prototype.toJSON()`
- `Date.prototype.toUTCString()`
- `Date.prototype.toDateString()`
- `Date.prototype.toTimeString()`
- `Date.prototype.valueOf()`
- `Date.prototype.getTime()`

Berikut contoh penggunaannya:

```
1 var foo = new Date();
2
3 console.log( foo.toISOString() );
4 // 2016-12-04T09:41:34.309Z
5
6 console.log( foo.toJSON() );
7 // 2016-12-04T09:41:34.309Z
8
9 console.log( foo.toUTCString() );
10 // Sun, 04 Dec 2016 09:41:34 GMT
11
12 console.log( foo.toDateString() );
```

---

<sup>2</sup>[https://en.wikipedia.org/wiki/Unix\\_time](https://en.wikipedia.org/wiki/Unix_time)

```
13 // Sun Dec 04 2016
14
15 console.log( foo.toTimeString() );
16 // 16:41:34 GMT+0700 (SE Asia Standard Time)
17
18 console.log( foo.valueOf() );
19 // 1460454094000
20
21 console.log( foo.getTime() );
22 // 1460454094000
```

Saya menjalankan kode program diatas pada tanggal 4 Desember 2016 pukul 16:41:34. Inilah isi dari **Date** object ke dalam variabel **foo**.

Method **toISOString()** digunakan untuk menampilkan tanggal dalam format **ISO**. Inilah format tanggal yang kita lihat sejak awal bab ketika menggunakan perintah **console.log()**. Tanggal yang ditampilkan menggunakan waktu **UTC**.

Method **toJSON()** menampilkan hasil yang sama dengan **toISOString()**. **JSON** adalah singkatan dari *JavaScript Object Notation*. **JSON** digunakan sebagai format standar pertukaran data antar website. Nantinya kita juga akan membahas tentang **JSON** dalam bab tersendiri.

Method **toUTCString()** menampilkan format tanggal **UTC** lengkap dengan nama hari (dalam bahasa inggris): Sun, 04 Dec 2016 09:41:34 GMT. Format ini lebih mudah dibaca daripada **ISO** maupun **JSON**.

Method **toDateString()** dan **toTimeString()** digunakan untuk menampilkan bagian tanggal saja atau waktu saja yang saling terpisah. Kedua method ini sebenarnya menampilkan waktu **locale**, bukan **UTC**. Karena itulah untuk **toTimeString()** hasilnya “16:41:34 GMT+0700 (SE Asia Standard Time)”.

Kenapa saya memasukkannya ke dalam kelompok **UTC** method? Karena nantinya ada juga method lain dengan nama sama untuk versi **locale**. Kedua method ini mendeteksi waktu yang ada di komputer saya adalah Indonesia, sehingga menambahkan string “**GMT+0700 (SE Asia Standard Time)**”.

Dua method terakhir, yakni **valueOf()** dan **getTime()** digunakan untuk menampilkan **Date** dalam bentuk milidetik sejak tanggal **UNIX Epoch**. Angka ini bisa digunakan untuk pemrosesan lebih lanjut, misalnya untuk mencari selisih tanggal atau diinput ke dalam argument dari perintah **new Date()**.

Selain menampilkan tanggal secara utuh, **Date** object juga memiliki method untuk menampilkan tanggal per bagian saja, misalnya bulan, tahun, tanggal, jam, dst. Berikut method yang tersedia:

- **Date.prototype.getUTCFullYear()**
- **Date.prototype.getUTCMonth()**
- **Date.prototype.getUTCDate()**
- **Date.prototype.getUTCDay()**
- **Date.prototype.getUTCHours()**

- `Date.prototype.getUTCMilliseconds()`
- `Date.prototype.getUTCSeconds()`
- `Date.prototype.getUTCMilliseconds()`

Berikut contoh penggunaannya:

```
1 var foo = new Date();
2 console.log( foo.getUTCFullYear() );           // 2016
3 console.log( foo.getUTCMonth() );              // 11
4 console.log( foo.getUTCDate() );               // 4
5 console.log( foo.getUTCDay() );                // 0
6 console.log( foo.getUTCHours() );              // 9
7 console.log( foo.getUTCMilliseconds() );        // 35
8 console.log( foo.getUTCSeconds() );             // 33
9 console.log( foo.getUTCMilliseconds() );        // 772
```

Method `getUTCFullYear()` digunakan untuk mengambil nilai tahun dari **Date object**. Tahun yang ditampilkan dalam format 4 digit.

Method `getUTCMonth()` digunakan untuk mengambil nilai bulan. Dan seperti yang pernah saya singgung sebelumnya, bulan disimpan dengan format angka 0 - 11. Bulan Januari = 0, Februari = 1, Maret = 2, dst hingga Desember = 11.

Method `getUTCDate()` digunakan untuk mengambil nilai tanggal. Dalam contoh diatas berarti tanggal 4. Format yang ditampilkan berupa nilai angka 1 - 31.

Method `getUTCDay()` digunakan untuk mengambil nilai hari. Angka 0 berarti hari minggu, angka 1 untuk senin, angka 2 selasa dst hingga angka 6 untuk hari sabtu.

Empat method selanjutnya: `getUTCHours()`, `getUTCMilliseconds()`, `getUTCSeconds()` dan `getUTCMilliseconds()` digunakan untuk mengambil nilai jam, menit, detik dan milidetik dari **Date object**.

Dengan pemisahan seperti ini, kita bisa merancang tampilan tanggal dan waktu yang sesuai dengan keinginan.

## 17.3 Method Getter Locale

**Method Getter Locale** adalah method dari **Date Object** yang digunakan untuk menampilkan tanggal dalam settingan **sistem lokal**. JavaScript mengetahui settingan ini dari web browser, dimana web browser mengambilnya dari Operating Sistem (yakni settingan tanggal dari **Windows**). Umumnya, tampilan seperti inilah yang akan kita pakai di website nanti.

Berikut method untuk menampilkan **Date object** ke dalam format lokal:

- `Date.prototype.toLocaleDateString()`
- `Date.prototype.toLocaleTimeString()`
- `Date.prototype.toLocaleString()`
- `Date.prototype.toString()`

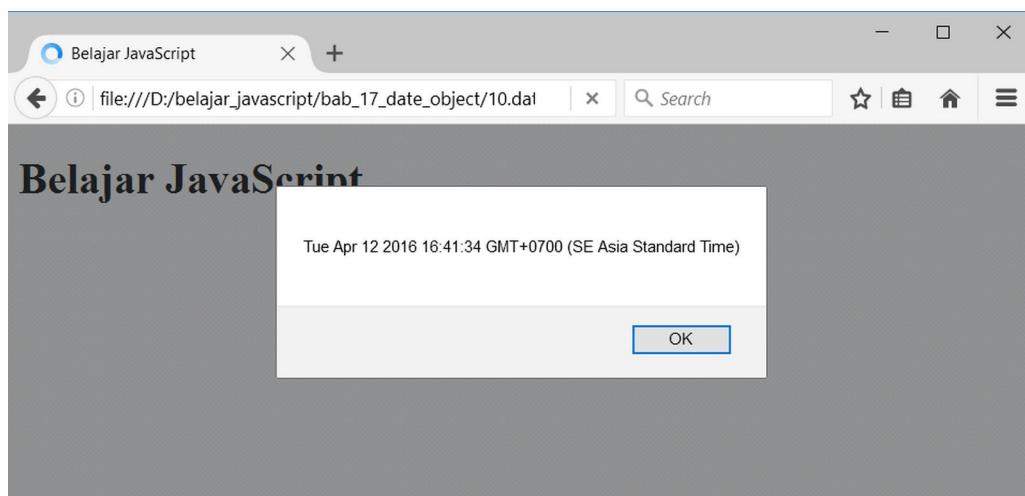
Berikut contoh penggunaannya:

```
1 var foo = new Date();
2
3 console.log( foo.toLocaleDateString() );
4 // 4/12/2016
5
6 console.log( foo.toLocaleTimeString() );
7 // 16.41.34
8
9 console.log( foo.toLocaleString() );
10 // 4/12/2016 16.41.34
11
12 console.log( foo.toString() );
13 // Sun Dec 04 2016 16:41:34 GMT+0700 (SE Asia Standard Time)
```

Perbedaan dari method `toLocaleDateString()` dan `toLocaleTimeString()` dengan method `toString()` dan `toLocaleString()` yang kita bahas pada bagian UTC sebelumnya, terletak di format tampilan. Untuk yang versi ‘**locale**’, tanggal tampil sesuai dengan settingan sistem, dimana berbentuk 4/12/2016. Bisa jadi tampilan formatnya berbeda tergantung settingan komputer anda.

Method `toString()` adalah tampilan default ketika tipe data **Date** di paksa tampil, misalnya jika dibuat ke dalam jendela `alert()`:

```
1 var foo = new Date();
2 alert( foo );
```



Gambar: xxxxxxxxx

Sama seperti versi UTC, Kita juga bisa mengambil bagian dari **Date object**, tapi untuk versi **locale**. Ini semua bisa diakses dari method-method berikut:

- `Date.prototype.getFullYear()`
- `Date.prototype.getMonth()`
- `Date.prototype.getDate()`

- Date.prototype.getDay()
- Date.prototype.getHours()
- Date.prototype.getMinutes()
- Date.prototype.getSeconds()
- Date.prototype.getMilliseconds()
- Date.prototype.getTimezoneOffset()

Berikut contoh penggunaannya:

```
1 var foo = new Date();
2 console.log( foo.getFullYear() );           // 2016
3 console.log( foo.getMonth() );             // 11
4 console.log( foo.getDate() );              // 4
5 console.log( foo.getDay() );               // 0
6 console.log( foo.getHours() );             // 16
7 console.log( foo.getMinutes() );            // 35
8 console.log( foo.getSeconds() );            // 33
9 console.log( foo.getMilliseconds() );       // 772
10 console.log( foo.getTimezoneOffset() );    // -420
```

Setiap method ini bersesuaian dengan versi UTC-nya, sehingga tidak perlu kita bahas lagi.

Sebagai tambahan, terdapat method `getTimezoneOffset()` yang akan menghasilkan selisih waktu antara UTC dengan waktu **local**. Hasilnya dalam satuan menit, sehingga  $-420/60 = -7$ . Artinya waktu UTC lebih lambat 7 jam dibandingkan sistem local (**WIB**).

## 17.4 Method Setter UTC

**Method Setter UTC** adalah method `Date` object yang digunakan untuk men-set tanggal berdasarkan waktu UTC. Jika method **getter** digunakan untuk mengambil data tanggal, method **setter** digunakan untuk menginput data tanggal.

Berikut method yang bisa digunakan untuk men-set `Date` object untuk tanggal UTC:

- Date.prototype.setUTCFullYear()
- Date.prototype.setUTCMonth()
- Date.prototype.setUTCDate()
- Date.prototype.setUTCHours()
- Date.prototype.setUTCMinutes()
- Date.prototype.setUTCSeconds()
- Date.prototype.setUTCMilliseconds()

Berikut contoh penggunaannya:

```
1 var foo = new Date(0);
2 console.log( foo.toUTCString() ); // Thu, 01 Jan 1970 00:00:00 GMT
3
4 foo.setUTCFullYear(2017);
5 console.log( foo.toUTCString() ); // Sun, 01 Jan 2017 00:00:00 GMT
6
7 foo.setUTCMonth(10);
8 console.log( foo.toUTCString() ); // Wed, 01 Nov 2017 00:00:00 GMT
9
10 foo.setUTCDate(20);
11 console.log( foo.toUTCString() ); // Mon, 20 Nov 2017 00:00:00 GMT
12
13 foo.setUTCHours(10);
14 console.log( foo.toUTCString() ); // Mon, 20 Nov 2017 10:00:00 GMT
15
16 foo.setUTCMilliseconds(30);
17 console.log( foo.toUTCString() ); // Mon, 20 Nov 2017 10:30:00 GMT
18
19 foo.setUTCSeconds(45);
20 console.log( foo.toUTCString() ); // Mon, 20 Nov 2017 10:30:45 GMT
21
22 foo.setUTCMilliseconds(125);
23 console.log( foo.toISOString() ); // 2017-11-20T10:30:45.125Z
```

Diawal kode program, saya membuat **Date** object dengan perintah `var foo = new Date(0)`. Artinya, variabel `foo` berisi tanggal **UNIX Epoch**, yakni 1 Januari 1970, pukul 00:00:00.

Setiap method yang ada akan mengubah tanggal ini. Ketika perintah `foo.setUTCFullYear(2017)` dijalankan, angka tahun dari variabel `foo` juga akan berubah dan tampil sebagai Sun, 01 Jan 2017 00:00:00 GMT.

Ketika perintah `foo.setUTCMonth(10)` dijalankan, angka bulan dari variabel `foo` juga akan berubah dan tampil sebagai Wed, 01 Nov 2017 00:00:00 GMT.

Begitu seterusnya hingga perintah `foo.setUTCMilliseconds(125)` yang akan mengubah bilangan milidetik dari variabel `foo` menjadi 125.

## 17.5 Method Setter Locale

Mungkin sudah bisa anda tebak bahwa jika ada versi UTC, maka akan tersedia juga method **setter** untuk versi **locale**. Berikut method yang tersedia:

- `Date.prototype.setFullYear()`
- `Date.prototype.setMonth()`
- `Date.prototype.setDate()`
- `Date.prototype.setHours()`

- `Date.prototype.setMinutes()`
- `Date.prototype.setSeconds()`
- `Date.prototype.setMilliseconds()`

Berikut contoh penggunaannya:

```
1 var foo = new Date(0);
2 console.log( foo.toLocaleString() ); // 1/1/1970 07.00.00
3
4 foo.setFullYear(2017);
5 console.log( foo.toLocaleString() ); // 1/1/2017 07.00.00
6
7 foo.setMonth(10);
8 console.log( foo.toLocaleString() ); // 1/11/2017 07.00.00
9
10 foo.setDate(20);
11 console.log( foo.toLocaleString() ); // 20/11/2017 07.00.00
12
13 foo.setHours(10);
14 console.log( foo.toLocaleString() ); // 20/11/2017 10.00.00
15
16 foo.setMinutes(30);
17 console.log( foo.toLocaleString() ); // 20/11/2017 10.30.00
18
19 foo.setSeconds(45);
20 console.log( foo.toLocaleString() ); // 20/11/2017 10.30.45
21
22 foo.setMilliseconds(125);
23 console.log( foo.toISOString() ); // 2017-11-20T03:30:45.125Z
```

Sama seperti versi UTC, saya juga membuat `Date` object menggunakan perintah `var foo = new Date(0)`. Karena kali ini saya menggunakan perintah `foo.toLocaleString()` untuk menampilkan isi variabel `foo`, akan terdapat tambahan 7 jam dari UNIX Epoch.

## 17.6 Latihan Program untuk Date Object

Setelah mempelajari berbagai method dari `Date` object, mari kita latihan kode program yang sekaligus sebagai ajang refreshing dari materi bab-bab sebelumnya.

### Menampilkan Tanggal dengan Format Tertentu

Latihan pertama, bagaimana kalau menampilkan tanggal seperti gambar dibawah ini:



Gambar: Tampilan tanggal dengan bahasa Indonesia

Disini saya menampilkan tanggal dengan format: **Selasa, 6 Desember 2016 16:50:15**. Ini adalah tanggal dimana saya menjalankan kode program tersebut. Artinya, menampilkan tanggal hari ini.

Selain itu, hasil diatas bukan pengaruh settingan sistem atau *locale*. Disini saya menggunakan beberapa kondisi **switch-case** untuk mengkonversi angka dari object **Date** menjadi nama hari dan bulan dalam bahasa Indonesia.

Baik, mari kita mulai pembahasannya. Bagaimana cara menampilkan nama hari dalam bahasa Indonesia? Yakni kata-kata "Senin", "Selasa", "Rabu", dst?

Jika anda masih ingat, di dalam **Date** object terdapat method **getDay()**. Hasilnya berupa angka hari dimana 0 untuk hari minggu, 1 untuk senin, 2 untuk selasa, dst. Data ini bisa kita ambil untuk diproses menjadi nama hari dalam bahasa Indonesia.

Caranya, kita bisa menggunakan struktur **if else**. Yakni jika 0, maka **var namaHari = "Minggu"**, jika 1 maka **var namaHari = "Selasa"**, dst. Setidaknya kita butuh 7 kali struktur **if else** untuk membuat kondisi 7 nama hari. Atau karena kondisi yang diperiksa cukup sederhana, bisa menggunakan struktur **switch-case**:

```
1 var foo = new Date();
2 var hari = foo.getDay();
3 var namaHari;
4
5 switch (hari) {
6     case 0 : var namaHari = "Minggu"; break;
7     case 1 : var namaHari = "Senin"; break;
8     case 2 : var namaHari = "Selasa"; break;
9     case 3 : var namaHari = "Rabu"; break;
10    case 4 : var namaHari = "Kamis"; break;
11    case 5 : var namaHari = "Jumat"; break;
12    case 6 : var namaHari = "Sabtu"; break;
13 }
14
15 console.log(namaHari); // Selasa
```

Jika anda lupa-lupa ingat mengenai kode **switch** ini, silahkan pelajari sejenak di dalam bab tentang Struktur Logika.

Hasil dari perintah `foo.getDay()` diproses menggunakan kondisi **switch**, sehingga variabel `namaHari` akan berisi nama hari dalam Bahasa Indonesia.

Bagaimana dengan nama bulan? Yup, caranya juga sama:

```
1 var foo = new Date();
2 var bulan = foo.getMonth();
3 var namaBulan;
4
5 switch (bulan) {
6     case 0 : var namaBulan = "Januari"; break;
7     case 1 : var namaBulan = "Februari"; break;
8     case 2 : var namaBulan = "Maret"; break;
9     case 3 : var namaBulan = "April"; break;
10    case 4 : var namaBulan = "Mei"; break;
11    case 5 : var namaBulan = "Juni"; break;
12    case 6 : var namaBulan = "Juli"; break;
13    case 7 : var namaBulan = "Agustus"; break;
14    case 8 : var namaBulan = "September"; break;
15    case 9 : var namaBulan = "Oktober"; break;
16    case 10 : var namaBulan = "November"; break;
17    case 11 : var namaBulan = "Desember"; break;
18 }
19
20 console.log(namaBulan); // namaBulan
```

Perintah `var bulan = foo.getMonth()` akan mengembalikan digit bulan dengan nilai 0-11. Menggunakan struktur **switch**, variabel `namaBulan` akan berisi nama bulan dalam bahasa Indonesia.

Sampai disini, kita sudah bisa menampilkan nama hari dan nama bulan. Sisa tampilan bisa diambil menggunakan beberapa method bawaan **Date** object. Berikut kode program lengkapnya:

```
1 var foo = new Date();
2
3 var tahun = foo.getFullYear();
4 var bulan = foo.getMonth();
5 var tanggal = foo.getDate();
6 var hari = foo.getDay();
7 var jam = foo.getHours();
8 var menit = foo.getMinutes();
9 var detik = foo.getSeconds();
10
11 var namaHari;
```

```

12 var namaBulan;
13
14 switch (hari) {
15     case 0 : var namaHari = "Minggu"; break;
16     case 1 : var namaHari = "Senin"; break;
17     case 2 : var namaHari = "Selasa"; break;
18     case 3 : var namaHari = "Rabu"; break;
19     case 4 : var namaHari = "Kamis"; break;
20     case 5 : var namaHari = "Jumat"; break;
21     case 6 : var namaHari = "Sabtu"; break;
22 }
23
24 switch (bulan) {
25     case 0 : var namaBulan = "Januari"; break;
26     case 1 : var namaBulan = "Februari"; break;
27     case 2 : var namaBulan = "Maret"; break;
28     case 3 : var namaBulan = "April"; break;
29     case 4 : var namaBulan = "Mei"; break;
30     case 5 : var namaBulan = "Juni"; break;
31     case 6 : var namaBulan = "Juli"; break;
32     case 7 : var namaBulan = "Agustus"; break;
33     case 8 : var namaBulan = "September"; break;
34     case 9 : var namaBulan = "Oktober"; break;
35     case 10 : var namaBulan = "November"; break;
36     case 11 : var namaBulan = "Desember"; break;
37 }
38
39 var tampilanTanggal = namaHari+", "+tanggal+" "+namaBulan+" "+tahun;
40     tampilanTanggal += " " + jam + ":" + menit+ ":" + detik;
41
42 console.log(tampilanTanggal); // Selasa, 6 Desember 2016 16:50:15

```

Kode program diatas sedikit panjang karena saya perlu membuat kondisi `switch` untuk menampilkan nama hari dan nama bulan. Kemudian mengabungkan string dari hasil beberapa method Date Object.

Sebagai latihan tambahan, anda bisa memodifikasi kode program diatas untuk tampilan: **Selasa, Enam Desember, Dua Ribu Enam Belas, 16:50:15**. Hanya saja untuk tanggal anda harus membuat kondisi `switch` dari 0 hingga 31. Sedangkan untuk tahun bisa dibatasi dari tahun 2010 hingga 2020 saja.

## Menghitung Selisih Tanggal

Salah satu operasi yang paling sering dipakai untuk tipe data `Date` adalah menghitung selisih waktu dari 2 tanggal. Untuk membuat kode program seperti ini, cari selisih nilai Unix Epoch dari kedua tanggal, lalu format sesuai keinginan. Mari kita rancang kode programnya.

Langkah pertama, siapkan 2 buah variabel yang berisi tanggal berbeda:

```
1 var tanggalAwal = new Date("06/05/2016");
2 var tanggalAkhir = new Date("12/20/2016");
```

Disini saya membuat 2 variabel: tanggalAwal dan tanggalAkhir. Masing-masing diisi dengan tanggal **05 Juni 2016** dan **20 Desember 2016**. Perhatikan bahwa urutan tanggal dan bulan terbalik. Ini merupakan salah satu kelemahan jika membuat tanggal menggunakan format **dateString**. Perintah `new Date("06/05/2016")` digunakan untuk membuat tanggal 05 Juni 2016, **bukan** 06 Mei 2016.

Untuk mencari selisih tanggal, kurangkan kedua variabel ini:

```
1 var tanggalAwal = new Date("06/05/2016");
2 var tanggalAkhir = new Date("12/20/2016");
3
4 var selisihTanggal = tanggalAkhir - tanggalAwal;
5 console.log( selisihTanggal ); // 17107200000
```

Pertanyaannya, dari mana angka **17107200000** muncul? Angka tersebut merupakan selisih waktu dalam satuan milidetik dari `tanggalAkhir - tanggalAwal`. Proses yang sebenarnya terjadi adalah sebagai berikut:

```
1 var tanggalAwal = new Date("06/05/2016");
2 var tanggalAkhir = new Date("12/20/2016");
3
4 var timeAwal = tanggalAwal.getTime();
5 var timeAkhir = tanggalAkhir.getTime();
6
7 console.log( timeAwal ); // 1465059600000
8 console.log( timeAkhir ); // 1482166800000
9
10 var selisihTanggal = timeAkhir - timeAwal;
11 console.log( selisihTanggal ); // 17107200000
```

Method `getTime()` digunakan untuk mengambil nilai **Unix Epoch** dari object `Date`. Artinya angka **17107200000** adalah selisih antara `tanggalAkhir` dengan `tanggalAwal` dalam satuan milidetik.

Bagaimana mengubahnya menjadi hari? Tinggal dibagi dengan banyak milidetik dalam 1 hari:

```
1 var tanggalAwal = new Date("06/05/2016");
2 var tanggalAkhir = new Date("12/20/2016");
3
4 var selisihTanggal = tanggalAkhir - tanggalAwal;
5 var satuHari = 1000*60*60*24;
6 var selisihHari = selisihTanggal/satuHari;
7
8 console.log("Selisih tanggal = " + selisihHari + " Hari");
9 // Selisih tanggal = 198 Hari
```

Hasilnya, selisih dari tanggal **12/20/2016** dengan **06/05/2016** adalah **198 hari**. Tantangan selanjutnya, bagaimana mengkonversi **198 hari** ini menjadi sekian tahun, sekian bulan dan sekian hari?

Untuk mendapatkan hasil tersebut, kita harus menggunakan sedikit analisis matematika sederhana, yakni mencari apakah hasilnya cukup dikurangi 365 (untuk menjadi 1 tahun) dan sisanya dibagi 30 (untuk menjadi 1 bulan).

Sebagai contoh, **500 hari** itu berapa tahun, berapa bulan dan berapa hari?

Pertama, bagi 500 dengan 365,  $500/365 = 1.37$ . Artinya 500 hari sama dengan 1.37 tahun. Simpan angka 1 tahun, dan kita akan konversi kelebihan 0.37 tahun menjadi bulan dan hari.

Karena 500 hari terdiri dari 1 tahun lebih, sisa hari bisa didapat dengan rumus  $500 - (1 * 365) = 135$ . Dengan mengasumsikan 1 bulan = 30 hari, maka  $135/30 = 4.5$  bulan. Simpan angka 4 bulan, dan kita akan konversi kelebihan 0.5 bulan ini menjadi hari.

Caranya juga sama. Sisa hari didapat dari pengurangan jumlah tahun dan jumlah bulan. Ini bisa dicari dengan rumus  $500 - (1*365) - (4*30) = 15$  hari. Akhirnya didapat bahwa **500 hari = 1 tahun, 4 bulan, 15 hari**.

Sekarang, bagaimana dengan 198 hari? Mari kita hitung.

- Jumlah tahun =  $198/365 = 0.54$ . Artinya tidak cukup 1 tahun. Simpan 0.
- Jumlah bulan =  $198 - (0*365) / 30 = 6.6$ . Artinya terdapat 6 bulan lebih. Simpan 6, dan kita akan cari berapa hari lebihnya.
- Jumlah hari =  $198 - (0*365) - (6*30) = 18$ .

Didapat bahwa 198 hari terdiri dari 0 tahun 6 bulan dan 18 hari.

Menerjemahkan satu masalah menjadi langkah-langkah spesifik seperti ini merupakan inti dari **Algoritma**. Analisis yang baru saja kita lakukan adalah sebuah proses pembuatan **Algoritma**, dimana kita mencari cara untuk mengkonversi jumlah hari menjadi tahun dan bulan. Kemampuan analisis algoritma akan semakin berkembang dengan banyaknya latihan soal dan membuat kode program.

Baik, mari kita terjemahkan langkah-langkah diatas ke dalam kode program JavaScript:

```
1 var tanggalAwal = new Date("06/05/2016");
2 var tanggalAkhir = new Date("12/20/2016");
3
4 var selisihTanggal = Math.abs(tanggalAkhir - tanggalAwal);
5
6 var satuHari = 1000*60*60*24;
7 var satuBulan = 1000*60*60*24*30;
8 var satuTahun = 1000*60*60*24*365;
9
10 var selisihTahun = Math.floor(selisihTanggal / satuTahun);
11 var selisihBulan = Math.floor((selisihTanggal - (selisihTahun * satuTahun)) /
12                               satuBulan);
13 var selisihHari = Math.floor((selisihTanggal - (selisihTahun * satuTahun) -
14                               (selisihBulan * satuBulan)) /satuHari);
15
16 var hasil = selisihTahun+" Tahun "+selisihBulan+" Bulan "+
17           selisihHari+" Hari";
18
19 console.log( hasil ); // 0 Tahun 6 Bulan 18 Hari
```

Terdapat beberapa tambahan method. Saya menggunakan method `Math.abs()` untuk mendapatkan angka mutlak dari selisih tanggal. Kita tidak peduli apakah hasilnya negatif atau positif, yang penting adalah jumlah selisih tanggal.

Saya juga menggunakan method `Math.floor()` untuk menghapus nilai pecahan di belakang koma. Ini dipakai untuk menghasilkan angka bulat pada nilai tahun, bulan dan hari (dibulatkan ke bawah).

Mari kita uji dengan angka lain:

```
1 var tanggalAwal = new Date("08/17/1945");
2 var tanggalAkhir = new Date(); // 07 Desember 2016
3
4 var selisihTanggal = Math.abs(tanggalAkhir - tanggalAwal);
5
6 var satuHari = 1000*60*60*24;
7 var satuBulan = 1000*60*60*24*30;
8 var satuTahun = 1000*60*60*24*365;
9
10 var selisihTahun = Math.floor(selisihTanggal / satuTahun);
11 var selisihBulan = Math.floor((selisihTanggal - (selisihTahun * satuTahun)) /
12                               satuBulan);
13 var selisihHari = Math.floor((selisihTanggal - (selisihTahun * satuTahun) -
14                               (selisihBulan * satuBulan)) /satuHari);
15
16 var hasil = selisihTahun+" Tahun "+selisihBulan+" Bulan "+
17           selisihHari+" Hari";
```

```
18  
19 console.log( hasil ); // 71 Tahun 4 Bulan 10 Hari
```

Saya menghitung selisih tanggal hari ini dengan **17 Agustus 1945** (saya jalankan pada **07 Desember 2016**). Hasilnya: **71 Tahun 4 Bulan 10 Hari**.

Sekarang kita sudah punya program penghitung selisih tanggal. Namun karena kompleksitasnya, kode program diatas tidak memperhitungkan tahun kabisat dan berbedaan hari dalam tiap bulan. Tapi setidaknya hasil yang di dapat mendekati angka sebenarnya.

Silahkan anda coba kembangkan kode program diatas, misalnya memisahkannya menjadi function tersendiri (seperti dalam buku **PHP Uncover**). Atau bisa juga membuat kode program lain untuk melatih kemampuan algoritma dan pemecahan masalah.

Sebagai contoh, bagaimana kalau membuat kode program yang membagi nilai rupiah. Misalnya jika inputan awal **257,500**. Hasil kode program berupa: **2 lembar uang 100.000, 1 lembar uang 50.000, 1 lembar uang 5000, 1 lembar uang 2000, 1 lembar uang 500**. Algoritmanya mirip-mirip seperti yang kita gunakan untuk membagi tanggal, tapi sekarang yang dibagi adalah uang.

---

Dalam bab ini kita telah membahas tentang object **Date** JavaScript, sekaligus latihan kode program untuk menampilkan dan mencari selisih dari 2 tanggal. Jika anda butuh pemrosesan tanggal yang lebih presisi, bisa mempelajari library JavaScript khusus seperti **MomentJS**<sup>3</sup>.

---

<sup>3</sup><http://momentjs.com>

# 18. Global Property dan Global Function

Global property dan global function adalah property dan function JavaScript yang tidak “melekat” ke object apapun, dan bisa diakses dari mana saja. Anggota dari global property dan global function sendiri tidak banyak karena mayoritas function bawaan JavaScript di simpan ke dalam Object, seperti yang kita bahas dalam beberapa bab sebelum ini.

## 18.1 Global Property

Global property bisa dikatakan sebagai konstanta spesial JavaScript, dimana ia berfungsi untuk menyimpan nilai khusus, yakni:

- `Infinity`
- `NAN`
- `undefined`
- `null`

Nilai-nilai ini sudah kita pelajari sebelumnya. Sebagai contoh, untuk membuat variabel yang berisi NaN, bisa ditulis sebagai berikut:

```
1 var foo = NaN;  
2 console.log(foo); // NaN
```

Begitu juga untuk nilai-nilai lain:

```
1 var foo = Infinity;  
2 console.log(foo); // Infinity  
3  
4 var bar = undefined;  
5 console.log(bar); // undefined  
6  
7 var baz = null;  
8 console.log(baz); // null
```

Yang juga perlu diingat, keempat property ini bersifat **case-sensitif**, dimana huruf kecil dan besar dianggap berbeda (sebagaimana layaknya property dan method JavaScript lain). Kode program berikut akan menghasilkan error:

```
1 var foo = Infinity;  
2 console.log(foo);      // ReferenceError: infinity is not defined  
3  
4 var bar = NAN;  
5 console.log(bar);      // ReferenceError: NAN is not defined  
6  
7 var baz = Undefined;  
8 console.log(baz);      // ReferenceError: Undefined is not defined  
9  
10 var qux = Null;  
11 console.log(qux);     // ReferenceError: Null is not defined
```

Khusus untuk NaN dan Infinity, keduanya juga tersedia sebagai property untuk object Number: Number.NaN, Number.NEGATIVE\_INFINITY, dan Number.POSITIVE\_INFINITY.

## 18.2 Global Function

Global Function adalah function JavaScript yang bisa diakses dari mana saja dan tidak terikat dengan object apapun. Berikut daftar function tersebut:

- eval()
- isFinite()
- isNaN()
- parseFloat()
- parseInt()
- decodeURI()
- decodeURIComponent()
- encodeURI()
- encodeURIComponent()

Mari kita bahas.

### 18.3 Function eval()

Function eval() digunakan untuk memproses string menjadi perintah JavaScript. Fungsi ini membuat string menjadi bagian dari kode program, termasuk mengakses variabel dan function JavaScript. Berikut contoh penggunaannya:

```

1 var foo = "100 + 30";
2 console.log( foo ); // 100 + 30
3
4 var bar = eval(foo);
5 console.log( bar ); // 130

```

Saya mendefenisikan variabel `foo` berisi string "`100 + 30`". Ini merupakan string, bukan kode program. Tapi jika diinput kedalam fungsi `eval()`, string akan dieksekusi dan menghasilkan nilai `130` (di dapat dari `100+30`).

Termasuk apabila di dalam string tersebut di defenisikan sebuah variabel:

```

1 var foo = "var bar = 500 * 3";
2 eval(foo);
3 console.log(bar); // 1500

```

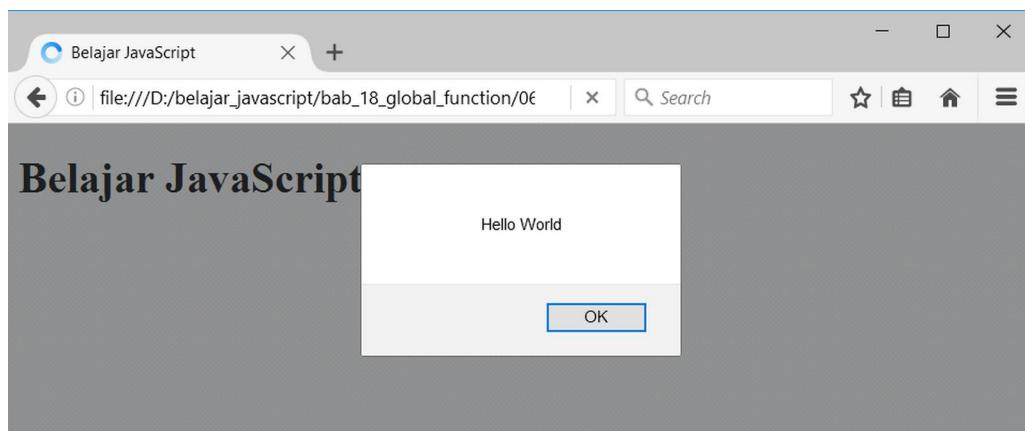
Jika kita melihat kode program, tidak terdapat pendefenisian variabel `bar`. Variabel `bar` ada di dalam string "`var bar = 500 * 3`". Menggunakan perintah `eval( foo )`, string ini dieksekusi dan variabel `bar` akan berisi nilai `1500`.

Bagaimana kalau function atau method?

```

1 var foo = "alert('Hello World')";
2 eval(foo);

```



Gambar: Jendela alert tampil karena perintah `eval()`

Hasilnya akan tampil jendela `alert('Hello World')` yang sebenarnya berupa string di dalam variabel `foo`. Perintah `eval( foo )` 'menghidupkan' string ini menjadi perintah JavaScript.

Melihat efeknya yang sangat powerful, fungsi `eval()` juga **sangat berbahaya**. Fungsi `eval()` banyak digunakan untuk mengeksekusi kode program berbahaya yang diinput ke dalam sebuah website (*Cross-site Scripting*).

Biasanya fungsi `eval()` dipakai untuk menjalankan kode JavaScript yang berasal dari situs lain, seperti penggunaan **API** (*Application Programming Interface*). Sedapat mungkin hindari penggunaan fungsi `eval()` jika tidak sangat terpaksa.

## 18.4 Function isFinite()

Function `isFinite()` sebenarnya sama persis seperti method `Number.isFinite()`. Fungsi ini digunakan untuk mengecek apakah sebuah nilai/variabel berisi angka yang bisa dihitung. Fungsi `isFinite()` mengembalikan boolean `true` jika nilai tersebut bisa dihitung (berupa angka biasa), dan mengembalikan nilai `false` jika berupa `infinity`, `NaN`, atau bukan tipe data `Number`.

Berikut contoh penggunaannya:

```
1 var foo;  
2  
3 foo = 6 ;  
4 console.log( isFinite(foo) ); // true  
5  
6 foo = 3.21456 ;  
7 console.log( isFinite(foo) ); // true  
8  
9 foo = 'a';  
10 console.log( isFinite(foo) ); // false  
11  
12 foo = 1/0 ;  
13 console.log( isFinite(foo) ); // false  
14  
15 foo = Number.NEGATIVE_INFINITY ;  
16 console.log( isFinite(foo) ); // false  
17  
18 foo = NaN;  
19 console.log( isFinite(foo) ); // false
```

Contoh ini sama seperti yang saya jalankan untuk method `Number.isFinite()`.

## 18.5 Function isNaN()

Function `isNaN()` juga merupakan penulisan singkat dari method `Number.isNaN()`. Fungsi ini digunakan untuk mengecek apakah hasil operasi / suatu variabel berisi `NaN` atau bukan. Hasilnya `true` jika itu `NaN`, dan `false` jika bukan `NaN`:

```
1 var foo;  
2  
3 foo = 5 ;  
4 console.log( isNaN(foo) ); // false  
5  
6 foo = 5/'a' ;  
7 console.log( isNaN(foo) ); // true  
8  
9 foo = NaN ;  
10 console.log( isNaN(foo) ); // true
```

## 18.6 Function parseInt()

Function `parseInt()` sama fungsinya dengan method `Number.parseInt()`, yakni digunakan untuk mengkonversi nilai atau variabel menjadi angka integer (angka bulat). Jika di dalam string asal terdapat nilai pecahan, bagian pecahan akan dibuang. Jika tidak bisa dikonversi menjadi number, method ini mengembalikan nilai `NaN`.

Sama seperti method `Number.parseInt()` fungsi `parseInt()` juga memiliki argumen kedua yang bersifat opsional. Argumen kedua ini bisa diisi dengan `radix`, yakni basis bilangan.

Berikut contoh penggunaannya:

```
1 var foo = "1234.567";  
2 console.log( parseInt(foo) ); // 1234  
3 console.log( parseInt(99.99) ); // 99  
4  
5 console.log( parseInt("1100011",2) ); // 99  
6 console.log( parseInt("143",8) ); // 99  
7 console.log( parseInt("63",16) ); // 99
```

## 18.7 Function parseFloat()

Function `parseFloat()` adalah penulisan singkat dari method `Number.parseFloat()`. Fungsi ini dipakai untuk mengkonversi nilai atau variabel ke bentuk angka float. Float adalah angka yang memiliki nilai pecahan, serta juga termasuk angka bulat (integer). Jika string tersebut tidak bisa dikonversi menjadi number, method ini akan mengembalikan nilai `NaN`.

Berikut contoh penggunaannya:

```

1 var foo = "1234";
2 console.log( typeof foo );      // string
3
4 foo = parseFloat(foo) ;
5 console.log( foo );           // 1234
6 console.log( typeof foo );     // number
7
8 var bar = "-1234.5678";
9 console.log( typeof bar );     // string
10 bar = parseFloat(bar) ;
11
12 console.log( bar );          // -1234.5678
13 console.log( typeof bar );    // number
14
15 console.log( parseFloat("12.045 potong ayam") ); // 12.045
16 console.log( parseFloat("Ada 12.045 potong ayam") ); // NaN

```

## 18.8 Function encodeURI() dan encodeURIComponent()

Kedua fungsi ini digunakan untuk meng-*encode* atau mengkodekan beberapa karakter khusus yang biasanya ada di alamat URI.

**URI** (*Uniform Resource Identifier*) merupakan istilah yang lebih luas untuk menyebut **URL** (*Uniform Resource Locator*). URL sendiri merupakan alamat internet yang sering kita jumpai, seperti <https://www.google.com>, <http://www.duniaIlkom.com>, atau [https://en.wikipedia.org/wiki/Uniform\\_Resource\\_Locator](https://en.wikipedia.org/wiki/Uniform_Resource_Locator).

Berikut contoh penggunaan dari function encodeURI() dan encodeURIComponent():

```

1 var foo = "http://www.duniaIlkom.com/Belajar #JavaScript";
2 var bar = encodeURI(foo);
3 var baz = encodeURIComponent(foo);
4
5 console.log(bar);
6 // "http://www.duniaIlkom.com/Belajar%20#JavaScript"
7 console.log(baz);
8 // http%3A%2F%2Fwww.duniaIlkom.com%2FBelajar%20%23JavaScript

```

Disini saya memiliki variabel foo yang diisi dengan string `http://www.duniaIlkom.com/Belajar #JavaScript`. Perhatikan bahwa alamat ini memiliki sebuah spasi. Spasi ini dikodekan menjadi `%20` oleh fungsi `encodeURI()`. Sedangkan untuk fungsi `encodeURIComponent()` mengkodekan lebih banyak karakter, termasuk `,` `/` dan `#`.

## 18.9 Function decodeURI() dan decodeURIComponent()

Function decodeURI() dan decodeURIComponent() merupakan kebalikan dari fungsi encodeURI() dan encodeURIComponent(). Fungsi decodeURI() dan decodeURIComponent() digunakan untuk membalik string yang telah dikodekan oleh fungsi encodeURI() dan encodeURIComponent().

Berikut contoh penggunaannya:

```
1 var foo = "http://www.duniaIlkom.com/Belajar #JavaScript";
2 var bar = encodeURI(foo);
3 var baz = encodeURIComponent(foo);
4
5 console.log(bar);
6 // "http://www.duniaIlkom.com/Belajar%20#JavaScript"
7 console.log(baz);
8 // http%3A%2F%2Fwww.duniaIlkom.com%2FBelajar%20%23JavaScript
9
10 console.log( decodeURI(bar) );
11 // "http://www.duniaIlkom.com/Belajar #JavaScript"
12 console.log( decodeURIComponent(bar) );
13 // "http://www.duniaIlkom.com/Belajar #JavaScript"
14
15 console.log( decodeURI(baz) );
16 // http%3A%2F%2Fwww.duniaIlkom.com%2FBelajar %23JavaScript
17 console.log( decodeURIComponent(baz) );
18 // "http://www.duniaIlkom.com/Belajar #JavaScript"
```

Disini, variabel `bar` dan `baz` hasil `encodeURI(foo)` dan `encodeURIComponent(foo)` diproses kembali dengan perintah `decodeURI(bar)`, `decodeURIComponent(bar)`, `decodeURI(baz)` dan `decodeURIComponent(baz)`.

Terlihat bahwa fungsi ini mengembalikan karakter `%20` menjadi spasi, `%3A` menjadi `:` dan `%2F` menjadi `/`.

Secara umum, fungsi `encodeURI()`, `encodeURIComponent(foo)`, `decodeURI()` dan `decodeURIComponent(baz)` tidak terlalu sering dipakai.

---

Dalam bab ini kita membahas sedikit property dan function global di dalam JavaScript. Seperti yang terlihat, mayoritas property dan function ini merupakan penulisan singkat dari property dan method yang tersimpan di **Number Object**.

Berikutnya kita akan membahas tentang **DOM**, singkatan dari **Document Object Model**.

# 19. DOM (Document Object Model)

Selamat! Saya ucapkan kepada anda yang masih tetap bertahan membaca buku *JavaScript Uncover* hingga bab ini. Tidak dapat dipungkiri bahwa materi yang saya bahas dari awal buku hingga bab 19 cukup membosankan. Apalagi kita belum bisa melakukan apa-apa selain menggunakan perintah `console.log()` untuk menampilkan hasil kode JavaScript.

Kabar baiknya, penderitaan sudah berakhir. Mulai dari bab ini dan selanjutnya, kita akan mempraktekkan seluruh dasar-dasar JavaScript ke web browser. Disinilah pemahaman tentang HTML (dan sedikit CSS) sangat dibutuhkan. Kita akan menggunakan JavaScript untuk memanipulasi element HTML.

## 19.1 Materi Tentang JavaScript Sudah Selesai!

Saya juga bisa mengatakan bahwa bab tentang *Global Property* dan *Global Function* sebelum ini, menutup materi kita tentang JavaScript. Atau dengan kata lain, **kita sudah selesai membahas JavaScript**.

*Loh kenapa? Apakah JavaScript hanya “itu” saja? Lalu sisa buku ini membahas tentang apa?*

JavaScript atau lebih tepatnya **ECMAScript**, adalah sebuah bahasa pemrograman “generic” yang bisa digunakan untuk apa saja, tidak hanya web programming. Tapi memang JavaScript lahir dan besar sebagai sebuah bahasa pemrograman web client side.

Untuk dapat membuat program yang memproses halaman web, kita membutuhkan materi tambahan yang sebenarnya terpisah dengan JavaScript, yakni **DOM (Document Object Model)**. DOM-lah yang akan menyediakan tombol, warna, teks, gambar, serta form HTML yang bisa kita manipulasi dengan JavaScript.

Sebagai analogi, pernahkan anda membeli peralatan yang harus dirakit dulu? Kita ambil contoh lemari. Untuk menekan harga, beberapa mall menjual lemari kayu sederhana yang mesti dirakit dulu (tidak menjual lemari yang sudah jadi).

Di dalam paket penjualan, terdapat kayu, triplek, baut, paku, dan komponen lain penyusun lemari. Selain itu disertakan kertas petunjuk mengenai cara merakitnya. Agar mudah dimengerti, petunjuk ini dibuat dalam bahasa indonesia, tapi tidak jarang juga menggunakan bahasa inggris (untuk lemari import).

**Bahasa yang digunakan** untuk menulis instruksi ini bisa di samakan dengan **JavaScript**. Sedangkan **komponen lemari** adalah **DOM**.

Untuk dapat membuat lemari yang utuh, kita mengikuti instruksi yang ada dalam kertas petunjuk. Begitu juga dengan **JavaScript**, ia digunakan untuk merakit komponen **DOM** agar menjadi halaman web dinamis.

Efeknya, bahasa petunjuk yang digunakan untuk merakit lemari bisa saja ditulis dalam bahasa indonesia, bahasa inggris, atau bahasa spanyol tergantung lokasi penjualan lemari.

Begitu juga untuk halaman web. Pemrosesan DOM **tidak harus menggunakan JavaScript**, tapi bisa dengan bahasa pemrograman lain. Hanya saja JavaScript memang sangat dominan, dan nyaris tidak tersaingi dengan bahasa pemrograman lain. Dulunya terdapat bahasa **JScript**, **VBScript**, dan **Flash** yang juga bisa memproses DOM, tapi sudah sangat jarang dipakai.



Tidak menutup kemungkinan di masa depan ada bahasa pemrograman lain yang hadir sebagai pengganti JavaScript. Saat ini terdapat proyek **asm.js**, **Google Native Client (NaCl)**, dan **Web Assembly**. Ketiganya digunakan untuk pemrograman client side, namun tidak menargetkan DOM, tapi pemrograman yang lebih rumit seperti game programming.

Jika anda tertarik, bisa dilihat demo game yang dibuat menggunakan **Web Assembly: Angry Bots Demo**<sup>1</sup>. Hanya saja kita harus menggunakan web browser khusus atau klik “*play asm.js fallback*” untuk versi **asm.js**.

Begitu pula sebaliknya, bahasa indonesia yang digunakan dalam menulis instruksi tidak hanya dipakai untuk membuat lemari saja, tapi juga bisa untuk membuat meja, kursi, dan berbagai perangkat lain (dengan menggunakan bahan yang berbeda).

JavaScript tidak hanya digunakan untuk memproses DOM, tapi juga bisa untuk pemrograman server side (**Node.js**). Bahkan baru-baru ini terdapat proyek yang menjadikan JavaScript sebagai bahasa pemrograman untuk **IoT** (Internet of Things), yakni perangkat pintar untuk alat sehari-hari seperti jam tangan, sepatu, gelang, yang semuanya terhubung ke internet: [Why JavaScript and the Internet of Things?](#)<sup>2</sup>.

Untuk mempertegas perbedaan antara JavaScript dengan DOM, keduanya dikembangkan oleh tim yang berbeda. JavaScript atau ECMAScript dirilis oleh **ECMA**, sedangkan DOM dirilis oleh **W3C**. Jika JavaScript memiliki versi ECMAScript 5, 6, dan 7. DOM juga memiliki versi, yakni DOM Level 1, Level 2, dan Level 3. Kita akan bahas hal ini dengan lebih dalam nantinya.

## 19.2 Pengertian DOM (Document Object Model)

DOM (Document Object Model) adalah representasi kode HTML ke dalam bentuk object agar bisa diproses oleh bahasa pemrograman, seperti JavaScript.

Jika dihubungkan dengan kepanjangannya, DOM merupakan **pemodelan dokumen HTML ke dalam bentuk object**. Artinya, setiap tag-tag HTML seperti `<h1>`, `<p>`, atau `<form>` dimodelkan atau dibentuk menjadi sebuah **object**.

Sebagaimana layaknya **object**, tag-tag HTML ini nantinya memiliki **property** dan **method** yang bisa digunakan untuk mengatur tampilan. “**Model**” yang dipakai di dalam DOM adalah dengan “memetakan” seluruh object HTML layaknya sebuah **pohon (tree)**. Mari, perhatikan kode HTML berikut:

<sup>1</sup><http://webassembly.org/demo/>

<sup>2</sup><https://www.sitepoint.com/javascript-internet-things/>

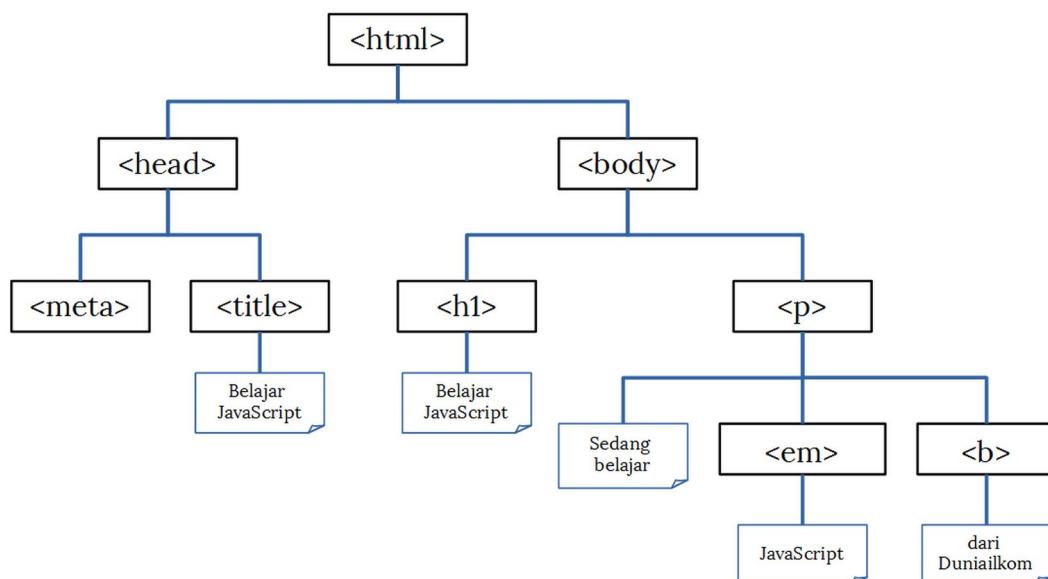
```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10 </body>
11 </html>

```

Kode HTML ini cukup sederhana. Di dalam tag `<head>` terdapat 2 tag lain: `<meta>` dan `<title>`. Di dalam tag `<body>` juga ada 2 tag: `<h1>` dan `<p>`. Untuk tag `<p>`, didalamnya saya menambahkan tag `<em>` dan `<b>`.

Jika digambarkan ke dalam pohon DOM (DOM tree), hasilnya adalah sebagai berikut:



Gambar: DOM tree

Diagram pohon diatas bisa dibaca sebagai pohon keluarga. Paling puncak terdapat tag `<html>`. Tag `<html>` disebut juga sebagai **root element**, yakni akar dari seluruh element HTML lain.

Tag `<html>` memiliki 2 anak (**child**) yakni `<head>` dan `<body>`. Dengan kata lain, tag `<html>` adalah induk (**parent**) dari tag `<head>` dan `<body>`. Tag `<head>` dan `<body>` merupakan saudara kandung (**sibling**).

Urutan dari tag ini juga memiliki sebutan tersendiri. Tag `<head>` merupakan anak pertama (**first child**) dari tag `<html>`, sedangkan tag `<body>` merupakan anak terakhir (**last child**) dari tag `<html>`.



Dalam bahasan selanjutnya, saya lebih banyak menggunakan istilah bahasa inggris: **child**, **parent**, **sibling**, **first child** dan **last child**. Karena akan bersesuaian dengan method dan property object DOM nantinya.

Berdasarkan diagram pohon diatas, berikut beberapa fakta yang bisa kita ambil:

- Tag <head> memiliki 2 **child**: tag <meta> dan <title>.
- Tag <body> memiliki 2 **child**: tag <h1> dan <title>.
- Tag <h1> merupakan **first child** dari tag <body>.
- Tag <p> merupakan **last child** dari tag <body>.
- Tag <body> merupakan **parent** dari tag <h1> dan <p>.
- Tag <p> merupakan **sibling** dari tag <h1>.
- Tag <h1> merupakan **sibling** dari tag <p>.
- Tag <b> merupakan **last child** dari tag <p>.

Selain tag HTML, saya juga mencantumkan **teks**, yakni seperti teks "Belajar JavaScript" yang merupakan **child** dari tag <title> dan tag <h1>. Di dalam struktur DOM, teks merupakan bagian terpisah dari tag, dimana teks yang ada didalam tag tersebut akan menjadi **child**.

Artinya, tag <p> memiliki 3 child: Teks "Sedang Belajar", tag <em>, dan tag <b>. Masing-masing tag <em> dan <b> memiliki 1 **child**, yakni teks "JavaScript" dan "dari DuniaIlkom".

Silahkan anda bandingkan kembali diagram pohon DOM dengan tag HTML yang saya tulis sebelumnya. Pastikan anda paham cara membacanya. Yakni kenapa tag yang satu disebut **child**, dan yang lain sebagai **parent** atau **sibling**. Kalau perlu silahkan tambah tag-tag lain, dan simpulkan apakah dia sebagai **child** atau **parent** dari tag apa.

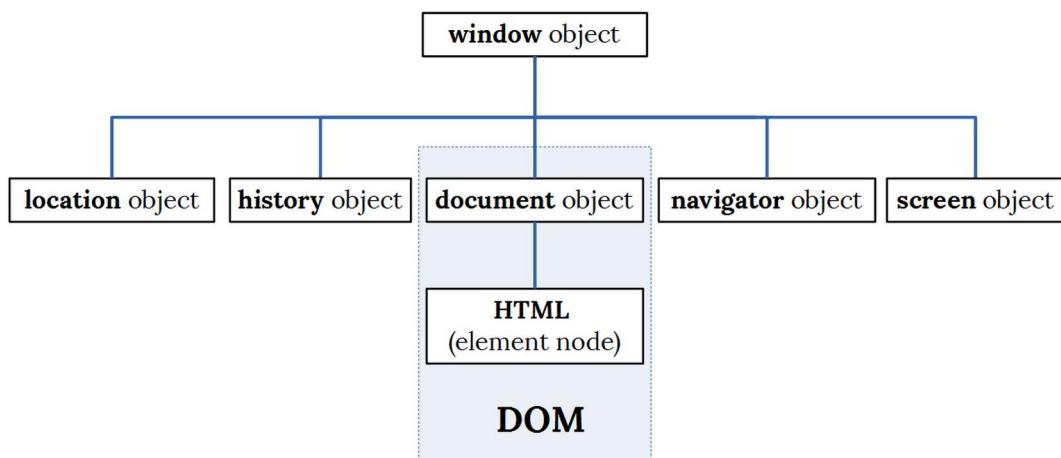
## 19.3 Pengertian BOM (Browser Object Model)

Sebelum kita mengenal DOM dengan lebih detail, saya ingin membahas sekilas tentang istilah lain, yakni **BOM** (Browser Object Model).



Mohon tidak disamakan dengan "bom" yang bisa meledak, kalau "bom" yang itu sangat sangat sensitif untuk dibahas :)

**BOM** (Browser Object Model) adalah **model dari seluruh object yang ada di dalam web browser**, dimana salah satu object tersebut adalah **DOM**. Artinya, **BOM** merupakan ruang lingkup yang lebih luas daripada **DOM**. Berikut diagramnya:



Gambar: Diagram dari BOM (Browser Object Model)

Semua kotak ini tergabung ke dalam **DOM**. Terlihat bahwa **DOM** merupakan bagian dari **BOM**. Pemodelan pohon keluarga juga berlaku disini, dimana **window object** merupakan **parent** dari seluruh object lain, termasuk **DOM** (**document object**).

Seperti yang akan kita bahas nantinya, **DOM** hanya berurusan dengan dokumen **HTML**, yakni seluruh tag yang ada di **HTML**. Tapi bagaimana dengan lebar jendela web browser? Scroll bar web browser? Atau alamat situs yang ditulis di address bar web browser? Ini semua berada di luar **DOM**.

Materi tentang **BOM** akan dibahas dalam bab tersendiri. Disini saya hanya ingin memperkenalkan kepada anda bahwa ada struktur yang lebih tinggi daripada **DOM**, yakni **window object**.

Dalam diagram diatas, **window object** berperan sebagai **global object**. Seluruh object lain (termasuk **document object**) merupakan turunan dari **window object**.

Dengan statusnya sebagai **global object**, semua variabel dan fungsi yang kita buat di JavaScript, melekat ke **window object**. Dengan kata lain, variabel sebenarnya adalah **property** dari **window object**, dan function adalah **method** dari **window object**. Berikut pembuktian dari hal ini:

```

1 var foo = "DuniaIlkom";
2 console.log(foo);           // DuniaIlkom
3
4 window.foo = "JavaScript";
5 console.log(foo);           // JavaScript
6
7 window.bar = "Belajar";
8 console.log(bar);           // Belajar
9
10 function salam(a){
11     return "Hello " + a;
12 }
13
14 var baz = window.salam("Jakarta");
15 console.log( baz );         // Hello Jakarta
  
```

Di awal kode program, saya mendefenisikan variabel `foo` dengan string "DuniaIlkom", kemudian menampilkannya dengan perintah `console.log(foo)`. Seperti yang bisa kita tebak, hasilnya adalah string "DuniaIlkom".

Pada baris selanjutnya, terdapat perintah `window.foo = "JavaScript"`. Jika anda masih ingat tentang konsep **object**, ini merupakan cara peng-inputan nilai "JavaScript" ke dalam **property** `foo` dari object `window`.

Ketika perintah `console.log(foo)` dijalankan kembali, hasilnya adalah "JavaScript", bukan lagi "DuniaIlkom". Artinya, perintah `window.foo = "JavaScript"` telah mengubah nilai variabel `foo`. Kesimpulan yang dapat diambil, variabel `foo` dengan `window.foo` adalah variabel yang sama.

Di baris berikutnya, saya menjalankan perintah `window.bar = "Belajar"`. Perintah ini berarti saya ingin meng-input nilai string "Belajar" ke property `bar` dari object `window`. Apakah ini artinya variabel `bar` juga sama dengan `window.bar`? Hasil dari perintah `console.log(bar)` membuktikan hal ini.

Bagaimana dengan function? Saya mendefenisikan sebuah function `salam()` yang ternyata bisa dipanggil menggunakan perintah `window.salam("Jakarta")`. Artinya, function `salam()` merupakan sesuatu yang sama dengan method `salam()` pada `window` object.

Seluruh object JavaScript lain seperti **Number**, **Math**, atau **Array** juga melekat ke dalam **window** object:

```
1 console.log( window.Math.PI );
2 // 3.141592653589793
3
4 console.log( window.Number.parseInt("10101101",2) );
5 // 173
```

Masih ingat dengan perintah `alert()`? `alert()` sebenarnya adalah sebuah method dari **window object**:

```
1 window.alert("Saya merupakan method dari window object");
```

Selain `alert()`, masih banyak method dan property dari **window object**, yang akan kita bahas dalam bab khusus nantinya.

Tapi kalau `alert()` merupakan method dari **window object**, kenapa bisa dipanggil tanpa menulis `window`?

Alasannya karena **window object** merupakan **global object**, sehingga kita tidak harus menulisnya. Web browser secara otomatis menambahkan `window object` diawal seluruh property dan method JavaScript. Perintah `alert("Hello World")` sama artinya dengan `window.alert("Hello World")`.

**i** **Window object** merupakan bagian dari **BOM**, bukan JavaScript. Hanya saja disini kita menggunakan JavaScript di dalam web browser, sehingga **window object** terintegrasi dengan JavaScript. Namun jika JavaScript digunakan di tempat lain, seperti **Node.js**, **window object** sudah tidak ditemukan lagi.

**Window object** juga berisi berbagai informasi lain terkait frame web browser, yakni jendela web browser yang saat ini ditampilkan. Kita bisa mengatur lebar jendela web browser dengan mengakses **window object** ini.

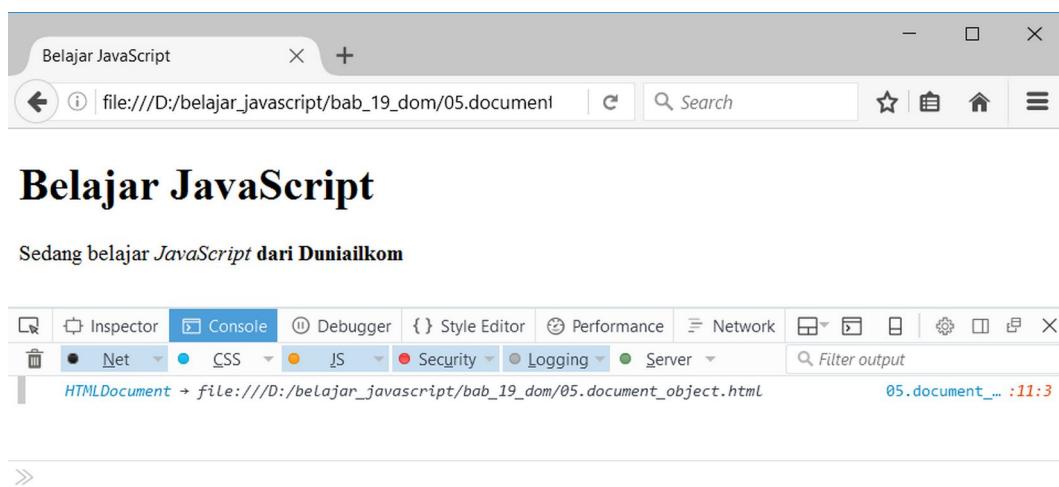
## 19.4 Document Object

Mari kita kembali membahas **DOM**. Dalam diagram sebelumnya, terlihat bahwa DOM berisi **document object**. Tag `<html>` yang menjadi root dari DOM juga ada di dalam **document object**. Apa sebenarnya isi dari **document object** ini? Mari kita lihat:

```

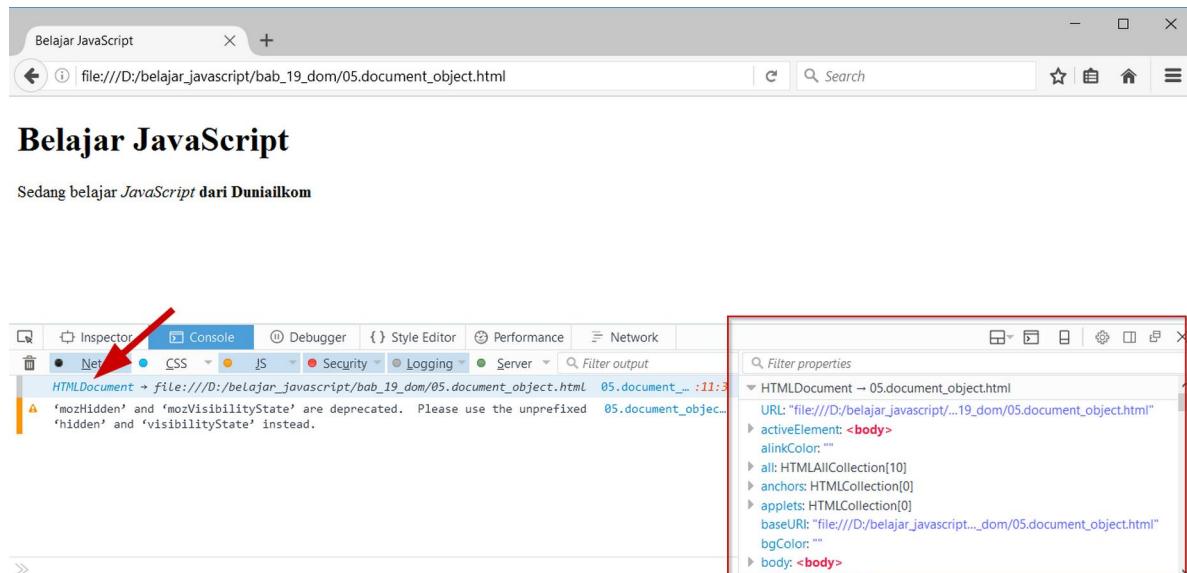
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10 <script>
11   console.log(window.document);
12 </script>
13 </body>
14 </html>

```



Gambar: Hasil dari perintah `console.log(window.document)`

Perintah `console.log()` tidak hanya menampilkan string hasil kode program, tapi terintegrasi dengan web browser itu sendiri. Hasil dari perintah `console.log(window.document)` hanya 1 baris, tapi bisa diklik. Silahkan anda klik bagian “**HTMLDocument**”, dan akan tampil jendela baru di sisi kanan **console**:



Gambar: Jendela property dari document object

Tab di sebelah kanan akan menampilkan *seluruh property* yang ada di dalam document object. Jumlahnya? **Lebih dari 100 property!**

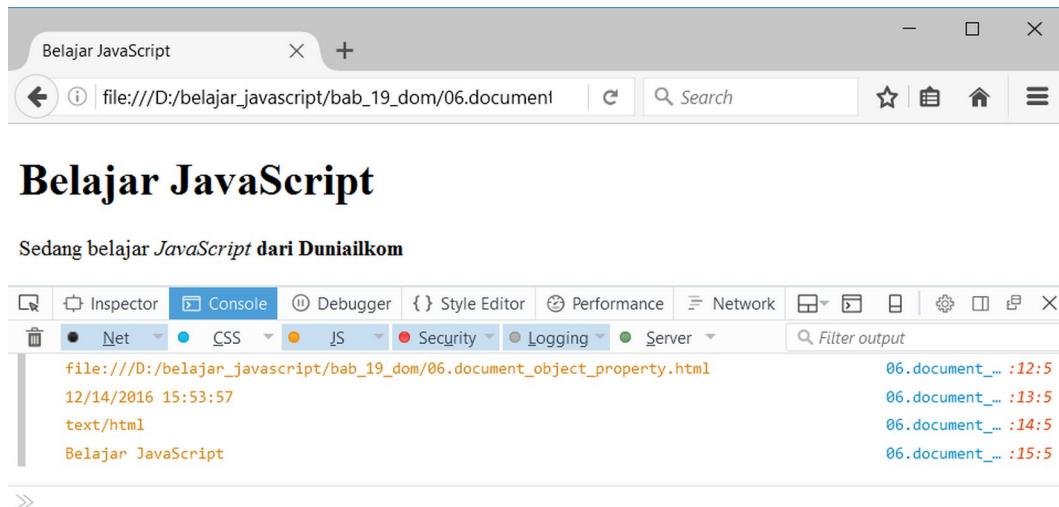
Kita tidak akan membahas semua property ini karena sebagian besar sangat jarang dipakai. Dari ribuan property yang ada di dalam DOM object, yang sering digunakan tidak sampai 10%-nya, bahkan kurang. Untuk permulaan, kita cukup mempelajari yang 10% ini.

Jika anda menjalankan kode program diatas secara langsung, silahkan lihat-lihat sebentar daftar property yang ada di tab console. Semua property dari **document object** bisa diakses sebagaimana kita mengakses property dari object JavaScript biasa:

```

1 var foo = window.document;
2 console.log(foo.URL);
3 console.log(foo.lastModified);
4 console.log(foo.contentType);
5 console.log(foo.title);

```



Gambar: Mengakses property dari document object

Disini saya membuat variabel `foo` yang isinya berupa `window.document`, yakni **document object**. Perintah `foo.URL` sama saja dengan `window.document.URL`. Property ini berisi informasi mengenai alamat URL dari file yang saat ini sedang dijalankan. Dapat anda lihat bahwa saya menyimpan file diatas di drive D dengan nama folder `D:\belajar_javascript\bab_19-dom.html`

Property `window.document.lastModified` berisi informasi modifikasi terakhir dari file dokumen. Hasilnya adalah `12/14/2016 15:53:57`, yakni tanggal pada saat saya mengedit file tersebut.

Property `window.document.contentType` berisi informasi tentang MIME type dari file, yakni `text/html`. Sedangkan property `window.document.title` berisi informasi dari judul dokumen yang diambil dari tag `<title>`.

Anda bisa samakan nilai-nilai yang didapat dengan isi dari jendela console sebelumnya. Jika perlu, silahkan akses property lain dan lihat apakah isinya sama dengan yang ditampilkan menggunakan perintah `console.log()`.

Apa yang kita praktekkan disini menunjukkan bahwa **document object** merupakan bagian dari **window object**. Dan karena **window object** sendiri sebenarnya tidak perlu ditulis, saya juga bisa menjalankan kode berikut:

```

1 var foo = document;
2 console.log(foo.URL);
3 console.log(foo.lastModified);
4 console.log(foo.contentType);
5 console.log(foo.title);

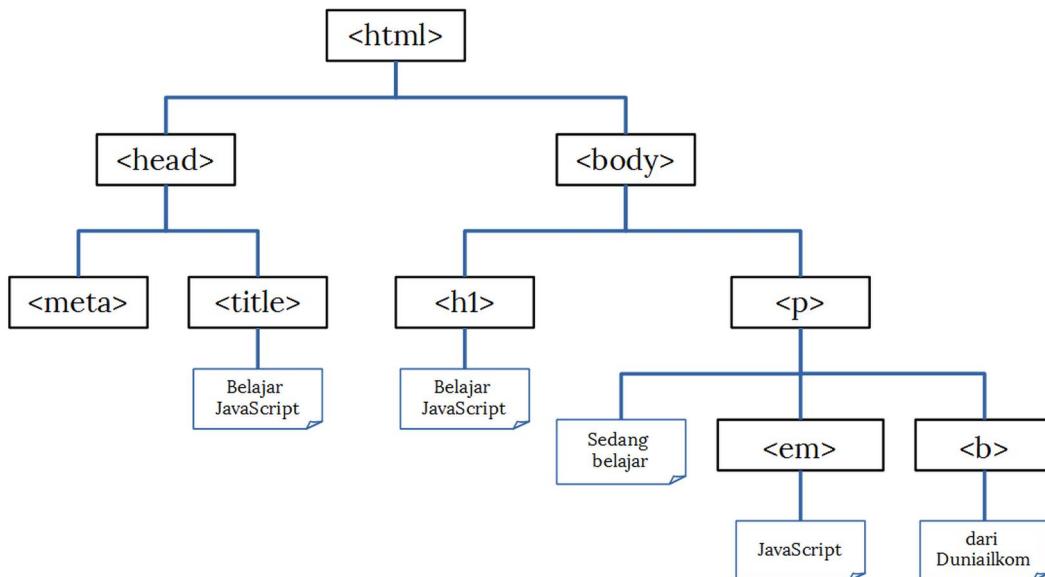
```

Perhatikan bahwa kali ini saya langsung membuat perintah `var foo = document`, tidak lagi `var foo = window.document`. Selain memiliki ratusan property, **document object** juga memiliki berbagai method. Kita akan membahas hal ini di pertengahan bab nanti.

Baik, setelah melihat hubungan dari **document object** dengan **window object**, selanjutnya bagaimana dengan struktur pohon yang berisi tag-tag HTML? Apakah masih berkaitan dengan **document object**? Jawabannya ada di dalam **Node object**.

## 19.5 Node Object

Di dalam konsep DOM, setiap element atau tag HTML diproses sebagai object yang bernama **Node Object**. Mari kita lihat kembali gambar DOM tree yang pernah saya tampilkan di awal bab:



Gambar: DOM tree

Setiap kotak yang terdapat dalam diagram diatas merupakan **node object**, yang terdiri dari seluruh element HTML beserta teks yang menjadi *child* dari element HTML tersebut. Element HTML merupakan **node object**, teks juga **node object**.

Jika yang diproses adalah element HTML seperti `<body>`, `<h1>` atau `<p>` dinamakan sebagai **element node**. Sedangkan teks yang ada di dalam element HTML dinamakan sebagai **text node**. Dalam diagram diatas, terdapat **9 element node** dan **5 text node**.

Sebagaimana layaknya object JavaScript lain, **node object** juga memiliki berbagai property dan method. Property dan method inilah yang nantinya bisa kita akses untuk mengubah tampilan dari element tersebut.

Misalnya saya ingin mengubah teks yang ada di dalam sebuah paragraf. Caranya adalah dengan mengubah property `textContent` yang terdapat di **element node** `<p>`. Namun sebelum sampai kesini, tentunya kita harus menemukan **element node** yang ingin diubah.

Terdapat berbagai cara untuk mencari **element node**. Yang paling terkenal dan juga paling sering digunakan adalah dengan method `document.getElementById()`. Jika anda pernah melihat kode program JavaScript yang melibatkan DOM, hampir selalu menggunakan method ini.

Perintah `document.getElementById()` artinya kita memakai method `getElementById()` kepuanyaan **document object**. Apakah terdapat method lain untuk mencari **element node**? Yup, **document object** memiliki method `getElementsByName()`, `querySelector()` hingga `querySelectorAll()`.

Mencari sebuah **element node** menggunakan method-method diatas saya anggap sebagai “cara cepat”. Karena penggunaannya sangat mudah. Namun sebelumnya, saya ingin mengajak anda untuk menggunakan “cara susah” terlebih dahulu. Karena konsep ini sangat penting untuk memahami DOM secara keseluruhan.

“Cara susah” itu adalah dengan menelusuri pohon DOM untuk mencari element yang diinginkan. Sebagai contoh, bagaimana cara mengakses **element node** `<em>` dari diagram diatas? Kita harus mulai dari **element node** `<html>`, lalu ke `<body>`, ke `<p>`, dan baru ketemu element `<em>`.

Apakah anda masih ingat bahwa saya menggunakan istilah **child**, **parent**, **sibling**, **first child** dan **last child** untuk menelusuri struktur DOM? Karena memang terdapat method yang berfungsi untuk mencari **element node** berdasarkan hubungan seperti ini.

Baik, mari kita mulai prakteknya.

Sebelum masuk ke pembahasan tentang **node object**, kita telah mencoba mengakses property dari **document object**. **Document object** sendiri sebenarnya juga termasuk ke dalam **node object**, sehingga memiliki property yang berkaitan dengan **Node**. Berikut beberapa diantaranya:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10 <script>
11   console.log(document.nodeName);      // #document
12   console.log(document.nodeType);     // 9
13   console.log(document.childNodes);   // NodeList [ DocumentType, <html> ]
14   console.log(document.childNodes.length); // 2
15 </script>
16 </body>
17 </html>
```

Property `nodeName` berisi informasi mengenai nama dari **node object** yang saat ini sedang diakses. Jika node tersebut merupakan element HTML, hasilnya berupa nama element tersebut, seperti `p`, `h1`, `table`, `form`, dll. Disini saya mengaksesnya dari **document object**, maka hasilnya adalah `#document`.

Property `nodeType` berisi informasi mengenai tipe atau jenis dari **node object** yang saat ini sedang diakses. Angka 9 artinya document object termasuk ke dalam kelompok Document node.

Terdapat 12 jenis node di dalam DOM, yang bisa dilihat pada tabel berikut:

Nilai	Jenis Node	Penjelasan
1	Element node	Element HTML
2	Attr node	Atribut dari sebuah element HTML
3	Text node	Teks yang biasanya berada di dalam element HTML
4	CDATASection node	CDATA section dari dokumen XML
5	EntityReference node	Sebuah referensi ke entity dari dokumen XML
6	Entity node	Entity dari sebuah DTD
7	ProcessingInstruction node	Sebuah instruksi pemrosesan
8	Comment	Komentar HTML atau XML
9	Document node	Root node dari seluruh element HTML
10	DocumentType node	DTD dari sebuah dokumen XML
11	DocumentFragment node	Ruang sementara (temporary storage) untuk menampung bagian dari dokumen
12	Notation node	Notation dari DTD

Dengan mengakses property `nodeType` dari sebuah **node object**, kita bisa melihat angka yang menunjukkan jenis dari node tersebut.

Sekedar informasi, DOM tidak hanya digunakan oleh dokumen HTML, tapi juga untuk dokumen XML seperti SVG (Scalable Vector Graphics). Oleh karena itulah dalam tabel diatas juga terdapat node object kepunyaan XML.

Untuk dokumen HTML, kita hanya akan berurusan dengan 5 jenis node saja: **Document**, **Element**, **Text**, **Attr** dan **Comment**.

- **Document node** merupakan root object dari seluruh tag HTML. Object ini berisi berbagai property dan method yang bisa digunakan untuk mencari dan mengubah node lain.
- **Element node** merupakan node object untuk tag HTML, seperti `<p>`, `<h1>`, atau `<form>`. Element node bisa menjadi parent sekaligus child dari element node lain. Node inilah yang nantinya paling banyak kita pakai.
- **Text node** adalah node object yang ada di dalam sebuah element HTML. Text node merupakan child dari element node, serta tidak bisa memiliki child node lain.
- **Attr node** adalah node object yang berisi atribut dari sebuah element HTML. Dalam standar terbaru, penggunaannya sudah tidak disarankan lagi. Untuk mengubah atribut dari element HTML, kita bisa mengakses langsung dari element node.
- **Comment node** adalah node object yang terbentuk dari komentar HTML, yakni tag `<!--` dan `-->`. Walaupun komentar tidak akan diproses oleh web browser, tapi bisa diakses dari DOM.

Mari kita lanjutkan membahas property dari document object.

Property `childNodes` berisi daftar anak (child) dari sebuah **node object**. Berapa jumlahnya? Kita bisa akses dari property `childNodes.length`. Dari hasil tampilan kode terlihat bahwa

**document object** memiliki 2 child, yang ditampilkan sebagai: `NodeList [ DocumentType, <html> ]`.

Artinya, child dari **document object** ada 2, yakni: `DocumentType` dan `<html>`. **NodeList** adalah istilah khusus untuk menyebut kumpulan dari berbagai node. **NodeList** ini mirip seperti array, tapi bukan array. Dalam JavaScript, ia termasuk kategori **collection**.

Karena terdiri dari kumpulan element, **collection** (termasuk `NodeList`) memiliki property `length`, yang berisi informasi mengenai jumlah anggota collection tersebut. Collection pun bisa diakses sebagaimana layaknya element array (menggunakan tanda kurung siku).

Kita akan sering berhubungan dengan `NodeList` ini nantinya. Sebagai contoh, kumpulan dari seluruh tag `<p>` yang ada di halaman HTML akan menjadi sebuah `NodeList`.

Anak pertama, atau **first child** dari **document object** adalah `DocumentType`. Node ini merupakan tag `<!DOCTYPE html>` yang ditulis di baris paling awal dokumen HTML. Sedangkan anak kedua yang sekaligus sebagai **last child** dari **document object** adalah `<html>`. Inilah *root element* yang berisi seluruh tag HTML. Melalui element inilah kita bisa menelusuri element HTML lainnya.

Baik, sekarang bagaimana cara mengakses isi dari kedua child ini? Ingat bahwa keduanya disimpan ke dalam `NodeList` yang berbentuk collection. Artinya bisa kita akses sebagaimana layaknya array:

```
1 var foo = document.childNodes[0];
2 var bar = document.childNodes[1];
```

Inilah cara kita untuk “turun” menelurusui DOM tree.

Variabel `foo` akan berisi `childNodes` pertama dari **document object**, sedangkan variabel `bar` akan berisi `childNodes` kedua dari **document object**. *Collection* kurang lebih sama seperti array, sehingga element pertama juga berada di index 0.

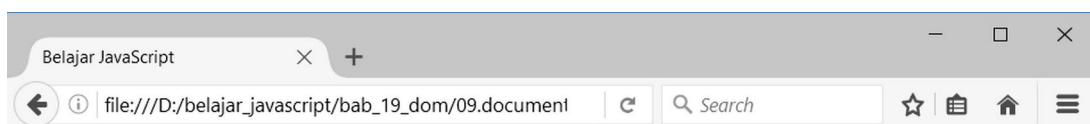
Sekarang, mau kita “apakan” variabel `foo` dan `bar` ini? Kedua variabel sebenarnya berisi **node object**, karena itu bagaimana kalau kita akses menggunakan property yang sama seperti **document object**?

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10  <script>
11    var foo = document.childNodes[0];
12    console.log(foo.nodeName);           // html
13    console.log(foo.nodeType);          // 10
```

```

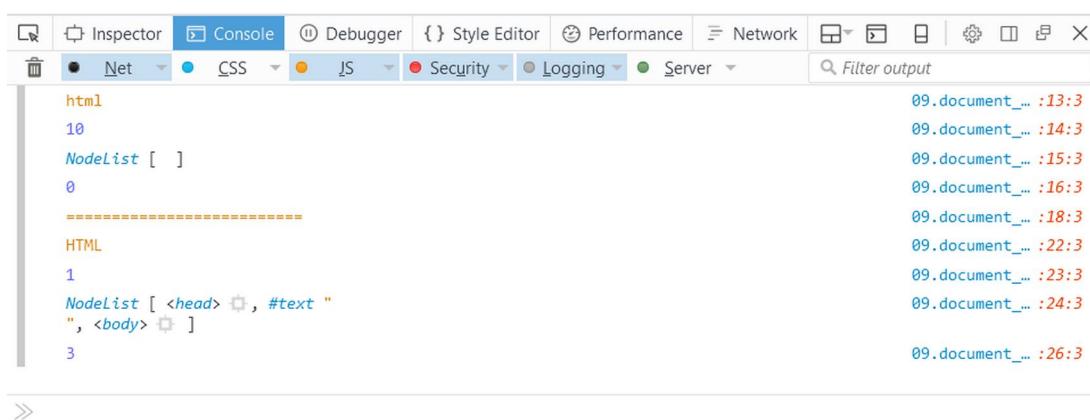
14  console.log(foo.childNodes);           // NodeList [ ]
15  console.log(foo.childNodes.length);   // 0
16
17  console.log("=====");
18
19  var bar = document.childNodes[1];
20  console.log(bar.nodeName);           // HTML
21  console.log(bar.nodeType);          // 1
22  console.log(bar.childNodes);         // NodeList [ <head>, #text " ", <body> ]
23  console.log(bar.childNodes.length);   // 3
24  </script>
25 </body>
26 </html>

```



## Belajar JavaScript

Sedang belajar *JavaScript* dari **Duniailkom**



Gambar: Mengakses child node dari document object

Perintah `console.log(foo.nodeName)` artinya akses nilai property `nodeName` dari variabel `foo`. Apa isi dari variabel `foo`? Yakni **node object** yang merupakan **first child** dari `document object`. Isi variabel `foo` berasal dari `document.childNodes[0]`.

Dari tampilan diatas terlihat bahwa **first child** dari `document object` memiliki:

- `nodeName: html`
- `nodeType: 10`
- `childNodes: NodeList [ ]`
- `childNodes.length: 0`

Loh, bukannya first child document object adalah `DocumentType`? Betul, hasil dari property `nodeName` memang `html`, tapi memiliki `nodeType` 10. Dari tabel jenis-jenis node type terlihat bahwa nilai 10 berarti object ini adalah **DocumentType node**, yang menyimpan informasi mengenai **DTD** (Document Type Definition), yakni sebutan untuk tag `<!DOCTYPE html>`.

Mari kita lihat informasi yang didapat dari **last child** document object:

- `nodeName: HTML`
- `nodeType: 1`
- `childNodes: NodeList [ <head>, #text " ", <body> ]`
- `childNodes.length: 3`

Hasil `nodeName: HTML` dan `nodeType: 1`, artinya object ini merupakan sebuah **element node** bernama HTML. Element node ini memiliki 3 child, yakni: `<head>`, `#text " "`, dan `<body>`.

Tapi terdapat sesuai yang cukup aneh. Kalau element ini adalah tag `<html>` (**root element**), kenapa ada 3 child? Seharusnya di dalam tag `<html>` hanya terdapat 2 child, yakni `<head>` dan `<body>`.

Inilah salah satu masalah yang sering membuat pusing jika menelusuri DOM tree satu per satu. Child kedua dari HTML node adalah sebuah **text node**, yang isinya berupa karakter enter yang kita gunakan sebagai pemisah antara tag penutup `</head>` dengan tag pembuka `<body>`.

Andai saja saya menulis tag `</head>` dan `<body>` di dalam 1 baris yang sama, teks node ini akan hilang:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8">
5      <title> Belajar JavaScript </title>
6  </head><body>
7      <h1>Belajar JavaScript</h1>
8      <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
9      <script>
10         var bar = document.childNodes[1];
11
12         console.log(bar.nodeName);           // HTML
13         console.log(bar.nodeType);          // 1
14         console.log(bar.childNodes);         // NodeList [ <head>, <body> ]
15         console.log(bar.childNodes.length); // 2
16     </script>
17 </body>
18 </html>
```

Apakah anda menemukan perbedaan dari struktur HTML ini? Perhatikan di baris ke 6, saya menyambung penulisan `</head><body>`. Setelah tag penutup `</head>`, saya tidak menekan enter

untuk pindah ke baris baru, tetapi langsung menyambung penulisannya. Jika ditekan enter, (dimana tag </head> dan <body> akan berada di baris yang terpisah) karakter enter ini akan menjadi child kedua dari element node HTML.

Ini pula yang menjadi alasan kenapa di tampilan tab console kode program sebelumnya, hasil `bar.childNodes` tidak ditampilkan dalam 1 baris, tapi pindah ke baris dibawahnya. Dikarenakan **text node** itu berisi sebuah karakter enter, atau dikenal dengan istilah *carriage return*:

```

1 html
2 10
3 NodeList [ ]
4 0
=====
5 HTML
6 1
7 NodeList [ <head>, #text ""
8 "", <body> ]
9 3

```

Gambar: Text node ditampilkan di baris yang berbeda (karakter *carriage return*)

Sebagai tambahan, **element node** memiliki property `firstChild` dan `lastChild` yang bisa digunakan untuk mengakses anak pertama dan anak terakhir dari sebuah **Nodelist**. Karena **document object** hanya memiliki 2 child, efeknya akan sama seperti kode kita sebelumnya:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10  <script>
11    var foo = document.firstChild;
12
13    console.log(foo.nodeName);           // html
14    console.log(foo.nodeType);          // 10
15    console.log(foo.childNodes);         // NodeList [ ]
16    console.log(foo.childNodes.length); // 0
17
18    console.log("=====");
19
20    var bar = document.lastChild;
21
22    console.log(bar.nodeName);           // HTML
23    console.log(bar.nodeType);          // 1

```

```
24     console.log(bar.childNodes);      // NodeList [ <head>, #text " ", <body> ]
25     console.log(bar.childNodes.length); // 3
26     </script>
27 </body>
28 </html>
```

Artinya untuk document object, `document.firstChild` sama dengan `document.childNodes[0]`. Sedangkan `document.lastChild` sama dengan `document.childNodes[1]`.

## 19.6 Menelusuri DOM Tree

Sebelum melanjutkan materinya, saya ingin review sedikit tentang apa yang sudah kita pelajari. Bisakah anda menjelaskan maksud dari istilah: `node object`, `element node`, `nodeType`, `childNodes`, serta `NodeList` ?

Semua istilah ini mirip satu sama lain, tapi sangat penting dipahami:

- **Node object** adalah seluruh “object” yang ada di dalam struktur DOM, termasuk element HTML, teks HTML, atribut, dll. Terdapat 12 jenis node object di dalam DOM. Yang akan sering kita akses hanyalah *element node* dan *text node*.
- **Element node** adalah salah satu jenis node yang isinya berupa tag atau element HTML, seperti `<p>`, `<h1>`, termasuk `<html>`.
- **nodeType** adalah salah satu property dari node object, isinya angka yang menginformasikan jenis dari node object tersebut.
- **childNodes** adalah salah satu property dari node object, isinya berupa kumpulan child nodes yang tergabung ke dalam *NodeList*.
- **NodeList** adalah kumpulan berbagai node object. Nodelist berbentuk *collection* yang mirip seperti array.

Jika anda masih kurang paham mengenai istilah-istilah ini, saya sarankan untuk mengulang kembali pembahasan sebelumnya.

Baik, mari kita lanjutkan.

Element node `<html>` ternyata memiliki 3 child node: `<head>`, `#text " "`, dan `<body>`. Bagaimana kalau kita lihat isi dari masing-masing node ini? Berikut kode programnya:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10 <script>
11   var foo = document.childNodes[1].childNodes[0];
12
13   console.log(foo.nodeName);           // HEAD
14   console.log(foo.nodeType);          // 1
15   console.log(foo.childNodes);
16   // NodeList [ #text " ", <meta>, #text " ", <title>, #text " " ]
17   console.log(foo.childNodes.length); // 5
18
19   console.log("=====");
20
21   var bar = document.childNodes[1].childNodes[1];
22
23   console.log(bar.nodeName);          // #text
24   console.log(bar.nodeType);          // 3
25   console.log(bar.childNodes);         // NodeList [ ]
26   console.log(bar.childNodes.length); // 0
27
28   console.log("=====");
29
30   var baz = document.childNodes[1].childNodes[2];
31
32   console.log(baz.nodeName);          // BODY
33   console.log(baz.nodeType);          // 1
34   console.log(baz.childNodes);
35   // NodeList [ #text " ", <h1>, #text " ", <p>, #text " ", <script> ]
36   console.log(baz.childNodes.length); // 6
37 </script>
38 </body>
39 </html>
```

Kode program diatas cukup panjang, karena saya perlu mengakses 3 child node HTML. Mari kita bahas hasil yang di dapat.

**Child pertama** dari HTML node adalah `<head>`. Node ini diakses menggunakan perintah `document.childNodes[1].childNodes[0]`. Inilah cara untuk turun dari document object, ke `childNodes[1]`, yakni tag `<html>`, lalu ke `childNodes[0]`, yakni tag `<head>`. Dengan kata lain,

element node **HEAD** adalah cucu pertama dari anak kedua document object (mudah-mudahan anda mengerti maksudnya).

Penulisan alamat sebuah node bisa disambung terus menerus, targantung struktur HTML yang ditulis serta element apa yang ingin dicari. Sebagai contoh, bisa saja untuk menemukan sebuah element, kita mencarinya menggunakan perintah:

```
var foo = document.childNodes[1].childNodes[2].childNodes[6].childNodes[14];
```

Ini bisa dibaca dengan: “*mulai dari document object, turun ke child node index 1, turun lagi ke child node dengan index 2, turun lagi ke child node dengan index 6, dan terakhir turun lagi ke child node dengan index 14*”. Inilah materi yang paling penting dari DOM tree, yakni cara menelusuri struktur DOM untuk menemukan sebuah element node.

Dari hasil yang didapat, element node `<head>` memiliki nama `nodeName HEAD`, dan mempunyai 5 child: `NodeList [ #text " ", <meta>, #text " ", <title>, #text " " ]`.

Kenapa begitu banyak child? Bukannya tag `<head>` berisi hanya 2 tag saja, yakni `<meta>` dan `<title>`? Jawabannya sama seperti masalah yang kita temui sewaktu membahas element node HTML. **HEAD** memiliki 3 text node, yang semuanya berasal dari karakter *carriage return*.

Jika saya menulis seluruh tag di dalam tag `<head>` dalam 1 baris panjang, text node ini tidak akan ditemui:

```
1  <!DOCTYPE html>
2  <html><head><meta charset="utf-8"><title>Belajar JavaScript</title></head>
3  <body>
4      <h1>Belajar JavaScript</h1>
5      <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
6      <script>
7          var foo = document.childNodes[1].childNodes[0];
8
9          console.log(foo.nodeName);           // HEAD
10         console.log(foo.nodeType);        // 1
11         console.log(foo.childNodes);       // NodeList [ <meta>, <title> ]
12         console.log(foo.childNodes.length); // 2
13     </script>
14 </body>
15 </html>
```

Dibaris ke-2, seluruh isi tag `<head>` yakni `<meta>` dan `<title>` saya tulis dalam 1 baris panjang. Hasilnya, `childNodes` dari **HEAD** element hanya ada 2, yakni: `NodeList [ <meta>, <title> ]`. Ini sesuai dengan diagram DOM tree yang ada di awal bab ini.

Isi teks title sengaja saya persingkat dari "Belajar JavaScript" menjadi "JavaScript" saja agar baris tersebut tidak terlalu panjang. Ini tidak akan berpengaruh ke hasil yang di dapat.

Kembali ke pembahasan tentang `childNode` dari **HTML** element, child kedua adalah sebuah text node, yang beralamat di `document.childNodes[1].childNodes[2]`. Text node ini berupa karakter *carriage return* yang berada antara tag `<head>` dengan `<body>`.

Child ketiga dari element node **HTML** adalah element node **BODY**, yang beralamat di `document.childNodes[1].childNodes[2]`. Berapakah jumlah child dari node ini? Ada 6: `NodeList [ #text " ", <h1>, #text " ", <p>, #text " ", <script> ]`. Dimana 3 diantaranya berupa text node.

Yang cukup unik, last child dari **BODY** adalah `<script>`. Ini merupakan tag `<script>` untuk menginput kode JavaScript. Kenapa bisa tampil disini? Karena tag `<script>` memang saya tempatkan di bagian akhir tag `<body>`. Hal ini juga membuktikan bahwa DOM bisa mengakses tag apapun, termasuk tag `<script>`.

Sampai disini saya yakin anda sudah cukup paham cara menelusuri DOM tree secara berurutan dari document object.

Sebagai latihan, bisakah anda menampilkan informasi mengenai tag `<p>`? Dari hasil sebelumnya terlihat bahwa tag `<p>` merupakan child ke-4 dari node element **BODY**.

Baik, berikut kode yang saya gunakan:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="utf-8">
5  <title> Belajar JavaScript </title>
6  </head>
7  <body>
8  <h1>Belajar JavaScript</h1>
9  <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10 <script>
11   var foo = document.childNodes[1].childNodes[2].childNodes[3];
12
13   console.log(foo.nodeName);           // P
14   console.log(foo.nodeType);          // 1
15   console.log(foo.childNodes);
16   // NodeList [ #text "Sedang belajar ", <em>, <b> ]
17   console.log(foo.childNodes.length); // 3
18 </script>
19 </body>
20 </html>
```

Terlihat bahwa node element **P** memiliki 3 child, yakni `NodeList [ #text "Sedang belajar ", <em>, <b> ]`.

Sekali lagi, bagaimana kalau menampilkan isi dari element `<em>` yang ada di dalam tag `<p>` ini?

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10 <script>
11   var foo = document.childNodes[1].childNodes[2].childNodes[3].childNodes[1];
12
13   console.log(foo.nodeName);           // EM
14   console.log(foo.nodeType);          // 1
15   console.log(foo.childNodes);         // NodeList [ #text "JavaScript" ]
16   console.log(foo.childNodes.length);  // 1
17 </script>
18 </body>
19 </html>
```

Element node EM memiliki 1 child, berupa text node: NodeList [ #text "JavaScript" ]. Ini sesuai dengan diagram DOM.

Sebagai tambahan, alamat dari node element juga bisa dipecah ke dalam variabel, kemudian disambung kembali. Perhatikan kode program berikut:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10 <script>
11   var foo = document.childNodes[1].childNodes[2].childNodes[3];
12   var bar = foo.lastChild;
13
14   console.log(bar.nodeName);           // B
15   console.log(bar.nodeType);          // 1
16   console.log(bar.childNodes);         // NodeList [ #text " dari DuniaIlkom" ]
17   console.log(bar.childNodes.length);  // 1
18 </script>
19 </body>
20 </html>
```

Dengan membandingkan hasil yang kita dapat sebelumnya, variabel `foo` akan berisi node object `P`, yakni yang berada di alamat `document.childNodes[1].childNodes[2].childNodes[3]`. Untuk variabel `bar`, saya mengambil variabel `foo`, kemudian mencari `lastChild` dari `foo`. Hasilnya adalah element node `B`, yakni tag `<b>` yang menjadi `last child` dari tag `<p>`.

Nantinya kita akan banyak menggunakan cara penulisan seperti ini, dimana node element yang tersimpan di sebuah variabel akan dilanjutkan oleh variabel lain.

## 19.7 Node Object Property

Sebagaimana object lain di dalam JavaScript, `node object` juga memiliki berbagai property dan method. Saya akan membahas property terlebih dahulu.

Dalam materi sebelum ini, sebenarnya kita telah memakai beberapa property node object, yakni `nodeName`, `nodeType`, dan `childNodes`. Berikut daftar property lain dari node object:

- `Node.baseURI` (read only)
- `Node.childNodes` (read only)
- `Node.firstChild` (read only)
- `Node.lastChild` (read only)
- `Node.nextSibling` (read only)
- `Node.nodeName` (read only)
- `Node.nodeType` (read only)
- `Node.nodeValue`
- `Node.ownerDocument` (read only)
- `Node.parentNode` (read only)
- `Node.parentElement` (read only)
- `Node.previousSibling` (read only)
- `Node.rootNode` (read only)
- `Node.textContent`



Daftar property diatas saya ambil dari [Node References - Mozilla Developer Network](#)<sup>3</sup>.

Keterangan (read only) artinya nilai property tersebut tidak bisa diubah (hanya bisa diakses saja).

Sebagian besar dari property ini berkaitan dengan node object lain. Sebagai contoh, property `nextSibling` dan `previousSibling` berguna untuk menampilkan node object yang menjadi saudara kandung (*sibling*) dari object saat ini. Property `parentNode` berguna untuk menampilkan node object yang menjadi orangtua (*parent*) dari object saat ini.

Mari kita coba akses semua property:

<sup>3</sup><https://developer.mozilla.org/en-US/docs/Web/API/Node>

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10 <script>
11   var pNode = document.childNodes[1].childNodes[2].childNodes[3];
12
13   console.log( pNode.baseURI );
14   // file:///D:/belajar_javascript/bab_19_dom/17.node_properties.html
15
16   console.log( pNode.childNodes );
17   // NodeList [ #text "Sedang belajar ", <em>, <b> ]
18
19   console.log( pNode.firstChild );           // #text "Sedang belajar "
20   console.log( pNode.lastChild );          // <b>
21   console.log( pNode.nextSibling );         // #text " "
22   console.log( pNode.nodeName );           // P
23   console.log( pNode.nodeType );           // 1
24   console.log( pNode.nodeValue );          // null
25   console.log( pNode.ownerDocument );
26   // HTMLDocument ->
27   // file:///D:/belajar_javascript/bab_19_dom/17.node_properties.html
28
29   console.log( pNode.parentNode );          // <body>
30   console.log( pNode.parentElement );        // <body>
31   console.log( pNode.previousSibling );      // #text " "
32   console.log( pNode.textContent );
33   // Sedang belajar JavaScript dari DuniaIlkom
34 </script>
35 </body>
36 </html>
```

Variabel pNode berisi **element node** di lokasi `document.childNodes[1].childNodes[2].childNodes[3]`. Element apakah itu? Jika anda memeriksa hasil kode program kita sebelum ini, variabel pNode akan berisi element node P, atau tag `<p>`.

Property `pNode.baseURI` akan menampilkan posisi dokumen dimana pNode berada, hasilnya adalah `file:///D:/belajar_javascript/bab_19_dom/17.node_properties.html`, karena disinilah saya menyimpan file HTML tersebut.

Property `pNode.childNodes` sudah sering kita pakai, yakni untuk menampilkan nilai **child nodes**. Untuk element node P, terdapat 3 child: `#text "Sedang belajar "`, `<em>` dan `<b>`.

Property `pNode.firstChild` dan `pNode.lastChild` berguna untuk menampilkan anak pertama dan anak terakhir dari element node **P**. Terlihat bahwa `pNode.childNodes` adalah `#text "Sedang belajar "`, sedangkan `pNode.lastChild` adalah `<b>`.

Property `pNode.nextSibling` berguna untuk menampilkan saudara kandung berikutnya dari `pNode`, yakni node setelah element **P**. Isinya adalah `#text " "`. Karena setelah tag `<p>` memang terdapat sebuah karakter carriage return (perhatikan kembali struktur HTMLnya).

Property `pNode.nodeName` dan `pNode.nodeType` juga sudah sering kita bahas, yang digunakan untuk menambil informasi tentang nama node dan tipe node. Variabel `pNode` memiliki nama **P** dan bertipe 1 (merupakan sebuah **element node**).

Property `pNode.nodeValue` menghasilkan nilai **null**. Untuk element node, property `nodeValue` akan selalu menghasilkan nilai null. Jika yang sedang diakses adalah text node, property ini akan menampilkan isi dari text tersebut.

Property `pNode.ownerDocument` digunakan untuk mencari informasi mengenai **root document** dari `pNode`. Hasilnya adalah document object (**HTMLDocument**) yang berada dalam file:///D:/belajar\_javascript/bab\_19\_dom/17.node\_properties.html.

Property `pNode.parentNode` dan `pNode.parentElement` hampir sama, yakni mengembalikan nilai **parent node**. Hanya saja untuk `parentElement`, akan menghasilkan nilai **null** jika parent node bukan sebuah element node. Untuk element node **P**, parentnya adalah tag `<body>` yang juga sebuah element node.

Property `pNode.previousSibling` berguna untuk menampilkan saudara kandung **sebelum** `pNode`, yakni node sebelum element **P**. Isinya adalah `#text " "`. Karena sebelum tag `<p>`, terdapat sebuah karakter carriage return.

Property `pNode.textContent` berisi nilai teks yang ada di dalam sebuah node. Untuk element node **P**, isinya adalah: **Sedang belajar JavaScript dari DuniaIlkom**. Dari daftar property sebelum ini, property `textContent` tidak diberi tanda *read only*, artinya nilai ini bisa diubah. Kita akan mempraktekkannya sesaat lagi.

Menggunakan seluruh property ini, kita bisa “berjalan” diantara node ke node untuk mencari node lain. Sebagai contoh, variabel `pNode` berisi node element **P**. Bisakah kita akses element **H1** dari element ini? Tentu saja.

Perhatikan bahwa element **H1** berada sebelum element **P**. Dengan demikian, saya bisa menggunakan perintah `pNode.previousSibling`. Sayangnya hasil dari `pNode.previousSibling` adalah sebuah **text node** yang berisi karakter carriage return. Ini karena tag `<p>` dengan tag `<h1>` ditulis terpisah di baris baru (tidak menyambung) sehingga terdapat karakter carriage return diantara keduanya.

Namun bagaimana dengan `pNode.previousSibling.previousSibling`? Berikut hasilnya:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10 <script>
11   var pNode = document.childNodes[1].childNodes[2].childNodes[3];
12   var h1Node = pNode.previousSibling.previousSibling;
13
14   console.log( h1Node.childNodes );
15 // NodeList [ #text "Belajar JavaScript" ]
16   console.log( h1Node.firstChild );           // #text "Belajar JavaScript"
17   console.log( h1Node.lastChild );          // #text "Belajar JavaScript"
18   console.log( h1Node.nextSibling );        // #text " "
19   console.log( h1Node.nodeName );          // H1
20   console.log( h1Node.nodeType );          // 1
21   console.log( h1Node.parentNode );        // <body>
22   console.log( h1Node.previousSibling );    // #text " "
23   console.log( h1Node.textContent );        // Belajar JavaScript
24 </script>
25 </body>
26 </html>
```

Yup, dengan menulis `pNode.previousSibling.previousSibling` kita sampai ke element H1. Property `previousSibling` perlu dipanggil 2 kali karena ada sebuah text node diantara tag `<h1>` dengan `<p>`.

Sebagai latihan, bisakah anda mengakses tag `<title>` dengan berangkat dari `pNode`? Perjalanan kita memang cukup panjang, karena harus naik dulu ke tag `<body>`, pindah ke tag `<head>`, kemudian baru turun ke tag `<title>`. Yang juga harus di waspadai adalah text node yang berisi karakter carriage return.

Baik, berikut kode yang saya gunakan:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
```

```

10 <script>
11   var pNode = document.childNodes[1].childNodes[2].childNodes[3];
12   var bodyNode = pNode.parentNode;
13   var headNode = bodyNode.previousSibling.previousSibling;
14   var titleNode = headNode.lastChild.previousSibling;
15
16   console.log( titleNode.childNodes );
17   // NodeList [ #text " Belajar JavaScript " ]
18   console.log( titleNode.firstChild );           // #text " Belajar JavaScript "
19   console.log( titleNode.lastChild );          // #text " Belajar JavaScript "
20   console.log( titleNode.nextSibling );         // #text " "
21   console.log( titleNode.nodeName );            // TITLE
22   console.log( titleNode.nodeType );             // 1
23   console.log( titleNode.parentNode );           // <head>
24   console.log( titleNode.previousSibling );      // #text " "
25   console.log( titleNode.textContent );          // Belajar JavaScript
26 </script>
27 </body>
28 </html>

```

Huff... perjalanan untuk mencari tag `<title>` ini cukup jauh. Tapi semoga anda bisa paham untuk apa tujuan setiap variabel bantu tersebut. Mencari element dengan cara seperti ini memang sangat ribet, apalagi kita harus memperhitungkan teks carriage return yang biasanya ada di antara dua buah element HTML. Apakah ada cara yang lebih mudah? tentu, dan itu akan menjadi materi untuk bab setelah ini.

## 19.8 Mengubah Text dari Node Object

Melihat daftar property yang ada di **node object**, terdapat property `textContent`. Property ini berisi teks dari sebuah **node object**. Menariknya, nilai `textContent` bisa diubah, yang artinya akan mengubah nilai teks dari element tersebut. Mari kita coba:

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta charset="utf-8">
5    <title> Belajar JavaScript </title>
6  </head>
7  <body>
8    <h1>Belajar JavaScript</h1>
9    <p>Sedang belajar <em>JavaScript</em><em><b> dari DuniaIlkom</b></em></p>
10   <script>
11     var pNode = document.childNodes[1].childNodes[2].childNodes[3];
12     console.log(pNode.textContent); //Sedang belajar JavaScript dari DuniaIlkom
13     pNode.textContent = "Berubah";

```

```

14  </script>
15 </body>
16 </html>

```

Disini saya mengakses element node P yang disimpan ke dalam variabel pNode. Kemudian isinya ditampilkan dengan perintah `console.log( pNode.textContent )` untuk memastikan kita berada di node yang benar.

Di baris terakhir, terdapat perintah `pNode.textContent = "Berubah"`, artinya saya ingin meng-input nilai teks "Berubah" ke dalam element node P. Bagaimana hasilnya?



Gambar: Teks yang ada di node element P telah “Berubah”

Perhatikan teks di bawah judul h1. Jika yang dijalankan adalah kode HTML, seharusnya yang tampil adalah teks "Sedang belajar JavaScript dari DuniaIlkom", sekarang sudah digantikan dengan teks "Berubah".

Selamat! Inilah pertama kalinya kita mengubah element HTML menggunakan JavaScript, dan property `textContent` hanyalah salah satu dari banyak cara lain. Nantinya kita juga bisa mengubah nilai yang lebih rumit, seperti atribut hingga style CSS.

Sebagai latihan, bagaimana jika mengubah isi teks element `<b> dari DuniaIlkom</b>` menjadi `<b> dari JavaScript Uncover</b>`? Caranya sama persis seperti sebelumnya, hanya saja kita perlu mencari dulu node element `<b>`:

```

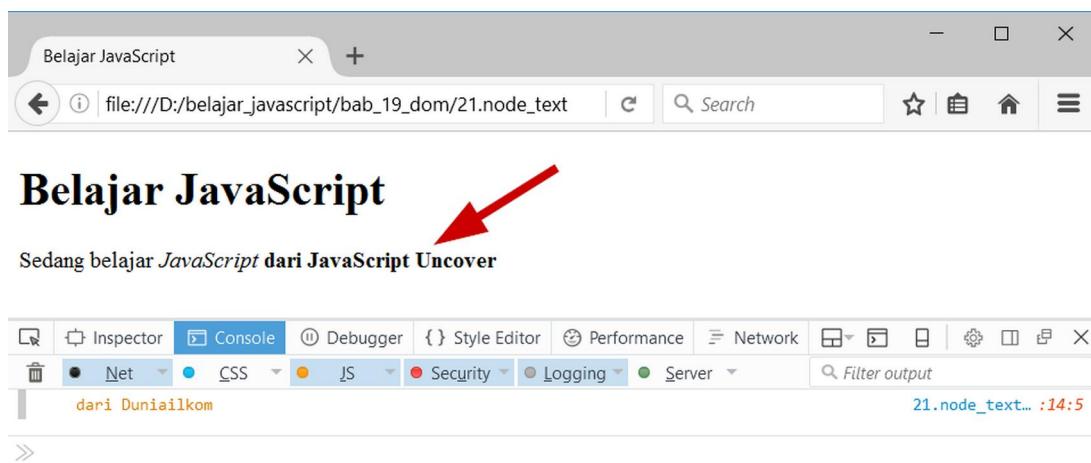
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta charset="utf-8">
5    <title> Belajar JavaScript </title>
6  </head>
7  <body>
8    <h1>Belajar JavaScript</h1>
9    <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10   <script>
11     var pNode = document.childNodes[1].childNodes[2].childNodes[3];

```

```

12     var bNode = pNode.lastChild;
13     console.log( bNode.textContent );           // dari DuniaIlkom
14     bNode.textContent = " dari JavaScript Uncover";
15   </script>
16 </body>
17 </html>

```



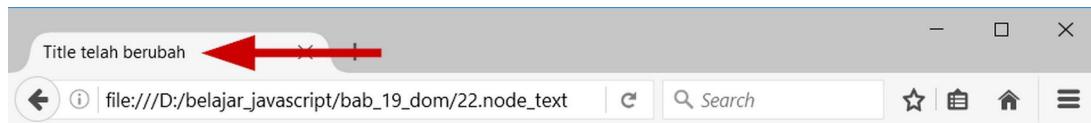
Gambar: Teks berubah menjadi “dari JavaScript Uncover”

Tidak hanya untuk element yang ada di `<body>`, kita juga bisa mengubah element yang ada di tag `<head>`. Bagaimana kalau tag `<title>`? Tidak masalah:

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta charset="utf-8">
5    <title> Belajar JavaScript </title>
6  </head>
7  <body>
8    <h1>Belajar JavaScript</h1>
9    <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10   <script>
11     var headNode = document.childNodes[1].childNodes[0];
12     var titleNode = headNode.childNodes[3];
13     titleNode.textContent = "Title telah berubah";
14   </script>
15 </body>
16 </html>

```



## Belajar JavaScript

Sedang belajar *JavaScript* dari **DuniaIlkom**

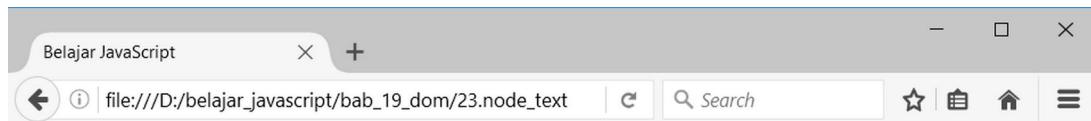
Gambar: Title halaman juga bisa diubah

Sekarang tampilan dari `title` juga telah berubah, sesuai dengan teks yang diinput kedalam property `textContent` dari element node `TITLE`.

Silahkan anda coba mengubah berbagai element HTML lain. Akses node yang dituju, kemudian input teks baru ke dalam property `textContent`. Atau bisa juga anda rancang sebuah halaman HTML lain untuk bermain-main dengan property `textContent`.

Terakhir, bagaimana jika kita input kode HTML ke dalam property `textContent`? Saya ingin mengubah isi teks tag `<h1>` dari "Belajar JavaScript" menjadi "<`emem`>". Artinya, selain teks yang berubah, teks tersebut juga akan tampil miring karena ada tambahan tag `<em>`. Mari kita coba:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10 <script>
11   var bodyNode = document.childNodes[1].childNodes[2];
12   var h1Node = bodyNode.childNodes[1];
13   h1Node.textContent = "<em>Belajar Node Object</em>";
14 </script>
15 </body>
16 </html>
```



Gambar: Kenapa tidak miring?

Ternyata yang ditampilkan adalah teks asli yang diinput. Tag `<em>` akan diproses sebagai teks, bukan sebagai element HTML. Inilah salah satu kekurangan jika menggunakan property `textContent` untuk mengubah isi sebuah element HTML. Bagaimana solusinya? Kita bisa memanfaatkan berbagai method dari node object.

## 19.9 Node Object Method

Node object memiliki berbagai method yang bisa digunakan terutama untuk memanipulasi DOM, seperti menambah node baru, menghapus node, mencopy node, dst. Berikut daftar method dari node object yang akan kita bahas:

- `Node.appendChild()`
- `Node.cloneNode()`
- `Node.contains()`
- `Node.getFeature()`
- `Node.hasChildNodes()`
- `Node.insertBefore()`
- `Node.removeChild()`
- `Node.replaceChild()`



Daftar lengkap dari method node object bisa dilihat ke [Node References<sup>4</sup>](#) - Mozilla Developer Network.

Sebelum membahas method-method ini, mari siapkan file latihan. Berikut kode yang akan saya pakai:

<sup>4</sup><https://developer.mozilla.org/en-US/docs/Web/API/Node>

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10 <script>
11   var bodyNode = document.childNodes[1].childNodes[2];
12   var h1Node = bodyNode.childNodes[1];
13   var pNode = bodyNode.childNodes[3];
14   var emNode = pNode.childNodes[1];
15   var bNode = pNode.childNodes[2];
16
17   console.log(bodyNode.nodeName);      // BODY
18   console.log(h1Node.nodeName);        // H1
19   console.log(pNode.nodeName);         // P
20   console.log(emNode.nodeName);        // EM
21   console.log(bNode.nodeName);         // B
22 </script>
23 </body>
24 </html>
```

Kode HTML yang dipakai masih sama seperti sebelumnya. Agar menghemat tempat, dalam contoh kode program selanjutnya struktur HTML tidak akan saya tampilkan lagi, kita akan fokus kepada kode JavaScript saja.

Di dalam kode JavaScript, saya mendefenisikan 5 variabel yang berisi beberapa node element. Semua variabel ini berfungsi sebagai ‘shorcut’ untuk menemukan element node <body>, <h1>, <p>, <em> dan <b>. Semuanya disimpan ke dalam variabel bodyNode, h1Node, pNode, emNode, dan bNode.

## 19.10 Method Document.createElement() dan Document.createTextNode()

Kedua method ini bukanlah kepunyaan node object, tapi milik **document object**. Bahasan lebih lengkap tentang **document object** akan kita bahas pada bab berikutnya. Method Document.createElement() dan Document.createTextNode() akan saya bahas sekarang karena dibutuhkan oleh method-method node object.

Method Document.createElement() digunakan untuk membuat **element node** baru. Sebagai contoh, untuk membuat sebuah element node <p>, perintahnya adalah:

```
var newPNode = document.createElement("p");
```

Disini, variabel `newPNode` akan berisi sebuah **node object** bertipe **element**. Element tersebut adalah P atau tag `<p>`. Apa isi dari element node ini? Secara bawaan masih kosong dan kita harus menginput sebuah text node ke dalamnya.

Untuk membuat sebuah text node, kita bisa menggunakan method `Document.createTextNode()`:

```
var newTextNode = document.createTextNode("Sebuah paragraf baru");
```

Variabel `newTextNode` akan berisi sebuah **node object** bertipe **text**, dimana isi teks adalah "Sebuah paragraf baru".

Berikut pembuktian jenis dari variabel `newPNode` dan `newTextNode`:

```
1 var newPNode = document.createElement("p");
2 console.log (newPNode.nodeName);           // P
3 console.log (newPNode.nodeType);          // 1
4 console.log (newPNode.nodeValue );         // null
5
6 var newTextNode = document.createTextNode("Sebuah paragraf baru");
7 console.log (newTextNode.nodeName);         // #text
8 console.log (newTextNode.nodeType);         // 3
9 console.log (newTextNode.nodeValue );        // Sebuah paragraf baru
```

Hasil dari `newPNode.nodeType = 1` dan `newPNode.nodeName = P`, artinya variabel `newPNode` berisi **element node P**. Sedangkan untuk `newTextNode.nodeType = 3` dan `newTextNode.nodeValue = "Sebuah paragraf baru"`, ini adalah ciri-ciri sebuah **text node**.

Kedua node ini harus disatukan agar bisa berfungsi, dimana `newTextNode` harus menjadi child dari `newPNode`. Ini kita lakukan menggunakan method dari **node object**: `Node.appendChild()`.

## 19.11 Method Node.appendChild()

Method **node object** pertama yang akan kita bahas adalah `appendChild()`. Fungsinya, untuk menginput sebuah node ke posisi terakhir dari node saat ini.

Sebagai contoh, untuk menginput variabel `newTextNode` yang beripe **text node** ke dalam **element node** `newPNode`, perintahnya adalah:

```
1 var newPNode = document.createElement("p");
2 var newTextNode = document.createTextNode("Sebuah paragraf baru");
3 newPNode.appendChild(newTextNode);
```

Hasil dari perintah diatas, element node newPNode akan berisi teks "Sebuah paragraf baru". Atau dengan kata lain, newPNode memiliki sebuah child node, yakni newTextNode.

Apa yang kita lakukan disini adalah membuat sebuah tag <p>Sebuah paragraf baru</p>, dimana alurnya bisa digambarkan sebagai berikut

- var newPNode = document.createElement("p"): buat sebuah tag <p>.
- var newTextNode = document.createTextNode("Sebuah paragraf baru"): buat sebuah teks "Sebuah paragraf baru".
- newPNode.appendChild(newTextNode): Input teks "Sebuah paragraf baru" ke dalam tag <p>, sehingga menghasilkan <p>Sebuah paragraf baru</p>.

Pekerjaan kita belum selesai, node object newPNode ini baru tersedia di dalam memory komputer, kita harus menginputnya ke dalam struktur DOM. Bagaimana caranya? Juga bisa menggunakan method appendChild(). Hanya saja kita akan menginput newPNode ke dalam element node BODY:

```
1 var bodyNode = document.childNodes[1].childNodes[2];
2 var h1Node = bodyNode.childNodes[1];
3 var pNode = bodyNode.childNodes[3];
4 var emNode = pNode.childNodes[1];
5 var bNode = pNode.childNodes[2];
6
7 var newPNode = document.createElement("p");
8 var newTextNode = document.createTextNode("Sebuah paragraf baru");
9 newPNode.appendChild(newTextNode);
10
11 bodyNode.appendChild(newPNode);
```

Setelah membuat element node P dan menginput teks, perintah bodyNode.appendChild(newPNode) akan menginput variabel newPNode ke dalam struktur DOM, tepatnya sebagai node terakhir dari tag <body>.

Berikut tampilannya:



## Belajar JavaScript

Sedang belajar *JavaScript* dari **DuniaIlkom**

Sebuah paragraf baru

Gambar: Penambahan sebuah paragraf baru menggunakan method appendChild()

Sebagai latihan, bisakah anda membuat sebuah element node SPAN, yakni tag <span>. Tag ini berisikan teks " dan JavaScript Uncover", lalu ditempatkan sebagai element terakhir dari tag <p>.

Hasil akhir yang saya inginkan, akan tampil teks: "Sedang belajar JavaScript dari DuniaIlkom dan JavaScript Uncover". Jika dilihat dari strukur HTML, paragraf tersebut akan menjadi seperti ini:

```
1 <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b>
2 <span> dan JavaScript Uncover</span></p>
```



## Belajar JavaScript

Sedang belajar *JavaScript* dari **DuniaIlkom** dan *JavaScript Uncover*

Gambar: Penambahan teks baru di dalam tag paragraf

Silahkan anda coba sebentar, caranya sama seperti sebelumnya, hanya saja sekarang kita membuat element span lalu menginputnya ke dalam tag <p>.

Baik, berikut kode yang saya gunakan:

```
1 var bodyNode = document.childNodes[1].childNodes[2];
2 var h1Node = bodyNode.childNodes[1];
3 var pNode = bodyNode.childNodes[3];
4 var emNode = pNode.childNodes[1];
5 var bNode = pNode.childNodes[2];
6
7 var newSpanNode = document.createElement("span");
8 var newTextNode = document.createTextNode(" dan JavaScript Uncover");
9 newSpanNode.appendChild(newTextNode);
10
11 pNode.appendChild(newSpanNode);
```

Variabel newSpanNode akan berisi sebuah element node span. Variabel newTextNode akan berisi teks " dan JavaScript Uncover". Kedua node ini disatukan dengan perintah newSpanNode.appendChild(newTextNode).

Perintah pNode.appendChild(newSpanNode) digunakan untuk menambah newSpanNode ke posisi terakhir di dalam tag <p>.

Sekarang, bagaimana jika kita ingin menambah element baru di posisi awal? Ini bisa dilakukan dengan method insertBefore().

## 19.12 Method Node.insertBefore()

Jika method `appendChild()` berguna untuk menginput node baru ke posisi terakhir (sebagai last child), maka method `insertBefore()` digunakan untuk menginput node baru di posisi tertentu.

Berbeda dengan `appendChild()`, method `insertBefore()` membutuhkan 2 argumen: node baru yang ingin ditambahkan, dan node patokan. Node baru akan diinput pada posisi **sebelum** node patokan. Penjelasannya memang cukup *njelitemet*, mari langsung kita lihat contoh penerapannya:

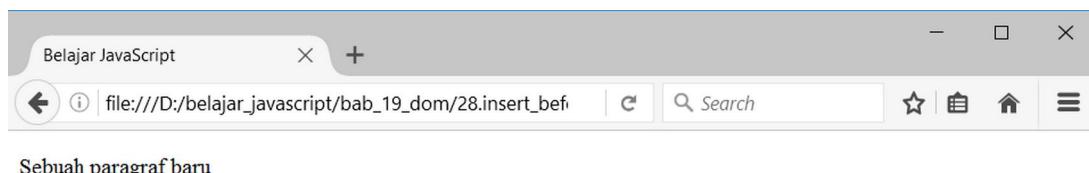
```

1 var bodyNode = document.childNodes[1].childNodes[2];
2 var h1Node = bodyNode.childNodes[1];
3 var pNode = bodyNode.childNodes[3];
4 var emNode = pNode.childNodes[1];
5 var bNode = pNode.childNodes[2];
6
7 var newPNode = document.createElement("p");
8 var newTextNode = document.createTextNode("Sebuah paragraf baru");
9 newPNode.appendChild(newTextNode);
10
11 bodyNode.insertBefore(newPNode, h1Node);

```

Disini saya kembali membuat `newPNode` yang berisi kode `<p>Sebuah paragraf baru</p>`. Cara pembuatannya tetap menggunakan method `appendChild()`, karena `newTextNode` akan dilekatkan sebagai child dari `newPNode`.

Perintah `bodyNode.insertBefore(newPNode, h1Node)` artinya: “input `newPNode` sebagai child dari `bodyNode`, di posisi sebelum `h1Node`”. Bagaimana hasilnya?



Sebuah paragraf baru

## Belajar JavaScript

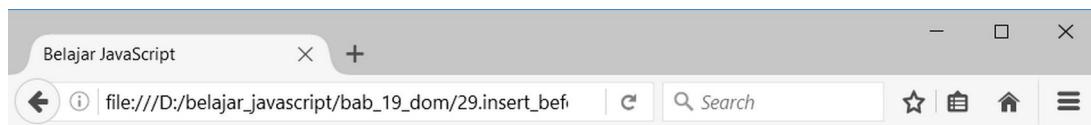
Sedang belajar *JavaScript* dari **Duniailkom**

Gambar: Paragraf baru ditambahkan sebelum tag `<h1>`

Kembali, sebagai latihan saya ingin menantang anda untuk menambahkan sebuah node element **DEL** pada posisi sebelum tag `<em>` di dalam paragraf `<p>`. Isi teks dari node element **DEL** berupa sebuah teks "PHP". Sekedar mengingatkan, element **DEL** atau tag `<del>` berfungsi untuk membuat teks tercoret.

Hasil akhir dari tag `<p>` adalah sebagai berikut:

```
1 <p>Sedang belajar <del>PHP </del><em>JavaScript</em><b> dari DuniaIlkom</b>
2 </p>
```



## Belajar JavaScript

Sedang belajar *PHP-JavaScript* dari **DuniaIlkom**

Gambar: Tag ~~PHP~~ sukses ditambah ke dalam tag *p*

Baik, berikut kode yang saya gunakan:

```
1 var bodyNode = document.childNodes[1].childNodes[2];
2 var h1Node = bodyNode.childNodes[1];
3 var pNode = bodyNode.childNodes[3];
4 var emNode = pNode.childNodes[1];
5 var bNode = pNode.childNodes[2];
6
7 var newDelNode = document.createElement("del");
8 var newTextNode = document.createTextNode("PHP ");
9 newDelNode.appendChild(newTextNode);
10
11 pNode.insertBefore(newDelNode, emNode);
```

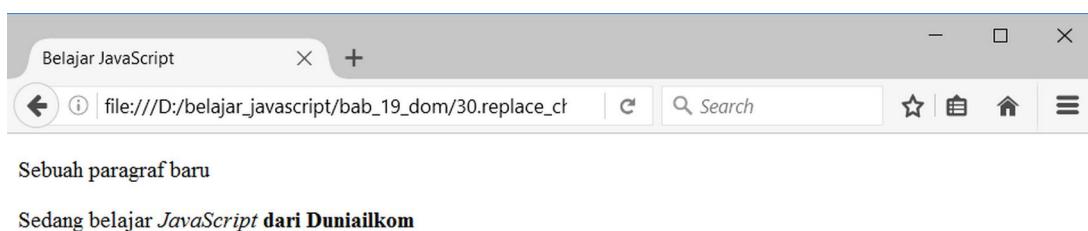
Perintah `pNode.insertBefore(newDelNode, emNode)` artinya: “input node element `newDelNode` sebagai child dari `pNode`, di posisi sebelum `emNode`”.

### 19.13 Method Node.replaceChild()

Method `replaceChild()` digunakan untuk mengganti sebuah node dengan node lain. Method ini membutuhkan 2 argumen: node baru yang akan ditambahkan dan node yang akan dihapus.

Berikut contoh penggunaannya:

```
1 var bodyNode = document.childNodes[1].childNodes[2];
2 var h1Node = bodyNode.childNodes[1];
3 var pNode = bodyNode.childNodes[3];
4 var emNode = pNode.childNodes[1];
5 var bNode = pNode.childNodes[2];
6
7 var newPNode = document.createElement("p");
8 var newTextNode = document.createTextNode("Sebuah paragraf baru");
9 newPNode.appendChild(newTextNode);
10
11 bodyNode.replaceChild(newPNode, h1Node);
```

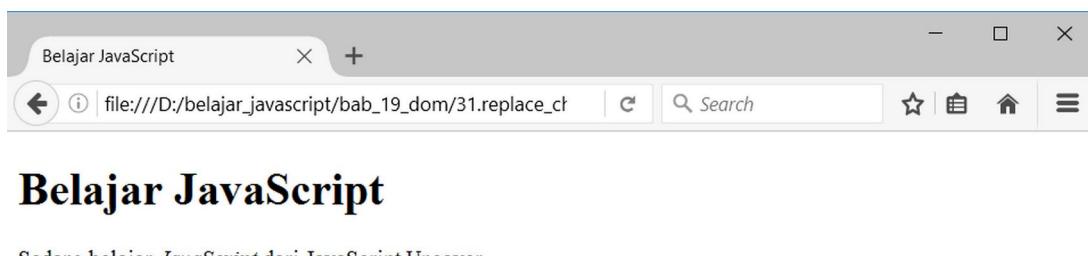


Gambar: Tag <h1> telah diganti dengan tag <p>

Perintah `bodyNode.replaceChild(newPNode, h1Node)` artinya: "ganti h1Node dengan newPNode yang keduanya merupakan child dari bodyNode". Terlihat dari tampilan, tag <h1> atau h1Node telah terhapus dan digantikan oleh tag <p> atau newPNode.

Sebagai latihan, saya ingin membuat sebuah element U atau tag <u> dengan isi teks " dari JavaScript Uncover". Element ini akan menggantikan tag <b> dari DuniaIlkom</b> dari paragraf P.

Hasil akhir yang akan tampil:



Gambar: Teks " dari DuniaIlkom" digantikan dengan " dari JavaScript Uncover"

Baik, berikut kode program yang saya gunakan:

```

1 var bodyNode = document.childNodes[1].childNodes[2];
2 var h1Node = bodyNode.childNodes[1];
3 var pNode = bodyNode.childNodes[3];
4 var emNode = pNode.childNodes[1];
5 var bNode = pNode.childNodes[2];
6
7 var newUNode = document.createElement("u");
8 var newTextNode = document.createTextNode(" dari JavaScript Uncover");
9 newUNode.appendChild(newTextNode);
10
11 pNode.replaceChild(newUNode, bNode);

```

Perintah `pNode.replaceChild(newUNode, bNode)` artinya: “Ganti element `bNode` dengan `newUNode` yang sama-sama menjadi child dari `pNode`”.

## 19.14 Method Node.removeChild()

Method `removeChild()` digunakan untuk menghapus sebuah element dari DOM tree. Kita tinggal menginput node yang ingin dihapus sebagai argumen method ini.

Berikut contoh penggunaan method `removeChild()`:

```

1 var bodyNode = document.childNodes[1].childNodes[2];
2 var h1Node = bodyNode.childNodes[1];
3 var pNode = bodyNode.childNodes[3];
4 var emNode = pNode.childNodes[1];
5 var bNode = pNode.childNodes[2];
6
7 pNode.removeChild(bNode);

```

Perintah `pNode.removeChild(bNode)` artinya saya ingin menghapus `bNode` atau `<b>` yang menjadi child dari `pNode`. Hasilnya, element `bNode` terhapus, termasuk teks "dari DuniaIlkom" yang ada di dalam tag `<b>`.



## Belajar JavaScript

Sedang belajar *JavaScript*

Gambar: Element HTML `<b> dari DuniaIlkom</b>` telah dihapus

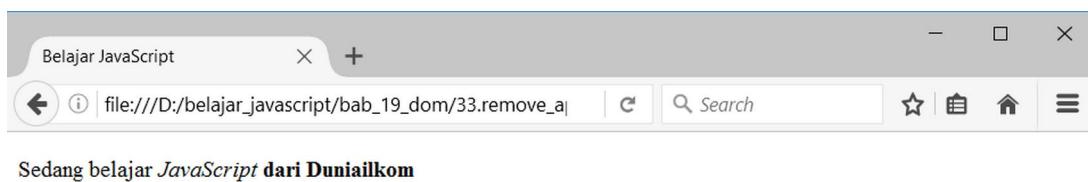
Method `removeChild()` sebenarnya juga mengembalikan sebuah nilai, yakni node yang telah dihapus tersebut. Node ini nantinya bisa diproses kembali, misalnya diinput ke tempat lain.

```

1 var bodyNode = document.childNodes[1].childNodes[2];
2 var h1Node = bodyNode.childNodes[1];
3 var pNode = bodyNode.childNodes[3];
4 var emNode = pNode.childNodes[1];
5 var bNode = pNode.childNodes[2];
6
7 var deletedNode = bodyNode.removeChild(h1Node);
8 bodyNode.appendChild(deletedNode);

```

Variabel `deletedNode` berisi `h1Node` yang telah terhapus. Variabel `deletedNode` ini kemudian saya tambah kembali ke dalam `bodyNode` menggunakan perintah `bodyNode.appendChild(deletedNode)`. Artinya tag `<h1>` yang sebelumnya dihapus, diinput kembali sebagai last child dari tag `<body>`:



Gambar: Tag `<h1>` dihapus, kemudian diinput kembali di bawah tag `<p>`

## 19.15 Method Node.cloneNode()

Method `cloneNode()` digunakan untuk men-copy sebuah node. Method ini membutuhkan 1 argumen berupa nilai boolean. Jika diinput `true`, seluruh child dari node tersebut akan ikut dicopy. Jika diinput `false`, hanya node itu saja yang dicopy (child node tidak ikut).

Berikut contoh penggunannya:

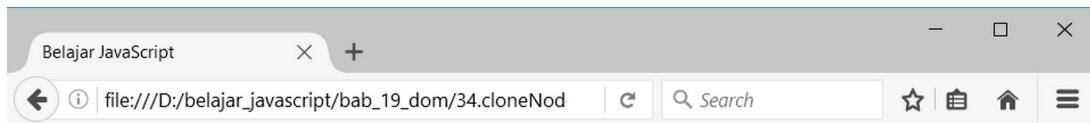
```

1 var bodyNode = document.childNodes[1].childNodes[2];
2 var h1Node = bodyNode.childNodes[1];
3 var pNode = bodyNode.childNodes[3];
4 var emNode = pNode.childNodes[1];
5 var bNode = pNode.childNodes[2];
6
7 var newH1Node = h1Node.cloneNode(true);
8 bodyNode.appendChild(newH1Node);

```

Perintah `var newH1Node = h1Node.cloneNode(true)` artinya saya ingin mencopy isi dari `h1Node`, lengkap dengan child dari node tersebut. Hasilnya disimpan ke dalam variabel `newH1Node`.

Variabel `newH1Node` kemudian saya input ke dalam struktur DOM sebagai last child dari tag `<body>`. Ini dilakukan dari perintah `bodyNode.appendChild(newH1Node)`.



## Belajar JavaScript

Sedang belajar *JavaScript* dari **DuniaIlkom**

## Belajar JavaScript

Gambar: Tag <h1> di cloning, kemudian di input kembali ke dalam tag <body>

### 19.16 Method Node.contains()

Method `contains()` digunakan untuk memeriksa apakah sebuah node memiliki child node tertentu. Node yang ingin diperiksa diinput sebagai argument dari method `contains()`. Jika node tersebut ada, hasilnya `true`. Jika tidak ditemukan, hasilnya `false`.

Berikut contoh penggunaannya:

```
1 var bodyNode = document.childNodes[1].childNodes[2];
2 var h1Node = bodyNode.childNodes[1];
3 var pNode = bodyNode.childNodes[3];
4 var emNode = pNode.childNodes[1];
5 var bNode = pNode.childNodes[2];
6
7 var cekNode1 = pNode.contains(emNode);
8 console.log( cekNode1 );      // true
9
10 var cekNode2 = pNode.contains(bNode);
11 console.log( cekNode2 );     // true
12
13 var cekNode3 = bodyNode.contains(h1Node);
14 console.log( cekNode3 );     // true
15
16 var cekNode4 = bodyNode.contains(bNode);
17 console.log( cekNode4 );     // true
18
19 var cekNode5 = pNode.contains(h1Node);
20 console.log( cekNode5 );     // false
```

Dalam program ini, saya mengecek berbagai node. Perintah `pNode.contains(emNode)` artinya apakah di dalam tag <p> terdapat <em>? Benar, sehingga hasilnya `true`.

Method `contains()` juga mencari ke dalam seluruh child node, termasuk children dari child node. Perintah `bodyNode.contains(bNode)` hasilnya `true`, padahal tag `<b>` bukan berada setelah tag `<body>`, tapi merupakan cucu dari tag `<body>`.

Perintah `pNode.contains(h1Node)` akan menghasilkan `false` karena memang tidak ditemukan node `H1` di dalam paragraf.

## 19.17 Method Node.hasChildNodes()

Method `hasChildNodes()` digunakan untuk memeriksa apakah sebuah node memiliki anak (child) atau tidak. Hasilnya `true` jika terdapat child (walaupun 1 node), dan `false` jika tidak punya child. Method ini tidak memerlukan argumen.

Berikut contohnya:

```
1 var bodyNode = document.childNodes[1].childNodes[2];
2 var h1Node = bodyNode.childNodes[1];
3 var pNode = bodyNode.childNodes[3];
4 var emNode = pNode.childNodes[1];
5 var bNode = pNode.childNodes[2];
6
7 var cekNode1 = pNode.hasChildNodes();
8 console.log( cekNode1 );      // true
9
10 var cekNode2 = bodyNode.hasChildNodes();
11 console.log( cekNode2 );      // true
12
13 var cekNode3 = bNode.hasChildNodes();
14 console.log( cekNode3 );      // true
15
16 var cekNode4 = bNode.firstChild.hasChildNodes();
17 console.log( cekNode4 );      // false
```

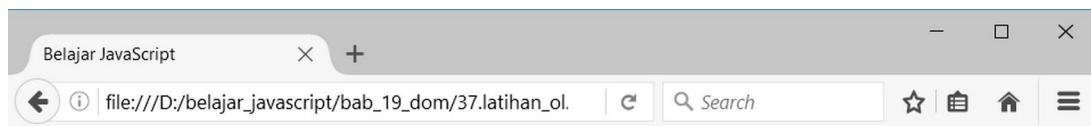
Secara umum, hampir setiap element node memiliki child, kecuali element node yang tidak butuh teks seperti `<br>` atau `<hr>`.

Sedangkan text node memang tidak bisa memiliki child, inilah yang coba saya periksa dari perintah `bNode.firstChild.hasChildNodes()`. Anak pertama dari tag `<b>` adalah sebuah text node, dan text node ini tidak memiliki child.

## 19.18 Case Study: Latihan Node Object

Menutup pembahasan tentang Node Object, saya ingin membuat 2 buah case study yang akan menguji pemahaman anda tentang node object, DOM, dan materi seputar JavaScript.

Yang pertama, saya ingin menambahkan sebuah **ordered list** ke dalam struktur DOM. Di dalam HTML, ini biasa dibuat menggunakan tag `<ol>` dan `<li>`. Berikut hasil akhir yang saya inginkan:



## Belajar JavaScript

1. HTML
2. CSS
3. JavaScript

Sedang belajar *JavaScript* dari **DuniaIlkom**

Gambar: Menambahkan sebuah list ke dalam DOM

Struktur HTML untuk latihan ini sama seperti sebelumnya, yakni sebuah tag `<h1>`, dan sebuah paragraf `<p>`. Tiga list yang berisi HTML, CSS dan JavaScript digenerate menggunakan JavaScript. Sebagai bantuan bagi yang lupa-lupa ingat tentang HTML, untuk membuat ordered list kita menggunakan kode berikut:

```
1 <ol>
2   <li>HTML</li>
3   <li>CSS</li>
4   <li>JavaScript</li>
5 </ol>
```

Untuk membuatnya dari JavaScript, langkah-langkah yang dibutuhkan adalah sebagai berikut:

1. Buat 1 element node OL.
2. Buat 3 element node LI.
3. Buat 3 text node yang berisi teks: "HTML", "CSS", dan "JavaScript".
4. Input text node ke dalam setiap element LI, yakni jadikan text node sebagai child dari element node LI.
5. Input ketiga node element LI ke dalam OL.
6. Input OL kedalam struktur DOM, tempatkan sebelum tag `<p>`.

Beberapa langkah juga bisa digabung, misalnya proses pembuatan text node bisa digabung dengan menginputnya langsung ke element node LI.

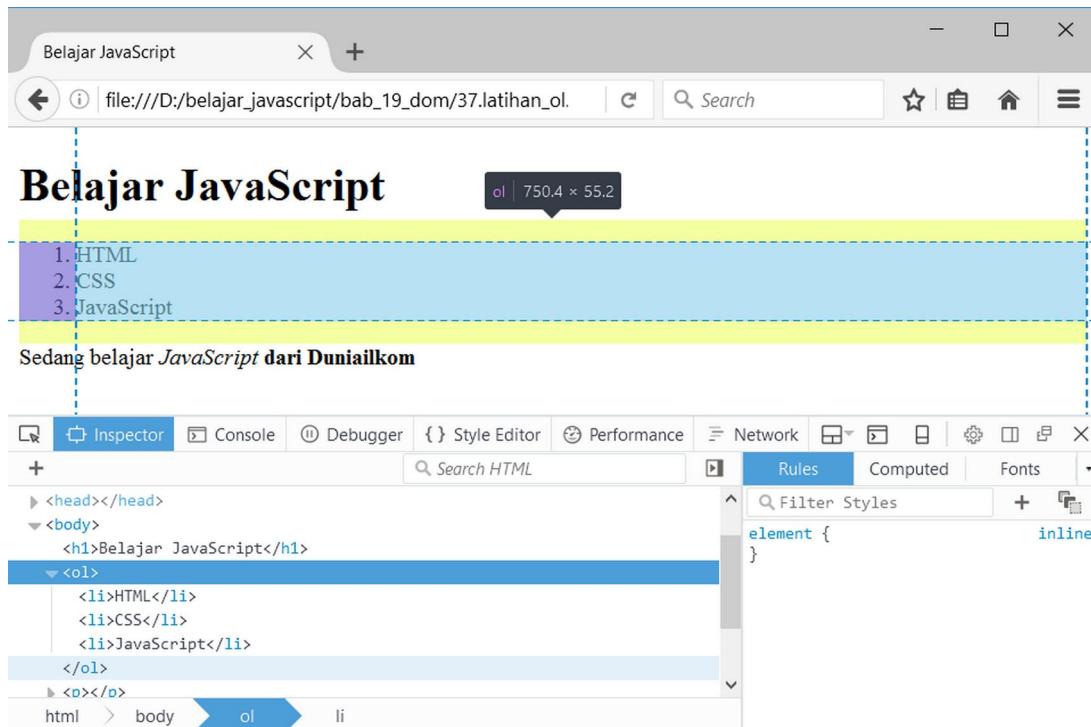
Silahkan anda coba rancang kode programnya. Tidak masalah jika harus mempelajari lagi ke awal bab.

Baik, berikut kode program yang saya buat:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10 <script>
11   // Siapkan beberapa variabel shortcut untuk Node
12   var bodyNode = document.childNodes[1].childNodes[2];
13   var h1Node = bodyNode.childNodes[1];
14   var pNode = bodyNode.childNodes[3];
15   var emNode = pNode.childNodes[1];
16   var bNode = pNode.childNodes[2];
17
18   // Buat node element OL, yakni tag <ol>
19   var olNode = document.createElement("ol");
20
21   // Buat 3 node element LI, yakni 3 buah tag <li>;
22   var liNode1 = document.createElement("li");
23   var liNode2 = document.createElement("li");
24   var liNode3 = document.createElement("li");
25
26   // buat text node dan langsung input ke tiap LI
27   liNode1.appendChild(document.createTextNode("HTML"));
28   liNode2.appendChild(document.createTextNode("CSS"));
29   liNode3.appendChild(document.createTextNode("JavaScript"));
30
31   // tambahkan semua tag <li> kedalam tag <ol>
32   olNode.appendChild(liNode1);
33   olNode.appendChild(liNode2);
34   olNode.appendChild(liNode3);
35
36   // masukkan tag <ol> kedalam DOM tree, sebelum tag <p>
37   bodyNode.insertBefore(olNode,pNode);
38 </script>
39 </body>
40 </html>
```

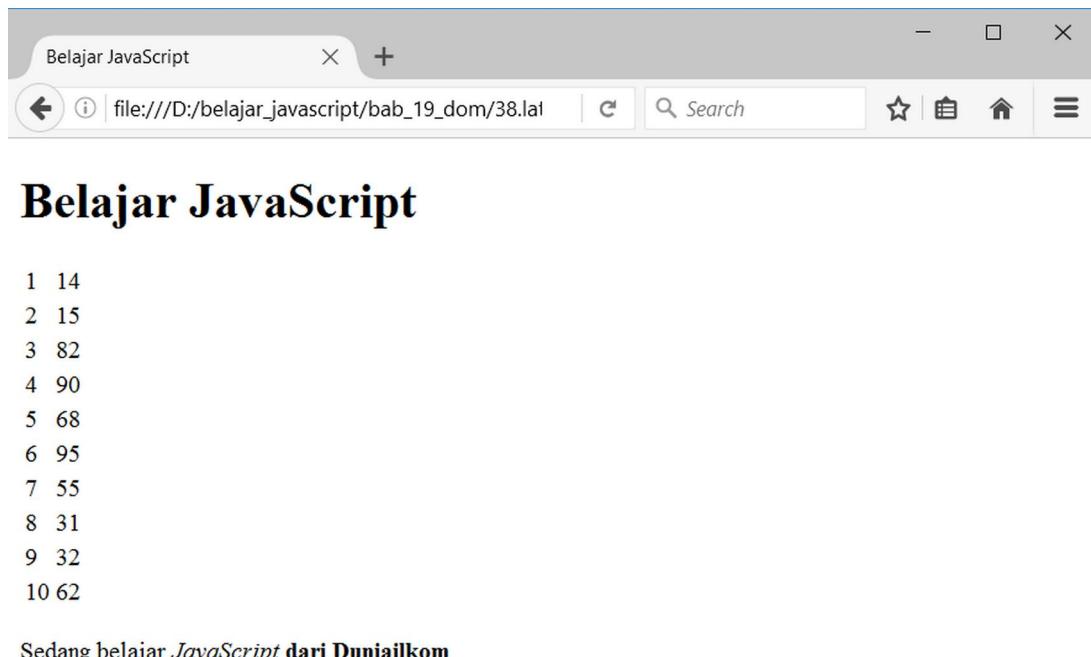
Tidak ada hal yang baru disini. Semuanya sudah kita bahas. Dan tentu saja anda bisa menggunakan nama variabel maupun urutan yang berbeda. Selama hasilnya terdapat sebuah list dan digenerate dari JavaScript, hal tersebut bisa dibenarkan.

Anda juga bisa menggunakan tab **Inspector** dari Web Developer Tools untuk memastikan struktur HTML sudah di generate dengan benar:



Gambar: Memeriksa struktur HTML dengan Web Developer Tools

Latihan kedua mirip seperti latihan pertama. Kali ini saya ingin membuat sebuah tabel yang digenerate dengan JavaScript. Tabel ini berisi dua kolom, yang pertama nomor urut, yang kedua angka acak antara 10 - 99. Jumlah baris tabel harus 10, yang tampilannya seperti berikut ini:



Gambar: Membuat tabel berisi angka acak dengan JavaScript

Kolom tersebut tidak terlihat seperti tabel karena kita belum belajar tentang cara menambah atribut atau kode CSS ke dalam node element (akan saya bahas dalam bab selanjutnya).

Struktur tabel juga lebih rumit daripada list. Setidaknya butuh 3 element HTML untuk membuat 1 tabel HTML: <table>, <tr>, dan <td>. Jika tabel ini dibuat menggunakan HTML, strukturnya adalah sebagai berikut:

```
1 <table>
2 <tr>
3   <td>1.</td><td>67</td>
4 </tr>
5 <tr>
6   <td>2.</td><td>23</td>
7 </tr>
8 <tr>
9   <td>3.</td><td>12</td>
10 </tr>
11 <tr>
12   <td>4.</td><td>99</td>
13 </tr>
14 </table>
```

Sekilas anda mungkin protes, untuk membuat 4 baris tabel saja kita butuh 13 element node (setiap baris butuh 2 tag <td> dan 1 <tr>) belum termasuk text node, apalagi jika sampai 10 baris.

Solusinya, bagaimana jika menggunakan perulangan? Struktur tabel untuk setiap baris sama persis, yang berbeda hanya isi teks saja. Dan apakah isi teks ini memungkinkan untuk dibuat dalam perulangan?

Kolom pertama berisi angka nomor urut, ini tidak menjadi masalah di dalam perulangan, karena kita cukup menampilkan isi dari variabel counter. Kolom kedua berisi angka acak, inipun tidak menjadi masalah, kita bisa menggenerate angka acak baru dalam setiap perulangan.

Dengan demikian langkah pembuatannya adalah:

1. Buat 1 element node TABLE.
2. Buat 1 element node TR.
3. Buat 2 element node TD.
4. Buat text node nomor urut, input ke element TD pertama.
5. Buat text node angka acak, input ke element TD kedua.
6. Input kedua node TD ke dalam TR.
7. Input node TD ke dalam TABLE.
8. Ulangi langkah 2 - 7 untuk setiap baris tabel.
9. Input TABLE kedalam struktur DOM, tempatkan sebelum tag <p>.

Baik, berikut kode program yang saya gunakan:

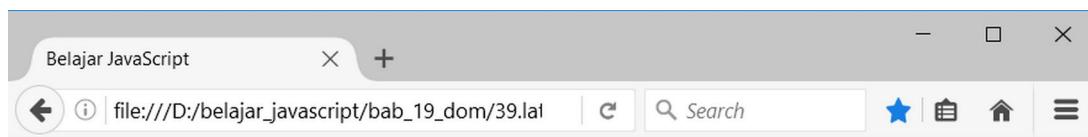
```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10 <script>
11   // Siapkan beberapa variabel shortcut untuk Node
12   var bodyNode = document.childNodes[1].childNodes[2];
13   var h1Node = bodyNode.childNodes[1];
14   var pNode = bodyNode.childNodes[3];
15   var emNode = pNode.childNodes[1];
16   var bNode = pNode.childNodes[2];
17
18   // Buat tag <table>
19   var tableNode = document.createElement("table");
20
21   // siapkan beberapa variabel untuk looping
22   var i, trNode, tdNode1, tdNode2, textNo, randomNumber, textRandom;
23   for (i = 1; i <= 10; i++){
24     // buat 1 tag <tr> dan 2 tag <td>
25     trNode = document.createElement("tr");
26     tdNode1 = document.createElement("td");
27     tdNode2 = document.createElement("td");
28
29     // buat text node untuk nomor urut
30     textNo = document.createTextNode(i+" . ");
31
32     // buat text node untuk angka acak
33     randomNumber = Math.floor(Math.random() * (90)) + 10;
34     textRandom = document.createTextNode(randomNumber);
35
36     // rangkai text node dengan element node <td>
37     tdNode1.appendChild(textNo);
38     tdNode2.appendChild(textRandom);
39
40     // input 2 tag <td> ke dalam tag <tr>
41     trNode.appendChild(tdNode1);
42     trNode.appendChild(tdNode2);
43
44     // input tag <tr> kedalam table
45     tableNode.appendChild(trNode);
46 }
```

```
47
48     // masukkan tag <table> kedalam DOM tree, sebelum tag <p>
49     bodyNode.insertBefore(tableNode,pNode);
50 </script>
51 </body>
52 </html>
```

Silahkan anda pelajari secara perlahan setiap baris kode program diatas. Langkah-langkah yang digunakan sama seperti yang kita bahas sepanjang bab ini, plus tambahan tentang looping (perulangan) dan pembuatan angka acak menggunakan method `Math.random()`.

Anda juga bisa menambahkan fitur-fitur baru ke dalam kode diatas. Misalnya bagaimana kalau ditambah dengan judul kolom tabel? Atau bisakah anda menampilkan hingga 1000 baris? Bagaimana jika 10 tabel dengan masing-masing tabel berisi 10 baris? Silahkan berkreasi dan uji pemahaman anda tentang **node object**, **DOM**, dan **JavaScript**.

Sebagai bonus tambahan, bagaimana jika asah ulang skill CSS dengan menambahkan beberapa style?



1	53
2	44
3	99
4	35
5	85
6	68
7	21
8	90
9	78
10	13

Sedang belajar *JavaScript* dari **Duniailkom**

Gambar: Tabel di style menggunakan CSS

Tampilan ini dihasilkan dengan sedikit tambahan kode CSS di bagian `<head>`:

```
1 <style>
2   table {
3     border-collapse:collapse;
4     border-spacing:0;
5     border:1px black solid;
6   }
7   td {
8     padding:8px 15px;
9     border:1px black solid;
10  }
11  tr:nth-child(2n) {
12    background-color: #F2F2F2;
13  }
14 </style>
```

---

Dalam bab ini kita telah membahas banyak hal seputar **DOM** dan **node object**. Pemahaman yang kuat tentang node object sangat dibutuhkan sepanjang anda mempelajari JavaScript di web browser.

Berikutnya, kita akan membahas lebih jauh tentang document object dan element object.

# 20. Document dan Element Object

Dalam bab sebelumnya kita telah berkenalan dengan struktur DOM serta **node object**. Bab kali ini masih berkaitan dengan DOM, dimana kita akan membahas lebih lanjut tentang **document object** dan **element object**.

Berbagai property dan method yang tersedia dalam kedua object ini, bisa digunakan untuk mengeksplorasi lebih jauh cara mengubah element atau tag HTML, terutama yang berkaitan dengan atribut dan style CSS.

## 20.1 DOM Level

Sebelum kita masuk ke document dan element object, mari bahas sejenak tentang **DOM Level**, atau versi-versi DOM.

Sama seperti perkembangan teknologi lainnya, DOM juga memiliki tahap atau level. Setiap peningkatan level menandakan perkembangan yang lebih baru dari sebelumnya. ECMAScript 6 memiliki fitur yang lebih banyak daripada ECMAScript 5, begitu juga dengan DOM yang memiliki level 0, 1, 2, 3 dan 4.

### DOM Level 0

**DOM Level 0** sebenarnya tidak ada, tapi ini digunakan untuk menyebut fitur yang disediakan oleh web browser sebelum istilah DOM hadir. DOM Level 0 diperkenalkan sejak kehadiran JavaScript di Netscape 2.0.

Sebagai contoh, untuk mencari seluruh gambar atau tag `<img>` di sebuah dokumen, bisa menggunakan perintah `document.images[]`. Hasilnya berupa *collection* (array) dari seluruh tag `<img>`. Untuk mencari seluruh link di halaman HTML, bisa di dapat dari perintah `document.links[]` yang akan menghasilkan *collection* dari tag `<a>`.

Walaupun sudah ‘jadul’, cara pengaksesan seperti ini masih didukung oleh mayoritas web browser modern.

### DOM Level 1

**DOM Level 1** adalah versi pertama dari standar DOM yang dikeluarkan **W3C**. Pada saat itu DOM dipecah menjadi 2 modul. Modul pertama mendefenisikan inti DOM yang berlaku untuk HTML dan XML, sedangkan modul kedua khusus ditujukan untuk HTML.

Beberapa object yang diperkenalkan dalam DOM level 1 adalah **Document**, **Node**, **Attr**, **Element**, dan **Text**.

## DOM Level 2

DOM Level 2 hadir di tahun 2000 untuk melengkapi beberapa method, property, dan event baru. Sebagai contoh, method `getElementsByClassName()` dikembangkan pada DOM level 2. Dukungan untuk CSS juga ditambahkan.

Pada level ini, modul terbaru dipecah menjadi beberapa jenis: The DOM2 Core, Views, Events, Style, Traversal dan Range, serta the DOM2 HTML.

## DOM Level 3

DOM Level 3 dikembangkan pada tahun 2004. DOM level ini menambahkan fitur-fitur lanjutan, yang terdiri dari 5 jenis modul: The DOM3 Core, Load and Save, Validation, Events, dan XPath.

## DOM Level 4

DOM Level 4 dikembangkan pada tahun 2015. DOM ini menjadi bagian dari **WHATWG living standard**. Artinya, standar DOM Level 4 akan terus dikembangkan dan diimplementasikan ke dalam web browser tanpa menunggu standar tersebut komplit. Tujuannya agar fitur-fitur terbaru bisa hadir dengan lebih cepat.

### Apa pengaruh dari Level DOM ini?

Pengaruhnya ada di ketersediaan fitur yang didukung oleh web browser. Sama seperti standar HTML dan CSS, spesifikasi tentang DOM hanyalah “rekomendasi” ke perusahaan pembuat web browser dan bukanlah suatu kewajiban.

Akibatnya, bisa saja terdapat web browser yang sudah mendukung DOM level terbaru, sedangkan web browser lain belum menyediakannya.

Sebagai web programmer, sebaiknya kita hanya menggunakan fitur yang berjalan di mayoritas web browser, untuk menghindari kemungkinan kode program gagal berjalan di web browser yang relatif tua.

Mayoritas web browser yang beredar saat ini sudah mendukung penuh DOM Level 2. DOM Level 3 masih di dukung sebagian, sedangkan untuk fitur terbaru yang ada di DOM level 4, kita mungkin harus menunggu beberapa tahun agar mayoritas web browser mengimplementasikan fitur terbaru ini.

Jika sebelumnya anda sudah belajar CSS3, bisa melihat bahwa beberapa property CSS3 terbaru ada yang berjalan sempurna di satu web browser, dan ada yang belum di dukung oleh sebagian web browser lain. Begitu juga yang terjadi pada DOM.

## 20.2 Referensi Property dan Method DOM

Object penyusun DOM berjumlah sangat banyak. Sebagian besar tidak akan saya bahas dalam buku ini karena kita hanya fokus kepada object, property dan method yang paling sering digunakan saja.

Jika anda tertarik mempelajari object lainnya, bisa mengakses [Mozilla Developer Network](#)<sup>1</sup>.

Pada pembahasan property atau method, dibagian akhir akan ditampilkan versi DOM Level serta web browser yang sudah mendukungnya.

## Specification

Specification	Status	Comment
<a href="#">Document Object Model (DOM) Level 1 Specification</a> The definition of 'getElementById' in that specification.	<span style="background-color: #00AEEF; color: white; padding: 2px 5px;">REC</span> Recommendation	Initial definition for the interface
<a href="#">Document Object Model (DOM) Level 2 Core Specification</a> The definition of 'getElementById' in that specification.	<span style="background-color: #00AEEF; color: white; padding: 2px 5px;">REC</span> Recommendation	Supersede DOM 1
<a href="#">Document Object Model (DOM) Level 3 Core Specification</a> The definition of 'getElementById' in that specification.	<span style="background-color: #00AEEF; color: white; padding: 2px 5px;">REC</span> Recommendation	Supersede DOM 2
<a href="#">DOM</a> The definition of 'getElementById' in that specification.	<span style="background-color: #00AEEF; color: white; padding: 2px 5px;">LS</span> Living Standard	Intend to supersede DOM 3

## Browser compatibility

	Desktop	Mobile						
Feature	Chrome	Edge	Firefox (Gecko)	Internet Explorer	Opera	Safari (WebKit)		
Basic support	1.0	(Yes)	1.0 (1.7 or earlier)	5.5	7.0	1.0		

Gambar: Versi DOM dan dukungan web browser untuk method `document.getElementById()`

Gambar diatas adalah potongan dari penjelasan method `document.getElementById()`<sup>2</sup> dari [Mozilla Developer Network](#).

Disini terlihat bahwa method `document.getElementById()` awalnya berasal dari DOM Level 1, walaupun fiturnya baru dikembangkan di DOM Level 2. Selain itu method ini sudah tersedia di seluruh web browser modern.

## 20.3 Mencari Element Node

Dalam bab sebelumnya kita telah membahas cara mencari element node dengan menelusuri DOM tree. Sebagai contoh untuk mencari sebuah tag `<h1>`, kita harus mulai dari **document object**, masuk ke tag `<html>`, ke tag `<body>`, baru ketemu tag `<h1>`. Ini adalah “cara susah” untuk menemukan sebuah element node.

Sekarang saatnya kita mempelajari “cara mudah” untuk mencari element node. Document object menyediakan berbagai method untuk keperluan ini:

<sup>1</sup><https://developer.mozilla.org/en-US/docs/Web/API>

<sup>2</sup><https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementById>

- `document.getElementById()`
- `document.getElementsByTagName()`
- `document.getElementsByClassName()`
- `document.querySelector()`
- `document.querySelectorAll()`

Perhatikan bahwa method-method ini lumayan panjang dan semuanya harus ditulis persis seperti diatas. Tidak boleh ada perubahan huruf besar ke kecil atau sebaliknya.

Mari kita akan bahas satu per satu.

## 20.4 Method `document.getElementById()`

Bisa dibilang, method `document.getElementById()` adalah cara pencarian element HTML yang paling banyak dipakai dan juga paling sederhana. Hampir setiap contoh kode program yang melibatkan DOM akan menggunakan method ini.

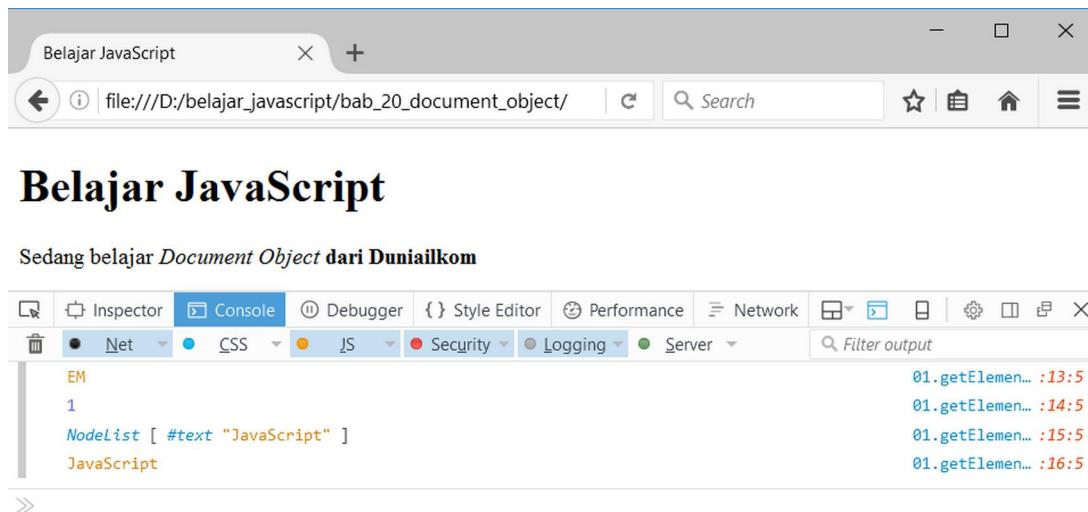
Method `document.getElementById()` berfungsi untuk mencari element node HTML berdasarkan nilai atribut `id`. Atribut `id` ini harus sudah tersedia di dalam tag HTML yang akan dicari.

Berikut contoh penggunaannya:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1 id="judul">Belajar JavaScript</h1>
9   <p>
10    Sedang belajar <em id="emTag">JavaScript</em><b> dari DuniaIlkom</b>
11  </p>
12  <script>
13    var emNode = document.getElementById("emTag");
14
15    console.log(emNode.nodeName);      // EM
16    console.log(emNode.nodeType);     // 1
17    console.log(emNode.childNodes);   // NodeList [ #text "JavaScript" ]
18    console.log(emNode.textContent);  // JavaScript
19
20    emNode.textContent = "Document Object";
21  </script>
22 </body>
23 </html>
```

Perintah `var emNode = document.getElementById("emTag")` akan mencari sebuah tag HTML yang memiliki atribut `id = "emTag"`, kemudian menyimpannya ke dalam variabel `emNode`. Apa isi dari variabel `emNode`? Yakni sebuah **element node**.

Pada baris terakhir saya mencoba mengganti text dari node tersebut dengan mengakses property `emNode.textContent`, hasilnya isi dari tag `<em>` juga ikut berubah:



Gambar: Contoh penggunaan `document.getElementById()`

Sebagai latihan, bisakah anda mencari element dengan `id="judul"`, lalu ubah isi teksnya menjadi "Belajar Document Object". Dari kode HTML yang saya gunakan sebelum ini, `id="judul"` berada di dalam tag `<h1>`.

Baik, berikut kode program yang saya gunakan:

```
1 var h1Node = document.getElementById("judul");
2
3 console.log(h1Node.nodeName);      // H1
4 console.log(h1Node.nodeType);     // 1
5 console.log(h1Node.childNodes);   // NodeList [ #text "Belajar JavaScript" ]
6 console.log(h1Node.textContent);  // Belajar JavaScript
7
8 h1Node.textContent = "Belajar Document Object";
```



Gambar: Contoh penggunaan `document.getElementById()`

Dalam teorinya, pada 1 dokumen HTML tidak boleh terdapat lebih dari 1 atribut id yang memiliki nama sama. Bagaimana kalau kita coba membuat 2 buah atribut id yang sama?

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1 id="judul">Belajar JavaScript</h1>
9   <p id="judul">
10    Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b>
11   </p>
12   <script>
13     var sebuahNode = document.getElementById("judul");
14
15     console.log(sebuahNode.nodeName);
16     console.log(sebuahNode.nodeType);
17     console.log(sebuahNode.childNodes);
18     console.log(sebuahNode.textContent);
19
20     sebuahNode.textContent = "Membuat 2 atribut id";
21   </script>
22 </body>
23 </html>

```

Disini saya membuat tag `<h1>` dan tag `<p>` dengan atribut id yang sama, yakni `id="judul"`. Bagaimana hasilnya?



Gambar: Hasil penggunaan 2 atribut id yang identik

Seperti yang terlihat, method `document.getElementById()` akan mengambil tag HTML pertama yang memiliki `id="judul"`, yakni tag `<h1>`. Karena tag inilah yang berada di urutan teratas sebelum tag `<p>`.

Meskipun dalam prakteknya kita bisa membuat 2 buah atribut id yang sama (dan tidak error), tapi ini tidak disarankan. Sedapat mungkin selalu gunakan atribut id yang berlainan untuk setiap element HTML.

## 20.5 Method `document.getElementsByTagName()`

Method `getElementsByName()` digunakan untuk mencari element node berdasarkan nama tag, seperti "p", "h1", "em" atau "b".

Karena di dalam sebuah dokumen HTML bisa terdapat lebih dari 1 tag yang sama, method ini akan mengembalikan `HTMLCollection`. `HTMLCollection` adalah kumpulan (*collection*) dari element node yang mirip seperti `NodeList`. Sebagaimana layaknya collection, `HTMLCollection` juga bisa diakses menggunakan tanda kurung siku seperti array.

Berikut contoh penggunaan method `getElementsByTagName()`:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar HTML</h1>
9   <h1>Belajar JavaScript</h1>
10  <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
11  <script>
```

```
12  var h1TagName = document.getElementsByTagName("h1");
13
14  console.log(h1TagName); // HTMLCollection [ <h1>, <h1> ]
15
16  console.log(h1TagName[0].nodeName); // H1
17  console.log(h1TagName[0].nodeType); // 1
18  console.log(h1TagName[0].childNodes);
19  // NodeList [ #text "Belajar HTML" ]
20  console.log(h1TagName[0].textContent); // Belajar HTML
21
22  console.log(h1TagName[1].nodeName); // H1
23  console.log(h1TagName[1].nodeType); // 1
24  console.log(h1TagName[1].childNodes);
25  // NodeList [ #text "Belajar JavaScript" ]
26  console.log(h1TagName[1].textContent); // Belajar JavaScript
27  </script>
28 </body>
29 </html>
```

Di dalam kode HTML, terdapat 2 buah tag `<h1>`. Inilah yang ditangkap dari perintah `var h1TagName = document.getElementsByTagName("h1")`. Variabel `h1TagName` akan berisi **HTML-Collection** yang terdiri dari 2 element node `<h1>`.

Untuk mengakses element node pertama, bisa menggunakan perintah `h1TagName[0]`, sedangkan untuk element node kedua bisa diakses dari `h1TagName[1]`.

Sebagai contoh kedua, saya ingin menampilkan isi dari seluruh element tag `<li>` yang ada di dalam sebuah document. Agar lebih mudah, saya akan menggunakan perulangan `for of`.

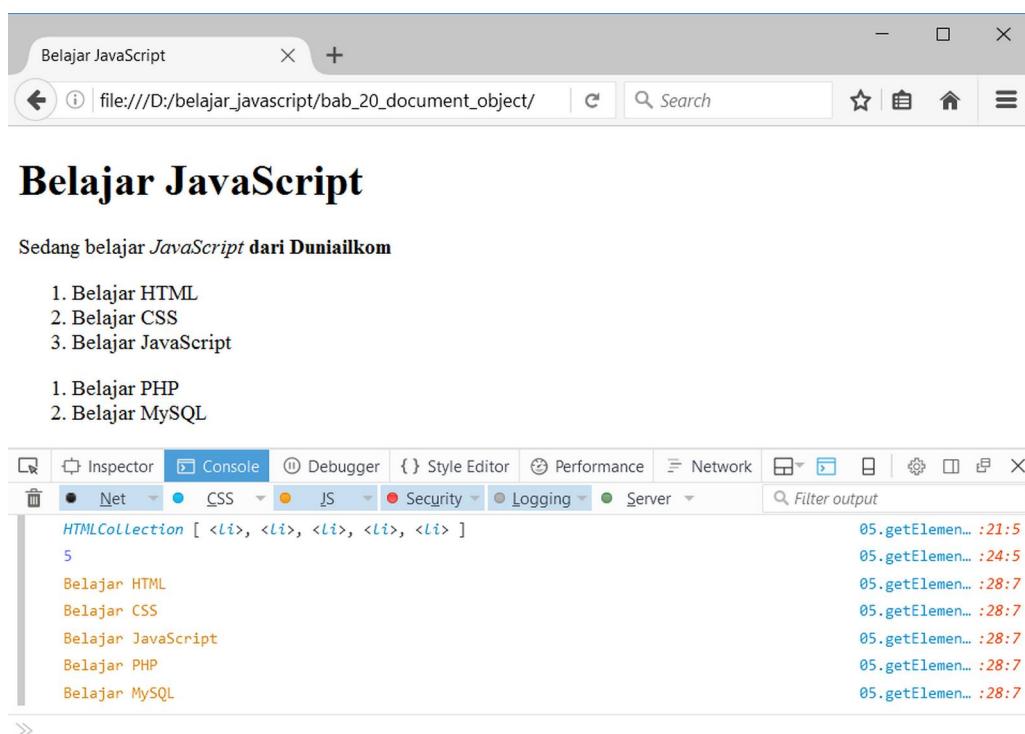
Berikut kode programnya:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta charset="utf-8">
5    <title> Belajar JavaScript </title>
6  </head>
7  <body>
8    <h1>Belajar JavaScript</h1>
9    <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10   <ol>
11     <li>Belajar HTML</li>
12     <li>Belajar CSS</li>
13     <li>Belajar JavaScript</li>
14   </ol>
15   <ol>
16     <li>Belajar PHP</li>
```

```

17    <li>Belajar MySQL</li>
18  </ol>
19  <script>
20    var liTagName = document.getElementsByTagName("li");
21    console.log(liTagName);
22    // HTMLCollection [ <li>, <li>, <li>, <li>, <li> ]
23
24    console.log(liTagName.length);           // 5
25
26    var liNode;
27    for (liNode of liTagName){
28      console.log(liNode.textContent);
29    }
30  </script>
31 </body>
32 </html>

```



Gambar: Menampilkan isi dari seluruh tag <li> menggunakan method getElementsByTagName()

Di dalam struktur HTML, terdapat 2 buah ordered list, yakni tag <ol>. Walaupun saling terpisah, perintah `document.getElementsByTagName("li")` akan mengangkap kelima tag <li> ini.

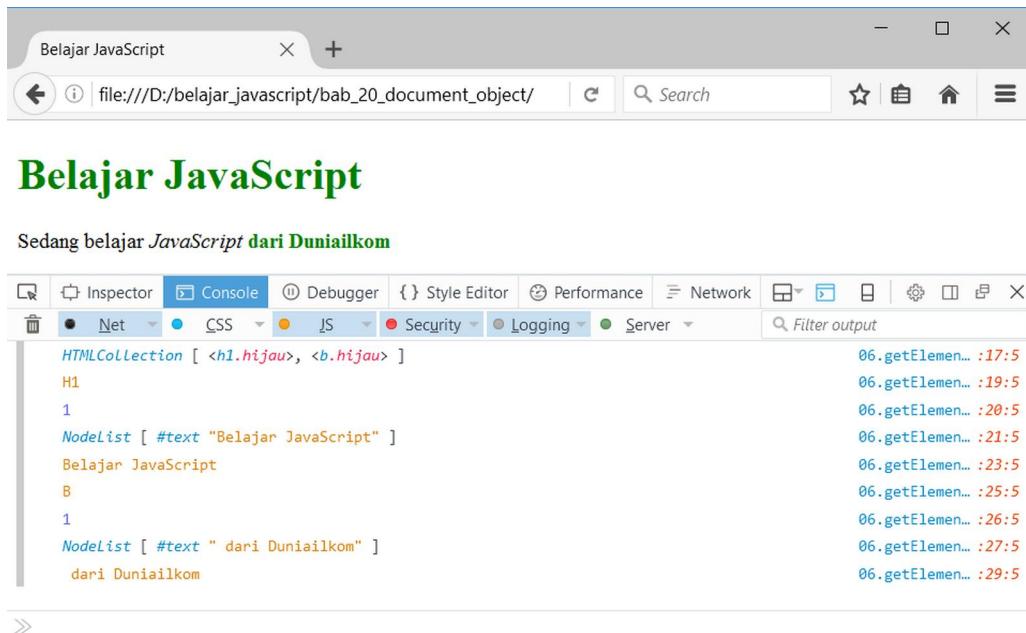
Variabel `liTagName` berisi 5 tag <li>, yang kemudian saya proses menggunakan perulangan `for of` untuk menampilkan isi property `textContent` dari setiap element.

## 20.6 Method document.getElementsByClassName()

Sesuai dengan namanya, method `getElementsByClassName()` berfungsi untuk mencari element node yang memiliki `class` tertentu. Nama class ini diinput sebagai atribut `class` ke dalam tag HTML. Atribut `class` sebenarnya digunakan untuk memasukkan kode CSS.

Karena nama class bisa dimiliki oleh lebih dari 1 tag HTML, maka hasil dari method ini juga berupa `HTMLCollection`. Berikut contoh penggunaannya:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <style>
7     .hijau { color: green; }
8   </style>
9 </head>
10 <body>
11   <h1 class="hijau">Belajar JavaScript</h1>
12   <p>Sedang belajar <em>JavaScript</em>
13   <b class="hijau"> dari DuniaIlkom</b></p>
14 <script>
15   var classNode = document.getElementsByClassName("hijau");
16
17   console.log(classNode); // HTMLCollection [ <h1.hijau>, <b.hijau> ]
18
19   console.log(classNode[0].nodeName); // H1
20   console.log(classNode[0].nodeType); // 1
21   console.log(classNode[0].childNodes);
22   // NodeList [ #text "Belajar HTML" ]
23   console.log(classNode[0].textContent); // Belajar HTML
24
25   console.log(classNode[1].nodeName); // B
26   console.log(classNode[1].nodeType); // 1
27   console.log(classNode[1].childNodes);
28   // NodeList [ #text " dari DuniaIlkom" ]
29   console.log(classNode[1].textContent); // dari DuniaIlkom
30 </script>
31 </body>
32 </html>
```



Gambar: Hasil tampilan penggunaan method getElementsByClassName()

Saya membuat 1 class CSS: `.hijau { color: green; }`. Class "hijau" ini digunakan oleh 2 buah element, yakni tag `<h1>` dan tag `<b>`. Perintah `getElementsByClassName("hijau")` akan mencari seluruh element HTML yang memiliki atribut `class="hijau"`.

Sebagaimana layaknya penggunaan atribut class, satu tag HTML bisa memiliki beberapa class, demikian juga 1 class bisa dimiliki oleh beberapa tag HTML. Semua ini bisa diambil oleh method `getElementsByClassName()`.

## 20.7 Method document.querySelector()

Method `document.querySelector()` berfungsi untuk mencari element node menggunakan **selector** CSS. Yup, method ini terbilang sangat powerfull, karena kita bisa membuat pencarian yang cukup kompleks, sama seperti yang digunakan oleh CSS.

Berikut contoh penggunaannya:

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta charset="utf-8">
5    <title> Belajar JavaScript </title>
6  </head>
7  <body>
8    <h1>Belajar JavaScript</h1>
9    <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10   <script>
11     var sebuahNode = document.querySelector("p b");
12     console.log(sebuahNode.nodeName);      // B

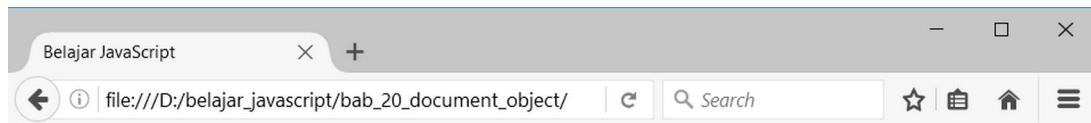
```

```
13     console.log(sebuahNode.nodeType);           // 1
14     console.log(sebuahNode.childNodes);
15     // NodeList [ #text " dari DuniaIlkom" ]
16     console.log(sebuahNode.textContent);        // dari DuniaIlkom
17 </script>
18 </body>
19 </html>
```

Perintah `document.querySelector("p b")` artinya, cari element HTML `<b>` yang ada di dalam tag `<p>`. Kemudian saya mengecek element ini menggunakan beberapa property dari node object. Method `querySelector()` juga bisa digunakan untuk selector yang cukup rumit, seperti contoh berikut:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10  <ol>
11    <li>Belajar HTML</li>
12    <li>Belajar CSS</li>
13    <li>Belajar PHP</li>
14    <li>Belajar JavaScript</li>
15    <li>Belajar MySQL</li>
16  </ol>
17  <script>
18    var sebuahNode = document.querySelector("li:nth-child(4)");
19    sebuahNode.textContent = "Belajar Document Object";
20  </script>
21 </body>
22 </html>
```

Perintah `document.querySelector("li:nth-child(4)")` artinya cari tag `<li>` yang berada di urutan keempat dari parent element-nya. Kemudian saya mengubah nilai `textContent` menjadi "Belajar Document Object". Terlihat text pada urutan list keempat sudah berubah dari "Belajar JavaScript" menjadi "Belajar Document Object".



## Belajar JavaScript

Sedang belajar *JavaScript* dari **DuniaIlkom**

1. Belajar HTML
2. Belajar CSS
3. Belajar PHP
4. Belajar Document Object
5. Belajar MySQL

Gambar: Menggunakan method `querySelector()` untuk mencari element `<li>`

**i** String "li:nth-child(4)" adalah selector CSS yang dikenal sebagai "*Structural Pseudo Class Selectors*". Jika anda tertarik mempelajari selector CSS dan dasar-dasar CSS secara keseluruhan, bisa membaca buku **CSS Uncover** DuniaIlkom.

Percobaan berikutnya, bagaimana jika method `querySelector()` cocok dengan lebih dari 1 element? Mari kita coba:

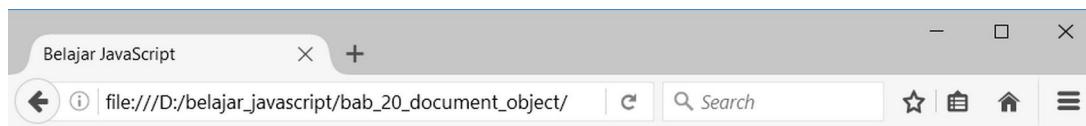
```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10  <ol>
11    <li>Belajar HTML</li>
12    <li>Belajar CSS</li>
13    <li>Belajar PHP</li>
14    <li>Belajar JavaScript</li>
15    <li>Belajar MySQL</li>
16  </ol>
17  <script>
18    var sebuahNode = document.querySelector("li");
19    sebuahNode.textContent = "Belajar Document Object";
20  </script>
21 </body>
22 </html>

```

Perintah `document.querySelector("li")` sebenarnya akan cocok dengan 5 element, yakni 5 buah tag `<li>`, tapi ternyata hanya element pertama saja yang berubah. Inilah syarat lain dari

method `querySelector()`. Jika terdapat lebih dari 1 element, yang akan diambil hanya element pertama saja.



## Belajar JavaScript

Sedang belajar *JavaScript* dari **DuniaIlkom**

1. Belajar Document Object
2. Belajar CSS
3. Belajar PHP
4. Belajar JavaScript
5. Belajar MySQL

Gambar: Hanya tag <li> pertama yang ikut berubah

Sekarang, bagaimana kalau kita memang ingin mencari banyak element menggunakan selector CSS? Solusinya bisa menggunakan method `document.querySelectorAll()`.

## 20.8 Method `document.querySelectorAll()`

Sama seperti method `querySelector()`, method `querySelectorAll()` juga menggunakan selector CSS untuk mencari element HTML. Bedanya, method ini bisa digunakan untuk mengakses banyak node sekaligus.

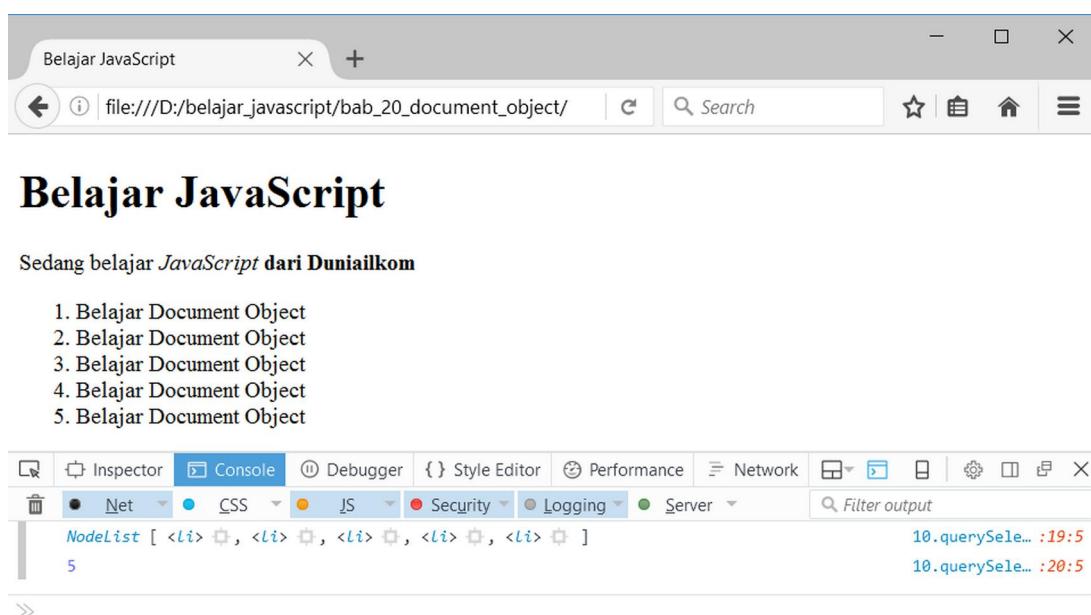
Method `getElementsByName()` akan mengembalikan kumpulan element dalam berbentuk **NodeList**. Berikut contoh penggunaannya:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10  <ol>
11    <li>Belajar HTML</li>
12    <li>Belajar CSS</li>
13    <li>Belajar PHP</li>
14    <li>Belajar JavaScript</li>
15    <li>Belajar MySQL</li>
16  </ol>
17  <script>
```

```

18  var kumpulanNode = document.querySelectorAll("li");
19  console.log(kumpulanNode);
20  // NodeList [ <li>, <li>, <li>, <li>, <li> ]
21  console.log(kumpulanNode.length);    // 5
22
23  var isiNode;
24  for (isiNode of kumpulanNode){
25      isiNode.textContent = "Belajar Document Object";
26  }
27  </script>
28 </body>
29 </html>

```



Gambar: Method `querySelectorAll("li")` digunakan untuk mencari seluruh tag `<li>`

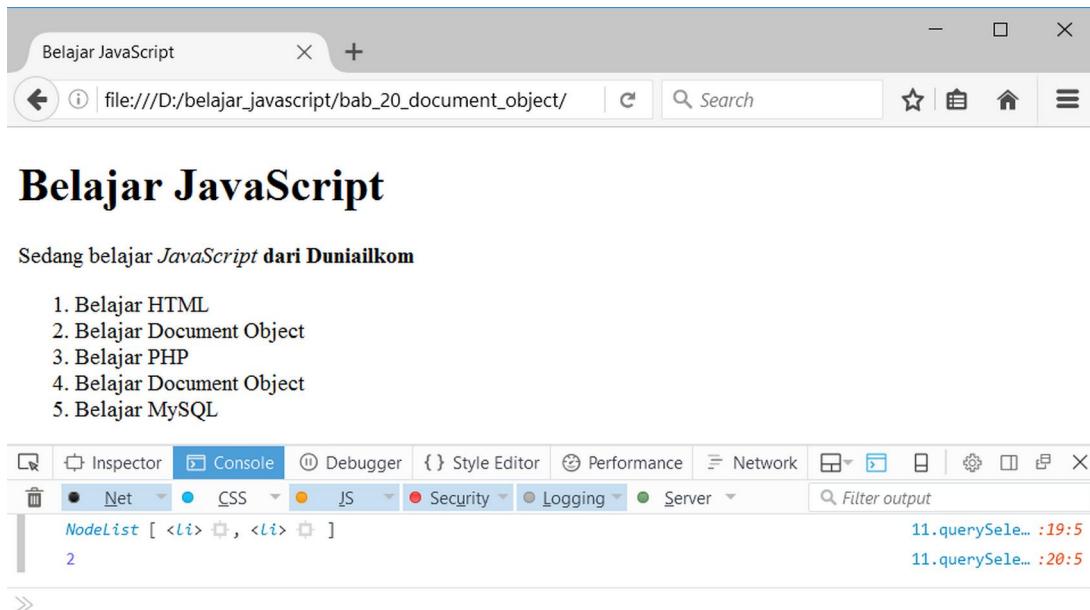
Saya kembali menggunakan list seperti contoh pada method `querySelector()`. Perintah `document.querySelectorAll("li")` akan mencari seluruh element HTML yang cocok dengan selector `li`. Di dalam CSS, ini artinya sama dengan mencari seluruh tag `<li>`.

Menggunakan perulangan `for of`, saya mengubah nilai property `textContent` dari setiap element. Hasilnya, text pada semua daftar list sudah berubah menjadi "Belajar Document Object".

Bagaimana dengan selector yang lebih kompleks? Tidak ada masalah. Dengan mengganti selector diatas menjadi:

```
1  var kumpulanNode = document.querySelectorAll("li:nth-child(even)");
```

Didapat hasil sebagai berikut:



Gambar: Method `querySelectorAll("li:nth-child(even)")` akan mencari tag li dengan urutan genap

Perintah `document.querySelectorAll("li:nth-child(even)")` artinya, saya ingin mencari tag `<li>` yang berada pada urutan genap (*even*). Karena sisanya kode program sama seperti contoh sebelumnya, text pada tag `<li>` kedua dan keempat akan berubah menjadi "Belajar Document Object".

## 20.9 Element Object

Dalam pemrograman berbasis object seperti JavaScript, sebuah object bisa "diturunkan" dari object lain, dimana istilah programmingnya disebut "**inheritance**". Saya memang tidak membahas cara menurunkan object dalam buku ini, karena materi tersebut lumayan rumit dan kurang cocok untuk buku JavaScript untuk pemula. Tapi konsepnya cukup simple.

Sebagaimana yang sudah kita pelajari, di dalam DOM tree semuanya adalah **node object**. **Node Object** ini terbagi menjadi 12 kelompok, dimana yang sering kita bahas adalah **element node** dan **text node**.

Secara teknis, **element node** dan **text node** adalah "turunan" dari **node object**. Setiap object yang diturunkan dari object lain, otomatis akan memiliki seluruh property dan method dari object induknya tersebut. Baik **element node** maupun **text node** sama-sama memiliki property `nodeName`, `nodeType` dan `textContent`. Semua property ini berasal dari **node object**.

Selanjutnya, **element node** maupun **text node** juga memiliki property dan method khusus yang hanya ada di dalam object tersebut. Contoh lain adalah **attr node** atau **attribute node** yang juga merupakan turunan dari **node object**\*. Selain memiliki property dan method bawaan node object, **attr node** juga memiliki property dan method sendiri.



\*Dalam DOM level 3, **Attr node** sudah tidak lagi diturunkan dari **node object**.

Inti dari penjelasan ini adalah, **element node** memiliki property dan method yang khusus tersedia untuk node jenis ini. Namun **element node** juga tetap bagian dari **node object**.

Daftar lengkap mengenai property dan method dari **element node** bisa anda lihat di [Element References<sup>3</sup>](#) - Mozilla Developer Network . Kita akan membahas beberapa diantaranya.

## 20.10 Mengubah Konten Element

Salah satu aspek yang sering dimanipulasi dari sebuah element HTML adalah mengubah konten atau isi teks yang ada di dalam element tersebut. Ini sudah kita praktekkan beberapa kali dengan cara mengubah property `textContent` dari sebuah node.

Kelemahan dari cara ini adalah, kita tidak bisa menambahkan tag HTML ke dalam `textContent`. Dalam bab sebelumnya, solusi dari keterbatasan ini adalah dengan membuat **node object** yang kemudian diinput ke dalam struktur DOM.

Terdapat cara lain yang sebenarnya lebih mudah, yakni menggunakan 2 buah property bawaan **element node**: `Element.innerHTML` dan `Element.outerHTML`.

## 20.11 Property Element.innerHTML

Property ini disebut dengan `innerHTML` karena berisi teks yang ada di dalam tag tersebut, namun tag itu sendiri tidak termasuk. Berikut contoh penggunaannya:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1 id="judul">Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10  <script>
11    var sebuahNode1 = document.getElementById("judul");
12    console.log(sebuahNode1.innerHTML );
13    // Belajar JavaScript
14
15    var sebuahNode2 = document.querySelector("p");
16    console.log(sebuahNode2.innerHTML );
17    // Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b>
18  </script>
19 </body>
20 </html>
```

<sup>3</sup><https://developer.mozilla.org/en-US/docs/Web/API/Element>

Disini saya menampilkan nilai `innerHTML` dari 2 buah element node. Variabel `sebuahNode1` berisi element node `<h1>` yang dicari melalui perintah `getElementById("judul1")`. Variabel `sebuahNode2` berisi element node `<p>` yang dicari melalui perintah `querySelector("p")`.

Terlihat hasil dari property `innerHTML` menampilkan teks yang terdapat di dalam kedua element. Bagaimana jika kita ubah nilai property ini?

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1 id="judul1">Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10  <script>
11    var sebuahNode1 = document.getElementById("judul1");
12    sebuahNode1.innerHTML = "Belajar Element Object";
13
14    var sebuahNode2 = document.querySelector("p");
15    sebuahNode2.innerHTML = "Sedang belajar <em>Element Object</em>";
16  </script>
17 </body>
18 </html>

```



## Belajar Element Object

Sedang belajar *Element Object*

Gambar: Mengubah teks dari tag `<h1>` dan `<p>` menggunakan `innerHTML`

Selain teks yang juga berubah, tag `<em>` tetap dianggap sebagai tag HTML, bukan lagi sebagai teks sebagaimana yang terjadi pada property `textContent`.

Masih ingatkah anda dengan 2 latihan terakhir dari bab sebelumnya? Disana saya membuat kode program untuk memasukkan ordered list `<ol>` ke dalam struktur DOM. Caranya adalah dengan cara membuat node element satu persatu.

Sebagai alternatif, kita bisa menggunakan property `innerHTML`, seperti contoh berikut:

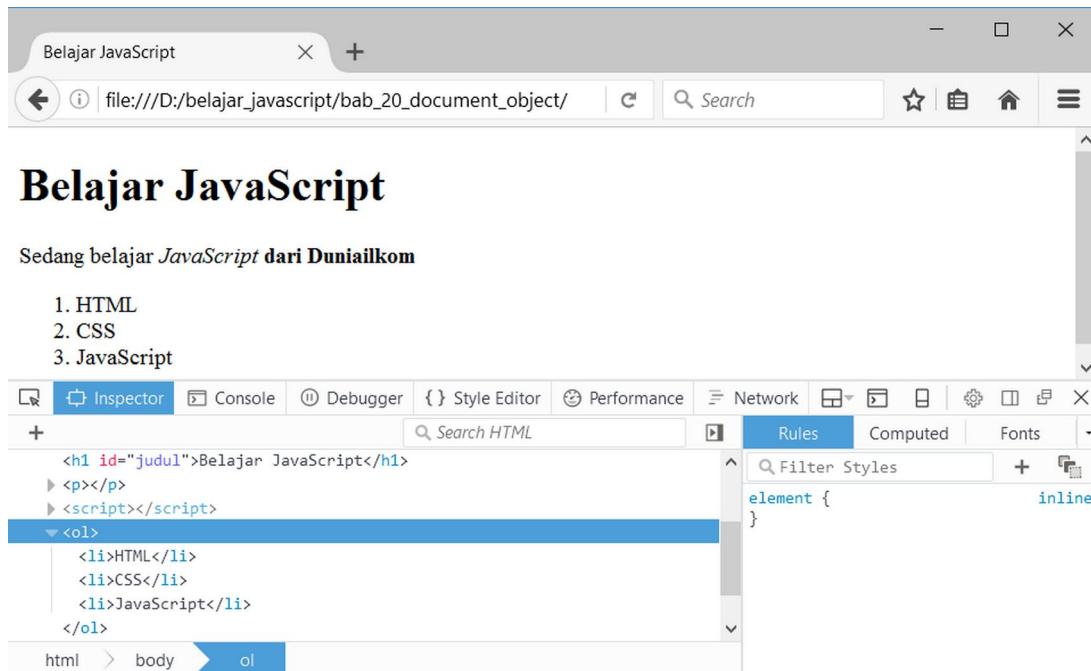
```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1 id="judul">Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10  <script>
11    var listBaru = "<ol><li>HTML</li><li>CSS</li><li>JavaScript</li></ol>";
12    var sebuahNode = document.querySelector("body");
13    var isiBody = sebuahNode.innerHTML;
14    sebuahNode.innerHTML = isiBody + listBaru;
15  </script>
16 </body>
17 </html>
```

Pertama, saya menyiapkan sebuah variabel `listBaru` yang berisi string ordered list. Di dalam string ini terdapat tag `<ol>`, `<li>` dan teks penyusun list. Jika menggunakan node object, kita harus membuatnya satu persatu sebagaimana latihan pada bab sebelumnya.

Kemudian saya mengakses element node tag `<body>` menggunakan perintah `var sebuahNode = document.querySelector("body")`. Variabel `sebuahNode` akan berisi element node tag `<body>`.

Saya tidak bisa langsung menginput isi variabel `listBaru` ke dalam `sebuahNode.innerHTML`, karena jika itu dilakukan, akan menimpa seluruh isi dari tag `<body>` (efeknya menghapus seluruh isi tag `<body>`).

Solusinya, isi dari `sebuahNode.innerHTML` dipindah dulu kedalam variabel `isiBody`. Barulah perintah `sebuahNode.innerHTML = isiBody + listBaru` dijalankan untuk menambah variabel `listBaru` ke dalam isi dari tag `<body>`.



Gambar: Tag <ol> akan diletakkan sebagai element terakhir dari tag <body>

Hasil dari kode diatas, sebuah list baru akan tampil di akhir tag <body>. Karena kode JavaScript juga ditulis di bagian akhir tag <body>, list akan tampil setelah tag <script>.

Bagaimana kalau kita ingin menampilkan list ini dibagian tertentu? Seperti di antara tag <h1> dan <p>?

Jika kita membuat node object satu persatu seperti latihan bab sebelumnya, ini cukup mudah dilakukan, kerena terdapat method insertBefore dari node object. Namun hal ini cukup sulit jika menggunakan innerHTML, kita harus memecah isi dari tag <body> terlebih dahulu.

Trik yang sering dipakai adalah dengan membuat sebuah tag HTML yang berfungsi sebagai *placeholder*, atau tempat penampungan element baru. Biasanya bisa menggunakan tag ‘generic’ <div> atau <span>, seperti contoh berikut:

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8">
5      <title> Belajar JavaScript </title>
6  </head>
7  <body>
8      <h1 id="judul1">Belajar JavaScript</h1>
9      <div id="penampung"></div>
10     <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
11     <script>
12         var listBaru = "<ol><li>HTML</li><li>CSS</li><li>JavaScript</li></ol>";
13         var sebuahNode = document.getElementById("penampung");
14         sebuahNode.innerHTML = listBaru;

```

```
15  </script>
16 </body>
17 </html>
```

Dalam struktur HTML, saya membuat sebuah tag `<div id="penampung">` diantara tag `<h1>` dan `<p>`. Tag `<div>` inilah yang nantinya akan diisi dari JavaScript.

Proses pengisian sendiri cukup sederhana, ambil element node menggunakan perintah `getElementById("penampung")`, kemudian input teks baru menggunakan property `innerHTML`. Cara seperti ini akan sering anda temui dalam tutorial atau materi tentang JavaScript.

## 20.12 Property Element.outerHTML

Property `outerHTML` mirip seperti `innerHTML`, bedanya property `outerHTML` juga mengembalikan tag HTML dari element itu sendiri. Berikut contoh penggunaannya:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta charset="utf-8">
5    <title> Belajar JavaScript </title>
6  </head>
7  <body>
8    <h1 id="judul">Belajar JavaScript</h1>
9    <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10 <script>
11   var sebuahNode1 = document.getElementById("judul");
12   console.log(sebuahNode1.outerHTML );
13   // <h1 id="judul">Belajar JavaScript</h1>
14
15   var sebuahNode2 = document.querySelector("p");
16   console.log(sebuahNode2.outerHTML );
17   // <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
18 </script>
19 </body>
20 </html>
```

Disini saya menjalankan kode program yang sama seperti contoh `innerHTML`. Terlihat bahwa tag `<h1>` dan `<p>` ikut ditampilkan. Efeknya, kita bisa mengubah element tersebut:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1 id="judul">Belajar JavaScript</h1>
9   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
10  <script>
11    var sebuahNode1 = document.getElementById("judul");
12    sebuahNode1.outerHTML = "<h2>Belajar Element Object</h2>";
13
14    var sebuahNode2 = document.querySelector("p");
15    sebuahNode2.outerHTML = "<h4>Sedang belajar <i>Element Object</i></h4>";
16  </script>
17 </body>
18 </html>

```



Gambar: Element node <h1> dan <p> telah berubah menjadi <h2> dan <h4>

Variabel sebuahNode1 dan sebuahNode2 pada awalnya berisi element node <h1> dan <p>, namun karena isi dari property outerHTML telah diubah, tag ini juga ikut berubah menjadi <h2> dan <h4>.

## 20.13 Mengubah Atribut Element

**Atribut** adalah keterangan tambahan yang melekat ke sebuah tag HTML. Beberapa atribut berperan penting bagi tag HTML tersebut dan harus ditulis, seperti atribut **href** pada tag <a> dan atribut **src** pada tag <img>. Menggunakan JavaScript, kita bisa mengubah atribut element dari setiap tag HTML.

Terdapat beberapa method dari element object yang berkaitan dengan atribut, yakni:

- **Element.getAttribute()**
- **Element.getAttribute()**
- **Element.setAttribute()**

- `Element.removeAttribute()`

Serta 1 property:

- `Element.attributes`

## 20.14 Method Element.getAttribute()

Method `hasAttribute()` digunakan untuk memeriksa apakah sebuah atribut sudah di set di dalam element HTML atau belum. Atribut yang akan dicek diinput sebagai argumen dari method ini.

Hasil akhir dari method `hasAttribute()` adalah boolean `true` atau `false`. `True` jika atribut tersebut ditemukan, dan `false` jika atribut itu tidak ditulis.

Berikut contoh penggunaannya:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1 id="judul" title="Sedang Belajar">Belajar JavaScript</h1>
9   <script>
10    var sebuahNode = document.getElementById("judul");
11
12    console.log (sebuahNode.hasAttribute("id"));      // true
13    console.log (sebuahNode.hasAttribute("title"));   // true
14    console.log (sebuahNode.hasAttribute("class"));   // false
15  </script>
16 </body>
17 </html>
```

Saya membuat tag `<h1>` dengan 2 atribut, yakni `id` dan `title`. Jika di cek menggunakan method `hasAttribute()`, hasilnya adalah `true`. Khusus untuk `hasAttribute("class")` akan menghasilkan `false` karena atribut ini memang tidak ada.

## 20.15 Method Element.getAttribute()

Method `getAttribute()` digunakan untuk mengambil nilai dari atribut. Atribut yang ingin diambil nilainya diinput sebagai argumen dari method ini. Berikut contoh penggunaannya:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1 id="judul" title="Sedang Belajar">Belajar JavaScript</h1>
9   <script>
10    var sebuahNode = document.getElementById("judul");
11
12    console.log (sebuahNode.getAttribute("id"));      // judul
13    console.log (sebuahNode.getAttribute("title"));    // Sedang Belajar
14    console.log (sebuahNode.getAttribute("class"));    // null
15  </script>
16 </body>
17 </html>
```

Menggunakan contoh yang sama, perintah `getAttribute("id")` dan `getAttribute("title")` akan mengembalikan nilai "judul" dan "Sedang Belajar", yakni sesuai nilai yang diketik pada kode HTML. Perintah `getAttribute("class")` akan menghasilkan `null` karena atribut `class` memang tidak dimiliki oleh tag `<h1>`.

## 20.16 Method Element.setAttribute()

Method `setAttribute()` digunakan untuk mengubah atau menambah sebuah atribut baru ke dalam element HTML. Atribut ini membutuhkan 2 buah argumen, yakni **nama atribut** dan **nilai dari atribut**. Berikut contoh penggunaannya:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1 id="judul" title="Sedang Belajar">Belajar JavaScript</h1>
9   <script>
10    var sebuahNode = document.getElementById("judul");
11
12    console.log (sebuahNode.getAttribute("id"));      // judul
13    console.log (sebuahNode.getAttribute("title"));    // Sedang Belajar
14
15    sebuahNode.setAttribute("id","judulArtikel");
16    sebuahNode.setAttribute("title","Jangan di ganggu");
```

```

17    sebuahNode.setAttribute("class", "header");
18
19    console.log (sebuahNode.getAttribute("id"));      // judulArtikel
20    console.log (sebuahNode.getAttribute("title"));   // Jangan di ganggu
21    console.log (sebuahNode.getAttribute("class"));   // header
22  </script>
23 </body>
24 </html>
```

Saya masih menggunakan kode program yang sama. Di dalam JavaScript saya mencoba mengubah 3 atribut: "id", "title" dan "class" menggunakan method `setAttribute()`. Method `getAttribute()` akan menampilkan nilai dari atribut-atribut ini sebelum dan sesudah proses perubahan.

## 20.17 Method Element.removeAttribute()

Method `removeAttribute()` digunakan untuk menghapus sebuah atribut. Atribut yang ingin dihapus diinput sebagai argumen. Berikut contoh penggunaannya:

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta charset="utf-8">
5    <title> Belajar JavaScript </title>
6  </head>
7  <body>
8    <h1 id="judul" title="Sedang Belajar">Belajar JavaScript</h1>
9    <script>
10       var sebuahNode = document.getElementById("judul");
11
12       console.log (sebuahNode.getAttribute("id"));      // judul
13       console.log (sebuahNode.getAttribute("title"));   // Sedang Belajar
14
15       sebuahNode.removeAttribute("id");
16       sebuahNode.removeAttribute("title");
17
18       console.log (sebuahNode.getAttribute("id"));      // null
19       console.log (sebuahNode.getAttribute("title"));   // null
20     </script>
21   </body>
22 </html>
```

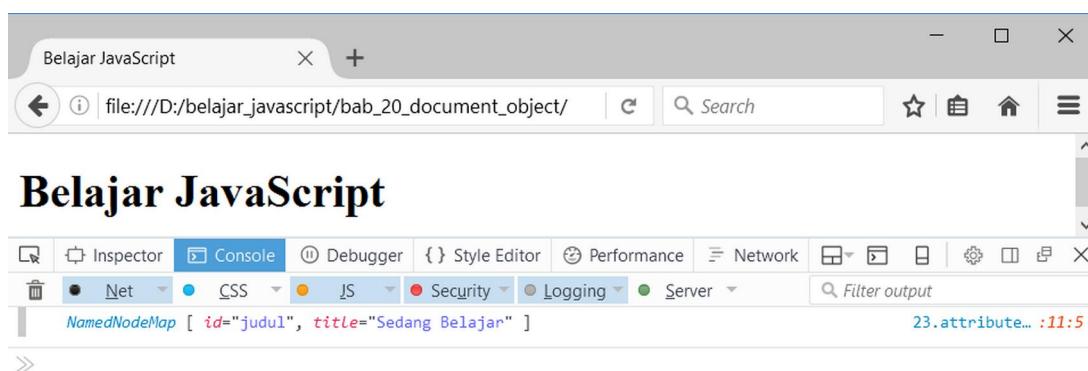
Disini saya menghapus atribut "id" dan "title" menggunakan perintah `removeAttribute("id")` dan `removeAttribute("title")`. Method `getAttribute()` menghasilkan nilai **null** untuk atribut yang sudah dihapus.

## 20.18 Property Element.attributes

Property `attributes` berisi daftar seluruh atribut yang sudah di set nilainya. Property ini bersifat **read only**, sehingga kita tidak bisa mengubah nilai atribut dari property `attributes`.

Berikut contoh penggunaannya:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1 id="judul" title="Sedang Belajar">Belajar JavaScript</h1>
9   <script>
10    var sebuahNode = document.getElementById("judul");
11    console.log (sebuahNode.attributes);
12    // NamedNodeMap [ id="judul", title="Sedang Belajar" ]
13  </script>
14 </body>
15 </html>
```



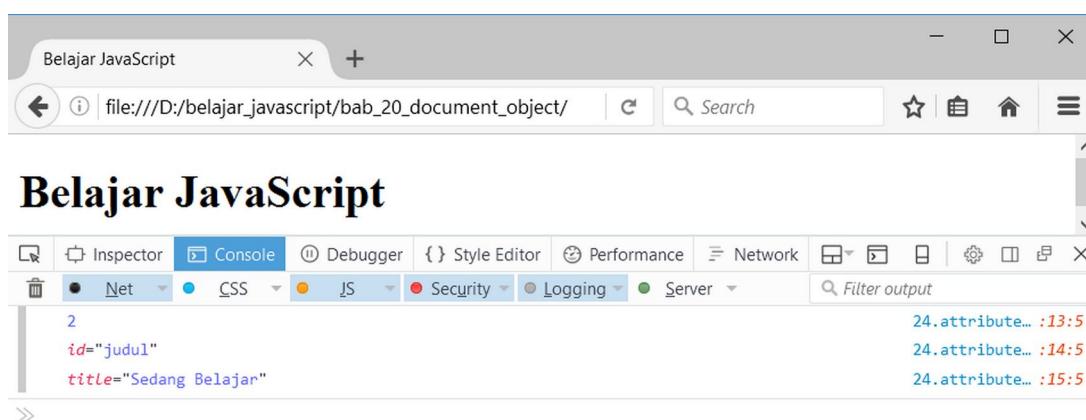
Gambar: Property `attributes` mengembalikan daftar seluruh atribut

Dapat terlihat bahwa hasil dari property `attributes` adalah sebuah `NamedNodeMap`. Jika ‘insting’ JavaScript anda sudah mulai matang, bisa ditebak bahwa `NamedNodeMap` ini adalah sebuah *collection*. Dengan demikian untuk mengakses setiap atribut kita bisa menggunakan cara pengaksesan array:

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8">
5      <title> Belajar JavaScript </title>
6  </head>
7  <body>
8      <h1 id="judul" title="Sedang Belajar">Belajar JavaScript</h1>
9  <script>
10     var sebuahNode = document.getElementById("judul");
11     var nodeAttribute = sebuahNode.attributes;
12
13     console.log (nodeAttribute.length);    // 2
14     console.log (nodeAttribute[0]);        // id="judul"
15     console.log (nodeAttribute[1]);        // title="Sedang Belajar"
16 </script>
17 </body>
18 </html>

```



Gambar: Menampilkan NamedNodeMap hasil property attributes

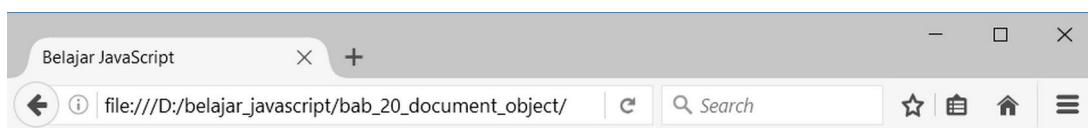
Dikarenakan `NamedNodeMap` adalah sebuah `collection`, maka bisa di cek jumlahnya dari property `length`. Nilai atribut bisa diakses menggunakan tanda kurung siku sebagaimana layaknya array.

Contoh kode program yang saya gunakan memang sangat sederhana agar mudah dipelajari. Dengan kemampuan mengubah atribut, efek yang dihasilkan hanya dibatasi oleh pengetahuan anda tentang jenis-jenis atribut HTML. Sebagai contoh, kita bisa menambahkan kode **inline CSS** menggunakan method `setAttribute()`:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1 id="judul">Belajar JavaScript</h1>
9   <script>
10    var sebuahNode = document.getElementById("judul");
11    sebuahNode.setAttribute("style","color:red; text-shadow:8px 8px pink");
12  </script>
13 </body>
14 </html>

```



## Belajar JavaScript

Gambar: Teks <h1> di style dari JavaScript menggunakan method setAttribute()

Perintah sebuahNode.setAttribute("style","color:red; text-shadow:8px 8px pink") akan menambahkan atribut style ke dalam kode HTML. Ini sama artinya dengan:

```
<h1 style="color:red; text-shadow:8px 8px pink">Belajar JavaScript</h1>
```

Hasilnya, tag <h1> akan tampil dengan warna merah dan memiliki efek bayangan berwarna pink. Semua ini merupakan hasil dari kode CSS yang diinput menggunakan atribut style (inline style CSS).

Kode CSS merupakan bagian yang tidak terpisahkan dari HTML, oleh karena itu JavaScript juga menyediakan beberapa method yang khusus memproses CSS (sebagai alternatif dari method setAttribute()). Inilah yang akan kita bahas selanjutnya.

## 20.19 Mengubah Style Element

Selain mengubah isi konten dan atribut dari element HTML, aspek lain yang bisa kita manipulasi adalah style atau CSS. JavaScript menyediakan berbagai method untuk keperluan ini. Yang akan kita bahas adalah:

Property:

- **HTMLElement.style**

- `Element.classList`
- `Element.className`

Method:

- `Window.getComputedStyle()`



Karena bagian ini akan membahas CSS, saya berasumsi anda sudah paham dengan CSS atau setidaknya pernah menulis kode CSS.

## 20.20 Property HTMLElement.style

Jika anda perhatikan judul property ini, bisa ditebak bahwa kita akan membahas property `style` yang ada di dalam `HTMLElement object`. Apa itu `HTMLElement object`?

`HTMLElement object` merupakan turunan dari `Element object`. Dimana `Element object` juga diturunkan dari `Node object`. Referensi lengkap tentang property dan method dari `HTMLElement` bisa dilihat di [HTML Element References<sup>4</sup>](#).

Dalam prakteknya, `HTMLElement` ini sebenarnya tidak berbeda dengan `Element object`, yakni sebuah element node seperti tag `<p>`, `<h1>`, atau `<table>`. Hanya saja `HTMLElement` mengkhususkan diri untuk element HTML “saja”. Apakah ada element lain selain HTML? Betul, DOM tree tidak hanya digunakan oleh HTML, tapi juga XML.

Hubungan antara `HTMLElement object`, `Element object` dan `Node object` mirip seperti pengelompokan buah apel hijau (`HTMLElement object`), buah apel (`Element object`), dan buah-buahan (`Node object`). Disini `HTMLElement object` adalah `Node object` yang lebih spesifik, sebagaimana buah apel hijau juga merupakan bagian dari buah-buahan.

Baik, jadi apa fungsi dari property `HTMLElement.style`? Property `style` berfungsi untuk menampilkan dan menginput **inline style** CSS ke dalam sebuah element HTML.

Berikut contoh penggunaan property `style` untuk membaca inline style CSS:

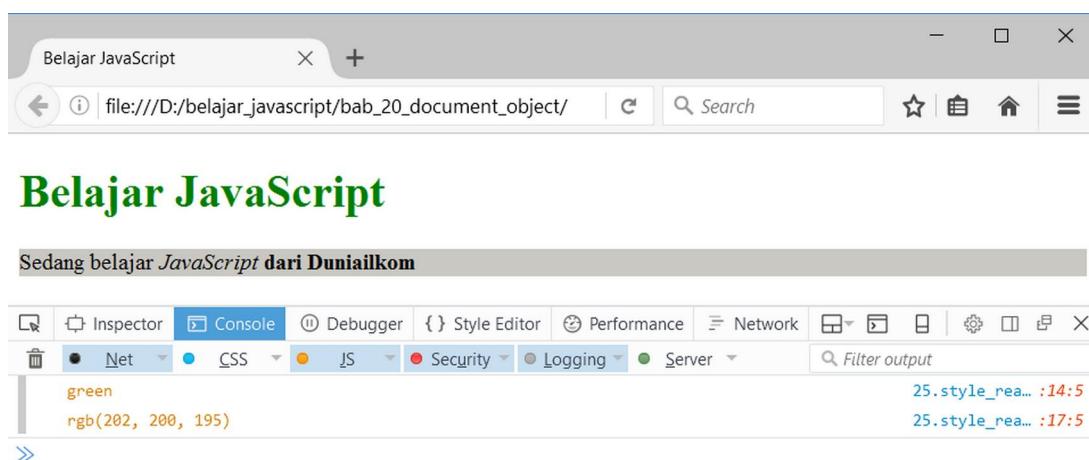
```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1 id="judul" style="color: green">Belajar JavaScript</h1>
9   <p style="background-color: #cac8c3">
```

<sup>4</sup><https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement>

```

10     Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b>
11   </p>
12   <script>
13     var h1Node = document.getElementById("judul");
14     console.log (h1Node.style.color);           // green
15
16     var pNode = document.querySelector("p");
17     console.log (pNode.style.backgroundColor); // rgb(202, 200, 195)
18   </script>
19 </body>
20 </html>

```



Gambar: Membaca property CSS menggunakan atribut style

Di dalam kode HTML, saya membuat atribut style untuk tag <h1> dan <p>. Untuk tag <h1>, property CSS yang dipakai adalah `style="color: green"`, sehingga hasil dari `h1Node.style.color` juga green. Untuk tag <p> property CSSnya adalah `background-color: #cac8c3`, dan hasil dari `pNode.style.backgroundColor` adalah `rgb(202, 200, 195)`.

Inilah cara penggunaan property `style`, kita cukup memanggilnya langsung dari **Element object**. Untuk mengecek nilai dari property CSS, formatnya adalah `namaElementNode.namaPropertyCSS`.

Sedikit catatan, di dalam CSS property yang memiliki 2 suku kata ditulis menggunakan tanda pemisah strip “ - ”, seperti `background-color`, `font-size`, dan `line-height`. Karakter strip ini tidak bisa digunakan di dalam JavaScript. Oleh karena itu penulisannya diganti menjadi **CamelCase**. Property `background-color` menjadi `backgroundColor`, `font-size` menjadi `fontSize`, `line-height` menjadi `lineHeight`, dst.

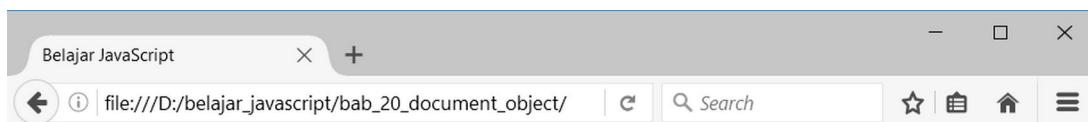
Beberapa nilai property juga akan dikonversi. Sebagai contoh dalam kode diatas saya men-set nilai `background-color: #cac8c3`, yakni mengisi kode warna menggunakan format heksadesimal. Tapi hasilnya ditampilkan dalam format RGB: `rgb(202, 200, 195)`.

Property `style` dari **HTMLElement object** juga bisa diisi nilainya:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1 id="judul" style="color: green">Belajar JavaScript</h1>
9   <p style="background-color: #cac8c3">
10    Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b>
11  </p>
12 <script>
13   var h1Node = document.getElementById("judul");
14   h1Node.style.color = "red";
15
16   var pNode = document.querySelector("p");
17   pNode.style.backgroundColor = "pink";
18   pNode.style.fontSize = "1.4em";
19   pNode.style.textDecoration = "underline";
20   pNode.style.textAlignment = "center";
21   pNode.style.border = "3px solid black";
22   pNode.style.padding = "20px";
23   pNode.style.width = "400px";
24 </script>
25 </body>
26 </html>

```



## Belajar JavaScript

Sedang belajar JavaScript dari DuniaIlkom

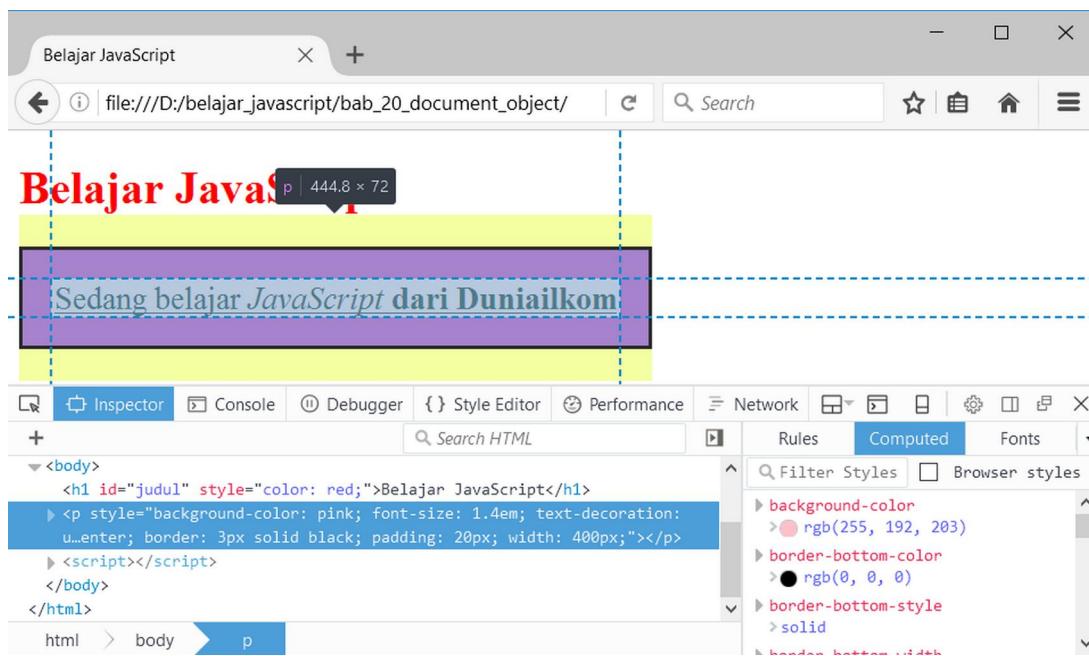
Gambar: Mengubah berbagai property CSS dari JavaScript

Di dalam kode HTML, saya sudah men-set property CSS `color` untuk tag `<h1>`, dan property CSS `background-color` untuk tag `<p>`. Nilai ini akan ditimpak oleh perintah `h1Node.style.color = "red"` dan `pNode.style.backgroundColor = "pink"` yang dijalankan dari JavaScript. Selain itu saya juga menambahkan berbagai property CSS ke dalam `pNode`.

Sebelum anda menggunakan cara ini untuk mengubah kode CSS dari JavaScript, terdapat beberapa catatan.

Pertama, property `HTMLElement.style` ini akan menambahkan kode CSS sebagai **inline CSS**, yakni sama seperti kita menulis langsung atribut `style` ke dalam tag HTML. Dalam beberapa situasi, ini mungkin tidak diinginkan, karena inline style CSS memiliki kekuatan paling tinggi dan sangat susah untuk ditimpak dengan style lain.

Ini bisa dilihat menggunakan **Web Developer Tools** di dalam tab **Inspector**:



Gambar: Tampilan tab Inspector dari Web Developer Tools

Terlihat bahwa semua property yang saya set melalui `HTMLElement.style` diproses sebagai atribut `style` di dalam tag HTML.

Kelemahan kedua, property `HTMLElement.style` hanya bisa membaca property CSS yang diset dari **inline CSS**, tapi tidak bisa membaca property CSS yang diset di dalam tag `<style>`, seperti **internal CSS** dan **external CSS**:

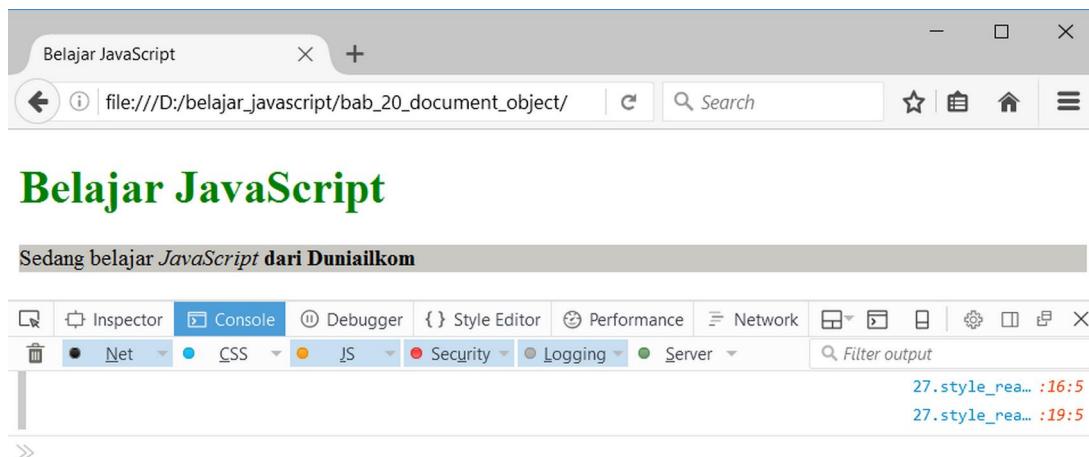
```

1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta charset="utf-8">
5    <title> Belajar JavaScript </title>
6    <style>
7      h1 { color:green; }
8      p { background-color: #cac8c3; }
9    </style>
10   </head>
11   <body>
12     <h1 id="judul">Belajar JavaScript</h1>
13     <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
14     <script>
15       var h1Node = document.getElementById("judul");

```

```

16   console.log (h1Node.style.color);           // (kosong)
17
18   var pNode = document.querySelector("p");
19   console.log (pNode.style.backgroundColor); // (kosong)
20 </script>
21 </body>
22 </html>
```



Gambar: Internal style CSS tidak terbaca dari JavaScript

Di bagian <head>, saya menambahkan tag <style> untuk menginput kode CSS. Terdapat 2 baris kode CSS yang digunakan untuk men-set property color untuk selector h1 dan property background-color untuk selector p.

Ketika perintah h1Node.style.color dijalankan, tidak tampil hasil apa-apa, padahal jelas-jelas warna teks sudah berubah yang menandakan isi dari property color untuk tag <h1> sudah berjalan. Begitu juga kasusnya dengan perintah pNode.style.backgroundColor yang juga menghasilkan nilai kosong.

Jadi bagaimana caranya untuk mendapatkan informasi mengenai style yang sudah berjalan? Kita bisa meminta bantuan kepada method Window.getComputedStyle().

## 20.21 Method Window.getComputedStyle()

Untuk mencari informasi mengenai seluruh nilai property CSS yang saat ini sedang berjalan (tidak hanya dari inline style CSS saja), kita bisa menggunakan method Window.getComputedStyle(). Yup, ini adalah sebuah method kepunyaan dari **Window object**.

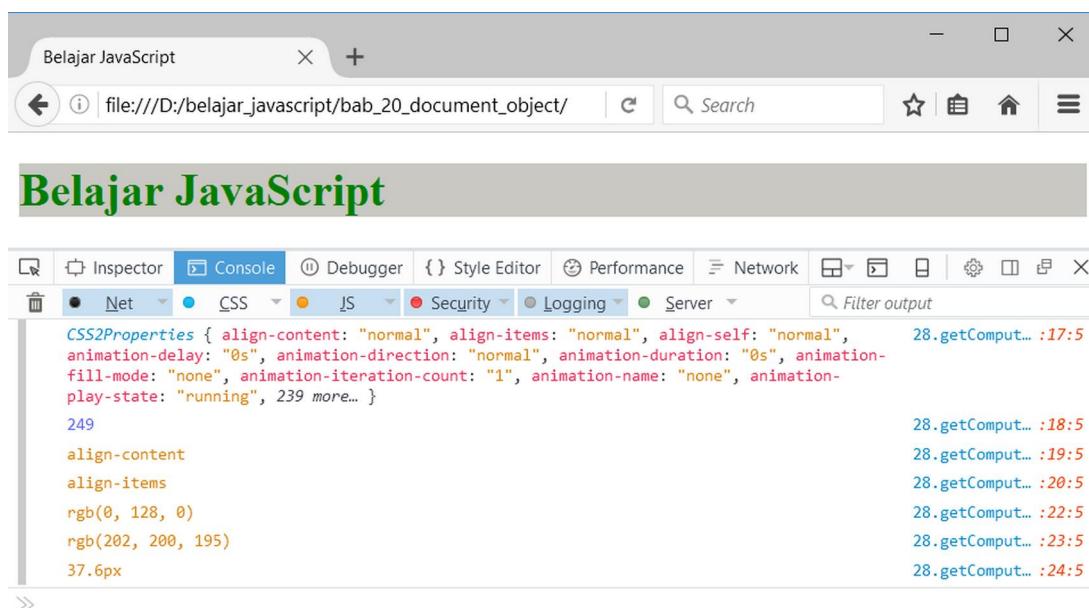
Mehtod getComputedStyle() membutuhkan dua buah argumen: yang pertama berupa node element yang ingin di cek kode CSSnya, dan yang kedua berupa *pseudo-element* untuk kondisi CSS tertentu seperti :after. Argumen kedua ini boleh dikosongkan atau diisi nilai **null** jika kita hanya ingin mengecek property normal CSS.

Berikut hasil percobaannya:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <style>
7     h1 { color:green;
8           background-color: #cac8c3; }
9   </style>
10 </head>
11 <body>
12   <h1 id="judul">Belajar JavaScript</h1>
13   <script>
14     var h1Node = document.getElementById("judul");
15     var hasilStyle = getComputedStyle(h1Node, null);
16
17     console.log (hasilStyle);                      // CSS2Properties...
18     console.log (hasilStyle.length);               // 249...
19     console.log (hasilStyle[0]);                   // align-content
20     console.log (hasilStyle[1]);                   // align-items
21
22     console.log (hasilStyle.color);                // rgb(0, 128, 0)
23     console.log (hasilStyle.backgroundColor);      // rgb(202, 200, 195)
24     console.log (hasilStyle.height);                // 37.6px
25   </script>
26 </body>
27 </html>

```



Gambar: Hasil pemanggilan method `getComputedStyle()` untuk tag `<h1>`

Di dalam tag <style>, saya membuat 2 buah property CSS untuk selector h1. Property ini akan membuat warna teks menjadi hijau dan warna background silver. Jika menggunakan property `HTMLElement.style`, kedua nilai CSS ini tidak akan terlihat, karena bukan di set sebagai inline CSS.

Bagaimana hasil dari pemanggilan method `getComputedStyle()`? Hasilnya adalah sebuah *collection* yang berjenis `CSS2Properties` (atau di referensi mozilla disebut sebagai `CSSStyleDeclaration`). Seperti yang bisa anda lihat, ternyata method ini mengembalikan SEMUA property CSS. Total terdapat 249 property CSS.

Bisa disimpulkan bahwa method `getComputedStyle()` tidak hanya berisi nilai property CSS yang kita set di dalam tag <style>, tapi juga seluruh style CSS lain yang biasanya di set sebagai nilai awal bawaan web browser.

Bagaimana cara mengakses *collection* ini? Jika saya menggunakan tanda kurung siku, seperti `hasilStyle[0]` dan `hasilStyle[1]` hasilnya hanya nama property CSS, tidak beserta nilainya. Tapi jika diakses menggunakan nama property CSS, seperti `hasilStyle.color` atau `hasilStyle.backgroundColor`, akan tampil nilai dari property CSS tersebut.

Hasil dari property CSS ini nantinya bisa kita proses tergantung kebutuhan, misalnya jika sekarang tag <p> memiliki background merah, jalankan kode ini, atau jika tag <h1> memiliki tinggi 10px, jalankan kode program yang lain.

## 20.22 Property Element.className

Menambah style CSS menggunakan property `HTMLElement.style` memang praktis namun tidak efisien. Bagaimana jika property yang harus ditambahkan ada 10 atau 20? daripada menulis satu persatu property CSS dari JavaScript, kita bisa menyiapkan sebuah class CSS, kemudian tinggal ditambahkan kedalam Element HTML.

Untuk proses pembacaan dan penginputan class CSS ke dalam sebuah element HTML, bisa menggunakan property `className`. Berikut contoh penggunaannya:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8">
5      <title> Belajar JavaScript </title>
6      <style>
7          .hijau { color:green; }
8          .merah { color:red; }
9      </style>
10     </head>
11     <body>
12         <h1 id="judul" class="merah">Belajar JavaScript</h1>
13         <script>
14             var h1Node = document.getElementById("judul");
15             console.log(h1Node.className); // merah
```

```

16
17     h1Node.className = "hijau";
18     console.log(h1Node.className); // hijau
19 </script>
20 </body>
21 </html>

```

Di dalam tag `<style>` saya menyiapkan 2 buah class CSS : `hijau` dan `merah`. Masing-masing class ini berisi property `color:green` dan `color:red`. Pada saat kode HTML diproses, teks `<h1>` akan berwarna merah karena terdapat atribut `class="merah"` di dalam tag tersebut.

Saat JavaScript diproses, variabel `h1Node` akan berisi node element dari tag `<h1>`. Perintah `h1Node.className` menghasilkan string `merah`, yakni nama class yang dimiliki oleh tag `<h1>`.

Di baris berikutnya saya menulis `h1Node.className = "hijau"`. Artinya class CSS dari `h1Node` akan berubah menjadi `hijau`. Akibat perubahan ini, teks di dalam tag `<h1>` akan berubah menjadi hijau. Saat di cek kembali menggunakan perintah `h1Node.className`, hasilnya juga `hijau`.

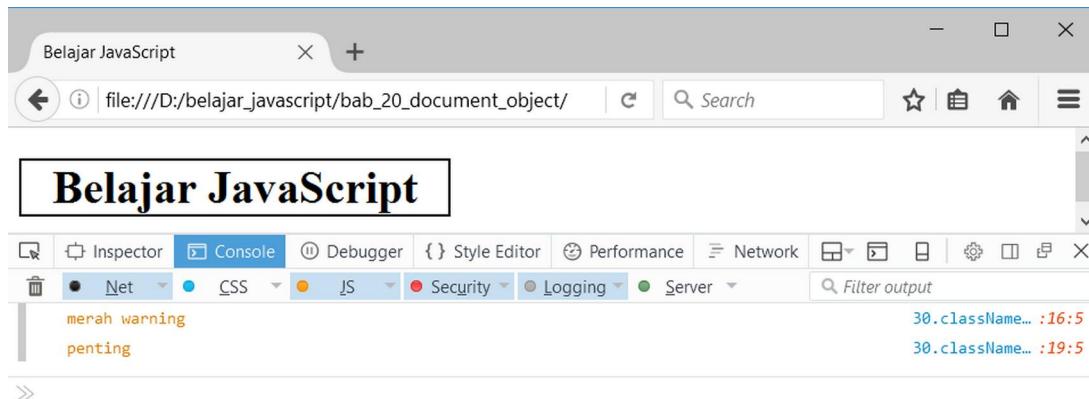
Yang juga perlu diperhatikan, property `className` ini akan menimpa class terdahulu, seperti contoh berikut:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <style>
7     .penting { border: 2px solid black; width: 300px; text-align: center}
8     .merah { color: red; padding: 20px;}
9     .warning { background-color: pink}
10  </style>
11 </head>
12 <body>
13  <h1 id="judul" class="merah warning">Belajar JavaScript</h1>
14  <script>
15    var h1Node = document.getElementById("judul");
16    console.log(h1Node.className); // merah warning
17
18    h1Node.className = "penting";
19    console.log(h1Node.className); // penting
20  </script>
21 </body>
22 </html>

```

Sebelum diproses oleh JavaScript, tag `<h1>` sudah memiliki 2 class, yakni `merah` dan `warning`. Ketika perintah `h1Node.className = "penting"` dijalankan, kedua class ini terhapus dan digantikan dengan class `penting`.



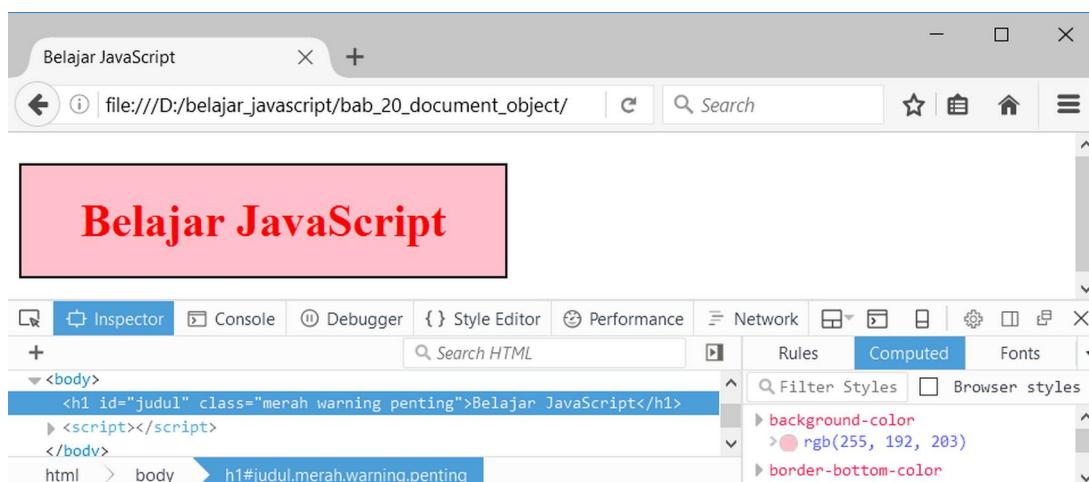
Gambar: Class CSS sebelumnya ikut tertimpa

Solusi dari masalah ini, nama class yang ada harus disimpan dulu ke sebuah variabel, lalu baru ditambahkan dengan nama class baru:

```

1 var h1Node = document.getElementById("judul");
2 var namaClass = h1Node.className;
3 h1Node.className = namaClass + " penting";
4 console.log (h1Node.className); // merah warning penting

```



Gambar: Class CSS yang lama disambung dengan class baru

Atau bisa juga dengan menggunakan operator gabungan assignment, sehingga tidak perlu menggunakan variabel perantara:

```

1 var h1Node = document.getElementById("judul");
2 h1Node.className += " penting";
3 console.log (h1Node.className); // merah warning penting

```

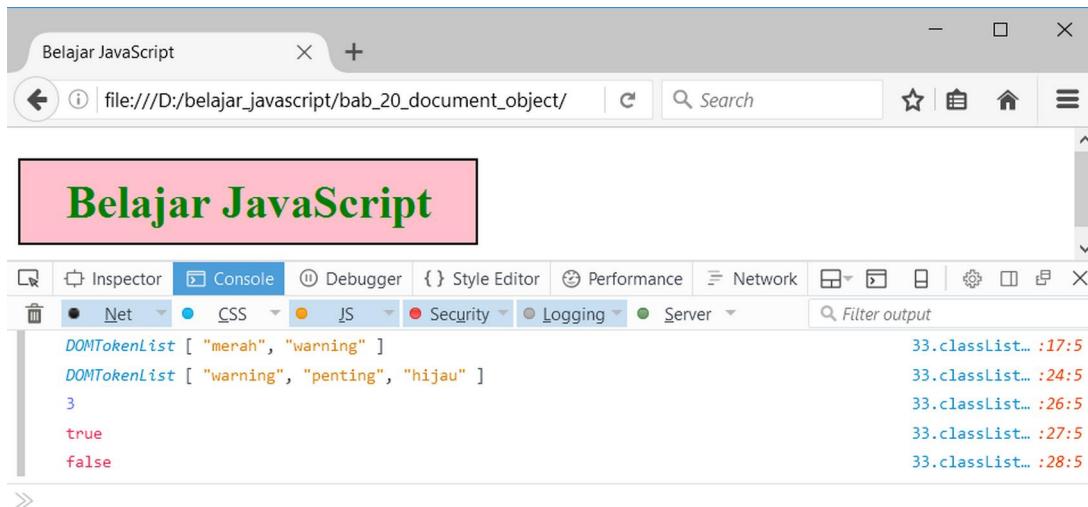
Perintah `h1Node.className += " penting"` artinya tambahkan hasil `className` saat ini dengan string " penting".

Dengan menggunakan property `Element.className`, kita bisa dengan mudah menambah dan mengganti class CSS dari JavaScript. Tapi bagaimana jika class yang ada sudah cukup kompleks? Katakan kita punya 5 class CSS dan ingin menghapus salah satu diantaranya. Inilah masalah yang bisa dipecahkan dengan menggunakan property `Element.classList`.

## 20.23 Property Element.classList

Properti `Element.classList` berguna untuk menampilkan, menambah, dan menghapus class CSS dengan lebih mudah. Langsung saja kita lihat cara penggunaannya:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <style>
7     .penting { border: 2px solid black; width: 300px; text-align: center }
8     .merah { color: red; padding: 20px; }
9     .warning { background-color: pink }
10    .hijau { color: green; padding: 10px; }
11  </style>
12 </head>
13 <body>
14   <h1 id="judul" class="merah warning">Belajar JavaScript</h1>
15   <script>
16     var h1Node = document.getElementById("judul");
17     console.log(h1Node.classList);
18     // DOMTokenList [ "merah", "warning" ]
19
20     h1Node.classList.add("penting");
21     h1Node.classList.remove("merah");
22     h1Node.classList.add("hijau");
23
24     console.log(h1Node.classList);
25     // DOMTokenList [ "warning", "penting", "hijau" ]
26     console.log(h1Node.classList.length);           // 3
27     console.log(h1Node.classList.contains("penting")); // true
28     console.log(h1Node.classList.contains("merah")); // false
29   </script>
30 </body>
31 </html>
```



Gambar: Menampilkan, menambah dan menghapus class CSS menggunakan property classList

Di dalam tag `<style>`, saya membuat 4 buah class CSS: penting, merah, warning, dan hijau. Tag `<h1>` sendiri langsung di set menggunakan atribut `class="merah warning"`. Artinya tag `<h1>` sudah memiliki 2 buah class.

Dengan memanggil perintah `h1Node.classList`, hasilnya adalah sebuah **DOMTokenList** yang merupakan sebuah *collection*, isinya adalah `[ "merah", "warning" ]`. Terlihat bahwa keduanya merupakan 2 buah class yang sudah ada di dalam tag `<h1>`.

Selanjutnya, bagaimana cara menambah dan menghapus class? Kita bisa menggunakan method tambahan untuk `classList`, yakni `add()` dan `remove()`.

Perintah `h1Node.classList.add("penting")` artinya tambahkan class "penting" ke dalam `h1Node`. Sedangkan perintah `h1Node.classList.remove("merah")` artinya hapus class "merah" dari `h1Node`. Terakhir saya menambah class "hijau" ke dalam `h1Node` menggunakan perintah `h1Node.classList.add("hijau")`.

Setelah itu proses penghapusan dan penambahan ini, mari kita cek dengan perintah `console.log(h1Node.classList)`. Hasilnya adalah `DOMTokenList [ "warning", "penting", "hijau" ]`, yang berarti tag `<h1>` sudah memiliki 3 class: "warning", "penting" dan "hijau".

Karena **DOMTokenList** ini adalah sebuah *collection*, kita bisa mengakses property `length`, yang bisa digunakan untuk mengetahui jumlah class dari sebuah element node. Perintah `h1Node.classList.length` akan menghasilkan nilai 3.

Method lain yang bisa diakses ke dalam property `classList` adalah `contains()`, yang berguna untuk memeriksa apakah sebuah class ada di dalam Element atau tidak.

Perintah `h1Node.classList.contains("penting")` akan menghasilkan **true** karena class "penting" memang ditemukan di dalam `h1Node`. Sedangkan perintah `h1Node.classList.contains("merah")` menghasilkan **false** karena class "merah" sudah kita hapus sebelumnya.

Bermodalkan property dan method untuk mengubah kode CSS dari JavaScript, nantinya kita bisa menampilkan efek-efek yang menarik. Terutama setelah mempelajari event yang akan saya bahas dalam bab berikutnya.

## 20.24 Method document.write()

Menutup bab tentang **document** dan **element object**, saya ingin membahas 1 lagi method yang biasanya sangat sering dipakai terutama untuk tutorial terkait JavaScript, yakni method **document.write()**.

Method **write()** digunakan untuk menulis sebuah teks ke dalam document HTML. Fungsinya mirip seperti perintah **echo** atau **print** di dalam PHP. Teks yang ada diinput sebagai argumen akan ditampilkan ke dalam halaman.

Berikut contoh penggunaan method **document.write()**:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1>Belajar JavaScript</h1>
9   <script>
10     document.write("Sebuah teks dari JavaScript");
11     document.write("<h2>Sebuah teks dari JavaScript</h2>");
12   </script>
13 </body>
14 </html>
```



## Belajar JavaScript

Sebuah teks dari JavaScript

### Sebuah teks dari JavaScript

Gambar: Teks yang dihasilkan dari method **document.write()**

Terlihat 3 baris teks, baris pertama berasal dari tag **<h1>** yang ditulis langsung dari HTML. Baris kedua dan ketiga berasal dari method **write()**.

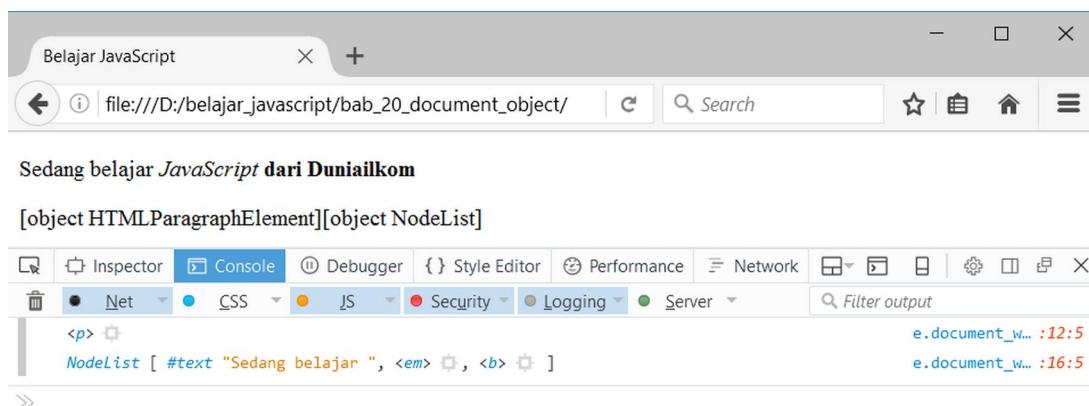
Pertanyaan yang cukup menarik, kenapa saya tidak menggunakan method ini untuk menampilkan teks JavaScript? Tapi menggunakan method **console.log()** dan **alert()** sepanjang buku ini ?

Sebenarnya lebih ke pilihan pribadi, terutama karena method **console.log()** bisa menampilkan informasi yang lebih lengkap, terutama untuk tipe data object seperti **nodeList**. Jika menggunakan **document.write()**, JavaScript akan memaksa object tersebut menjalankan method **toString()**.

Berikut contoh perbandingannya:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <p>Sedang belajar <em>JavaScript</em><b> dari DuniaIlkom</b></p>
9   <script>
10    var h1Node = document.querySelector("p");
11    document.write(h1Node);      // [object HTMLParagraphElement]
12    console.log(h1Node);       // <p>
13
14    var h1Child = h1Node.childNodes;
15    document.write(h1Child);    // [object NodeList]
16    console.log(h1Child);
17    // NodeList [ #text "Sedang belajar ", <em>, <b> ]
18  </script>
19 </body>
20 </html>
```



Gambar: Perbedaan tampilan `document.write()` dengan `console.log()`

Terlihat hasil yang ditampilkan dengan `console.log()` jauh lebih informatif. Untuk proses pembuatan program atau proses pencarian kesalahan (*debugging*), menggunakan method `console.log()` lebih baik dibandingkan `document.write()`. Jika dibandingkan dengan PHP, method `console.log()` sama seperti fungsi `var_dump()` atau `print_r()`.

Kelemahan lain dari `document.write()` adalah, method ini akan menghapus seluruh **DOM tree** jika dipanggil setelah halaman selesai diproses (kebanyakan kode JavaScript diproses seperti ini). Contohnya belum bisa saya tampilkan karena butuh materi tentang **event** (akan dibahas dalam bab berikutnya).

Terdapat beberapa alasan teknis lain kenapa method `document.write()` tidak populer dipakai. Anda bisa membacanya kesini: [Why is `document.write` considered a “bad practice”?](#)<sup>5</sup>.

Intinya, kita disarankan menggunakan method `innerHTML` atau membuat `Text node` dan `Element node` jika ingin menambah sesuatu ke dalam **DOM tree**. Sama seperti fungsi `eval()`, method `document.write()` sebaiknya tidak dipakai. Namun Jika anda sedang membuat artikel atau tutorial singkat mengenai JavaScript, method ini bisa digunakan untuk menampilkan hasil teks dengan cepat.

---

Dalam bab ini kita telah membahas berbagai property dan method untuk keperluan manipulasi element HTML, yakni mencari element node, mengubah teks, mengubah atribut, hingga mengubah kode CSS.

Semua ini akan makin lengkap ketika kita mengubah element ini saat “sesuatu” terjadi, misalnya tombol di klik, keyboard di ketik, dsb. Ini semua akan kita bahas dalam bab berikutnya, yakni tentang **Event**.

---

<sup>5</sup> <http://stackoverflow.com/questions/802854/why-is-document-write-considered-a-bad-practice>

# 21. DOM Event

Materi JavaScript dan DOM yang telah kita pelajari hingga saat ini boleh dikatakan sebagai program yang statis, dimana semuanya langsung diproses begitu halaman terbuka, dan selesai. **Event**-lah yang nantinya “menghidupkan” kode JavaScript untuk membuat halaman web yang dinamis. Kali ini kita akan membahas lebih jauh tentang **DOM Event**.

## 21.1 Pengertian Event

Di dalam DOM, **event** adalah segala sesuatu yang bisa kita lakukan dengan halaman web, seperti men-klik sebuah tombol, klik kanan paragraf, menggeser cursor mouse ke atas sebuah menu, menginput sesuatu ke dalam form, menekan tombol tab, menekan tombol enter, dll.

Ketika event terjadi, kita bisa menyiapkan kode JavaScript untuk melakukan sesuatu, yakni sebagai respon dari event tersebut. Misalnya saat sebuah tombol di klik, tampilkan pesan `alert()`, atau ketika cursor mouse berada di atas menu, ubah warna background menu tersebut.



Secara teknis, kode program yang dibuat untuk “menangkap” event ini dikenal dengan istilah **event handler** atau **event listener**.

DOM Event tidak hanya terbatas untuk mouse dan keyboard saja, tapi juga bisa berbentuk sesuatu yang berasal dari proses DOM itu sendiri, seperti ketika halaman di tampilkan, atau saat sebuah DOM berubah.

Jumlah DOM event yang tersedia sangat banyak, lebih dari 200 dan terus bertambah. Secara garis besar dalam kelompokkan dalam beberapa tipe:

- **Mouse events:** Event yang terjadi ketika menggunakan mouse, seperti di klik, double klik, klik kanan, seret (drag), dll.
- **Keyboard events:** Event yang terjadi saat mengetik sesuatu menggunakan keyboard (biasanya sebagai sarana input ke dalam form). Contohnya ketika sebuah tombol ditekan, saat sebuah tombol dilepaskan, dll.
- **Progression events:** Event yang terjadi pada “siklus hidup” DOM, seperti saat halaman di tampilkan atau ketika halaman ditutup.
- **Form events:** Event yang terjadi saat suatu hal terjadi pada form, contohnya ketika mengubah pilihan menu dari list, atau ketika sebuah checkbox dipilih.
- **Mutation events:** Event yang terjadi saat struktur DOM diubah, misalnya ketika ditambahkan sebuah paragraf baru.
- **Touch events:** Event yang terjadi saat menyentuh layar, ini biasanya untuk perangkat smartphone atau tablet yang memiliki fitur touchscreen.
- **Error events:** Event yang akan dijalankan ketika terjadi error.

Dari semua event ini, yang paling banyak dipakai adalah **event mouse**, **keyboard**, dan **form**. Dalam bab ini saya akan membahas event **event mouse** terlebih dahulu. Untuk event **keyboard** dan **form** akan dibahas dalam bab berikutnya. Namun jika anda sudah paham prinsip penulisan event, untuk event-event yang lain akan sangat mudah diikuti.

Kita ambil contoh **event mouse**, apa saja yang bisa dilakukan dengan menggunakan mouse? Berikut diantaranya:

- **click**: Event ketika tombol kiri mouse di klik 1 kali.
- **dblclick**: Event ketika tombol kiri mouse di klik 2 kali dengan cepat (double click).
- **contextmenu**: Event ketika tombol kanan mouse di klik 1 kali.
- **mouseenter**: Event ketika cursor mouse masuk ke sebuah element.
- **mouseover**: Event ketika cursor mouse berada diatas sebuah element.
- **mouseleave**: Event ketika cursor mouse keluar dari sebuah element.



Daftar lengkap dari seluruh event dalam DOM bisa diakses dari [Event Reference<sup>1</sup>](#) atau [HTML DOM Events<sup>2</sup>](#).

Selanjutnya, bagaimana cara menjalankan event-event ini? Terdapat 3 cara menginput event ke dalam Element HTML:

- Diinput sebagai **atribut HTML**.
- Diinput sebagai **property khusus** dari Element node.
- Memanggil sebuah **method khusus** dari Element node.

Mari kita bahas satu per satu.

## 21.2 Event Handler dari Atribut HTML

Cara paling mudah dan paling singkat untuk menginput event adalah dengan menulisnya langsung sebagai atribut dari sebuah element HTML.

Nama event ditulis dengan tambahan awalan "on". Sebagai contoh, event **click** ditulis sebagai "**onclick**", event **mouseover** ditulis sebagai "**onmouseover**". Nilai dari atribut ini berupa kode JavaScript yang akan dijalankan. Berikut contohnya:

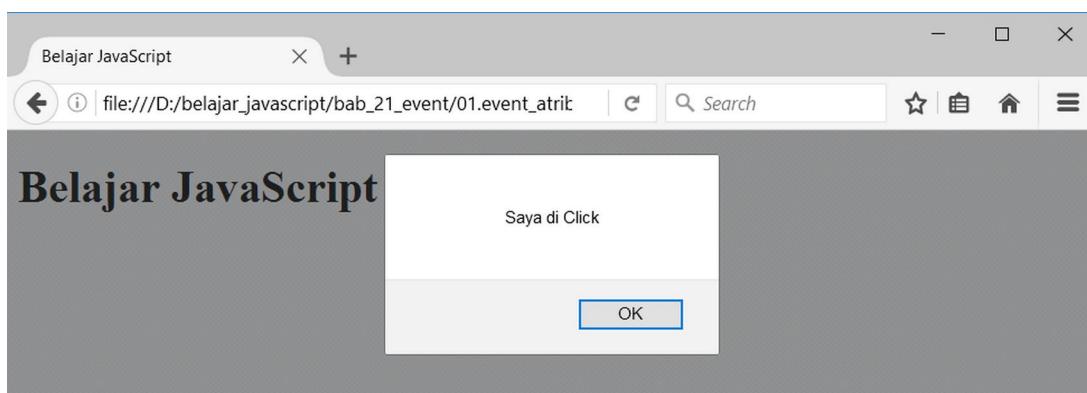
---

<sup>1</sup><https://developer.mozilla.org/en-US/docs/Web/events>

<sup>2</sup>[http://www.w3schools.com/jsref/dom\\_obj\\_event.asp](http://www.w3schools.com/jsref/dom_obj_event.asp)

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <h1 onclick="alert('Saya di Click');">Belajar JavaScript</h1>
9 </body>
10 </html>
```

Tambahan atribut `onclick="alert('Saya di Click');"` artinya ketika tag `<h1>` ini mengalami event `click` (yakni ketika teks `<h1>` di klik), jalankan perintah JavaScript `alert('Saya di Click')`. Mari kita coba:



Gambar: Jendela alert tampil ketika teks di klik

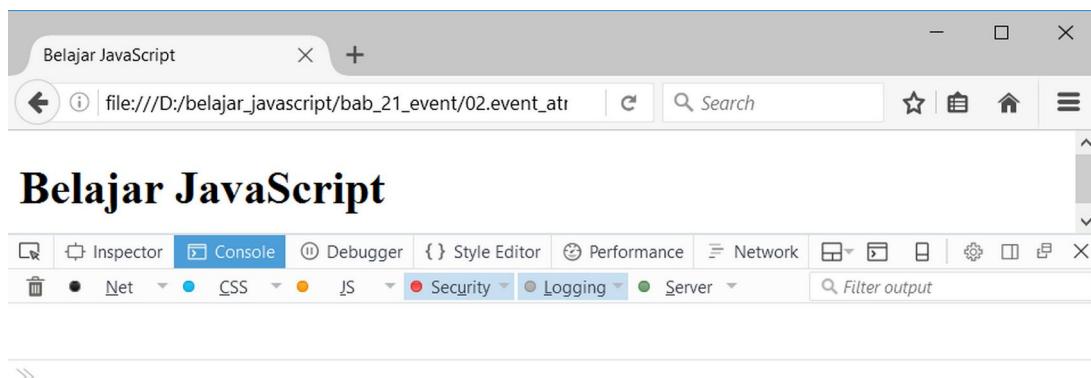
Inilah contoh event pertama kita.

Ada beberapa hal yang harus diperhatikan ketika menulis event sebagai atribut. Pertama, sebagaimana yang kita ketahui, sebuah atribut HTML sebaiknya ditulis dalam tanda kutip, dan menjadi keharusan jika nilai atribut itu mengandung spasi, seperti `title="Ini teks title"`.

Dalam contoh diatas, saya menggunakan tanda kutip dua untuk atribut `onclick`, yakni `onclick="..."`. Dengan demikian, saya tidak bisa menggunakan tanda kutip dua di dalam JavaScript dan harus menggunakan tanda kutip satu. Oleh karena itu perintah `alert()` harus ditulis sebagai `alert('Saya di Click')`, tidak boleh `alert("Saya di Click")`.

Mari kita coba kemungkinan yang kedua ini:

```
<h1 onclick="alert("Saya di Click");">Belajar JavaScript</h1>
```



Gambar: Event tidak berjalan, dan juga tidak terdapat error

Event menjadi tidak berjalan, namun di dalam Tab Console juga tidak menampilkan error apapun. Jika anda menggunakan teks editor yang memiliki fitur *error reporting* seperti **Komodo Edit**, akan tampil tanda peringatan jika ditemukan kasus seperti ini:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>          JavaScript: Unexpected early end of program.
8   <h1 onclick='alert("Saya di Click");'>Belajar JavaScript</h1>
9 </body>
10 </html>
```

Gambar: Pesan peringatan dari Komodo Edit

Atau boleh juga tanda kutip ini dibalik, seperti berikut:

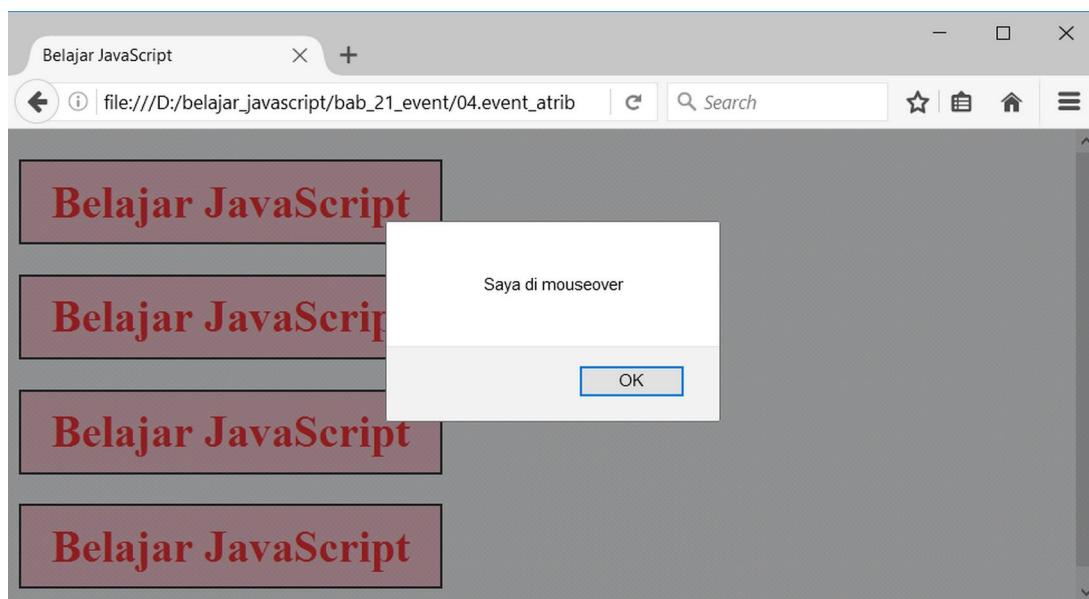
```
1 <h1 onclick='alert("Saya di Click");'>Belajar JavaScript</h1>
```

Kali ini saya menggunakan tanda kutip satu di awal nilai atribut dan tanda kutip dua di dalam JavaScript. Dengan demikian, kedua tanda kutip ini ada tidak akan bentrok.

Hal kedua yang harus diperhatikan adalah karena isi dari atribut **event** ini berupa kode JavaScript, kita sebaiknya menambahkan tanda titik koma di akhir kode: `onclick="alert('Saya di Click');".`

Baik, mari kita coba penulisan event lain dengan membuat beberapa tag `<h1>` :

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <style>
7     h1 { border: 2px solid black;
8           width: 280px;
9           padding: 10px;
10          text-align: center;
11          color: red;
12          background-color: pink; }
13 </style>
14 </head>
15 <body>
16   <h1 onclick="alert('Saya di klik');">Belajar JavaScript</h1>
17   <h1 ondblclick="alert('Saya di double klik');">Belajar JavaScript</h1>
18   <h1 oncontextmenu="alert('Saya di klik kanan');">Belajar JavaScript</h1>
19   <h1 onmouseover="alert('Saya di mouseover');">Belajar JavaScript</h1>
20 </body>
21 </html>
```



Gambar: Jendela alert saat tag <h1> terakhir di mouseover

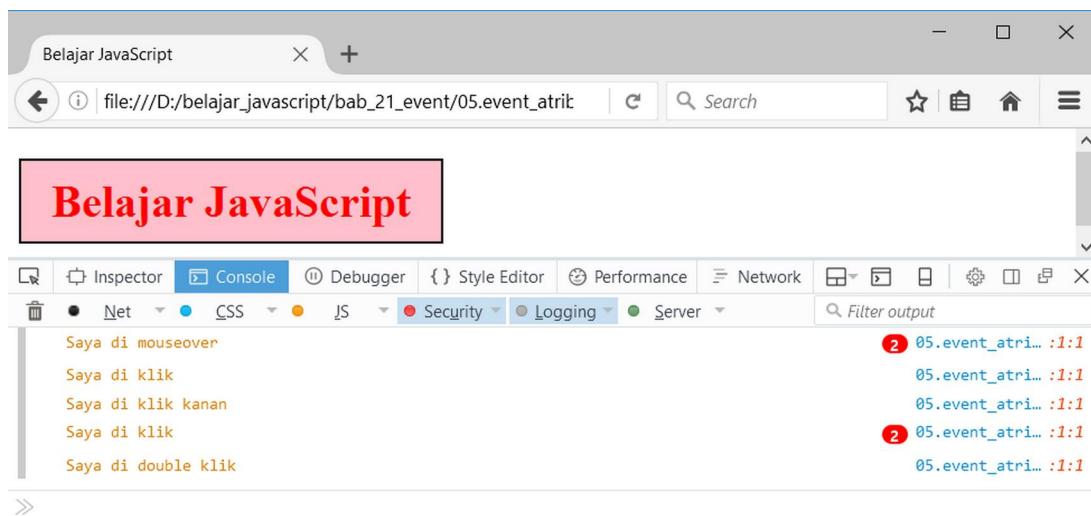
Disini saya membuat 4 buah tag <h1> yang memiliki event berbeda-beda. Atribut `onclick` untuk event **click**, atribut `ondblclick` untuk event **double click**, atribut `oncontextmenu` untuk event **klik kanan**, dan atribut `onmouseover` untuk event saat **cursor mouse berada di atas tag <h1>**.

Saya juga menambahkan sedikit kode CSS untuk membuat tampilan tag <h1> tampak lebih menarik.

Silahkan anda coba berbagai event ini, dan akan tampil jendela `alert()` yang berbeda-beda.

Bagaimana jika menginput banyak event dalam 1 element HTML? Tidak masalah, kita tinggal membuat beberapa atribut event:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <style>
7     h1 { border: 2px solid black;
8       width: 280px;
9       padding: 10px;
10      text-align: center;
11      color: red;
12      background-color: pink; }
13 </style>
14 </head>
15 <body>
16   <h1 onclick = "console.log('Saya di klik');"
17     ondblclick = "console.log('Saya di double klik');"
18     oncontextmenu = "console.log('Saya di klik kanan');"
19     onmouseover = "console.log('Saya di mouseover');">
20     Belajar JavaScript
21   </h1>
22 </body>
23 </html>
```



Gambar: Berbagai event yang dieksekusi ke console.log()

Disini saya membuat 4 atribut event ke dalam 1 tag <h1>. Selain itu isi dari perintah event saya ganti dari alert() menjadi console.log() agar tidak terlalu mengganggu. Sekarang, setiap event yang terjadi akan tampil di tab console.

**i** Untuk menghemat tempat, saya hanya akan menampilkan kode dalam tag <body> saja. Kecuali ditampilkan semua, bagian <head> dan tag <style> akan sama untuk contoh-contoh kode program selanjutnya.

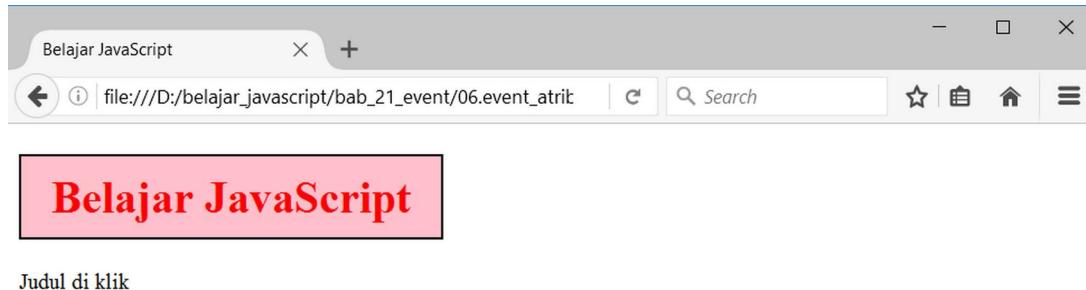
Isi dari kode JavaScript untuk atribut event tidak hanya terbatas pada perintah sederhana seperti `alert()` dan `console.log()`. Kita juga bisa membuat kode yang lebih kompleks seperti memanipulasi struktur DOM. Berikut contohnya:

```
1 <body>
2   <h1 onclick="document.querySelector('p').innerHTML='Judul di klik';">
3     Belajar JavaScript
4   </h1>
5   <p></p>
6 </body>
```

Di bawah tag <h1> saya membuat sebuah tag <p> kosong (tanpa berisi teks). Teks akan diinput dari kode JavaScript yang di letakkan pada atribut **onclick**. Apa isinya? Yakni perintah:

```
1 document.querySelector('p').innerHTML='Judul di klik';
```

Artinya, ketika tag <h1> di klik, tag <p> akan berisi teks ‘**Judul di klik**’, seperti tampilan berikut:



Gambar: Teks ‘Judul di klik’ tampil saat tag <h1> di click

Salah satu kelemahan ketika menginput event sebagai atribut adalah keterbatasan ruang. Jika kode program JavaScript sudah terlalu panjang, sangat susah untuk diketik dan diedit.

Solusinya, kita bisa memindahkan kode JavaScript ke dalam sebuah function, kemudian memanggil function tersebut dari dalam atribut. Seperti contoh berikut:

```
1 <body>
2   <h1 onclick="tampilkan();">Belajar JavaScript</h1>
3   <p></p>
4   <script>
5     function tampilkan(){
6       document.querySelector("p").innerHTML="Judul di klik";
7     }
8   </script>
9 </body>
```

Di dalam tag `<script>` saya membuat sebuah function `tampilkan()`. Isinya sama seperti contoh sebelumnya, yakni mengisi tag `<p>` dengan teks "Judul di klik". Function `tampilkan()` hanya akan berjalan ketika tag `<h1>` di klik, yakni saat event `click` terjadi.

Menulis event sebagai atribut memang praktis dan singkat, tapi ada beberapa kelemahan dari metode ini:

- Kode program JavaScript “bercampur” dengan HTML, sehingga susah untuk dikelola. Bayangkan jika di dalam dokumen terdapat ratusan baris program yang terdiri dari campuran HTML dan JavaScript, lalu kita butuh memperbaiki beberapa baris.
- Tidak bisa menghapus event yang sudah ditulis tanpa mengedit langsung kode HTML.
- Event hanya bisa ditempatkan ke dalam tag HTML yang sudah ada, tapi tidak bisa untuk element HTML yang digenerate secara dinamis seperti hasil `document.createElement()`.

Cara penulisan event sebagai atribut hanya cocok untuk kode program singkat atau untuk men-demo-kan sesuatu. Jika anda mengikuti artikel-artikel yang membahas JavaScript, akan sering menemui cara seperti ini. Untuk aplikasi “real world”, sebaiknya kita menggunakan metode lain.

## 21.3 Event Handler dari Property Element

Cara kedua untuk “memasang” event adalah dengan menginputnya ke dalam property khusus dari Element Node (secara teknis property ini melekat ke [GlobalEventHandlers<sup>3</sup>](#) ).

Langsung saja kita lihat contoh penggunaannya:

---

<sup>3</sup><https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers>

```
1 <body>
2   <h1 id="judul">Belajar JavaScript</h1>
3   <p></p>
4   <script>
5     function tampilkan(){
6       document.querySelector("p").innerHTML="Judul di klik";
7     }
8
9     var h1Node = document.getElementById("judul");
10    h1Node.onclick = tampilkan;
11  </script>
12 </body>
```

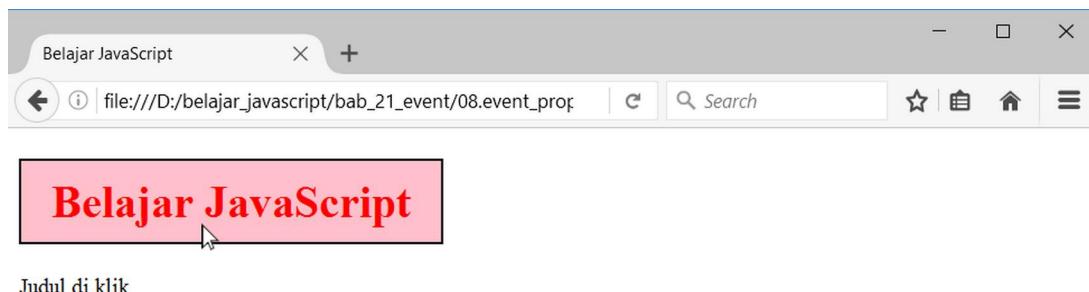
Untuk kode HTML, saya tetap membuat sebuah tag `<h1>` yang akan menjadi object dari event, serta sebuah tag `<p>` kosong.

Di dalam JavaScript terdapat function `tampilkan()`. Function ini bertujuan untuk mengisi teks "Judul di klik" ke dalam tag `<p>`. Tentunya function ini baru akan berjalan ketika sudah dipanggil.

Selanjutnya saya menyiapkan variabel `h1Node` yang berisi element node dari tag `<h1>`. Element node ini didapat dari perintah `document.getElementById("judul")`.

Proses penginputan event terjadi di baris terakhir. Perintah `h1Node.onclick = tampilkan` artinya, input function `tampilkan()` ke dalam property `onclick` dari element node `h1Node`.

Masih ingat dengan materi tentang *function expression*? Disana kita menginput sebuah function ke dalam variabel. Hal yang sama juga terjadi disini. Saat event `click` terjadi, function `tampilkan()` akan berjalan.



Gambar: Function `tampilkan()` akan diproses saat tag `<h1>` di klik

Cara penulisan alternatif adalah langsung menginput anonymous function ke dalam property `onclick`:

```
1 <body>
2   <h1 id="judul1">Belajar JavaScript</h1>
3   <p></p>
4   <script>
5     var h1Node = document.getElementById("judul1");
6
7     h1Node.onclick = function () {
8       document.querySelector("p").innerHTML="Judul di klik";
9     };
10    </script>
11  </body>
```

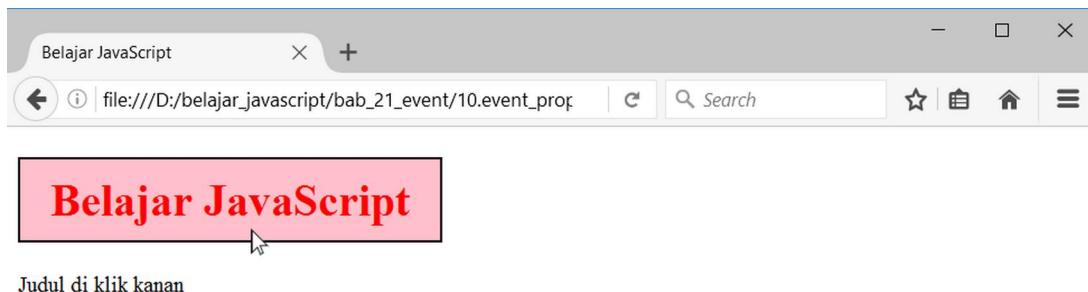
Bagaimana cara menambahkan event lain seperti dblclick dan contextmenu? Caranya juga sama, kita tinggal mengisi property event yang sudah ditambahkan awalan "on" dengan fungsi yang ingin dijalankan, seperti contoh berikut:

```
1 <body>
2   <h1 id="judul1">Belajar JavaScript</h1>
3   <p></p>
4   <script>
5     function tampilanClick(){
6       document.querySelector("p").innerHTML="Judul di klik";
7     }
8
9     function tampilanDblclick(){
10       document.querySelector("p").innerHTML="Judul di double klik";
11     }
12
13    function tampilanContextmenu(){
14      document.querySelector("p").innerHTML="Judul di klik kanan";
15    }
16
17    var h1Node = document.getElementById("judul1");
18    h1Node.onclick = tampilanClick;
19    h1Node.ondblclick = tampilanDblclick;
20    h1Node.oncontextmenu = tampilanContextmenu;
21  </script>
22 </body>
```

Disini saya menyiapkan 3 buah function yang akan dipanggil untuk 3 kondisi event:

- Function `tampilanClick()` akan mengubah isi teks tag `<p>` menjadi "Judul di klik" saat event `onclick`. Ini dibuat melalui perintah `h1Node.onclick = tampilanClick`.
- Function `tampilanDblclick()` akan mengubah isi teks tag `<p>` menjadi "Judul di double klik" saat event `ondblclick`. Ini dibuat melalui perintah `h1Node.ondblclick = tampilanDblclick`.

- Function `tampilkanContextmenu()` akan mengubah isi teks tag `<p>` menjadi "Judul di klik kanan" saat event `oncontextmenu` (sebutan untuk event klik kanan). Ini dibuat melalui perintah `h1Node.oncontextmenu = tampilkanContextmenu`.



Gambar: Teks akan tampil saat tag `<h1>` di klik kanan (event `oncontextmenu`)

Dengan menggunakan metode input ke property seperti ini, kode program JavaScript sudah sepenuhnya terpisah dari HTML. Kita tidak perlu lagi mengutak-atik atribut HTML jika ingin menambahkan event baru.

Termasuk jika ingin menghapus sebuah event, cukup memberikan nilai `null` kedalam property tersebut:

```
1 h1Node.onclick = null;
```

Meskipun sudah relatif lebih baik daripada menginput event ke dalam atribut, masih terdapat kelemahan dari metode ini. Kita tidak bisa menggabungkan dua event yang sama, event baru akan menimpa event sebelumnya, seperti contoh berikut:

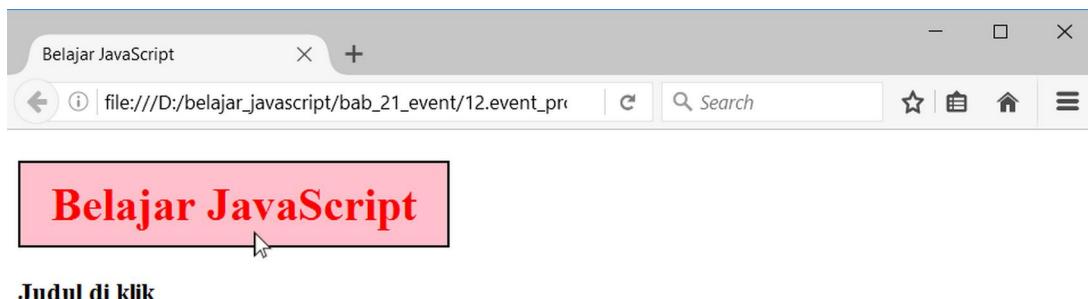
```
1 <body>
2   <h1 id="judul">Belajar JavaScript</h1>
3   <p></p>
4   <h3></h3>
5   <script>
6     function tampilkanClickP(){
7       document.querySelector("p").innerHTML="Judul di klik";
8     }
9     function tampilkanClickH3(){
10       document.querySelector("h3").innerHTML="Judul di klik";
11     }
12
13   var h1Node = document.getElementById("judul");
14   h1Node.onclick = tampilkanClickP;
15   h1Node.onclick = tampilkanClickH3;
16   </script>
17 </body>
```

Disini saya menambahkan sebuah tag <h3> kosong di bawah tag <p>. Kedua tag ini digunakan untuk menampung output saat tag <h1> di klik.

Pada bagian JavaScript, saya membuat dua buah function: `tampilkanClickP()` dan `tampilkanClickH3()`. Kedua function ini akan menambahkan teks "Judul di klik" ke dalam tag <p> dan tag <h3> secara terpisah.

Proses penambahan event dilakukan pada 2 baris terakhir: `h1Node.onclick = tampilkanClickP` dan `h1Node.onclick = tampilkanClickH3`. Setiap perintah berfungsi untuk menginput function `tampilkanClickP()` dan `tampilkanClickH3()` ke dalam event `onclick`. Harapannya, ketika judul **h1** diklik, akan tampil teks "Judul di klik" ke dalam tag <p> dan tag <h3>.

Bagaimana hasilnya?



Gambar: Hanya tampil 1 teks

Ternyata hanya tampil 1 teks saja. Melihat hasil yang dalam huruf tebal, bisa dipastikan ini adalah teks yang berasal dari function `tampilkanClickH3()`. Bagaimana dengan function `tampilkanClickP()`? function ini telah tertimpa oleh function `tampilkanClickH3()`. Artinya cara penginputan event melalui property hanya bisa untuk 1 function saja (function terakhir yang diinput).

Solusinya? Kita masuk ke metode ketiga, yakni dengan menggunakan method dari Element Node.

## 21.4 Event Handler dari Method Element

Cara ketiga untuk menginput event adalah dengan menggunakan method khusus yang bernama `addEventListener()`. Ini merupakan metode yang paling disarankan dan menjadi standar DOM W3C.

Menginput event melalui atribut dan property element seperti yang sudah kita pelajari sebelumnya, bukanlah berasal dari W3C, melainkan dari era IE vs Netscape (sebelum W3C didirikan). Namun kedua cara itu masih di dukung web browser modern.

Membuat *event handler* menggunakan method `addEventListener()` tidak sulit, cukup menulis method ini dengan 2 argument. Argumen pertama berupa nama event tanpa awalan on, seperti "click", "dblclick" dan "mouseover". Dan argumen kedua diisi dengan nama function yang akan dijalankan saat event terjadi.

Berikut contoh penggunaannya:

```
1 <body>
2   <h1 id="judul">Belajar JavaScript</h1>
3   <p></p>
4   <script>
5     function tampilkan(){
6       document.querySelector("p").innerHTML="Judul di klik";
7     }
8
9     var h1Node = document.getElementById("judul");
10    h1Node.addEventListener("click",tampilkan);
11  </script>
12 </body>
```

Kode HTML yang saya pakai sama persis seperti contoh-contoh sebelumnya, yakni terdiri dari tag `<h1>` dan sebuah tag `<p>` yang berfungsi sebagai *placeholder* atau penampung hasil ketika event `click` terjadi.

Di dalam JavaScript terdapat function `tampilkan()` yang isinya akan mengubah teks tag `<p>` menjadi "Judul di klik". Setelah itu perintah `document.getElementById("judul")` digunakan untuk mengisi variabel `h1Node` dengan element node dari tag `<h1>`.

Khusus di baris terakhir, terdapat perintah `h1Node.addEventListener("click",tampilkan)`. Inilah cara mendaftarkan *event handler* menggunakan method `addEventListener()`. Artinya, jalankan fungsi `tampilkan` ketika `h1Node` mengalami event `click`.

Hasilnya, saat tag `<h1>` di klik, akan tampil teks "Judul di klik" di dalam tag `<p>`.

Sama seperti menggunakan property, argument kedua dari method `addEventListener()` juga bisa diinput langsung dengan *anonymous function*:

```
1 <body>
2   <h1 id="judul">Belajar JavaScript</h1>
3   <p></p>
4   <script>
5     var h1Node = document.getElementById("judul");
6
7     h1Node.addEventListener("click",function (){
8       document.querySelector("p").innerHTML="Judul di klik";
9     });
10    </script>
11 </body>
```

Jika anda menggunakan cara seperti ini, hati-hati dengan tanda penutup function. Sangat sering terjadi error dikarenakan tanda kurung atau tanda titik koma yang lupa ditulis.

Untuk menambahkan event-event lain caranya juga sama, seperti contoh berikut:

```
1 <body>
2   <h1 id="judul">Belajar JavaScript</h1>
3   <p></p>
4   <script>
5     function tampilanClick(){
6       document.querySelector("p").innerHTML="Judul di klik";
7     }
8
9     function tampilanDblclick(){
10    document.querySelector("p").innerHTML="Judul di double klik";
11  }
12
13    function tampilanContextmenu(){
14      document.querySelector("p").innerHTML="Judul di klik kanan";
15    }
16
17    var h1Node = document.getElementById("judul");
18
19    h1Node.addEventListener("click",tampilanClick);
20    h1Node.addEventListener("dblclick",tampilanDblclick);
21    h1Node.addEventListener("contextmenu",tampilanContextmenu);
22  </script>
23 </body>
```

Tanpa perlu dijelaskan, saya yakin anda sudah paham maksud kode program diatas. Caranya sangat mirip seperti metode penginputan event ke dalam property yang sudah kita pelajari sebelumnya.

Dalam kode JavaScript, saya membuat 3 buah event handler untuk tag `<h1>`, yakni event saat tag di **click**, di **double click** dan **right click**. Untuk setiap event ini akan dijalankan fungsi yang berbeda-beda.

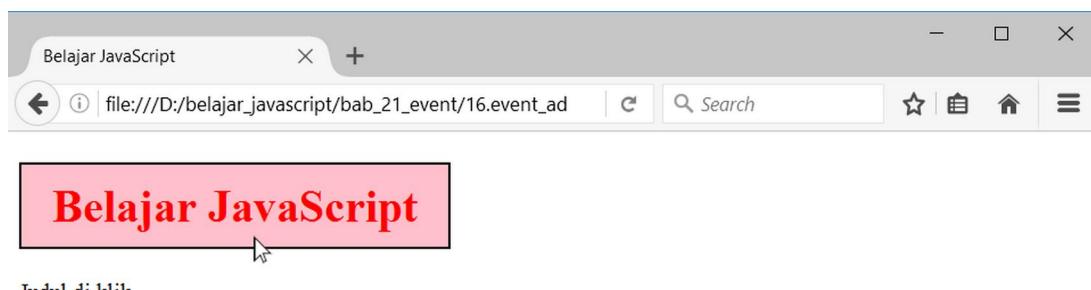
Sekarang, mari kita test contoh kasus yang sebelumnya tidak bisa ditangani dengan cara menginputan event dari property, yakni menggabungkan 2 buah event:

```
1 <body>
2   <h1 id="judul">Belajar JavaScript</h1>
3   <p></p>
4   <h3></h3>
5   <script>
6     function tampilanClickP(){
7       document.querySelector("p").innerHTML="Judul di klik";
8     }
9     function tampilanClickH3(){
10    document.querySelector("h3").innerHTML="Judul di klik";
11  }
12
```

```
13     var h1Node = document.getElementById("judul");
14     h1Node.addEventListener("click",tampilkanClickP);
15     h1Node.addEventListener("click",tampilkanClickH3);
16 </script>
17 </body>
```

Dalam dua baris terakhir, saya menulis 2 kali method `addEventListener()` untuk event yang sama, yakni `click`. Perbedaannya hanya dari fungsi yang dijalankan. Fungsi `tampilkanClickP()` akan menampilkan teks di tag `<p>`, sedangkan fungsi `tampilkanClickH3()` akan menampilkan teks di tag `<h3>`.

Sekarang, apakah kedua fungsi ini bisa dijalankan sekaligus? atau hanya salah satu saja? Mari kita coba:



Gambar: Teks tampil di dua tempat

Ternyata tampil 2 teks, yang artinya kedua fungsi diatas bisa berjalan. Inilah keunggulan dari method `addEventListener()`, dimana kita bisa menambahkan fungsi-fungsi baru untuk event yang sama.

Jika sudah tidak dibutuhkan, kita juga bisa menghapus salah satu function dari event tersebut, dan function lain tetap akan berjalan normal. Berikut contohnya:

```
1 <body>
2   <h1 id="judul">Belajar JavaScript</h1>
3   <p></p>
4   <h3></h3>
5   <script>
6     function tampilkanClickP(){
7       document.querySelector("p").innerHTML="Judul di klik";
8     }
9     function tampilkanClickH3(){
10       document.querySelector("h3").innerHTML="Judul di klik";
11     }
12
13   var h1Node = document.getElementById("judul");
14   h1Node.addEventListener("click",tampilkanClickP);
```

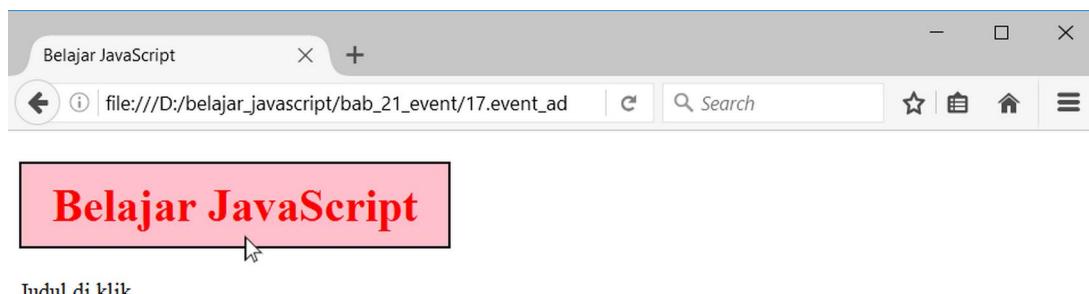
```
15 h1Node.addEventListener("click",tampilkanClickH3);  
16  
17 h1Node.removeEventListener("click",tampilkanClickH3);  
18 </script>  
19 </body>
```

Dalam contoh diatas, saya menginput 2 kali event **click** ke dalam **h1Node**. Ini dilakukan dengan 2 buah perintah:

```
1 h1Node.addEventListener("click",tampilkanClickP);  
2 h1Node.addEventListener("click",tampilkanClickH3);
```

Di baris terakhir terdapat method **removeEventListener()** yang digunakan untuk menghapus event.

Ketika perintah **h1Node.removeEventListener("click",tampilkanClickH3)** diproses, event yang terhapus hanyalah untuk function **tampilkanClickH3()**. Untuk event click **tampilkanClickP()** tetap diproses seperti biasa.



Gambar: Hanya satu function yang ikut terhapus

Syarat lain untuk method **removeEventListener()** adalah, function yang akan dihapus harus sama dengan method **addEventListener()** ketika proses penambahan event tersebut. Dalam contoh ini perintah **h1Node.removeEventListener("click",tampilkanClickH3)** adalah kebalikan dari perintah **h1Node.addEventListener("click",tampilkanClickH3)**



Sedikit catatan untuk method **addEventListener()** dan **removeEventListener()**. Walaupun kedua method ini adalah cara yang paling disarankan dan sesuai dengan standar W3C, tapi keduanya belum di dukung oleh web browser Internet Explorer 8 ke bawah.

Jika anda butuh membuat aplikasi JavaScript untuk IE8 kebawah (yang saat ini cukup jarang dipakai), bisa menggunakan cara penginputan event lain, atau menggunakan method pengganti di IE, yakni **attachEvent()** dan **detachEvent()**.

## 21.5 Event Object

Sampai disini kita telah mengenal 3 cara penginputan *event handler* ke dalam HTML: diinput dari atribut tag HTML, diinput sebagai property element node, dan menggunakan method `addEventListener()`. Karena cara yang paling disarankan adalah menggunakan method `addEventListener()`, inilah yang akan banyak saya pakai.

Melanjutkan materi tentang event, kita akan membahas tentang **event object**. Apa itu *event object*?

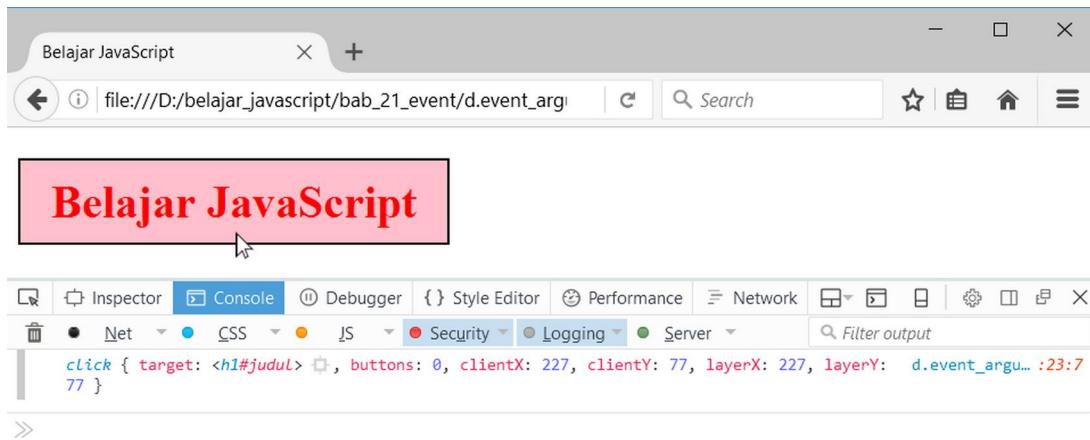
**Event object** adalah object khusus yang dibuat secara otomatis oleh web browser ketika event terjadi. Object ini berisi berbagai informasi terkait event tersebut, seperti element apa yang diklik, tombol apa yang diketik, posisi dari cursor mouse, dsb.

Berikut contoh penggunaan dari **event object**:

```
1 <body>
2   <h1 id="judul">Belajar JavaScript</h1>
3   <script>
4     var h1Node = document.getElementById("judul");
5
6     h1Node.addEventListener("click", function (event) {
7       console.log(event);
8     });
9   </script>
10 </body>
```

Perhatikan penulisan *event handler* untuk event **click**. Dalam bagian function, saya menginput sebuah argumen dengan nama **event**, kemudian menampilkannya dengan perintah `console.log(event)`. Artinya, ketika tag `<h1>` diklik, perintah `console.log(event)` akan dijalankan.

Yang cukup unik, biasanya argumen sebuah fungsi berisi variabel yang telah difenisikan atau sudah memiliki nilai, tapi disini variabel **event** langsung saya input tanpa diisi nilai apapun. Nilai dari argument **event** akan di generate otomatis oleh web browser. Berikut hasilnya:



Gambar: Tampilan dari isi event object

Hasil yang ada di dalam tab **Console** merupakan isi dari **event object**. Terlihat ada property **target** yang berisi element node `<h1#judul>`, property **buttons** dengan nilai 0, **clientX** bernilai 227, **clientY** bernilai 77, **layerX** bernilai 227, dan **layerY** yang bernilai 77.

Semua ini merupakan property dari **event object**. Jika anda menjalankan kode program diatas, bisa jadi nilainya akan sedikit berbeda.

Nama argumen untuk **event object** bisa apa saja, tidak harus event, tapi bisa juga variabel lain seperti contoh berikut:

```
1 h1Node.addEventListener("click", function (sebuahObject) {
2   console.log(sebuahObject);
3 });
```

Disini saya menggunakan variabel `sebuahObject` sebagai penampung **event object**. Kesimpulannya, apapun nama variabel yang diinput, akan terisi dengan **event object**.

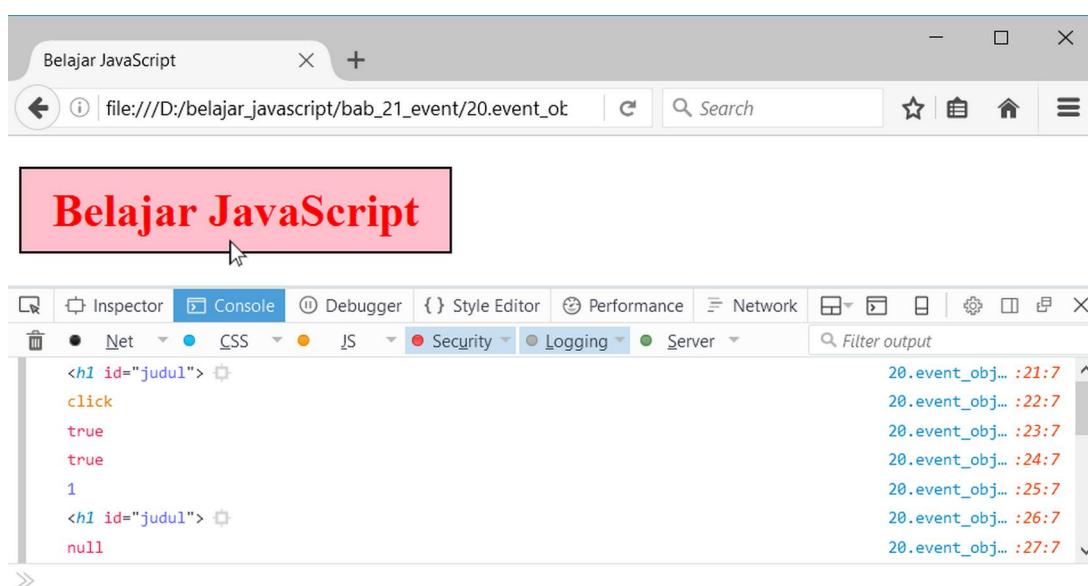
Selanjutnya, karena nilai-nilai **event object** ini berbentuk property, tentunya bisa kita akses menggunakan *dot notation*, mari kita coba:

```
1 <body>
2   <h1 id="judul">Belajar JavaScript</h1>
3   <script>
4     var h1Node = document.getElementById("judul");
5
6     h1Node.addEventListener("click", function (event) {
7       console.log(event.target);           // <h1 id="judul">
8       console.log(event.type);          // click
9       console.log(event.bubbles);        // true
10      console.log(event.cancelable);    // true
11      console.log(event.defaultPrevented); // false
12      console.log(event.detail );       // 1
13      console.log(event.currentTarget); // <h1 id="judul">
14      console.log(event.relatedTarget); // null
```

```

15     console.log(event.screenX);           // 236
16     console.log(event.screenY);           // 165
17     console.log(event.clientX);          // 209
18     console.log(event.clientY);          // 65
19     console.log(event.button);           // 0
20     console.log(event.ctrlKey);          // false
21     console.log(event.shiftKey);         // false
22     console.log(event.altKey);           // false
23   });
24 </script>
25 </body>

```



Gambar: Hasil berbagai property dari event object

Disini saya mengakses berbagai property dari event object. Berikut penjelasan dari nilai-nilai ini:

- **event.target:** Berisi node object tempat event terjadi.
- **event.type:** Tipe dari event yang saat ini dijalankan.
- **event.bubbles:** Berisi nilai boolean apakah event bersifat bubbles (akan kita bahas nantinya).
- **event.defaultPrevented:** Berisi nilai boolean apakah event default sudah dihentikan (akan kita bahas nantinya).
- **event.cancelable:** Berisi nilai boolean apakah event bisa di cancel.
- **event.detail:** Jumlah klik dalam waktu singkat, untuk event dblclick akan berisi 2.
- **event.currentTarget:** Node tempat event ditulis.
- **event.relatedTarget:** Node yang berhubungan dengan event saat ini, hanya berisi nilai untuk event tertentu seperti mouseover, mouseout, mouseenter dan mouseleave.
- **event.screenX:** Posisi cursor mouse pada sumbu x dengan patokan lebar layar.

- **event.screenY**: Posisi cursor mouse pada sumbu y dengan patokan tinggi layar.
- **event.clientX**: Posisi cursor mouse pada sumbu x dengan patokan lebar element.
- **event.clientY**: Posisi cursor mouse pada sumbu y dengan patokan tinggi element.
- **event.button**: Tombol mouse yang di klik, 0 = tombol kiri, 1 = tombol tengah, 2 = tombol kanan.
- **event.ctrlKey**: Berisi nilai boolean apakah tombol keyboard CTRL di tahan saat event click terjadi.
- **event.shiftKey**: Berisi nilai boolean apakah tombol keyboard SHIFT di tahan saat event click terjadi.
- **event.altKey**: Berisi nilai boolean apakah tombol keyboard ALT di tahan saat event click terjadi.

Cukup banyak property yang bisa kita ambil dari event object. Jika anda mencoba klik tag `<h1>` di posisi yang berbeda-beda, nilai dari property `screenX`, `screenY`, `clientX`, dan `clientY` juga akan berubah-ubah. Nilai keempat property ini diambil berdasarkan posisi cursor mouse pada saat event click terjadi.

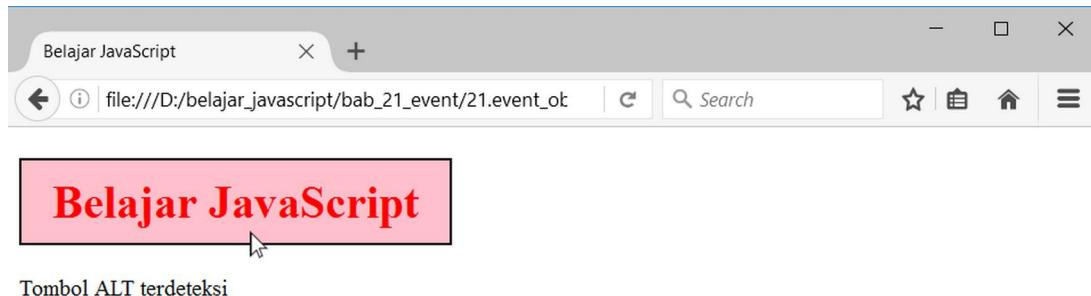
Coba juga tahan tombol CTRL sambil men-klik tag `<h1>`, property `ctrlKey` akan bernilai `true`. Atau tahan tombol ALT dan giliran property `altKey` yang akan bernilai `true`. Informasi ini bisa kita proses sesuai kebutuhan, misalnya ketika user men-klik tag `<h1>` sambil menekan tombol CRTL, tampilkan pesan yang berbeda.

Mari kita coba:

```
1 <body>
2   <h1 id="judul">Belajar JavaScript</h1>
3   <p></p>
4   <script>
5     var h1Node = document.getElementById("judul");
6     pNode = document.querySelector("p");
7
8     h1Node.addEventListener("click", function (event) {
9       if (event.ctrlKey === true){
10         pNode.innerHTML = "Tombol CTRL terdeteksi";
11       }
12       else if (event.altKey === true){
13         pNode.innerHTML = "Tombol ALT terdeteksi";
14       }
15       else {
16         pNode.innerHTML = "Judul di click";
17       }
18     });
19   </script>
20 </body>
```

Disini saya membuat sebuah kondisi **if else**. Jika `event.ctrlKey === true`, jalankan perintah `pNode.innerHTML = "Tombol CTRL terdeteksi"`. Artinya, jika tag `<h1>` di click sambil menahan tombol CRTL, akan tampil teks "Tombol CTRL terdeteksi".

Kemudian jika `event.altKey === true`, tag `<p>` akan menampilkan teks "Tombol ALT terdeteksi". Teks ini akan tampil jika tag `<h1>` di click sambil menahan tombol ALT. Terakhir jika tidak terdeteksi tombol apapun yang ditekan, tampilkan pesan "Judul di click".



Gambar: Tombol ALT terdeteksi pada saat event click terjadi

Daftar property diatas adalah property untuk event **click**. Untuk event yang melibatkan keyboard, tentunya tidak akan ada property `screenX` atau `screenY`, tapi berganti menjadi property `key` untuk mencari info tombol apa yang ditekan. Kita akan membahas event untuk keyboard dalam bab selanjutnya.



Jika anda ingin melihat apa saja property yang bisa diambil untuk event mouse, bisa akses ke [MouseEvent Object<sup>4</sup>](#).

Dalam 3 contoh tentang event object sebelum ini , saya menulis langsung function ke dalam method `addEventListener()`. Agar lebih rapi, kita bisa memindahkan function ke bagian terpisah, seperti contoh berikut:

```

1 <body>
2   <h1 id="judul">Belajar JavaScript</h1>
3   <p></p>
4   <script>
5     var h1Node = document.getElementById("judul");
6     var pNode = document.querySelector("p");
7
8     function tampilkan(event){
9       if (event.ctrlKey === true){
10         pNode.innerHTML = "Tombol CTRL terdeteksi";
11       }
12       else if (event.altKey === true){
13         pNode.innerHTML = "Tombol ALT terdeteksi";
14       }

```

<sup>4</sup><https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent>

```
15     else {
16         pNode.innerHTML = "Judul di click";
17     }
18 }
19
20 h1Node.addEventListener("click", tampilkan);
21 </script>
22 </body>
```

Dibaris terakhir terdapat perintah `h1Node.addEventListener("click", tampilkan);`. Artinya ketika `h1Node` di **click**, jalankan function `tampilkan()`.

Isi dari function `tampilkan()` sama seperti sebelumnya, namun perhatikan penambahan argumen event, inilah cara penulisan argumen yang akan diisi dengan **event object** oleh web browser.

Dari seluruh daftar property **event object**, yang paling penting adalah property **target**. Property ini berisi element node dimana event terjadi. Loh, bukannya sudah jelas event berlangsung di tag `<h1>`?

Betul, untuk contoh yang melibatkan 1 element, nilai dari property `target` sama dengan nilai property `currentTarget`, yakni lokasi node dimana *event handler* ditulis. Namun akan berbeda jika terdapat beberapa element yang menggunakan 1 function yang sama, seperti contoh berikut:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <meta charset="utf-8">
5     <title> Belajar JavaScript </title>
6     <style>
7         h1, h2, div { border: 2px solid black;
8             width: 280px;
9             padding: 10px;
10            text-align: center;
11            color: red;
12            background-color: pink; }
13     </style>
14 </head>
15 <body>
16     <h1 id="judul">Belajar HTML</h1>
17     <h2 id="judul2">Belajar CSS</h2>
18     <div id="tagDiv">Belajar JavaScript</div>
19     <p id="placeholder"></p>
20     <script>
21         var h1Node = document.getElementById("judul");
22         var h2Node = document.getElementById("judul2");
23         var divNode = document.getElementById("tagDiv");
```

```
24 var pNode = document.getElementById("placeholder");
25
26 function tampilkan(e){
27     pNode.innerHTML = e.target.nodeName;
28 }
29
30 h1Node.addEventListener("click",tampilkan);
31 h2Node.addEventListener("click",tampilkan);
32 divNode.addEventListener("click",tampilkan);
33 </script>
34 </body>
35 </html>
```



Gambar: Hasil dari property target.nodeName

Untuk contoh kali ini saya membuat tiga buah element yang akan dijadikan target *event*, yakni `<h1>`, `<h2>`, dan `<div>`. Ketiganya di style menggunakan kode CSS yang sama. Di akhir kode program saya membuat 3 event handler untuk `click`. Masing-masingnya akan menjalankan function `tampilkan()`.

Mari kita bahas apa yang akan dijalankan oleh function `tampilkan()`. Sebagai argumen penampung **event object**, saya menggunakan huruf `e` yang merupakan singkatan dari *event*. Penggunaan huruf `e` untuk **event object** cukup sering dipakai dan menjadi kebiasaan programmer JavaScript (walaupun tidak harus). Sama seperti kebiasaan menggunakan huruf `i` untuk variabel counter *looping*.

Di dalam function `tampilkan()`, hanya terdapat 1 baris: `pNode.innerHTML = e.target.nodeName`. Perintah ini artinya, input nilai `e.target.nodeName` ke dalam tag `<p>`.

Berdasarkan penjelasan sebelumnya, property `target` akan mengembalikan element node tempat dimana event berlangsung. Dengan demikian, kita bisa mengakses property apapun yang biasanya ada di sebuah element node . Property `nodeName` akan berisi informasi nama node yang saat ini sedang di klik.

Hasilnya, ketika tag <h1> di klik, akan tampil teks H1, saat tag <h2> di klik, akan tampil teks H2, begitu juga dengan tag <div> yang jika di klik akan menampilkan teks DIV.

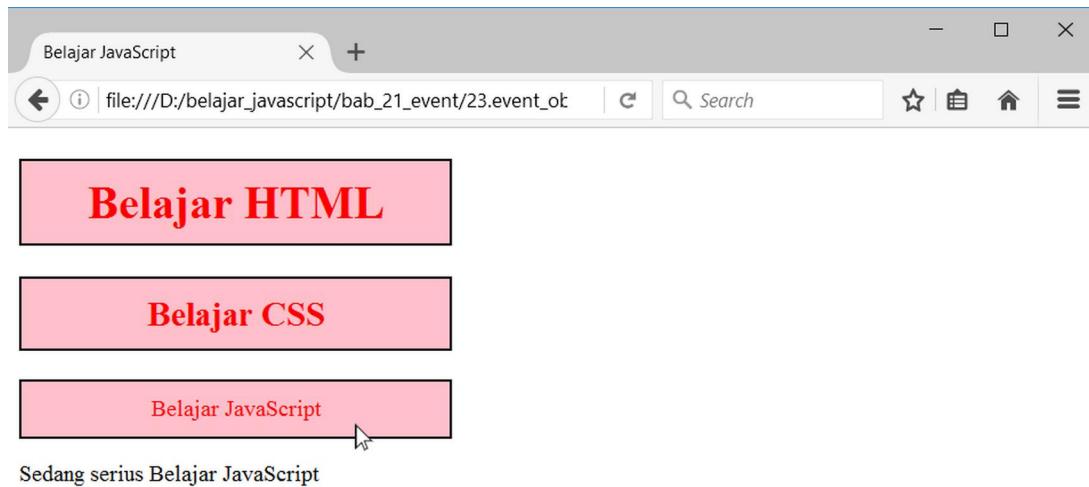
Dengan metode seperti ini, kita bisa membuat 1 buah function untuk menghandle berbagai element node, kemudian mencari tahu element apa yang mengalami *event* dari property `event.target`.

Agar lebih paham, mari kita coba contoh kedua, dapatkah anda memahami arti kode program berikut?

```
1 <body>
2   <h1 id="judul">Belajar HTML</h1>
3   <h2 id="judul2">Belajar CSS</h2>
4   <div id="tagDiv">Belajar JavaScript</div>
5   <p id="placeholder"></p>
6   <script>
7     var h1Node = document.getElementById("judul");
8     var h2Node = document.getElementById("judul2");
9     var divNode = document.getElementById("tagDiv");
10    var pNode = document.getElementById("placeholder");
11
12    function tampilkan(e){
13      var pesan = "Sedang serius "+ e.target.innerHTML;
14      pNode.innerHTML = pesan;
15    }
16
17    h1Node.addEventListener("click",tampilkan);
18    h2Node.addEventListener("click",tampilkan);
19    divNode.addEventListener("click",tampilkan);
20  </script>
21 </body>
```

Program diatas sangat mirip seperti sebelumnya. Perubahan hanya ada di dalam function `tampilkan()`. Di dalam function ini, saya membuat sebuah variabel `pesan` yang akan diisi nilai "Sedang serius "+ `e.target.innerHTML`.

Artinya, jika saat ini yang di klik adalah tag <h2>, variabel `pesan` akan berisi teks Sedang serius Belajar CSS. Teks Belajar CSS diambil dari `e.target.innerHTML`. Mari kita coba:



Gambar: Teks Sedang serius Belajar JavaScript tampil saat tag <div> di klik

Sebagai latihan selanjutnya, bisakah anda memodifikasi kode program diatas untuk tampilan seperti ini?



Gambar: Latihan penggunaan event.target

Gambar di kiri adalah tampilan awal halaman. Jika tag <h1> di klik, warna background akan berubah menjadi kuning (yellow). Jika tag <h2> di klik, warna background akan berubah menjadi aqua, dan jika tag <div> di klik, warna background akan menjadi silver.

Syaratnya, hanya boleh menggunakan 1 function untuk ketiga element, tidak boleh menggunakan 3 function untuk masing-masing element.

Untuk membuat program seperti ini, kita butuh sebuah kondisi **if else** untuk mengecek tag apa yang saat ini sedang di klik. Jika tag itu H1, ubah property `style.backgroundColor` menjadi `yellow`. Jika tag itu H2, ubah property `style.backgroundColor` menjadi `aqua`, dan jika tag itu DIV, ubah property `style.backgroundColor` menjadi `silver`.

Silahkan anda coba rancang kode programnya.

Baik, berikut kode yang saya gunakan:

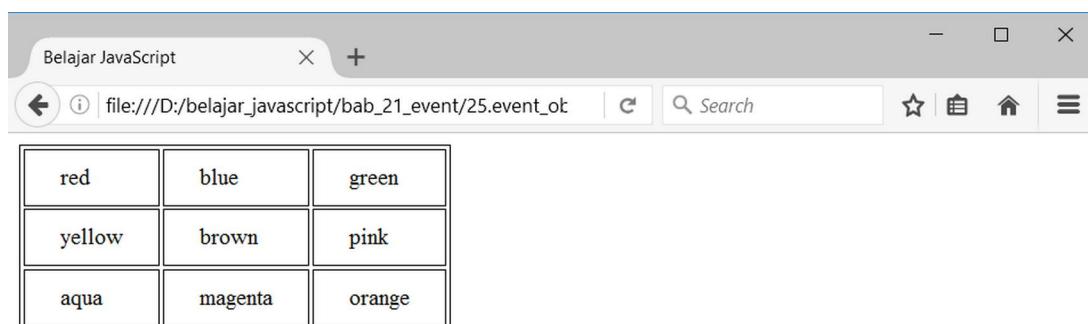
```
1 <body>
2   <h1 id="judul1">Belajar HTML</h1>
3   <h2 id="judul2">Belajar CSS</h2>
4   <div id="tagDiv">Belajar JavaScript</div>
5   <script>
6     var h1Node = document.getElementById("judul1");
7     var h2Node = document.getElementById("judul2");
8     var divNode = document.getElementById("tagDiv");
9
10    function tampilkan(e){
11      var idTarget = e.target.getAttribute("id");
12      if (idTarget === "judul1"){
13        e.target.style.backgroundColor = "yellow";
14      }
15      else if (idTarget === "judul2"){
16        e.target.style.backgroundColor = "aqua";
17      }
18      else if (idTarget === "tagDiv"){
19        e.target.style.backgroundColor = "silver";
20      }
21    }
22
23    h1Node.addEventListener("click",tampilkan);
24    h2Node.addEventListener("click",tampilkan);
25    divNode.addEventListener("click",tampilkan);
26  </script>
27 </body>
```

Di dalam kondisi **if else**, saya mengecek nilai atribut **id** dari **event object**. Nilai ini bisa didapat dari perintah `e.target.getAttribute("id")`. Cara penggunaan method `getAttribute()` sudah kita bahas dalam bab sebelumnya. Nilai inilah yang saya simpan ke dalam variabel `idTarget` dan masuk ke percabangan **if else**. Untuk mengubah warna background, saya cukup membuat `e.target.style.backgroundColor = "kode warna"`.

Mengecek atribut **id** hanya salah satu dari cara membedakan tiap-tiap element. Anda juga bisa menggunakan `e.target.nodeName` untuk memeriksa apakah element yang saat ini sedang di click.

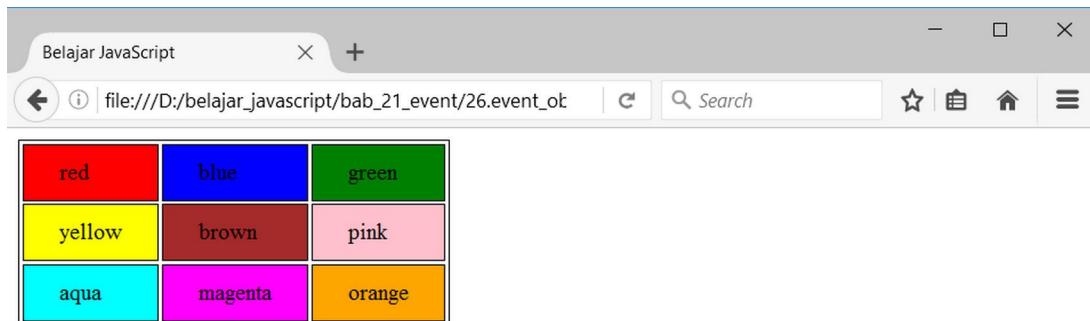
Ingin latihan yang lebih menantang? Mari lanjut. Perhatikan tampilan dan coding berikut:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <style>
7     table,td { border: 1px solid black; }
8     td {padding: 10px 25px; }
9   </style>
10 </head>
11 <body>
12   <table>
13     <tr>
14       <td>red</td><td>blue</td><td>green</td>
15     </tr>
16     <tr>
17       <td>yellow</td><td>brown</td><td>pink</td>
18     </tr>
19     <tr>
20       <td>aqua</td><td>magenta</td><td>orange</td>
21     </tr>
22   </table>
23   <script>
24     // kode javascript disini...
25   </script>
26 </body>
27 </html>
```



Gambar: Tabel dengan teks warna

Disini saya memiliki tabel dengan 9 sel. Masing-masing sel berisi teks dengan *keyword* warna. Tugas anda adalah, membuat event yang jika sel tabel tersebut di klik, warna backgroundnya akan berubah sesuai warna yang terdapat di dalam masing-masing sel, seperti tampilan berikut:



Gambar: Sel tabel berubah warna ketika di klik

Petunjuknya, kita bisa mengambil nilai teks setiap sel (`innerHTML`), kemudian input ke property `style.backgroundColor` dari sel tersebut. Event handler cukup di letakkan ke dalam `<table>`, dan gunakan `event.target` untuk menangani event `click` dari tiap sel tabel.

Silahkan anda coba merancang codingnya sebentar.

Baik, berikut kode yang saya gunakan:

```

1 var tableNode = document.querySelector("table");
2
3 function ubahWarna(e){
4     e.target.style.backgroundColor = e.target.innerHTML;
5 }
6
7 tableNode.addEventListener("click", ubahWarna);

```

Yup, tidak perlu panjang-panjang. Kita hanya butuh 1 baris di dalam *event handler* `ubahWarna()`, yakni `e.target.style.backgroundColor = e.target.innerHTML`.

Event `click` di “tempel” ke dalam `tableNode` dengan perintah `tableNode.addEventListener("click", ubahWarna)`. Ketika sebuah sel tabel di click, property `e.target` otomatis berisi element node dimana terjadi klik. Selanjutnya kita tinggal mengolah isi teks sel tersebut dan menginputnya ke dalam atribut `style`.

Jika kode program ini kurang anda pahami, saya sarankan untuk mencoba membuat sendiri contoh-contoh lain. Usahakan anda benar-benar *ngerti* apa itu `event.target`, karena sangat sering di gunakan dalam pembuatan program JavaScript.

## 21.6 Event Propagation

**Event propagation** adalah istilah yang merujuk kepada cara event “ditangkap” oleh web browser. Sebagai contoh, dalam latihan untuk mewarnai tabel sebelum ini, saya meletakkan event `click` ke dalam tag `<table>`, namun jika yang di klik adalah sel tabel, isi dari property `e.target` adalah tag `<td>`, bukan tag `<table>` itu sendiri.

Atau dengan sudut pandang lain, ketika tag `<h1>` di klik, bukan kah kita sebenarnya juga men-klik tag `<body>`? Aspek inilah yang dimaksud dengan **Event propagation**.

Sebagai contoh, silahkan anda pelajari kode program berikut:

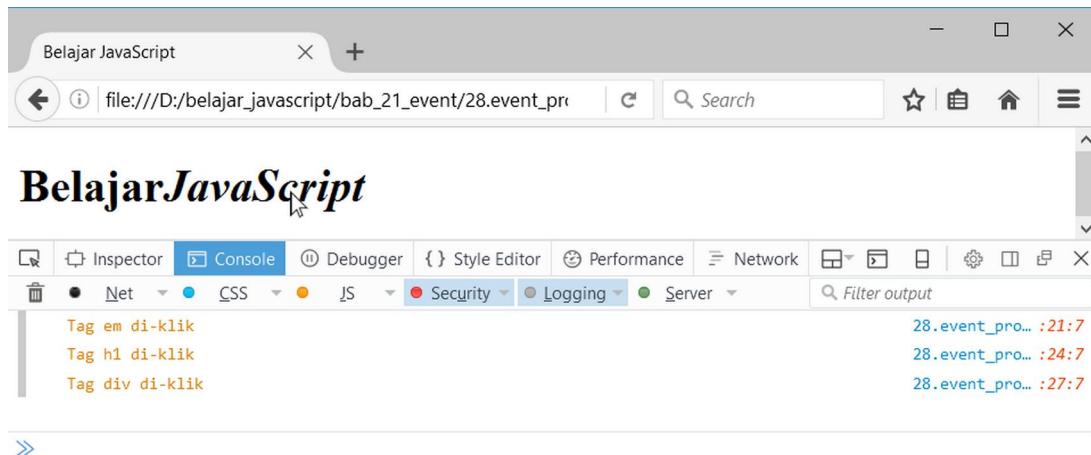
```
1 <body>
2   <div>
3     <h1>Belajar<em>JavaScript</em></h1>
4   </div>
5   <script>
6     var emNode = document.querySelector("em");
7     var h1Node = document.querySelector("h1");
8     var divNode = document.querySelector("div");
9
10    function tampilkanEm (){
11      console.log("Tag em di-klik");
12    }
13    function tampilkanH1 (){
14      console.log("Tag h1 di-klik");
15    }
16    function tampilkanDiv (){
17      console.log("Tag div di-klik");
18    }
19
20    divNode.addEventListener("click",tampilkanDiv);
21    emNode.addEventListener("click",tampilkanEm);
22    h1Node.addEventListener("click",tampilkanH1);
23  </script>
24 </body>
```

Untuk kode HTML, saya membuat struktur element dengan tag `<em>` berada di dalam tag `<h1>`, dimana tag `<h1>` juga ada di dalam tag `<div>`. Ini sama artinya bahwa tag `<div>` adalah *parent element* dari tag `<h1>`, dan tag `<h1>` juga *parent element* dari tag `<em>`.

Untuk kode JavaScript, ketiga tag ini memiliki event `click` yang akan menampilkan pesan ke `console.log()`. Isi pesan ini berbeda-beda untuk setiap tag.

Pertanyaannya, jika tag `<em>` di klik, apakah hasil dari tab `console`? Perhatikan bahwa jika yang di klik adalah tag `<em>`, yakni tulis "JavaScript", secara tidak langsung kita juga men-klik tag `<h1>` dan `<div>`.

Berikut hasilnya:

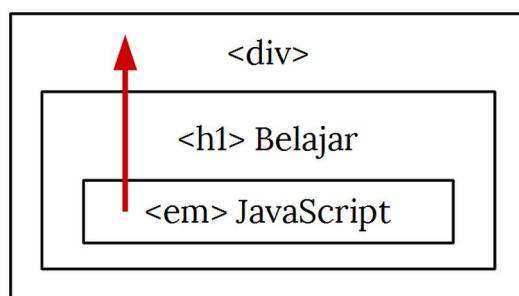


Gambar: Hasil ketika tag <em> di klik

Ternyata tampil 3 teks, dimulai dari "Tag em di-klik", diikuti teks "Tag h1 di-klik", dan terakhir "Tag div di-klik". Artinya, event di jalankan dari element paling dalam yakni tag <em>, hingga element terluar, yakni tag <div>. Seandainya saya membuat event click untuk tag <body>, teks tersebut akan berada di urutan paling akhir.

Urutan event ini dikenal dengan istilah: **bubbling**. Dalam bahasa inggris, *bubble* berarti gelembung. Sebuah gelembung akan mulai dari bawah ke atas.

## Event Propagation: Bubbling

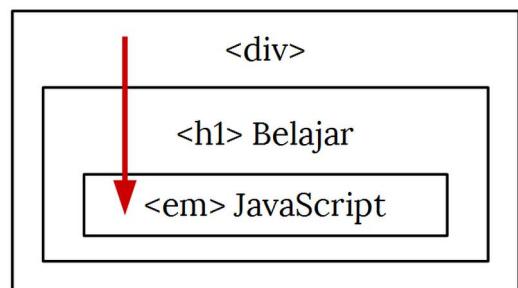


Gambar: Ilustrasi eksekusi event dengan cara “bubbling”

**Bubbling** merupakan mekanisme default yang dijalankan untuk **event propagation**, dimana event akan mulai dari element terdalam, kemudian naik secara berurutan ke element terluar.

Selain *bubbling*, terdapat istilah lain, yakni **Capturing**. **Capturing** adalah kebalikan dari **Bubbling**, dimana event akan dijalankan dari yang paling terluar, baru turun ke element paling dalam.

## Event Propagation: Capturing

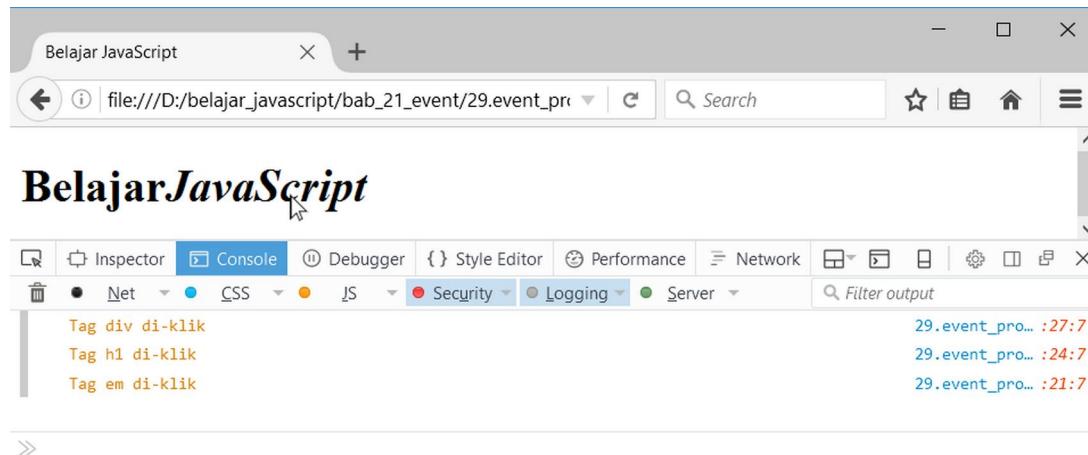


Gambar: Ilustrasi eksekusi event dengan cara “capturing”

Bagaimana caranya? Jika kita ingin event berprilaku seperti ini, input nilai **true** ke dalam argumen ketiga dari method `addEventListener()`, seperti contoh berikut:

```
1 <body>
2   <div>
3     <h1>Belajar<em>JavaScript</em></h1>
4   </div>
5   <script>
6     var emNode = document.querySelector("em");
7     var h1Node = document.querySelector("h1");
8     var divNode = document.querySelector("div");
9
10    function tampilkanEm (){
11      console.log("Tag em di-klik");
12    }
13    function tampilkanH1 (){
14      console.log("Tag h1 di-klik");
15    }
16    function tampilkanDiv (){
17      console.log("Tag div di-klik");
18    }
19
20    divNode.addEventListener("click",tampilkanDiv,true);
21    emNode.addEventListener("click",tampilkanEm,true);
22    h1Node.addEventListener("click",tampilkanH1,true);
23  </script>
24 </body>
```

Perhatikan penambahan nilai **true** di 3 baris terakhir. Mari klik klik tag `<em>`:



Gambar: Hasil ketika tag <em> di klik

Sekarang, urutan event yang dijalankan sudah bertukar. Saat tag <em> di klik, yang tampil lebih dulu adalah "Tag div di-klik", diikuti dengan teks "Tag h1 di-klik" dan terakhir "Tag em di-klik".

Implementasi dari *event propagation bubbling* maupun *capturing* mungkin tidak terlalu banyak dipakai, tapi konsep dasarnya cukup penting untuk dipahami. Bisa jadi anda membuat kode program yang berprilaku "aneh" karena event-nya ternyata di "tangkap" oleh element anak dari tag tersebut.

Dalam beberapa situasi, efek event propagation ini cukup bermanfaat. Contohnya seperti table warna yang kita rancang sebelumnya. Cukup membuat 1 event untuk tag <table>, seluruh tag <td> yang ada di dalam tabel otomatis punya event yang sama.

Tapi bagaimana jika hal seperti ini tidak kita inginkan? **Event object** punya method `stopPropagation()` untuk menghentikannya. Berikut contoh dari penggunaan method ini:

```
1 <body>
2   <div>
3     <h1>Belajar<em>JavaScript</em></h1>
4   </div>
5   <script>
6     var emNode = document.querySelector("em");
7     var h1Node = document.querySelector("h1");
8     var divNode = document.querySelector("div");
9
10    function tampilkanEm (e){
11      console.log("Tag em di-klik");
12      e.stopPropagation();
13    }
14    function tampilkanH1 (){
15      console.log("Tag h1 di-klik");
16    }
17    function tampilkanDiv (){
18      console.log("Tag div di-klik");
```

```
19     }
20
21     divNode.addEventListener("click",tampilkanDiv);
22     emNode.addEventListener("click",tampilkanEm);
23     h1Node.addEventListener("click",tampilkanH1);
24     </script>
25 </body>
```

Perhatikan isi dari function `tampilkanEm()`, terdapat baris perintah `e.stopPropagation()`. Sekarang, ketika tag `<em>` di klik, efek *event propagation* tidak akan terjadi. Event `click` tidak akan dikirim ke tag `<h1>` maupun tag `<div>`. Ini berlaku untuk metode *bubbling* maupun *capturing*.



Perbedaan antara *bubbling* dan *capturing* sebenarnya berasal dari era IE vs Netscape. Web browser IE menggunakan *event bubbling*, sedangkan Netscape memilih *event capturing*. IE 8 ke bawah malah hanya mendukung *event bubbling* saja.

Saat ini semua web browser modern mendukung kedua metode ini dan menjadikan *event bubbling* sebagai prilaku standar (termasuk ke dalam spesifikasi DOM W3C).

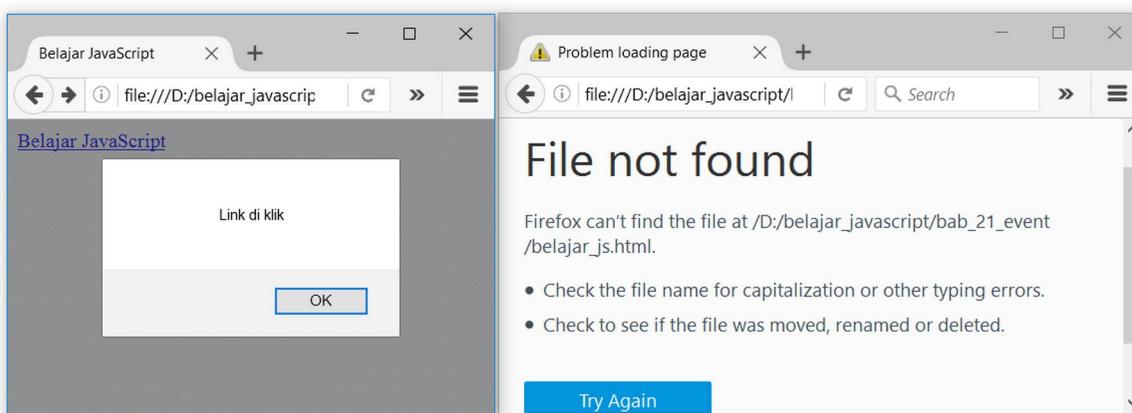
## 21.7 Menghentikan Event Default

Beberapa tag HTML memiliki event bawaan. Sebagai contoh, link yang dibuat dari tag `<a>` memiliki event `click` yang akan membuat halaman saat ini pindah ke halaman lain. Jika kita menginput event `click` lagi ke dalam tag `<a>`, hasilnya akan “bergabung”.

Berikut percobaannya:

```
1 <body>
2   <a href="belajar_js.html">Belajar JavaScript </a>
3   <script>
4     function tampilkan(){
5       alert("Link di klik");
6     }
7
8     var aNode = document.querySelector("a");
9     aNode.addEventListener("click",tampilkan);
10    </script>
11 </body>
```

Saya membuat sebuah link dari tag `<a>`, kemudian memberikan event `click`. Event `click` ini akan menampilkan sebuah jendela `alert`. Berikut hasilnya:



Gambar: Akan tampil jendela alert, kemudian halaman pindah

Saat link di klik, jendela alert "Link di klik" akan tampil. Begitu tombol OK di klik, halaman langsung berpindah ke `belajar_js.html`. Hasilnya akan error karena halaman ini memang tidak tersedia.

Sekarang bagaimana caranya untuk menghentikan event bawaan dari HTML? Saya ingin ketika link di klik, halaman tidak berpindah dan hanya menampilkan jendela alert saja.

Solusinya adalah dengan menggunakan method `preventDefault()` kepuanyaan **event object**. Berikut contoh penggunaannya:

```

1 <body>
2   <a href="belajar_js.html">Belajar JavaScript </a>
3   <script>
4     function tampilkan(e){
5       alert("Link di klik");
6       e.preventDefault();
7     }
8
9     var aNode = document.querySelector("a");
10    aNode.addEventListener("click",tampilkan);
11  </script>
12 </body>
```

Dengan memanggil perintah `e.preventDefault()`, method bawaan tag `<a>` akan dihentikan. Efeknya, link tidak akan berfungsi.

Sekilas, menghentikan link seperti ini tidak terlalu bermanfaat. Toh kalau efek linknya dihapus, tag `<a>` tersebut akan sama seperti teks biasa. Namun dalam situasi tertentu, sangat diperlukan.

Dalam bab berikutnya, kita akan membahas event untuk form. Secara default, jika tombol submit form di klik, isi form langsung dikirim ke server untuk diproses. Menggunakan JavaScript, kita bisa melakukan pengecekan di sisi client terlebih dahulu (di dalam web browser). Jika ditemui data yang salah, jalankan `preventDefault()`. Dengan demikian, form tidak jadi di submit.

## 21.8 Mouse Events

Mouse merupakan alat interaksi yang paling banyak dipakai dalam mengakses halaman web. Karena itu pula mouse event perlu kita bahas dengan lebih detail.

Berikut daftar **mouse event** yang bisa diprogram menggunakan JavaScript:

- **click**: Event yang terjadi saat tombol kiri mouse di klik.
- **contextmenu**: Event yang terjadi saat tombol kanan mouse di klik.
- **dblclick**: Event yang terjadi saat tombol kiri mouse di klik 2x dengan cepat.
- **mousedown**: Event yang terjadi saat tombol mouse di tekan.
- **mouseup**: Event yang terjadi saat tombol mouse di lepaskan.
- **mouseenter**: Event yang terjadi saat cursor mouse masuk ke dalam area element.
- **mouseleave**: Event yang terjadi saat cursor mouse keluar dari area element.
- **mouseover**: Event yang terjadi saat cursor mouse masuk ke dalam area sebuah element atau ke child dari element tersebut.
- **mouseout**: Event yang terjadi saat cursor mouse keluar dari area element atau ke child dari element tersebut.
- **mousemove**: Event yang terjadi saat cursor mouse digerakkan dalam element.

Event *click*, *contextmenu*, dan *dblclick* sudah beberapa kali digunakan dan tidak akan saya bahas lagi. Kita akan lihat contoh praktek untuk 7 event lainnya.

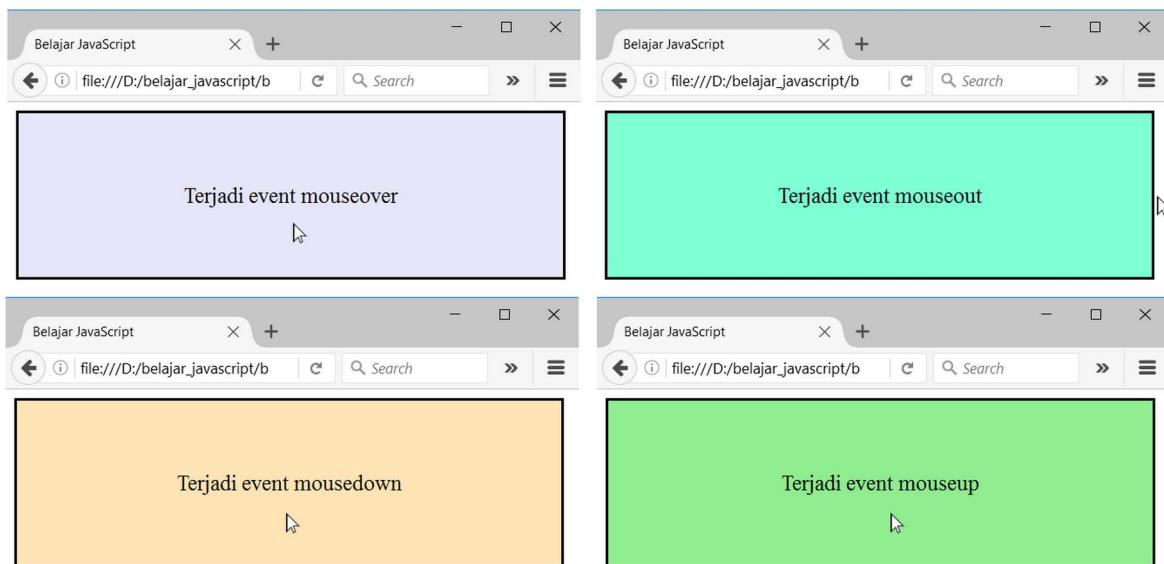
### Event mousedown, mouseup, mouseover dan mouseout

Langsung saja kita praktek dengan kode program. Dalam contoh berikut saya menggunakan 4 event: *mousedown*, *mouseup*, *mouseover* dan *mouseout*:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <style>
7     div { border: 3px solid black;
8           width: 500px;
9           height: 150px;
10          line-height: 150px;
11          text-align: center;
12          font-size: 20px; }
13 </style>
14 </head>
15 <body>
16   <div>Cursor mouse ke sini</div>
```

```
17 <script>
18     function mouseoverEvent(e){
19         e.target.style.backgroundColor = "Lavender";
20         e.target.innerHTML = "Terjadi event mouseover";
21     }
22
23     function mouseoutEvent(e){
24         e.target.style.backgroundColor = "Aquamarine";
25         e.target.innerHTML = "Terjadi event mouseout";
26     }
27
28     function mousedownEvent(e){
29         e.target.style.backgroundColor = "Moccasin";
30         e.target.innerHTML = "Terjadi event mousedown";
31     }
32
33     function mouseupEvent(e){
34         e.target.style.backgroundColor = "LightGreen";
35         e.target.innerHTML = "Terjadi event mouseup";
36     }
37
38     var divNode = document.querySelector("div");
39     divNode.addEventListener("mouseover",mouseoverEvent);
40     divNode.addEventListener("mouseout",mouseoutEvent);
41     divNode.addEventListener("mousedown",mousedownEvent);
42     divNode.addEventListener("mouseup",mouseupEvent);
43 </script>
44 </body>
45 </html>
```

Kode program ini cukup panjang, tapi semuanya sudah kita bahas. Saya membuat sebuah tag `<div>` yang di style menggunakan kode CSS. Tag ini kemudian di “tempel” dengan 4 event: *mousedown*, *mouseup*, *mouseover* dan *mouseout*. Untuk setiap event, lakukan 2 hal: ubah isi teks dari tag `<div>`, dan tukar warna background.



Gambar: Contoh mousedown, mouseup, mouseover dan mouseout event

Silahkan anda coba arahkan mouse ke dalam tag <div>. Pada saat cursor mouse masuk ke element ini, event *mouseover* akan dijalankan 1 kali, hasilnya teks berubah menjadi "Terjadi event mouseover" dan warna background berubah menjadi "Lavender" (ungu muda).

Kemudian coba tekan dan tahan sebentar tombol mouse di dalam tag <div>, event *mousedown* akan berjalan. Hasilnya teks berubah menjadi "Terjadi event mousedown" dan warna background berubah menjadi "Moccasin" (kuning muda).

Saat tombol mouse dilepaskan, akan terjadi event *mouseup*. Hasilnya teks berubah menjadi "Terjadi event mouseup" dan warna background berubah menjadi "LightGreen" (hijau muda).

Sampai disini, jika anda menggerakkan cursor mouse di atas element, warna background tetap berwarna "LightGreen" yang berasal dari event *mouseup*. Artinya, efek *mouseover* tidak terjadi lagi.

Jika cursor mouse dibawa keluar dari element, teks akan berubah menjadi "Terjadi event mouseout" dan warna background berubah menjadi "Aquamarine" (biru muda). Ini terjadi karena event *mouseout* di panggil saat cursor keluar dari tag <div>.

Inilah cara penggunaan dari event *mousedown*, *mouseup*, *mouseover* dan *mouseout*.

## Event **mouseenter** dan **mouseleave**

Dari daftar mouse event, terdapat 2 pasang event yang sangat mirip, yakni *mouseover* dengan *mouseenter* serta *mouseout* dengan *mouseleave*.

Event *mouseover* dan *mouseenter* sama-sama aktif ketika cursor mouse berada masuk ke dalam sebuah element, sedangkan event *mouseout* dan *mouseleave* akan aktif ketika cursor mouse keluar dari element tersebut. Perbedaannya ada di cara menangani **child element**.

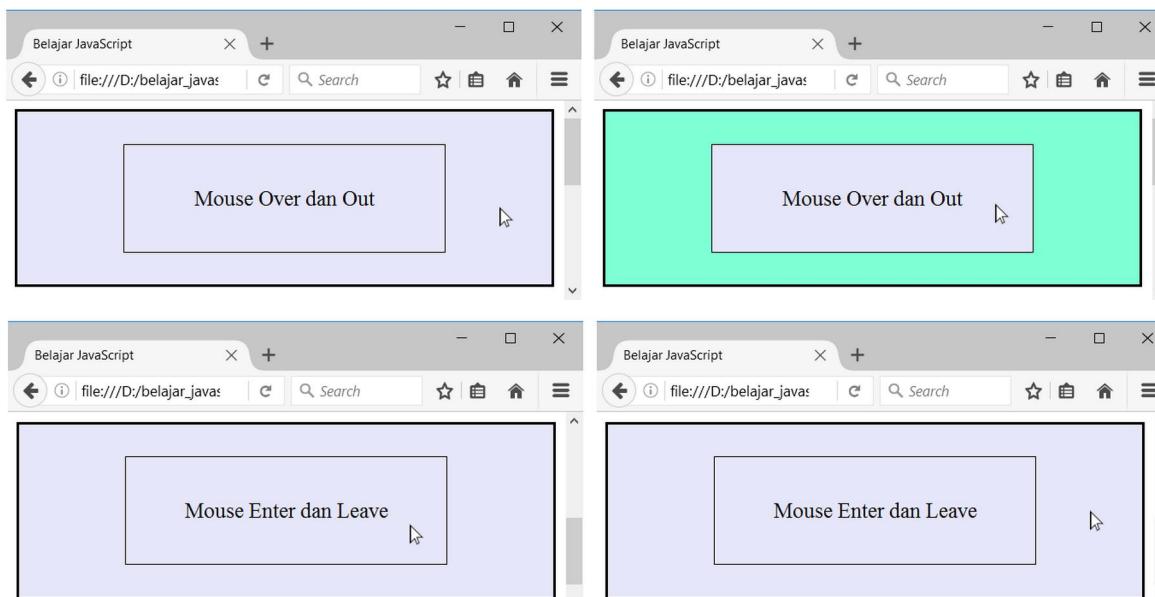
Berikut contoh prakteknya:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <style>
7     div { border: 3px solid black;
8           width: 500px;
9           font-size: 20px;
10          margin-bottom: 30px;
11         }
12    p { border: 1px solid black;
13      width: 300px;
14      height: 100px;
15      line-height: 100px;
16      text-align: center;
17      font-size: 20px;
18      margin: 30px auto;
19     }
20   </style>
21 </head>
22 <body>
23   <div id="div1"><p>Mouse Over dan Out</p></div>
24   <div id="div2"><p>Mouse Enter dan Leave</p></div>
25   <script>
26     function keLavender(e){
27       e.target.style.backgroundColor = "Lavender";
28     }
29
30     function keAquamarine(e){
31       e.target.style.backgroundColor = "Aquamarine";
32     }
33
34     var divNode1 = document.getElementById("div1");
35     divNode1.addEventListener("mouseover",keLavender);
36     divNode1.addEventListener("mouseout",keAquamarine);
37
38     var divNode2 = document.getElementById("div2");
39     divNode2.addEventListener("mouseenter",keLavender);
40     divNode2.addEventListener("mouseleave",keAquamarine);
41   </script>
42 </body>
43 </html>
```

Kali ini saya menggunakan dua buah tag `<div>`, dimana dalam masing-masing tag `<div>` terdapat tag `<p>` yang berperan sebagai **child** dari tag `<div>`. Semuanya di design dengan CSS

agar berbentuk kotak persegi panjang.

Untuk kotak pertama (posisi atas), saya membuat event *mouseover* dan *mouseout*. Sedangkan untuk kotak kedua (posisi bawah) di tempatkan event *mouseenter* dan *mouseleave*. Semua event ini akan mengubah warna background. Mari kita lihat apa yang terjadi:



Gambar: Event *mouseover* dan *mouseout* vs *mouseenter* dan *mouseleave*

Dua gambar di bagian atas adalah efek yang terjadi untuk event *mouseover* dan *mouseout*. Ketika cursor mouse berada di atas tag `<div>`, warna background dari tag `<div>` akan berubah menjadi "Lavender". Ini berasal dari event *mouseover*.

Warna background dari tag `<p>` juga menjadi "Lavender" karena secara default warna background element adalah transparent (tembus pandang). Tag `<p>` ada di dalam tag `<div>`, sehingga warna yang tampil adalah kepunyaan tag `<div>`.

Tapi begitu cursor mouse masuk ke tag `<p>`, event *mouseout* dari tag `<div>` akan aktif, dan event *mouseover* dari tag `<p>` juga aktif. Artinya, saat masuk ke **child element**, baik event *mouseover* dan *mouseout* akan dijalankan.

Hasilnya, warna background dari tag `<p>` akan menjadi "Lavender", sedangkan untuk tag `<div>` akan berubah menjadi "Aquamarine" (sesuai dengan event handler yang ditulis). Saat cursor mouse meninggalkan tag `<p>` dan masuk kembali ke tag `<div>`, event *mouseover* dan *mouseout* dari kedua element akan kembali dipanggil.

Dua kotak bagian bawah adalah hasil dari event *mouseenter* dan *mouseleave*. Saat cursor mouse memasuki tag `<div>`, warna background akan berubah menjadi "Lavender", ini adalah efek dari event *mouseenter*.

Akan tetapi, jika anda menggerakkan cursor mouse ke dalam tag `<p>`, tidak akan terjadi perubahan. Begitu pula saat meninggalkan tag `<p>`, artinya event *mouseenter* dan *mouseleave* tidak menganggap **child element** sebagai element lain (dianggap satu kesatuan dengan parent element). Inilah yang menjadi perbedaan dengan event *mouseover* dan *mouseout*.

## Event mousemove

Event terakhir dari mouse event yang belum kita bahas adalah *mousemove*. Event ini aktif saat cursor mouse di gerakkan di atas element. Berbeda dengan event *mouseover* maupun *mouseenter* yang hanya dijalankan hanya 1 kali (yakni saat memasuki sebuah element), event *mousemove* akan selalu dipanggil setiap cursor mouse bergerak, artinya bisa ribuan kali tergantung berapa jauh cursor mouse digerakkan.

Informasi yang umumnya diambil dari event *mousemove* adalah posisi cursor, yang bisa diambil dari property `screenX`, `screenY`, `clientX` dan `clientY`. Karena event ini selalu dipanggil setiap saat, hasilnya akan selalu berubah-ubah tergantung posisi cursor.

Berikut contoh prakteknya:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <style>
7     div { border: 3px solid black;
8           width: 500px;
9           height: 150px;
10          line-height: 150px;
11          text-align: center;
12          font-size: 20px;
13          background-color: Lavender;
14        }
15   </style>
16 </head>
17 <body>
18   <div>Cursor mouse ke sini</div>
19   <p>
20     Koordinat (screen): X = <span id="sumbuXS"></span>
21     Y = <span id="sumbuYS"></span>
22   </p>
23   <p>
24     Koordinat (client): X = <span id="sumbuXC"></span>
25     Y = <span id="sumbuYC"></span>
26   </p>
27   <script>
28     function mousemoveEvent(e){
29       spanXSNode.innerHTML = e.screenX;
30       spanYSNode.innerHTML = e.screenY;
31       spanXCNode.innerHTML = e.clientX;
32       spanYCNode.innerHTML = e.clientY;
33     }
```

```

34
35     var divNode = document.querySelector("div");
36     var spanXSNode = document.getElementById("sumbuXS");
37     var spanYSNode = document.getElementById("sumbuYS");
38     var spanXCNode = document.getElementById("sumbuXC");
39     var spanYCNode = document.getElementById("sumbuYC");
40
41     divNode.addEventListener("mousemove", mousemoveEvent);
42 </script>
43 </body>
44 </html>

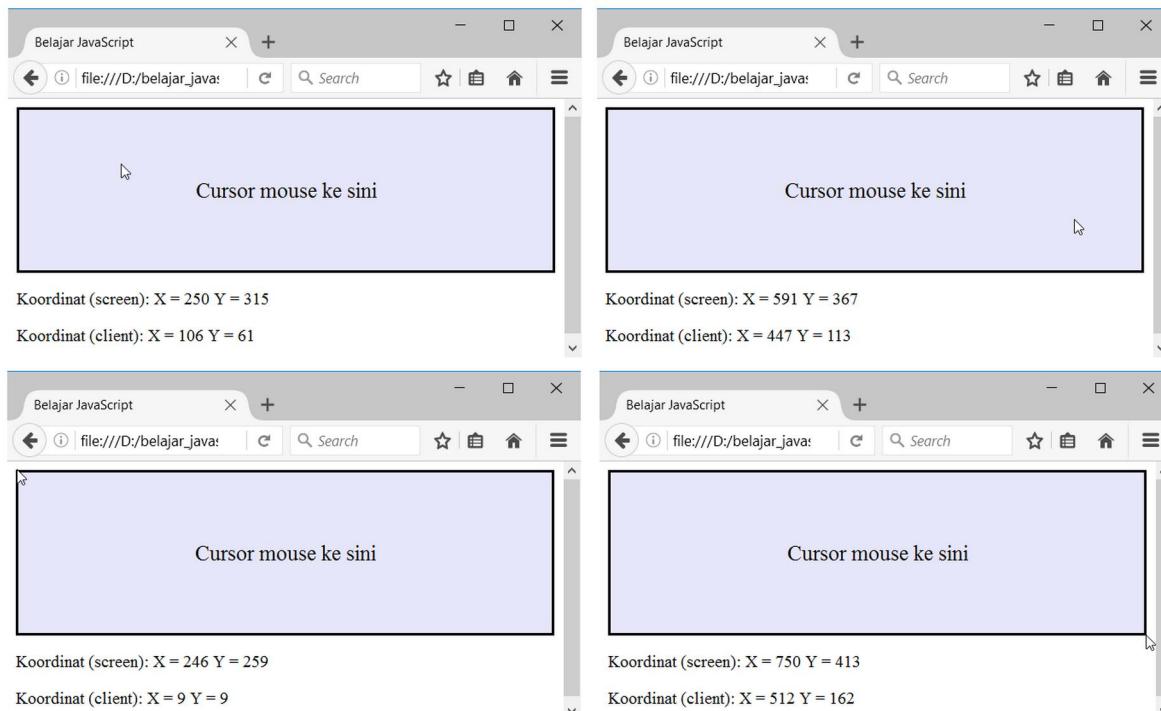
```

Saya membuat sebuah tag `<div>` yang style menggunakan kode CSS agar berbentuk kotak persegi panjang. Inilah tempat untuk event `mousemove`.

Dibawahnya saya membuat 2 buah tag `<p>` yang masing-masing berisi dua buah tag `<span>`. Tag `<span>` ini nantinya akan berisi nilai dari `screenX`, `screenY`, `clientX` dan `clientY` dari **event object**.

Di dalam javascript, saya membuat sebuah function `mousemoveEvent()`. Function ini akan dipanggil setiap terjadi event `mousemove`. Isinya, tempatkan nilai event object `screenX`, `screenY`, `clientX` dan `clientY` ke dalam masing-masing tag `<span>` yang sudah dipersiapkan.

Hasilnya, anda bisa melihat nilai posisi cursor mouse di bagian bawah tag `<div>`:



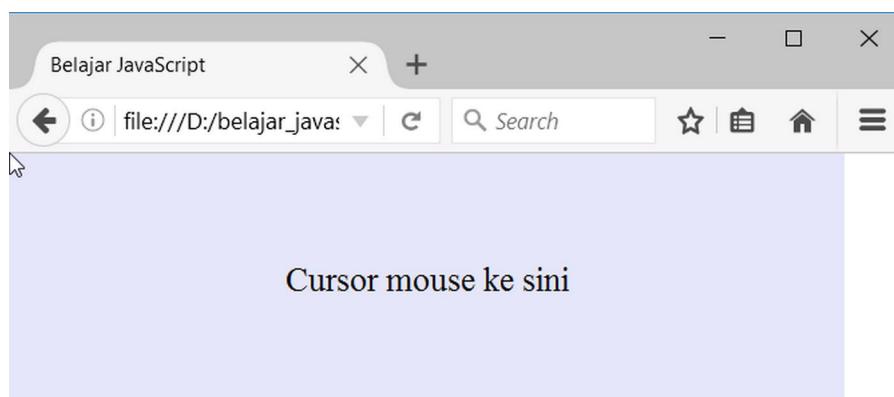
Gambar: Hasil dari penggunaan event `mousemove`

Nilai property `screenX` dan `screenY` adalah posisi cursor relatif kepada jendela layar secara keseluruhan (bukan hanya jendela web browser). Artinya jika anda menggunakan layar dengan

resolusi 1336 x 768, inilah nilai maksimum dari screenX dan screenY. Silahkan perkecil jendela web browser dan geser ke sudut lain dari layar. Nilai screenX dan screenY juga akan berubah (meskipun posisi cursor tetap sama di dalam web browser).

Nilai property clientX dan clientY adalah posisi cursor relatif kepada jendela web browser. Sudut kiri atas web browser adalah koordinat 0,0. Sewaktu saya mencoba mengarahkan cursor mouse ke sudut paling kiri dan paling atas tag <div>, hasil yang di dapat adalah 9,9 bukan 0,0. Ini terjadi karena secara default kotak <div> memiliki margin sebesar 9 pixel (style bawaan web browser). Untuk bisa mendapatkan nilai 0,0, saya harus menghapus margin bawaan web browser dan menghapus property border CSS:

```
1 <style>
2   body { margin:0; }
3   div { width: 500px;
4         height: 150px;
5         line-height: 150px;
6         text-align: center;
7         font-size: 20px;
8         background-color: Lavender;
9     }
10 </style>
```



Koordinat (screen): X = 287 Y = 190

Koordinat (client): X = 0 Y = 1

Gambar: Hasil clientX dan clientY setelah margin dan border dihapus

Ternyata hasil yang di dapat masih 0,1. Bisa jadi ini disebabkan posisi cursor yang kurang pas atau resolusi monitor yang saya gunakan.

---

Dalam bab ini kita telah membahas aspek yang membuat JavaScript menjadi dimanis, yakni **event**. Selain cara penggunaan event, kita juga membahas tentang **mouse event**, yakni event yang dihasilkan dari input mouse.

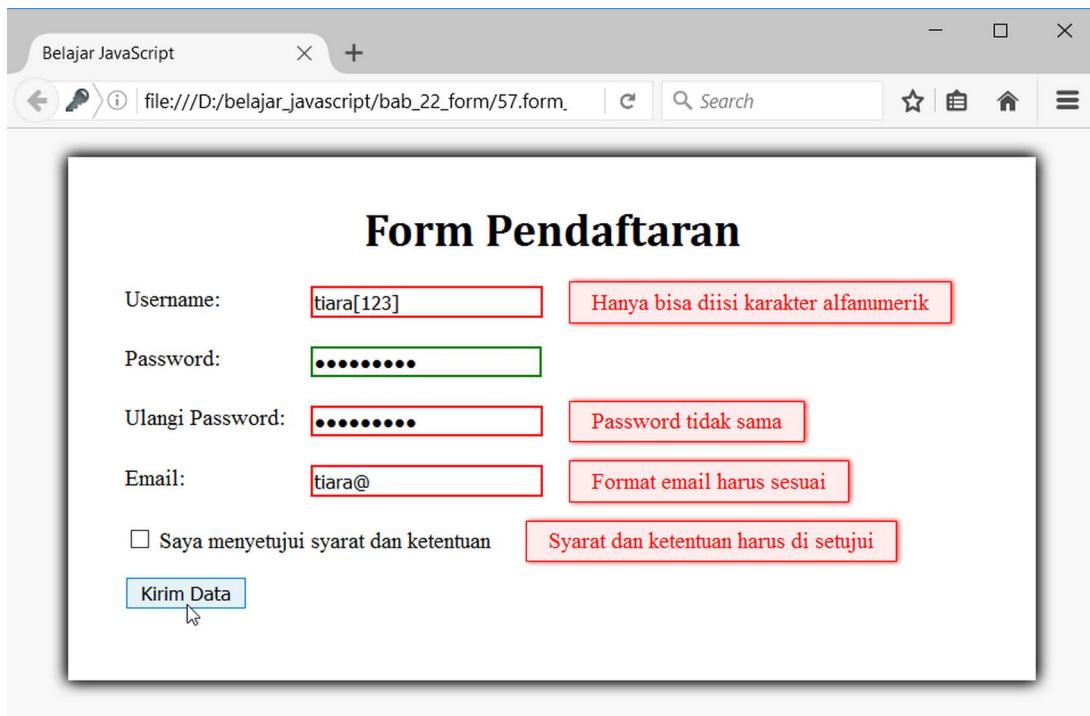
Berikutnya kita akan masuk ke dalam pemrosesan form menggunakan JavaScript.

## 22. Form Processing

Form merupakan salah satu aspek terpenting di dalam sebuah web. Dengan menggunakan form, kita bisa meminta pengunjung menginput data seperti form login, register, pencarian, mengisi survei, mengisi biodata, ujian online, dsb. Semua data ini bisa dikumpulkan menggunakan form. Pembuatan dan pemrosesan form sebenarnya tidak butuh JavaScript. Form dibuat menggunakan HTML dan diproses dengan bahasa pemrograman server seperti PHP.

CSS, MySQL dan JavaScript hanya sebagai “pelengkap” dari form processing. Menggunakan CSS kita bisa mendesign form dengan lebih menarik. MySQL di butuhkan agar data hasil inputan form bisa disimpan secara permanen.

Fungsi JavaScript sendiri di dalam pemrosesan form adalah untuk menambah aspek kenyamanan (*user friendly*) dan membuat form lebih interaktif. Dengan JavaScript kita bisa memeriksa isian form apakah sudah sesuai atau belum. Jika ada yang kurang, tampilkan pesan kesalahan agar user bisa memperbaiki bagian yang salah.



Gambar: Proses validasi menggunakan JavaScript

Tanpa JavaScript, proses validasi ini dilakukan di server menggunakan PHP. Jika ditemukan kesalahan, server akan mengirim kembali form ke web browser untuk diperbaiki. Setelah dinilai ulang, form disubmit kembali. Jika masih ditemukan kesalahan, form akan dikembalikan lagi, demikian seterusnya hingga form dinyatakan valid dan bisa diproses lebih lanjut oleh PHP.

Untuk server yang sibuk dan jaringan yang cukup lambat, proses pengiriman dari web browser ke web server dan sebaliknya bisa memakan waktu yang lama. Bayangkan jika anda mengisi

sebuah form register, kemudian men-klik tombol submit, mengunggu sekitar 2 menit, ternyata tanggal lahir belum diisi. Setelah diisi, klik lagi tombol submit dan menunggu lagi selama 2 menit untuk mengetahui bahwa password harus diisi minimal 6 karakter.

Menggunakan JavaScript, validasi seperti ini bisa dilakukan di web browser sebelum dikirim ke web server. Dengan demikian, form baru benar-benar di submit jika semua data sudah benar dan valid. Inilah yang akan kita belajari dalam bab ini.

**i** Yang juga perlu diingat, validasi di sisi web browser menggunakan JavaScript hanya sebagai proses tambahan. Ini karena JavaScript bisa dimatikan dari web browser. Jika ini terjadi, proses validasi dari JavaScript tidak akan bisa berjalan. Validasi akhir tetap harus di lakukan di web server menggunakan PHP.

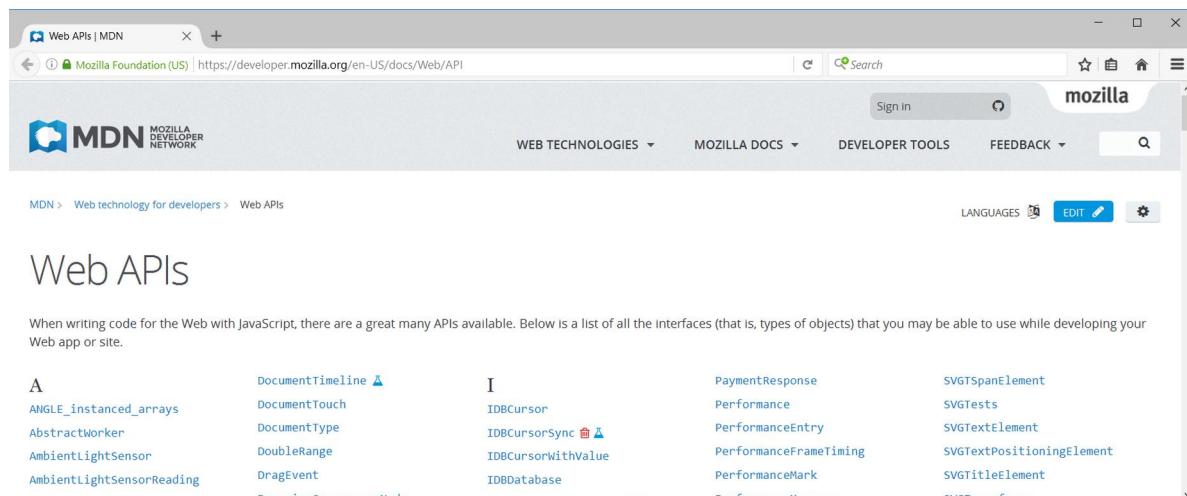
Selain itu, tidak semua proses validasi bisa dilakukan dengan JavaScript. Untuk validasi yang melibatkan database, hanya bisa di proses di server. Misalnya mencari apakah username yang diinput sudah ada yang menggunakan atau tidak. Proses ini tidak bisa ditangani JavaScript, kecuali menggunakan teknologi tambahan seperti AJAX.

## 22.1 Form Element: Node

Sampai disini saya yakin anda sudah familiar dengan “banyaknya” object di dalam struktur DOM HTML. Sebagai contoh, kita sudah mempelajari **Node object**, **Element Node object**, **Text Node object**, **Event object**, **Document object**, serta **Windows object**.

Apakah masih ada yang lain? banyak..!

Dokumentasi lengkap tentang object DOM ini bisa dilihat dari situs Mozilla Developer Network: [Web APIs<sup>1</sup>](#). Jumlahnya? Terdapat lebih dari 500 DOM object.



The screenshot shows a browser window displaying the MDN (Mozilla Developer Network) website. The URL in the address bar is <https://developer.mozilla.org/en-US/docs/Web/API>. The page title is "Web APIs". The content area lists various Web API interfaces, organized into sections for each letter of the alphabet (A, I, P, S). Some examples of listed APIs include DocumentTimeline, IDBCursor, PaymentResponse, SVGTSpanElement, and SVGTests. The MDN logo is visible at the top left, and the Mozilla logo is at the top right. The page has a clean, modern design with a light gray background and white text.

Gambar: Tampilan daftar **Web API** dari Mozilla Developer Network

<sup>1</sup><https://developer.mozilla.org/en-US/docs/Web/API>

Tentunya saya tidak akan membahas semua object ini. Kita hanya fokus kepada object yang paling umum digunakan. Tapi ini sekaligus sebagai gambaran betapa luasnya DOM di dalam HTML. Dengan JavaScript, kita bisa mengakses dan mengolah berbagai object ini.

Untuk pemrosesan form, kita akan berkenalan dengan 5 object baru: **HTMLFormElement**, **HTMLInputElement**, **HTMLTextAreaElement**, **HTMLSelectElement**, dan **HTMLOptionElement**.

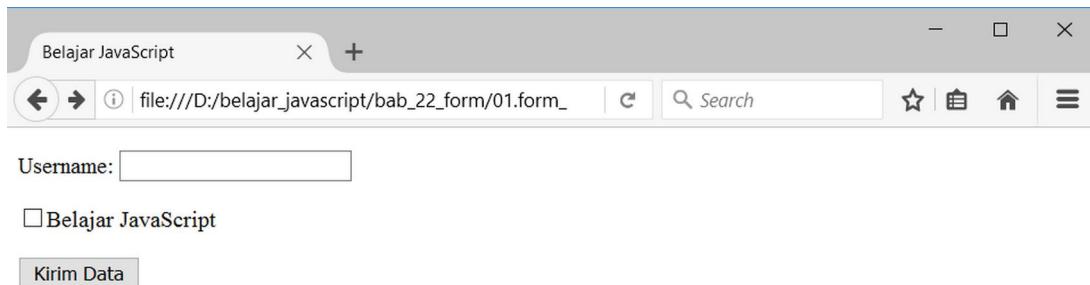
Semua object ini sebenarnya turunan dari **Element object** dengan tambahan method dan property baru. Artinya, setiap object yang ada di dalam form tetap punya property dan method sebagaimana layaknya **Element object** biasa, termasuk berbagai event yang sudah kita pelajari: *click, mouseover, mouseout*, dll.

## Mencari form element

Agar bisa “memprogram” form, hal pertama yang harus kita lakukan adalah mencari form element yang ingin dimanipulasi, atau lebih tepatnya mencari **node object** dari tag `<form>`, tag `<input>`, atau tag `<select>` yang menjadi penyusun form.

Sebagai contoh, perhatikan kode HTML untuk form berikut ini:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <form id="formKu" name="formKu" method="get" action="proses.php">
9     <p>Username: <input type="text" name="username" id="username"></p>
10    <p><input type="checkbox" name="js" id="js"
11      value="Belajar JavaScript">Belajar JavaScript</p>
12    <p><input type="submit" name="submit" id="submit" value="Kirim Data"></p>
13  </form>
14  <script>
15    // Kode JavaScript disini...
16  </script>
17 </body>
18 </html>
```

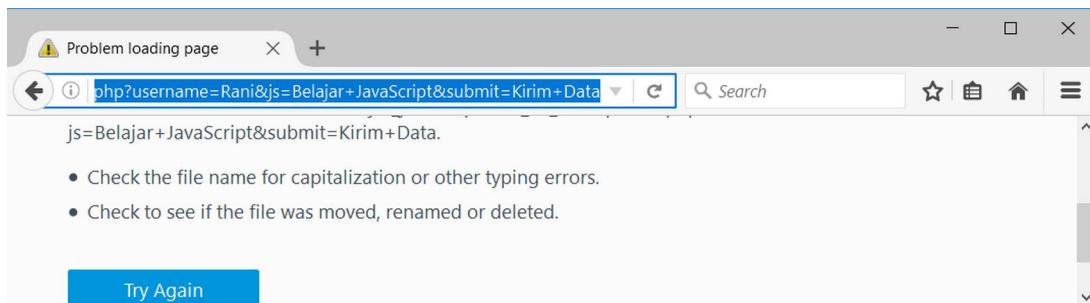


Gambar: Form HTML

Disini saya memiliki kode form HTML dengan 3 buah element, yakni: sebuah text input, sebuah checkbox dan sebuah tombol submit. Setiap element di tempatkan ke dalam tag <p> agar lebih rapi.

Form diatas dilengkapi berbagai atribut seperti name, id, serta type. Untuk tag <form>, terdapat atribut id, name, method dan action. Isi atribut action adalah proses.php, artinya jika tombol "Kirim Data" di-klik, hasil form akan dikirim ke halaman proses.php.

Halaman proses.php tidak akan saya buat karena sudah bagiannya PHP. Kita akan fokus kepada proses sebelum dikirim ke PHP. Jika anda klik tombol "Kirim Data", hasilnya akan error sebab halaman proses.php tidak ditemukan. Tapi karena saya menggunakan method="get", hasil form akan terlihat di bagian address bar web browser (sebagai *query string*):



Gambar: Tampilkan ketika form di submit

Alamat di address bar web browser adalah: file:///D:/belajar\_javascript/bab\_22\_form/proses.php?username=Rani&js=Belajar+JavaScript&submit=Kirim+Data.

Bisa terlihat bahwa data yang dikirim adalah username=Rani, js=Belajar+JavaScript, dan submit=Kirim+Data. Jika anda tertarik mempelajari bagaimana cara memproses dan mengambil nilai-nilai ini, bisa lanjut mempelajari bahasa pemrograman PHP (pemrosesan form saya bahas lengkap di buku **PHP Uncover**).

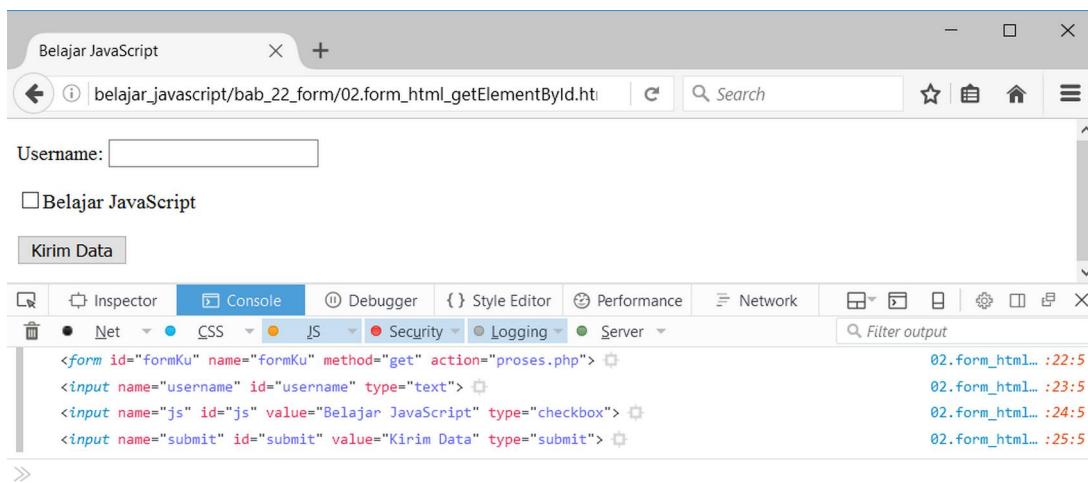
Kembali ke form HTML, bagaimana cara mengambil nilai **Node object** dari tag <form>, tag <input type="text">, tag <input type="checkbox"> dan tag <input type="submit">? Cara yang paling praktis adalah menggunakan method andalan kita: getElementById(). Berikut kode programnya:

```

1 var formNode = document.getElementById("formKu");
2 var usernameNode = document.getElementById("username");
3 var jsNode = document.getElementById("js");
4 var submitNode = document.getElementById("submit");

5
6 console.log(formNode);
7 console.log(usernameNode);
8 console.log(jsNode);
9 console.log(submitNode);

```



Gambar: Mengambil node object dengan method getElementById()

Disini saya membuat 4 buah variabel untuk menampung hasil perintah `getElementById()`, yakni `formNode`, `usernameNode`, `jsNode` dan `submitNode`. Hasil dari perintah `console.log()` memperlihatkan isi dari keempat variabel ini berupa element node.

Cara diatas merupakan metode umum yang sesuai dengan standar DOM W3C. Selain itu, terdapat cara lain. Cara ini berasal dari era sebelum DOM W3C namun tetap didukung oleh mayoritas web browser.

Sebelum era DOM, web browser menyediakan *collection* untuk element HTML yang sering dipakai, seperti *form*, *image* dan *link*. Sebagai contoh, seluruh form yang ada di sebuah halaman disimpan ke dalam collection `document.forms`. Form pertama bisa diakses dari `document.forms[0]`, form kedua di `document.forms[1]`, dst.

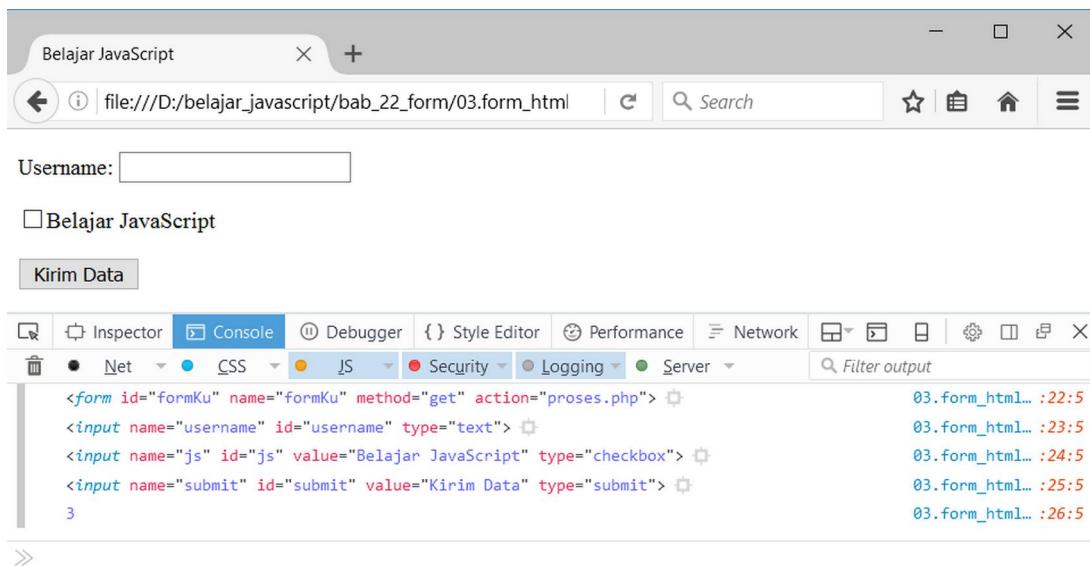
Anggota dari form itu juga disimpan sebagai *sub collection*. Sebagai contoh, anggota pertama dari form bisa diakses dengan perintah `document.forms[0][0]`, anggota kedua form di `document.forms[0][1]`, dst.

Berikut cara untuk mendapatkan **Node object** menggunakan teknik ini:

```

1 var formNode = document.forms[0];
2 var usernameNode = document.forms[0][0];
3 var jsNode = document.forms[0][1];
4 var submitNode = document.forms[0][2];
5
6 console.log(formNode);
7 console.log(usernameNode);
8 console.log(jsNode);
9 console.log(submitNode);
10 console.log(formNode.length);

```



Gambar: Mengambil node object dengan collection document.forms

Hasil yang didapat sama persis seperti method `getElementById()`, dimana setiap variabel akan berisi node object dari setiap anggota form. Di baris terakhir saya menggunakan property `document.forms[0].length` untuk mencari informasi jumlah anggota form. Hasilnya adalah 3, karena memang terdapat 3 element penyusun form, yakni sebuah text input, sebuah checkbox, dan sebuah tombol submit.

Alternatifnya, kita juga bisa memanggil colection ini menggunakan atribut `name`. Jika anda perhatikan, di dalam tag `<form>` saya juga menambahkan atribut `name="formKu"`. Dengan demikian, form ini bisa diakses dari alamat `document.forms.formKu`.

Bagaimana dengan anggota dari form tersebut? Juga bisa diakses menggunakan atribut `name` dari masing-masing element. Berikut caranya:

```
1 var formNode = document.forms.formKu;
2 var usernameNode = document.forms.formKu.username;
3 var jsNode = document.forms.formKu.js;
4 var submitNode = document.forms.formKu.submit;
5
6 console.log(formNode);
7 console.log(usernameNode);
8 console.log(jsNode);
9 console.log(submitNode);
```

Jadi cara mana yang sebaiknya dipakai? Kalau mengikuti standar W3C, cara yang paling disarankan adalah menggunakan method `getElementById()`. Tapi jika anda ingin mengaksesnya dari `forms collection` juga tidak masalah.

## 22.2 Form Element: Property

Setelah berhasil mengambil form element node, mari kita lihat apa saja **property** yang tersedia dari tag `<form>` ini. Form element yang saya maksud adalah tag `<form>` yang digunakan sebagai pembuka form HTML. Untuk isi atau anggota dari form itu, seperti tag `<input>`, `<textarea>` dan `<select>` akan kita bahas secara terpisah.

Di dalam struktur DOM, tag `<form>` berisi element node yang disebut sebagai **HTMLFormElement** object. Selain memiliki property dan method sebagaimana layaknya Element object biasa, **HTMLFormElement** juga memiliki property dan method khusus. Kita akan mulai dari property terlebih dahulu.



Daftar lengkap property dan method **HTMLFormElement** bisa dilihat di [HTMLFormElement Reference<sup>2</sup>](#).

Langsung saja kita lihat cara pengaksesan property dari **HTMLFormElement** object:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <form id="formKu" name="formKu" method="get" action="proses.php"
9     target="_self" enctype="multipart/form-data"
10    accept-charset="utf-8">
11   <p>Username: <input type="text" name="username" id="username"></p>
12   <p><input type="checkbox" name="js" id="js">
```

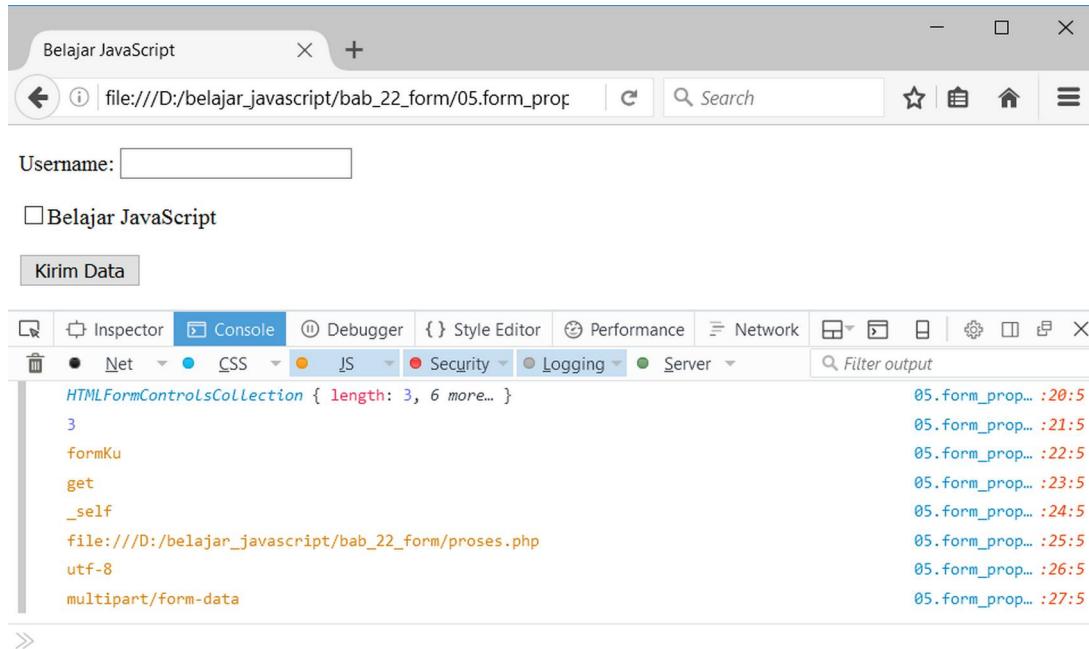
<sup>2</sup><https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement>

```

13     value="Belajar JavaScript">Belajar JavaScript</p>
14     <p><input type="submit" name="submit" id="submit" value="Kirim Data"></p>
15   </form>
16   <script>
17     var formNode = document.getElementById("formKu");
18     console.log (formNode.elements);
19     // HTMLFormControlsCollection { length: 3, 6 more... }
20     console.log (formNode.length);      // 3
21     console.log (formNode.name);       // formKu
22     console.log (formNode.method);     // get
23     console.log (formNode.target);     // _self
24     console.log (formNode.action);
25     // file:///D:/belajar_javascript/bab_22_form/proses.php
26     console.log (formNode.acceptCharset); // utf-8
27     console.log (formNode.encoding);    // multipart/form-data
28   </script>
29 </body>
30 </html>

```

Khusus untuk tag <form>, saya isi dengan berbagai atribut tambahan karena mayoritas property dari *HTMLFormElement* object digunakan untuk mengambil nilai-nilai ini. Hasilnya adalah sebagai berikut:



Gambar: Berbagai property dari tag <form>

Berikut penjelasan dari nilai property diatas:

- **HTMLFormElement.elements**: Berisi *collection* dari anggota / element penyusun form.
- **HTMLFormElement.length**: Berisi jumlah anggota / element yang menyusun form.

- **HTMLFormElement.name**: Berisi nilai atribut name dari tag <form>.
- **HTMLFormElement.method**: Berisi nilai atribut method dari tag <form>.
- **HTMLFormElement.target**: Berisi nilai atribut target dari tag <form>.
- **HTMLFormElement.action**: Berisi nilai atribut action dari tag <form>.
- **HTMLFormElement.acceptCharset**: Berisi nilai atribut acceptCharset dari tag <form>.
- **HTMLFormElement.encoding**: Berisi nilai atribut encoding dari tag <form>.

Sebagian besar property dari **HTMLFormElement** digunakan untuk mengakses nilai atribut dari tag <form>.

Selain property `elements` dan `length`, property diatas juga bisa digunakan untuk mengubah nilai atribut, misalnya:

```
1 var formNode = document.getElementById("formKu");
2 formNode.action = "login.php";
```

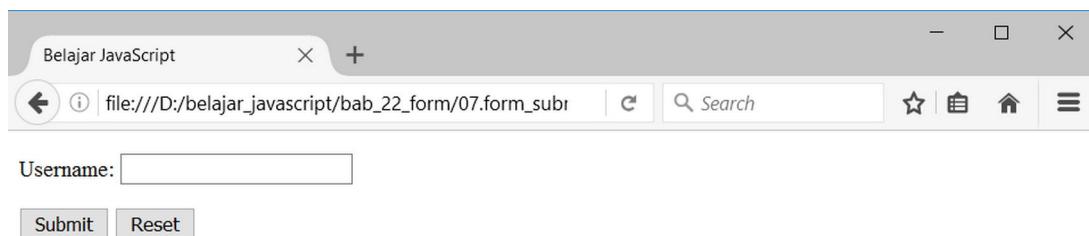
Sekarang jika tombol submit form di klik, akan terkirim ke halaman "login.php", bukan lagi "proses.php".

## 22.3 Form Element: Method

Selain memiliki property, **HTMLFormElement** object memiliki 2 method yang cukup penting, yakni `submit()` dan `reset()`. Sesuai namanya, kedua method ini digunakan untuk men-submit form (mengirim form ke server untuk di proses) dan me-reset form (mengosongkan seluruh kotak isian form).

Secara bawaan, proses `submit` dan `reset` form akan berjalan dari tombol `submit` dan tombol `reset`. Kedua tombol ini dibuat menggunakan tag <input type="submit"> dan <input type="reset">, seperti contoh berikut:

```
1 <body>
2   <form id="formKu" name="formKu" method="get" action="proses.php">
3     <p>Username: <input type="text" name="username" id="username"></p>
4     <p><input type="submit" name="submit" id="submit" value="Submit">
5     <input type="reset" name="reset" id="reset" value="Reset"></p>
6   </form>
7 </body>
```

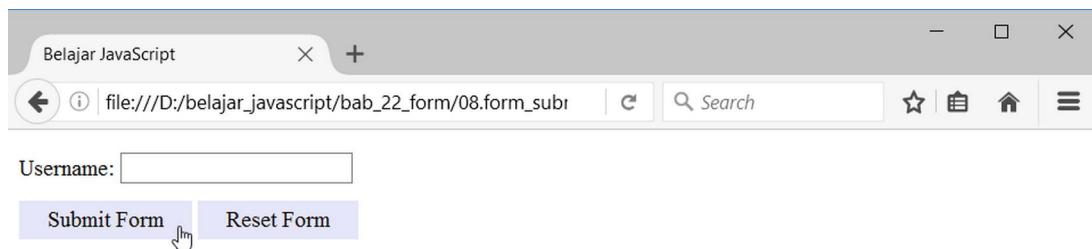


Gambar: Tombol submit dan reset form

Dengan memanggil method `submit()` dan `reset()` kepunyaan `HTMLFormElement` object, kita bisa membuat cara lain untuk men-submit dan me-reset form, tanpa harus menggunakan tombol submit dan tombol reset bawaan.

Berikut contoh penggunannya:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <style>
7     span {background-color: Lavender; padding: 5px 20px; cursor: pointer}
8   </style>
9 </head>
10 <body>
11   <form id="formKu" name="formKu" method="get" action="proses.php">
12     <p>Username: <input type="text" name="username" id="username"></p>
13   </form>
14   <p><span id="spanSubmit">Submit Form</span>
15   <span id="spanReset">Reset Form</span></p>
16   <script>
17     var formNode = document.getElementById("formKu");
18     var spanSubmitNode = document.getElementById("spanSubmit");
19     var spanResetNode = document.getElementById("spanReset");
20
21     function diSubmit(){ formNode.submit(); }
22     function diReset(){ formNode.reset(); }
23
24     spanSubmitNode.addEventListener("click", diSubmit);
25     spanResetNode.addEventListener("click", diReset);
26   </script>
27 </body>
28 </html>
```



Gambar: Submit dan Reset form di luar form

Disini saya membuat form yang sama seperti sebelumnya, hanya berisi 1 buah teks input yang dibuat dari `<input type="text">`, tapi tanpa tombol submit dan reset.

Untuk menggantikan tombol submit dan reset, di luar form saya membuat 2 buah tag <span>. Masing-masing tag ini berisi teks "Submit Form" dan "Reset Form". Keduanya di *style* dengan sedikit kode CSS agar tampak seperti tombol.

Perhatikan bahwa kedua tag <span> ini berada di luar form, artinya tidak terhubung dengan form. Sekarang, bagaimana caranya agar ketika tombol <span>"Submit Form"</span> di klik, form langsung terkirim? Caranya adalah dengan memanggil method `formNode.submit()` ketika <span> tersebut di klik.

Pertama, saya harus mencari Node element untuk 3 element HTML: <form id="formKu">, <span id="spanSubmit">, dan <span id="spanReset">. Dengan menggunakan method `document.getElementById()`, ketiga node element ini disimpan ke dalam variabel `formNode`, `spanSubmitNode`, dan `spanResetNode`.

Selanjutnya saya membuat event **click** untuk kedua span, ketika `spanSubmitNode` di klik, jalankan fungsi `diSubmit()` yang isinya memanggil method `formNode.submit()`. Hasilnya, ketika tag <span> di klik, `formKu` akan di kirim ke server, sama seperti ketika tombol submit biasa.

Begitu juga ketika `spanResetNode` di klik, jalankan fungsi `diReset()` yang isinya akan menjalankan method `formNode.reset()`. Efeknya akan sama seperti men-klik tombol reset, dan isian form akan terhapus.

Inilah hasil dari pemanggilan method `submit()` dan `reset()` dari `HTMLFormElement` object.

## 22.4 Form Element: Event

Selain memiliki property dan method, form object juga memiliki 2 buah event, yakni **submit event** dan **reset event**. Kedua event ini dipanggil saat form di submit dan saat form di reset.

Penggunaan paling banyak dari kedua event ini adalah untuk "menahan" form sebelum dikirim ke server atau sebelum di reset. Berikut contoh penggunaannya:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="utf-8">
5  <title> Belajar JavaScript </title>
6  </head>
7  <body>
8  <form id="formKu" name="formKu" method="get" action="proses.php">
9    <p>Username: <input type="text" name="username" id="username"></p>
10   <p><input type="submit" name="submit" id="submit" value="Submit Form">
11   <input type="reset" name="reset" id="reset" value="Reset Form"></p>
12 </form>
13 <script>
14   function diStop(e){
15     console.log(e.type);
```

```
16     e.preventDefault();
17     console.log(e.defaultPrevented);
18 }
19 var formNode = document.getElementById("formKu");
20 formNode.addEventListener("submit",diStop);
21 formNode.addEventListener("reset",diStop);
22 </script>
23 </body>
24 </html>
```

Disini saya membuat sebuah form dengan 1 tag `<input type="text">`, sebuah tombol submit, dan sebuah tombol reset. Secara bawaan, kedua tombol ini akan men-submit form dan me-reset form saat di klik.

Di dalam kode JavaScript, saya membuat 2 buah *event listener* yang dipanggil dari object `formNode`:

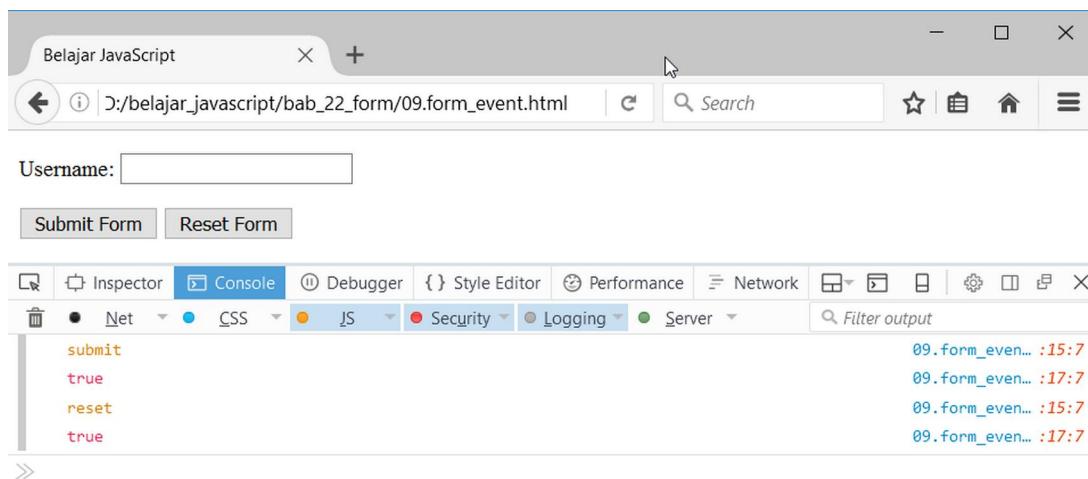
```
1 formNode.addEventListener("submit",diStop);
2 formNode.addEventListener("reset",diStop);
```

Artinya, ketika form di submit, jalankan function `diStop()`. Begitu juga ketika form di reset, jalankan fungsi `diStop()`. Apa isi dari function `diStop()` ini?

```
1 function diStop(e){
2   console.log(e.type);
3   e.preventDefault();
4   console.log(e.defaultPrevented);
5 }
```

Sebenarnya fungsi `diStop()` hanya terdiri dari 1 perintah saja: `e.preventDefault()`, 2 perintah lainnya hanya untuk menampilkan informasi mengenai tipe event dan apakah event default sudah dihentikan.

Sebagaimana yang telah kita pelajari, method `preventDefault()` akan menghentikan event bawaan dari sebuah object. Efeknya, form akan ditahan untuk tidak ter-submit dan tidak ter-reset. Berikut hasil yang di dapat saat tombol submit dan reset di klik:



Gambar: Hasil fungsi diStop() saat tombol submit dan reset di klik

Jika kita ingin memeriksa isian form untuk proses validasi, method `preventDefault()` seperti ini perlu dipanggil. Setelah isian form sudah valid dan sesuai, kita bisa memanggil method `submit()` agar form segera di kirim ke server.

Sampai disini saya juga ingin memastikan pemahaman anda tentang perbedaan *method submit* dari form object, dengan *event submit* dari form object. Apa maksud keduanya?

**Method submit** digunakan untuk mengirim form ke server, sedangkan **event submit** adalah event yang dipanggil saat form di submit. Jika masih kurang menegerti perbedaan dari kedua istilah ini, boleh dibaca kembali penjelasan sebelumnya. Kalau perlu jalankan langsung kode program yang ada agar lebih di pahami.

## 22.5 Input Element: Property

Sebagian besar isi form dibuat menggunakan 1 tag HTML saja, yakni `<input>`. Nilai dari atribut `type` lah yang akan membedakan bentuk dan jenis element yang dihasilkan.

Untuk membuat inputan teks box, kita menggunakan `<input type="text">`, untuk checkbox menggunakan `<input type="checkbox">`, untuk radio button menggunakan tag `<input type="radio">`, dan masih banyak bentuk form lain, termasuk yang dari HTML5 seperti `<input type="email">` atau `<input type="date">`.

Di dalam DOM, tag `<input>` akan menghasilkan sebuah element yang dikenal sebagai **HTMLInputElement**.

**HTMLInputElement** memiliki berbagai property yang bisa diakses. Tergantung jenis atribut `type`, nilai property ini juga bisa bertambah. Kita akan mulai dari tag `input type text` terlebih dahulu, yakni kotak isian berupa text box.

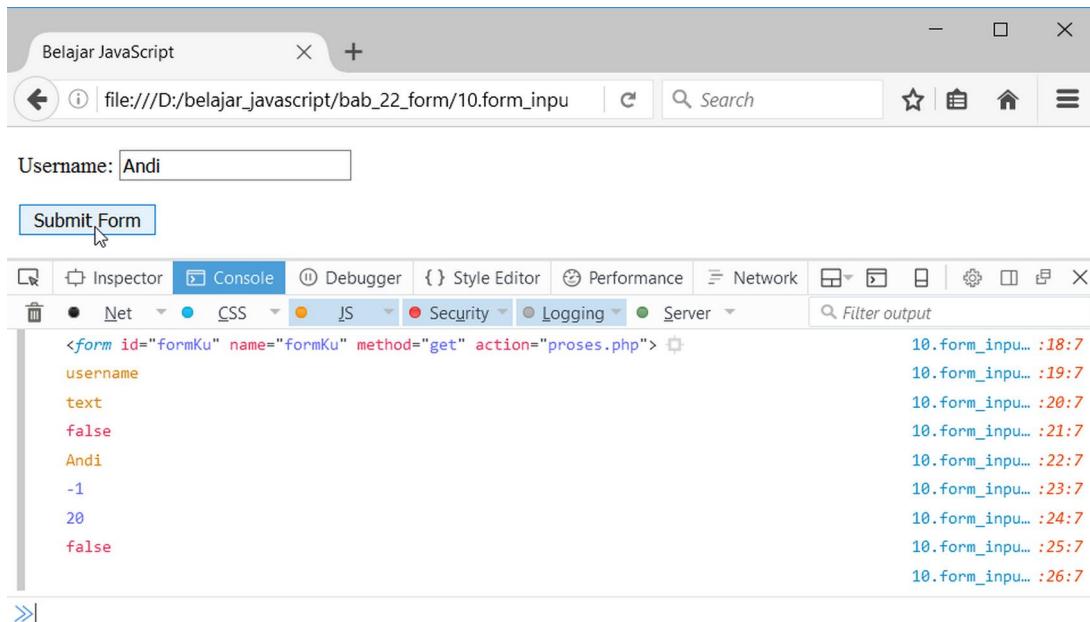


Daftar lengkap property dan method **HTMLInputElement** bisa dilihat di [HTMLInputElement Reference<sup>3</sup>](#).

Berikut contoh pengaksesan nilai property dari **HTMLInputElement** untuk tag `input type text`:

<sup>3</sup><https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement>

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <form id="formKu" name="formKu" method="get" action="proses.php">
9     <p>Username: <input type="text" name="username" id="username"></p>
10    <p><input type="submit" name="submit" id="submit" value="Submit Form"></p>
11  </form>
12 <script>
13   var formNode = document.getElementById("formKu");
14   var usernameNode = document.getElementById("username");
15
16   function diProses(e){
17     e.preventDefault();
18     console.log(usernameNode.form);
19     // <form id="formKu" name="formKu" method="get" action="proses.php">
20     console.log(usernameNode.name);           // username
21     console.log(usernameNode.type);          // text
22     console.log(usernameNode.autofocus);       // false
23     console.log(usernameNode.value);          // Andi
24     console.log(usernameNode.maxLength);       // -1
25     console.log(usernameNode.size);           // 20
26     console.log(usernameNode.readOnly);        // false
27     console.log(usernameNode.placeholder);      // (kosong)
28   }
29   formNode.addEventListener("submit", diProses);
30 </script>
31 </body>
32 </html>
```



Gambar: Tampilan berbagai property HTMLInputElement object

Untuk kode HTML, saya kembali membuat form dengan 1 tag `<input type="text">` serta 1 tombol submit.

Di baris awal kode JavaScript, variabel `formNode` dan `usernameNode` digunakan untuk menampung Node Object dari tag `<form id="formKu">` dan `<input type="text" id="username">`.

Selanjutnya terdapat function `diProses()` yang akan dipanggil saat form di submit. Ini karena saya menempatkan sebuah *event listener* untuk `submit` di baris terakhir, yakni baris kode program: `formNode.addEventListener("submit", diProses)`. Artinya, ketika form di submit, jalankan function `diProses()`.

Function `diProses()` diawali dengan method `e.preventDefault()`. Method ini berfungsi untuk menahan form agar tidak dikirim ke server. Setelah itu terdapat berbagai perintah `console.log()` untuk menampilkan isi dari property `usernameNode`, yakni element node yang bertipe `HTMLInputElement`.

Berikut penjelasan dari berbagai property yang diakses:

- `HTMLInputElement.form`: Berisi element node dari form tempat dimana tag `<input>` berada.
- `HTMLInputElement.name`: Berisi nilai atribut `name` dari tag `<input>`.
- `HTMLInputElement.type`: Berisi nilai atribut `type` dari tag `<input>`.
- `HTMLInputElement.autofocus`: Berisi nilai boolean apakah atribut `autofocus` aktif atau tidak.
- `HTMLInputElement.value`: Berisi nilai atribut `value` dari tag `<input>`. Property ini juga berisi nilai teks yang saat ini ada di dalam tag `<input>`,
- `HTMLInputElement.maxLength`: Berisi nilai atribut `maxLength` dari tag `<input>`, akan menghasilkan nilai `-1` jika atribut `maxLength` tidak ditulis.
- `HTMLInputElement.size`: Berisi nilai atribut `size` dari tag `<input>`. Secara default, panjang dari tag `<input>` adalah `20`.

- **HTMLInputElement.readOnly**: Berisi nilai boolean apakah atribut readOnly aktif atau tidak.
- **HTMLInputElement.placeholder**: Berisi nilai atribut placeholder dari tag <input>.

Sama seperti property **HTMLFormElement**, property dari **HTMLInputElement** sebagian besar juga digunakan untuk mengakses atribut yang ada.

Dari semua property ini, yang akan paling sering kita akses adalah property **value**, karena property inilah yang berisi nilai teks dari tag <input>. Silahkan anda input teks sembarang, lalu tekan tombol "Submit Form". Property **value** akan berisi nilai teks tersebut.

Sebagai contoh berikutnya, bagaimana jika nilai dari property **value** ini kita kirim ke object lain (tidak hanya ke `console.log`)?

Tidak masalah, berikut kode programnya:

```

1 <body>
2   <form id="formKu" name="formKu" method="get" action="proses.php">
3     <p>Username: <input type="text" name="username" id="username"></p>
4     <p><input type="submit" name="submit" id="submit" value="Submit Form"></p>
5   </form>
6   <p>Hasil: <span id="hasil"></span></p>
7   <script>
8     var formNode = document.getElementById("formKu");
9     var usernameNode = document.getElementById("username");
10    var hasilNode = document.getElementById("hasil");
11
12    function diProses(e){
13      e.preventDefault();
14      hasilNode.innerHTML = usernameNode.value;
15    }
16
17    formNode.addEventListener("submit",diProses);
18  </script>
19 </body>
```

Dalam kode program ini, setelah penulisan form HTML saya menambahkan dua buah tag baru, yakni:

```
<p>Hasil: <span id="hasil"></span></p>
```

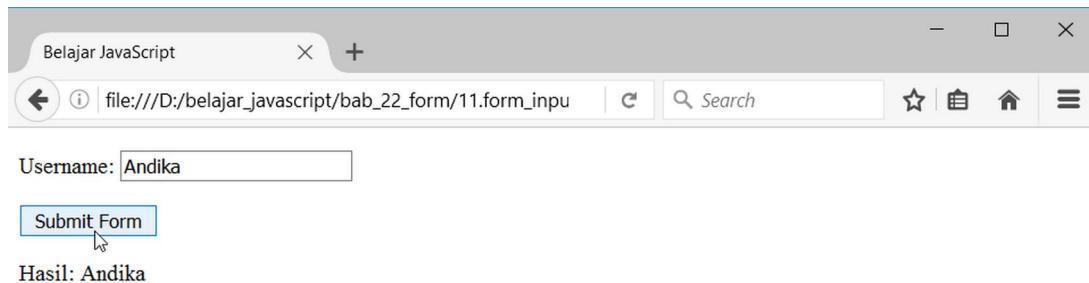
Di dalam tag <p>, terdapat tag <span id="hasil">. Ke dalam tag <span> inilah nilai property **value** dari tag <input> akan ditampilkan.

Pada awal kode program JavaScript, saya membuat 3 buah variabel: `formNode`, `usernameNode` dan `hasilNode`. Semuanya berisi node object dari tag <form>, tag <input>, dan tag <span>.

Di akhir kode program saya juga menambahkan *event listener* untuk event **submit** seperti sebelumnya. Artinya, ketika form di submit jalankan function `diProses()`.

Isi dari function `diProses()` hanya 2 baris, pertama perintah `e.preventDefault()` untuk menghentikan form dikirim ke server, dan perintah `hasilNode.innerHTML = usernameNode.value` yang akan mengambil nilai property `value` dari `usernameNode`, kemudian diinput ke dalam `hasilNode.innerHTML`.

Silahkan anda input teks sembarang ke dalam text box, lalu klik tombol "Submit Form". Nilai yang ada di form akan tampil ke dalam tag `<span id="hasil">`:



Gambar: Menampilkan nilai tag `<input>` ke dalam tag `<span>`

Mari kita ubah sedikit contoh praktik ini. Bagaimana dengan men-copy hasil text box ke dalam text box lain?

Caranya tidak jauh berbeda. Cukup memindahkan nilai property `value` dari element pertama ke property `value` elemen kedua, seperti contoh berikut:

```
1 <body>
2   <form id="formKu" name="formKu" method="get" action="proses.php">
3     <p>Username: <input type="text" name="username" id="username"></p>
4     <p><input type="submit" name="submit" id="submit" value="Copy Data"></p>
5   </form>
6   <p>Hasil: <input type="text" id="hasil"></p>
7 <script>
8   var formNode = document.getElementById("formKu");
9   var usernameNode = document.getElementById("username");
10  var hasilNode = document.getElementById("hasil");
11
12  function diProses(e){
13    e.preventDefault();
14    hasilNode.value = usernameNode.value;
15    hasilNode.disabled = "true";
16  }
17  formNode.addEventListener("submit",diProses);
18 </script>
19 </body>
```

Disini saya hanya mengubah tag `<span id="hasil"></span>` menjadi `<input type="text" id="hasil">`. Sedangkan untuk function `diProses()`, saya menjalankan 3 perintah:

```

1 function diProses(e){
2   e.preventDefault();
3   hasilNode.value = usernameNode.value;
4   hasilNode.disabled = "true";
5 }
```

Perintah pertama, `e.preventDefault()` digunakan untuk menghentikan form di kirim ke server. Perintah kedua `hasilNode.value = usernameNode.value` digunakan untuk mengambil nilai `value` dari `usernameNode` ke `hasilNode`. Perintah ketiga `hasilNode.disabled = "true"` sekedar tambahan yang akan membuat tag `<input type="text" id="hasil">` menjadi *disabled* (berwarna abu-abu dan tidak bisa diisi).

Berikut tampilannya:



Gambar: Men-copy nilai value dari tag `<input>` ke tag `<input>` lain

## Form yang tidak harus di dalam “form”

Jika anda perhatikan contoh sebelum ini, terlihat bahwa tag `<input type="text" id="hasil">` berada di luar tag `<form>` namun property-nya tetap bisa diakses. Artinya, sebuah tag tidak harus berada di dalam `<form>` agar bisa diproses dengan JavaScript. Ini juga berlaku untuk anggota form lainnya, seperti `<textarea>` maupun `<select>`.

Berikut percobaannya:

```

1 <body>
2   <p>Username: <input type="text" id="username"></p>
3   <p><button id="copyData">Copy Data</button></p>
4   <p>Hasil: <input type="text" id="hasil"></p>
5   <script>
6     var copyDataNode = document.getElementById("copyData");
7     var usernameNode = document.getElementById("username");
8     var hasilNode = document.getElementById("hasil");
9
10    function diProses(){
11      hasilNode.value = usernameNode.value;
12      hasilNode.disabled = "true";
13    }

```

```
14     copyDataNode.addEventListener("click",diProses);  
15 </script>  
16 </body>
```

Dalam kode HTML diatas, saya menghapus tag `<form>`. Kemudian mengganti tombol submit form dengan tag `<button>`. Event submit form juga diganti dengan event click tag `<button>`.

Hasilnya? Sama persis seperti sebelumnya. Ketika tombol "Copy Data" di klik, ambil nilai `usernameNode.value` dan copy nilainya ke `hasilNode.value`.



Gambar: Memproses isian form tanpa form

Cara seperti ini hanya cocok untuk form yang tidak akan dikirim ke server dan hanya diproses menggunakan JavaScript saja. Jika nilai form perlu dikirim ke web server, kita harus menempatkan isian form ini ke dalam tag `<form>`.

Karena relatif lebih singkat dan tidak perlu memanggil `e.preventDefault()` setiap kali form di submit, cara seperti ini akan banyak saya pakai dalam pembahasan selanjutnya.

## 22.6 Input Element: Event

Input element atau lengkapnya **HTMLInputElement** alias tag `<input>` memiliki berbagai event yang bisa diakses. Karena **HTMLInputElement** adalah turunan dari element object, maka event standar seperti `click`, `dblclick`, `mouseover`, dll juga berlaku kepada element ini.

Selain event tersebut, terdapat 3 event khusus yang sering digunakan untuk element form, termasuk tag `<input>`, yakni **focus**, **blur** dan **change**. Tidak hanya tag `<input>` saja, element lain seperti `<textarea>` dan `<select>` juga mendukung ketiga event ini.

### Event Focus

Event **focus** terjadi saat sebuah element di pilih. Caranya bisa dengan menggunakan mouse (element tersebut di klik) atau menggunakan keyboard dengan menekan tombol tab.

Berikut contoh penggunaannya:

```

1 <body>
2   <p>Username: <input type="text" id="username"></p>
3   <p>Email: <input type="text" id="email"></p>
4   <script>
5     var usernameNode = document.getElementById("username");
6     var emailNode = document.getElementById("email");
7
8     function diFocus(e){
9       e.target.style.border = "2px solid aqua";
10    }
11
12    usernameNode.addEventListener("focus",diFocus);
13    emailNode.addEventListener("focus",diFocus);
14  </script>
15 </body>

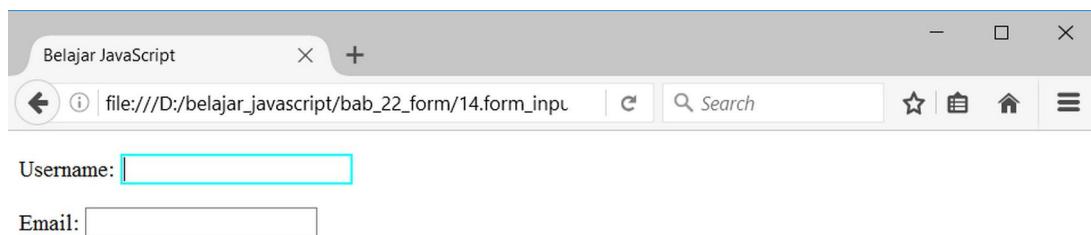
```

Untuk kode HTML, saya membuat dua buah tag `<input>`, masing-masing dengan id `username` dan `email`.

Di dalam JavaScript, variabel `usernameNode` dan `emailNode` berisi node element dari kedua tag `<input>`, yang diikuti dengan pendefenisian function `diFocus()`.

Terakhir terdapat 2 buah event listener: `focus`, masing-masing untuk `usernameNode` dan `emailNode`. Saat kedua element ini di-**focus**, jalankan function `diFocus()`. Isi dari function `diFocus()` sendiri hanya 1 baris: `e.target.style.border = "2px solid aqua"`. Artinya, ubah style CSS `border` untuk element tersebut menjadi warna *aqua* sebesar 2 pixel.

Silahkan anda jalankan kode diatas lalu klik box inputan, otomatis bordernya menjadi warna biru muda (*aqua*). Kemudian tekan tombol tab di keyboard untuk pindah ke box email, bordernya juga akan berubah warna. Inilah yang dimaksud dengan event `focus`.



Gambar: Event focus mengubah warna border text box username

## Event Blur

Event **blur** adalah kebalikan dari **focus**. Event ini terjadi ketika sebuah element kehilangan fokus, yakni saat kita pindah ke element lain.

Untuk melihat efek dari event **blur**, saya akan sambung contoh sebelumnya. Sekarang, ketika text box kehilangan focus, ubah warna border menjadi merah (red). Berikut kode programnya:

```

1 <body>
2   <p>Username: <input type="text" id="username"></p>
3   <p>Email: <input type="text" id="email"></p>
4   <script>
5     var usernameNode = document.getElementById("username");
6     var emailNode = document.getElementById("email");
7
8     function diFocus(e){
9       e.target.style.border = "2px solid aqua";
10    }
11
12    function diBlur(e){
13      e.target.style.border = "2px solid red";
14    }
15
16    usernameNode.addEventListener("focus",diFocus);
17    emailNode.addEventListener("focus",diFocus);
18    usernameNode.addEventListener("blur",diBlur);
19    emailNode.addEventListener("blur",diBlur);
20  </script>
21 </body>

```

Dalam kode program diatas, saya menambah event **blur** untuk `usernameNode` dan `emailNode`. Ketika kedua element ini di-blur, jalankan function `diBlur()`. Apa isinya? Mengubah warna border menjadi merah dengan perintah: `e.target.style.border = "2px solid red"`.

Silahkan anda jalankan kode program diatas, kemudian klik atau tab dari satu text box ke text box lainnya, warna border akan berubah-ubah dari merah ke biru, dan sebaliknya.

Saat sebuah text box dipilih, akan aktif event **focus**, ketika kita pindah ke element lain, akan aktif event **blur**.



Gambar: Event focus dan blur mengubah warna border text box username dan email

## Event Change

Event **change** akan aktif ketika nilai sebuah element berubah, baik jika nilainya ditambah, dikurangi atau dihapus. Khusus untuk text box, event ini akan aktif pada saat element tersebut kehilangan focusnya.

Mari kita lihat menggunakan contoh:

```

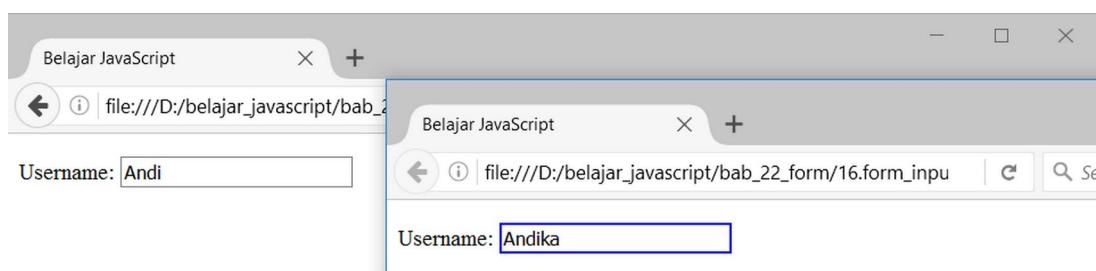
1 <body>
2   <p>Username: <input type="text" id="username" value="Andi"></p>
3   <script>
4     var usernameNode = document.getElementById("username");
5
6     function diChange(e){
7       e.target.style.border = "2px solid blue";
8     }
9
10    usernameNode.addEventListener("change",diChange);
11  </script>
12 </body>

```

Kali ini saya menghapus text box email dan hanya menyisakan text box username. Di dalam atributnya terdapat value="Andi". Nilai string "Andi" ini akan menjadi nilai awal dari text box username.

Di dalam kode JavaScript, saya membuat *event listener* untuk event **change**. Ketika terjadi event **change**, jalankan function **diChange()** yang isinya akan mengubah warna border menjadi biru: `e.target.style.border = "2px solid blue"`.

Silahkan jalankan kode program diatas, kemudian coba berbagai kemungkinan, misalnya tambah atau kurangi huruf yang ada di dalam text box dan lihat apa yang terjadi.



Gambar: Event change akan aktif saat nilai text box diubah

Anda akan melihat bahwa event **change** baru akan aktif ketika text box kehilangan focus. Saat kita mengubah nilai teks dengan menambah beberapa huruf, tidak akan ada perubahan apa-apa. Tapi begitu keluar dari text box, misalnya dengan men-klik bagian lain dari halaman web (seperti background putih di belakang), barulah warna border berubah.

Event **change** juga tidak akan aktif jika isinya belum diubah atau sama seperti nilai awal. Ketika di jalankan, text box akan berisi string "Andi". Jika saya menghapus teks ini, kemudian mengetik kembali string "Andi" dan bepindah ke element lain, event **change** tidak akan terpanggil. Karena nilainya memang tidak berubah.

Tapi jika di tambah 1 huruf saja misalnya menjadi "Andik" kemudian pindah ke element lain, event **change** akan aktif.

Sebagai latihan tambahan, silahkan anda gabung ketiga event yang kita pelajari ini, yakni **focus**, **blur** dan **change** ke dalam satu kode program. Anda juga bisa berkreasi dengan mengubah **style** lain, misalnya jika terjadi event **focus**, warna teks menjadi abu-abu, atau ketika terjadi event **blur**, ubah warna background tag `<body>` menjadi pink.

## 22.7 Menjalankan Event Element Lain

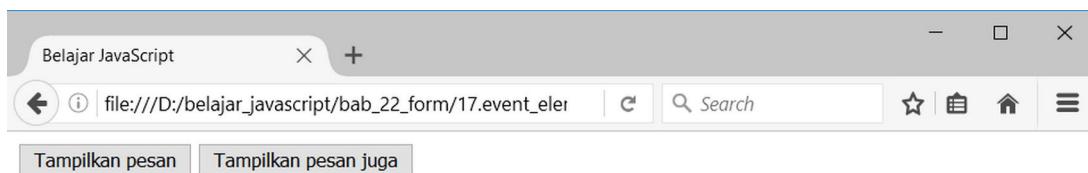
Dalam pembahasan mengenai **event**, kita masih menjalankan event untuk element itu sendiri. JavaScript juga membolehkan menjalankan event milik element lain, yakni dengan cara memanggil **method** dari element tersebut.

Misalnya ketika sebuah tombol diklik, tombol lain juga bisa mengalami event `click`. Atau saat sebuah tombol diklik, bisa jadi element lain akan mengalami event `focus`.

Penjelasan ini lebih mudah dijelaskan menggunakan contoh kode program:

```

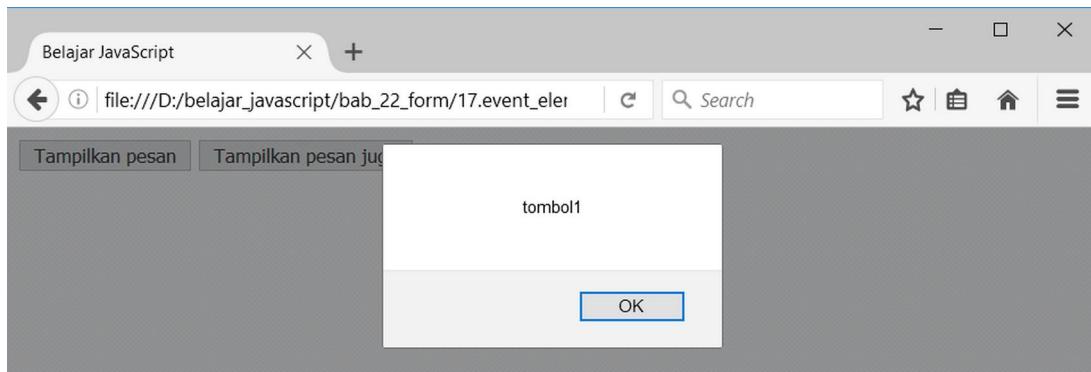
1 <body>
2   <button id="tombol1">Tampilkan pesan</button>
3   <button id="tombol2">Tampilkan pesan juga</button>
4   <script>
5     var tombol1Node = document.getElementById("tombol1");
6     var tombol2Node = document.getElementById("tombol2");
7
8     function tampilkanPesan(e){
9       alert(e.target.id);
10    }
11    tombol1Node.addEventListener("click",tampilkanPesan);
12
13    function tampilkanPesanjuga(){
14      tombol1Node.click();
15    }
16    tombol2Node.addEventListener("click",tampilkanPesanjuga);
17  </script>
18 </body>
```



Gambar: Dua buah tombol yang dibuat dari tag `<button>`

Saya membuat 2 buah tombol dengan tag `<button id="tombol1">`, dan `<button id="tombol2">`. Di dalam JavaScript keduanya disimpan ke dalam variabel `tombol1Node` dan `tombol2Node`.

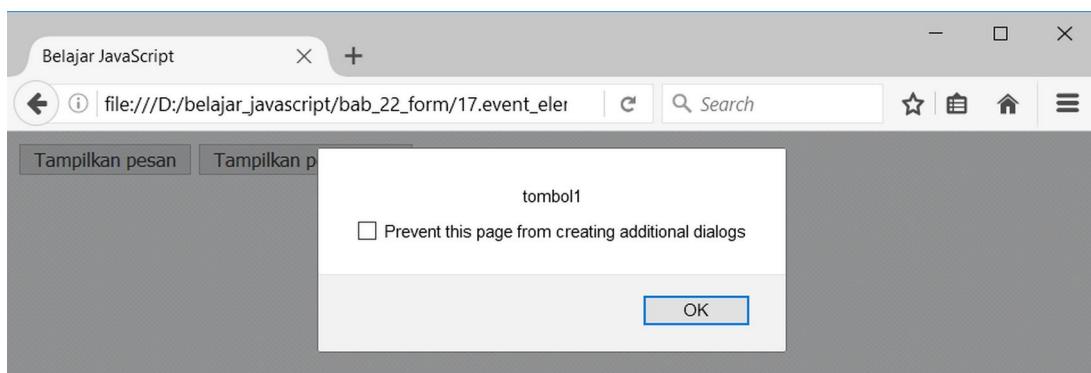
Saat `tombol1Node` diklik, jalankan function `tampilkanPesan()`. Isinya berupa perintah `alert(e.target.id)`. Artinya, ketika tombol "Tampilkan pesan" diklik, akan tampil jendela alert: `tombol1`.



Gambar: Tampilan pesan alert saat tombol "Tampilkan pesan" di klik

Teks tombol1 tampil karena `e.target.id` akan mengakses atribut `id` dari element dimana event dijalankan. Karena `tombol1Node` memiliki atribut `id="tombol1"`, maka hasilnya adalah teks `tombol1`.

Sekarang, perhatikan isi function `tampilkanPesanjuga()`. Isinya adalah `tombol1Node.click()`. Ini artinya, jalankan event `click` kepunningan `tombol1Node`. Mari kita coba:



Gambar: Hasilnya tetap tombol1, bukan tombol2

Hasilnya tetap `tombol1`, bukan `tombol2`. Karena walaupun saya men-klik `<button id="tombol2">`, kode yang dijalankan adalah `tombol1Node.click()`, jadi saya memanggil event `click` kepunningan dari `<button id="tombol1">`. Inilah yang dimaksud dengan *memanggil event milik element lain*.

Event yang bisa dipanggil bukan hanya `click` saja, tapi juga bisa event lain seperti **focus** dan **blur**. Berikut contoh penggunaannya:

```

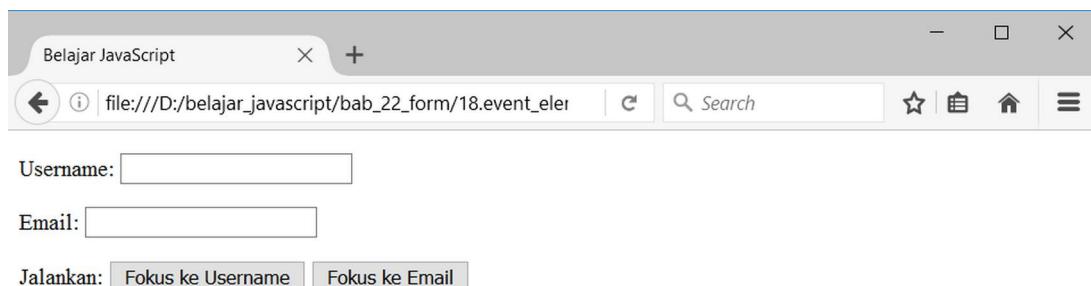
1 <body>
2   <p>Username: <input type="text" id="username"></p>
3   <p>Email: <input type="text" id="email"></p>
4   <p>Jalankan: <button id="focusU">Fokus ke Username</button>
5   <button id="focusE">Fokus ke Email</button></p>
6   <script>
7     var usernameNode = document.getElementById("username");
8     var emailNode = document.getElementById("email");
9

```

```

10  function diFocus(e){
11      e.target.style.border = "2px solid aqua";
12  }
13
14  function diBlur(e){
15      e.target.style.border = "2px solid red";
16  }
17
18  usernameNode.addEventListener("focus",diFocus);
19  emailNode.addEventListener("focus",diFocus);
20  usernameNode.addEventListener("blur",diBlur);
21  emailNode.addEventListener("blur",diBlur);
22
23  var focusUNode = document.getElementById("focusU");
24  var focusENode = document.getElementById("focusE");
25
26  focusUNode.addEventListener("click",function(){ usernameNode.focus(); });
27  focusENode.addEventListener("click",function(){ emailNode.focus(); });
28 </script>
29 </body>

```

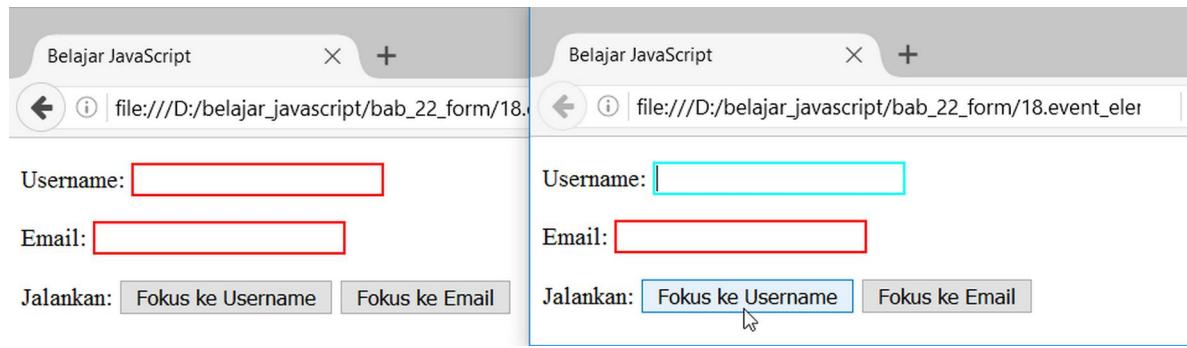


Gambar: Tambahan tombol untuk focus ke text box Username dan Email

Kode diatas terlihat cukup panjang, namun ini merupakan modifikasi dari contoh yang saya pakai saat membahas event *blur* dan *focus*. Tambahannya, sekarang ada 2 buah tombol: <button id="focusU"> dan <button id="focusE">.

Saat tombol "Fokus ke Username" di-klik, saya ingin tag <input id="username"> langsung ter-focus. Bagaimana caranya? Cukup memanggil perintah `usernameNode.focus()` saat terjadi event *click* dari `focusUNode`.

Begitu juga saat tombol "Fokus ke Email" di-klik, akan dijalankan perintah `emailNode.focus()`, yang membuat tag <input id="username"> mengalami event *focus*.



Gambar: Focus kepada text box Username saat tombol Fokus ke Username di klik

## 22.8 Keyboard Event

Di akhir bab sebelumnya, kita telah membahas 7 mouse event, yakni **click**, **mouseover**, **mouseout**, dll. Saya sengaja belum membahas tentang **keyboard** event karena melibatkan tag `<input>` form sebagai penampung teks yang akan diketik.

### Event Keydown, Keypress, dan Keyup

Keyboard memiliki 3 event: **keydown**, **keypress** dan **keyup**. Karena mirip satu sama lain, saya akan membahas ketiganya sekaligus.

Event **keydown** dan **keypress** terjadi saat tombol keyboard ditekan. Kedua event ini nyaris tidak berbeda, namun event **keydown** akan dipanggil terlebih dahulu, baru kemudian diikuti dengan event **keypress**. Situasi ini mirip seperti event **mousedown** dengan **click**.

Sedangkan untuk event **keyup** baru dipanggil saat tombol keyboard dilepaskan. Berikut contoh penggunaan ketiga event ini:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <style>
7     #hasil {width:300px; height: 50px; border: 2px solid black; }
8   </style>
9 </head>
10 <body>
11   <p>Ketik disini: <input type="text" id="ketik"></p>
12   <div id="hasil"></div>
13   <script>
14     var ketikNode = document.getElementById("ketik");
15     var hasilNode = document.getElementById("hasil");
16

```

```

17  function diKeydown(){
18      hasilNode.style.backgroundColor = "red";
19  }
20  function diKeypress(){
21      hasilNode.style.backgroundColor = "blue";
22  }
23  function diKeyup(){
24      hasilNode.style.backgroundColor = "yellow";
25  }
26
27  ketikNode.addEventListener("keydown",diKeydown);
28  ketikNode.addEventListener("keypress",diKeypress);
29  ketikNode.addEventListener("keyup",diKeyup);
30 </script>
31 </body>
32 </html>

```



Gambar: Efek dari berbagai keyboard event

Saya membuat sebuah tag input type text yang memiliki id="ketik" dan sebuah tag <div> yang memiliki id="hasil". Untuk tag <div> di-style menggunakan kode CSS agar tampak lebih menarik.

Di dalam JavaScript, tag <input> dan <div> disimpan ke dalam variabel ketikNode dan hasilNode. Untuk ketikNode, saya menambahkan 3 *event listener*: **keydown**, **keypress** dan **keyup**. Dimana dalam setiap event, jalankan function yang berfungsi untuk mengubah warna background dari tag <div>.

Jika terjadi event **keydown**, jalankan perintah `hasilNode.style.backgroundColor = "red";`. Hasilnya, warna background dari tag <div> berubah menjadi merah. Jika terjadi event **keypress**, ubah warna background menjadi biru. Dan jika terjadi event **keyup**, ubah warna background menjadi kuning.

Silahkan anda jalankan dan coba ketik sesuatu di dalam text box. Saat tombol keyboard di tekan, warna tag <div> akan berubah menjadi biru (event **keypress**), dan saat tombol key biar di lepaskan, warna tag <div> akan berubah menjadi kuning (event **keyup**).

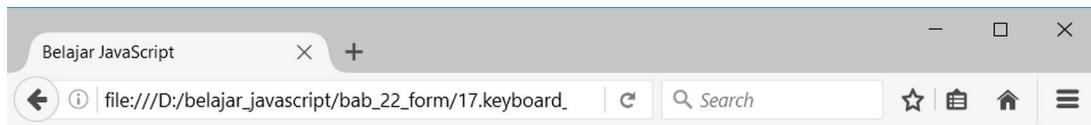
Anda akan dapati bahwa warna tag <div> tidak pernah menjadi merah, seolah-olah event **keydown** tidak pernah terjadi. Event **keydown** sebenarnya tetap dipanggil, hanya saja langsung digantikan dengan event **keypress**. Akibatnya, warna merah dari **keydown** langsung tertimpa dengan warna biru dari **keypress**.

Pengecualiannya adalah ketika tombol fungsi ditekan, seperti tombol Capslock, SHIFT, ALT dan CTRL, semuanya akan mengubah warna tag <div> menjadi merah. Artinya event **keydown** dijalankan tapi tidak diikuti oleh event **keypress**. Inilah salah satu pembeda antara event **keydown** dengan **keypress**.

Selain itu, keyboard event juga akan dipanggil saat tombol apapun ditekan (walaupun tidak ada teks yang diketik) seperti tombol panah atas, bawah, kiri, dan kanan yang ada di keyboard.

Berikut contoh lain dari penggunaan keyboard event:

```
1 <body>
2   <p>Ketik disini: <input type="text" id="ketik"></p>
3   <p>Hasil keydown: <span id="hasilKeydown"></span></p>
4   <p>Hasil keypress: <span id="hasilKeyPress"></span></p>
5   <p>Hasil keyup: <span id="hasilKeyUp"></span></p>
6   <script>
7     var ketikNode = document.getElementById("ketik");
8     var hasilKeydownNode = document.getElementById("hasilKeydown");
9     var hasilKeyPressNode = document.getElementById("hasilKeyPress");
10    var hasilKeyUpNode = document.getElementById("hasilKeyUp");
11
12    function diKeydown(){
13      hasilKeydownNode.innerHTML = ketikNode.value;
14    }
15    function diKeypress(){
16      hasilKeyPressNode.innerHTML = ketikNode.value;
17    }
18    function diKeyup(){
19      hasilKeyUpNode.innerHTML = ketikNode.value;
20    }
21
22    ketikNode.addEventListener("keydown",diKeydown);
23    ketikNode.addEventListener("keypress",diKeypress);
24    ketikNode.addEventListener("keyup",diKeyup);
25  </script>
26 </body>
```



Ketik disini:

Hasil keydown:

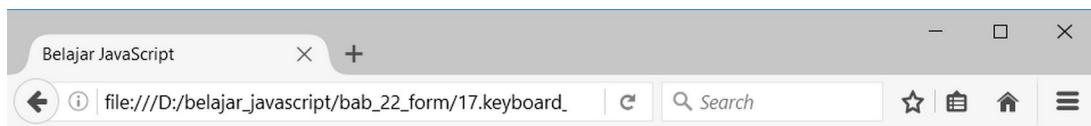
Hasil keypress:

Hasil keyup:

Gambar: Mencopy teks saat event keyboard terjadi

Kali ini saya ingin mencopy isi dari `<input type="text" id="ketik">` ke dalam 3 buah tag `<p><span>`. Untuk setiap event keyboard, copy nilai `ketikNode.value` ke setiap tag `<span>`.

Pada saat saya mengetik 1 huruf, yang akan terisi hanyalah Hasil **keyup**, sedangkan "Hasil keydown" dan "Hasil keypress" tetap kosong:



Ketik disini:

Hasil keydown:

Hasil keypress:

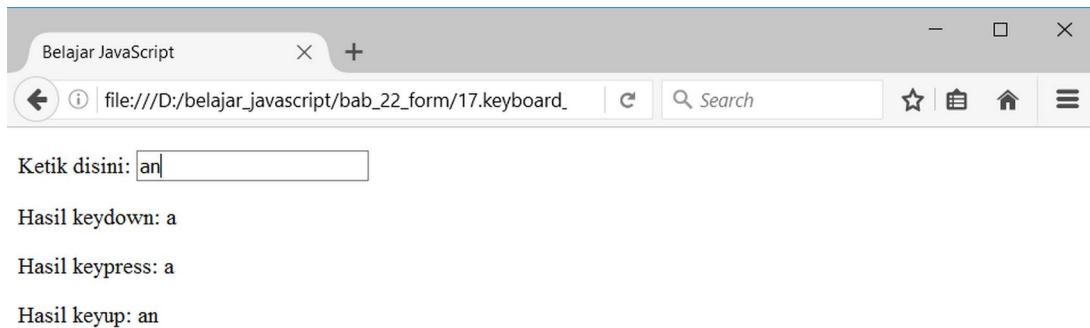
Hasil keyup: a

Gambar: Hanya "hasil keyup" yang berisi teks

Kenapa ini terjadi? Bukankah seharusnya event **keydown** dan **keypress** yang dipanggil terlebih dahulu?

Betul, dan karena alasan itulah isi dari "Hasil keydown" dan "Hasil keypress" tetap kosong. Pada saat event **keydown** dan **keypress** dipanggil, isi dari tag `<input>` masih kosong. huruf "a" baru akan tampil sesaat setelah tombol keyboard dilepaskan. Akibatnya, yang dicopy adalah teks kosong ini.

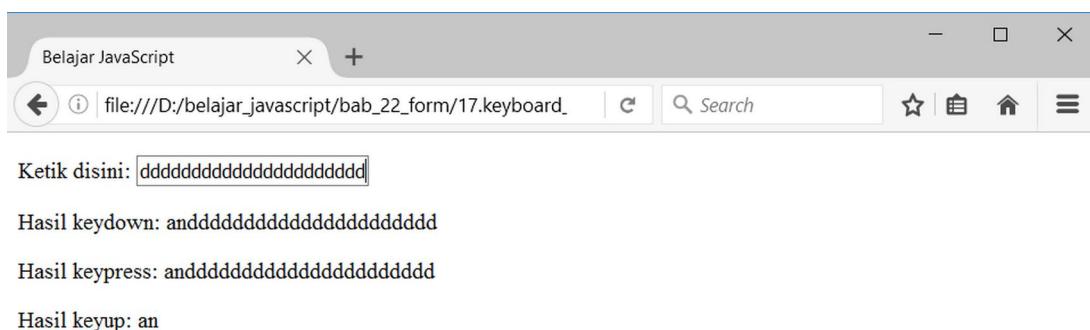
Jika saya mengetik huruf kedua, barulah huruf "a" akan tampil di "Hasil keydown" dan "Hasil keypress", sebelum event **keyup** berjalan:



Gambar: "Hasil keydown" dan "Hasil keypress" sudah berisi teks

Silahkan anda ketik teks lain dan semuanya akan tercopy kedalam 3 tag <span>. Hanya saja, untuk "Hasil keydown" dan "Hasil keypress" akan terlihat 'telat' sebanyak 1 huruf, disebabkan alasan seperti penjelasan diatas.

Jika anda tahan 1 tombol keyboard cukup lama dan tidak dilepaskan, huruf yang sama akan muncul beberapa kali. Tapi karena kita tidak melepaskan tombol keyboard tersebut, event **keyup** tidak akan dipanggil:



Gambar: Event keydown dan keypress dipanggil berulang kali

Yang terjadi adalah, event **keydown** dan **keypress** dipanggil secara berulang terus menerus, sedangkan event **keyup** baru dipanggil saat tombol keyboard dilepaskan.

## Keyboard event object

Ketiga keyboard event yang kita bahas akan menghasilkan sebuah event object khusus yang bernama **KeyboardEvent**.

**KeyboardEvent** object juga memiliki berbagai property yang bisa diakses, yang berisi apa tombol yang saat ini sedang diketik, apakah tombol CRTL di tahan atau tidak, dll. Daftar lengkap dari property dan method dari **KeyboardEvent** object bisa dilihat di [KeyboardEvent Reference<sup>4</sup>](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent).

Mari kita lihat beberapa property **KeyboardEvent** object:

<sup>4</sup><https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent>

```

1 <body>
2   <p>Ketik disini: <input type="text" id="ketik"></p>
3   <script>
4     var ketikNode = document.getElementById("ketik");
5
6     function diProses(e){
7       console.log(e.key);           // a
8       console.log(e.charCode);     // 97
9       console.log(e.keyCode);     // 0
10      console.log(e.which);       // 97
11      console.log(e.altKey);      // false
12      console.log(e.ctrlKey);     // false
13      console.log(e.shiftKey);    // false
14    }
15
16    ketikNode.addEventListener("keypress", diProses);
17  </script>
18 </body>

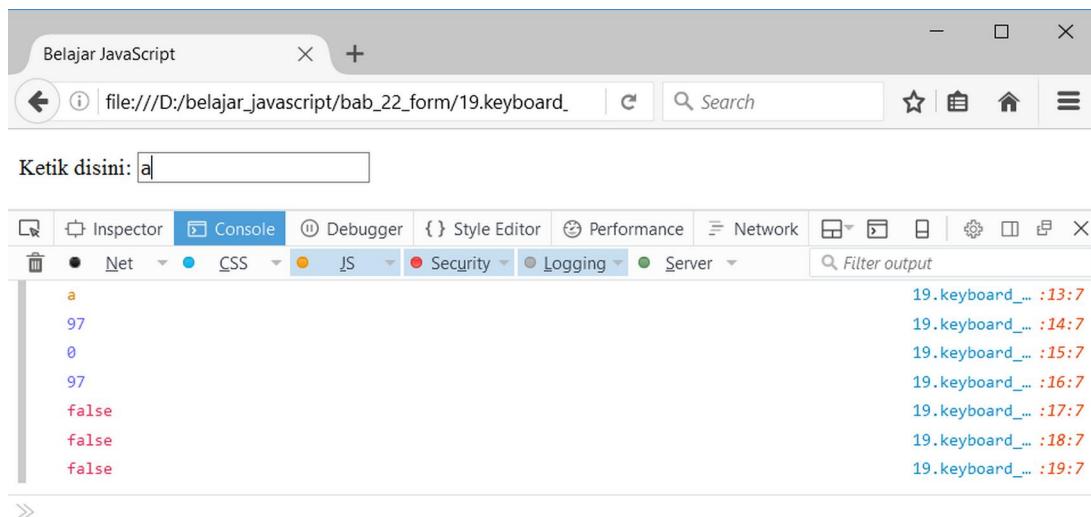
```

Disini saya memiliki 1 buah tag `<input>` dengan `id="ketik"`. Di dalam JavaScript, tag `<input>` ini disimpan ke dalam variabel `ketikNode`.

Variabel `ketikNode` saya berikan event handler untuk `keypress`. Untuk setiap event `keypress`, jalankan fungsi `diProses()`.

Isi dari fungsi `diProses()` berupa berbagai perintah `console.log()` untuk mengakses property dari `KeyboardEvent` object. Artinya, untuk setiap karakter yang diketik, akan menampilkan seluruh informasi ini.

Berikut tampilannya:



Gambar: Berbagai property dari `KeyboardEvent` object

Silahkan anda ketik karakter lain, hasilnya juga akan berbeda. Bisa dicoba juga dengan kombinasi tombol seperti SHIFT, CTRL, atau ALT.

Berikut penjelasan dari property-property tersebut:

- **KeyboardEvent.key**: Berisi string dari tombol yang ditekan, seperti "A", "c", "5", juga tombol seperti "Enter", "Tab" atau "ArrowUp".
- **KeyboardEvent.charCodeAt**: Berisi kode Unicode dari tombol yang ditekan, contohnya "A" akan menjadi 65, karena kode Unicode untuk karakter "A" adalah 65. Property ini hanya akan berisi untuk event **keypress**. `charCode` sebaiknya tidak dipakai lagi (*deprecated*), disarankan menggunakan `KeyboardEvent.key`.
- **KeyboardEvent.keyCode**: Sama seperti `charCode`, khusus untuk event **keydown** dan **keyup**. Di beberapa sistem, huruf 'a' kecil akan menghasilkan nilai unicode untuk huruf "A" besar. `keyCode` juga sebaiknya tidak dipakai lagi (*deprecated*), disarankan menggunakan `KeyboardEvent.key`.
- **KeyboardEvent.which**: Sama seperti `keyCode` dan `charCode`, berlaku untuk event **keydown**, **keyup** maupun **keypress**. Property `which` sebaiknya tidak dipakai lagi (*deprecated*), disarankan menggunakan `KeyboardEvent.key`.
- **KeyboardEvent.altKey**: Berisi nilai boolean apakah tombol ALT di tahan saat terjadi event
- **KeyboardEvent.ctrlKey**: Berisi nilai boolean apakah tombol CRTL di tahan saat terjadi event
- **KeyboardEvent.shiftKey**: Berisi nilai boolean apakah tombol SHIFT di tahan saat terjadi event

Dari daftar property ini, terdapat 4 property yang bisa digunakan untuk "mengangkap" isi karakter yang saat ini sedang diketik, yakni: `key`, `keyCode`, `charCode` dan `which`. Banyaknya property ini menunjukkan kompleksitas pembacaan karakter keyboard di dalam JavaScript.

Setiap web browser juga ada yang hanya menggunakan `keyCode` saja atau `charCode` saja. Selain itu kode Unicode yang dihasilkan juga bisa berbeda-beda untuk tombol sistem seperti SHIFT, ALT dan CTRL. Daftar perbedaan ini bisa anda baca-baca di: [JavaScript Madness: Keyboard Events<sup>5</sup>](#).

Untungnya, saat ini mayoritas web browser modern sudah mengikuti standar W3C. Kita disarankan hanya menggunakan property `KeyboardEvent.key` untuk mengetahui tombol apa yang sedang diinput. Daftar lengkap nama dari tombol-tombol ini bisa dilihat di [Key Values References<sup>6</sup>](#).

Sebagai tambahan, jika anda ingin membuat kode program yang bisa membaca tombol khusus seperti SHIFT, ALT dan CTRL secara terpisah (tidak di kombinasikan dengan tombol lain), hanya akan terbaca untuk event **keydown** dan **keyup** tapi tidak untuk event **keypress**.

Untuk pembuktianya, silahkan anda ubah event **keypress** contoh kode kita sebelumnya, dari:

```
ketikNode.addEventListener("keypress", diProses);
```

Menjadi event **keydown**:

---

<sup>5</sup> <http://unixpapa.com/js/key.html>

<sup>6</sup> [https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/key/Key\\_Values](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/key/Key_Values)

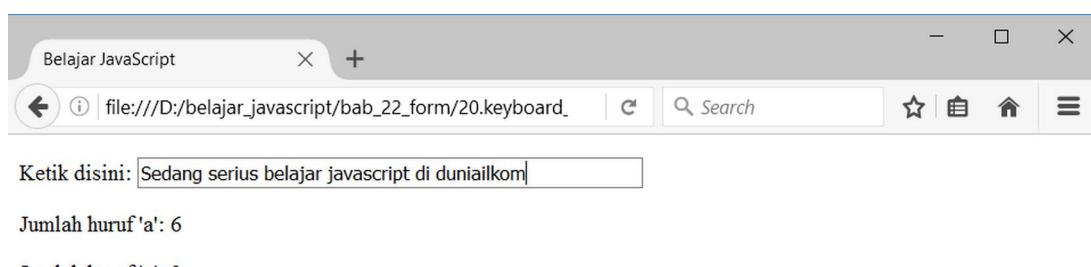
```
ketikNode.addEventListener("keydown", diProses);
```

Kemudian lihat perbedaan kedua event ini.

Latihan terakhir untuk keyboard event, saya ingin membuat program untuk menghitung banyak huruf "a" dan huruf "e" yang diketik dari sebuah tag <input>. Berikut kode programnya:

```

1  <body>
2    <p>Ketik disini: <input type="text" id="ketik" size="50"></p>
3    <p>Jumlah huruf 'a': <span id="hasilA"></span></p>
4    <p>Jumlah huruf 'e': <span id="hasilE"></span></p>
5    <script>
6      var ketikNode = document.getElementById("ketik");
7      var hasilNodeA = document.getElementById("hasilA");
8      var hasilNodeE = document.getElementById("hasilE");
9
10     var jumlahA = 0;
11     var jumlahE = 0;
12
13     function diProses(e){
14       if (e.key == "a"){
15         jumlahA++;
16         hasilNodeA.innerHTML = jumlahA;
17       }
18       if (e.key == "e"){
19         jumlahE++;
20         hasilNodeE.innerHTML = jumlahE;
21       }
22     }
23
24     ketikNode.addEventListener("keypress", diProses);
25   </script>
26 </body>
```



Gambar: Menghitung berapa banyak huruf 'a' dan 'e' di ketik

Disini saya membuat sebuah tag <input type="text"> dan 2 buah tag <p><span> yang berfungsi sebagai penampung hasil. Tag <span> memiliki atribut id="hasilA" dan id="hasilE" yang disimpan ke dalam hasilNodeA dan hasilNodeE.

Di dalam kode JavaScript, saya juga menyiapkan 2 buah variabel: jumlahA dan jumlahE. Kedua variabel ini akan menjadi *variabel counter* untuk menghitung jumlah huruf 'a' dan huruf 'e'.

Untuk tag <input> disimpan ke dalam ketikNode dan saya tambahkan sebuah event keypress. Saat event keypress terjadi, jalankan function diProses().

Isi dari function diProses() terdapat 2 buah kondisi if else. Jika e.key == "a", naikkan nilai variabel jumlahA sebanyak 1 angka, kemudian tampilkan ke hasilNodeA.innerHTML. Jika e.key == "e", naikkan nilai variabel jumlahE sebanyak 1 angka, kemudian tampilkan ke hasilNodeE.innerHTML.

Hasilnya, ketika sebuah huruf diketik di dalam tag <input>, function diProses() akan selalu dijalankan, jika yang diketik huruf 'a' atau huruf 'e', nilainya akan dihitung dan tampil di dalam tag <span>.

## 22.9 Input Element: Type Password

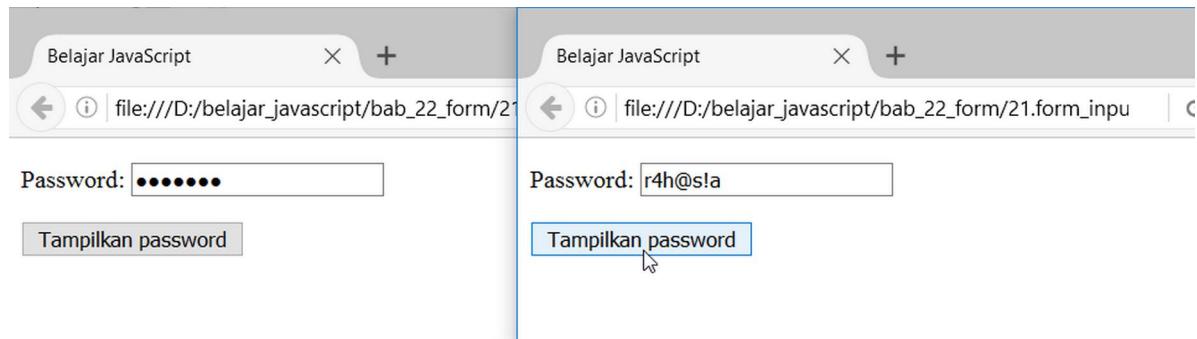
Input element type password adalah sebutan bagi tag <input type="password">. Element ini digunakan untuk membuat kotak inputan password yang pada saat di ketik, teks akan disamarkan dengan tanda bulatan hitam atau tanda bintang (tergantung web browser).

Secara umum, tag <input type="password"> tidak banyak berbeda dari tag <input type="text">. Seluruh atribut, property, dan event yang telah kita pelajari untuk tag <input type="text"> juga berlaku untuk tag <input type="password">. Meskipun teks yang diinput tersamarkan, hasil dari property value tetap berisi teks asli. Property ini bisa diambil untuk diproses lebih lanjut.

Sebagai latihan, saya ingin membuat sebuah kode program untuk input element type password:

```
1 <body>
2   <p>Password: <input type="password" id="password"></p>
3   <p><button id="tombol">Tampilkan password</button></p>
4   <script>
5     var passwordNode = document.getElementById("password");
6     var tombolPassNode = document.getElementById("tombol");
7
8     function proses(){
9       passwordNode.type = "text";
10    }
11
12    tombolPassNode.addEventListener("click",proses);
13  </script>
14 </body>
```

Bisakah anda menebak seperti apa "maskud" kode diatas? Semuanya sudah kita pelajari sepanjang bab ini. Berikut tampilan akhirnya:

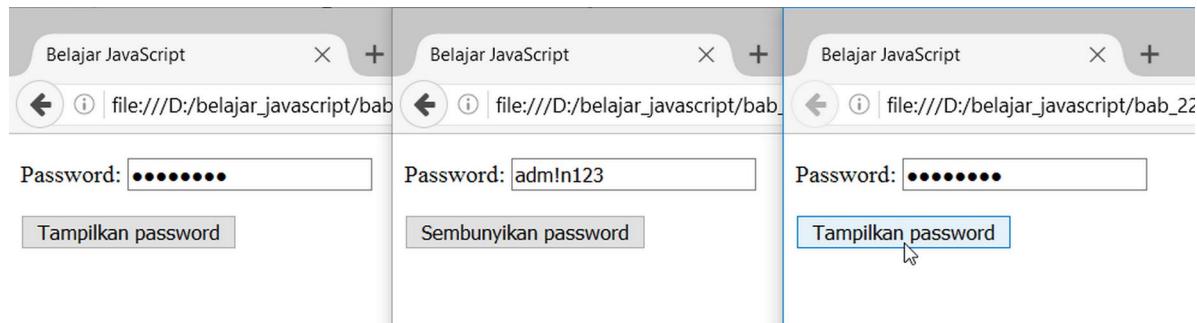


Gambar: Menampilkan teks asli dari input element type password

Disini saya membuat sebuah tag `<input type="password">` dan sebuah tombol. Ketika tombol di klik (terjadi event `click`), jalankan fungsi `proses()`. Isinya menukar property `type` dari `passwordNode` menjadi `text`.

Artinya, kode diatas akan menukar tag `<input type="password">` menjadi `<input type="text">`. Dengan demikian, isi teks tidak lagi tersamarkan karena tipe element dari tag `<input>` sudah berubah.

Sebagai latihan tambahan, dapatkah anda mengubah kode program diatas menjadi sebagai berikut?



Gambar: Menampilkan dan menyembunyikan password

Tambahannya, ketika tombol "Tampilkan password" di klik, password akan terlihat. Selain itu tombol juga akan berganti menjadi "Sembunyikan password".

Saat tombol "Sembunyikan password" di klik, tampilan teks password akan kembali disamarkan, dan nama di tombol juga akan berganti kembali menjadi "Tampilkan password" yang jika di klik akan kembali menampilkan password.

Untuk membuat program seperti ini, diperlukan sebuah kondisi `if else` untuk mengecek apakah saat ini password tampil atau tidak.

Berikut kode program yang saya gunakan:

```

1 <body>
2   <p>Password: <input type="password" id="password"></p>
3   <p><button id="tombol">Tampilkan password</button></p>
4   <script>
5     var passwordNode = document.getElementById("password");
6     var tombolPassNode = document.getElementById("tombol");
7
8     function proses(){
9       if (tombolPassNode.innerHTML === "Tampilkan password"){
10         passwordNode.type = "text";
11         tombolPassNode.innerHTML = "Sembunyikan password";
12       }
13     else {
14       passwordNode.type = "password";
15       tombolPassNode.innerHTML = "Tampilkan password";
16     }
17   }
18
19   tombolPassNode.addEventListener("click",proses);
20 </script>
21 </body>

```

Pengecekan kondisi saya rancang dengan melihat apa isi dari `tombolPassNode.innerHTML`. Jika isinya "Tampilkan password", berarti saat ini password masih tersembunyi. Jika kondisi ini terpenuhi (`true`), ubah tipe tag `<input>` menjadi `text`, lalu ubah `tombolPassNode.innerHTML` menjadi "Sembunyikan password".

Untuk kondisi berikutnya, saya tidak perlu mengecek lagi tapi langsung menggunakan kondisi `else`. Maksudnya, jika kondisi `tombolPassNode.innerHTML === "Tampilkan password"` menghasilkan nilai `false`, sudah pasti `tombolPassNode.innerHTML` akan berisi "Sembunyikan password". Jika ini yang terjadi, ubah tipe tag `<input>` menjadi `password`, lalu ubah `tombolPassNode.innerHTML` menjadi "Tampilkan password".

Sekarang, anda bisa menampilkan dan menyembunyikan password secara bergantian.

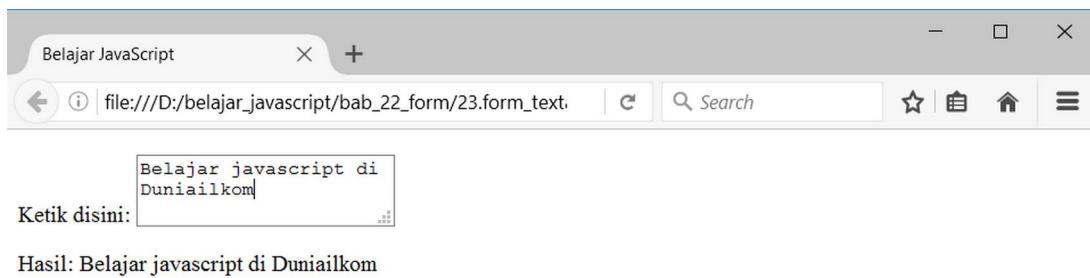
## 22.10 Textarea Element

**Textarea** adalah tag `<textarea>` yang digunakan untuk membuat kotak input teks dengan ukuran lebih besar dari pada tag `<input type="text">`. Walaupun texarea menggunakan nama yang berbeda, seluruh **atribut**, **property**, dan **event** untuk tag ini sama seperti yang tersedia di dalam `<input type="text">`.

Tag `<textarea>` tidak memiliki atribut `value`, namun property `value` tetap berisi text yang saat ini ditulis. Isi teks awal (jika diperlukan) ditulis diantara tag pembuka `<textarea>` dan tag penutup `</textarea>`.

Berikut contoh pengaksesan property `value` dari textarea:

```
1 <body>
2   <p>Ketik disini: <textarea id="ketik">Belajar javascript</textarea></p>
3   <p>Hasil: <span id="hasil"></span></p>
4   <script>
5     var ketikNode = document.getElementById("ketik");
6     var hasilNode = document.getElementById("hasil");
7
8     function diProses(){
9       hasilNode.innerHTML = ketikNode.value;
10    }
11    ketikNode.addEventListener("keyup",diProses);
12  </script>
13 </body>
```



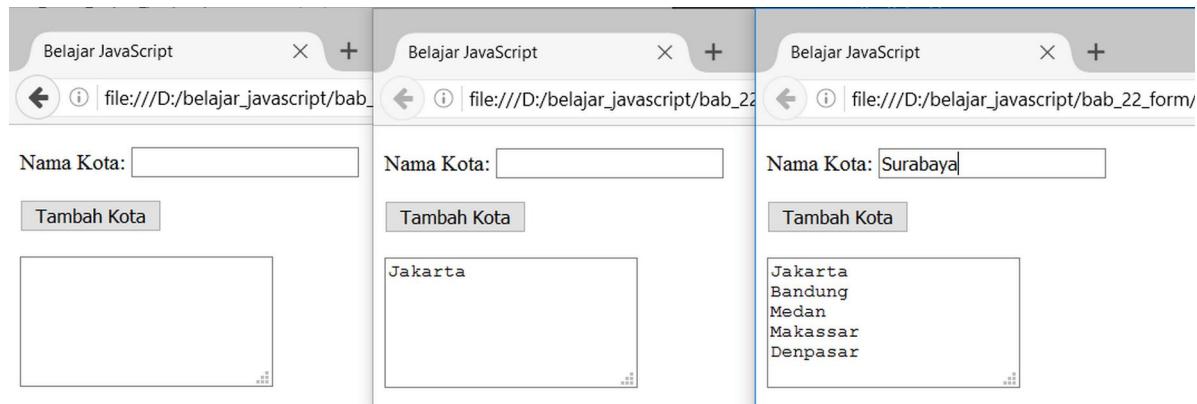
Gambar: Menampilkan teks dari tag <textarea>

Disini saya membuat tag <textarea id="ketik"> dan tag <span id="hasil">. Saat teks di ketik di dalam textarea, langsung copy nilai atribut value ke tag span. Ini dijalankan dengan bantuan event **keyup**.

Saat program dijalankan, text area sudah berisi teks "Belajar javascript". Namun text ini belum ter-copy ke dalam tag span. Hanya saat sebuah karakter lain mulai di ketik, barulan teks di copy, karena proses copy teks ini butuh event **keyup**.

Perhatikan bahwa nilai teks dari textarea disimpan di dalam `ketikNode.value`, meskipun textarea itu sendiri tidak memiliki atribut `value`.

Sebagai contoh kedua, saya ingin merancang teks inputan yang nilainya akan ditampung ke dalam textarea. Seperti tampilan berikut:



Gambar: Membuat daftar nama kota dengan text box dan text area

Nama kota diinput dari text box. Saat tombol "Tambah Kota" di klik, pindahkan isi teks tersebut ke dalam textarea. Text box kemudian menjadi kosong dan bisa diinput kembali nama kota kedua, ketiga, dst. Bisakah anda merancang kode program seperti ini?

Kunci dari aplikasi ini ada di cara kita “menumpuk” nama kota di dalam textarea.

Berikut kode program yang saya gunakan:

```

1 <body>
2   <p>Nama Kota: <input type="text" id="namaKota"></p>
3   <p><button id="tambahKota">Tambah Kota</button></p>
4   <p><textarea id="daftarKota" rows="5" cols="20"></textarea></p>
5   <script>
6     var namaKotaNode = document.getElementById("namaKota");
7     var tambahKotaNode = document.getElementById("tambahKota");
8     var daftarKotaNode = document.getElementById("daftarKota");
9
10    function diTambahKota(){
11      var namaKotaSekarang = namaKotaNode.value + "\n";
12      var daftarKotaSekarang = daftarKotaNode.value;
13      daftarKotaNode.value = daftarKotaSekarang + namaKotaSekarang;
14      namaKotaNode.value = "";
15      namaKotaNode.focus();
16    }
17
18    tambahKotaNode.addEventListener("click",diTambahKota);
19  </script>
20 </body>
```

Disini saya memiliki 3 buah node object: namaKotaNode untuk tag `<input id="namaKota">`, tambahKotaNode untuk tag `<button id="tambahKota">`, dan daftarKotaNode untuk tag `<textarea id="daftarKota">`.

Saat tombol "Tambah Kota" di klik, jalankan function `diTambahKota()`. Dalam function ini, variabel `namaKotaSekarang` akan berisi nilai teks yang di dapat dari `namaKotaNode.value`.

Artinya, variabel `namaKotaSekarang` berisi nama kota yang saat ini sedang diketik di dalam teks box.

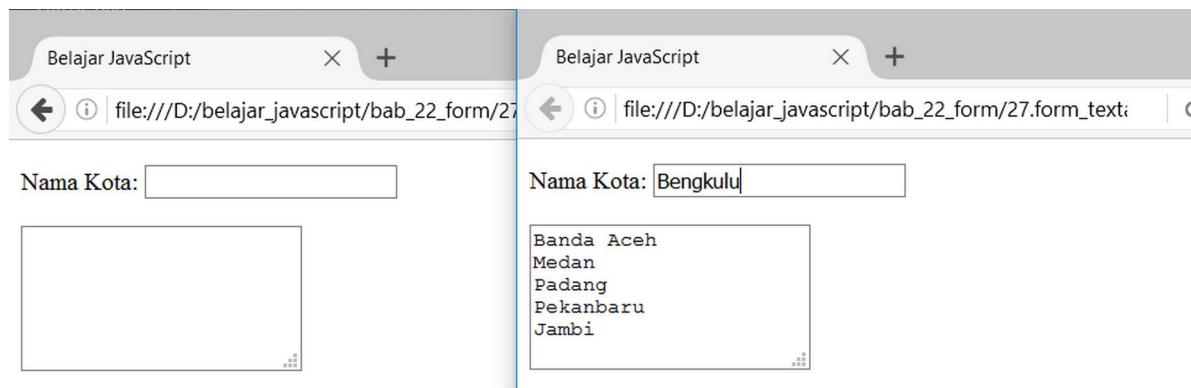
Perhatikan tambahan karakter "`\n`". Karakter "`\n`" adalah "*carriage return*", yakni karakter yang membuat efek spasi kebawah seperti layaknya menekan tombol "Enter" di dalam keyboard. Jika ini tidak ditambahkan, teks akan disambung ke kanan terus menerus (bukan secara vertikal ke bawah).

Variabel `namaKotaSekarang` tidak bisa saya input langsung ke dalam textarea karena akan menimpa nilai yang nantinya ada di dalam textarea. Saya harus "mengambil" nilai yang ada di dalam textarea terlebih dahulu, dan disimpan kedalam variabel `daftarKotaSekarang`. Isinya berupa nilai teks yang saat ini ada di dalam textarea, yakni hasil dari `daftarKotaNode.value`.

Selanjutnya, `daftarKotaNode.value` akan diisi dengan hasil dari penambahan string `daftarKotaSekarang + namaKotaSekarang`. Disinilah nilai textarea *diupdate* dengan penambahan nama kota baru.

Terakhir saya mengosongkan isi textbox menggunakan perintah `namaKotaNode.value = ""`, dan memberinya event **focus** lewat perintah `namaKotaNode.focus()`. Dengan demikian, saat nilai kota diambil, textbox kembali kosong dan cursor keyboard langsung berada di dalam textbox (event **focus**).

Sebagai latihan tambahan, bisakah anda merevisi kode program diatas dengan menghilangkan tombol "Tambah Kota"? Untuk menambah nama kota kita cukup menekan tombol "Enter" di keyboard.



Gambar: Menambah nama kota tanpa bantuan tombol

Tipsnya, manfaatkan event **keypress** untuk memeriksa apa tombol yang sedang diketik saat ini. Jika itu tombol "Enter", pindahkan teks ke textarea. Silahkan anda coba terlebih dahulu karena sekaligus melatih pemahaman tentang **keyboard event**.

Baik, berikut kode program yang saya gunakan:

```

1 <body>
2   <p>Nama Kota: <input type="text" id="namaKota"></p>
3   <p><textarea id="daftarKota" rows="5" cols="20"></textarea></p>
4   <script>
5     var namaKotaNode = document.getElementById("namaKota");
6     var daftarKotaNode = document.getElementById("daftarKota");
7
8     function diTambahKota(e){
9       if (e.key == "Enter"){
10         var namaKotaSekarang = namaKotaNode.value + "\n";
11         var daftarKotaSekarang = daftarKotaNode.value;
12         daftarKotaNode.value = daftarKotaSekarang + namaKotaSekarang;
13         namaKotaNode.value = "";
14         namaKotaNode.focus();
15     }
16   }
17
18   namaKotaNode.addEventListener("keypress",diTambahKota);
19 </script>
20 </body>

```

Tidak banyak penambahan. Saya menghapus tag `<button>` dari kode HTML. Event `keypress` kemudian di “tempel” ke `namaKotaNode`, yakni node object dari `<input id="namaKota">`.

Setiap terjadi event `keypress`, jalankan function `diTambahKota()`. Di dalam function ini, pertama kali akan diperiksa apakah `e.key == "Enter"`. Jika hasilnya `true`, yakni karakter yang ditekan di keyboard adalah "Enter", lakukan proses pemindahan.



Di dalam DOM W3C, `textarea` merupakan object dari `HTMLTextAreaElement`, bukan lagi masuk ke `HTMLInputElement` sebagaimana tag `<input>`. `HTMLTextAreaElement` memiliki beberapa property khusus `textarea` seperti `rows` dan `cols`.

Karena keterbatasan tempat, saya tidak akan membahasnya. Anda bisa melihat daftar lengkap dari property ini ke [HTMLTextAreaElement References<sup>7</sup>](#).

## 22.11 Input Element: Type Checkbox

Element form berikutnya yang akan kita bahas adalah **checkbox**. Checkbox berbentuk tombol checklist yang bisa dipilih. Element ini dibuat menggunakan tag HTML: `<input type="checkbox">`.



Karena masih dibuat dari tag `<input>`, checkbox termasuk ke dalam `HTMLInputElement` object.

---

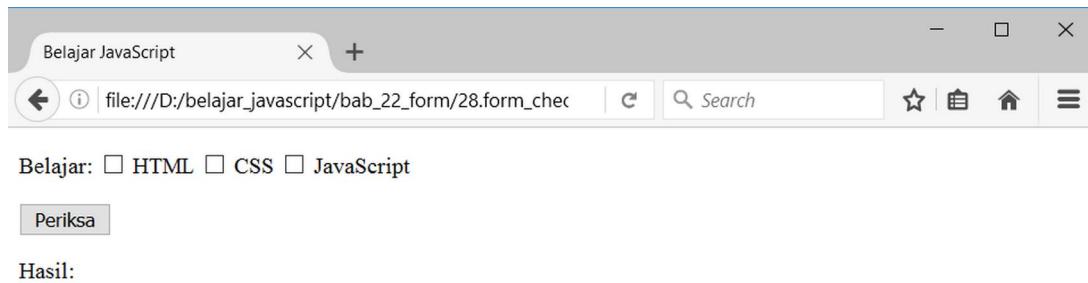
<sup>7</sup><https://developer.mozilla.org/en-US/docs/Web/API/HTMLTextAreaElement>

Sama seperti *textarea* dan *input type password*, mayoritas **atribut**, **property**, dan **event** untuk checkbox sama seperti element input form biasa, termasuk atribut **value**, event **focus**, **blur**, **click**, dll.

Hal yang paling sering kita lakukan dengan checkbox adalah memeriksa apakah sebuah checkbox sudah dipilih atau tidak. Untuk keperluan ini, checkbox memiliki property **checked**. Property **checked** akan mengembalikan nilai boolean **true** jika checkbox tersebut terpilih (dicentang) atau mengembalikan nilai boolean **false** jika checkbox tidak terpilih.

Mari kita lihat contoh kode programnya:

```
1 <body>
2   <p>Belajar:
3     <input type="checkbox" id="belajarHTML" value="HTML"> HTML
4     <input type="checkbox" id="belajarCSS" value="CSS"> CSS
5     <input type="checkbox" id="belajarJS" value="JavaScript"> JavaScript
6   </p>
7   <p><button id="periksa">Periksa</button></p>
8   <p>Hasil: <span id="hasil"></span></p>
9 <script>
10  var belajarHTMLNode = document.getElementById("belajarHTML");
11  var belajarCSSNode = document.getElementById("belajarCSS");
12  var belajarJSNode = document.getElementById("belajarJS");
13  var periksaNode = document.getElementById("periksa");
14  var hasilNode = document.getElementById("hasil");
15
16  function diPeriksa(){
17    hasilNode.innerHTML = "";
18
19    if(belajarHTMLNode.checked) {
20      hasilNode.innerHTML += belajarHTMLNode.value + " ";
21    }
22    if(belajarCSSNode.checked) {
23      hasilNode.innerHTML += belajarCSSNode.value + " ";
24    }
25    if(belajarJSNode.checked) {
26      hasilNode.innerHTML += belajarJSNode.value + " ";
27    }
28  }
29
30  periksaNode.addEventListener("click",diPeriksa);
31 </script>
32 </body>
```



Gambar: Memeriksa apakah checkbox terpilih atau tidak

Di dalam kode HTML, saya membuat 3 buat checkbox dengan `id="belajarHTML"`, `id="belajarCSS"` dan `id="belajarJS"`. Semua checkbox ini akan disimpan kedalam variabel `belajarHTMLNode`, `belajarCSSNode` dan `belajarJSNode`.

Selanjutnya terdapat sebuah tombol dengan `id="periksa"`. Node object dari tombol ini disimpan ke dalam variabel `periksaNode`.

Terakhir terdapat span element dengan `id="hasil1"`. Node object untuk tag span disimpan ke dalam variabel `hasil1Node`.

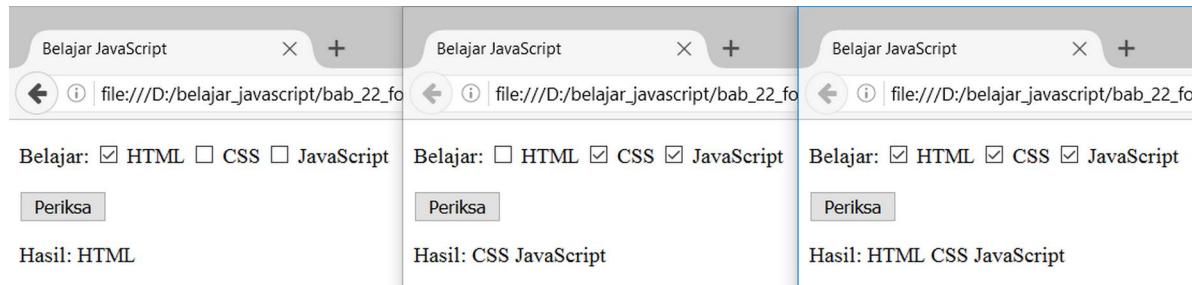
Maksud dari kode program diatas adalah, ketika tombol "Periksa" di klik, Cek ketiga checbox. Jika ada yang di pilih (di *checklist*), ambil nilai `value` dari checkbox tersebut kemudian tampilkan ke dalam `hasil1Node`.

Proses pemeriksaan berlangsung di dalam function `diPeriksa()` dan ditangani dengan 3 kondisi `if`, dimana terdapat 1 kondisi `if` untuk setiap checkbox.

Kondisi `if(belajarHTMLNode.checked)` hanya akan bernilai `true` jika tag `<input type="checkbox" id="belajarHTML">` di checklist.

Apabila bernilai `true`, jalankan perintah `hasil1Node.innerHTML += belajarHTMLNode.value + " "`. Artinya, ambil nilai `belajarHTMLNode.value` saat ini, kemudian tambahkan dengan nilai `belajarHTMLNode.value` dan sebuah spasi.

Hal yang sama berlaku untuk ketiga checkbox. Mari kita coba:

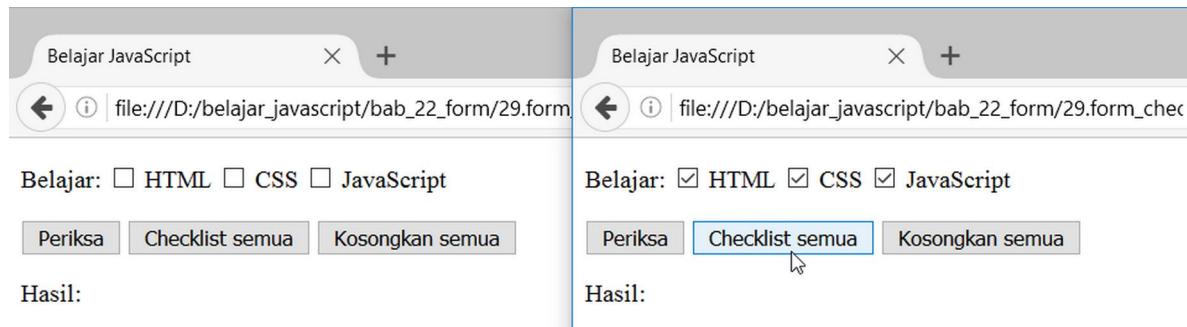


Gambar: Saat tombol "Periksa" di klik, tampilkan isi dari checkbox yang dipilih

Saat tombol "Periksa" di klik, isi dari tag `<span>` akan menampilkan pilihan checkbox apa saja yang saat ini sedang dipilih.

Property `checked` tidak hanya bisa dibaca, tapi juga bisa diisi nilai. Jika ini yang terjadi, tombol checkbox tersebut akan terpilih/dihapus tergantung nilai yang diinput. Jika diinput `true`, checkbox akan terpilih. Jika diinput `false`, pilihan checkbox akan dihapus.

Merevisi kode sebelumnya, saya ingin menambahkan 2 buah tombol: "Checklist semua" dan "Kosongkan semua". Ketika tombol ini di klik, checkbox akan tercentang semua atau di kosongkan semua (tidak dipilih). Berikut tampilan akhir dari kode program tersebut:



Gambar: Tambahan tombol "Checklist semua" dan "Kosongkan semua"

Untuk membuat yang seperti ini, saya cukup memberikan nilai **true** atau **false** ke semua checkbox. Karena programnya menjadi cukup panjang, saya akan menampilkan kode yang perlu ditambahkan saja:

```

1 <button id="checklistSemua">Checklist semua</button>
2 <button id="kosongkanSemua">Kosongkan semua</button>
3 <script>
4   var checklistSemuaNode = document.getElementById("checklistSemua");
5   var kosongkanSemuaNode = document.getElementById("kosongkanSemua");
6
7   function diChecklist(){
8     belajarHTMLNode.checked = true;
9     belajarCSSNode.checked = true;
10    belajarJSNode.checked = true;
11  }
12
13  function diKosongkan(){
14    belajarHTMLNode.checked = false;
15    belajarCSSNode.checked = false;
16    belajarJSNode.checked = false;
17  }
18
19  checklistSemuaNode.addEventListener("click",diChecklist);
20  kosongkanSemuaNode.addEventListener("click",diKosongkan);
21 </script>
```

Yup, kita hanya perlu memberikan nilai **true** ke property `belajarHTMLNode.checked`, `belajarCSSNode.checked` dan `belajarJSNode.checked` saat tombol "Checklist semua" di klik, dan untuk tombol "Kosongkan semua", isi ketiga property dengan nilai **false**.

Sekedar tambahan, kita bisa men-klik kotak checkbox untuk membuatnya ter-checklist. Artinya, jika saya merancang sebuah kode program yang bisa menjalankan event `click` untuk checkbox, maka checkbox tersebut akan ter-checklist secara otomatis.

Dengan demikian, untuk men-cheklist sebuah chekbox, juga menggunakan perintah berikut:

```
belajarHTMLNode.click();
```

Perintah `belajarHTMLNode.click()` artinya panggil event `click` kepunyaan `belajarHTMLNode`. Jika `belajarHTMLNode` adalah sebuah checkbox, maka checkbox itu akan langsung di-checklist.

## 22.12 Input Element: Type Radio

**Input element type radio** atau disebut juga sebagai **radio button** sangat mirip seperti checkbox. Bedanya, di radio button hanya bisa memilih satu nilai dari pilihan yang ada. Untuk membuat radio, bisa menggunakan tag HTML `<input type="checkbox">`.

-  Karena masih dibuat dari tag `<input>`, radio button termasuk ke dalam `HTMLInputElement` object.

Dan juga sama seperti checkbox, radio button mendukung semua **atribut**, **property** dan **method** seperti input element type text. Untuk menentukan apakah sebuah radio button dipilih atau tidak, juga disimpan ke dalam property `checked`. Penggunaannya sama persis seperti checkbox.

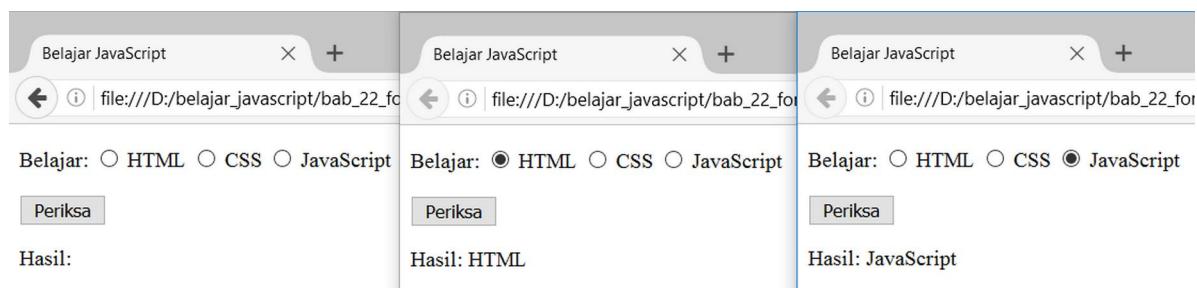
Berikut contoh kode programnya:

```
1 <body>
2   <p>Belajar:
3     <input type="radio" name="belajar" id="belajarHTML" value="HTML">
4     HTML
5     <input type="radio" name="belajar" id="belajarCSS" value="CSS">
6     CSS
7     <input type="radio" name="belajar" id="belajarJS" value="JavaScript">
8     JavaScript
9   </p>
10  <p><button id="periksa">Periksa</button></p>
11  <p>Hasil: <span id="hasil"></span></p>
12  <script>
13    var belajarHTMLNode = document.getElementById("belajarHTML");
14    var belajarCSSNode = document.getElementById("belajarCSS");
15    var belajarJSNode = document.getElementById("belajarJS");
16    var periksaNode = document.getElementById("periksa");
17    hasilNode = document.getElementById("hasil");
18
19    function diPeriksa(){
20      hasilNode.innerHTML = "";
21
22      if(belajarHTMLNode.checked) {
```

```

23     hasilNode.innerHTML = belajarHTMLNode.value + " ";
24 }
25 if(belajarCSSNode.checked) {
26     hasilNode.innerHTML = belajarCSSNode.value + " ";
27 }
28 if(belajarJSNode.checked) {
29     hasilNode.innerHTML = belajarJSNode.value + " ";
30 }
31 }
32
33 periksaNode.addEventListener("click",diPeriksa);
34 </script>
35 </body>

```



Gambar: Memeriksa pilihan radio button

Kode program diatas sangat mirip seperti contoh yang saya gunakan pada checkbox. Bedanya, tag `<input type="checkbox">` diganti menjadi `<input type="radio">`. Selain itu untuk membuat radion button bergabung dalam satu kelompok, harus memiliki atribut name yang sama. Dalam contoh diatas, ketiga radio button memiliki atribut name="belajar".

Saat tombol "Periksa" di **click**, cek satu persatu apakah property checked bernilai **true** atau **false**. Jika **true**, ambil nilai atribut value, kemudian input ke `hasilNode.innerHTML`.

Program diatas berjalan seperti yang diharapkan. Tapi bagaimana jika pilihan radio button lumayan banyak? akan cukup merepotkan jika kita harus memeriksa setiap checkbox dan membuat satu persatu kondisi **if**. Apakah ada cara yang lebih praktis?

Tentu, kita bisa menggunakan sebuah perulangan, tapi kode programnya memang sedikit lebih rumit:

```

1 <body>
2   <p>Belajar:
3     <input type="radio" name="belajar" id="belajarHTML" value="HTML">
4     HTML
5     <input type="radio" name="belajar" id="belajarCSS" value="CSS">
6     CSS
7     <input type="radio" name="belajar" id="belajarJS" value="JavaScript">
8     JavaScript
9     <input type="radio" name="belajar" id="belajarPHP" value="PHP">

```

```

10     PHP
11     <input type="radio" name="belajar" id="belajarMYSQL" value="MySQL">
12     MySQL
13     </p>
14     <p><button id="periksa">Periksa</button></p>
15     <p>Hasil: <span id="hasil"></span></p>
16     <script>
17         var belajarNode = document.querySelectorAll("input");
18         var periksaNode = document.getElementById("periksa");
19         var hasilNode = document.getElementById("hasil");
20
21         function diPeriksa(){
22             // console.log(belajarNode);
23             var jumlahRadio = belajarNode.length;
24             for (var i = 0; i < jumlahRadio; i++){
25                 if (belajarNode[i].checked === true){
26                     hasilNode.innerHTML = belajarNode[i].value;
27                 }
28             }
29         }
30
31         periksaNode.addEventListener("click",diPeriksa);
32     </script>
33 </body>

```



Gambar: Memeriksa 5 pilihan radio button

Di dalam kode HTML terdapat 5 buah pilihan radio button: **HTML, CSS, JavaScript, PHP** dan **MySQL**. Jika menggunakan cara sebelumnya, saya harus mengecek satu persatu kelima radio button ini. Dengan menggunakan perulangan, kode programnya bisa jadi lebih singkat.

Persiapan untuk perulangan dimulai dari variabel `belajarNode`. Variabel ini berisi *collection* dari seluruh tag `<input>`. Ini di dapat dari perintah `document.querySelectorAll("input")`.

Pemrosesan akan berjalan saat tombol "Periksa" di klik, dimana function `diPeriksa()` akan berjalan (terjadi event `click` untuk tag `<button id="periksa">`).

Di baris pertama terdapat kode program yang saya buat sebagai komentar, yakni baris `// console.log(belajarNode)`. Kode ini berfungsi untuk memastikan variabel `belajarNode` sudah berisi *collection* yang ingin diproses.

Saya sangat menyarankan anda untuk selalu memeriksa variabel penting dengan cara seperti ini, sekedar memastikan bahwa isinya sesuai dengan keinginan. Perintah `console.log()` mirip seperti perintah `var_dump()` di PHP. Kita bisa memeriksa dengan detail apa isi sebenarnya dari sebuah variabel.

Setelah memastikan isi dari variabel `belajarNode` adalah *collection* dari kelima radio button, saya membuat variabel kedua, yakni `jumlahRadio` yang diisi dengan nilai `belajarNode.length`.

Perintah `belajarNode.length` akan mengembalikan jumlah element yang ada di dalam *collection* `belajarNode`. Karena terdapat 5 tag radio button, maka variabel `jumlahRadio` akan berisi angka 5.

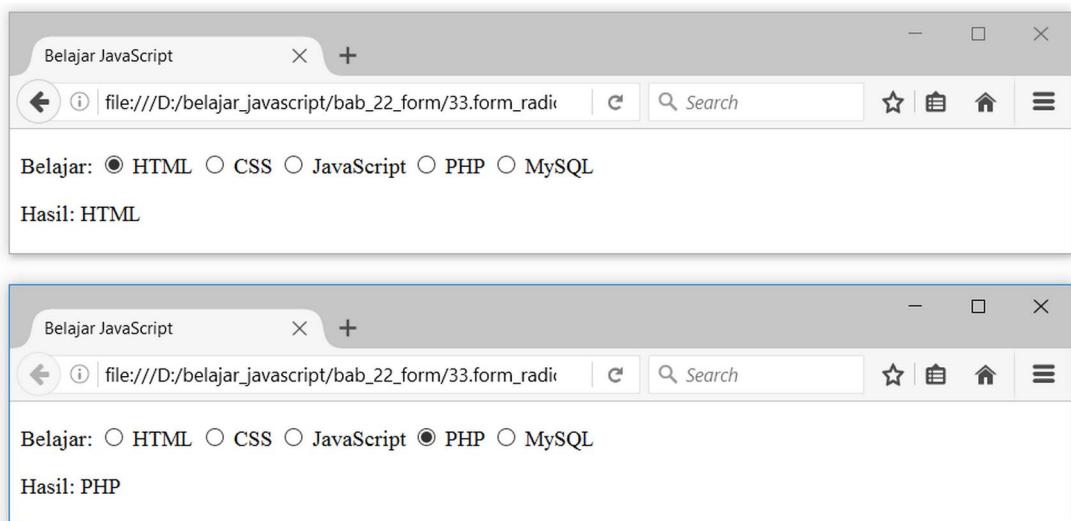
Selanjutnya kita masuk ke dalam perulangan `for (var i = 0; i < jumlahRadio; i++)`. Disini perulangan akan dijalankan sebanyak 5 kali, mulai dari variabel `i` berisi angka 0, 1, 2, 3, hingga 4.

Untuk setiap perulangan, cek apakah `belajarNode[i].checked === true`. Ingat, `belajarNode` merupakan sebuah *collection* yang berfungsi layaknya array. Perintah ini akan mengecek `if (belajarNode[0].checked === true)`, `if (belajarNode[1].checked === true)`, dst hingga `if (belajarNode[4].checked === true)`.

Jika didapati hasil property `checked` bernilai `true`, yang berarti radio button tersebut sedang dalam kondisi terpilih, jalankan perintah `hasilNode.innerHTML = belajarNode[i].value`. Ini bertujuan untuk mengisi tag `<span id="hasil">` dengan nilai atribut `value` dari radio button tersebut.

Jika masih kurang paham, silahkan anda pelajari secara perlahan penjelasan diatas. Trik seperti ini akan sering kita buat untuk memproses isian form yang cukup kompleks.

Modifikasi terakhir, bisakah anda mengubah sedikit kode program kita dengan menghilangkan tombol "Periksa"? Maksudnya, kita cukup pilih salah satu radio button, dan nilai atribut `value` akan langsung tampil di `<span id="hasil">`.



Gambar: Menampilkan hasil radio button tanpa tombol "Periksa"

Triknya, kita bisa dengan memanfaatkan efek **event bubble**. Cukup pindahkan event *click* dari tag `<button>` ke dalam tag `<p>`. Dengan cara mengubah baris:

```
var periksaNode = document.getElementById("periksa");
```

Menjadi:

```
var periksaNode = document.querySelector("p");
```

Hasilnya, ketika kita men-klik tombol radio button, secara tidak langsung akan menjalankan event **click** milik tag `<p>`. Event **click** ini selanjutnya akan menjalankan function `diPeriksa()` seperti yang kita bahas sebelumnya.

Disini saya menggunakan `document.querySelector("p")` dengan alasan di dalam tag `<p>` tidak terdapat atribut `id`. Jika perintah ini diganti dengan `document.getElementById()` juga tidak ada masalah selama anda juga menambahkan atribut `id` ke dalam tag `<p>`. Ini disarankan jika di dalam halaman terdapat lebih dari 1 tag `<p>`.

## 22.13 Select Element

Jika dibandingkan dengan element form lain, **select element** mungkin yang paling kompleks, karena kita butuh gabungan tag `<select>` dan tag `<option>` untuk membuat sebuah pilihan dropdown.

Di dalam DOM W3C, select element dikelompokkan ke dalam object tersendiri, yakni **HTMLSelectElement** dan **HTMLOptionElement**. **HTMLSelectElement** adalah object untuk tag `<select>`, sedangkan **HTMLOptionElement** adalah object untuk tag `<option>`.

Sebagaimana biasa, karena ini adalah sebuah object baru, kita akan lihat apa saja **property** yang bisa diakses dari **HTMLSelectElement** dan **HTMLOptionElement**.

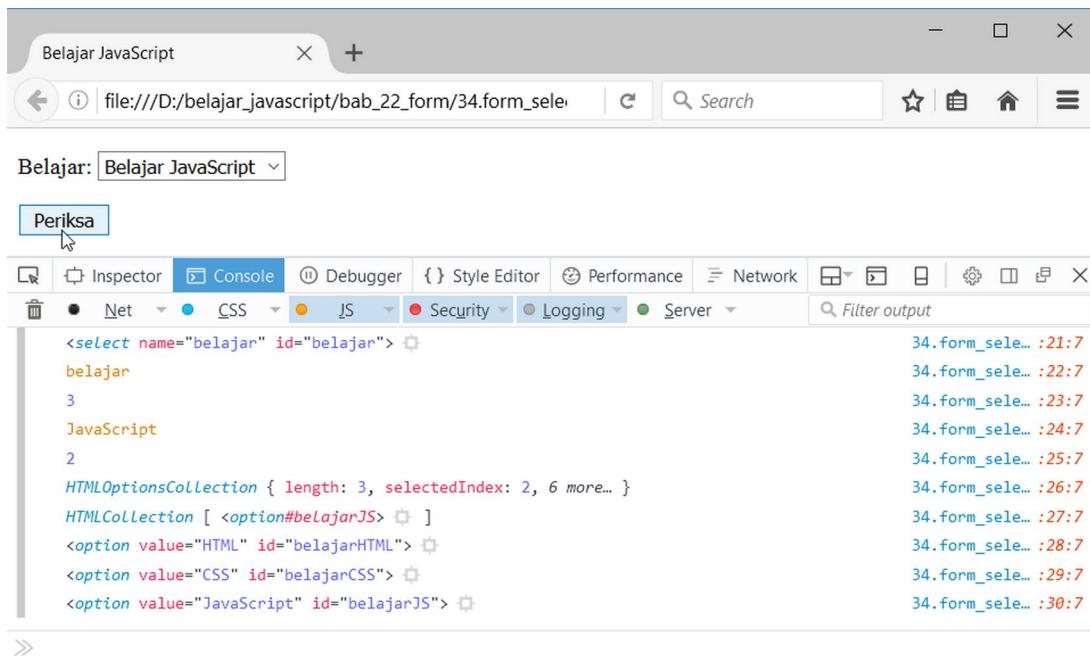
Kita mulai dari **HTMLSelectElement** terlebih dahulu:

```
1 <body>
2   <p>Belajar:
3     <select name="belajar" id="belajar">
4       <option value="HTML" id="belajarHTML">Belajar HTML</option>
5       <option value="CSS" id="belajarCSS">Belajar CSS</option>
6       <option value="JavaScript" id="belajarJS">Belajar JavaScript</option>
7     </select>
8   </p>
9   <p><button id="periksa">Periksa</button></p>
10  <script>
11    var periksaNode = document.getElementById("periksa");
12    var belajarNode = document.getElementById("belajar");
13  
```

```

14  function diPeriksa(){
15    console.log (belajarNode);
16    // <select name="belajar" id="belajar">
17    console.log (belajarNode.name);           // belajar
18    console.log (belajarNode.length);         // 3
19    console.log (belajarNode.value);          // JavaScript
20    console.log (belajarNode.selectedIndex);   // 2
21    console.log (belajarNode.options);
22    // HTMLOptionsCollection { length: 3, selectedIndex: 2, 6 more... }
23    console.log (belajarNode.selectedOptions);
24    // HTMLCollection [ <option#belajarJS> ]
25    console.log (belajarNode[0]);
26    // <option value="HTML" id="belajarHTML">
27    console.log (belajarNode[1]);
28    // <option value="CSS" id="belajarCSS">
29    console.log (belajarNode[2]);
30    // <option value="JavaScript" id="belajarJS">
31  }
32
33  periksaNode.addEventListener("click",diPeriksa);
34  </script>
35 </body>

```



Gambar: Berbagai property dari tag <select> (HTMLSelectElement)

Disini saya membuat sebuah pilihan dropdown menggunakan tag <select>. Terdapat 3 buah pilihan yang tersedia, yakni Belajar HTML, Belajar CSS dan Belajar JavaScript. Pilihan ini berasal dari tag <option>.

Saat tombol "Periksa" di klik, jalankan function diPeriksa(). Function ini berisi berbagai property dari **HTMLSelectElement**. Berikut penjelasan dari property tersebut:

- **HTMLSelectElement.name**: Berisi nilai atribut name dari tag <select>.
- **HTMLSelectElement.length**: Berisi jumlah pilihan yang tersedia. Pilihan ini dibuat dari tag <option>.
- **HTMLSelectElement.value**: Berisi nilai atribut value dari tag <option> yang saat ini dipilih.
- **HTMLSelectElement.selectedIndex**: Berisi nomor index dari tag <option> yang saat ini dipilih. Nomor index dimulai dari 0 untuk tag <option> pertama.
- **HTMLSelectElement.options**: Berisi object **HTMLOptionsCollection**, sebuah collection yang berisi seluruh tag <option> penyusun <select>.
- **HTMLSelectElement.selectedOptions**: Berisi object **HTMLCollection**, sebuah collection dari tag <option> yang saat ini terpilih.
- **HTMLSelectElement[0]**: Berisi object dari tag <option> di index ke-0 atau urutan pertama.
- **HTMLSelectElement[1]**: Berisi object dari tag <option> di index ke-1 atau urutan kedua.
- **HTMLSelectElement[2]**: Berisi object dari tag <option> di index ke-2 atau urutan ketiga.

Dari berbagai property ini, yang akan paling sering kita akses adalah property **value** dan **selectedIndex**. Karena di kedua property inilah bisa dilihat apa pilihan yang saat ini dipilih oleh pengguna. Silahkan anda ubah pilihan dropdown lalu tekan tombol "Periksa". Nilai dari property **value** dan **selectedIndex** juga akan berubah.



Daftar lengkap mengenai property dari **HTMLSelectElement** dapat anda lihat di [HTMLSelectElement References<sup>8</sup>](#).

Selain **HTMLSelectElement**, tag <select> juga membutuhkan dari tag <option> atau **HTMLOptionElement**. Berikut beberapa property dari **HTMLOptionElement**:

```
1 <body>
2   <p>Belajar:
3     <select name="belajar" id="belajar">
4       <option value="HTML" id="belajarHTML">Belajar HTML</option>
5       <option value="CSS" id="belajarCSS" selected>Belajar CSS</option>
6       <option value="JavaScript" id="belajarJS">Belajar JavaScript</option>
7     </select>
8   </p>
9   <p><button id="periksa">Periksa</button></p>
10  <script>
11    var belajarCSSNode = document.getElementById("belajarCSS");
12    var periksaNode = document.getElementById("periksa");
```

<sup>8</sup><https://developer.mozilla.org/en-US/docs/Web/API/HTMLSelectElement>

```
14  function diPeriksa(){
15    console.log (belajarCSSNode);
16    // <option value="CSS" id="belajarCSS" selected="">
17    console.log (belajarCSSNode.value);           // CSS
18    console.log (belajarCSSNode.index);          // 1
19    console.log (belajarCSSNode.text);           // Belajar CSS
20    console.log (belajarCSSNode.selected);        // true
21    console.log (belajarCSSNode.defaultSelected); // true
22    console.log (belajarCSSNode.disabled);        // false
23  }
24
25  periksaNode.addEventListener("click",diPeriksa);
26  </script>
27 </body>
```

Saya masih menggunakan contoh yang sama seperti sebelumnya. Bedanya di dalam JavaScript saya memeriksa variabel `belajarCSSNode`. Variabel ini berisi node object yang di berasal dari perintah `document.getElementById("belajarCSS")`. Di dalam HTML, atribut `id="belajarCSS"` di miliki oleh sebuah tag `<option>`.

Berikut penjelasan dari property-property tersebut:

- **HTMLOptionElement.value**: Berisi nilai atribut `value` dari tag `<option>`.
- **HTMLOptionElement.index**: Berisi nomor index tag `<option>` yang dihitung secara berurutan dari tag `<select>`.
- **HTMLOptionElement.text**: Berisi teks yang ada di dalam tag `<option>`. Ini mirip seperti `innerHTML` untuk element lain.
- **HTMLOptionElement.selected**: Berisi nilai boolean apakah tag `<option>` saat ini dipilih atau tidak.
- **HTMLOptionElement.defaultSelected**: Berisi nilai boolean apakah tag `<option>` di set sebagai pilihan default. Pilihan default aktif jika terdapat atribut `selected` di dalam tag `<option>`.
- **HTMLOptionElement.disabled**: Berisi nilai boolean apakah atribut `disabled` aktif atau tidak.

Terlihat sebagian besar dari property **HTMLOptionElement** digunakan untuk mengakses atribut dari tag `<option>`.



Daftar lengkap mengenai property dari `HTMLOptionElement` dapat anda lihat di [HTMLOptionElement References<sup>9</sup>](#).

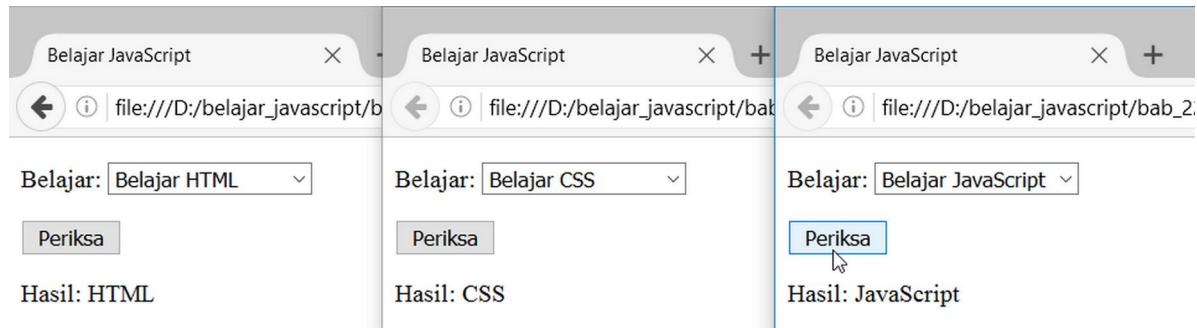
---

<sup>9</sup><https://developer.mozilla.org/en-US/docs/Web/API/HTMLOptionElement>

## Menampilkan Pilihan Select

Dengan bermodalkan berbagai property dari tag `<select>` dan `<option>`, kita sudah bisa mulai praktek memproses isian `select` element.

Contoh pertama adalah menampilkan isi dari pilihan tag `<select>`. Berikut tampilan akhir yang akan dirancang:



Gambar: Menampilkan pilihan tag `<select>`

Tag `<select>` yang digunakan masih sama seperti sebelumnya, yakni terdiri dari 3 pilihan: Belajar HTML, Belajar CSS, dan Belajar JavaScript. Saat tombol "Periksa" di klik, tampilkan nilai value dari `<option>` yang sedang dipilih.

Berikut kode programnya:

```

1 <body>
2   <p>Belajar:
3     <select name="belajar" id="belajar">
4       <option value="HTML" id="belajarHTML">Belajar HTML</option>
5       <option value="CSS" id="belajarCSS">Belajar CSS</option>
6       <option value="JavaScript" id="belajarJS">Belajar JavaScript</option>
7     </select>
8   </p>
9   <p><button id="periksa">Periksa</button></p>
10  <p>Hasil: <span id="hasil"></span></p>
11  <script>
12    var periksaNode = document.getElementById("periksa");
13    var belajarNode = document.getElementById("belajar");
14    var hasilNode = document.getElementById("hasil");
15
16    function diPeriksa(){
17      hasilNode.innerHTML = belajarNode.value;
18    }
19
20    periksaNode.addEventListener("click",diPeriksa);
21  </script>
22 </body>

```

Secara umum kode program ini mirip seperti kode-kode kita sebelumnya, dimana saat tombol <button id="periksa"> diklik, jalankan perintah hasilNode.innerHTML = belajarNode.value.

Variabel hasilNode adalah sebuah tag <span id="hasil"> yang digunakan untuk menampung hasil teks. Sedangkan variabel belajarNode berisi tag <select id="belajar">.

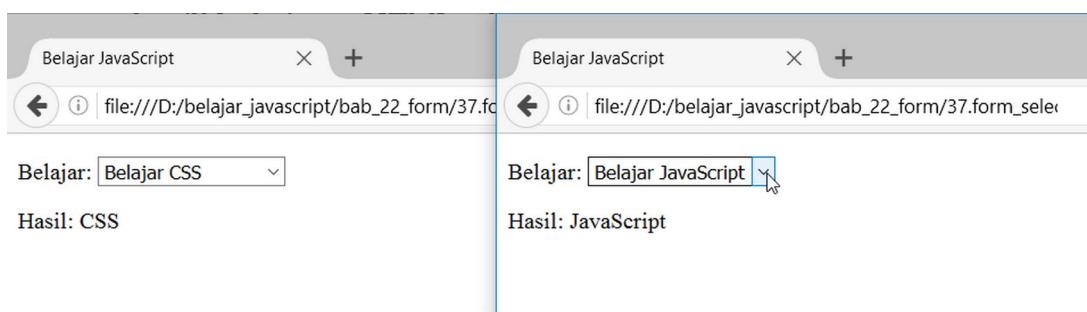
Untuk mencari nilai yang saat ini sedang dipilih, cukup dengan mengakses property value dari belajarNode.

Selanjutnya, bagaimana dengan menghapus tombol "Periksa" sama sekali? Saya ingin teks tampil begitu pilihan dropdown di tukar (tanpa harus men-klik tombol "Periksa").

Kita sudah beberapa kali membuat tantangan seperti ini. Kuncinya adalah, **event** mana yang akan dipakai sebagai pengganti tombol. Sepintas mungkin anda berfikir bagaimana jika menggunakan event **click** untuk belajarNode? Mari kita coba.

```

1 <body>
2   <p>Belajar:
3     <select name="belajar" id="belajar">
4       <option value="HTML" id="belajarHTML">Belajar HTML</option>
5       <option value="CSS" id="belajarCSS">Belajar CSS</option>
6       <option value="JavaScript" id="belajarJS">Belajar JavaScript</option>
7     </select>
8   </p>
9   <p>Hasil: <span id="hasil"></span></p>
10  <script>
11    var belajarNode = document.getElementById("belajar");
12    var hasilNode = document.getElementById("hasil");
13
14    function diPeriksa(){
15      hasilNode.innerHTML = belajarNode.value;
16    }
17
18    belajarNode.addEventListener("click",diPeriksa);
19  </script>
20 </body>
```

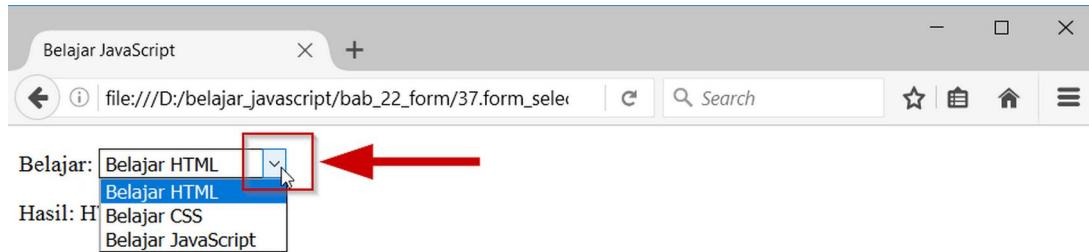


Gambar: Menampilkan hasil <select> tanpa tombol

Saya menghapus tag <button>, kemudian “menempelkan” event **click** ke dalam belajarNode. Artinya ketika tag <select> di klik, jalankan function diPeriksa(). Function diPeriksa() digunakan untuk menampilkan hasil tag <select>.

Sepintas tidak ada yang salah dari kode program tersebut. Saat saya menukar pilihan tag <select>, hasilnya langsung tampil ke dalam tag <span>.

Namun ada sedikit efek dari penggunaan event **click**. Ketika cursor mouse baru mulai memilih pilihan (saat men-klik tombol “v” di sisi kanan) hasilnya sudah langsung ke dalam tag <span>.



Gambar: Ketika men-klik pilihan, hasil akan langsung tampil

Ini terjadi karena event **click** akan langsung berjalan ketika tombol “v” di klik.

Walaupun nyaris tidak mengganggu, kadang efek seperti ini tidak diinginkan. Saya ingin teks baru akan tampil setelah pilihan di tukar, artinya kita bisa menggunakan event **change**.

Caranya, cukup mengganti baris:

```
belajarNode.addEventListener("click",diPeriksa);
```

Dengan:

```
belajarNode.addEventListener("change",diPeriksa);
```

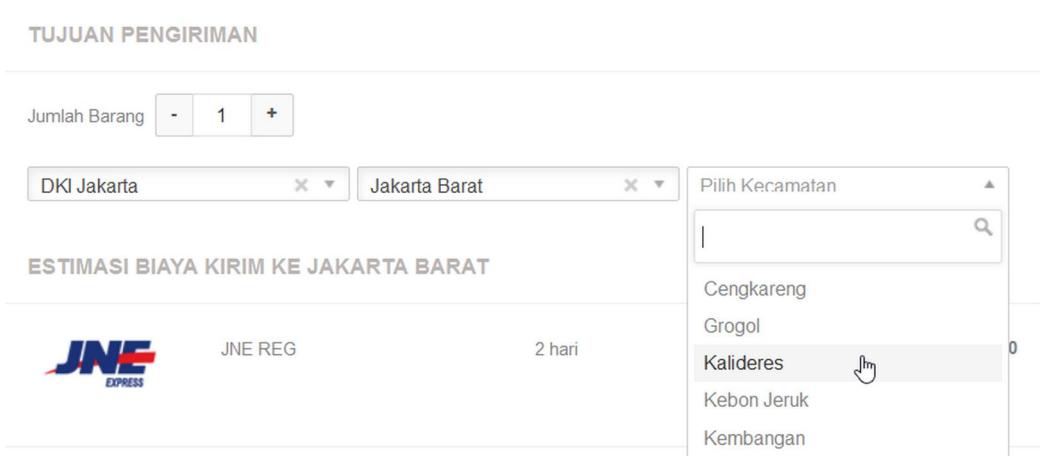
Dalam membuat program JavaScript, anda akan sering melakukan percobaan dengan berbagai event seperti ini. Apakah ada event lain yang dirasa lebih cocok, atau apakah sebuah event bisa membuat efek samping, dsb.

Ini sama seperti memutuskan apakah akan menggunakan event **kepress**, **keyup**, atau **keydown** untuk membaca teks keyboard. Silahkan anda berkreasi dengan berbagai event JavaScript untuk mencari yang paling sesuai.

## 22.14 Case Study: Membuat Dropdown Dinamis

Sebagai case study dari tag <select>, saya ingin membuat sebuah pilihan **dropdown dinamis**. Fitur seperti ini sering kita jumpai di internet. Dimana daftar pilihan akan berubah tergantung pilihan dropdown lainnya.

Sebagai contoh, inputan alamat biasanya butuh 3 data: nama provinsi, nama kota/kabupaten, dan nama kecamatan, sebagaimana yang digunakan oleh situs bukalapak dibawah ini:

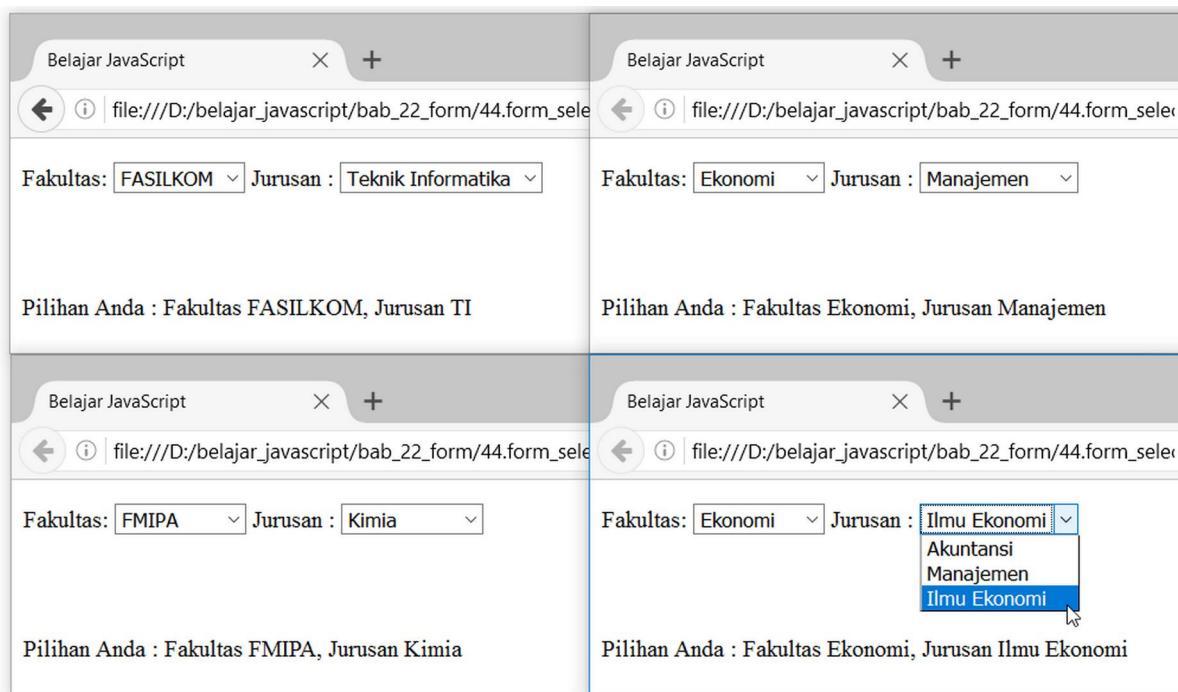


Gambar: Contoh menu dropdown dinamis dari situs bukalapak.com

Saat kita memilih nama provinsi **DKI Jakarta**, pilihan kota/kabupaten otomatis berisi kota-kota yang ada di DKI Jakarta. Kemudian saat memilih kota **Jakarta Barat**, pilihan kecamatan hanya akan berisi kecamatan yang ada di wilayah Jakarta Barat saja.

Inilah yang saya maksud dengan **dropdown dinamis**. Isi dari tag <select> akan berubah tergantung pilihan tag <select> lainnya.

Agar lebih sederhana, saya akan menggunakan 2 buah tag <select>. Keduanya untuk memilih nama **Fakultas** dan **Jurusan**. Berikut tampilan akhir dari kode program yang akan kita rancang:



Gambar: Pemilihan Fakultas dan Jurusan

- Untuk fakultas, terdapat 3 pilihan: **FASILKOM, Ekonomi, dan FMIPA**.
- Jika fakultas **FASILKOM** dipilih, nama jurusan yang tersedia adalah: Ilmu Komputer, Teknik Informatika dan Sistem Informasi.
- Jika fakultas **Ekonomi** dipilih, nama jurusan yang tersedia adalah: Akuntansi, Manajemen dan Ilmu Ekonomi.
- Jika fakultas **FMIPA** dipilih, nama jurusan yang tersedia adalah: Matematika, Fisika, Kimia dan Biologi.

Selain mengubah tag `<select>`, hasil akhirnya akan tampil sebagai teks bagian bawah, seperti **Fakultas FMIPA, Jurusan Kimia**.

Dapatkankah anda membayangkan bagaimana kode program untuk membuat fitur seperti ini? Semuanya sudah kita pelajari, dan dengan sedikit “keterampilan” dalam merangkai kode program JavaScript, mari kita rancang kode programnya.

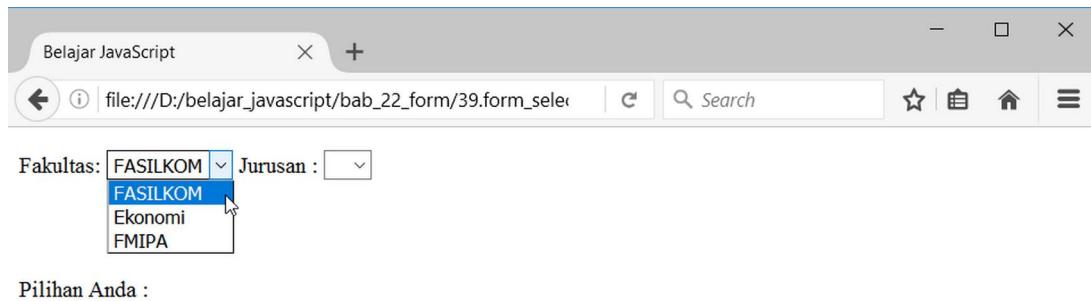
Baik, mulai dengan HTML. Berikut struktur HTML yang dibutuhkan:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6 </head>
7 <body>
8   <p>Fakultas:
9     <select name="fakultas" id="fakultas">
10    <option value="FASILKOM">FASILKOM</option>
11    <option value="Ekonomi">Ekonomi</option>
12    <option value="FMIPA">FMIPA</option>
13  </select>
14  Jurusan :
15  <select name="jurusan" id="jurusan"></select>
16 </p>
17 <br><br>
18 <p> Pilihan Anda : <span id=hasil></span></p>
19 </body>
20 </html>
```

Tidak jauh berbeda seperti contoh-contoh sebelumnya. Saya membuat pilihan **Fakultas** dengan tag `<select name="fakultas" id="fakultas">`. Ketiga fakultas, yakni **FASILKOM, Ekonomi** dan **FMIPA** langsung di tulis dengan tag `<option>`.

Untuk pilihan jurusan, akan digenerate menggunakan JavaScript. Sehingga saya hanya membuat “tempat” yang nantinya akan diisi, yakni `<select name="jurusan" id="jurusan">`.

Terakhir terdapat tag `<span id=hasil>` untuk menampilkan teks akhir. Mari coba jalankan:



Gambar: Tampilan HTML

Tampak sesuai dengan keinginan. Tampilan teks "Pilihan Anda : " saya tempatkan agak kebawah supaya tidak terhalang dari pilihan tag <select>.

Sebelum masuk ke kode Javascript, saya ingin menyampaikan sedikit tips. Jika kode JavaScript yang akan dirancang cukup panjang, sangat disarankan untuk memeriksa hasilnya secara berkala.

Buat beberapa baris, cek dengan `console.log()`. Jika OK, baru lanjut. Setelah beberapa baris, cek lagi dengan `console.log()`, demikian hingga akhir kode program. Ini untuk memastikan kita bisa mengatasi masalah sedini mungkin.

Akan jauh lebih mudah mencari error di 10 baris kode program dibandingkan 100 baris. Dengan memeriksa secara reguler, kita juga menciptakan sebuah "*save point*". Yakni jika terjadi error, bisa ditelusuri itu berasal dari beberapa baris kode terakhir, karena sebelumnya kita sudah periksa dan tidak ada error.

Kembali ke pembuatan **dropdown dinamis**, saya akan masuk ke program JavaScript.

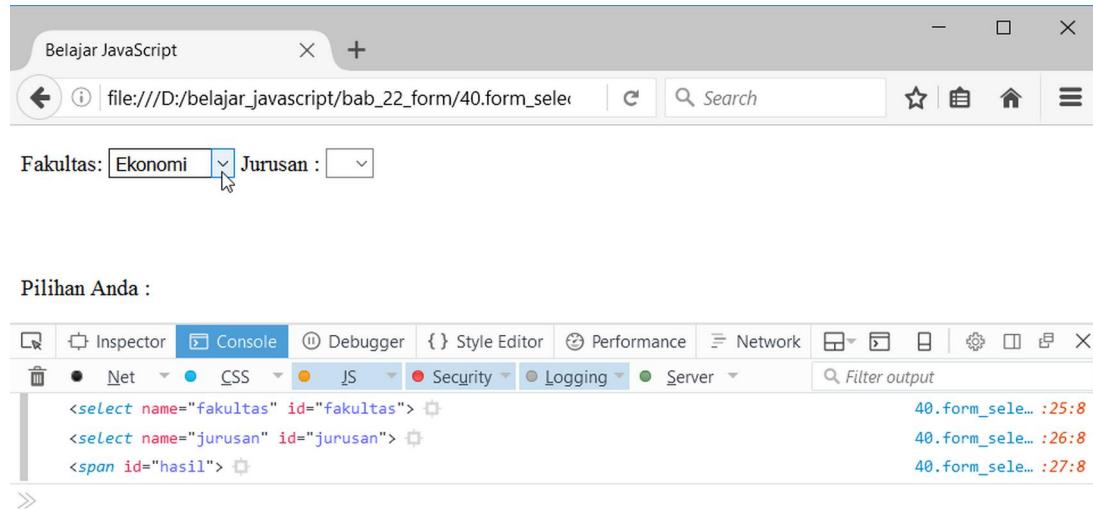
Hal yang pertama ditulis adalah membuat beberapa variabel untuk menampung node object. Kemudian membuat event yang akan dipakai nantinya:

```
1 <script>
2     var fakultasNode = document.getElementById("fakultas");
3     var jurusanNode = document.getElementById("jurusan");
4     var hasilNode = document.getElementById("hasil");
5
6     function tampilkanJurusan(){
7         console.log(fakultasNode);
8         console.log(jurusanNode);
9         console.log(hasilNode);
10    }
11
12    fakultasNode.addEventListener("change",tampilkanJurusan);
13 </script>
```

Variabel `fakultasNode`, `jurusanNode`, dan `hasilNode` digunakan untuk menampung node object dari 3 element: tag <select> **fakultas**, tag <select> **jurusan**, dan tag <span> **hasil**.

Setelah itu ada sebuah function `tampilkanJurusan()`. Isinya menampilkan `console.log()` dari ketiga variabel node object sebelumnya. Function `tampilkanJurusan()` akan dipanggil jika terjadi event `change` dari `fakultasNode`.

Artinya, jika pilihan **Fakultas** di ganti, jalankan function `tampilkanJurusan()`:



Gambar: Tampilan `console.log()` ketika pilihan Jurusan di ganti

Saat saya menukar pilihan **Fakultas**, akan tampil hasil `console.log()`. Inilah yang saya maksud dengan memeriksa hasil secara berkala. Jika sudah tampil seperti diatas, dapat disimpulkan bahwa ketiga variabel sudah berisi node object yang sesuai. Selain itu, event `change` juga sukses berjalan.

Langkah berikutnya adalah membuat kondisi `if` di dalam function `tampilkanJurusan()`:

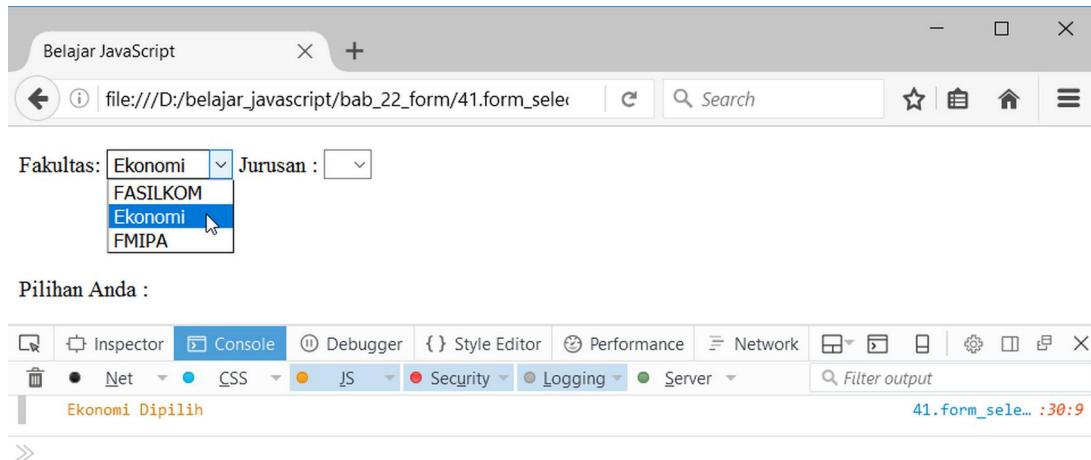
```

1 <script>
2   var fakultasNode = document.getElementById("fakultas");
3   var jurusanNode = document.getElementById("jurusan");
4   var hasilNode = document.getElementById("hasil");
5
6
7   function tampilkanJurusan(){
8     if (fakultasNode.value === "FASILKOM"){
9       console.log("Fasilkom Dipilih");
10    }
11    if (fakultasNode.value === "Ekonomi"){
12      console.log("Ekonomi Dipilih");
13    }
14    if (fakultasNode.value === "FMIPA"){
15      console.log("FMIPA Dipilih");
16    }
17  }
18
19  fakultasNode.addEventListener("change",tampilkanJurusan);

```

20 </script>

Sekarang, jika pilihan **Fakultas** di ganti, akan tampil teks di tab Console:



Gambar: Akan tampil teks sesuai dengan pilihan Fakultas

Kembali saya membuat langkah ini sebagai “save point” dan memastikan kondisi **if** sudah berjalan.

Sampai disini, kita sudah bisa masuk ke pembuatan kode program untuk meng-*generate* isi dari tag <select> **Jurusan**. Bagaimana caranya?

Cara yang paling simple adalah dengan menginput kode HTML yang berisi rangkaian tag <option> ke dalam jurusanNode.innerHTML.

Sebagai contoh, untuk membuat 1 buah pilihan ke dalam jurusanNode, bisa menggunakan kode berikut:

```
jurusanNode.innerHTML = "<option value='Ilkom'>Ilmu Komputer</option>";
```

Cara alternatif (yang lebih panjang) adalah membuat element node menggunakan perintah `document.createElement("option")` dan `document.createTextNode("Ilmu Komputer")`, kemudian `append()` ke dalam jurusanNode. Cara seperti ini kita bahas dalam bab tentang DOM.

Saya memilih cara `innerHTML` karena lebih singkat. Namun perhatikan teks yang akan diinput berupa sebuah string. String ini memiliki pembatas tanda kutip dua (""). Dengan demikian, untuk bagian atribut `value` dari tag <option> saya hanya bisa menggunakan tanda kutip 1 (''). Jika ingin menggunakan tanda kutip dua, saya harus men-escapenya seperti berikut:

```
jurusanNode.innerHTML = "<option value=\"Ilkom\">Ilmu Komputer</option>";
```

Ini diperlukan agar tanda penutup string tidak “bentrok” jika sama-sama menggunakan tanda kutip dua.

Cara diatas hanya menambahkan 1 pilihan option. Bagaimana dengan 3 pilihan? Tidak ada masalah, kita cukup menyambungnya:

```
1 var jurFASILKOM = "";
2
3 jurFASILKOM = "<option value='Ilkom'>Ilmu Komputer</option>";
4 jurFASILKOM += "<option value='TI'>Teknik Informatika</option>";
5 jurFASILKOM += "<option value='SI'>Sistem Informasi</option>";
6 jurusanNode.innerHTML = jurFASILKOM;
```

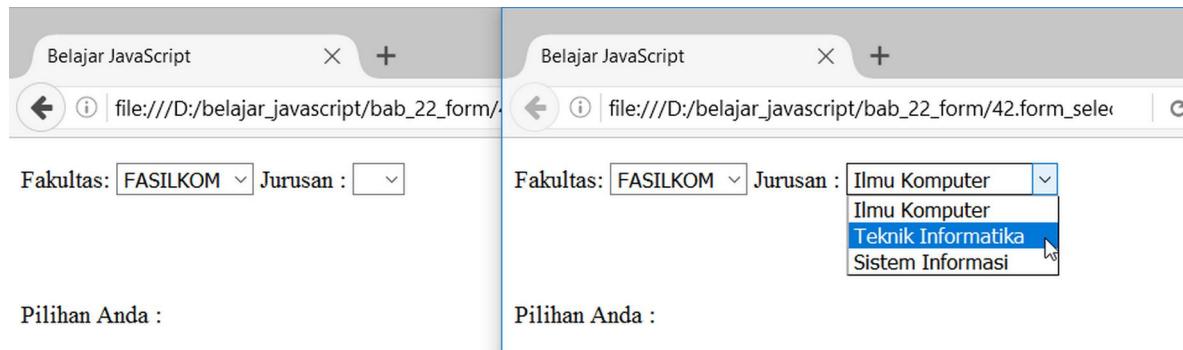
Sekarang, variabel jurFASILKOM akan berisi kode HTML untuk menggenerate tampilan pilihan dropdown 3 buah jurusan dari fakultas **FASILKOM**.

Saya memecah string diatas menjadi 3 baris agar lebih rapi. Tapi anda bisa saja menyambung seluruh string ini menjadi 1 baris panjang.

Kemanakah kode ini akan kita tempatkan? karena ini digunakan untuk menggenerate jurusan **FASILKOM**, kode diatas harusnya berada di dalam kondisi **if** saat isi dari `fakultasNode.value === "FASILKOM"` bernilai **true**.

Berikut hasil gabungan ke dalam kode kita sebelumnya:

```
1 <script>
2     var fakultasNode = document.getElementById("fakultas");
3     var jurusanNode = document.getElementById("jurusan");
4     var hasilNode = document.getElementById("hasil");
5
6     var jurFASILKOM = "";
7
8     function tampilkanJurusan(){
9         if (fakultasNode.value === "FASILKOM"){
10             jurFASILKOM = "<option value='Ilkom'>Ilmu Komputer</option>";
11             jurFASILKOM += "<option value='TI'>Teknik Informatika</option>";
12             jurFASILKOM += "<option value='SI'>Sistem Informasi</option>";
13             jurusanNode.innerHTML = jurFASILKOM;
14         }
15         if (fakultasNode.value === "Ekonomi"){
16             console.log("Ekonomi Dipilih");
17         }
18         if (fakultasNode.value === "FMIPA"){
19             console.log("FMIPA Dipilih");
20         }
21     }
22
23     fakultasNode.addEventListener("change",tampilkanJurusan);
24 </script>
```



Gambar: Tampilan saat file di load pertama kali (kiri), setelah pilihan fakultas ditukar (kanan)

Pada saat file berjalan, pilihan **Fakultas** langsung terpilih **FASILKOM**. Karena secara bawaan web browser akan menggunakan tag `<option>` pertama sebagai pilihan default. Tapi pilihan **Jurusan** tetap kosong (tampilan gambar sebelah kiri).

Agar pilihan **Jurusan** bisa terisi, kita harus pilih fakultas lain terlebih dahulu, kemudian baru pilih kembali fakultas **FASILKOM**. Hasilnya, pilihan **Jurusan** akan terisi (tampilan gambar sebelah kanan).

Dapatkankah anda menjelaskan kenapa kita harus memilih nama fakultas lain terlebih dahulu, baru daftar jurusan **FASILKOM** bisa tampil? Hal ini berkaitan dengan event dimana **Jurusan** akan di generate.

Isi dropdown **Jurusan** hanya akan di-*generate* saat function `tampilkanJurusan()` dipanggil. Kapan function ini dipanggil? Hanya jika terjadi event **change** dari `fakultasNode`. Yakni hanya ketika pilihan jurusan berubah nilai (**change** event).

Solusi dari masalah ini, kita harus memanggil function `tampilkanJurusan()` tepat saat halaman tampil.

Ketika membahas tentang event, saya sempat menyinggung tentang "**Progression events**", yakni event yang dijalankan di dalam "siklus hidup" DOM, seperti saat halaman di tampilkan atau ketika halaman ditutup. Inilah event yang kita perlukan sekarang.

Event tersebut adalah event **load** kepunyaan **window object**. Lebih jauh tentang **window object** akan saya bahas di dalam bab tentang **BOM** (Browser Object Model).

Untuk menjalankan function `tampilkanJurusan()` ketika file tampil pertama kali, penulisan event-nya adalah sebagai berikut:

```
window.addEventListener("load", tampilkanJurusan);
```

Dengan demikian ketika halaman di-load, pilihan Jurusan untuk fakultas **FASILKOM** akan langsung berisi nilai.

Bagaimana dengan nilai Jurusan lain? caranya sama persis. Kita tinggal mengisinya ke dalam kondisi if untuk masing-masing jurusan:

```
1 if (fakultasNode.value === "Ekonomi"){
2     jurEkonomi = "<option value='Akuntansi'>Akuntansi</option>";
3     jurEkonomi += "<option value='Manajemen'>Manajemen</option>";
4     jurEkonomi += "<option value='Ilmu Ekonomi'>Ilmu Ekonomi</option>";
5     jurusanNode.innerHTML = jurEkonomi;
6 }
7 if (fakultasNode.value === "FMIPA"){
8     jurFMIPA = "<option value='Matematika'>Matematika</option>";
9     jurFMIPA += "<option value='Fisika'>Fisika</option>";
10    jurFMIPA += "<option value='Kimia'>Kimia</option>";
11    jurFMIPA += "<option value='Biologi'>Biologi</option>";
12    jurusanNode.innerHTML = jurFMIPA;
13 }
```

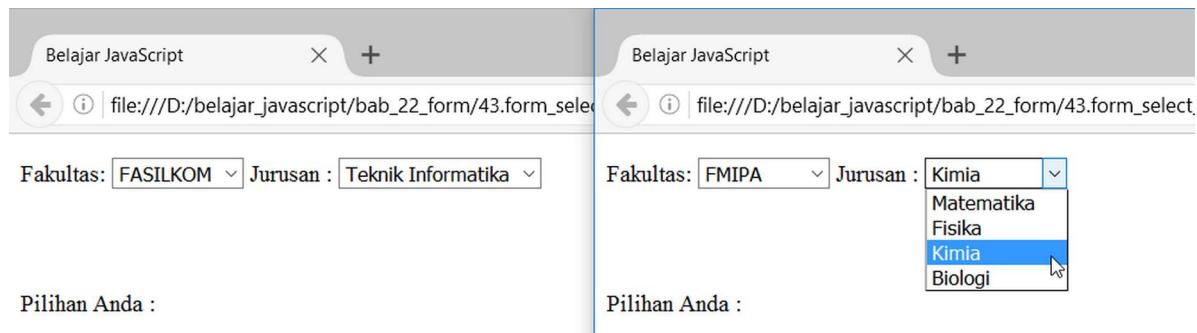
Berikut kode lengkap untuk menggenerate ketiga Fakultas:

```
1 <script>
2     var fakultasNode = document.getElementById("fakultas");
3     var jurusanNode = document.getElementById("jurusan");
4     var hasilNode = document.getElementById("hasil");
5
6     var jurFASILKOM = "";
7     var jurEkonomi = "";
8     var jurFMIPA = "";
9
10    function tampilkanJurusan(){
11        if (fakultasNode.value === "FASILKOM"){
12            jurFASILKOM = "<option value='Ilkom'>Ilmu Komputer</option>";
13            jurFASILKOM += "<option value='TI'>Teknik Informatika</option>";
14            jurFASILKOM += "<option value='SI'>Sistem Informasi</option>";
15            jurusanNode.innerHTML = jurFASILKOM;
16        }
17        if (fakultasNode.value === "Ekonomi"){
18            jurEkonomi = "<option value='Akuntansi'>Akuntansi</option>";
19            jurEkonomi += "<option value='Manajemen'>Manajemen</option>";
20            jurEkonomi += "<option value='Ilmu Ekonomi'>Ilmu Ekonomi</option>";
21            jurusanNode.innerHTML = jurEkonomi;
22        }
23        if (fakultasNode.value === "FMIPA"){
24            jurFMIPA = "<option value='Matematika'>Matematika</option>";
25            jurFMIPA += "<option value='Fisika'>Fisika</option>";
26            jurFMIPA += "<option value='Kimia'>Kimia</option>";
27            jurFMIPA += "<option value='Biologi'>Biologi</option>";
28            jurusanNode.innerHTML = jurFMIPA;
29        }
30    }
```

```

31
32     window.addEventListener("load",tampilkanJurusan);
33     fakultasNode.addEventListener("change",tampilkanJurusan);
34 </script>

```



Gambar: Seluruh jurusan sudah di-generate

Sampai disini kita sudah selesai membuat **dropdown dinamis**.

Terakhir adalah menampilkan teks berdasarkan pilihan. Ini bisa dibuat dengan cara memeriksa nilai dari `fakultasNode.value` dan `jurusanNode.value`, kemudian diinput ke dalam `hasilNode.innerHTML`.

Saya memutuskan untuk membuat function khusus, yakni `tampilkanHasil()`. Isinya adalah sebagai berikut:

```

1 function tampilkanHasil(){
2     hasilNode.innerHTML = "Fakultas " + fakultasNode.value + ", ";
3     hasilNode.innerHTML += "Jurusan " + jurusanNode.value;
4 }

```

Kapan function ini dipanggil? Saya ingin teks tampil begitu pilihan Fakultas dan Jurusan berubah, artinya kita perlu memanggil function ini untuk event **change** dari `fakultasNode` dan `jurusanNode`:

```

1 fakultasNode.addEventListener("change",tampilkanHasil);
2 jurusanNode.addEventListener("change",tampilkanHasil);

```

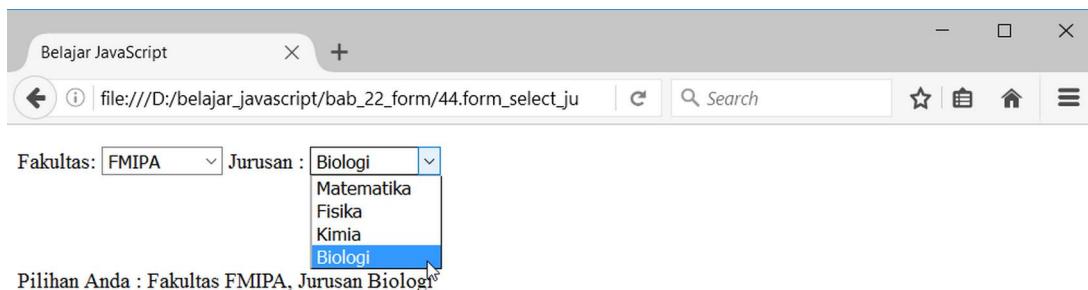
Akan tetapi karena event **change** untuk `fakultasNode` sudah tersedia, yakni untuk menjalankan function `tampilkanJurusan()`, saya bisa menyisipkan pemanggilan function `tampilkanHasil()` ke dalam `tampilkanJurusan()`.

Selain itu, karena function `tampilkanJurusan()` juga akan dipanggil saat halaman ditampilkan (berasal dari event **load** dari **window object**), maka teks juga akan otomatis tampil begitu halaman di-load.

Akhirnya, berikut kode program lengkap **dropdown dinamis** untuk pemilihan Fakultas dan Jurusan:

```
1 <body>
2   <p>Fakultas:
3     <select name="fakultas" id="fakultas">
4       <option value="FASILKOM">FASILKOM</option>
5       <option value="Ekonomi">Ekonomi</option>
6       <option value="FMIPA">FMIPA</option>
7     </select>
8   Jurusan :
9     <select name="jurusan" id="jurusan"></select>
10  </p>
11  <br><br>
12  <p> Pilihan Anda : <span id=hasil></span></p>
13  <script>
14    var fakultasNode = document.getElementById("fakultas");
15    var jurusanNode = document.getElementById("jurusan");
16    var hasilNode = document.getElementById("hasil");
17
18    var jurFASILKOM = "";
19    var jurEkonomi = "";
20    var jurFMIPA = "";
21
22    function tampilkanJurusan(){
23      if (fakultasNode.value === "FASILKOM"){
24        jurFASILKOM = "<option value='Ilkom'>Ilmu Komputer</option>";
25        jurFASILKOM += "<option value='TI'>Teknik Informatika</option>";
26        jurFASILKOM += "<option value='SI'>Sistem Informasi</option>";
27        jurusanNode.innerHTML = jurFASILKOM;
28      }
29      if (fakultasNode.value === "Ekonomi"){
30        jurEkonomi = "<option value='Akuntansi'>Akuntansi</option>";
31        jurEkonomi += "<option value='Manajemen'>Manajemen</option>";
32        jurEkonomi += "<option value='Ilmu Ekonomi'>Ilmu Ekonomi</option>";
33        jurusanNode.innerHTML = jurEkonomi;
34      }
35      if (fakultasNode.value === "FMIPA"){
36        jurFMIPA = "<option value='Matematika'>Matematika</option>";
37        jurFMIPA += "<option value='Fisika'>Fisika</option>";
38        jurFMIPA += "<option value='Kimia'>Kimia</option>";
39        jurFMIPA += "<option value='Biologi'>Biologi</option>";
40        jurusanNode.innerHTML = jurFMIPA;
41      }
42      tampilkanHasil();
43    }
44
45    function tampilkanHasil(){
46      hasilNode.innerHTML = "Fakultas " + fakultasNode.value + ", ";
```

```
47     hasilNode.innerHTML += "Jurusan " + jurusanNode.value;
48 }
49
50 window.addEventListener("load",tampilkanJurusan);
51 fakultasNode.addEventListener("change",tampilkanJurusan);
52 jurusanNode.addEventListener("change",tampilkanHasil);
53 </script>
54 </body>
```



Gambar: Tampilan akhir program dropdown dinamis

Berbekal skill seperti ini, saya sarankan anda untuk mencoba berbagai kombinasi lain dalam memanipulasi form, misalkan jika sebuah tombol checkbox di pilih, tampilkan element form baru. Atau membuat fitur yang mirip seperti yang kita buat diatas tetapi menggunakan radio button. Jika pilihan diubah, tampilan radio button lain juga akan berubah. Silahkan berkreasi! :)

## 22.15 Form Validation

Penggunaan terbesar JavaScript di dalam form adalah untuk memeriksa apakah isian form sudah sesuai dengan syarat yang ditetapkan atau belum. Misalnya apakah kolom username sudah diisi, apakah password sudah terdiri dari kombinasi angka dan huruf, atau apakah alamat email sudah valid. Ini semua dikenal sebagai **form validation**.

Prinsip dari **form validation** adalah, ambil atribut `value` dari sebuah element, kemudian bandingkan dengan syarat. Jika tidak memenuhi, tampilkan pesan error dan jalankan `preventDefault()` untuk form element. Perintah `preventDefault()` diperlukan agar form tidak sampai di kirim ke server.

Kode program yang digunakan untuk **form validation** sudah kita pelajari semua dan tidak ada materi baru, hanya saja perlu sebuah cara dalam “merangkai” berbagai kode JavaScript + DOM untuk men-validasi sebuah form. Inilah yang akan segera kita bahas.

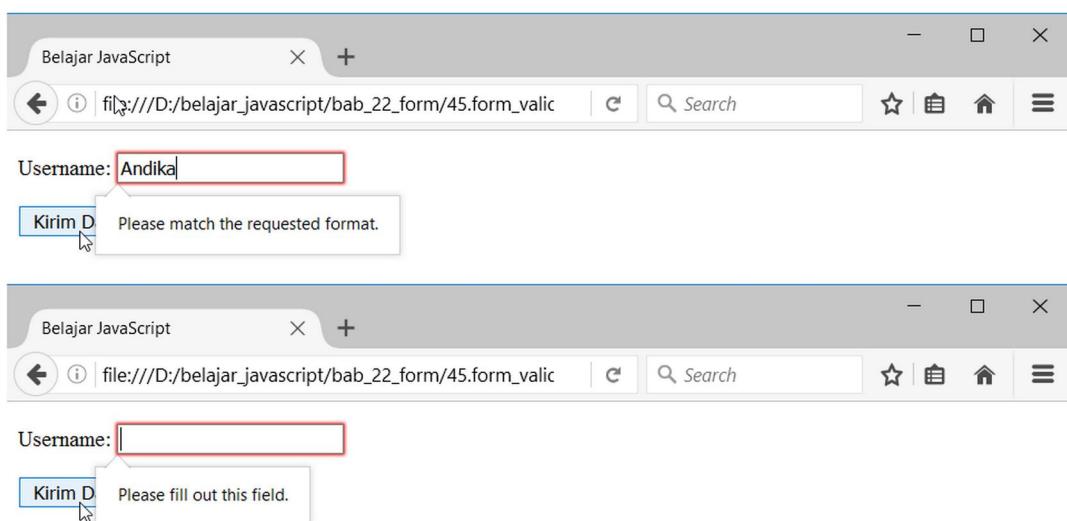
### Form Validation HTML5

Sebelum masuk ke form validation dengan JavaScript, sebenarnya form validation juga bisa dibuat dengan vitur terbaru dari HTML5.

Jika anda sudah membaca buku **HTML Uncover**, tentunya sudah kenal dengan fitur ini. Saya membuat 1 bab khusus tentang **Form HTML5**, termasuk atribut yang bisa digunakan untuk membuat validasi.

Berikut contohnya:

```
1 <body>
2   <form id="formKu" name="formKu" method="get" action="proses.php">
3
4     <p>Username: <input type="text" name="username" id="username"
5       required pattern="[A-Z]{3}[0-9]{3}"></p>
6
7     <p><input type="submit" name="submit" id="submit" value="Kirim Data"></p>
8
9   </form>
10 </body>
```

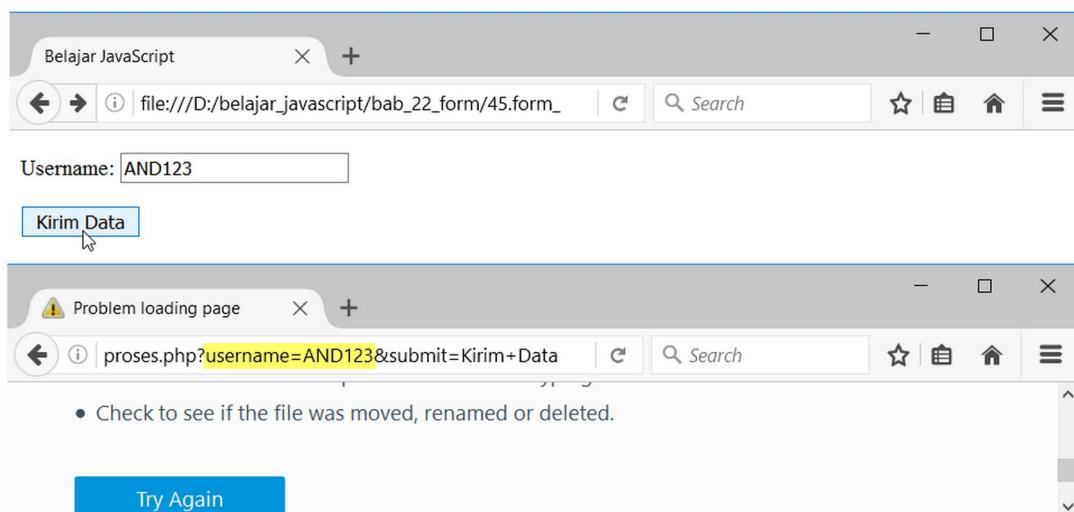


Gambar: Fitur validasi bawaan HTML5

Dalam kode HTML diatas, saya menggunakan fitur validasi bawaan HTML5. Pertama, terdapat atribut `required`. Atribut ini akan menampilkan pesan error jika textbox `username` tidak diisi (kosong).

Kedua, terdapat atribut `pattern="[A-Z]{3}[0-9]{3}"`. Ini merupakan atribut untuk menginput kode **Regular Expression**. Artinya, inputan `username` harus sesuai dengan format *RegEx* yang ditulis.

Dalam contoh diatas, `username` harus terdiri dari 3 digit huruf besar dan diikuti 3 digit angka. Jika tidak sesuai, akan tampil pesan error. String "AND123" cocok dengan pola *RegEx* dan form akan dikirim ke server:



Gambar: Inputan sudah sesuai dan dikirim ke server

Walaupun sudah lolos validasi, hasilnya tetap error karena halaman proses.php tidak tersedia. Namun kita bisa melihat data yang kirim di bagian alamat address bar web browser (dari query string). Jika tampil seperti ini, artinya form sudah dikirim ke server.

Jika anda butuh membuat validasi dengan cepat, atribut HTML5 seperti ini bisa digunakan.

## Validasi dengan JavaScript

Sebelum era HTML5, satu-satunya cara membuat validasi di sisi web browser (client) adalah menggunakan JavaScript. Meskipun lebih ribet dan butuh kode program yang lumayan panjang, validasi yang dirancang dengan JavaScript lebih fleksibel. Selain itu kita juga bisa menentukan dimana pesan error akan tampil.

Seperti yang sudah disinggung sebelumnya, prinsip dari **form validation** adalah ambil atribut value dari setiap form element, kemudian bandingkan dengan syarat. Jika tidak memenuhi, tampilkan pesan error dan jalankan preventDefault(). Prinsip seperti akan kita gunakan sepanjang pembuatan proses validasi.

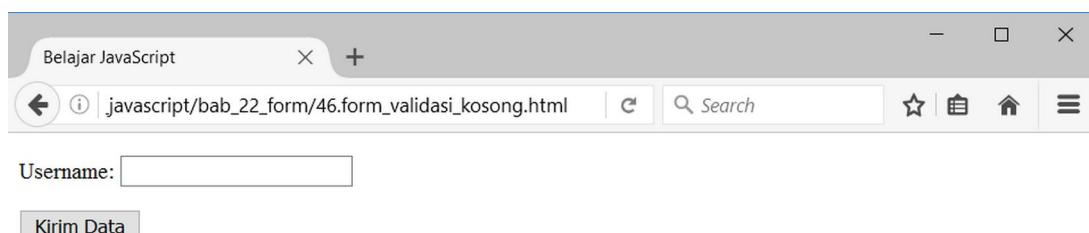
Sebagai ‘template’ dari praktik yang akan dibahas, saya menyiapkan kode HTML untuk form sederhana. Form ini hanya terdiri dari 1 buat text box username dan sebuah tombol submit:

```
1 <body>
2   <p id="pesan"></p>
3   <form id="formKu" name="formKu" method="get" action="proses.php">
4     <p>Username: <input type="text" name="username" id="username"></p>
5     <p><input type="submit" name="submit" id="submit" value="Kirim Data"></p>
6   </form>
7   <script>
8     // kode JavaScript disini...
9   </script>
10 </body>
```

Di bagian atas, saya membuat sebuah tag `<p id="pesan">`. Disinilah nantinya pesan error akan ditampilkan. Untuk bagian form, terdapat 3 element: `<form id="formKu">`, `<input type="text" id="username">`, dan `<input type="submit" name="submit">`.

Mari kita mulai. Saya akan merancang sebuah kode program untuk mengecek apakah text box username sudah diisi atau tidak:

```
1 <script>
2   var formKuNode = document.getElementById("formKu");
3   var usernameNode = document.getElementById("username");
4   var pesanNode = document.getElementById("pesan");
5
6   function diProses(e){
7     if (usernameNode.value === ""){
8       pesanNode.innerHTML = "Username harus diisi";
9       e.preventDefault();
10    }
11  }
12  formKuNode.addEventListener("submit",diProses);
13 </script>
```



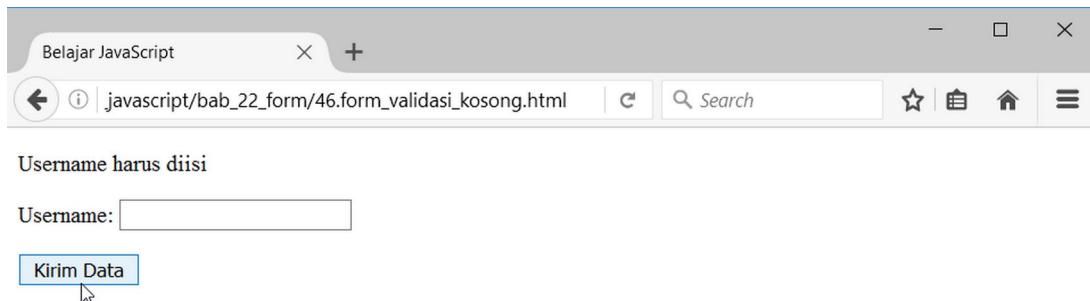
Gambar: Form untuk menginput sebuah username

Di dalam JavaScript sama membuat 3 buah variabel untuk menampung node object. Tag `<p id="pesan">` di simpan ke dalam variabel pesanNode. Tag `<form id="formKu">` disimpan ke dalam formKuNode, dan tag `<input type="text" id="username">` ke dalam usernameNode.

Saat form di submit, yakni ketika tombol **submit** di klik, jalankan function `diProses()`.

Pertama saya membuat sebuah kondisi `if (usernameNode.value === "")`. Kondisi ini akan bernilai **true** jika text box usernameNode tidak berisi teks sama sekali. Jika ini yang terjadi, tampilkan pesan error dengan perintah `pesanNode.innerHTML = "Username harus diisi"`. Kemudian "tahan" form agar tidak terkirim ke server menggunakan perintah `e.preventDefault()`.

Berikut percobaannya:



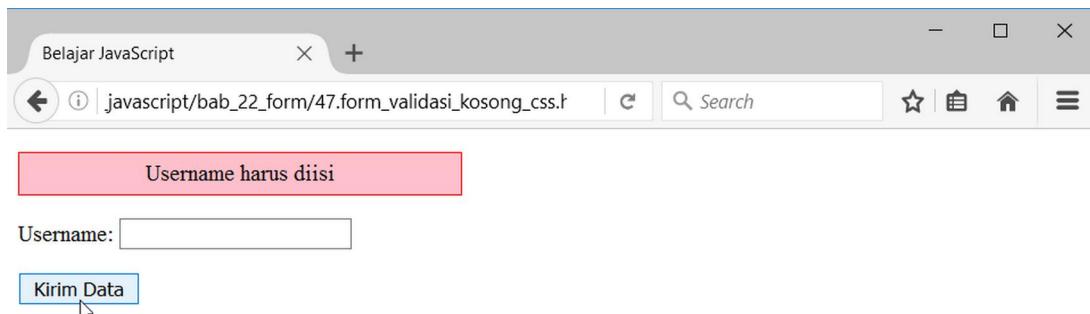
Gambar: Pesan error tampil di bagian atas halaman

Ketika text box username tidak diisi teks apapun atau tombol submit di klik begitu halaman selesai di load, proses validasi akan dijalankan. Hasilnya terlihat pesan error "Username harus diisi" di bagian atas form. Namun jika sebuah teks apa saja diinput ke dalam text box username, form akan ter-submit ke web server.

Alur seperti ini akan terus saya gunakan selama pembahasan **form validation**. Agar menghemat tempat, dalam contoh-contoh selanjutnya saya akan fokus kepada isi dari function `diProses()` saja. Jika tidak ditulis, bagian lain dari kode program HTML dan JavaScript dianggap tidak ada perubahan.

## Menambahkan Style CSS

Pesan error yang tampil sudah cukup informatif, tapi terkesan kurang menarik. Bagaimana dengan sedikit kode CSS? Saya ingin pesan error tampil di dalam sebuah box berwarna merah, seperti tampilan berikut:



Gambar: Pesan error dengan style CSS

Terdapat berbagai cara untuk membuat tampilan seperti ini. Yang paling sederhana adalah, siapkan sebuah **class** di dalam kode CSS, kemudian jika terjadi error, tambahkan class tersebut ke dalam pesanNode.

Untuk kode class CSS, saya membuat kode program berikut:

```
1 <style>
2     .error {
3         background-color: pink;
4         border: 1px solid red;
5         width: 300px;
6         text-align: center;
7         padding: 5px;
8     }
9 </style>
```

Class `.error` berisi berbagai property CSS untuk membuat warna background menjadi pink, dengan border merah setebal 1 pixel, lebar element 300 pixel, teks berada di tengah, dan padding sebesar 5 pixel.

Selanjutnya, saya akan *menempelkan* class `.error` ini ke dalam tag `<p id="pesan">` ketika error terjadi. Caranya adalah, di dalam function `diProses()` tambahkan perintah `pesanNode.className = "error"` seperti berikut ini:

```
1 function diProses(e){
2     if (usernameNode.value === ""){
3         pesanNode.innerHTML = "Username harus diisi";
4         pesanNode.className = "error";
5         e.preventDefault();
6     }
7 }
```

Sekarang, ketika function `diProses()` berjalan, otomatis tag `<p id="pesan">` akan memiliki class CSS: `.error`.

## Fitur Validasi: Trim

Dengan tampilan pesan error yang lebih rapi, kita bisa kembali membahas tentang validasi.

Terdapat 1 kelemahan dari kode program sebelumnya. Jika text box `username` diinput dengan karakter *whitespace* seperti spasi, validasi akan lolos. Alasannya, karena spasi atau `tab` tetap dianggap sebagai sebuah karakter.

Bagaimana cara mengatasi hal ini? kita bisa meminta bantuan kepada method `trim()` dari **String object**:

```
1 function diProses(e){  
2     if (usernameNode.value.trim() === ""){  
3         pesanNode.innerHTML = "Username harus diisi";  
4         pesanNode.className = "error";  
5         e.preventDefault();  
6     }  
7 }
```

Perhatikan di bagian kondisi, sekarang menjadi `if (usernameNode.value.trim() === "")`. Artinya, sebelum dibandingkan dengan string kosong, string hasil `usernameNode.value` akan dilewatkan dulu ke dalam method `trim()`. Tujuannya untuk membuang seluruh karakter whitespace yang ada di dalam text box `username`.

Dengan demikian, form tidak akan lolos jika yang diinput hanya terdiri dari karakter spasi atau tab.



Lebih lanjut tentang penggunaan method `trim()`, telah saya bahas di dalam bab tentang **String Object**, yakni bagian **Method String.prototype.trim()**.

## Fitur Validasi: Regular Expression

Untuk membuat syarat validasi yang cukup rumit, tidak ada yang bisa mengalahkan apa yang ditawarkan oleh **Regular Expression**.

Dengan menggunakan **RegExp**, kita bisa merancang hampir segala bentuk validasi. Misalnya saya ingin `username` hanya bisa diinput dengan karakter angka saja, atau hanya karakter alfanumerik.

Juga bisa untuk membuat pola yang lebih rumit, misalnya `username` harus terdiri dari 6 digit karakter, dimana 3 digit karakter pertama harus berbentuk huruf besar dan 3 karakter terakhir berupa angka. Semua ini bisa ditangani oleh **RegExp**.

Sebagai contoh, saya ingin membatasi isian `username` hanya boleh diisi dengan karakter alfanumerik saja, yakni hanya bisa terdiri dari huruf + angka, tidak boleh terdapat karakter lain. Bagaimana caranya?

Pertama, kita harus merancang kode **RegExp** yang diperlukan. Pembahasan mengenai **RegExp** sudah saya bahas di bab tentang **Regular Expression Object**. Anda bisa melihat sebentar jika lupa-lupa ingat tentang cara membuat **RegExp Object** di dalam JavaScript.

Untuk membuat pola alfanumerik, kode **RegExp** yang diperlukan sangat simple, yakni: `/\w/`. Pola `/\w/` cocok dengan seluruh karakter alfabet dan angka (alfanumerik) serta karakter underscore: `_`. Ini sama artinya dengan pola `[A-Za-zA-Z_0-9_]`.

Sebelum menginputnya ke dalam function `diProses()`. Mari kita lakukan sedikit percobaan dengan pola regular expression ini:

```

1 var pola = /\w/;
2 console.log ( pola.test("aaa") );           // true
3 console.log ( pola.test("123") );           // true
4 console.log ( pola.test("ad43") );          // true
5 console.log ( pola.test("AAA! ") );         // true
6 console.log ( pola.test("@aku") );          // true
7 console.log ( pola.test("zzzzzzz.") );      // true

```

Loh, kenapa hasilnya **true** semua? Padahal untuk percobaan 4-6 terdapat karakter non-alfanumerik, yakni karakter !, @ dan tanda titik (.).

Inilah prinsip dari **RegExp**, pola `/\w/` akan cocok dengan string apapun selama di dalamnya terdapat alfanumerik. Tidak masalah jika ditemukan karakter lain, selama ada karakter alfanumerik, function `test()` akan menghasilkan nilai **true**.

Jadi bagaimana jika yang kita ingin memastikan **tidak ada** karakter alfanumerik di dalam `username`? Kita harus membalik proses logika. Carilah pola regular expression untuk karakter **selain** alfanumerik. Jika pola tersebut menghasilkan nilai **true**, berarti ada sebuah karakter **non-alfanumerik** di dalam string tersebut.

**RegExp** memiliki karakter spesial `\W` yang artinya cocok dengan seluruh huruf **selain** alfabet dan angka (alfanumerik) serta karakter underscore: `_`. Ini sama artinya dengan pola `[^A-Za-z0-9_]`.

Mari kita coba:

```

1 var pola = /\W/;
2 console.log ( pola.test("aaa") );           // false
3 console.log ( pola.test("123") );           // false
4 console.log ( pola.test("ad43") );          // false
5 console.log ( pola.test("AAA! ") );         // true
6 console.log ( pola.test("@aku") );          // true
7 console.log ( pola.test("zzzzzzz.") );      // true

```

Inilah yang kita cari! Sekarang, jika string yang ada berisi karakter non-alfanumerik, hasilnya adalah **true** (string ke-4, ke-5 dan ke-6). Dengan demikian, kondisi atau syarat untuk validasi form bisa ditulis seperti ini:

```

1 if (/^\W/.test(usernameNode.value)){
2   pesanNode.innerHTML = "Hanya bisa diisi karakter alfanumerik";
3   pesanNode.className = "error";
4   e.preventDefault();
5 }

```

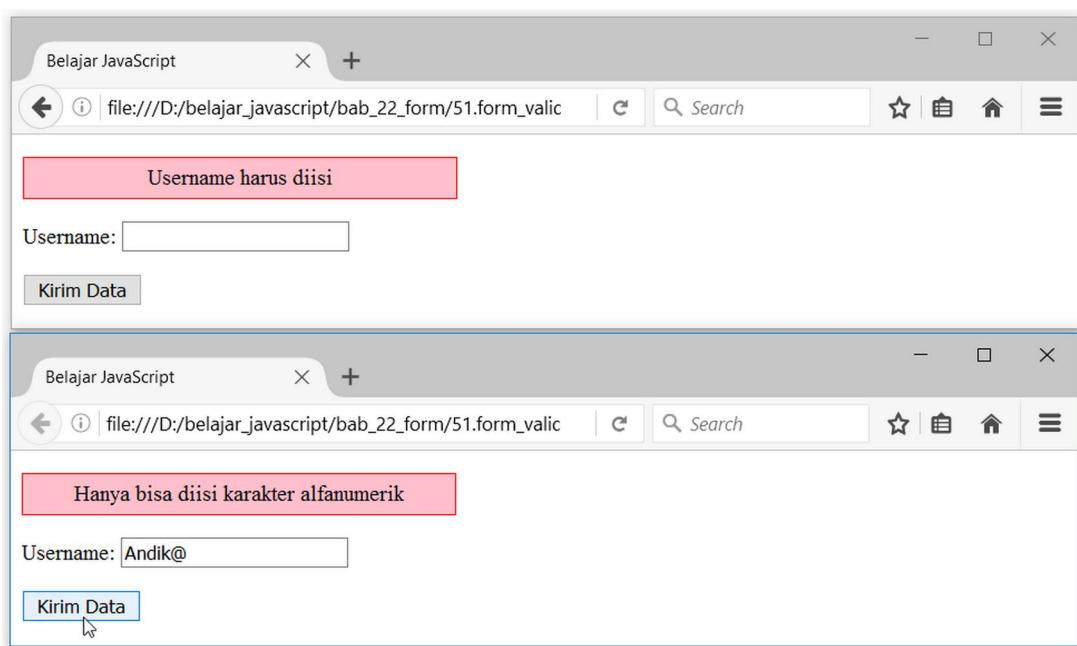
Artinya, jika `usernameNode.value` memiliki sebuah karakter non-alfanumerik, kondisi if diatas bernilai **true**. Namun jika `usernameNode.value` hanya berisi karakter alfanumerik, kondisi diatas akan bernilai **false** dan error tidak akan tampil.

Berikut penambahan Regular Expression diatas ke dalam function `diProses()`:

```

1 function diProses(e){
2     if (usernameNode.value.trim() === ""){
3         pesanNode.innerHTML = "Username harus diisi";
4         pesanNode.className = "error";
5         e.preventDefault();
6     }
7     else if (/^\W/.test(usernameNode.value.trim())){
8         pesanNode.innerHTML = "Hanya bisa diisi karakter alfanumerik";
9         pesanNode.className = "error";
10        e.preventDefault();
11    }
12 }

```



Gambar: Pesan error akan tampil jika username kosong atau jika berisi karakter non-alfanumerik

Dengan kode program diatas, saya sudah memiliki 2 proses validasi: cek apakah username kosong dan cek apakah username memiliki karakter lain diluar karakter alfanumerik. Kedua validasi ini saya gabung dengan perintah **if else**, artinya validasi regular expression hanya akan berjalan jika username tidak kosong.

Saya juga menggunakan `usernameNode.value.trim()` sebagai input untuk method `/^\W/.test()`, karena bisa saja pengguna tidak sengaja menambahkan karakter spasi di akhir username. Karakter spasi akan menyebabkan hasil test **Regular Expression** bernilai **true**, karena karakter spasi termasuk karakter non-alfanumerik.

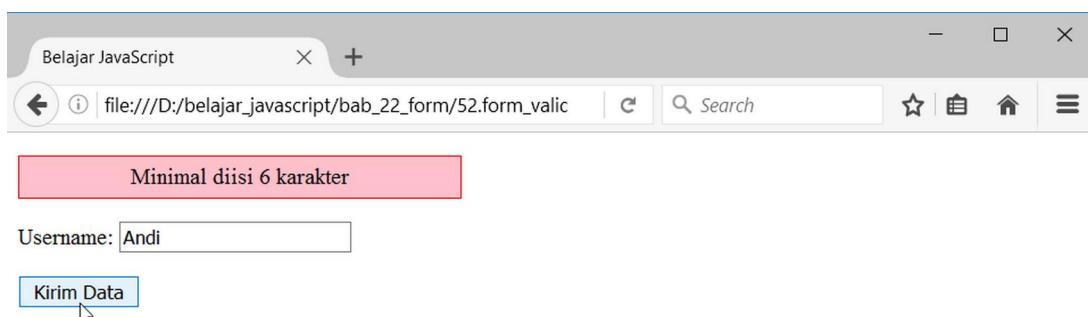
## Fitur Validasi: Panjang Karakter

Sampai disini saya yakin anda sudah bisa melihat pola penambahan fitur validasi. Jika butuh syarat lain, saya cukup menambahkan kondisi **if else** lainnya. Misalkan saya ingin membatasi panjang karakter username minimal 6 karakter, jika kurang tampilkan pesan error.

Bagaimana cara menghitung panjang karakter? Kita bisa mengakses property `length` dari `String object`. Berikut penambahannya ke dalam function `diProses()`:

```

1  function diProses(e){
2      if (usernameNode.value.trim() === ""){
3          pesanNode.innerHTML = "Username harus diisi";
4          pesanNode.className = "error";
5          e.preventDefault();
6      }
7      else if (/^\w/.test(usernameNode.value.trim())){
8          pesanNode.innerHTML = "Hanya bisa diisi karakter alfanumerik";
9          pesanNode.className = "error";
10         e.preventDefault();
11     }
12     else if (usernameNode.value.trim().length < 6 ){
13         pesanNode.innerHTML = "Minimal diisi 6 karakter";
14         pesanNode.className = "error";
15         e.preventDefault();
16     }
17 }
```



Gambar: Pesan error karena username diisi kurang dari 6 karakter

Kondisi `if (usernameNode.value.trim().length < 6 )` akan bernilai `true` jika username kurang dari 6 digit. Jika ini terjadi, tampilkan pesan error "Minimal diisi 6 karakter".

## Refactoring Kode Program

Jika anda sudah berpengalaman dalam membuat berbagai program, dapat dilihat bahwa kita sebaiknya memperbaiki struktur kode program validasi.

Alasannya, baris `pesanNode.className = "error"` dan `e.preventDefault()` selalu ditulis untuk setiap kondisi validasi. Akan lebih bagus jika 2 baris ini kita keluarkan dan menjadi sebuah kode "global".

Proses memperbaiki kode program tanpa mempengaruhi hasil akhir dikenal dengan sebutan: **refactoring**. Alasan untuk melalukan *refactoring* bisa bermacam-macam. Boleh untuk

menyederhanakan penulisan, mengurangi kode program yang *redundant* (ditulis berulang kali), atau agar kode program kita menjadi lebih fleksibel.

Alasan saya untuk melakukan proses **refactoring** terhadap function `diProses()` mencakup semua hal diatas. Tips singkatnya, jika anda sudah membuat baris program yang sama berulang kali, sudah saatnya melakukan *refactoring*.

Untuk function `diProses()` akan saya tulis ulang sebagai berikut:

```
1 function diProses(e){  
2     var usernameError="";  
3  
4     if (usernameNode.value.trim() === ""){  
5         usernameError = "Username harus diisi";  
6     }  
7     else if (/^\W/.test(usernameNode.value.trim()) ){  
8         usernameError = "Hanya bisa diisi karakter alfanumerik";  
9     }  
10    else if (usernameNode.value.trim().length < 6 ){  
11        usernameError = "Minimal diisi 6 karakter";  
12    }  
13  
14    if (usernameError != ""){  
15        pesanNode.innerHTML = usernameError;  
16        pesanNode.className = "error";  
17        e.preventDefault();  
18    }  
19 }
```

Silahkan anda pelajari sebentar kode program diatas.

Di awal function `diProses()`, saya membuat sebuah variabel `usernameError`. Variabel inilah yang akan diisi dengan berbagai pesan error sesuai kondisi validasi. Di akhir function, variabel `usernameError` akan di periksa. Jika berisi sesuatu (yang artinya terdapat error), jalankan 3 perintah:

```
1 pesanNode.innerHTML = usernameError;  
2 pesanNode.className = "error";  
3 e.preventDefault();
```

Dengan seperti ini, kode program validasi menjadi lebih fleksibel. Jika saya butuh menambah 1 lagi kondisi validasi, hanya perlu menambah 2 baris: kondisi yang harus dipenuhi, dan sebuah pesan error.

## Memindahkan Pesan Error

Tidak ada yang salah dengan menampilkan pesan error dibagian atas halaman. Tapi jika element form yang diperiksa cukup banyak, pesan error yang ada bisa “mendorong” form ke arah bawah. Oleh karena itu saya ingin memindahkan pesan error ke samping text box username, seperti tampilan berikut:



Gambar: Pesan error berada di samping text box username

Selain pesan error yang berpindah, terdapat efek border merah di dalam textbox. Untuk membuat tampilan sepertiini, kita harus merombak sedikit struktur HTML dan kode CSS:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <style>
7     .error {
8       color: red;
9       width: 300px;
10      text-align: center;
11      padding: 2px 10px;
12      margin-left: 10px;
13    }
14   </style>
15 </head>
16 <body>
17   <form id="formKu" name="formKu" method="get" action="proses.php">
18     <p>Username: <input type="text" name="username" id="username">
19     <span id="usernameSpan"></span></p>
20     <p><input type="submit" name="submit" id="submit" value="Kirim Data"></p>
21   </form>
22   <script>
23     var formKuNode = document.getElementById("formKu");
24     var usernameNode = document.getElementById("username");
25     var usernameSpanNode = document.getElementById("usernameSpan");
26
27     function diProses(e){
28       var usernameError="";

```

```
29
30     if (usernameNode.value.trim() === ""){
31         usernameError = "Username harus diisi";
32     }
33     else if (/^\W/.test(usernameNode.value.trim())){
34         usernameError = "Hanya bisa diisi karakter alfanumerik";
35     }
36     else if (usernameNode.value.trim().length < 6 ){
37         usernameError = "Minimal diisi 6 karakter";
38     }
39
40     if (usernameError !== ""){
41         usernameSpanNode.innerHTML = usernameError;
42         usernameSpanNode.className = "error";
43         usernameNode.style.border = "2px solid red";
44         e.preventDefault();
45     }
46 }
47
48 formKuNode.addEventListener("submit",diProses);
49 </script>
50 </body>
51 </html>
```

Pertama, saya membuat sebuah tag `<span id="usernameSpan">` disamping tag `<input id="username">`. Disinilah nantinya pesan error yang berkaitan dengan text box username ditampilkan. Kode CSS untuk class `.error` juga terdapat sedikit perubahan.

Di dalam kode JavaScript, saya membuat variabel `usernameSpanNode` untuk menampung node object dari `<span id="usernameSpan">`. Pesan error nantinya akan diinput ke `usernameSpanNode.innerHTML`.

Jika terdapat error, saya juga menambahkan perintah `usernameNode.style.border = "2px solid red"`. Inilah yang menambahkan efek border merah ke dalam tag `<input id="username">`.

## Efek Event Focus

Untuk menambah kesan *user friendly*, saya ingin ketika user mulai memperbaiki text box yang error, warna border akan kembali normal dan teks error hilang untuk sementara.



Gambar: Pesan error hilang saat user mulai mengedit text box

Efek seperti ini bisa didapat dengan menggunakan event **focus**. Caranya, ketika `usernameNode` mengalami event **focus**, hapus semua efek error.

Langkah pertama, kita siapkan sebuah event handler:

```
1 usernameNode.addEventListener("focus",hapusError);
```

Dengan kode program ini, ketika `usernameNode` mengalami event **focus**, jalankan function `hapusError()`. Apa isi dari function ini? Isinya berupa kode program untuk menghapus efek error:

```
1 function hapusError(e){  
2     e.target.style.border = "";  
3     e.target.parentElement.lastChild.innerHTML = "";  
4 }
```

Baris pertama: `e.target.style.border = ""` digunakan untuk menghapus warna border merah dari `usernameNode`.

Baris kedua: `e.target.parentElement.lastChild.innerHTML = ""` digunakan untuk menghapus pesan error.

Perhatikan bahwa saya menggunakan penulisan struktur DOM `e.target.parentElement.lastChild` untuk mencari tag `<span id="usernameSpan">`. Kenapa tidak langsung menggunakan `usernameSpanNode` saja? Karena saya ingin function `hapusError()` juga berlaku untuk object lain, tidak hanya `usernameNode` saja.

Tentunya struktur seperti ini hanya berlaku selama saya membuat struktur yang sama untuk element form lain, yakni tag `<span>` yang menampung pesan error harus ditulis tepat setelah tag `<input>` dan berada di dalam parent element tag `<p>`, seperti:

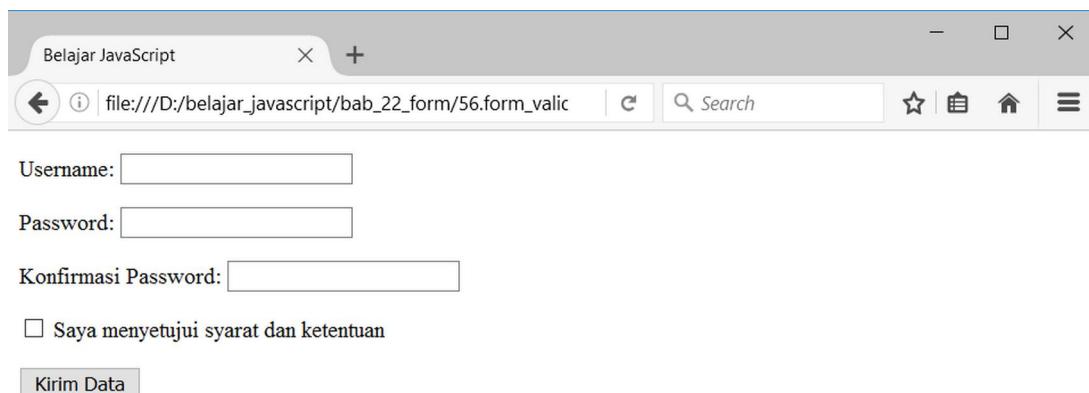
```
1 <p>Username: <input type="text" name="username" id="username">
2 <span id="usernameSpan"></span></p>
3 <p>Password: <input type="password" name="pass" id="pass">
4 <span id="passSpan"></span></p>
```

Dan inilah yang akan saya rancang berikutnya, yakni menambahkan element form lain dan membuat proses validasinya.

## 22.16 Case Study: Validasi Berbagai Element Form

Sepanjang pembahasan tentang **form validation**, saya hanya menggunakan 1 tag `<input>`. Ini semata-mata untuk menyederhanakan pembahasan. Jika anda sudah paham prinsip kerja dari apa yang sudah kita bahas, untuk form element lain cukup mengulang proses yang sama.

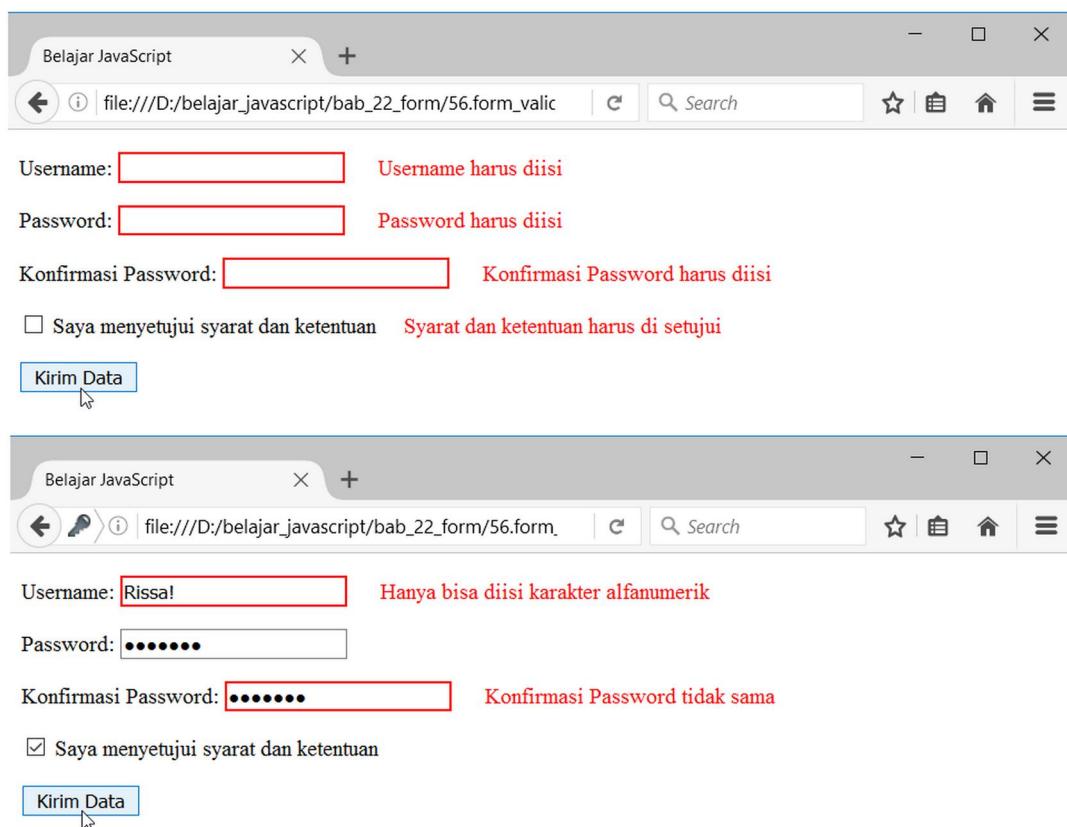
Sebagai case study bab ini, saya akan membuat 3 element form tambahan, yakni berupa 2 buah text box untuk inputan password, dan 1 buah checkbox. Total terdapat 4 element form. Berikut tampilannya:



Gambar: Form Dengan 4 Element

Selain inputan username, terdapat 2 inputan untuk password dan konfirmasi password. Kemudian terdapat 1 buah checkbox. Saya rasa tidak ada masalah dengan struktur HTML yang digunakan untuk membuat form seperti ini. Mari kita bahas tentang proses validasi:

- Untuk inputan **username**, tidak ada penambahan. Kita sudah membuat 3 syarat validasi: username tidak boleh kosong, username tidak boleh berisi karakter non-alfanumerik dan username minimal harus 6 karakter.
- Untuk inputan **password**, saya ingin membuat 2 buah syarat: password tidak boleh kosong dan password minimal harus 6 angka.
- Untuk inputan **konfirmasi password**, terdapat 3 syarat: tidak boleh kosong, minimal 6 angka, dan string yang diinput harus sama seperti inputan password.
- Terakhir untuk **checkbox** syaratnya harus di checklist.



Gambar: Berbagai validasi untuk form

Bisakah anda membayangkan kode program yang harus dirancang? Terasa cukup rumit? Betul, tapi semuanya mirip seperti apa yang kita buat untuk **username**.

Langsung saja saya tampilkan kode program final untuk membuat semua fitur validasi ini. Silahkan anda pelajari secara perlahan:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <style>
7     .error {
8       color: red;
9       width: 300px;
10      text-align: center;
11      padding: 2px 10px;
12      margin-left: 10px;
13    }
14   </style>
15 </head>
16 <body>
```

```
17 <form id="formKu" name="formKu" method="get" action="proses.php">
18   <p>Username: <input type="text" name="username" id="username">
19   <span id="usernameSpan"></span></p>
20   <p>Password: <input type="password" name="pass" id="pass">
21   <span id="passSpan"></span></p>
22   <p>Konfirmasi Password: <input type="password" name="konfPass"
23   id="konfPass"><span id="konfPassSpan"></span></p>
24   <p><input type="checkbox" name="syarat" id="syarat">
25   Saya menyetujui syarat dan ketentuan<span id="syaratSpan"></span></p>
26   <p><input type="submit" name="submit" id="submit" value="Kirim Data"></p>
27 </form>
28 <script>
29   var formKuNode = document.getElementById("formKu");
30
31   var usernameNode = document.getElementById("username");
32   var usernameSpanNode = document.getElementById("usernameSpan");
33
34   var passNode = document.getElementById("pass");
35   var passSpanNode = document.getElementById("passSpan");
36
37   var konfPassNode = document.getElementById("konfPass");
38   var konfPassSpanNode = document.getElementById("konfPassSpan");
39
40   var syaratNode = document.getElementById("syarat");
41   var syaratSpanNode = document.getElementById("syaratSpan");
42
43   function diProses(e){
44
45     //===== Untuk Validasi username ===== //
46     var usernameError="";
47
48     if (usernameNode.value.trim() === ""){
49       usernameError = "Username harus diisi";
50     }
51     else if (/^\W/.test(usernameNode.value.trim()) ){
52       usernameError = "Hanya bisa diisi karakter alfanumerik";
53     }
54     else if (usernameNode.value.trim().length < 6 ){
55       usernameError = "Username minimal 6 karakter";
56     }
57
58     if (usernameError !== ""){
59       usernameSpanNode.innerHTML = usernameError;
60       usernameSpanNode.className = "error";
61       usernameNode.style.border = "2px solid red";
62       e.preventDefault();
63     }
64   }
65
66   //===== Untuk Validasi Password ===== //
67   var passNode = document.getElementById("pass");
68   var passSpanNode = document.getElementById("passSpan");
69
70   if (passNode.value === ""){
71     passSpanNode.innerHTML = "Password tidak boleh kosong";
72     passSpanNode.className = "error";
73     passNode.style.border = "2px solid red";
74     e.preventDefault();
75   }
76
77   //===== Untuk Validasi Konfirmasi Password ===== //
78   var konfPassNode = document.getElementById("konfPass");
79   var konfPassSpanNode = document.getElementById("konfPassSpan");
80
81   if (konfPassNode.value !== passNode.value){
82     konfPassSpanNode.innerHTML = "Konfirmasi Password tidak cocok";
83     konfPassSpanNode.className = "error";
84     konfPassNode.style.border = "2px solid red";
85     e.preventDefault();
86   }
87
88   //===== Untuk Validasi Syarat ===== //
89   var syaratNode = document.getElementById("syarat");
90   var syaratSpanNode = document.getElementById("syaratSpan");
91
92   if (!syaratNode.checked){
93     syaratSpanNode.innerHTML = "Syarat dan ketentuan harus diperiksa";
94     syaratSpanNode.className = "error";
95     syaratNode.style.border = "2px solid red";
96     e.preventDefault();
97   }
98
99   //===== Submit Form ===== //
100  if (usernameError === "" && passNode.value !== "" && konfPassNode.value === passNode.value && !syaratNode.checked){
101    formKuNode.submit();
102  }
103
```

```
63     }
64
65     //===== Untuk Validasi Password ===== //
66     var passError="";
67     if (passNode.value.trim() === ""){
68         passError = "Password harus diisi";
69     }
70     else if (passNode.value.trim().length < 6 ){
71         passError = "Password minimal 6 karakter";
72     }
73
74     if (passError !== ""){
75         passSpanNode.innerHTML = passError;
76         passSpanNode.className = "error";
77         passNode.style.border = "2px solid red";
78         e.preventDefault();
79     }
80
81     //===== Untuk Validasi Konfirmasi Password ===== //
82     var konfPassError="";
83     if (konfPassNode.value.trim() === ""){
84         konfPassError = "Konfirmasi Password harus diisi";
85     }
86     else if (konfPassNode.value.trim().length < 6 ){
87         konfPassError = "Konfirmasi Password minimal 6 karakter";
88     }
89     else if (konfPassNode.value !== passNode.value){
90         konfPassError = "Konfirmasi Password tidak sama";
91     }
92
93     if (konfPassError !== ""){
94         konfPassSpanNode.innerHTML = konfPassError;
95         konfPassSpanNode.className = "error";
96         konfPassNode.style.border = "2px solid red";
97         e.preventDefault();
98     }
99
100    //===== Untuk Validasi Checkbox Syarat ===== //
101    var syaratError="";
102    if (syaratNode.checked !== true){
103        syaratError = "Syarat dan ketentuan harus di setujui";
104    }
105
106    if (syaratError !== ""){
107        syaratSpanNode.innerHTML = syaratError;
108        syaratSpanNode.className = "error";
```

```
109         e.preventDefault();
110     }
111 }
112
113 function hapusError(e){
114     e.target.style.border = "";
115     e.target.parentElement.lastChild.innerHTML = "";
116 }
117
118 formKuNode.addEventListener("submit",diProses);
119 usernameNode.addEventListener("focus",hapusError);
120 passNode.addEventListener("focus",hapusError);
121 konfPassNode.addEventListener("focus",hapusError);
122 syaratNode.addEventListener("focus",hapusError);
123 </script>
124 </body>
125 </html>
```

Total kode program ini butuh 125 baris! Terasa sangat panjang jika dibandingkan dengan kode program yang kita bahas sepanjang buku ini, tapi sebenarnya relatif singkat untuk website yang “real”.

Jika anda sudah memahami materi tentang form validation yang sudah kita pelajari sebelumnya, tidak akan kesulitan mempelajari kode program diatas. Untuk element form password, konfirmasi password dan checkbox, caranya sangat mirip seperti kode validasi untuk username.

Terdapat tag `<span>` di samping setiap tag `<input>`. Disinilah nantinya pesan error untuk setiap element form akan ditampilkan.

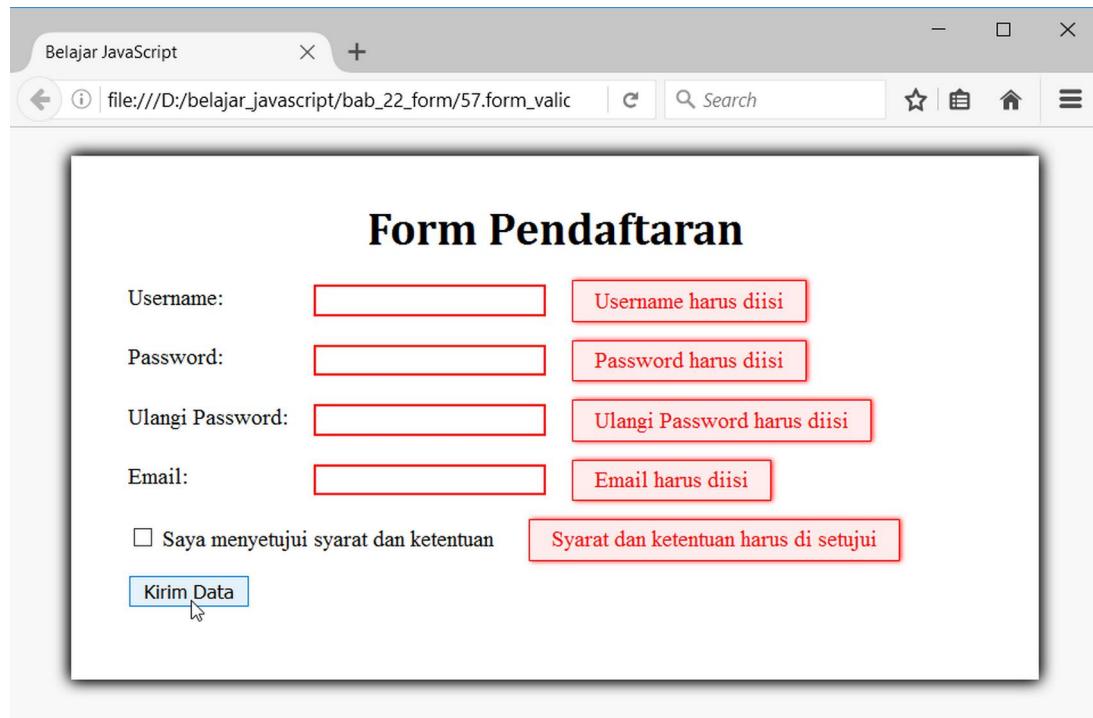
Selain itu, hal yang baru adalah cara membandingkan isi inputan password dan inputan konfirmasi password. Untuk mengecek keduanya, saya menggunakan kondisi `if (konfPassNode.value !== passNode.value)`. Jika hasilnya tidak cocok, maka variabel `konfPassError` akan berisi string “Konfirmasi Password tidak sama”. Pesan error ini akan tampil disamping text box konfirmasi password.

Untuk checkbox, pengecekan apakah checkbox sudah diisi atau tidak dilakukan dengan perintah `if (syaratNode.checked !== true)`. Jika checkbox syarat belum diisi, maka variabel `syaratError` akan berisi string “Syarat dan ketentuan harus di setujui”.

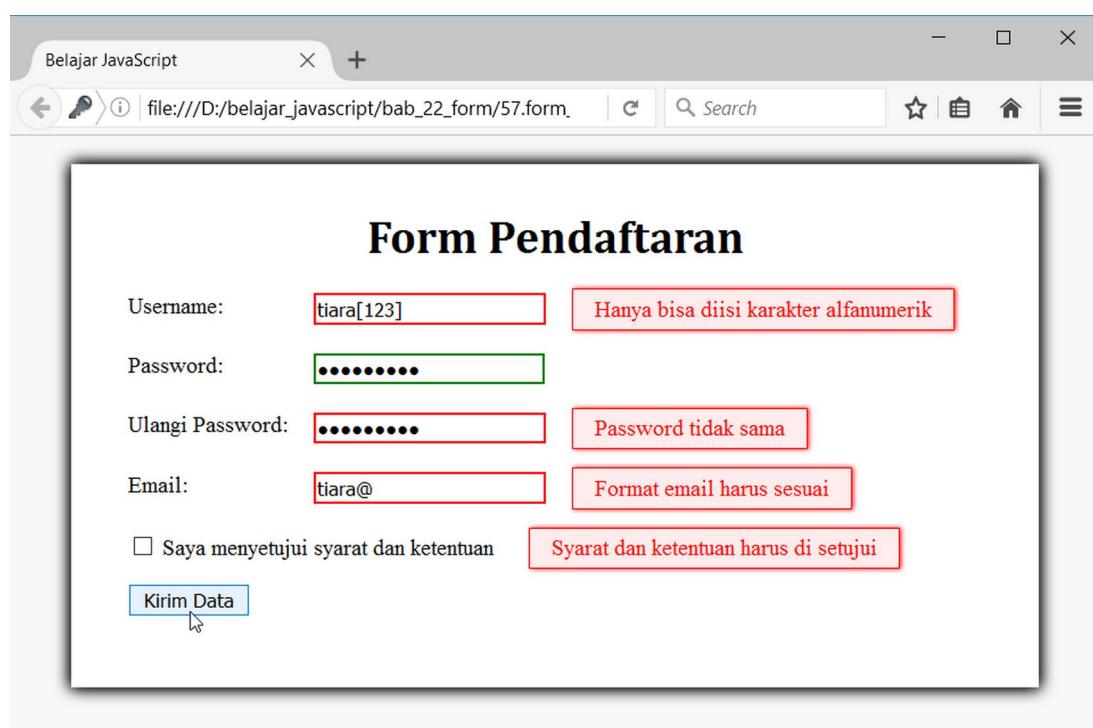
## Case Study: Validasi Berbagai Element Form + CSS

Menutup bab tentang form processing (akhirnya...) Saya ingin menyajikan sebuah form final dengan proses validasi + kode CSS.

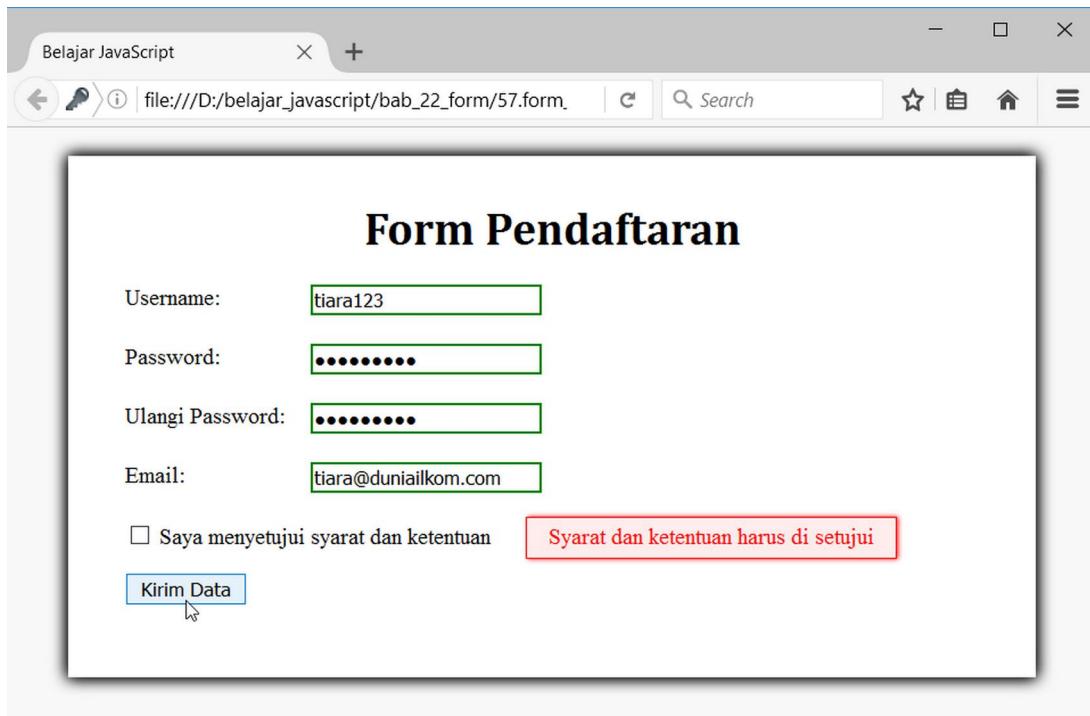
Berikut tampilan “Form Registrasi” yang saya rancang:



Gambar: Pesan error “form registrasi”



Gambar: Pesan error “form registrasi”



Gambar: Pesan error “form registrasi”

Pada dasarnya “form registrasi” ini adalah penambahan dari form yang kita buat sebelumnya. Saya menambahkan kode CSS untuk mendesign form agar tampil lebih cantik.

Selain itu, terdapat tambahan text input email. Syarat untuk email adalah tidak boleh kosong dan harus memenuhi pola regular expression / .+@.+.+/ . Artinya harus ada minimal 1 karakter sebelum dan sesudah tanda @, diikuti dengan sebuah tanda titik, dan minimal sebuah karakter lain.

Sebagai tambahan, ketika form di submit dan isian sudah benar, warna text box akan menjadi hijau, menandakan isian form sudah sesuai. Untuk menambahkan efek ini, saya menambahkan perintah berikut:

```
1 if (usernameError !== ""){  
2   usernameSpanNode.innerHTML = usernameError;  
3   usernameSpanNode.className = "error";  
4   usernameNode.style.border = "2px solid red";  
5   e.preventDefault();  
6 } else {  
7   usernameNode.style.border = "2px solid green";  
8 }
```

Artinya jika variabel usernameError tidak berisi apa-apa, perintah usernameNode.style.border = “2px solid green” akan dijalankan. Sehingga border dari tag usernameNode akan berwarna hijau. Perintah yang sama juga digunakan untuk element form lain.

Seperti yang bisa anda tebak, kode program untuk membuat tampilan ini lebih panjang daripada sebelumnya (butuh sekitar 180an baris). Saya memutuskan untuk tidak menampilkan

kode programnya. Jika anda tertarik bisa membukanya di file `belajar_javascript.zip`, folder `bab_22_form`, dan nama file `57.form_validasi_final_css.html`.

Silahkan pelajari kode program tersebut dan saya sangat sarankan anda memodifikasinya lebih jauh lagi. Misalkan dengan menambah beberapa element form lain seperti radio button atau pilihan dropdown dari tag `<select>`. Jika sebelumnya sudah mempelajari CSS, anda juga bisa menambahkan efek-efek lain.

---

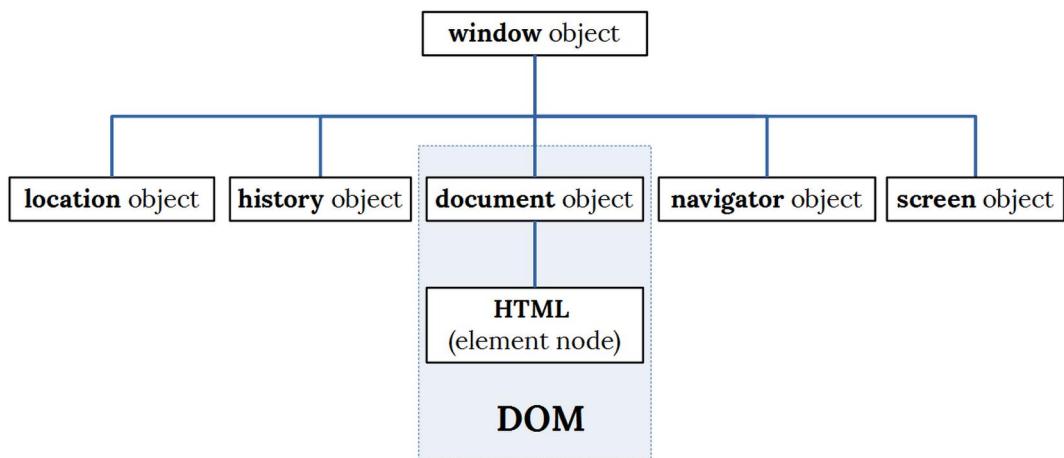
Bab tentang form processing di JavaScript saya buat “sangat panjang”, hampir 100 halaman untuk membahasnya. Tapi karena sangat penting dan sering digunakan, materi form JavaScript memang perlu dibahas dengan detail.

Berikutnya kita akan masuk ke **BOM (Browser Object Model)**.

# 23. BOM (Browser Object Model)

BOM (Browser Object Model) adalah object yang digunakan untuk memanipulasi web browser. Perkenalan singkat dengan BOM sudah kita bahas di bab tentang **Document Object Model (DOM)**.

Sama seperti DOM, BOM juga berada langsung dibawah **window object**, sebagaimana yang tersaji dari diagram dibawah ini:



Gambar: Diagram BOM (Browser Object Model) dan DOM (Document Object Model)

Terdapat berbagai object yang setingkat dengan **document object (DOM)**, seperti **location object**, **history object**, **navigator object** dan **screen object**. Object-object inilah yang menyusun **Browser Object Model**.

Berbeda dengan DOM atau **document object** yang standarnya di tetapkan oleh W3C, BOM tidak memiliki standar baku. Akibatnya, bisa saja terdapat perbedaan implementasi dari satu web browser dengan web browser lain.

Seiring dengan perkembangan teknologi, object penyusun BOM juga bisa bertambah. Dalam bab kali ini kita akan membahas object yang sudah didukung penuh oleh mayoritas web browser modern: **location object**, **history object**, **navigator object** dan **screen object**.

## 23.1 Location Object

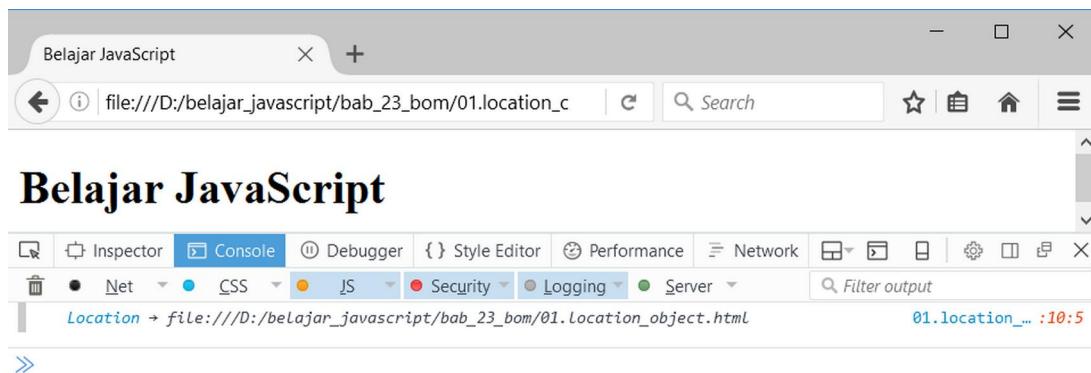
**Location object** adalah object yang berurusan dengan alamat website yang terdapat di *address bar* web browser. Kita bisa mengambil informasi mengenai alamat ini (*location*), mengisi alamat baru, serta me-refresh web browser.

**Location object** bisa diakses dengan perintah `window.location`. Tapi karena `window` merupakan *global object*, kita juga bisa langsung menulis `location` saja, seperti contoh berikut:

```

1 <body>
2   <h1>Belajar JavaScript</h1>
3   <script>
4     console.log(location);
5     // Location -> file:///D:/belajar_javascript/bab_23_bom/
6     //           01.location_object.html
7   </script>
8 </body>

```



Gambar: Tampilan dari perintah `console.log(location)`

Saya menyimpan file diatas di Drive D, folder `belajar_javascript/bab_23_bom/` dan nama file `01.location_object.html`. Hasil dari perintah `console.log(location)` sesuai dengan alamat yang tertulis di bagian address bar web browser.

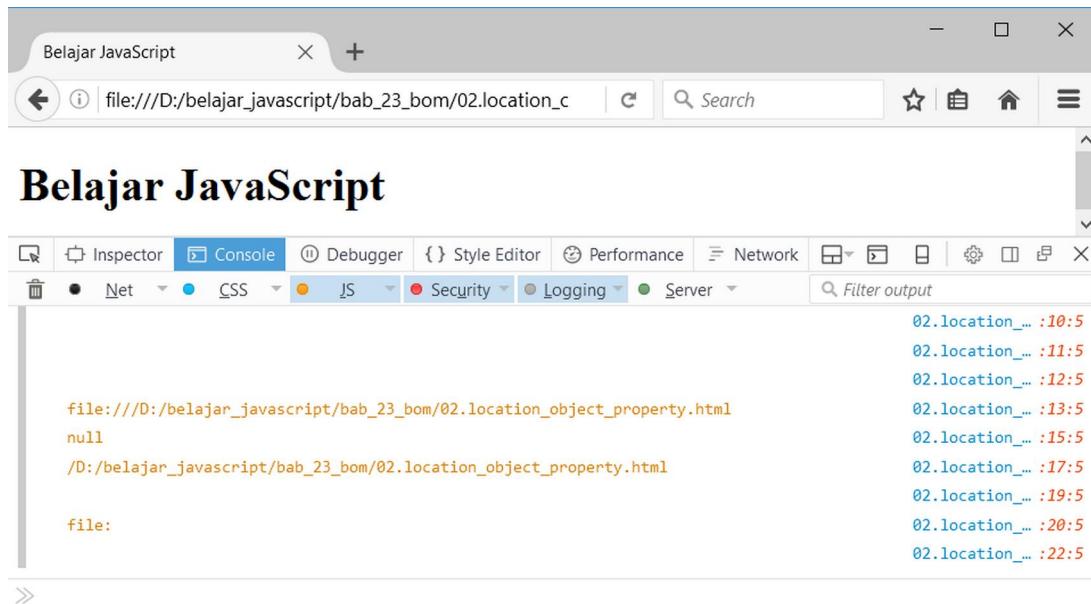
## Location Object Property

Sebagaimana layaknya sebuah object, **location object** juga memiliki berbagai property dan method. Kita akan lihat property terlebih dahulu:

```

1 <body>
2   <h1>Belajar JavaScript</h1>
3   <script>
4     console.log(location.hash);
5     console.log(location.host);
6     console.log(location.hostname);
7     console.log(location.href);
8     console.log(location.origin);
9     console.log(location.pathname);
10    console.log(location.port);
11    console.log(location.protocol);
12    console.log(location.search);
13  </script>
14 </body>

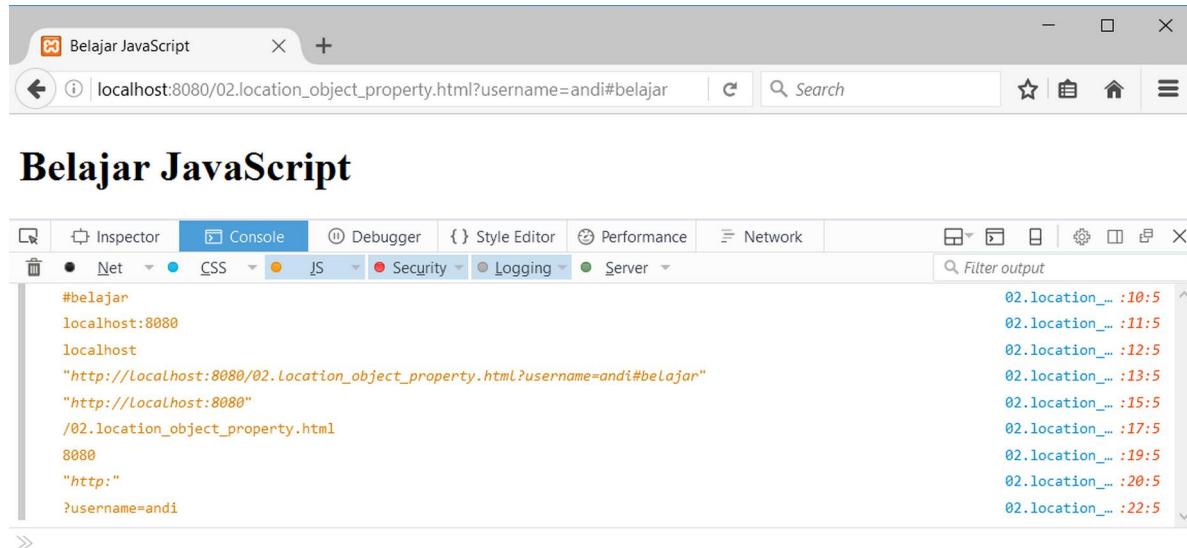
```



Gambar: Mengakses berbagai property dari location object

Dalam kode program diatas, terdapat berbagai perintah `console.log()` untuk mengakses 9 property dari **location object**. Sebagian besar tidak berisi nilai apapun karena saya mengakses file secara langsung dari harddisk komputer. Supaya bisa melihat hasilnya, kode program tersebut harus diakses dari dalam web server (atau di upload ke web hosting).

Saya akan menjalankan ulang dari web server apache (XAMPP). Jika anda sebelumnya sudah mempelajari bahasa pemrograman PHP, bisa juga mencobanya:



Gambar: Hasil dari property location object dari XAMPP

Dapat dilihat bahwa alamat file tidak lagi diawali dengan `file:///D:/belajar_javascript`, tapi `http://localhost:8080/02.location_object_property.html?username=andi#belajar`. Selain mengakses file tersebut dari web server, saya juga mengubah port apache menjadi **8080**, serta menambahkan *query string* dan alamat *hash*, yakni bagian `?username=andi#belajar`.

Berikut penjelasan dari setiap property:

- **location.hash**: Berisi bagian *hash* dari sebuah alamat, seperti #belajar atau #judul\_1.
- **location.host**: Berisi bagian *host* dari sebuah URL, dan termasuk nomor port (jika ada). Contohnya: www.duniaikom.com, atau localhost:8080.
- **location.hostname**: Berisi bagian *namehost* dari sebuah alamat, tidak termasuk nomor port, seperti www.duniaikom.com atau localhost.
- **location.href**: Berisi alamat lengkap dari sebuah URL, termasuk *nama protocol, hostname, path*, serta *hash*.
- **location.origin**: Berisi alamat URL seperti property `location.host`.
- **location.pathname**: Berisi alamat *path*, yakni bagian URL setelah nama domain, seperti /02.location\_object\_property.html.
- **location.port**: Berisi nomor port, seperti 8080. Jika didalam URL tidak ditulis nomor port, hasil property juga akan kosong. Ini berarti alamat URL menggunakan port default, yakni 80 untuk http, dan 443 untuk https.
- **location.protocol**: Berisi nama protocol, seperti `http:`, `https:`, atau `file:`, nama protocol termasuk karakter titik dua “ : ”.
- **location.search**: Berisi bagian alamat untuk *query string*, yakni bagian yang diawali tanda tanya, seperti ?username=andi.



Daftar lengkap mengenai property dan method dari **Location Object** bisa diakses ke [Location Object \(Mozilla Developer Network\)](#)<sup>1</sup>, atau [The Location Object \(w3schools\)](#)<sup>2</sup>.

Selain menampilkan nilai, daftar property diatas juga bisa diubah. Akibatnya, halaman web akan pindah sesuai dengan nilai yang diinput. Sebagai contoh, silahkan anda jalankan kode program berikut:

```

1 <body>
2   <h1>Belajar JavaScript</h1>
3   <script>
4     location.href = "http://www.duniaikom.com";
5   </script>
6 </body>
```

Sesaat setelah tampil, halaman akan langsung menampilkan situs Duniaikom. Ini terjadi karena ketika web browser memproses kode tersebut, terdapat instruksi untuk menukar `location.href` ke `http://www.duniaikom.com`.

Alternatif penulisan adalah dengan menginput langsung alamat URL ke dalam **location object**:

```
location = "http://www.duniaikom.com";
```

---

<sup>1</sup><https://developer.mozilla.org/en-US/docs/Web/API/Location>

<sup>2</sup>[http://www.w3schools.com/jsref/obj\\_location.asp](http://www.w3schools.com/jsref/obj_location.asp)

## Location Object Method

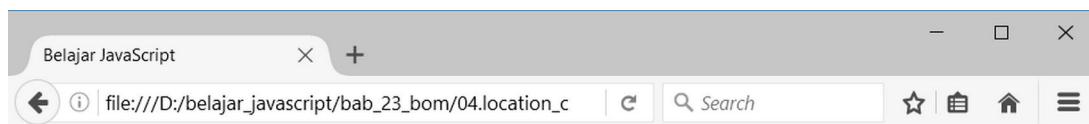
Location object memiliki 3 buah method, yakni:

- **location.assign()**: Membuka alamat baru, dimana alamat URL akan diinput sebagai argumen.
- **location.reload()**: Menampilkan ulang alamat (reload).
- **location.replace()**: Menggantikan alamat saat ini, dimana alamat URL akan diinput sebagai argumen.

Berikut contoh penggunaannya:

```

1 <body>
2   <h1>Belajar JavaScript</h1>
3   <button id="tombolAssign">Assign</button>
4   <button id="tombolReload">Reload</button>
5   <button id="tombolReplace">Replace</button>
6   <script>
7     var tombolAssignNode = document.getElementById("tombolAssign");
8     var tombolReloadNode = document.getElementById("tombolReload");
9     var tombolReplaceNode = document.getElementById("tombolReplace");
10
11    tombolAssignNode.addEventListener("click", function () {
12      location.assign("http://www.google.com");
13    });
14
15    tombolReloadNode.addEventListener("click", function () {
16      location.reload();
17    });
18
19    tombolReplaceNode.addEventListener("click", function () {
20      location.replace("http://www.google.com");
21    });
22  </script>
23 </body>
```



## Belajar JavaScript

Assign Reload Replace

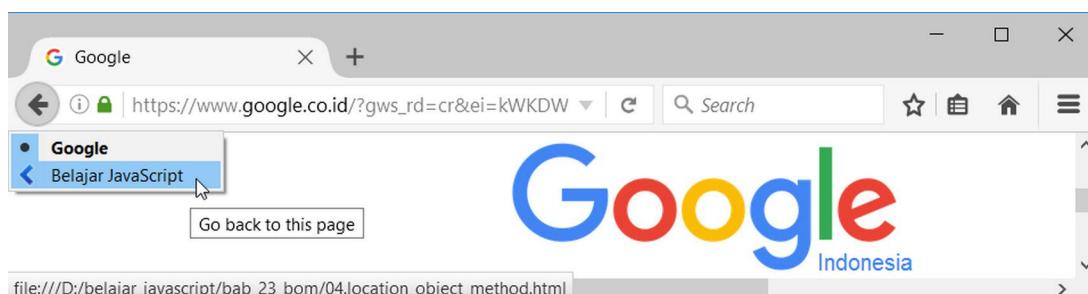
Gambar: Penggunaan method location object

Disini saya memiliki 3 buah tombol: **Assign**, **Reload**, dan **Replace**. Ketiganya memiliki method dari **location object**.

Saat tombol **Assign** dan **Replace** di klik, halaman akan berpindah ke situs [www.google.com](https://www.google.com). Sedangkan ketika tombol **Reload** di klik, halaman akan di reload / refresh.

Walaupun sama-sama digunakan untuk berpindah halaman, method `reload()` dan `replace()` memiliki sedikit perbedaan.

Method `replace()` akan menghapus alamat saat ini dari history web browser, sehingga tidak bisa diakses lagi dengan men-klik tombol back di web browser. Sedangkan method `assign()` akan menambah sebuah alamat baru ke web browser. Jika tombol back di klik, masih bisa kembali ke halaman saat ini.



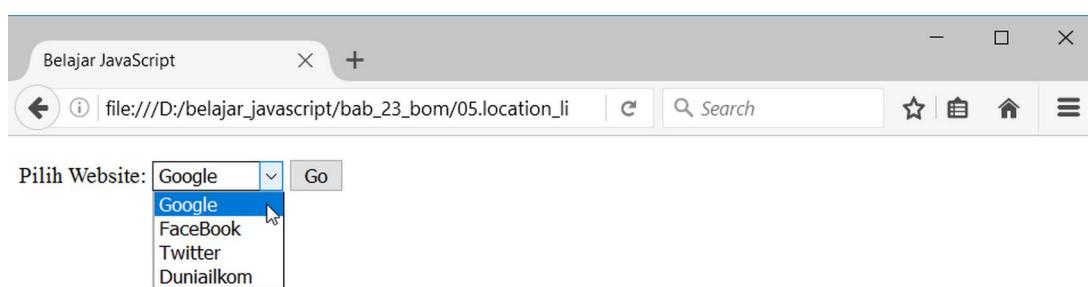
Gambar: Tombol back masih bisa digunakan untuk mundur jika menggunakan `assign()`

Khusus untuk `reload()`, method ini bisa diisi dengan argumen boolean `true` jika kita ingin memaksa proses reload bukan dari cache, tapi meminta halaman baru ke web server:

```
location.reload(true);
```

## Membuat Link Dropdown

Sebagai contoh praktek dari **location object**, saya ingin membuat sebuah link dropdown seperti tampilan berikut:



Gambar: Link dropdown

Tag `<select>` digunakan untuk membuat daftar pilihan situs. Saat tombol "Go" di klik, halaman akan langsung pindah ke website tersebut.

Pertama, tentu kita harus membuat dulu struktur HTML untuk pilihan dropdown. Kemudian, saat tombol "Go" di klik, ambil nilai atribut `value` dari tag `<select>`. Berdasarkan nilai tersebut, ubah nilai `location` untuk pindah halaman.

Berikut kode program yang saya gunakan:

```

1 <body>
2   <p>Pilih Website:
3   <select id="daftarWebsite">
4     <option value="google">Google</option>
5     <option value="facebook">FaceBook</option>
6     <option value="twitter">Twitter</option>
7     <option value="duniailkom">DuniaIlkom</option>
8   </select>
9   <button id="tombol">Go</button>
10  </p>
11  <script>
12    var tombolNode = document.getElementById("tombol");
13    var daftarWebsiteNode = document.getElementById("daftarWebsite");
14
15    function pilihWebsite(){
16      var namaWebsite = daftarWebsiteNode.value;
17      switch (namaWebsite) {
18        case ("google") :
19          window.location = "http://www.google.com";      break;
20        case ("facebook") :
21          window.location = "http://www.facebook.com";  break;
22        case ("twitter") :
23          window.location = "http://www.twitter.com";    break;
24        case ("duniailkom") :
25          window.location = "http://www.duniailkom.com"; break;
26      }
27    }
28
29    tombolNode.addEventListener("click",pilihWebsite);
30  </script>
31 </body>
```

Silahkan anda pelajari sebentar kode program diatas. Saya membuat sebuah *event handler* ketika tombol "GO" di klik. Isinya, ambil nilai atribut *value* dari *daftarWebsiteNode*, kemudian masuk ke kondisi **switch case**. Jika memenuhi salah satu syarat, input alamat website ke dalam **location object**.

Saya menginput alamat URL langsung ke dalam **location object**. Alternatifnya, anda juga bisa menggunakan method *location.assign()* maupun *location.replace()*.

Tantangan selanjutnya, bagaimana dengan menghapus tombol "Go" sama sekali?

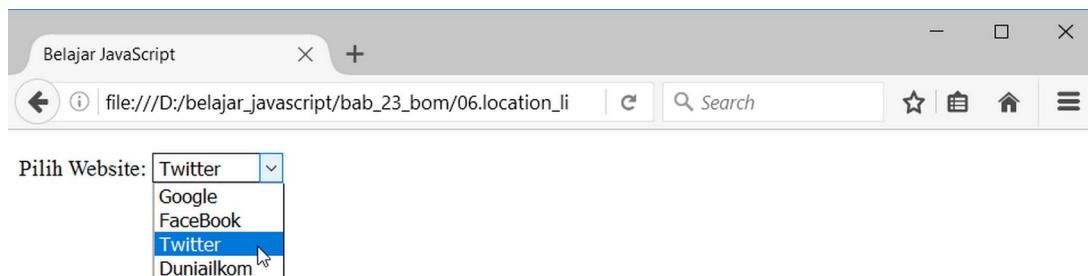
Tidak masalah. Saya yakin anda sudah paham apa yang mesti diubah. Kita tinggal mengganti event **click** dari tombol "Go", menjadi event **change** ke tag **<select>**:

```

1 <script>
2   var daftarWebsiteNode = document.getElementById("daftarWebsite");
3
4   function pilihWebsite(){
5     var namaWebsite = daftarWebsiteNode.value;
6     switch (namaWebsite) {
7       case ("google") :
8         window.location = "http://www.google.com";      break;
9       case ("facebook") :
10        window.location = "http://www.facebook.com";    break;
11       case ("twitter") :
12         window.location = "http://www.twitter.com";     break;
13       case ("duniaIlkom") :
14         window.location = "http://www.duniaIlkom.com";  break;
15     }
16   }
17
18   daftarWebsiteNode.addEventListener("change",pilihWebsite);
19 </script>

```

Sekarang, saat pilihan link diubah, web browser akan langsung pindah ke situs tersebut tanpa butuh tombol tambahan.



Gambar: Link dropdown tanpa tombol

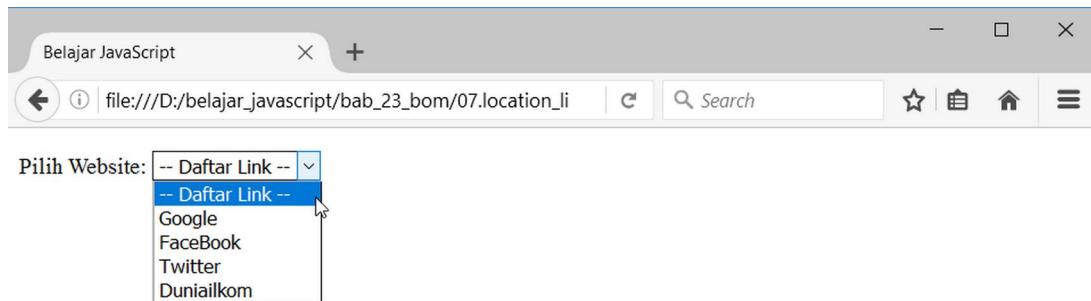
Tapi terdapat sedikit masalah untuk menu link **Google**. Kita tidak bisa menggunakan pilihan ini karena event **change** tidak akan terpanggil. Secara default, tag `<option>` pertama akan langsung terpilih begitu halaman dibuka. Kejadiannya sama seperti contoh kasus dalam bab *form processing*.

Solusinya, kita bisa menambahkan sebuah tag `<option>` yang berisi keterangan ke dalam dropdown. Tag ini diset sebagai pilihan default menggunakan atribut `selected` :

```

1 <select id="daftarWebsite">
2   <option value="" selected>-- Daftar Link --</option>
3   <option value="google">Google</option>
4   <option value="facebook">FaceBook</option>
5   <option value="twitter">Twitter</option>
6   <option value="duniailkom">DuniaIlkom</option>
7 </select>

```



Gambar: Tag <option>-- Daftar Link --</option> di set sebagai pilihan default

Sekarang, event **change** akan berjalan untuk setiap pilihan.

## Penulisan Singkat Link Dropdown

Link dropdown yang baru saja kita rancang cukup sering dipakai. Oleh karena itu saya juga menyiapkan penulisan singkatnya menggunakan inline JavaScript:

```

1 <body>
2   <p>Pilih Website:
3     <select onchange="location = this.value;">
4       <option value="">-- Daftar Link --</option>
5       <option value="http://www.google.com">Google</option>
6       <option value="http://www.facebook.com">FaceBook</option>
7       <option value="http://www.twitter.com">Twitter</option>
8       <option value="http://www.duniailkom.com">DuniaIlkom</option>
9     </select>
10    </p>
11  </body>

```

Kode yang sangat singkat jika dibandingkan dengan contoh kita sebelumnya. Alamat URL saya tempatkan sebagai nilai dari atribut `value` tag `<option>`. Dengan demikian, kita tidak membutuhkan kondisi `if else` atau `switch case` untuk memeriksa nilai yang saat ini sedang dipilih.

Saya menggunakan atribut `onchange="location = this.value;"` untuk menjalankan link. Artinya, ketika event **change** terjadi, `location object` akan diisi dengan atribut `value` dari `"this"`.

Penggunaan variabel `this` di dalam DOM memang belum ada saya bahas, tapi fungsinya kurang lebih sama seperti di dalam object, yakni merujuk kepada object yang saat ini sedang berjalan.

Kode program diatas bisa anda copy paste ke dalam website yang mendukung JavaScript.

## 23.2 History Object

History Object digunakan untuk mengakses history web browser, yakni alamat yang telah dikunjungi sebelum dan sesudah halaman saat ini. Kurang lebih sama seperti menekan tombol `back` dan `forward` yang ada di web browser.

Di karenakan pembatasan *privacy* dan keamanan, tidak banyak yang bisa diakses dari **history object**. Kita hanya bisa mendapatkan info mengenai jumlah history, mundur ke halaman sebelumnya, dan maju ke halaman selanjutnya.

**History object** memiliki 1 property dan 3 method:

- **history.length**: Berisi jumlah alamat URL yang tersimpan di daftar history.
- **history.back()**: Mundur ke alamat URL sebelumnya.
- **history.forward()**: Maju ke alamat URL berikutnya.
- **history.go()**: Pergi ke halaman tertentu dari daftar history (tergantung nilai argument). Jika diinput angka positif, digunakan untuk maju ke alamat berikutnya. Jika diinput angka negatif, digunakan untuk mundur ke alamat sebelumnya.



Daftar lengkap mengenai property dan method dari **History Object** bisa diakses ke [History Object \(Mozilla Developer Network\)<sup>3</sup>](#), atau [The History Object \(w3schools\)<sup>4</sup>](#).

Berikut contoh penggunaan dari **history object**:

```

1 <body>
2   <h1>Belajar JavaScript</h1>
3   <p>Jumlah alamat di history: <span id="hasil"></span></p>
4   <button id="back">Back</button>
5   <button id="forward">Forward</button>
6   <button id="back2">Back 2 kali</button>
7   <script>
8     var hasilNode = document.getElementById("hasil");
9     var backNode = document.getElementById("back");
10    var back2Node = document.getElementById("back2");
11    var forwardNode = document.getElementById("forward");
12
13    hasilNode.innerHTML = history.length;

```

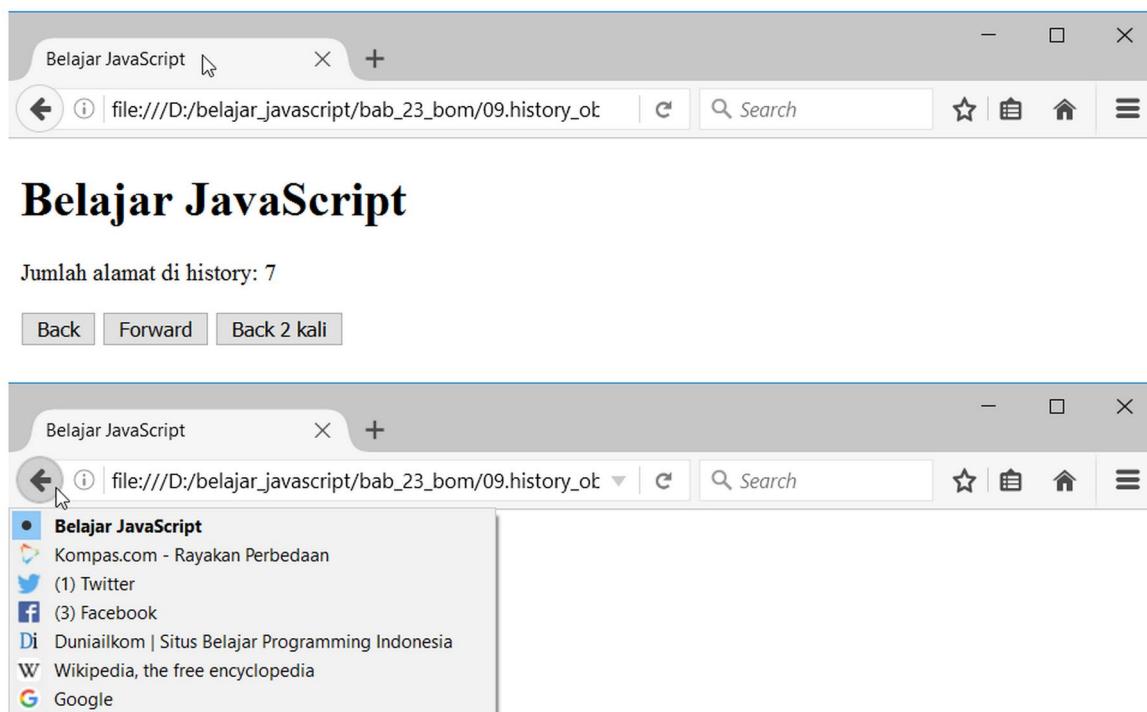
<sup>3</sup><https://developer.mozilla.org/en-US/docs/Web/API/History>

<sup>4</sup>[http://www.w3schools.com/jsref/obj\\_history.asp](http://www.w3schools.com/jsref/obj_history.asp)

```

14
15     backNode.addEventListener("click", function () {
16         history.back();
17     });
18
19     forwardNode.addEventListener("click",function () {
20         history.forward();
21     });
22
23     back2Node.addEventListener("click",function () {
24         history.go(-2);
25     });
26 </script>
27 </body>

```



Gambar: Penggunaan history object

Agar kode program diatas bisa berjalan, anda harus “mengisi” history web browser terlebih dahulu. Caranya silahkan kunjungi beberapa website, kemudian baru buka file yang berisi kode diatas.

Di awal program, saya menginput property `history.length` ke dalam `hasilNode.innerHTML`. Hasilnya terdapat angka 7. Artinya ada 7 alamat website yang sudah saya kunjungi dan tersimpan di dalam history web browser.

Tombol **Back** dan **Forward** berisi method `history.back()` dan `history.forward()`. Jika salah satu tombol ini di klik, jendela web browser akan pindah ke halaman lain yang terdapat di history. Tombol **back** untuk mundur, serta tombol **forward** untuk maju.

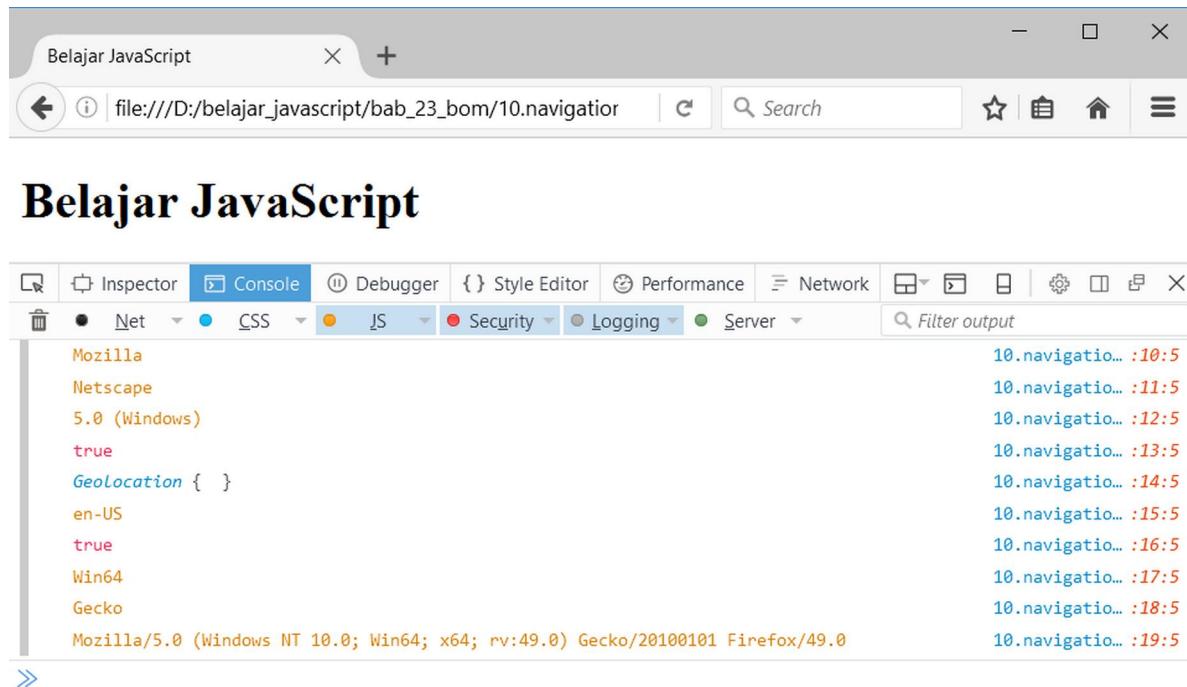
Tombol Back 2 kali berisi method `history.go(-2)`. Argumen negatif ke dalam method `go()` digunakan untuk mundur 2 kali dari daftar history. Jika anda ingin membuat tombol maju 3 kali, bisa menggunakan method `history.go(3)`. Tentunya ini baru akan berjalan jika daftar history web browser sudah berisi dengan alamat URL.

## 23.3 Navigator Object

**Navigator Object** berisi informasi mengenai web browser yang sedang dipakai. Biasanya object ini digunakan untuk fitur “*browser sniffing*”. **Browser sniffing** adalah sebuah cara untuk mengetahui jenis web browser dan menjalankan kode program yang sesuai untuk web browser tersebut.

**Navigator Object** memiliki beberapa property yang akan kita akses menggunakan contoh kode program berikut:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <script>
4     console.log(navigator.appCodeName);           // Mozilla
5     console.log(navigator.appName);              // Netscape
6     console.log(navigator.appVersion);           // 5.0 (Windows)
7     console.log(navigator.cookieEnabled);         // true
8     console.log(navigator.geolocation);          // Geolocation { }
9     console.log(navigator.language);             // en-US
10    console.log(navigator.onLine);                // true
11    console.log(navigator.platform);             // Win64
12    console.log(navigator.product);              // Gecko
13    console.log(navigator.userAgent);
14    // Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:49.0) Gecko/20100101
15    // Firefox/49.0
16  </script>
17 </body>
```



Gambar: Tampilan berbagai property navigation object di web browser **Mozilla Firefox**

Berikut penjelasan dari property diatas:

- **navigator.appCodeName**: Berisi nilai *code name* web browser.
- **navigator.appName**: Berisi nilai *name* web browser.
- **navigator.appVersion**: Berisi nilai versi web browser.
- **navigator.cookieEnabled**: Berisi nilai boolean akan fitur *cookie* aktif di web browser.
- **navigator.geolocation**: Berisi sebuah *Geolocation object* yang bisa digunakan untuk mengetahui posisi pengguna (tidak selalu bisa diakses).
- **navigator.language**: Berisi kode bahasa yang di gunakan web browser.
- **navigator.onLine**: Berisi nilai boolean apakah web browser sedang online.
- **navigator.platform**: Berisi kode sistem operasi yang digunakan.
- **navigator.product**: Berisi nilai *web browser engine*.
- **navigator.userAgent**: Berisi nilai *user-agent header* yang dikirim web browser ke web server.



Daftar lengkap mengenai property dan method dari **Navigator Object** bisa diakses ke [Navigator Object \(Mozilla Developer Network\)<sup>5</sup>](https://developer.mozilla.org/en-US/docs/Web/API/Navigator), atau [The Navigator Object \(w3schools\)<sup>6</sup>](http://www.w3schools.com/jsref/obj_navigator.asp).

Tampilan dari gambar sebelumnya adalah hasil yang saya dapat dari web browser **Mozilla Firefox**. Bagaimana dengan web browser lain? Mari kita coba:

<sup>5</sup><https://developer.mozilla.org/en-US/docs/Web/API/Navigator>

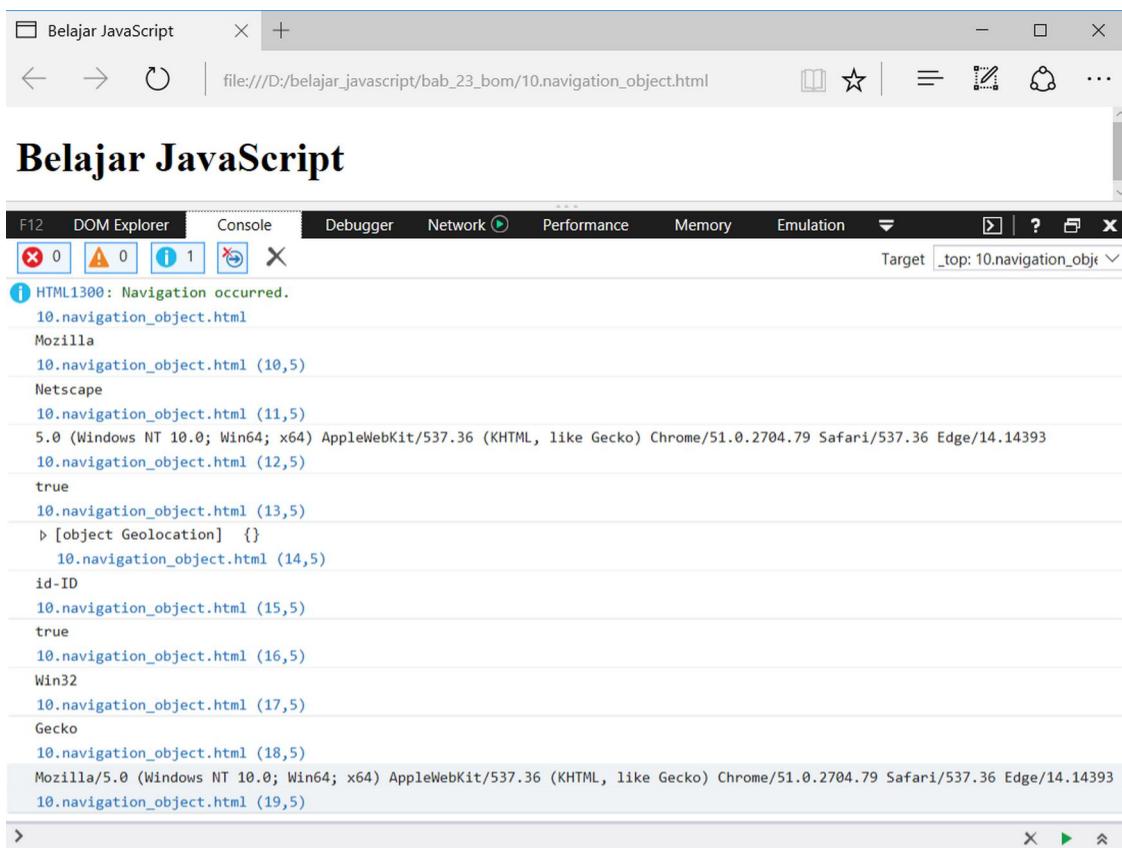
<sup>6</sup>[http://www.w3schools.com/jsref/obj\\_navigator.asp](http://www.w3schools.com/jsref/obj_navigator.asp)



## Belajar JavaScript



Gambar: Tampilan hasil navigator object di web browser **Opera**



Gambar: Tampilan hasil navigator object di web browser **Microsoft Edge**

Perhatikan hasil dari `navigator.appCodeName` dan `navigator.appName`. Meskipun saya menggunakan web browser **Opera** dan **Microsoft Edge**, hasilnya tetap sama: Mozilla dan Netscape. Bahkan seluruh web browser modern juga akan menampilkan hasil Mozilla dan Netscape. Apa yang terjadi?

Hasil ini adalah “warisan” sejak awal kemunculan web browser. Saat itu **Netscape Navigator** (codename: Mozilla) merupakan web browser paling populer dengan berbagai fitur mutakhir seperti *frame*. Banyak programmer mengandalkan **navigation object** untuk membuat tampilan yang menarik dan hanya bisa berjalan pada web browser Netscape.

Ketika muncul web browser lain seperti Internet Explorer, fitur yang sama juga tersedia, tapi tidak bisa berjalan sempurna. Penyebabnya, situs yang mayoritas beredar sudah terlanjur ditujukan untuk Netscape.

Mengatasi masalah ini, IE “menyamar” menjadi Netscape dengan mengembalikan nilai `navigator.object` sebagai **Netscape** dan juga menggunakan *code name* Mozilla. Lambat laun web browser lain juga mengikuti langkah yang sama.

Jadi, bagaimana cara memeriksa web browser jika hasil dari `navigator.appCodeName` dan `navigator.appName` tidak bermanfaat sama sekali? Kita bisa memakai informasi dari property `navigator.userAgent`.

Akan tetapi properti `navigator.userAgent` juga memiliki masalah yang sama. Berikut hasil property `navigator.userAgent` dari web browser **Opera**:

```
Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
Chrome/55.0.2883.87 Safari/537.36 OPR/42.0.2393.137
```

Perhatikan, terdapat banyak sekali informasi terkait “merk” web browser. Terdapat string Mozilla, AppleWebKit, KHTML, Chrome, dan Safari. Yang manakah kode untuk Opera? ternyata ada di string OPR.

Sebagai perbandingan, berikut hasil `navigator.userAgent` dari web browser **Microsoft IE 11**:

```
Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; .NET4.0C; .NET4.0E; .NET CLR  
R 2.0.50727; .NET CLR 3.0.30729; .NET CLR 3.5.30729; rv:11.0) like Gecko
```

Ternyata lebih parah lagi, tidak terdapat informasi apapun yang mengandung kode Microsoft, Internet Explorer, maupun IE.

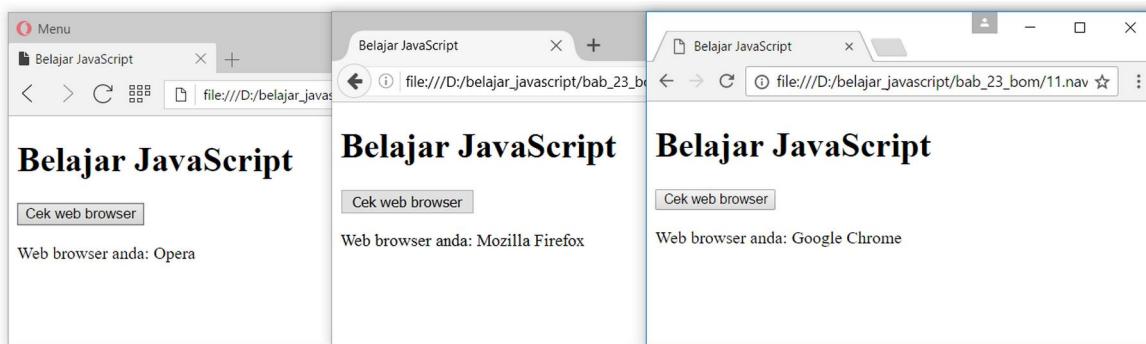
Inilah efek dari web browser yang satu ingin menyamarkan diri sebagai web browser lain. Jika anda tertarik membaca mengenai masalah ini, bisa ke: [History of the browser user-agent string<sup>7</sup>](#).

Untuk mendeteksi web browser yang digunakan (*browser sniffing*), kita harus memeriksa kemunculan pola string tertentu dari hasil property `navigator.userAgent`. Berikut kode program yang bisa digunakan untuk hal ini:

---

<sup>7</sup> <http://webaim.org/blog/user-agent-string-history/>

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <button id="tombol">Cek web browser</button>
4   <p>Web browser anda: <span id="hasil"></span></p>
5   <script>
6     var tombolNode = document.getElementById("tombol");
7     var hasilNode = document.getElementById("hasil");
8
9     function periksaWebBrowser() {
10       if((navigator.userAgent.indexOf("Opera") != -1) ||
11          (navigator.userAgent.indexOf("OPR") != -1)) {
12         hasilNode.innerHTML = "Opera";
13       }
14       else if(navigator.userAgent.indexOf("Edge") != -1) {
15         hasilNode.innerHTML = "Microsoft Edge";
16       }
17       else if(navigator.userAgent.indexOf("Chrome") != -1) {
18         hasilNode.innerHTML = "Google Chrome";
19       }
20       else if(navigator.userAgent.indexOf("Safari") != -1) {
21         hasilNode.innerHTML = "Apple Safari";
22       }
23       else if(navigator.userAgent.indexOf("Firefox") != -1) {
24         hasilNode.innerHTML = "Mozilla Firefox";
25       }
26       else if((navigator.userAgent.indexOf("MSIE") != -1) ||
27                 (!document.documentMode === true)) //IF IE > 10
28       {
29         hasilNode.innerHTML = "Microsoft Internet Explorer";
30       }
31       else {
32         hasilNode.innerHTML = "Tidak diketahui";
33       }
34     }
35
36     tombolNode.addEventListener("click",periksaWebBrowser);
37   </script>
38 </body>
```



Gambar: Hasil browser sniffing untuk web browser **Opera**, **Firefox**, dan **Chrome**

Kode program diatas saya ambil dari forum stackoverflow.com: [How to detect Safari, Chrome, IE, Firefox and Opera browser?](#)<sup>8</sup>. Kodenya cukup panjang karena terdapat 6 kondisi **if else** untuk mendeteksi 6 jenis web browser.

Untuk mengetahui jenis web browser yang dipakai, kita harus memeriksa apakah string tertentu muncul dari hasil `navigator.userAgent`. Caranya, dengan menggunakan method `indexOf()` milik string object. Jika kode tertentu ditemukan, method `indexOf()` akan menghasilkan nilai `true`.

Urutan dari kondisi **if else** juga berpengaruh. Seperti yang sudah kita lihat, hasil `navigator.userAgent` milik Opera juga mengandung string `Chrome` dan `Safari`. Oleh karena itu string `Opera` atau `OPR` milik web browser `opera` harus diperiksa paling awal.

Trik *browser sniffing* seperti ini dulunya umum dipakai, karena banyak fitur yang hanya bisa berjalan di web browser tertentu saja. Tapi saat ini penggunaannya sudah tidak disarankan lagi.

Jika anda ingin menggunakan fitur terbaru yang hanya di dukung oleh sedikit web browser, bisa menggunakan konsep **Feature Detection** sebagai pengganti *browser sniffing*.

**Feature Detection** adalah sebuah metode untuk memeriksa apakah sebuah fitur tersedia atau tidak di dalam web browser. Caranya dengan mencoba memanggil object dari fitur tersebut. Jika hasilnya `null` atau `false`, bisa dipastikan web browser tersebut tidak mendukungnya.

Berikut contoh penggunaan *Feature Detection*:

```

1 if (window.XMLHttpRequest) {
2     console.log("XMLHttpRequest object tersedia");
3     new XMLHttpRequest();
4 }
```

Dalam kode program diatas saya memeriksa apakah object `XMLHttpRequest` tersedia di dalam web browser. Hanya jika object tersebut bisa dipakai, kita baru masuk ke dalam kode program. Dengan cara seperti ini, kode JavaScript tidak menghasilkan error untuk web browser yang belum mendukung `XMLHttpRequest`.

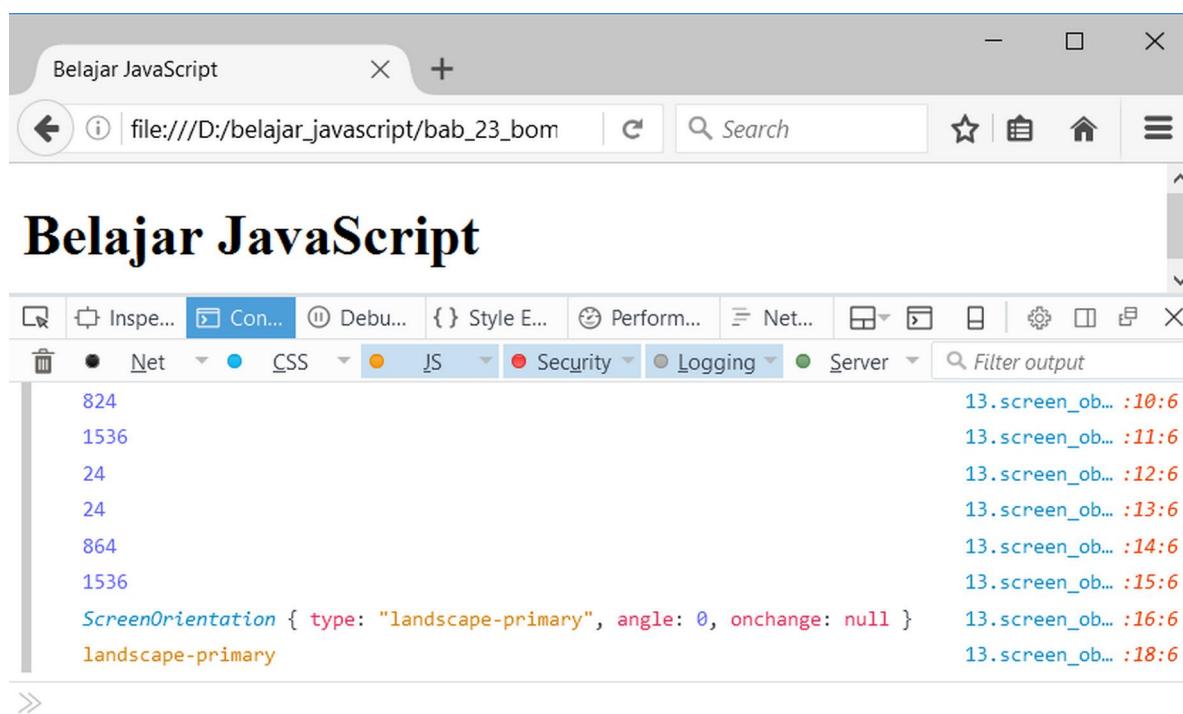
<sup>8</sup><http://stackoverflow.com/questions/9847580/how-to-detect-safari-chrome-ie-firefox-and-opera-browser>

## 23.4 Screen Object

Screen Object berisi informasi mengenai layar (*screen*) yang digunakan oleh pengunjung. Melalui *screen object*, kita bisa mengecek ukuran layar, kedalaman warna, serta orientasi layar.

Berikut kode program untuk mengakses property dari screen object:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <script>
4     console.log(screen.availHeight);      // 824
5     console.log(screen.availWidth);       // 1536
6     console.log(screen.colorDepth);      // 24
7     console.log(screen.pixelDepth);      // 24
8     console.log(screen.height);          // 864
9     console.log(screen.width);           // 1536
10    console.log(screen.orientation);
11    // ScreenOrientation { type: "landscape-primary", angle: 0, onchange: null \
12  }
13    console.log(screen.orientation.type); // landscape-primary
14  </script>
15 </body>
```



Gambar: Mengakses berbagai property dari screen object

Berikut penjelasan dari property diatas:

- **screen.availHeight**: Berisi nilai tinggi layar yang bisa dipakai dalam satuan pixel (dikurangi dengan bagian yang tidak bisa digunakan seperti *Windows Taskbar*).
- **screen.availWidth**: Berisi nilai lebar layar yang bisa dipakai dalam satuan pixel (dikurangi dengan bagian yang tidak bisa digunakan seperti *Windows Taskbar*).
- **screen.colorDepth**: Berisi kedalaman warna (*color depth*) dalam satuan bit per pixel.
- **screen.pixelDepth**: Berisi kedalaman bit (*bit depth*) dalam satuan bit per pixel. Nilai property ini biasanya sama dengan `colorDepth`.
- **screen.height**: Berisi nilai total tinggi layar dalam satuan pixel.
- **screen.width**: Berisi nilai lebar layar dalam satuan pixel.
- **screen.orientation**: Berisi *collection* dari orientation (orientasi layar). Property ini relatif baru dan tidak selalu bisa diakses. Property `screen.orientation.type` berisi keterangan apakah layar diset ke *landscape* atau *portrait*. Nilai dari `screen.orientation.type` bisa salah satu dari: *portrait-primary*, *portrait-secondary*, *landscape-primary* dan *landscape-secondary*.



Daftar lengkap mengenai property dan method dari **Screen Object** bisa diakses ke [Screen Object \(Mozilla Developer Network\)](#)<sup>9</sup>, atau [The Screen Object \(w3schools\)](#)<sup>10</sup>.

Semua property dari **screen object** bersifat *read only* dan tidak bisa ditukar. **Screen object** juga tidak terlalu sering dipakai. Tapi property `screen.orientation.type` cukup menarik untuk dicoba, terutama bagi pengunjung web smartphone dan tablet. Menggunakan property ini, kita bisa membuat kode program yang berbeda tergantung posisi layar, apakah tampil *portrait* atau *landscape*.

---

Dalam bab ini kita telah membahas 4 object yang menyusun **BOM (Browser Object Model)**, yakni **location object**, **history object**, **navigator object** dan **screen object**. Dari keempat object ini, yang cukup sering digunakan adalah **location object**, dimana kita bisa mengubah alamat situs yang tampil di jendela web browser.

Kedepannya, sangat mungkin terdapat object-object lain yang dimasukkan ke dalam BOM. Terutama dengan makin populernya pengaksesan website dari perangkat mobile seperti smartphone dan tablet.

---

<sup>9</sup> <https://developer.mozilla.org/en-US/docs/Web/API/Screen>

<sup>10</sup> [http://www.w3schools.com/jsref/obj\\_screen.asp](http://www.w3schools.com/jsref/obj_screen.asp)

# 24. Window Object

**Window object** merupakan global object sekaligus menjadi induk dari **DOM** (Document Object Model) dan **BOM** (Browser Object Model).

**Window object** juga menampung berbagai *event*, *method* dan *property* yang tidak dikelompokkan ke dalam object tersendiri, seperti method `alert()`. Selain itu **window object** bisa digunakan untuk membuat halaman baru yang dikenal dengan *popup window* atau jendela *popup*. Kita akan membahasnya dengan lebih dalam pada bab ini.

- i** Event, property dan method milik **window object** sangat banyak, dan saya hanya akan membahas yang sering digunakan saja. Untuk referensi yang lebih lengkap, bisa mengakses [Window Object - Mozilla Developer Network<sup>1</sup>](#) atau [The Window Object - w3school<sup>2</sup>](#).

## 24.1 Load Event

**Window object** memiliki berbagai event yang bisa kita pakai. Mayoritas event standar seperti *click*, *mouseover*, *keyup* dan *keydown* juga tersedia. Tapi event seperti ini tidak umum digunakan karena tentu saja efeknya akan berdampak ke seluruh jendela tampilan.

Sebagai contoh, dalam kode program berikut saya membuat event *click* untuk **window object**:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <script>
4     window.addEventListener("click", function () {
5       alert("Selamat Pagi...");
6     });
7   </script>
8 </body>
```

Efek dari kode program diatas, setiap kali kita men-klik object apapun di halaman web browser, akan tampil jendela `alert("Selamat Pagi...")`. Tentunya bukan sebuah ide yang bagus.

Event yang sering dipakai untuk **window object** adalah *load event*. **Load event** akan berjalan sesaat setelah dokumen HTML selesai di proses. Agar bisa memahami kegunaan dari *load event*, mari kita buat sebuah percobaan:

---

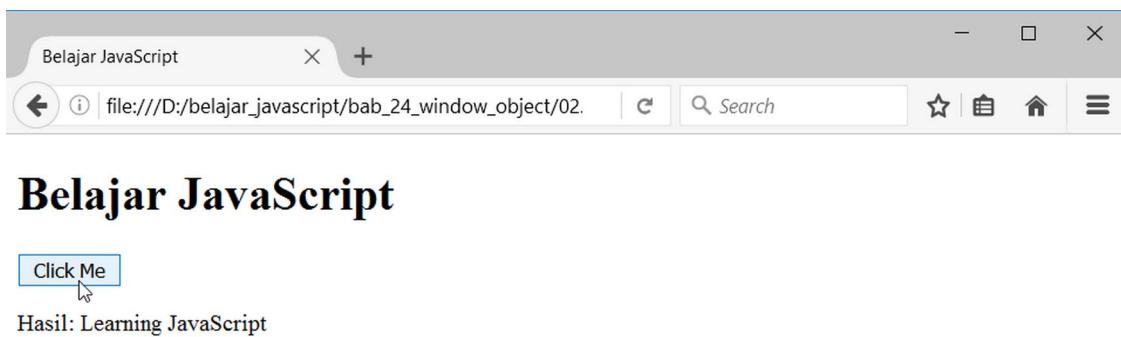
<sup>1</sup><https://developer.mozilla.org/en-US/docs/Web/API/Window>

<sup>2</sup>[http://www.w3schools.com/jsref/obj\\_window.asp](http://www.w3schools.com/jsref/obj_window.asp)

```

1 <body>
2   <h1>Belajar JavaScript</h1>
3   <button id="tombol">Click Me</button>
4   <p>Hasil: <span id="hasil"></span></p>
5   <script>
6     var tombolNode = document.getElementById("tombol");
7     var hasilNode = document.getElementById("hasil");
8
9     function jalankan(){
10       hasilNode.innerHTML = "Learning JavaScript";
11     }
12
13     tombolNode.addEventListener("click",jalankan);
14   </script>
15 </body>

```



Gambar: Kode JavaScript berjalan dengan sukses

Kode program diatas akan menampilkan teks "Learning JavaScript" pada saat tombol Click Me di-klik.

Saya membuat sebuah event **click** ke dalam `<button id="tombol">`. Pada saat event ini berjalan, input teks "Learning JavaScript" ke dalam atribut `innerHTML` dari `<span id="hasil">`. Tidak ada masalah, program seperti ini sudah sering kita buat.

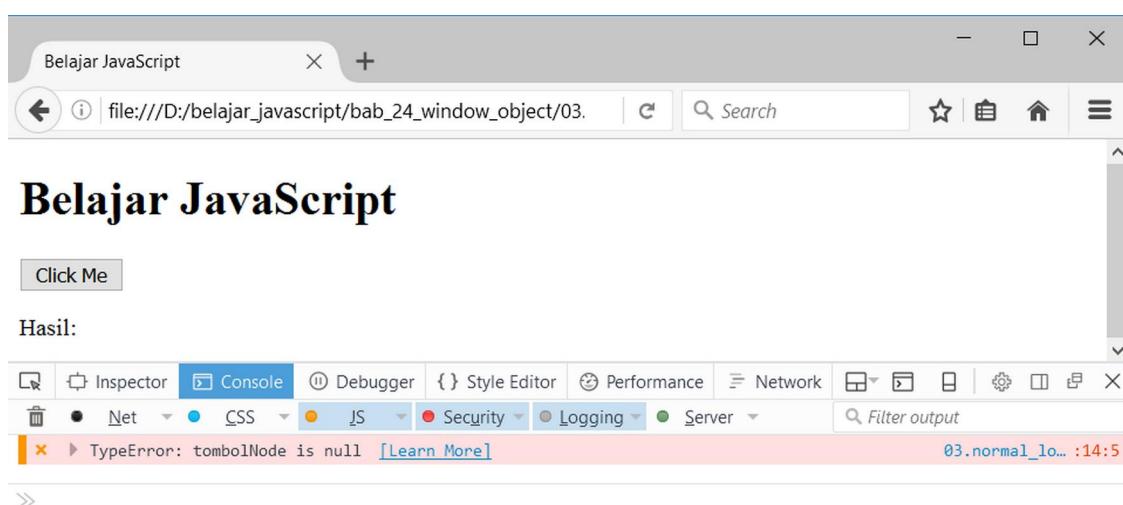
Sekarang, bagaimana jika kode JavaScript saya pindahkan ke bagian `<head>`?

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <script>
7     var tombolNode = document.getElementById("tombol");
8     var hasilNode = document.getElementById("hasil");
9
10    function jalankan(){

```

```
11     hasilNode.innerHTML = "Learning JavaScript";
12 }
13
14 tombolNode.addEventListener("click", jalankan);
15 </script>
16 </head>
17 <body>
18 <h1>Belajar JavaScript</h1>
19 <button id="tombol">Click Me</button>
20 <p>Hasil: <span id="hasil"></span></p>
21 </body>
22 </html>
```



Gambar: Error??

Ketika tombol **Click Me** di klik, tidak tampil teks apapun. Karena merasa ada yang salah, saya membuka tab console dan terlihat pesan `TypeError: tombolNode is null`. Apa yang terjadi?

Ini berkaitan dengan cara web browser memproses kode HTML. Sebuah file HTML akan diproses secara berurutan mulai dari baris 1, 2, dst hingga baris terakhir. Jika terdapat tag `<script>` yang berisi kode program JavaScript, kode tersebut juga akan di proses.

Dalam contoh diatas, pada saat web browser sampai ke tag `<script>`, di baris ke-7 terdapat kode `var tombolNode = document.getElementById("tombol")`. Kode JavaScript ini berguna untuk mencari node object yang memiliki id "tombol" dan menginputnya ke dalam variabel `tombolNode`.

Masalahnya, web browser belum memproses satu pun tag HTML yang memiliki `id="tombol"`. Tag `<button id="tombol">` baru akan diproses pada baris ke-19. Akibatnya pada titik ini kode JavaScript akan menghasilkan error.

Tapi ketika tag `<script>` ditempatkan pada akhir tag `<body>` seperti contoh kita yang pertama, masalah ini tidak terjadi. Karena saat web browser menjalankan kode JavaScript, tag `<button id="tombol">` sudah di proses terlebih dahulu.

Untuk mengatasi masalah seperti ini, kita bisa menggunakan event `load` dari **window object**:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <script>
7     window.addEventListener("load", function () {
8
9       var tombolNode = document.getElementById("tombol");
10      var hasilNode = document.getElementById("hasil");
11
12      function jalankan() {
13        hasilNode.innerHTML = "Learning JavaScript";
14      }
15
16      tombolNode.addEventListener("click", jalankan);
17
18    });
19  </script>
20 </head>
21 <body>
22   <h1>Belajar JavaScript</h1>
23   <button id="tombol">Click Me</button>
24   <p>Hasil: <span id="hasil"></span></p>
25 </body>
26 </html>
```

Perhatikan penambahan event **load** di baris pertama kode JavaScript. Dengan event ini, web browser akan mengeksekusi kode program JavaScript setelah seluruh tag HTML selesai diproses.

Saya juga tidak membuat function terpisah dan langsung menginput kode program JavaScript ke dalam *anonymous function* dari load event. Cara seperti ini umum digunakan, termasuk di dalam library JavaScript seperti **jQuery**. Tapi jika anda ingin memisahkannya menjadi sebuah function, juga tidak masalah:

```
1 <script>
2   function mulaiJavaScript(){
3     var tombolNode = document.getElementById("tombol");
4     var hasilNode = document.getElementById("hasil");
5     function jalankan(){
6       hasilNode.innerHTML = "Learning JavaScript";
7     }
8     tombolNode.addEventListener("click", jalankan);
9   }
10
11   window.addEventListener("load",mulaiJavaScript);
12 </script>
```

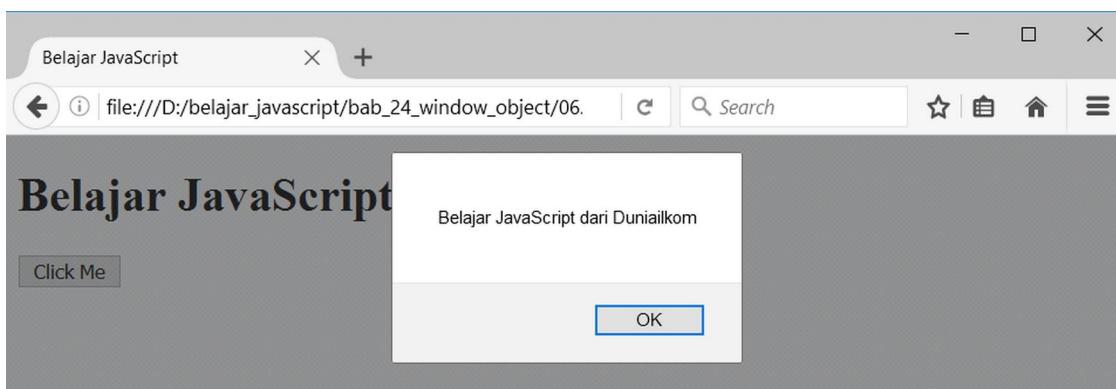
Dalam kode program diatas, function `mulaiJavaScript()` akan dijalankan tepat setelah kode HTML selesai diproses.

## 24.2 Method Window.alert()

Method `window.alert()` sudah sering kita gunakan. Selama ini saya menyamarkan penyebutannya sebagai perintah `alert()`, tapi sebutan yang tepat adalah method `alert()` kepunyaan **window object**.

Method `window.alert()` akan menampilkan jendela popup berisi teks yang diinput sebagai argumen. Walaupun sudah sangat sering kita pakai, berikut contoh penggunaannya:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <button id="tombol">Click Me</button>
4   <script>
5     var tombolNode = document.getElementById("tombol");
6
7     function jalankan(){
8       alert("Belajar JavaScript dari DuniaIlkom");
9     }
10
11    tombolNode.addEventListener("click",jalankan);
12  </script>
13 </body>
```



Gambar: Tampilan method `window.alert()`

Ketika tombol **Click Me** di klik, akan tampil sebuah jendela alert yang berisi string "Belajar JavaScript dari DuniaIlkom".

Jendela alert ini termasuk ke kelompok "**modal windows**", ini adalah sebutan untuk jendela yang memblokir seluruh respon dan pemrosesan sampai jendela tersebut di tutup. Selain `alert()`, juga terdapat method `window.confirm()` dan `window.prompt()` yang akan kita bahas sesaat lagi.

Penggunaan method `alert()` tidak lagi disarankan (selain untuk testing). Karena `alert()` sangat mengganggu dan beberapa web browser juga memblokirnya dengan alasan kenyamanan.

Jika anda butuh jendela seperti ini, bisa mencoba mempelajari cara membuat *modal window* di dalam DOM, misalnya dengan menampilkan kotak yang dibuat menggunakan tag `<div>`. Fitur seperti ini lebih mudah dibuat menggunakan library JavaScript seperti **jQuery**.

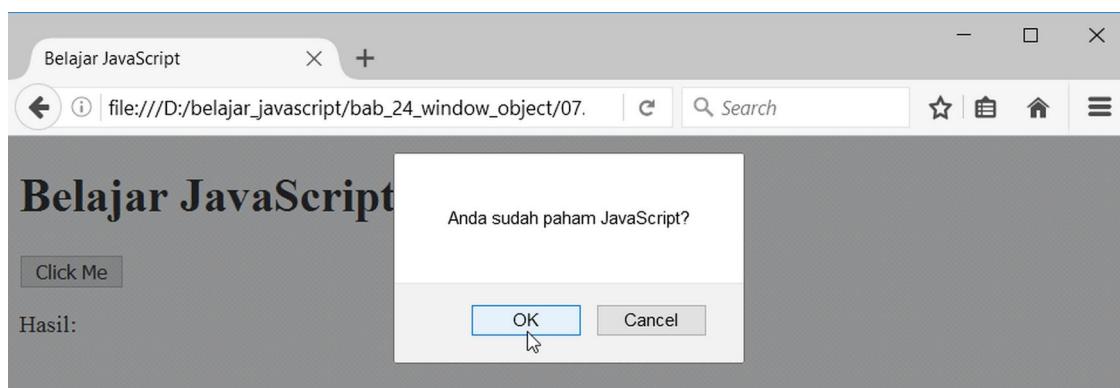
## 24.3 Method Window.confirm()

Method `window.confirm()` mirip seperti `alert()`, dimana akan tampil sebuah jendela *modal windows*. Bedanya, jendela `confirm()` menawarkan 2 buah pilihan tombol: “OK” dan “Cancel”.

Jika tombol “OK” di klik, method ini akan mengembalikan nilai boolean `true`. Jika tombol “Cancel” di klik atau jendela `confirm` di tutup dengan men-klik tombol x di sudut kanan atas, method ini akan mengembalikan nilai boolean `false`.

Method `window.confirm()` membutuhkan sebuah argumen bertipe data string yang berfungsi sebagai teks pertanyaan dari jendela `confirm`. Berikut contoh penggunaannya:

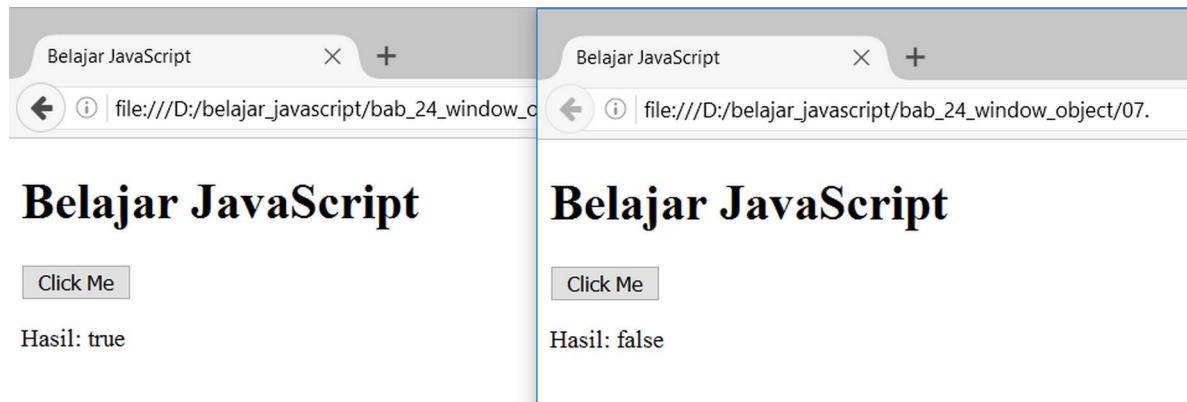
```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <button id="tombol">Click Me</button>
4   <p>Hasil: <span id="hasil"></span></p>
5   <script>
6     var tombolNode = document.getElementById("tombol");
7     var hasilNode = document.getElementById("hasil");
8
9     function jalankan(){
10       var sudahPaham = confirm("Anda sudah paham JavaScript?");
11       hasilNode.innerHTML = sudahPaham;
12     }
13
14     tombolNode.addEventListener("click",jalankan);
15   </script>
16 </body>
```



Gambar: Tampilan jendela `window.confirm()`

Kode program untuk menampilkan jendela confirm dibuat dalam perintah `var sudahPaham = confirm("Anda sudah paham JavaScript?")`. Hasilnya berupa sebuah jendela dengan pertanyaan "Anda sudah paham JavaScript?".

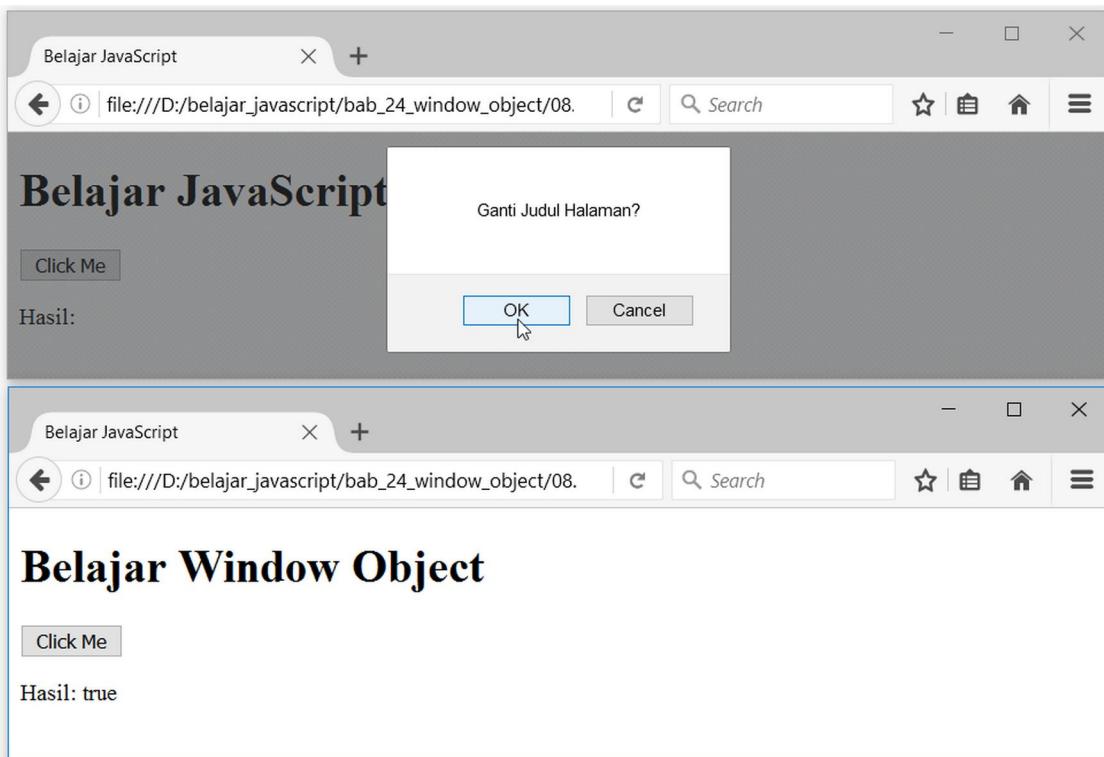
Jika dipilih **OK**, variabel `sudahPaham` akan berisi boolean true. Jika di klik tombol **Cancel**, variabel `sudahPaham` akan berisi boolean false.



Gambar: Hasil setelah tombol **OK** di klik (true), dan setelah tombol **Cancel** di klik (false)

Hasil dari method `window.confirm()` yang berupa nilai **true** atau **false** ini bisa kita proses lebih lanjut, seperti contoh berikut:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <button id="tombol">Click Me</button>
4   <p>Hasil: <span id="hasil"></span></p>
5   <script>
6     var tombolNode = document.getElementById("tombol");
7     var hasilNode = document.getElementById("hasil");
8     var h1Node = document.querySelector("h1");
9
10    function jalankan(){
11      var diGanti = confirm("Ganti Judul Halaman?");
12      hasilNode.innerHTML = diGanti;
13
14      if (diGanti) {
15        h1Node.innerHTML = "Belajar Window Object";
16      }
17    }
18
19    tombolNode.addEventListener("click",jalankan);
20  </script>
21 </body>
```



Gambar: Judul tag <h1> akan terganti saat di klik tombol OK di jendela konfirmasi

Saya memodifikasi sedikit kode sebelumnya. Sekarang ketika tombol **OK** dari jendela confirm di klik, teks di dalam tag <h1> akan berubah. Variabel `diGanti` akan berisi hasil dari perintah `confirm("Ganti Judul Halaman?")`. Jika tombol **OK** di klik, variabel `diGanti` akan berisi boolean **true**.

Selanjutnya, saya membuat sebuah kondisi **if**, yakni jika `diGanti` bernilai **true**, jalankan perintah `h1Node.innerHTML = "Belajar Window Object"`. Hasilnya, isi teks dari tag <h1> akan berubah menjadi "Belajar Window Object".

Pemeriksaan kondisi variabel `diGanti` juga bisa langsung kita akses langsung dari method `confirm`, sehingga tidak perlu lagi menggunakan variabel `diGanti`, seperti contoh berikut:

```

1 function jalankan(){
2   if (confirm("Ganti Judul Halaman?")) {
3     h1Node.innerHTML = "Belajar Window Object";
4   }
5 }
```

Sekarang, kondisi **if** langsung memeriksa hasil dari method `window.confirm()`. Jika nilai yang dihasilkan adalah boolean **true** (tombol **OK** di klik), ubah isi property `h1Node.innerHTML` menjadi "Belajar Window Object".

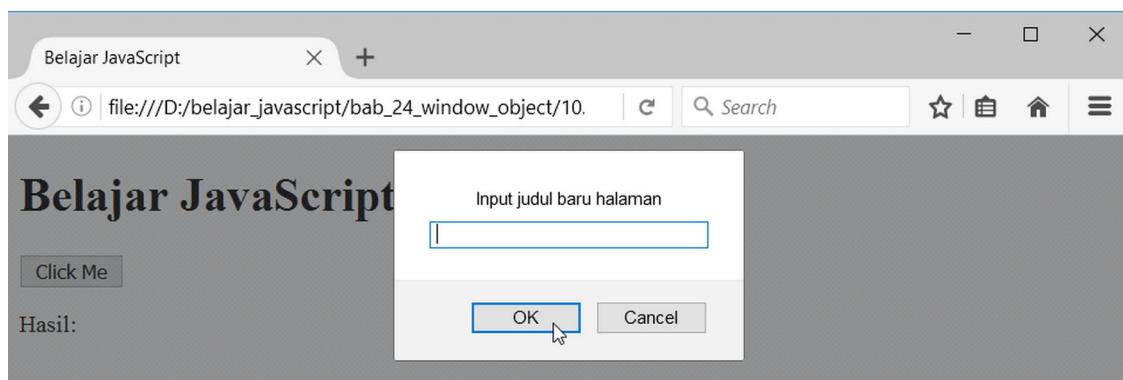
Jendela confirm cocok untuk pertanyaan konfirmasi, seperti apakah user yakin ingin menghapus sebuah data atau tidak. Namun karena efeknya yang cukup mengganggu dan dalam beberapa web browser *modal windows* seperti ini akan diblokir, alternatifnya bisa dengan membuat jendela konfirmasi sendiri menggunakan DOM.

## 24.4 Method Window.prompt()

Method `window.prompt()` digunakan untuk membuat jendela *modal windows* dimana pengguna bisa menginput sebuah teks. Sama seperti method `alert()` dan `confirm()`, method `prompt()` juga membutuhkan 1 argumen berupa pertanyaan yang akan muncul di dalam jendela.

Berikut contoh penggunaan dari `window.prompt()`:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <button id="tombol">Click Me</button>
4   <p>Hasil: <span id="hasil"></span></p>
5   <script>
6     var tombolNode = document.getElementById("tombol");
7     var h1Node = document.querySelector("h1");
8
9     function jalankan(){
10       var judulBaru = prompt("Input judul baru halaman");
11       h1Node.innerHTML = judulBaru;
12     }
13
14     tombolNode.addEventListener("click",jalankan);
15   </script>
16 </body>
```



Gambar: Tampilan jendela hasil `window.prompt()`

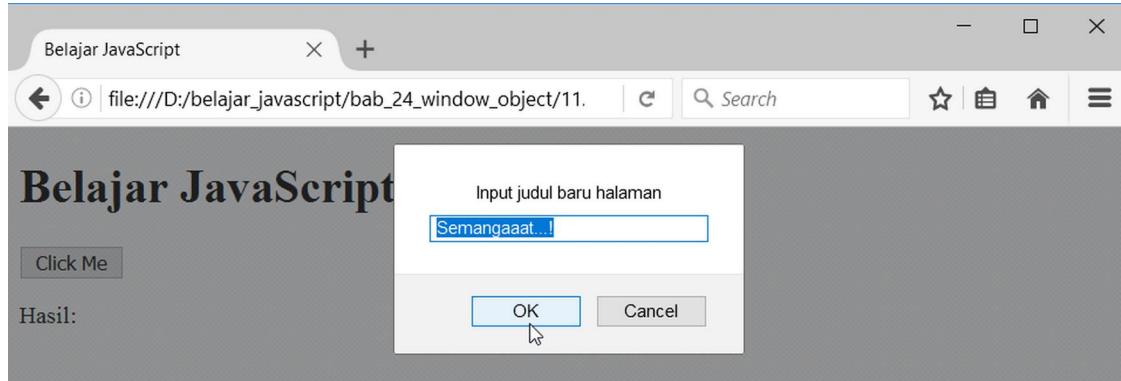
Ketika tombol **Click Me** di klik, akan tampil sebuah jendela input. Teks pertanyaan berasal dari argument pada saat method `prompt()` dipanggil, dalam kode JavaScript saya membuatnya dengan perintah `prompt("Input judul baru halaman")`.

Teks yang diinput ke dalam jendela `prompt()` akan menjadi nilai kembalian dari method tersebut. Dalam kode program, nilai ini disimpan ke dalam variabel `judulBaru`. Kemudian variabel `judulBaru` diinput kembali ke dalam `h1Node.innerHTML`. Akibatnya, isi teks tag `<h1>` akan berubah sesuai dengan isi text box.

Method `window.prompt()` juga memiliki argumen kedua yang bersifat opsional. Argumen ini akan menjadi teks *placeholder* atau teks awal dari inputan. Sebagai contoh, saya akan mengubah pemanggilan method `window.prompt()` dalam kode program sebelumnya menjadi seperti ini:

```
var judulBaru = prompt("Input judul baru halaman", "Semangaaat...!");
```

Pada saat tombol **Click Me** di klik, akan tampil jendela berikut:



Gambar: Teks inputan memiliki nilai awal "Semangaaat...!"

Terlihat bahwa teks "Semangaaat...!" akan menjadi teks awal dari inputan jendela `prompt()`.

Sama seperti method `alert()` dan `confirm()`, jendela `prompt()` juga kadang di blokir oleh web browser. Untuk proyek offline seperti tugas kampus / skripsi / web lokal, anda masih boleh menggunakan ketiga method ini.

Namun untuk web online, sebaiknya menggunakan cara lain dengan membuat sendiri *modal windows* menggunakan DOM. Kode programnya memang akan rumit. Sebagai alternatif, bisa memakai library JavaScript [jQuery UI Dialog box<sup>3</sup>](#), tapi tentunya setelah anda paham cara penggunaan **jQuery**.

## 24.5 Method Window.print()

Method `Window.print()` berguna untuk menampilkan jendela print yang sama seperti kita men-klik tombol **menu -> print**, atau tombol **CRTL + P**.

Penggunaannya juga sangat mudah, tinggal menjalankan perintah `window.print()` seperti contoh berikut:

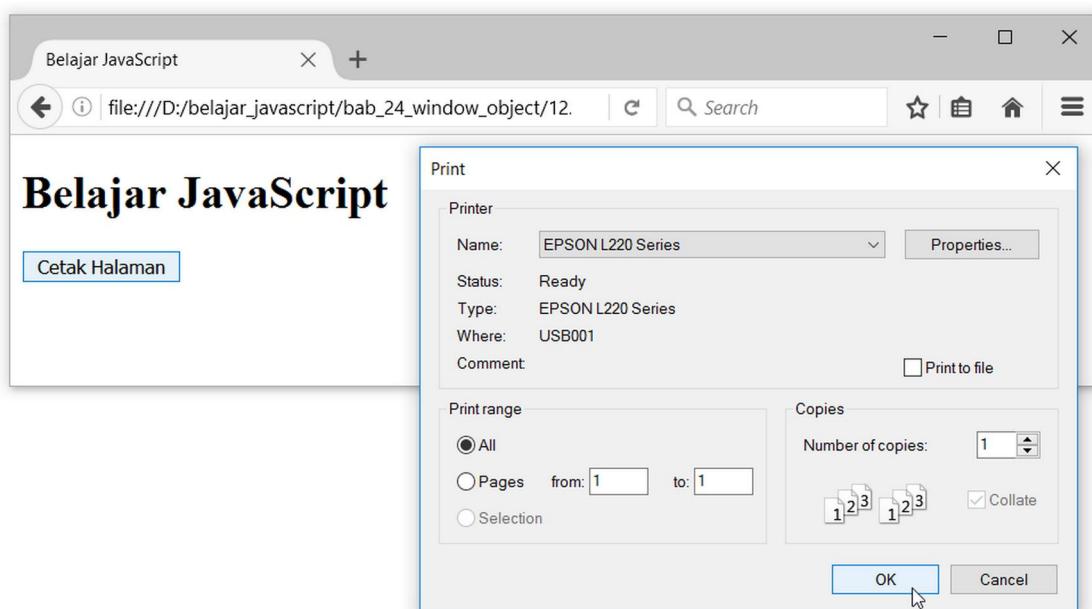
---

<sup>3</sup><https://jqueryui.com/dialog/>

```

1 <body>
2   <h1>Belajar JavaScript</h1>
3   <button id="tombol">Cetak Halaman</button>
4   <script>
5     var tombolNode = document.getElementById("tombol");
6
7     tombolNode.addEventListener("click", function () {
8       window.print();
9     });
10    </script>
11 </body>

```



Gambar: Jendela Print

Di dalam kode program JavaScript, saya membuat event **click** untuk tombol **Cetak Halaman**. Saat tombol ini di klik, jalankan perintah `window.print()`. Hasilnya, akan tampil jendela print.

Sebagai alternatif, berikut kode JavaScript inline untuk menghasilkan tampilan yang sama:

```

1 <body>
2   <h1>Belajar JavaScript</h1>
3   <button id="tombol" onclick="print();">Cetak Halaman</button>
4 </body>

```

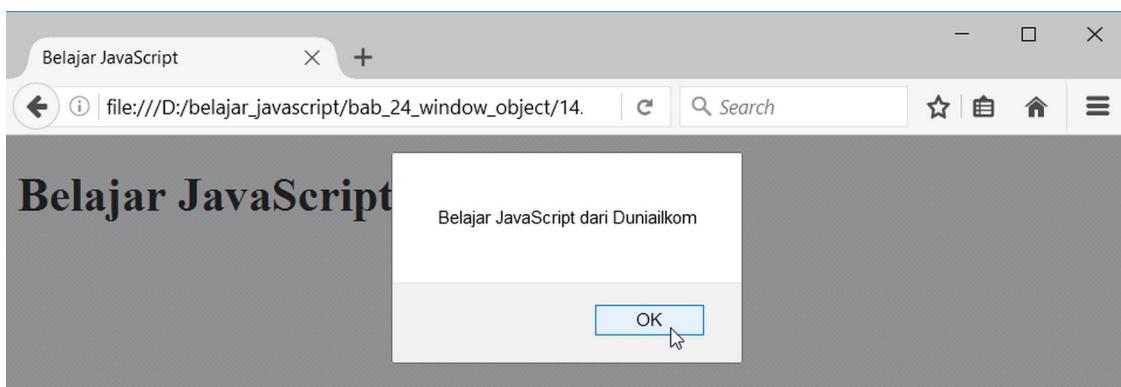
## 24.6 Method Window.setTimeout()

Method `window.setTimeout()` digunakan untuk menunda eksekusi kode JavaScript beberapa saat. Method ini membutuhkan dua argumen:

- Argumen pertama diisi dengan nama function yang akan dijalankan.
- Argumen kedua berupa angka dalam satuan milidetik kapan function akan berjalan.

Sebagai contoh, jika saya ingin menampilkan jendela `alert()` setelah 3 detik, bisa menggunakan kode program berikut:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <script>
4     function pesan(){
5       alert("Belajar JavaScript dari DuniaIlkom");
6     }
7
8     setTimeout(pesan,3000);
9   </script>
10 </body>
```



Gambar: Jendela alert akan tampil setelah 3 detik

Pada saat kode program berjalan, jendela `alert("Belajar JavaScript dari DuniaIlkom")` akan tampil setelah menunggu selama 3000 milidetik, atau setelah 3 detik. Inilah kegunaan dari method `window.setTimeout()`.

Perintah `setTimeout(pesan,3000)` artinya, jalankan function `pesan()` setelah 3000 milidetik. Di dalam function `pesan()` inilah saya membuat perintah `alert()`.

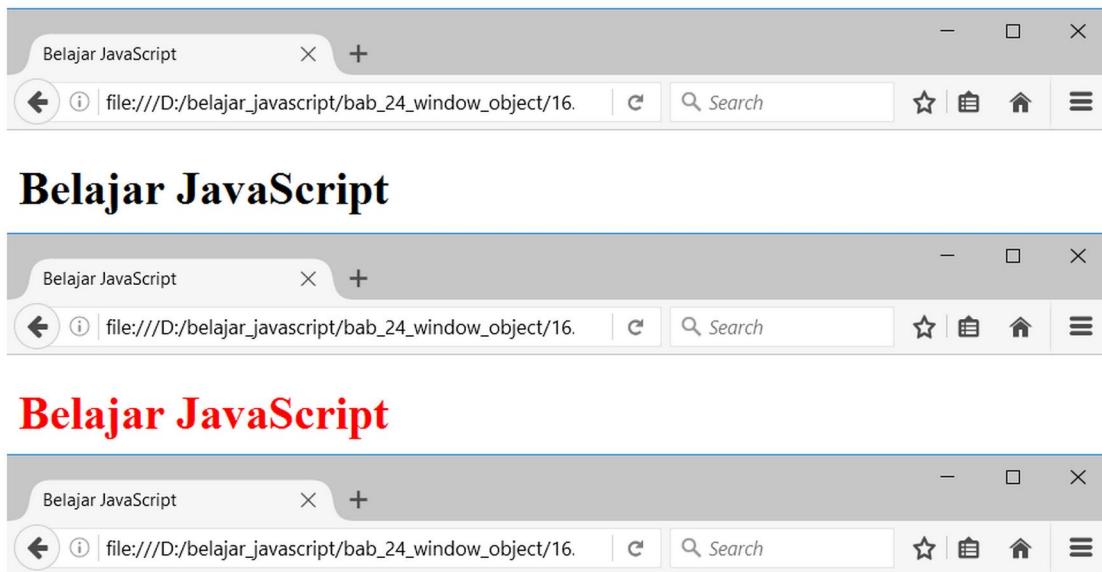
Jika saya ingin menjalankan perintah lain, tinggal mengganti isi dari function `pesan()`:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <script>
4     h1Node = document.querySelector("h1");
5
6     function gantiJudul(){
7       h1Node.innerHTML = "Belajar Method window.setTimeout()";
8     }
9
10    setTimeout(gantiJudul,3000);
11  </script>
12 </body>
```

Setelah 3 detik, isi dari tag `<h1>` akan berubah menjadi "Belajar Method `window.setTimeout()`".

Sebagai latihan selanjutnya, saya ingin membuat efek animasi. Setelah 2 detik, ubah warna teks `<h1>` menjadi **merah**. Dua detik kemudian, ubah warna teks `<h1>` menjadi **biru**. Berikut kode programnya:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <script>
4     h1Node = document.querySelector("h1");
5
6     function ubahWarna(){
7       h1Node.style.color = "red";
8     }
9
10    function ubahWarna2(){
11      h1Node.style.color = "blue";
12    }
13
14    setTimeout(ubahWarna,2000);
15    setTimeout(ubahWarna2,4000);
16  </script>
17 </body>
```



Gambar: Pergantian warna tag <h1>

Disini saya menjalankan 2 buah method `window.setTimeout()`, yakni `setTimeout(ubahWarna, 2000)` dan `setTimeout(ubahWarna2, 4000)`.

Setelah 2 detik, function `ubahWarna()` akan dipanggil. Isi dari function ini berupa perintah `h1Node.style.color = "red"`. Hasilnya, warna teks `<h1>` akan bertukar menjadi merah (red).

Dua detik kemudian (4 detik sejak file di jalankan), giliran function `ubahWarna2()` yang akan dipanggil. Isinya berupa perintah `h1Node.style.color = "blue"`. Ini akan mengubah warga tag `<h1>` menjadi biru.

## 24.7 Method Window.clearTimeout()

Method `window.clearTimeout()` digunakan untuk menghapus efek dari `setTimeout()`.

Method `setTimeout()` yang kita bahas sebelumnya ternyata mengembalikan sebuah nilai. Nilai tersebut berupa `timeoutID` yang bisa diinput ke dalam method `clearTimeout()`.

Dengan menggunakan `clearTimeout()`, kita bisa membatalkan jadwal eksekusi program yang sudah di buat dengan `setTimeout()`.

Berikut contoh penggunaannya:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <script>
4     h1Node = document.querySelector("h1");
5
6     function ubahWarna(){
7       h1Node.style.color = "red";
8     }
9
10    var timeoutID = setTimeout(ubahWarna,4000);
11    console.log(timeoutID);
12    clearTimeout(timeoutID);
13  </script>
14 </body>
```

Perhatikan perintah `var timeoutID = setTimeout(ubahWarna,4000)`, disini saya menginput kembalian dari method `setTimeout()` ke dalam variabel `timeoutID`.

Selanjutnya, terdapat perintah `clearTimeout(timeoutID)`. Ini akan menghapus efek dari method `setTimeout()` sebelumnya. Jika tab console dibuka, akan terlihat angka yang menjadi isi dari `timeoutID`.

Sebagai latihan, anda bisa membuat sebuah tombol “Stop” yang jika ditekan, method `setInterval()` tidak jadi berjalan. Namun jika dibiarkan saja, tag `<h1>` akan berubah warna menjadi merah.

## 24.8 Method Window.setInterval()

Method `window.setInterval()` digunakan untuk membuat jadwal pemanggilan sebuah function secara terus menerus. Misalnya saya ingin menampilkan pesan `alert()` setiap 5 detik sekali.

Penggunaan method `setInterval()` sangat mirip seperti method `setTimeout()`, seperti contoh berikut:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <script>
4     function pesan(){
5       alert("Belajar JavaScript dari DuniaIlkom");
6     }
7
8     setInterval(pesan,3000);
9   </script>
10 </body>
```

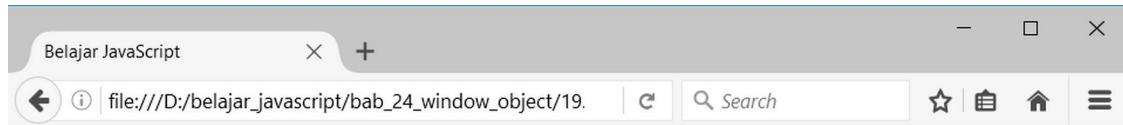
Kode program diatas sama persis seperti contoh pada `setTimeout()`, bedanya, kali ini method `setTimeout()` saya ganti menjadi `setInterval()`. Akibat pergantian ini, pesan `alert()` akan tampil terus menerus setiap 3 detik.

Untuk contoh yang lebih bermanfaat, saya ingin mengubah isi tag <h1> setiap 1 detik:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <script>
4     h1Node = document.querySelector("h1");
5
6     function cekTanggal(){
7       h1Node.innerHTML = new Date();
8     }
9
10    setInterval(cekTanggal,1000);
11  </script>
12 </body>
```

Kali ini, function cekTanggal() akan dijalankan setiap 1 detik. Isinya berupa 1 baris perintah h1Node.innerHTML = new Date().

Artinya, teks dari tag <h1> akan diisi dengan tanggal hari ini. Perintah tersebut akan dipanggil setiap 1 detik, sehingga tanggal yang tampil juga akan “bergerak” menyesuaikan dengan tanggal yang ada di komputer:



## Thu Jan 26 2017 14:41:36 GMT+0700 (SE Asia Standard Time)

Gambar: Menampilkan tanggal dengan method setInterval()

Sebagai latihan, bisakah anda membuat kode program yang menampilkan waktu saja (dalam format jam, menit dan detik)? Berikut kode yang saya gunakan:

```
1 <body>
2   <h1></h1>
3   <script>
4     h1Node = document.querySelector("h1");
5
6     function cekTanggal(){
7       var myDate = new Date();
8
9       var jam = myDate.getHours();
10      var menit = myDate.getMinutes();
11      var detik = myDate.getSeconds();
```

```
12
13     h1Node.innerHTML = "Waktu saat ini = "+jam+":"+menit+":"+detik;
14 }
15
16 cekTanggal();
17 setInterval(cekTanggal,1000);
18 </script>
19 </body>
```



## Waktu saat ini = 14:58:19

Gambar: Menampilkan waktu dengan method setInterval()

Prinsip kerja program diatas sama seperti sebelumnya. Hanya saja kali ini saya menggunakan beberapa method dari **date object** untuk mengambil nilai jam, menit, dan detik dari **date()**. Hasilnya kemudian diinput ke dalam tag **<h1>**.

Saya juga memanggil function **cekTanggal()** di luar **setInterval()**. Ini dilakukan agar tag **<h1>** langsung berisi nilai saat file dijalankan pertama kali. Jika tidak ditulis seperti ini, tag **<h1>** akan kosong selama 1 detik menunggu “panggilan” dari method **setInterval()**.

Method **setInterval()** juga bisa digunakan untuk membuat efek animasi. Misalnya dengan mengganti warna teks secara terus menerus, seperti contoh berikut:

```
1 <body>
2   <h1>Pesan sekarang juga, senin harga naik!!</h1>
3   <script>
4     h1Node = document.querySelector("h1");
5
6     function buatEfek(){
7       if (h1Node.style.color === "red"){
8         h1Node.style.color = "blue";
9       }
10      else {
11        h1Node.style.color = "red";
12      }
13    }
14
15    setInterval(buatEfek,200);
16  </script>
17 </body>
```



Gambar: Efek perubahan warna terus-menerus

Hasil dari kode program diatas, warna teks <h1> akan berubah-ubah dari merah ke biru setiap 0,2 detik. Silahkan anda berkreasi dengan mengubah efek CSS lain, seperti warna background atau ukuran font.

## 24.9 Method Window.clearInterval()

Untuk menghentikan `setInterval()`, kita bisa menggunakan method `window.clearInterval()`.

Penggunaannya sama persis seperti method `window.clearTimeout()`, dimana kita tinggal memberikan `clearIntervalID` ke dalam argumen method `clearInterval()`.

Berikut contoh penggunannya:

```
1 <body>
2   <h1>Pesan sekarang juga, senin harga naik!!</h1>
3   <button id="stop">Stop Animasi</button>
4   <script>
5     h1Node = document.querySelector("h1");
6     stopNode = document.getElementById("stop");
7
8     function buatEfek(){
9       if (h1Node.style.color === "red"){
10         h1Node.style.color = "blue";
11       }
12       else {
13         h1Node.style.color = "red";
14       }
15     }
16
17     var setIntervallID = setInterval(buatEfek,200);
18
19     stopNode.addEventListener("click",function(){
```

```
20     clearInterval(setIntervalID);
21   });
22 </script>
23 </body>
```



Gambar: Menghentikan efek setInterval() dengan clearInterval()

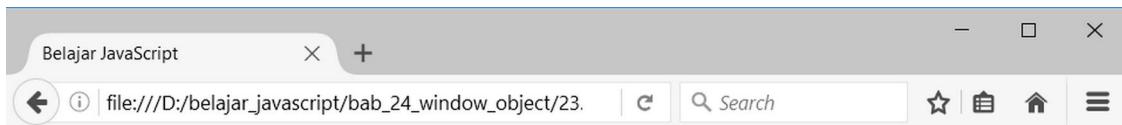
Kode program ini merupakan modifikasi dari kode sebelumnya. Saya menambahkan tombol “Stop Animasi” yang jika di klik akan menghentikan efek perubahan warna dari setInterval().

## 24.10 Method Window.open()

Sesuai dengan namanya, **Window object** berurusan dengan jendela. Karena itu kita juga bisa membuat jendela baru menggunakan method `window.open()`.

Method `window.open()` memiliki 3 buah argument opsional yang akan saya bahas secara bertahap. Pertama, jika method ini dipanggil tanpa argumen apapun, hasilnya berupa sebuah tab baru di web browser, seperti contoh berikut:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <button id="tombol">Buat jendela baru</button>
4   <script>
5     var tombolNode = document.getElementById("tombol");
6
7     function jendelaBaru(){
8       window.open();
9     }
10
11    tombolNode.addEventListener("click",jendelaBaru);
12 </script>
13 </body>
```



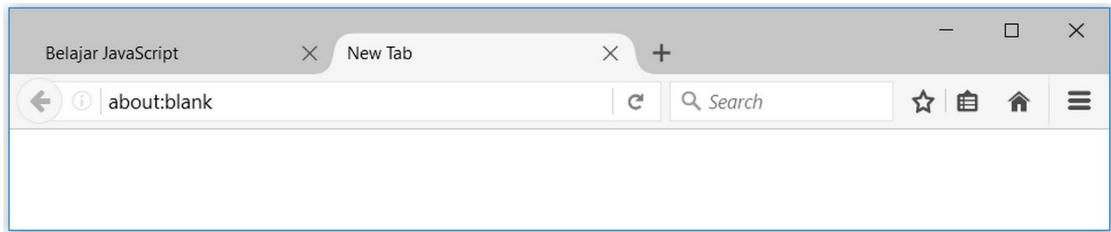
## Belajar JavaScript

Buat jendela baru

Gambar: Tombol "Buat jendela baru"

Dalam kode program diatas, saya membuat sebuah tombol “**Buat jendela baru**”. Saat tombol ini di klik, jalankan function `jendelaBaru()`. Isinya 1 baris perintah `window.open()`.

Di dalam web browser modern (yang menggunakan sistem tab) akan keluar sebuah tab baru:

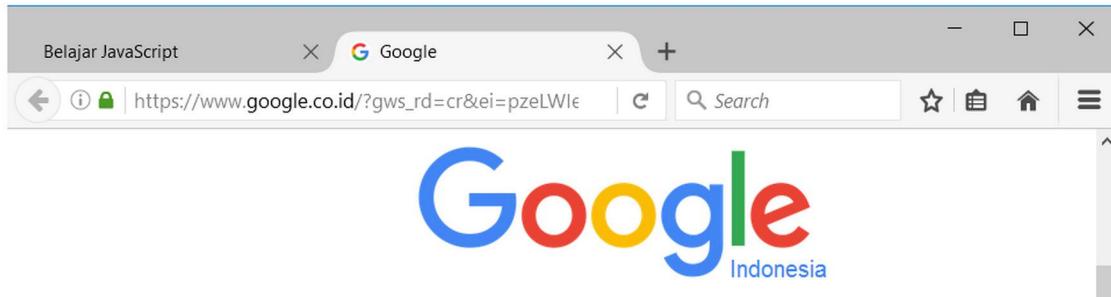


Gambar: Tab baru terbuka di web browser

Tab yang tampil berisi halaman kosong. Untuk “mengisi” tab tersebut, kita bisa menginput 1 argumen ke dalam method `window.open()`, seperti contoh berikut:

```
1 function jendelaBaru(){
2     window.open("http://www.google.com");
3 }
```

Sekarang, ketika tombol “**Buat jendela baru**” di klik, tab baru akan terbuka dan langsung menampilkan halaman [www.google.com](http://www.google.com):



Gambar: Tab baru berisi halaman situs google

Argumen pertama dari method `window.open()` ini tidak harus berupa alamat website online, tapi juga bisa file HTML lain:

```

1 function jendelaBaru(){
2   window.open("21.setInterval_style.html");
3 }
```

Saat tombol “**Buat jendela baru**” di klik, tab baru akan terbuka dan menampilkan isi dari file `21.setInterval_style.html`. Dalam istilah HTML, argument ini bisa diisi dengan *alamat relatif* maupun *alamat absolute* (perbedaan keduanya pernah saya bahas di buku **HTML Uncover**).

Sekarang kita masuk ke argumen kedua dari method `window.open()`. Argumen kedua berfungsi sebagai **WindowName**, yakni nama jendela dari tab yang terbuka. **WindowName** hanya digunakan secara internal oleh JavaScript dan tidak berefek apapun ke tampilan.

Sebagai contoh, saya bisa menggunakan kode program berikut:

```

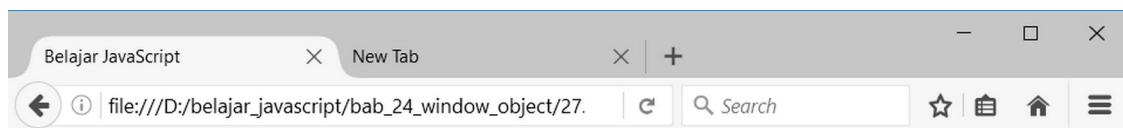
1 function jendelaBaru(){
2   window.open("21.setInterval_style.html", "jendelaBaru");
3 }
```

Tab yang terbuka tidak akan berbeda dengan tampilan sebelumnya. Jadi apa fungsi argumen kedua dari method `window.open()`?

String “`jendelaBaru`” bisa dijadikan sebagai nilai atribut `target` dari tag `<a>`, seperti contoh berikut:

```

1 <body>
2   <h1>Belajar JavaScript</h1>
3   <button id="tombol">Buat jendela baru</button>
4   <a href="21.setInterval_style.html" target="jendelaBaru">Load file...</a>
5   <script>
6     var tombolNode = document.getElementById("tombol");
7
8     function jendelaBaru(){
9       window.open("", "jendelaBaru");
10    }
11
12    tombolNode.addEventListener("click", jendelaBaru);
13  </script>
14 </body>
```



## Belajar JavaScript

[Buat jendela baru](#) [Load file...](#)

file:///D:/belajar\_javascript/bab\_24\_window\_object/27.

Gambar: Contoh penggunaan argumen kedua method `window.open()`

Perhatikan isi dari function jendelaBaru(), yakni sebuah perintah `window.open("", "jendelaBaru")`. Argumen pertama saya isi dengan string kosong, artinya tab baru akan tampil kosong (blank). Argumen kedua diisi dengan string "jendelaBaru".

Di dalam kode HTML, saya menambahkan 1 buah tag `<a>` dengan penulisan sebagai berikut:

```
<a href="21.setInterval_style.html" target="jendelaBaru">Load file...</a>
```

Atribut `href` berisi "21.setInterval\_style.html" sedangkan atribut `target` berisi string "jendelaBaru". Artinya, ketika link "Load file..." di klik, isi file 21.setInterval\_style.html akan dikirim ke dalam tab yang bernama "jendelaBaru".

Seperti yang terlihat, kita bisa mengontrol apa isi sebuah tab menggunakan atribut `target`. Cara ini hanya berlaku untuk tab yang dibuat dari JavaScript dan berasal dari domain yang sama.

Karena alasan keamanan dan privasi, kita tidak bisa mengontrol isi tab yang di buka manual dari web browser atau tab yang isinya berasal dari situs yang berbeda. Pembatasan ini dikenal sebagai [Same-origin policy<sup>4</sup>](#).

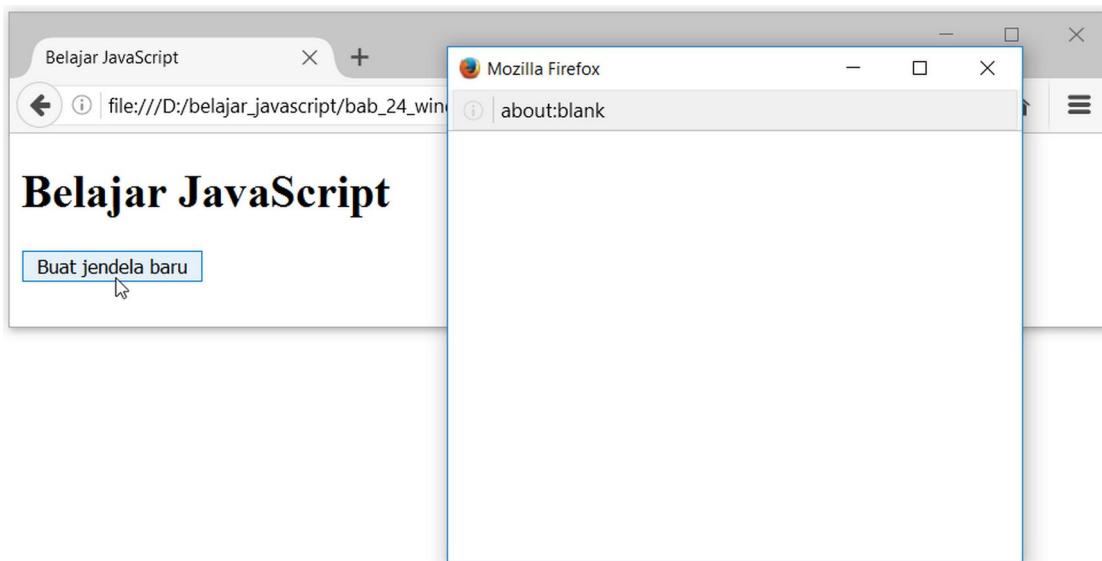
Argument ketiga dari method `window.open()` merupakan yang paling menarik. Kita bisa mengatur banyak hal terkait tab atau jendela yang dibuat, seperti lebar, tinggi, posisi tampilan, apakah memiliki toolbar, scrollbar, dll.

Pengaturan ini diinput sebagai sebuah string panjang yang dipisah dengan tanda koma. Sebagai contoh, berikut kode program untuk membuat jendela baru dengan lebar 400 pixel dan tinggi 300 pixel:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <button id="tombol">Buat jendela baru</button>
4   <script>
5     var tombolNode = document.getElementById("tombol");
6
7     function jendelaBaru(){
8       window.open("", "jendelaBaru", "width=400,height=300");
9     }
10
11    tombolNode.addEventListener("click", jendelaBaru);
12  </script>
13 </body>
```

---

<sup>4</sup>[https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin\\_policy](https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy)



Gambar: Jendela baru dengan lebar 400 pixel dan tinggi 300 pixel

Saat tombol “**Buat jendela baru**” di klik, akan keluar sebuah jendela yang memiliki lebar 400 pixel dan tinggi 300 pixel. Lebar dan tinggi jendela ini diatur dari argumen ketiga method `window.open()`, yakni string `"width=400,height=300"`.

Dapat anda lihat bahwa kali ini yang terbuka bukan lagi sebuah tab baru, tapi jendela baru. Jendela seperti ini sering disebut sebagai jendela pop up (*pop up window*).

Selain lebar dan tinggi jendela, kita juga bisa mengubah settingan lain. Jika saya ingin jendela tersebut tampil di posisi 300 pixel dari atas, bisa menambahkan pengaturan `top=300`:

```
1 function jendelaBaru(){
2     window.open("", "jendelaBaru", "width=400,height=300,top=300");
3 }
```

Atau jika ingin diposisikan 300 pixel dari atas dan 500 pixel dari kiri layar, bisa menambahkan pengaturan `left=500`:

```
1 function jendelaBaru(){
2     window.open("", "jendelaBaru", "width=400,height=300,top=300,left=500");
3 }
```

Agar lebih rapi, penulisan argumen ini bisa disimpan ke dalam variabel terpisah:

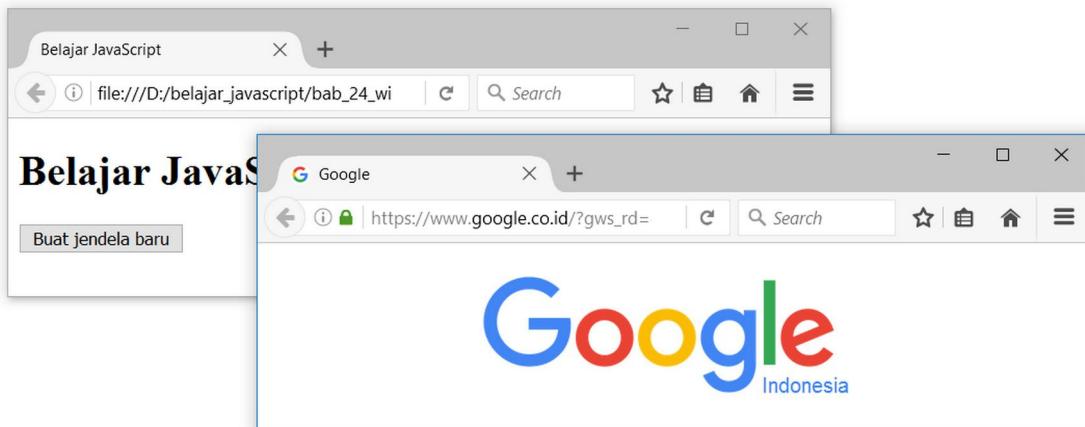
```
1 function jendelaBaru(){
2     var setting = "width=800,height=300,top=300,left=300";
3     window.open("http://www.google.com", "jendelaBaru", setting);
4 }
```

Jendela pop up yang tampil tampak tidak memiliki **toolbar**. Ini bisa diatur dengan tambahan settingan `toolbar=yes`:

```

1 function jendelaBaru(){
2   var setting = "width=800,height=300,top=300,left=300,toolbar=yes";
3   window.open("http://www.google.com", "jendelaBaru", setting);
4 }

```



Gambar: Popup window dengan toolbar

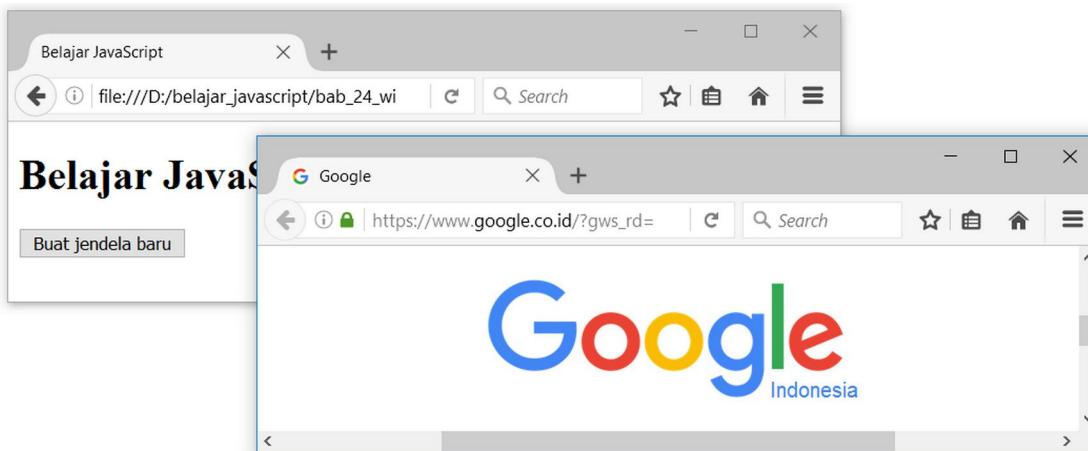
Dengan tambahan string `toolbar=yes`, jendela popup yang terbuka akan memiliki toolbar. Selain itu saya menambahkan string `http://www.google.com` ke dalam argumen pertama, sehingga saat terbuka langsung menampilkan halaman google.

Pengaturan lain yang bisa disetting adalah `scrollbar`. Secara default, jendela baru ditampilkan tanpa toolbar. Ini bisa dibat dengan `scrollbars=yes`:

```

1 function jendelaBaru(){
2   var setting = "width=800,height=300,toolbar=yes,scrollbars=yes";
3   window.open("http://www.google.com", "jendelaBaru", setting);
4 }

```



Gambar: Popup window dengan toolbar dan scrollbar

Sekarang, jendela popup akan tampil lengkap dengan **toolbar** dan **scrollbar**.

Selain pengaturan yang saya bahas, masih ada settingan lain yang bisa diinput ke dalam argumen ketiga dari method `window.open()`. Lengkapnya bisa anda akses ke [Window.open\(\) - Mozilla Developer Network<sup>5</sup>](#).

Sebagian besar dari pengaturan tersebut tidak berefek atau tidak bisa berjalan sama sekali, karena fitur dan dukungan setiap web browser terhadap setingan yang ada bisa berbeda-beda.

## 24.11 Method Window.close()

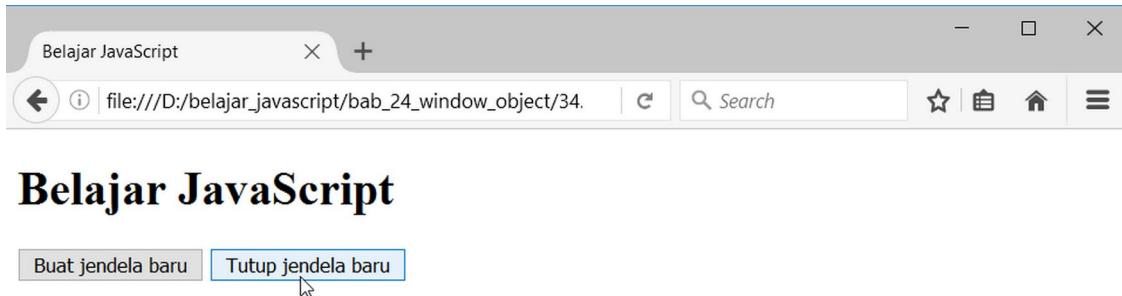
Dilihat dari namanya, method `window.close()` digunakan untuk menutup jendela. Dan itulah kegunaan dari method `window.close()`. Jendela yang sebelumnya kita buka menggunakan `window.open()` bisa ditutup menggunakan method `window.close()`.

Berikut contoh penggunaan dari method ini:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <button id="tombolBuka">Buat jendela baru</button>
4   <button id="tombolTutup">Tutup jendela baru</button>
5   <script>
6     var tombolBukaNode = document.getElementById("tombolBuka");
7     var tombolTutupNode = document.getElementById("tombolTutup");
8     var myJendela;
9
10    function jendelaBaru(){
11      var setting = "width=500,height=300,toolbar=yes";
12      myJendela = window.open("http://www.google.com", "jendelaBaru", setting);
13    }
14
15    function tutupJendela(){
16      myJendela.close();
17    }
18
19    tombolBukaNode.addEventListener("click", jendelaBaru);
20    tombolTutupNode.addEventListener("click", tutupJendela);
21  </script>
22 </body>
```

---

<sup>5</sup>[https://developer.mozilla.org/en-US/docs/Web/API/Window/open#Window\\_features](https://developer.mozilla.org/en-US/docs/Web/API/Window/open#Window_features)



Gambar: Tombol Tutup jendela baru yang akan menjalankan method `window.close()`

Saya memodifikasi kode program sebelumnya dengan menambahkan tombol “**Tutup jendela baru**”.

Di dalam function `jendelaBaru()` juga terdapat sedikit perubahan. Variabel `myJendela` saya buat untuk menampung hasil pemanggilan method `window.open()`. Variabel `myJendela` ini berisi sebuah `windowObjectReference`, yakni referensi dari jendela yang baru saja dibuat.

Dengan mengakses `windowObjectReference`, kita bisa mengontrol jendela baru, misalnya menutup jendela tersebut dengan menjalankan perintah `myJendela.close()`. Dan beberapa hal lain yang akan kita pelajari setelah ini.

Silahkan anda klik tombol “**Buat jendela baru**” untuk membuka jendela popup, kemudian klik tombol “**Tutup jendela baru**” untuk menutup jendela tersebut.

## 24.12 Case Study: Mengubah Konten Window

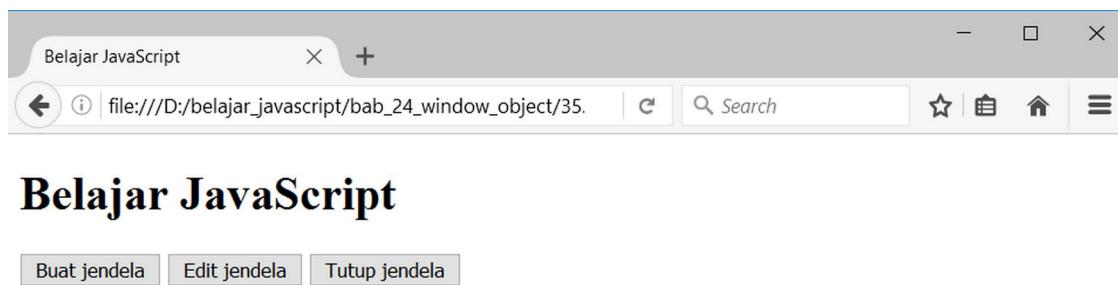
Dalam pembahasan tentang method `window.close()` diatas, saya menggunakan variabel `myJendela` untuk mengakses jendela yang dibuka dengan perintah `window.open()`.

Variabel `myJendela` sebenarnya merupakan **window object** dari jendela popup. Sehingga kita bisa mengakses **DOM** dan **BOM** milik jendela tersebut.

Sebagai contoh, silahkan anda pelajari sejenak kode program dibawah ini:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <button id="tombolBuka">Buat jendela</button>
4   <button id="tombolEdit">Edit jendela</button>
5   <button id="tombolTutup">Tutup jendela</button>
6   <script>
7     var tombolBukaNode = document.getElementById("tombolBuka");
8     var tombolEditNode = document.getElementById("tombolEdit");
9     var tombolTutupNode = document.getElementById("tombolTutup");
10    var myJendela;
11
12    function jendelaBaru(){
13      var setting = "width=900,height=300,toolbar=yes";
14      myJendela = window.open("http://www.google.com", "jendelaBaru", setting);
```

```
15      }
16
17      function editJendela(){
18          myJendela.location = "http://developer.mozilla.org";
19      }
20
21      function tutupJendela(){
22          myJendela.close();
23      }
24
25      tombolBukaNode.addEventListener("click",jendelaBaru);
26      tombolEditNode.addEventListener("click",editJendela);
27      tombolTutupNode.addEventListener("click",tutupJendela);
28  </script>
29 </body>
```



Gambar: Mengakses jendela baru

Dalam kode program diatas, saya membuat 3 buah tombol:

- **Buat jendela:** digunakan untuk membuat jendela baru menggunakan method `window.open()`. Variabel `myJendela` berisi **window object** dari jendela tersebut.
- **Tutup jendela:** digunakan untuk menutup jendela menggunakan method `myJendela.close()`.
- **Edit jendela:** digunakan untuk mengubah isi dari jendela, dengan cara mengakses variabel `myJendela`.

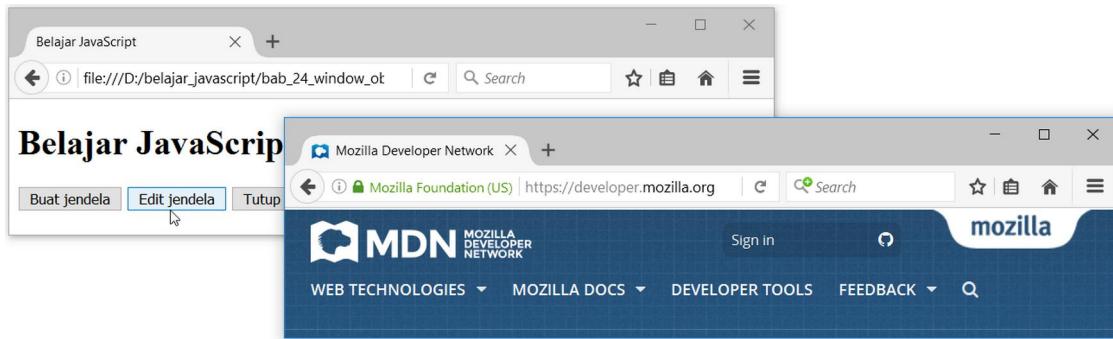
Di dalam function `jendelaBaru()`, saya membuat variabel `myJendela` yang berisi nilai kembalian dari method `window.open()`. Variabel `myJendela` ini adalah **window object** milik jendela yang baru dibuat.

Silahkan anda klik tombol “Buat jendela”, akan tampil jendela popup yang berisi situs google. lalu klik tombol “Edit jendela”. Isi dari jendela tersebut akan berubah dari `google.com` menjadi `developer.mozilla.org`.

Perubahan ini dijalankan dengan perintah:

```
myJendela.location = "http://developer.mozilla.org";
```

Artinya, saya ingin mengubah isi **location object** milik `myJendela` ke situs `developer.mozilla.org`.



Gambar: Isi dari `myJendela` berubah dari `google.com` ke `developer.mozilla.org`

Karena `myJendela` adalah sebuah **window object**, kita juga bisa mengakses DOM dari jendela yang dibuat. Sebagai contoh, saya akan memodifikasi function `jendelaBaru()` dan `editJendela()` menjadi sebagai berikut:

```

1  function jendelaBaru(){
2      var setting = "width=600,height=300";
3      myJendela = window.open("","");
4      myJendela.name = "jendelaBaru";
5      myJendela.setting = setting;
6  }
7
8  function editJendela(){
9      myJendela.document.write("<h1>Belajar JavaScript dari DuniaIlkom</h1>");
10 }

```

Saat tombol “Buat jendela” di klik, akan tampil halaman kosong. Kemudian saat “tombol Edit” jendela di klik, isi jendela tersebut akan berisi tag `<h1>`:



Gambar: Mengisi jendela dengan tag `<h1>`

Teks yang berisi tag `<h1>` bisa sampai ke jendela baru dengan memanggil method `myJendela.document.write()`.

Untuk mengisi DOM ke jendela lain, halaman awal dari jendela tersebut harus kosong (blank), atau berisi konten dari domain yang sama. Jika pada saat terbuka jendela sudah berisi situs lain, kita tidak bisa menginput tag HTML seperti ini.

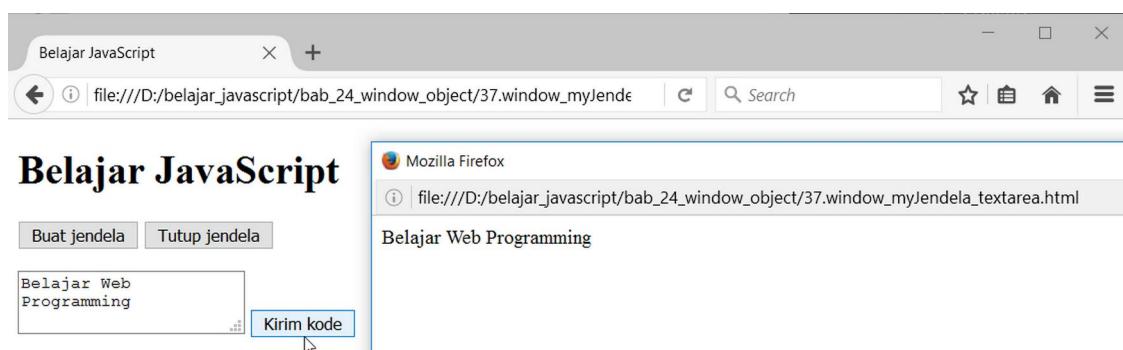
## 24.13 Case Study: Mengirim Teks ke Jendela Lain

Sebagai latihan terakhir, saya ingin mengirim teks ke jendela lain yang isinya berasal dari sebuah <textarea>. Textarea ini bisa diinput dengan teks apa saja, termasuk kode HTML. Ketika tombol “Kirim Kode” di klik, isi dari text area akan dikirim ke jendela tersebut.

Berikut tampilan akhir yang saya inginkan:



Gambar: Buat jendela baru



Gambar: Kirim teks ke jendela



Gambar: Kirim kode HTML ke jendela

Berikut kode program yang saya gunakan:

```
1 <body>
2   <h1>Belajar JavaScript</h1>
3   <button id="tombolBuka">Buat jendela</button>
4   <button id="tombolTutup">Tutup jendela</button>
5   <p>
6     <textarea id="kodeEdit"></textarea>
7     <button id="tombolKirim">Kirim kode</button>
8   </p>
9   <script>
10    var tombolBukaNode = document.getElementById("tombolBuka");
11    var tombolTutupNode = document.getElementById("tombolTutup");
12    var kodeEditNode = document.getElementById("kodeEdit");
13    var tombolKirimNode = document.getElementById("tombolKirim");
14    var myJendela;
15
16    function jendelaBaru(){
17      var setting = "width=800,height=300";
18      myJendela = window.open("", "jendelaBaru", setting);
19    }
20
21    function kirimkanKode(){
22      myJendela.document.write(kodeEditNode.value);
23    }
24
25    function tutupJendela(){
26      myJendela.close();
27    }
28
29    tombolBukaNode.addEventListener("click", jendelaBaru);
30    tombolKirimNode.addEventListener("click", kirimkanKode);
31    tombolTutupNode.addEventListener("click", tutupJendela);
32  </script>
33 </body>
```

Saya membuat tag `<textarea id="kodeEdit">` yang berfungsi sebagai penampung teks input. Proses pengiriman dilakukan ketika tombol “**Kirim kode**” di klik. Isinya, jalankan perintah `myJendela.document.write(kodeEditNode.value)`. Kode inilah yang akan mengirim nilai dari textarea ke `myJendela`.

Silahkan anda berkreasi dengan menggunakan method `window.open()`, `window.close()`, serta `document.write()`. Misalnya buat sebuah form yang ketika di submit hasilnya tampil di halaman baru. Atau buat 1 tombol yang jika di klik, akan tampil 3-4 jendela baru berisi situs favorit anda.

Namun sama seperti method `alert()`, `confirm()` dan `prompt()`, hampir semua web browser akan memblokir jendela popup jika ditempatkan ke sebuah web online. Ini karena popup windows banyak disalahgunakan untuk menampilkan iklan (sebagai spam).

Dalam bab ini kita telah membahas berbagai method dari **Window object** dan sebuah event **load**. Method `window.open()` juga sangat menarik untuk di eksplorasi. Namun karena sering digunakan untuk iklan popup yang menganggu, mayoritas web browser akan memblokirnya (khusus untuk website online).

Berikutnya, kita akan masuk ke sebuah materi lanjutan: **AJAX**.

# 25. AJAX

AJAX sebenarnya termasuk materi *advanced* (lanjutan) di dalam JavaScript. Namun karena penggunaannya cukup banyak dan istilah AJAX juga sangat populer, saya ingin membahasnya dalam buku ini.



Pembahasan terkait AJAX butuh pemahaman tentang PHP. Apabila anda belum pernah mempelajari PHP (misalnya dari buku **PHP Uncover**), boleh lompati bab ini untuk sementara.

## 25.1 Pengertian AJAX

AJAX adalah singkatan dari **Asynchronous JavaScript And XML**. Diberi nama seperti itu karena pada awalnya AJAX digunakan untuk mengirim dan menerima data XML dari web server. Sekarang, data yang dikirim tidak hanya XML saja, tapi juga bisa berupa file teks maupun JSON (*JavaScript Object Notation*).

Secara sederhana, AJAX digunakan untuk membuat HTML dan JavaScript bisa berkomunikasi dengan web server, tanpa harus reload/refresh.

Dengan menggunakan AJAX, kita bisa mengirim isian form HTML ke web server secara realtime, menampilkan isi database tanpa men-klik tombol submit, atau mengupdate sebagian data tanpa harus me-load ulang seluruh halaman web.

AJAX merupakan sebuah teknologi yang sangat revolusioner di dalam web programming. Dengan AJAX, situs web bisa dibuat menyerupai aplikasi desktop.

Tentunya anda sudah familiar dengan **gmail** (google mail). Jika diperhatikan, selama menggunakan gmail untuk membuat, membuka, dan mengirim email, halaman website gmail tidak pernah refresh.

Saat mengetik beberapa huruf di google, akan keluar saran pencarian (*search suggestion*). Data ini berasal dari database google dan ditampilkan menggunakan AJAX. Situs modern seperti Dropbox, Facebook, dan Twitter juga banyak menggunakan AJAX.

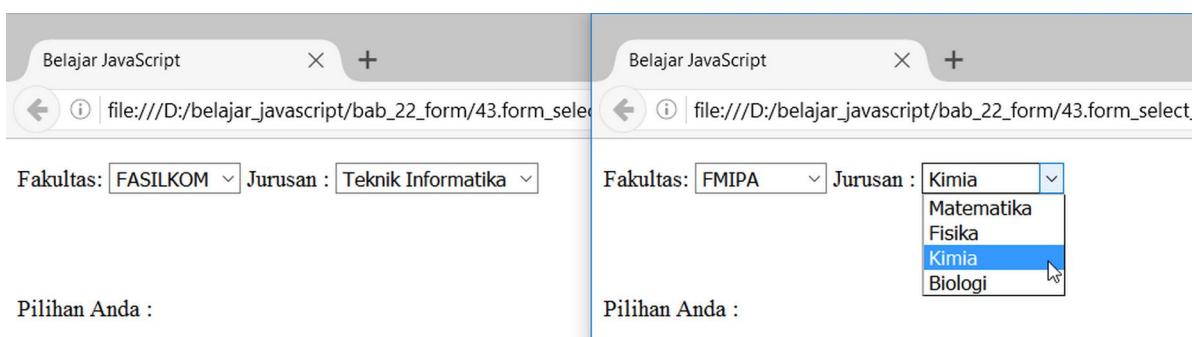
AJAX membuka banyak hal yang sebelumnya tidak bisa dilakukan. Online word processor seperti **Google Doc** atau peta interaktif seperti **Google Map** juga contoh website yang secara penuh menggunakan AJAX.



Gambar: Fitur search suggestion Google, menggunakan konsep AJAX

## 25.2 Konsep Penggunaan AJAX

Untuk lebih memahami konsep AJAX, mari lihat kembali contoh program yang pernah kita buat pada bab **Form Processing**:



Gambar: Membuat menu jurusan

Disini saya membuat menu dropdown dinamis yang isinya bisa berubah. Daftar pilihan fakultas dan jurusan digenerate menggunakan JavaScript.

Sekarang, bagaimana jika di kemudian hari saya ingin menambah fakultas dan jurusan lain? Saya harus menginputnya secara manual ke dalam JavaScript. Cukup rumit dan tidak praktis, apalagi dilakukan oleh pengguna yang tidak paham programming.

Solusinya, nama fakultas dan jurusan ini bisa diinput menggunakan form terpisah dan disimpan ke dalam database. Namun masalahnya beralih ke sisi coding. Bagaimana caranya agar daftar fakultas dan jurusan yang disimpan di dalam database dapat diambil ke dalam JavaScript?

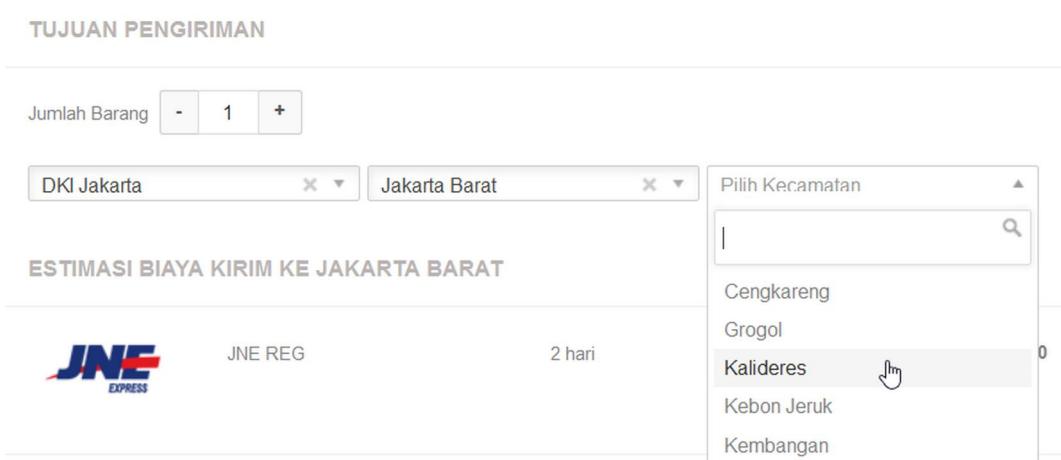
Alternatif pertama, pada saat halaman tampil pertama kali, ambil semua data yang dibutuhkan dari database menggunakan PHP, kemudian input ke dalam JavaScript. Solusi ini cukup praktis dan relatif mudah dibuat. Namun untuk data yang besar, bisa jadi daftar pilihan ini terdiri dari ribuan baris data.

Alternatif kedua, gunakan **AJAX**. Saat terjadi event change dari tag <select>, kirim perintah ke web server. Isinya, minta PHP mengambil data yang diperlukan dari database, kemudian kirim

ke tag <select>. Jika pilihan tag <select> ditukar, kembali kirim permintaan ke web server. Untuk data yang besar, cara ini akan lebih efisien. Karena kita hanya perlu mengambil data yang dibutuhkan saja.

Sebagai contoh, jika saya memilih fakultas FASILKOM, kirim permintaan ke server untuk mengambil daftar nama jurusan FASILKOM dari database. jika saya memilih fakultas Ekonomi, kirim permintaan ke server untuk mengambil daftar nama jurusan yang ada di fakultas Ekonomi. Ini semua dilakukan di background menggunakan **AJAX**.

Implementasinya bisa kita ambil contoh kolom pengisian alamat di bukalapak:



Gambar: Contoh dropdown untuk memilih alamat

Tanpa **AJAX**, bukalapak harus menggenerate ribuan kota, kabupaten, dan kecamatan dalam 1 kali proses. Menggunakan **AJAX**, data yang diminta hanyalah nama kecamatan yang ada di 1 provinsi saja (yang sudah dipilih dari tag <select> sebelumnya).

Fitur seperti inilah yang bisa kita buat menggunakan **AJAX**, yakni berkomunikasi dengan web server tanpa halaman harus di reload/refresh.

## 25.3 XMLHttpRequest Object

Untuk membuat **AJAX**, kita menggunakan **XMLHttpRequest Object**. Object ini merupakan bagian dari **DOM** (Document Object Model), dan sudah didukung oleh mayoritas web browser modern.

Agar bisa menggunakan **XMLHttpRequest Object**, kita harus membuat *instance* dari object ini:

```
var request = new XMLHttpRequest();
```

Variabel `request` berisi instance dari **XMLHttpRequest Object**. Anda boleh menggunakan nama variabel lain, tidak harus bernama `request`. Sebagaimana layaknya object di dalam JavaScript, **XMLHttpRequest Object** memiliki berbagai method dan property.

Selanjutnya, kita perlu memanggil method `XMLHttpRequest.open()` untuk membuat pengaturan bagaimana cara meminta data ke web server.

Method `XMLHttpRequest.open()` membutuhkan 3 buah argumen:

- **Argumen pertama:** mengatur cara mengirim data ke web server, apakah menggunakan metode "GET" atau "POST". Makna GET dan POST ini sama seperti cara mengirim form HTML ke web server.
- **Argumen kedua:** berisi alamat file yang akan memproses data di web server. Biasanya berupa alamat file PHP. Ini sama seperti isi atribut `action` di dalam tag `<form>` HTML.
- **Argumen ketiga:** berisi boolean `true` atau `false`. Jika diset sebagai `true`, data akan dikirim secara **asynchronous** (pilihan default). Jika di set sebagai `false`, data akan dikirim secara **synchronous**.

Sebagai contoh, untuk berkomunikasi dengan file `proses.php` secara *synchronous* dan menggunakan metode GET, kode programnya adalah sebagai berikut:

```
1 var request = new XMLHttpRequest();
2 request.open("GET", "proses.php", false);
```

Langkah berikutnya adalah mengirimkan permintaan ke web server menggunakan method `XMLHttpRequest.send()`. Jika data di kirim menggunakan metode GET, method `send()` tidak perlu diisi argumen apapun:

```
1 var request = new XMLHttpRequest();
2 request.open("GET", "proses.php", false);
3 request.send();
```

Dengan 3 perintah diatas, sebuah permintaan atau request telah dikirim ke web server. Hasil dari permintaan ini akan diterima oleh property `XMLHttpRequest.responseText`. Biasanya hasil ini berupa string teks yang bisa langsung diinput ke dalam DOM, seperti contoh berikut:

```
1 var request = new XMLHttpRequest();
2 request.open("GET", "proses.php", false);
3 request.send();
4 hasilNode.innerHTML = request.responseText;
```

Empat baris kode program diatas sudah cukup untuk membuat sebuah **AJAX Request**. Mari kita buat beberapa contoh praktek.

## 25.4 Menampilkan File Teks - Synchronous AJAX

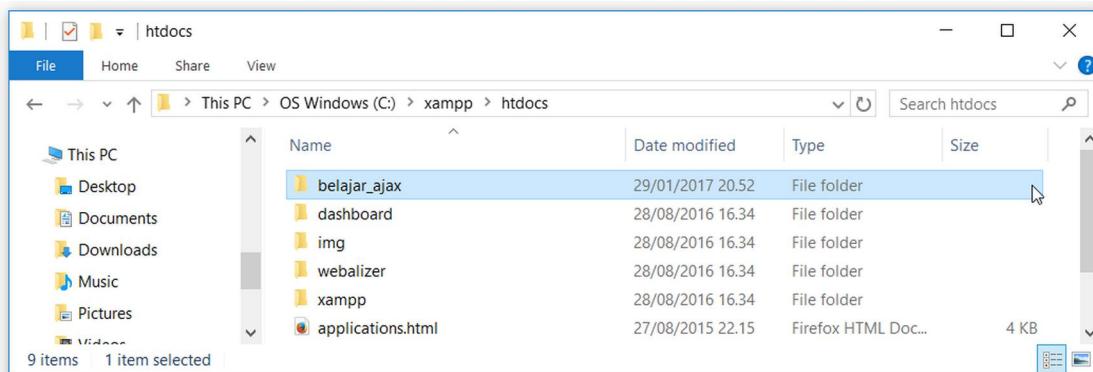
Praktek AJAX yang paling sederhana adalah menampilkan isi sebuah file teks yang berada di server secara *synchronous*.

Maksud dari **synchronous** adalah, proses di web browser akan berhenti sejenak selama AJAX diproses. Setelah selesai, barulah web browser bisa lanjut ke proses berikutnya (jika ada). Kurang lebih mirip seperti method `alert()` yang akan menahan proses sampai tombol OK dari jendela `alert` di klik.

Selain *synchronous*, terdapat metode lain yakni **asynchronous**. Untuk **asynchronous**, AJAX akan dijalankan secara simultan (bersamaan) dengan proses lain di web browser. Proses **asynchronous** memang lebih efisien, tapi perlu beberapa langkah tambahan. Saya akan membahas proses *synchronous* yang lebih sederhana terlebih dahulu.

Sebagaimana yang telah dijelaskan sebelumnya, AJAX berfungsi sebagai penghubung antara **web browser** dengan **web server**. Agar kita bisa melakukan praktik, seluruh file HTML dan JavaScript harus berada di web server. Dalam praktik ini saya akan menggunakan web server **apache** dari aplikasi **XAMPP**.

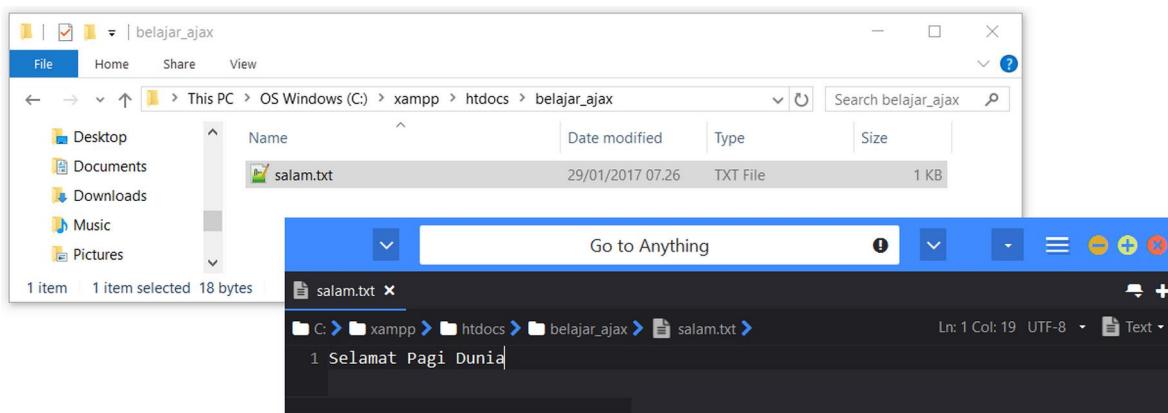
Jalankan web server **apache**, kemudian buat folder "belajar\_ajax" di folder **htdocs** **XAMPP**.



Gambar: Buat folder "belajar\_ajax" di htdocs XAMPP

Di dalam folder "belajar\_ajax", buat sebuah file teks bernama: `salam.txt`. Isi dari file teks tersebut hanya 1 baris teks:

`Selamat Pagi Dunia`



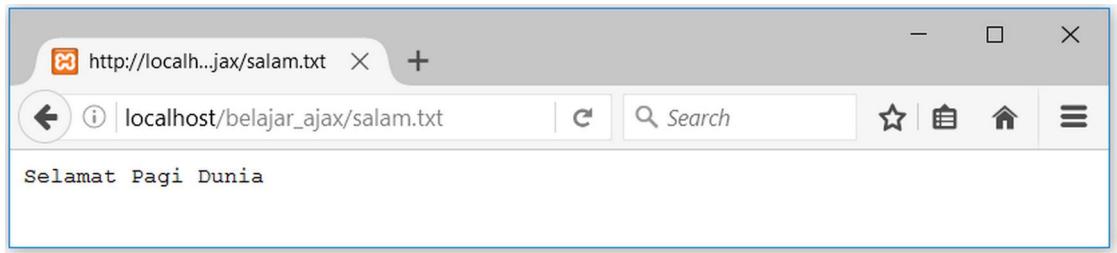
Gambar: File `salam.txt` di dalam folder "belajar\_ajax"



Seluruh file kode program yang akan kita buat harus ditempatkan ke dalam folder "`htdocs\belajar_ajax`", termasuk file HTML dan PHP.

Untuk menguji apakah web server telah berjalan dan file `salam.txt` sudah bisa diakses,

silahkan buka alamat "http://localhost/belajar\_ajax/salam.txt". Jika tidak ada masalah, akan tampil jendela berikut:

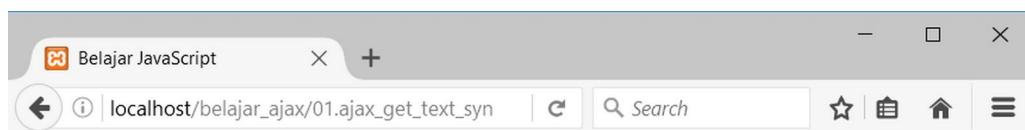


Gambar: File salam.txt sukses diakses dari localhost

File salam.txt telah sukses diakses, maka kita bisa masuk ke kode HTML dan JavaScript untuk menjalankan AJAX. Silahkan buat file HTML baru di folder "belajar\_ajax" yang isinya sebagai berikut:

```

1 <body>
2   <h1>Belajar AJAX</h1>
3   <button id="tombol">Ambil Data</button>
4   <p>Hasil: <span id="hasil"></span></p>
5   <script>
6     var tombolNode = document.getElementById("tombol");
7     var hasilNode = document.getElementById("hasil");
8
9     function getAJAX(){
10       var request = new XMLHttpRequest();
11       request.open("GET", "salam.txt", false);
12       request.send();
13       hasilNode.innerHTML = request.responseText;
14     }
15
16     tombolNode.addEventListener("click",getAJAX);
17   </script>
18 </body>
```



## Belajar AJAX

Hasil:

Gambar: Menambil isi file salam.txt menggunakan AJAX

Di bagian HTML, saya membuat tombol **Ambil Data** dari tag `<button id="tombol">`. Kemudian terdapat tag `<span id="hasil">` yang akan digunakan sebagai penampung hasil AJAX.

Di dalam JavaScript, variabel `tombolNode` akan menyimpan node object dari `<button id="tombol">`, sedangkan variabel `hasilNode` akan menyimpan node object dari tag `<span id="hasil">`.

Ketika tombol **Ambil Data** di klik, jalankan function `getAJAX()`. Di dalam function inilah kode program AJAX dibuat:

```

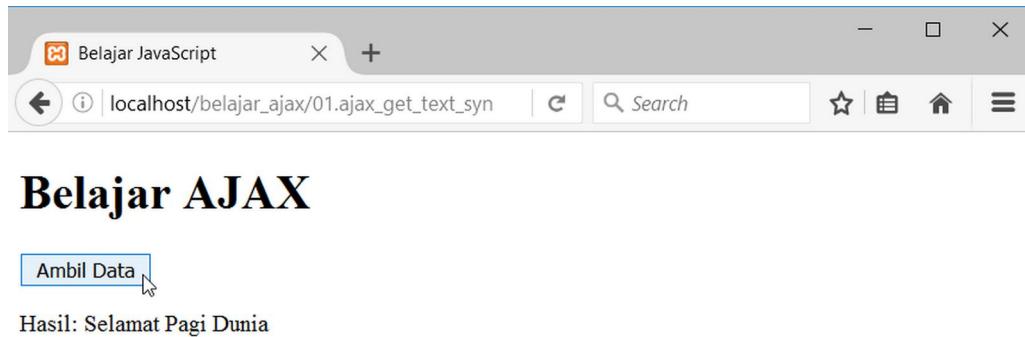
1 function getAJAX(){
2   var request = new XMLHttpRequest();
3   request.open("GET", "salam.txt", false);
4   request.send();
5   hasilNode.innerHTML = request.responseText;
6 }
```

Di awal function, saya membuat variabel `request` untuk menampung *instance* dari **XMLHttpRequest object**.

Berikutnya terdapat method `request.open()`. Isinya berupa setingan awal bagaimana proses AJAX akan dijalankan. Disini saya menulis `request.open("GET", "salam.txt", false)`. Artinya, kirim data menggunakan metode GET, ke file "salam.txt" secara *synchronous*. Kemudian jalankan proses ajax dengan memanggil method `request.send()`.

Terakhir, input hasil kembalian **AJAX** yang disimpan di `request.responseText` ke dalam `hasilNode.innerHTML`.

Jika semuanya berjalan lancar, saat tombol **Ambil Data** di klik akan tampil teks "Selamat Pagi Dunia". Inilah data yang diambil menggunakan **AJAX**!



Gambar: Hasil AJAX yang berasal dari file `salam.txt`

Jika isi file `salam.txt` ini ditukar dan men-klik kembali tombol **Ambil Data**, teks yang tampil juga akan berubah. Mari kita coba.

Tanpa menutup web browser, silahkan buka file `salam.txt` dan ubah teksnya menjadi:

Selamat Pagi Indonesia

Save file tersebut dan kembali buka web browser, kemudian klik tombol **Ambil Data**. Bagaimana hasilnya? Apabila anda kurang beruntung, teks yang tampil tetap "Selamat Pagi Dunia", bukan "Selamat Pagi Indonesia". Apa yang terjadi?

Bisa saya pastikan bahwa tidak ada yang salah dengan kode **AJAX** diatas. Masalah ini disebabkan oleh fitur **cache** dari web browser.

Web browser akan menyimpan teks "Selamat Pagi Dunia" di dalam **cache**, sehingga request atau permintaan data untuk file yang sama akan diambil dari dalam **cache**, bukan ke web server lagi.

Dalam kebanyakan kasus, **cache** akan mempercepat proses tampilan website. Sebab, web browser sudah meyimpan data tersebut di komputer kita dan tidak harus meminta ulang ke web server. Tapi efeknya, jika data di web server berubah, hasil di web browser belum tentu mengikuti.

Solusi dari masalah ini adalah dengan menambahkan sebuah *query string* yang berisi angka random ke dalam alamat file. Jika sebelumnya saya menulis method `request.open()` sebagai berikut:

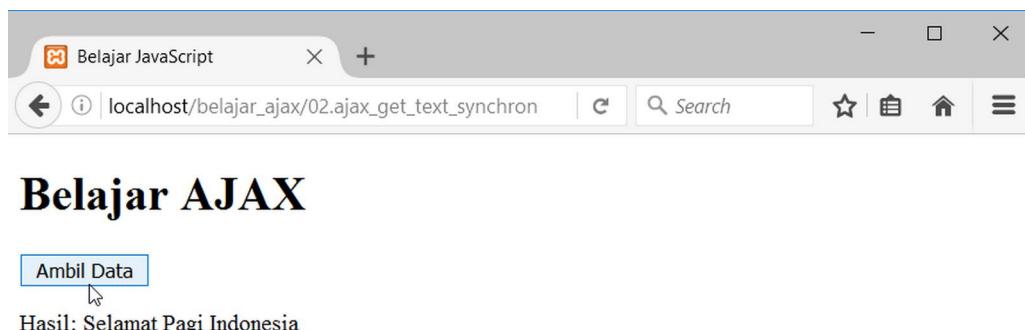
```
request.open("GET", "salam.txt", false);
```

Ubah baris tersebut menjadi:

```
request.open("GET", "salam.txt?s="+ Math.random(), false);
```

Dengan trik seperti ini, web browser akan melihat bahwa nama file `salam.txt` selalu berubah-ubah, padahal yang bertukar hanya di bagian *query string*, yakni tambahan "`?s=`" yang tidak berpengaruh apa-apa terhadap file.

Sekarang, silahkan anda tukar isi teks file `salam.txt` dan klik tombol **Ambil Data**, hasil teks yang tampil juga akan update.



Gambar: *Selamat Pagi Indonesia*

## 25.5 Menampilkan File Teks - Asynchronous AJAX

Setelah berhasil mempraktekkan **synchronous AJAX**, kita akan coba **asynchronous AJAX**. Keunggulan dari **asynchronous** adalah, proses di web browser tidak terganggu oleh **AJAX**. Proses

JavaScript dan AJAX bisa berjalan pada saat yang bersamaan. Namun karena kita mengakses file menggunakan web server lokal, perbedaannya tidak akan terasa.

Dari sisi coding, perbedaan dari asynchronous AJAX adalah: argumen ketiga dari method open(), diisi dengan **true**.

Selain itu kita juga butuh tambahan property request.onreadystatechange. Di dalam property inilah hasil teks dari web server di proses.

Berikut perubahan function getAJAX() jika menggunakan **asynchronous AJAX**:

```
1 function getAJAX(){
2     var request = new XMLHttpRequest();
3     request.open("GET", "salam.txt?s="+ Math.random(), true);
4     request.send();
5
6     request.onreadystatechange = function() {
7         if (request.readyState == 4 && request.status == 200) {
8             hasilNode.innerHTML = request.responseText;
9         }
10    };
11 }
```

Perhatikan isi dari property request.onreadystatechange, saya menginput property ini dengan sebuah function.

Di dalam function tersebut terdapat pengecekan kondisi `if (request.readyState == 4 && request.status == 200)`. Ini diperlukan untuk memastikan data dari web server sudah sampai dengan selamat ke web browser. Apabila kedua kondisi ini bernilai **true**, baru input `request.responseText` ke dalam `hasilNode.innerHTML`.

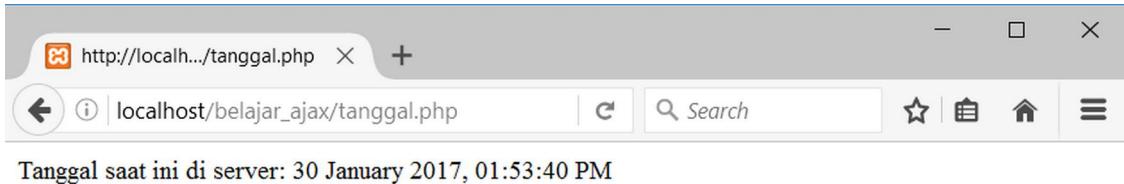
Hasil dari kode program ini sama seperti sebelumnya, karena file yang diakses juga masih file `salam.txt`.

## 25.6 Menampilkan File PHP dengan AJAX

Dalam latihan sebelumnya, kita masih mengakses sebuah file text statis, yakni `salam.txt`. Mari kita coba akses file PHP. Sebagai bahan praktek, saya membuat file `tanggal.php` yang berisi kode berikut:

```
1 <?php
2     date_default_timezone_set("Asia/Jakarta");
3
4     $tanggal = date("j F Y, h:i:s A");
5     echo "Tanggal saat ini di server: $tanggal";
6 ?>
```

Kode program PHP diatas akan menampilkan tanggal yang ada di server. Berikut hasilnya:



Gambar: Hasil dari file tanggal.php

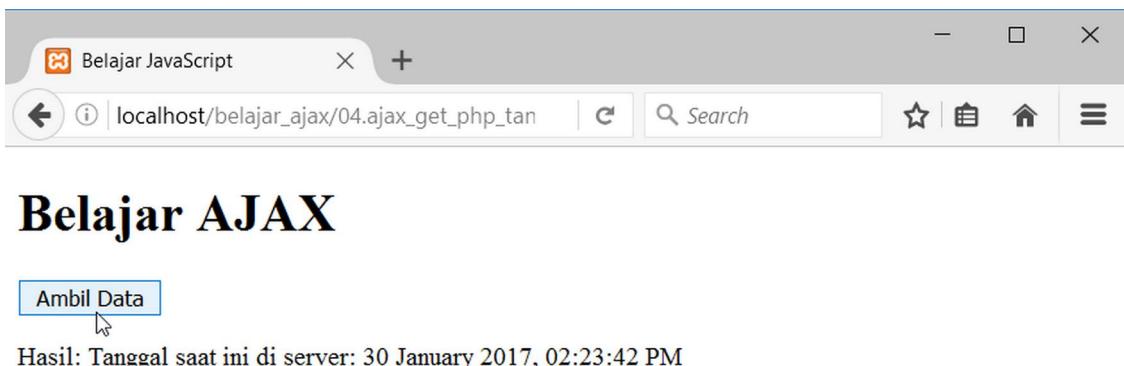
Agar hasil tersebut bisa diambil menggunakan AJAX, saya akan update function getAJAX():

```

1  function getAJAX(){
2      var request = new XMLHttpRequest();
3      request.open("GET", "tanggal.php", true);
4      request.send();
5
6      request.onreadystatechange = function() {
7          if (request.readyState == 4 && request.status == 200) {
8              hasilNode.innerHTML = request.responseText;
9          }
10     };
11 }
```

Tidak banyak perubahan, saya hanya mengganti nama file tujuan AJAX dari salam.txt menjadi tanggal.php. Sehingga pemanggilan method open menjadi: request.open("GET", "tanggal.php", true).

Berikut hasilnya:



Gambar: Menampilkan tanggal dengan AJAX

Silahkan anda tekan tombol “Ambil Data” beberapa kali, tampilan tanggal akan terus berubah menyesuaikan dengan tanggal dari server apache. Setiap kali tombol ditekan, AJAX akan berjalan dan mengambil data baru dari file tanggal.php.

Data yang dikirim dari file PHP tidak hanya file teks saja, tapi juga bisa tag HTML. Saya akan buat file PHP lain dengan nama tanggal\_format.php, yang isinya sebagai berikut:

```

1 <?php
2     date_default_timezone_set("Asia/Jakarta");
3
4     $tanggal = date("j F Y, h:i:s A");
5     echo "<h2>Tanggal saat ini di server: <i>$tanggal</i></h2>";
6 ?>

```

Disini saya menambahkan tag `<h2>` dan `<i>` ke dalam perintah echo PHP. Karena menggunakan file yang berbeda, kita juga harus update function `getAJAX()` dengan mengganti isi method `request.open` menjadi seperti berikut:

```
request.open("GET", "tanggal_format.php", true);
```



Gambar: Tanggal dari PHP dengan tambahan tag `<h1>` dan `<i>`

Dari hasil diatas dapat dilihat bahwa data yang dikirim lewat AJAX juga bisa berisi tag HTML.

## 25.7 Mengirim Data ke File PHP - Metode GET

Selain menampilkan data dari web server, AJAX juga bisa digunakan untuk mengirim data ke web server. Prosesnya sama seperti pemrosesan form HTML. Jika form tersebut menggunakan metode GET, data yang dikirim bisa ditulis ke dalam *query string*.

Mari kita praktekkan dengan memodifikasi function `getAJAX()`:

```

1 function getAJAX(){
2     var request = new XMLHttpRequest();
3     request.open("GET", "fakultas.php?f=fasilkom", false);
4     request.send();
5     hasilNode.innerHTML = request.responseText;
6 }

```

Saya kembali menggunakan **synchronous AJAX** semata-mata agar kode programnya menjadi lebih singkat.

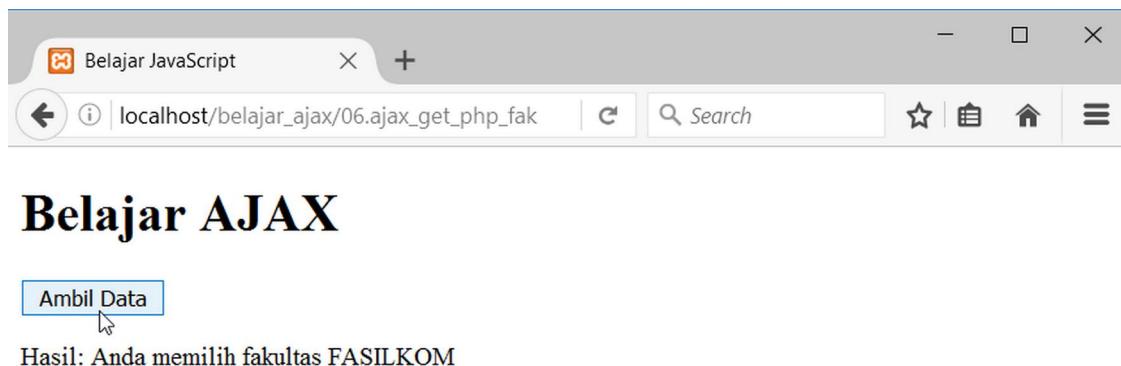
Perhatikan alamat file AJAX, saya menulis "fakultas.php?f=fasilkom". Terdapat sebuah *query string* f=fasilkom yang akan dikirim ke file fakultas.php.

Di dalam fakultas.php, *query string* ini bisa "ditangkap" menggunakan global variabel PHP, yakni \$\_GET. Berikut isi dari file fakultas.php:

```
1 <?php
2     $fakultas = $_GET["f"];
3
4     if ($fakultas === "fasilkom"){
5         echo "Anda memilih fakultas FASILKOM";
6     }
7     else if ($fakultas === "ekonomi"){
8         echo "Anda memilih fakultas Ekonomi";
9     }
10    ?>
```

Artinya, jika *query string* yang dikirim adalah f=fasilkom, jalankan perintah echo "Anda memilih fakultas FASILKOM". Jika yang dikirim adalah f=ekonomi, jalankan perintah echo "Anda memilih fakultas Ekonomi".

Berikut hasilnya:



Gambar: Mengirim query string "f=fasilkom" menggunakan metode GET

Akan tampil teks "Anda memilih fakultas FASILKOM" karena pada saat pemanggilan AJAX, *query string* yang ditulis adalah f=fasilkom.

Silahkan anda coba mengganti alamat file PHP menjadi "fakultas.php?f=ekonomi", lalu jalankan kembali file HTML. Hasilnya juga akan berubah menjadi "Anda memilih fakultas Ekonomi".

## 25.8 Mengirim Data ke File PHP - Metode POST

Jika ingin menggunakan metode POST, cara pengiriman data AJAX sedikit berbeda:

```

1 function getAJAX(){
2     var request = new XMLHttpRequest();
3     request.open("POST", "fakultas_post.php", false);
4     request.setRequestHeader("Content-type",
5                             "application/x-www-form-urlencoded");
6     request.send("f=ekonomi");
7     hasilNode.innerHTML = request.responseText;
8 }

```

Untuk method `request.open()`, argument pertama saya ganti dari **GET** menjadi **POST**. Selain itu terdapat penambahan method baru, yakni `request.setRequestHeader("Content-type", "application/x-www-form-urlencoded")`.

Method `request.setRequestHeader()` diperlukan agar web server bisa paham bahwa terdapat data yang akan dikirim menggunakan metode **POST**.

Data yang akan dikirim diinput sebagai argumen dari method `send()`. Jika saya ingin mengirim data `f=ekonomi`, maka perintahnya menjadi `request.send("f=ekonomi")`.

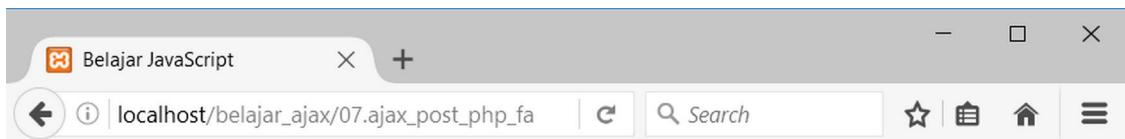
Sekarang, kita harus menyiapkan file `fakultas_post.php` yang isinya adalah sebagai berikut:

```

1 <?php
2     $fakultas = $_POST["f"];
3
4     if ($fakultas === "fasilkom"){
5         echo "Anda memilih fakultas FASILKOM";
6     }
7     else if ($fakultas === "ekonomi"){
8         echo "Anda memilih fakultas Ekonomi";
9     }
10 ?>

```

Persiapan sudah selesai, ketika tombol **“Ambil Data”** di klik akan tampil teks **“Anda memilih fakultas Ekonomi”**.



## Belajar AJAX

**Ambil Data**

Hasil: Anda memilih fakultas Ekonomi

Gambar: Mengirim data "f=ekonomi" menggunakan metode POST

Jika anda sudah sering latihan pemrosesan form menggunakan PHP, banyak hal yang bisa dibuat menggunakan AJAX. Misalnya mengambil data yang ada di database.

## 25.9 Case Study: Mengambil Data dari Database MySQL

Setelah paham konsep dasar penggunaan AJAX, kita akan masuk ke sebuah **case study** yang sedikit lebih rumit, yakni menggunakan AJAX untuk mengambil data dari database MySQL.

Untuk membuat fitur seperti ini, kita akan menggunakan 4 materi dasar web programming: **HTML, PHP, MySQL dan JavaScript**. Saya memilih untuk tidak menggunakan CSS agar kode programnya tidak semakin panjang.

Pertama, siapkan database yang akan diakses. Saya akan menggunakan contoh database dari studi kasus terakhir buku **PHP Uncover**, yakni database kampusku.

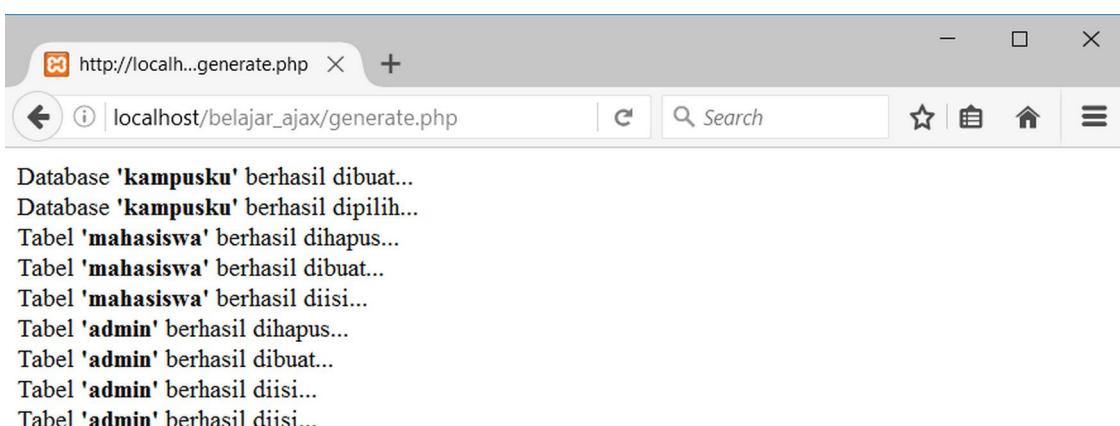
Database kampusku terdiri dari 2 tabel: `admin` dan `mahasiswa`. Kita hanya akan pakai tabel `mahasiswa` saja. Berikut isi dan struktur tabel tersebut:

	<input type="checkbox"/>	<a href="#">Edit</a>	<a href="#">Copy</a>	<a href="#">Delete</a>	nim	nama	tempat_lahir	tanggal_lahir	fakultas	jurusan	ipk
	<input type="checkbox"/>	<a href="#">Edit</a>	<a href="#">Copy</a>	<a href="#">Delete</a>	13012012	James Situmorang	Medan	1995-04-02	Kedokteran	Kedokteran Gigi	2.70
	<input type="checkbox"/>	<a href="#">Edit</a>	<a href="#">Copy</a>	<a href="#">Delete</a>	14005011	Riana Putria	Padang	1996-11-23	FMIPA	Kimia	3.10
	<input type="checkbox"/>	<a href="#">Edit</a>	<a href="#">Copy</a>	<a href="#">Delete</a>	15002032	Rina Kumala Sari	Jakarta	1997-06-28	Ekonomi	Akuntansi	3.40
	<input type="checkbox"/>	<a href="#">Edit</a>	<a href="#">Copy</a>	<a href="#">Delete</a>	15003036	Sari Citra Lestari	Jakarta	1997-12-31	Ekonomi	Manajemen	3.50
	<input type="checkbox"/>	<a href="#">Edit</a>	<a href="#">Copy</a>	<a href="#">Delete</a>	15021044	Rudi Permana	Bandung	1994-08-22	FASILKOM	Ilmu Komputer	2.90

Gambar: Tabel `mahasiswa` di dalam database kampusku

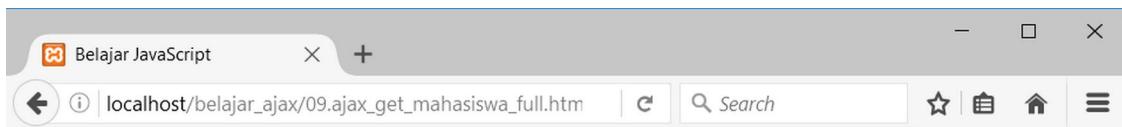
Untuk membuat tabel ini, silahkan akses file `generate.php` dari folder `bab_25_ajax` yang terdapat di dalam file `belajar_javascript.zip`. File tersebut sama persis seperti file `generate.php` yang ada di buku **PHP Uncover**.

**i** Karena kita akan mengakses database MySQL, pastikan **MySQL Server** sudah berjalan dari **XAMPP Control Panel**.



Gambar: Proses generate database kampusku

Setelah database tersedia dan bisa diakses, mari kita rancang kode program untuk membuat halaman berikut ini:



## Tabel Mahasiswa dengan AJAX

Nama Mahasiswa:

NIM	150
Nama	Sari
Tempat Lahir	Jakarta
Tanggal Lahir	31 - 12 - 1997
Fakultas	Ekonomi
Jurusan	Manajemen
IPK	3.50

Gambar: Menampilkan isi tabel Mahasiswa dengan AJAX

Pilihan dropdown berisi daftar nama mahasiswa. Nama mahasiswa ini diambil dari database menggunakan AJAX. Jika salah satu nama dipilih, detail dari mahasiswa tersebut tampil dalam bentuk tabel di bagian bawah. Tabel inipun akan ditampilkan menggunakan AJAX.

## Membuat Dropdown Pilihan Nama Mahasiswa

Langkah pertama, kita akan merancang kode program untuk pilihan dropdown nama mahasiswa. Ini dibuat menggunakan tag `<select>` yang berisi tag `<option>` nama mahasiswa. Tapi nama mahasiswa ini berada di database, bagaimana cara mengambilnya?

Terdapat 2 alternatif: Menggunakan PHP untuk menggenerate seluruh nama mahasiswa pada saat halaman di load, atau di generate menggunakan AJAX. Saya memilih cara kedua, yakni menggunakan AJAX. Berikut struktur HTML dan JavaScript yang diperlukan:

```

1 <body>
2   <h1>Tabel Mahasiswa dengan AJAX</h1>
3   <p>Nama Mahasiswa:
4     <select name="namaMahasiswa" id="namaMahasiswa"></select>
5   </p>
6   <script>
7     var namaMahasiswaNode = document.getElementById("namaMahasiswa");
8
9     function generateMahasiswa(){
10       var request = new XMLHttpRequest();
11       request.open("GET", "nama_mahasiswa.php", false);
12       request.send();
13       namaMahasiswaNode.innerHTML = request.responseText;

```

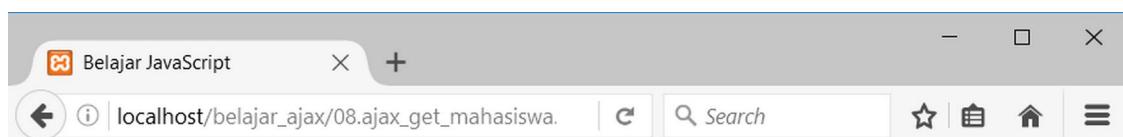
```

14      }
15
16      generateMahasiswa();
17  </script>
18 </body>
```

Di dalam kode HTML, saya membuat tag `<select name="namaMahasiswa" id="namaMahasiswa">`. Tag ini sengaja di kosongkan karena kita akan mengisinya dari AJAX. Variabel `namaMahasiswaNode` berisi *node object* dari tag ini.

Kemudian terdapat function `generateMahasiswa()` yang berisi kode AJAX. Saya menggunakan metode `GET` untuk mengakses file `nama_mahasiswa.php` secara *synchronous*. Hasilnya akan diinput ke `namaMahasiswaNode.innerHTML`. Anda bisa menebak kalau hasil AJAX ini nantinya berupa kumpulan tag `<option>` sebagai pengisi tag `<select>`.

Jika kita menjalankan kode HTML diatas, hasilnya berupa halaman dengan 1 tag `<select>` yang belum berisi pilihan apa-apa:



## Tabel Mahasiswa dengan AJAX

Nama Mahasiswa: 

Gambar: Dropdown pilihan nama mahasiswa masih kosong

Langkah berikutnya adalah membuat kode program untuk `nama_mahasiswa.php`. Disinilah tag `<option>` akan di-generate. Berikut isi dari file tersebut:

```

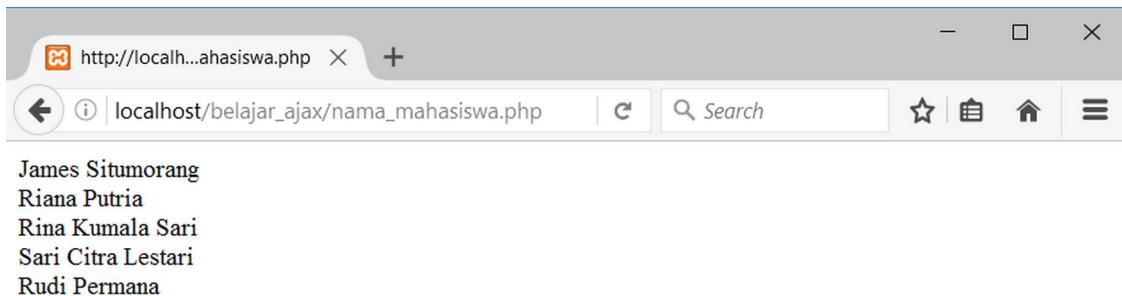
1 <?php
2 // buat koneksi dengan database mysql
3 $link = mysqli_connect("localhost", "root", "", "kampusku");
4
5 // ambil kolom nama dari tabel mahasiswa
6 $query = "SELECT nama FROM mahasiswa";
7 $result = mysqli_query($link, $query);
8
9 // tambahkan tag <option> untuk setiap nama mahasiswa
10 while($data = mysqli_fetch_array($result)) {
11     echo "<option value='{$data['nama']}'>{$data['nama']}</option>";
12 }
13 ?>
```

Setelah membuat koneksi dengan MySQL, saya mengambil seluruh kolom `nama` dari tabel `mahasiswa` menggunakan query "`SELECT nama FROM mahasiswa`".

Kode yang cukup rumit ada di perintah **echo**. Saya buat seperti itu supaya hasil akhirnya memiliki format seperti ini:

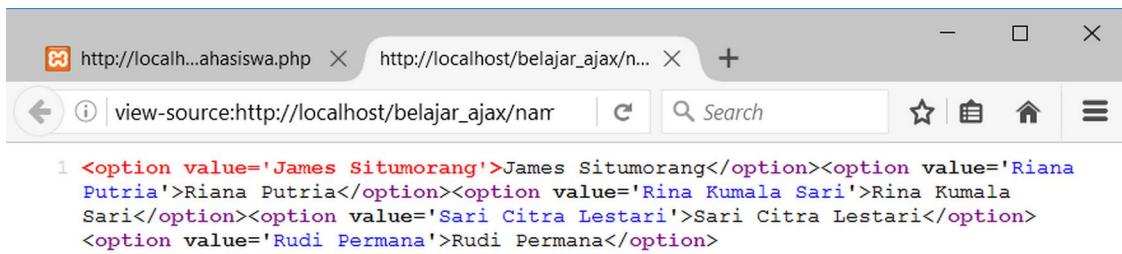
```
<option value='Nama mahasiswa'>Nama mahasiswa</option>
```

Mari kita uji kode PHP diatas:



Gambar: Nama mahasiswa di generate dari PHP

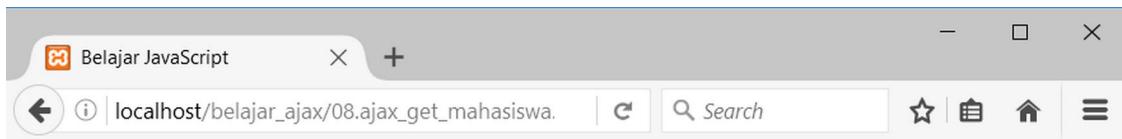
Tampak seluruh nama mahasiswa sudah tampil. Tapi kita juga harus memastikan setiap nama mahasiswa ini sudah berada di dalam tag `<option>`. Caranya, klik kanan halaman lalu pilih *View Page Source*:



Gambar: Tag `<option>` diantara nama siswa

Hasilnya tampak sedikit berantakan. Ini karena setiap tag `<option>` langsung disambung dengan tag `<option>` berikutnya. Tapi struktur yang kita inginkan sudah benar.

Mari akses kembali kode HTML yang sudah kita rancang:



## Tabel Mahasiswa dengan AJAX

Nama Mahasiswa:

James Situmorang

James Situmorang

Riana Putria

**Rina Kumala Sari**

Sari Citra Lestari

Rudi Permana

Gambar: Pilihan nama mahasiswa sukses ditampilkan

*Sip!* Pilihan dropdown nama mahasiswa sudah terisi. Berikutnya kita akan masuk ke perancangan kode program untuk menampilkan detail data mahasiswa.

## Menampilkan Data Mahasiswa

Saat pilihan dropdown nama mahasiswa diganti, sebuah tabel yang berisi data mahasiswa tersebut akan tampil di bagian bawah.

Dalam logika program, kalimat diatas dapat diterjemahkan menjadi: "Ketika terjadi event **change** dari tag <select>, jalankan kode program AJAX yang meminta data mahasiswa tersebut ke web server. Hasilnya di tampilkan di bagian bawah halaman". Artinya, kita akan membuat pemanggilan AJAX kedua.

Langkah pertama, buat sebuah tag yang akan menampung data mahasiswa. Untuk ini saya menggunakan tag <div>:

```
<div id="hasil"></div>
```

Di dalam JavaScript, node object dari tag diatas akan disimpan ke dalam variabel `hasilNode`. Selanjutnya tambahkan event **change** ke dalam tag <select>:

```
namaMahasiswaNode.addEventListener("change", tabelMahasiswa);
```

Maksud kode program diatas adalah, saat pilihan dropdown nama mahasiswa diganti, jalankan function `tabelMahasiswa()`. Di dalam function inilah proses AJAX dilakukan:

```

1 function tabelMahasiswa(){
2     var nama = namaMahasiswaNode.value;
3     var request = new XMLHttpRequest();
4     request.open("GET", "tabel_mahasiswa.php?n="+nama, false);
5     request.send();
6     hasilNode.innerHTML = request.responseText;
7 }

```

Kode program AJAX diatas juga terbilang “standar”, seperti kode-kode yang kita pakai sebelumnya. Disini saya merancang kode AJAX untuk mengakses halaman tabel\_mahasiswa.php menggunakan metode GET secara *synchronous*.

Namun perhatikan tambahan *query string* pada bagian alamat: tabel\_mahasiswa.php?n="+nama. Saya mengirim sebuah data "n" yang isinya berupa variabel nama. Variabel nama ini berasal dari namaMahasiswaNode.value. Artinya, kirim nilai atribut value dari tag <select> ke web server.

Jika di dalam tag <select> nama mahasiswa saya memilih "Rina Kumala Sari", maka *query string* yang dikirim akan menjadi:

tabel\_mahasiswa.php?n=Rina Kumala Sari

Jika yang dipilih adalah "Rudi Permana", maka query stringnya akan menjadi:

tabel\_mahasiswa.php?n=Rudi Permana

Isi dari query string ini nantinya akan ditangkap oleh halaman tabel\_mahasiswa.php.

Berikut kode program lengkap dari halaman HTML untuk menampilkan data mahasiswa:

```

1 <body>
2     <h1>Tabel Mahasiswa dengan AJAX</h1>
3     <p>Nama Mahasiswa:
4         <select name="namaMahasiswa" id="namaMahasiswa">
5             </select>
6     </p>
7     <div id="hasil"></div>
8     <script>
9         var namaMahasiswaNode = document.getElementById("namaMahasiswa");
10        var hasilNode = document.getElementById("hasil");
11
12        function generateMahasiswa(){
13            var request = new XMLHttpRequest();
14            request.open("GET", "nama_mahasiswa.php", false);
15            request.send();
16            namaMahasiswaNode.innerHTML = request.responseText;
17        }

```

```

18
19     function tabelMahasiswa(){
20         var nama = namaMahasiswaNode.value;
21         var request = new XMLHttpRequest();
22         request.open("GET", "tabel_mahasiswa.php?n="+nama, false);
23         request.send();
24         hasilNode.innerHTML = request.responseText;
25     }
26
27     generateMahasiswa();
28     namaMahasiswaNode.addEventListener("change", tabelMahasiswa);
29 </script>
30 </body>

```

Dari sisi tampilan, tidak ada perubahan apapun karena hanya terdapat penambahan tag `<div id="hasil">` yang belum berisi data apapun.

Selanjutnya, saya akan rancang kode program PHP untuk halaman `tabel_mahasiswa.php`:

```

1 <?php
2 // buat koneksi dengan database mysql
3 $link = mysqli_connect("localhost", "root", "", "kampusku");
4
5 // ambil nama mahasiswa dari query string
6 $nama_mahasiswa = $_GET["n"];
7
8 // ambil data dari tabel mahasiswa
9 $query = "SELECT * FROM mahasiswa WHERE nama = '$nama_mahasiswa' ";
10 $result = mysqli_query($link, $query);
11
12 //buat perulangan untuk element tabel dari data mahasiswa
13 while($data = mysqli_fetch_row($result))
14 {
15     // konversi date MySQL (yyyy-mm-dd) menjadi dd-mm-yyyy
16     $tanggal_php = strtotime($data[3]);
17     $tanggal = date("d - m - Y", $tanggal_php);
18
19     // tampilkan data dalam bentuk tabel HTML
20     echo "<table border='1'>";
21     echo "<tr><td>NIM</td><td>$data[0]</td></tr>";
22     echo "<tr><td>Nama</td><td>$data[1]</td></tr>";
23     echo "<tr><td>Tempat Lahir</td><td>$data[2]</td></tr>";
24     echo "<tr><td>Tanggal Lahir</td><td>$tanggal</td></tr>";
25     echo "<tr><td>Fakultas</td><td>$data[4]</td></tr>";
26     echo "<tr><td>Jurusan</td><td>$data[5]</td></tr>";
27     echo "<tr><td>IPK</td><td>$data[6]</td></tr>";

```

```
28     echo "</table>";  
29 }  
30 ?>
```

Jika anda sudah mempelajari PHP-MySQL (misalnya dari buku **PHP Uncover**) tidak akan kesulitan membaca maksud dari kode program diatas.

Setelah membuat koneksi dengan MySQL, saya menangkap query string "n" dengan perintah `$_GET["n"]` dan menyimpannya ke dalam variabel `$nama_mahasiswa`. Variabel ini berisi nama mahasiswa yang dipilih dari tag `<select>`.

Variabel `$nama_mahasiswa` ini selanjutnya diinput ke dalam query "SELECT \* FROM mahasiswa WHERE nama = '\$nama\_mahasiswa'". Hasilnya, MySQL akan menampilkan seluruh kolom tabel untuk mahasiswa tersebut. Data ini kemudian ditampilkan ke dalam bentuk tabel dengan perintah `echo`.

Agar bisa melihat hasil akhir dari `tabel_mahasiswa.php`, kita harus menjalankan file ini dengan tambahan sebuah *query string* "n" , misalnya:

`http://localhost/belajar_ajax/tabel_mahasiswa.php?n=Rudi Permana`

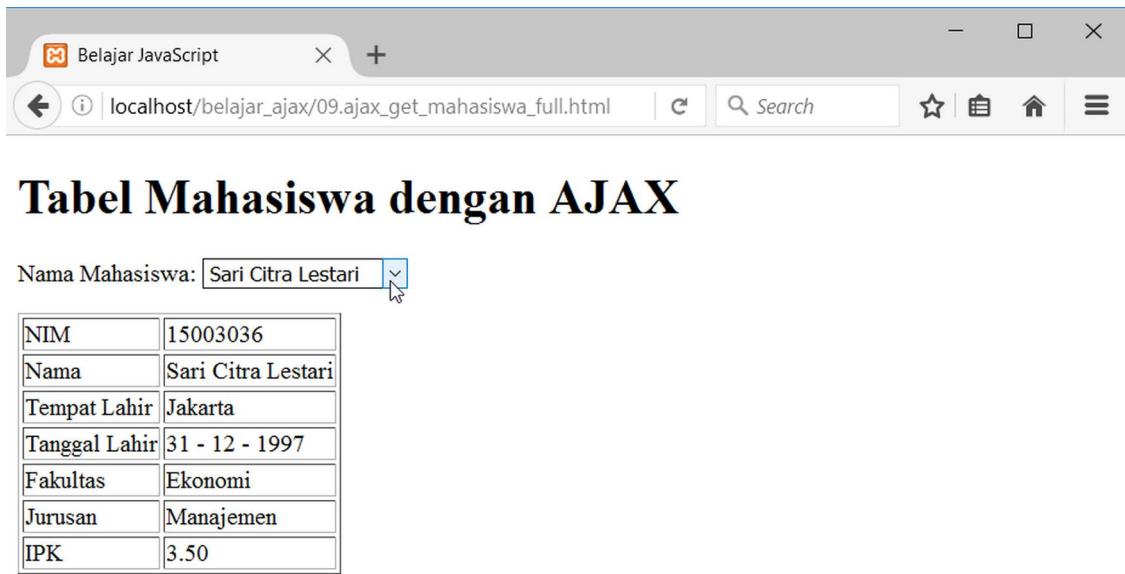


A screenshot of a web browser window. The address bar shows the URL `http://localhost/belajar_ajax/tabel_mahasiswa.php?n=Rudi Permana`. The main content area displays a table with the following data:

NIM	15021044
Nama	Rudi Permana
Tempat Lahir	Bandung
Tanggal Lahir	22 - 08 - 1994
Fakultas	FASILKOM
Jurusan	Ilmu Komputer
IPK	2.90

Gambar: Tampilan tabel data mahasiswa

Oke, tabel sudah tampil sempurna. Mari kita buka kembali halaman HTML:



Gambar: Tabel mahasiswa ditampilkan dengan AJAX

Saat tampil pertama kali, belum ada tabel apapun. Tapi begitu pilihan nama mahasiswa diubah, tabel data mahasiswa akan muncul. Kedua isi tag `<select>` dan tag `<table>` kita ambil menggunakan **AJAX**. Silahkan anda tukar dengan nama mahasiswa lain, isi tabel juga akan berubah.

Jika anda ingin uji lebih lanjut, silahkan update data salah satu mahasiswa ke database menggunakan PHPMyAdmin. Data yang tampil juga akan update (saat terjadi event `change`). Ini membuktikan bahwa AJAX mengambil data asli langsung dari database secara real time.

---

Dalam bab ini kita telah membahas sebuah materi lanjutan JavaScript: **AJAX**, yang tidak lain merupakan gabungan dari **HTML, PHP, dan JavaScript**.

Silahkan anda berkreasi lebih jauh menggunakan AJAX. Misalnya bagaimana dengan membuat sebuah halaman CRUD yang full AJAX? Kode programnya memang cukup rumit, tapi bisa menjadi latihan ideal untuk menguji pemahaman anda tentang HTML, PHP, MySQL, dan JavaScript. Jika memungkinkan, design secantik mungkin menggunakan CSS.

Atau ubah case study menampilkan dropdown fakultas dan jurusan menjadi full AJAX, seperti contoh gambar yang saya tampilkan di awal bab.

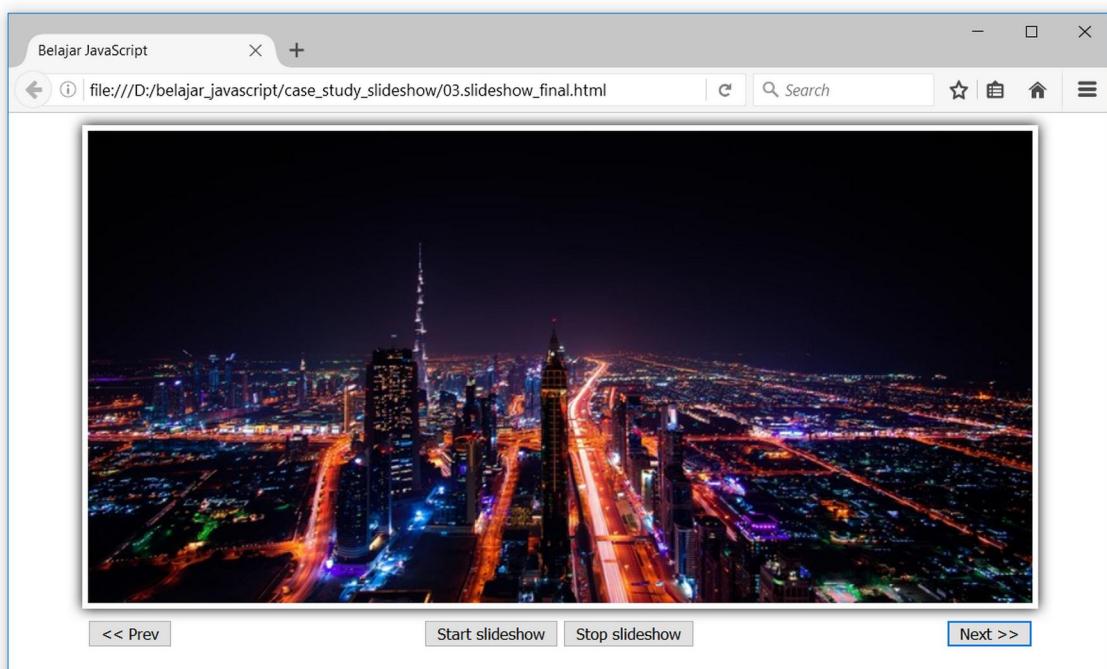
Materi **AJAX** di dalam bab ini sebenarnya juga masih dasar. Saya belum membahas tentang **JSON** (JavaScript Object Notation), sebuah format pertukaran data yang umum digunakan di dalam AJAX. Mudah-mudahan nanti akan hadir buku **JavaScript Uncover Lanjutan** yang akan membahasnya, plus materi advanced JavaScript lain.

# 26. Case Study: SlideShow

Sebagai bab penutup dari buku **JavaScript Uncover**, saya ingin membuat sebuah case study terakhir (sebagai final project sederhana). Kita akan membuat sebuah **slideshow**.

Fitur slideshow sangat sering ditemukan pada website modern. Selain mempercantik tampilan website, slideshow juga bisa digunakan untuk memajang produk atau menampilkan headline berita.

Berikut tampilan slideshow yang akan kita rancang:



Gambar: Slideshow

Perhatikan 4 tombol di bagian bawah: **Prev**, **Start slideshow**, **Stop slideshow**, dan **Next**. Inilah tombol yang akan dibuat menggunakan JavaScript. Berikut penjelasannya:

- **Prev**: mundur ke gambar slide sebelumnya.
- **Start slideshow**: memulai slideshow, dimana gambar akan berganti setiap beberapa detik.
- **Stop slideshow**: menghentikan slideshow.
- **Next**: maju ke gambar slide berikutnya.

- i** Slideshow diatas memang cukup sederhana tanpa efek apapun untuk pergantian gambar (gambar langsung berubah). Jika anda ingin membuat slideshow dengan berbagai animasi dan tampilan, bisa menggunakan library JavaScript **jQuery** dan script seperti [jssor](#)<sup>1</sup>.

## 26.1 Mempersiapkan kode HTML dan CSS

Sebelum masuk ke kode program JavaScript, mari kita rancang tampilannya menggunakan HTML dan CSS:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title> Belajar JavaScript </title>
6   <style>
7     div {
8       margin: 10px auto;
9       width: 800px;
10      height: 400px;
11      display: block;
12      border: 5px solid white;
13      box-shadow: 0px 0px 10px black;
14    }
15   p{
16     text-align: center;
17     width: 800px;
18     margin: 10px auto;
19   }
20   #prev{
21     float: left;
22   }
23   #next{
24     float: right;
25   }
26   </style>
27 </head>
28 <body>
29   <div>
30     <a id="linkshow" href="http://www.google.com">
31       
32     </a>
```

---

<sup>1</sup><http://www.jssor.com>

```
33  </div>
34  <p>
35      <button id="start">Start slideshow</button>
36      <button id="stop">Stop slideshow</button>
37      <button id="prev"><< Prev</button>
38      <button id="next">Next >></button>
39  </p>
40  <script>
41      // Kode JavaScript disini
42  </script>
43 </body>
44 </html>
```

Kode HTML untuk tampilan slideshow ini cukup sederhana. Saya membuat sebuah container dari tag `<div>`. Di dalamnya terdapat tag `<img>` yang diapit dengan tag `<a>`. Dengan dibuat seperti ini, gambar slide juga akan berfungsi sebagai link.

Tag `<img>` memiliki atribut `src="gambar0.jpg"`, sehingga ketika file HTML dijalankan, `gambar0.jpg` akan tampil sebagai gambar awal slideshow.

Tag `<img>` ini berada di dalam tag `<a>` yang memiliki atribut `href="http://www.google.com"`. Artinya jika gambar slide `gambar0.jpg` di klik, halaman akan pindah ke `www.google.com`.

Di dalam tag `<img>` juga terdapat atribut `width="800" height="400"`. Inilah ukuran gambar yang harus disiapkan.

Saya menempatkan 5 gambar untuk pengisi slideshow, dengan nama file: `gambar0.jpg`, `gambar1.jpg`, `gambar2.jpg`, `gambar3.jpg`, dan `gambar4.jpg`. Gambar ini diletakkan dalam folder yang sama dengan file HTML.

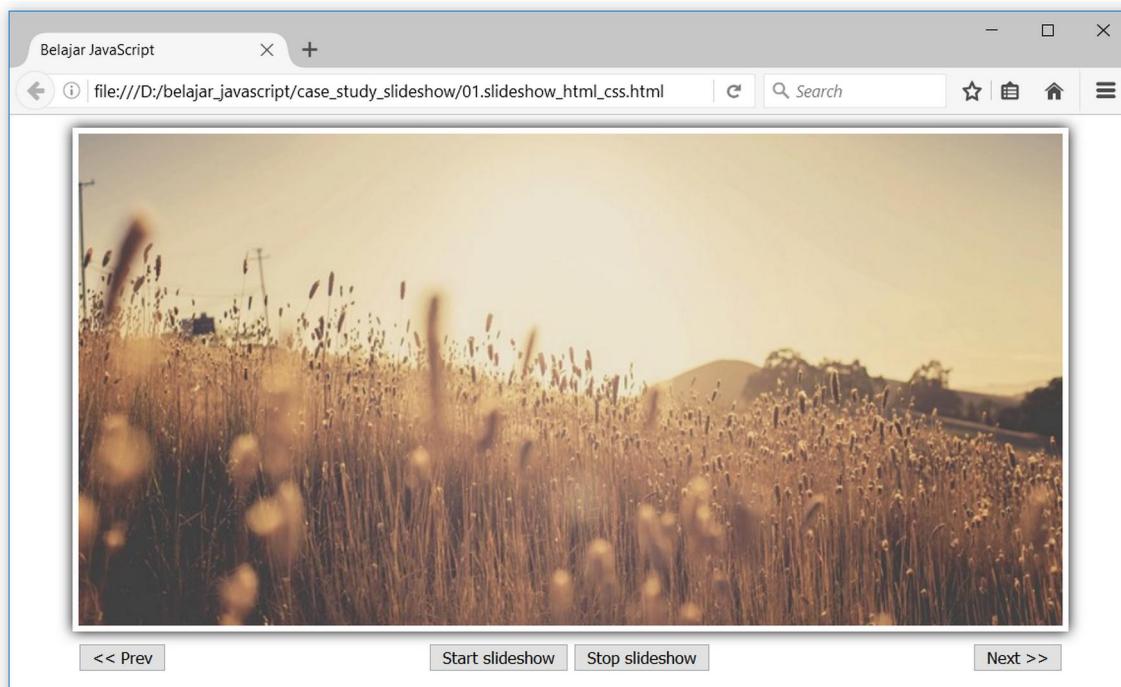


Kelima gambar juga ini tersedia didalam file program buku **JavaScript Uncover: belajar\_javascript.zip** folder `bab_26_case_study_slideshow`.

Tag `<div>` saya hias menggunakan CSS. Kode CSS tersebut berisi property `margin: 10px auto` untuk menempatkan tag `<div>` di tengah halaman, property `border: 5px solid white` untuk membuat bingkai putih setebal 5 pixel, serta property `box-shadow: 0px 0px 10px black` untuk efek bayangan (*shadow*).

Keempat tombol yang akan mengatur jalannya slideshow saya buat menggunakan tag `<button>`. Semua tombol ini berada di dalam tag `<p>` dan diposisikan sedikian rupa menggunakan kode CSS.

Berikut tampilan dari kode diatas:



Gambar: Tampilan slideshow sudah selesai

## 26.2 Merancang Tombol Next dan Prev

Saatnya kita masuk ke kode program JavaScript. Berikut beberapa variabel yang akan saya gunakan:

```
1 var slideshowNode = document.getElementById("slideshow");
2 var linkshowNode = document.getElementById("linkshow");
3 var startNode = document.getElementById("start");
4 var stopNode = document.getElementById("stop");
5 var prevNode = document.getElementById("prev");
6 var nextNode = document.getElementById("next");
7
8 var arrayGambar = ["gambar0.jpg", "gambar1.jpg",
9                     "gambar2.jpg", "gambar3.jpg", "gambar4.jpg"];
10 var arrayLink = ["http://www.google.com",
11                  "http://www.twitter.com",
12                  "http://www.facebook.com",
13                  "http://www.bing.com",
14                  "http://www.duniailkom.com"];
15
16 var counter = 0;
```

Enam baris pertama berisi variabel untuk mengakses *node object* dari setiap tag HTML. Dua variabel pertama untuk *node object* tag `<a>` dan `<img>`, serta empat variabel untuk *node object* tombol pengatur slideshow.

Setelah itu terdapat variabel `arrayGambar` dan `arrayLink`. Disinilah nama gambar dan alamat link slideshow disimpan. Urutan gambar dan link akan saling berhubungan. Jika `gambar0.jpg` di klik, akan menuju ke `google.com`. Jika `gambar1.jpg` di klik, akan menuju ke `twitter.com`, dst. Untuk 5 gambar, dibutuhkan 5 link.

Terakhir, variabel `counter` diisi nilai awal 0. Variabel `counter` nantinya berfungsi sebagai pengatur posisi gambar dalam slideshow.

Baik, mari masuk ke kode program untuk tombol **Next**:

```

1 function nextSlideshow(){
2     counter++;
3     if (counter === 5) {
4         counter = 0;
5     }
6     slideshowNode.setAttribute("src", arrayGambar[counter]);
7     linkshowNode.setAttribute("href", arrayLink[counter]);
8 }
9
10 nextNode.addEventListener("click", nextSlideshow);

```

Ide dari tombol **Next** ini adalah: jika di klik, ganti atribut `src` dan `href` ke index array selanjutnya.

Atribut `src` dari tag `<img>` digunakan untuk menginput lokasi dan nama gambar. Gambar pertama bisa diakses dari `arrayGambar[0]`, gambar kedua di `arrayGambar[1]`, dst hingga gambar kelima di `arrayGambar[4]`. Kembali kita ingat bahwa index array dimulai dari 0, bukan 1.

Begini juga halnya dengan atribut `href` sebagai link dari tag `<a>`. Alamat link pertama saya simpan di `arrayLink[0]`, alamat link kedua di `arrayLink[1]`, hingga alamat link kelima di `arrayLink[4]`.

Sekarang, bagaimana caranya agar isi atribut `src` dan `href` bisa naik 1 angka setiap kali tombol **Next** di klik? Inilah isi dari function `nextSlideshow()`. Function ini akan dijalankan untuk event `click` dari tombol **Next**.

Saat function `nextSlideshow()` di panggil, variabel `counter` akan dinaikkan sebanyak 1 angka menggunakan perintah `counter++`. Kemudian terdapat kondisi `if (counter === 5) { counter = 0; }`. Kondisi ini berfungsi untuk me-reset angka `counter` menjadi 0 jika sudah bernilai 5.

Nilai dari variabel `counter` dipakai untuk 2 baris selanjutnya, dimana saya mengubah atribut `"src"` dari tag `<img>` menjadi `arrayGambar[counter]`, dan atribut `"href"` dari tag `<a>` menjadi `arrayLink[counter]`.

Artinya, setiap kali function `nextSlideshow()` dijalankan, atribut `"src"` dan `"href"` akan naik ke index berikutnya. Mulai dari `arrayGambar[1]`, `arrayGambar[2]`, `arrayGambar[3]`, dan terus menaik karena perintah `counter++`.

Tapi bagaimana jika tombol **Next** terus di tekan berulang kali hingga lebih dari 5? Padahal isi dari `arrayGambar` hanya ada 5 gambar. Inilah fungsi dari kondisi `if (counter === 5) { counter = 0; }`. Jika tombol **Next** terus di klik, nilai variabel `counter` akan dikembalikan ke 0. Hasilnya, gambar slideshow akan kembali ke gambar awal.

Untuk tombol **Prev**, caranya juga hampir sama:

```

1 function prevSlideshow(){
2   counter--;
3   if (counter === -1) {
4     counter = 4;
5   }
6   slideshowNode.setAttribute("src",arrayGambar[counter]);
7   linkshowNode.setAttribute("href",arrayLink[counter]);
8 }
```

Hanya saja kali ini nilai variabel counter akan dikurangi 1 angka setiap kali tombol **Prev** di klik. Ini dijalankan dari perintah counter--.

Kondisi untuk me-reset angka counter juga disesuaikan menjadi **if (counter === -1) { counter = 4; }**. Perintah counter-- menyebabkan nilai variabel counter akan terus berkurang dan apabila sudah mencapai -1, kembalikan menjadi 4. Hasilnya, jika tombol **Prev** di tekan berulang kali, gambar slideshow akan kembali ke posisi awal.

Silahkan anda test kode diatas di web browser. Saat tombol **Next** dan **Prev** di klik, gambar slideshow juga akan berubah.

## 26.3 Merancang Tombol Start slideshow dan Stop slideshow

Kode program untuk tombol **Start slideshow** dan **Stop slideshow** sebenarnya cukup sederhana, kita tinggal memakai method **setInterval()** untuk memanggil function **nextSlideshow()**. Pemanggilan ini dilakukan setiap durasi tertentu (misalnya 2 detik sekali). Jika ingin menghentikan slideshow, cukup panggil method **stopInterval()**.

Pertama, saya butuh variabel tambahan :

```
1 var intervalID = 0;
```

Sesuai namanya, variabel **intervalID** digunakan untuk menampung **ID interval** hasil pemanggilan method **setInterval()**. Variabel ini nantinya digunakan sebagai input untuk method **stopInterval()**.

Selanjutnya kita masuk ke kode program untuk tombol **Start slideshow**:

```

1 function startSlideshow(){
2   if (intervalID === 0) {
3     intervalID = setInterval(nextSlideshow,2000);
4   }
5 }
6
7 startNode.addEventListener("click", startSlideshow);
```

Function `startSlideshow()` akan berjalan untuk event `click` dari `startNode`.

Di dalam function terdapat sebuah kondisi `if (intervalID === 0)`. Jika kondisi ini terpenuhi, jalankan perintah `intervalID = setInterval(nextSlideshow, 2000)`. Kode tersebut akan memanggil function `nextSlideshow()` secara berulang setiap 2000 milidetik sekali, atau setiap 2 detik.

Kondisi `if (intervalID === 0)` saya tambahkan untuk menghindari kemungkinan function `setInterval()` berjalan lebih dari 1 kali. Ini bisa terjadi seandainya tombol **Start slideshow** di klik berulang kali.

Berikutnya kode program untuk tombol **Stop slideshow**:

```
1 function stopSlideshow(){
2     clearInterval(intervalID);
3     intervalID = 0;
4 }
5
6 stopNode.addEventListener("click", stopSlideshow);
```

Hanya butuh 2 baris. Untuk menghentikan slideshow, saya menggunakan perintah `clearInterval(intervalID)`. Nilai `intervalID` didapat dari function `startSlideshow()` sebelumnya.

Baris `intervalID = 0` digunakan sebagai penanda bahwa slideshow telah berhenti. Hanya jika nilai `intervalID` menjadi nol, barulah perintah `setInterval()` untuk tombol **Start slideshow** bisa di proses kembali.

---

Dan... selesai sudah kode program untuk membuat slideshow :)

Sebagai latihan, silahkan anda modifikasi kode program slideshow ini. Misalnya menambah gambar hingga 10 buah, atau membuat 1 tombol lagi untuk membuat slideshow berjalan mundur. Bisa juga dengan membuat durasi slideshow menjadi 10 detik, kemudian tampilkan angka hitung mundur (*countdown*) setiap detiknya.

Silahkan berkreasi dan asah kemampuan JavaScript yang telah anda pelajari sepanjang buku ini. Lebih bagus lagi jika dikombinasikan dengan materi lain seperti HTML, PHP dan CSS.

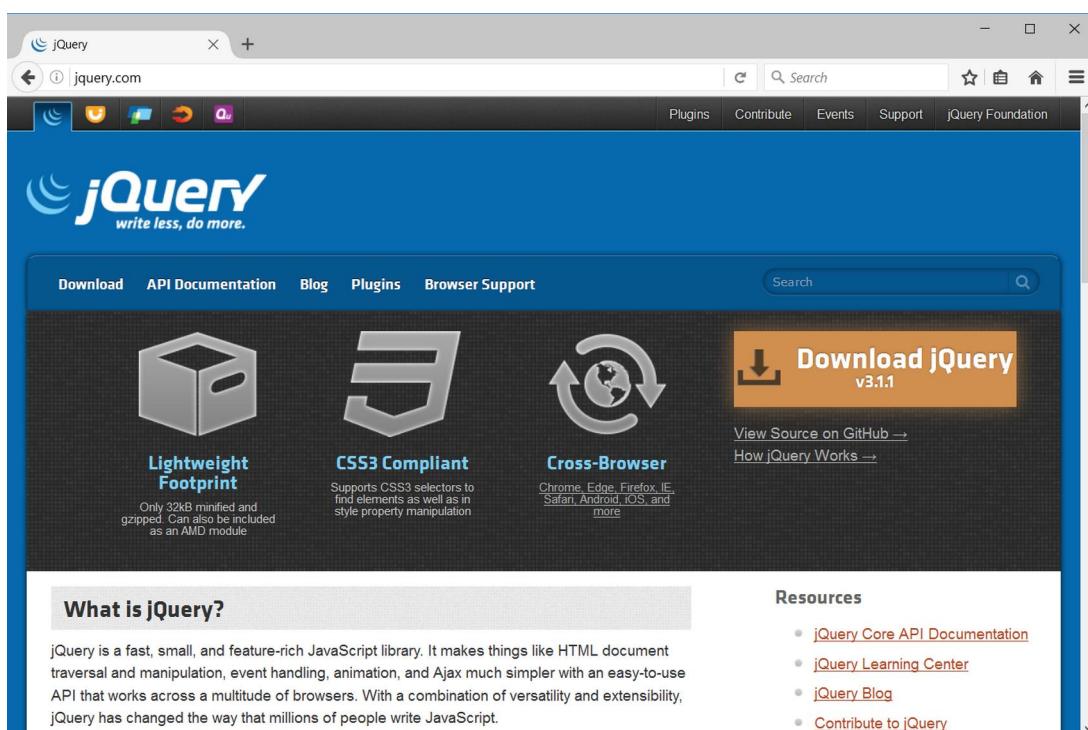
# Penutup: JavaScript Uncover

Tidak terasa, sudah 662 halaman saya membahas JavaScript di buku **JavaScript Uncover** ini. Sedikit banyak semoga anda bisa mendapat gambaran tentang apa itu JavaScript dan apa yang bisa dilakukan dengan JavaScript. Walaupun begitu, materi ini barulah ‘segelintir’ dari JavaScript.

Seperti yang pernah saya bahas di bab-bab awal buku, JavaScript saat ini menjadi pusat perhatian di dunia web programming. Begitu banyak teknologi yang berkembang di sekitar JavaScript. Mulai dari library, plugin, hingga framework.

Berikut beberapa materi advanced JavaScript yang bisa dipelajari:

## jQuery



Gambar: Tampilan halaman awal website jquery.com

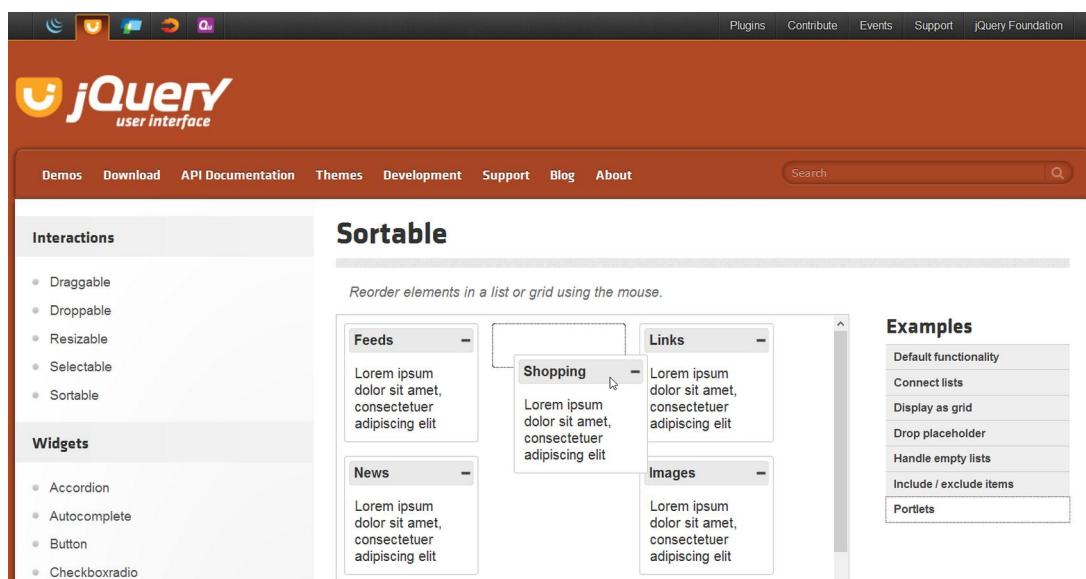
Saat ini tidak lengkap jika ada yang mengaku paham JavaScript tapi tidak tahu apa itu **jQuery**<sup>2</sup>. Kira-kira itulah pentingnya **jQuery** di dalam JavaScript.

**jQuery** adalah sebuah library yang berisi berbagai fungsi tambahan JavaScript. Apa yang bisa dilakukan dengan **jQuery** sebenarnya juga bisa dibuat dengan JavaScript saja, tapi jika

<sup>2</sup><http://jquery.com>

menggunakan **jQuery** akan jauh lebih singkat dan cepat. Ini sesuai dengan tagline **jQuery: write less, do more.**

Di tambah dengan materi **jQuery UI** (User Interface), kita bisa membuat berbagai efek menarik, seperti slider, tab, tooltip, autocomplete, hingga berbagai efek animasi.



Gambar: Fitur Sortable dari jQuery UI



Jika anda ingin “mencicipi” **jQuery**, bisa lanjut ke [Tutorial Belajar jQuery di DuniaIlkom<sup>3</sup>](#).

Mudah-mudahan nanti juga akan hadir buku khusus tentang **jQuery Uncover**. Tentunya dengan pembahasan yang lebih detail daripada yang ada di web duniaIlkom.

## HTML5 API (Application Program Interface)

**HTML5 API** mirip seperti **DOM** dan **BOM** (Browser Object Model), yang merupakan tambahan ke dalam JavaScript (bukan bagian dari inti JavaScript).

Jika anda pernah membaca buku **HTML Uncover**, disana terdapat bab tentang tag `<canvas>`. Untuk menggunakan tag `<canvas>`, harus dengan JavaScript. Inilah salah satu area penggabungan antara HTML5 dengan JavaScript.

Selain tag `<canvas>`, juga terdapat materi **HTML5** lain yang harus di program dengan JavaScript, contohnya: **Geolocation**, **Drag/Drop**, **LocalStorage**, **Web Workers** dan **Server-Sent Events**. Semua teknologi ini merupakan bagian dari **HTML5 API**.

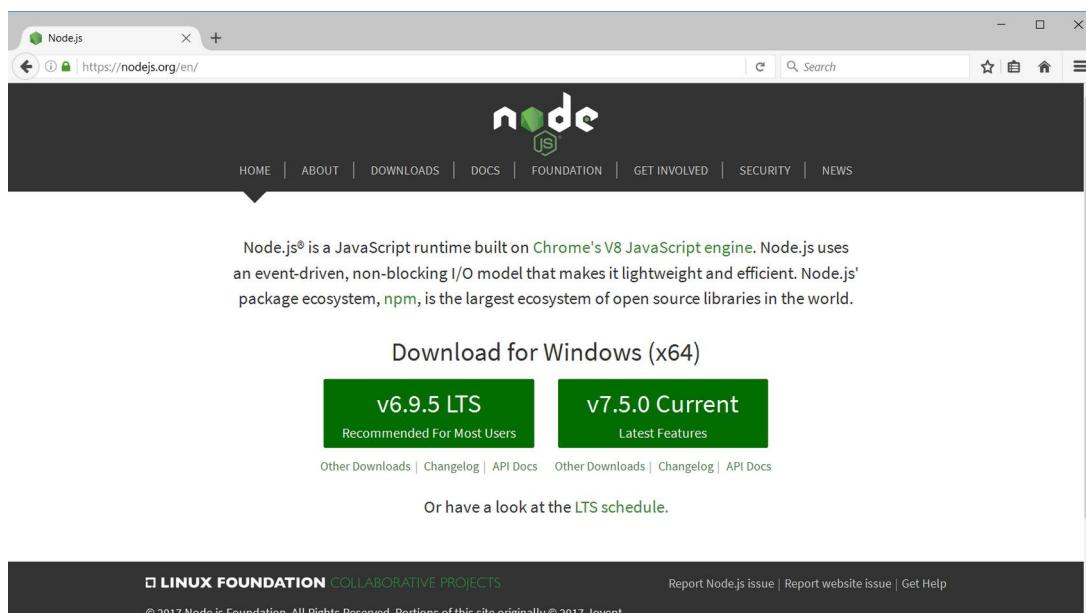
<sup>3</sup><http://www.duniaIlkom.com/tutorial-belajar-jquery-bagi-pemula/>

## The HTML 5 JavaScript API Index



Gambar: Beberapa materi dari HTML5 API ([html5index.org](http://html5index.org))

## Node.js



Gambar: Tampilan halaman awal website nodejs.org

**Node.js**<sup>4</sup> adalah penggunaan JavaScript di sisi server, yang bisa dibilang sebagai pengganti dari PHP. Salah satu keunggulan **Node.js** adalah *Event-driven, Non-Blocking I/O*.

Jika kita memberikan perintah untuk mengambil data ke database di PHP, perintah lain harus menunggu sampai proses tersebut selasai. Setelah itu PHP akan lanjut mengeksekusi program berikutnya.

Di **Node.js**, (biasanya) proses menunggu ini tidak terjadi. **Node.js** bisa memproses kode program lain, walaupun pengambilan data ke database sedang berlangsung.

Namun **Node.js** juga lebih rumit jika dibandingkan dengan PHP. Sebagai contoh, untuk membuat tampilan teks "Hello World!" di PHP, kodennya adalah sebagai berikut:

<sup>4</sup><http://nodejs.org>

```
1 <?php  
2 echo "Hello World!";  
3 ?>
```

Menggunakan **Node.js**, kode yang dibutuhkan jauh lebih panjang:

```
1 var http = require('http');  
2 http.createServer(function (req, res) {  
3     res.writeHead(200, {'Content-Type': 'text/plain'});  
4     res.end('Hello World!');  
5 }).listen(3000, '127.0.0.1');
```

Dengan semakin populernya website modern yang menggunakan **Node.js**, cepat atau lambat kita juga akan dituntut menguasai teknologi ini.

---

Selain materi diatas, masih banyak library, dan framework JavaScript yang saat ini sedang “booming”, seperti **Angular.js**<sup>5</sup> kepunyaan **Google**, maupun **React.js**<sup>6</sup> yang dikembangkan **Facebook**. Berbekal materi dasar JavaScript dari buku ini, silahkan anda lanjutkan *petualangan* mempelajari dunia JavaScript.

Akhir kata, semoga materi yang ada di buku **JavaScript Uncover** ini bisa bermanfaat. Mohon maaf jika ada kata-kata yang salah dan kekurangan di sana-sini.

Terimakasih atas dukungannya dengan membeli versi asli eBook **JavaScript Uncover**. Sampai jumpa di buku Duniailkom selanjutnya :)

---

<sup>5</sup><https://angularjs.org/>

<sup>6</sup><https://facebook.github.io/react/>