Javascript Fundamentals

What is Javascript?

Javascript dibuat hanya dalam 10 hari oleh Brendan Eich pada tahun 1995 ketika Eich sedang mengerjakan Netscape Navigator. Netscape Navigator adalah salah satu browser web pertama di Internet. Sebelum pembuatan Javascript oleh Eich, situs web hanya ada sebagai halaman HTML dan CSS. Javascript awalnya disebut LiveScript, tetapi diubah kembali ke Javascript sebagai keputusan pemasaran untuk mendukung popularitas Java pada saat itu.

Ketika bahasa pemrograman Javascript berkembang, versi yang lain muncul. Javascript akhirnya dibawa ke AGMA internasional sehingga standar resmi untuk bahasa Javascript dapat dibentuk. Saat ini, bahasa itu sendiri masih disebut sebagai Javascript, tetapi versi terbaru dirujuk oleh nomor dari versi ECMAScript mereka, seperti ES5 atau ES6. Bahkan baru-baru ini, standar telah beralih ke versi berbasis tahun untuk mempromosikan siklus rilis yang lebih konsisten. JavaScript sekarang memiliki ES2016, ES2017, dan seterusnya. dan JavaScript telah berkembang menjadi salah satu bahasa paling populer di dunia, dan itu dianggap sebagai salah satu dari tiga pilar dasar pengembangan web front-end.

Developer Tools

Tahukah Anda, setiap browser web modern menyertakan seperangkat alat Developer Tools sendiri?

Jika tidak, tidak apa-apa. Developer Tools bukan merupakan hal termudah untuk ditemukan di browser Anda. Jadi, kami telah memutuskan untuk membantu Anda dengan membuat panduan ini untuk membuka Developer Tools

Google Chrome

Chrome DevTools adalah seperangkat alat penulisan web dan proses debug yang disertakan dalam Google Chrome. Gunakan DevTools untuk mengiterasi, men-debug, dan membuat profil situs Anda. **Pelajari Chrome DevTools lebih lanjut di sini**.

Untuk membuka Chrome DevTools, klik kanan pada elemen halaman mana pun dan pilih `Inspect` atau buka menu setelan Chrome di sudut kanan atas jendela browser Anda dan pilih `More Tools > Developer Tools`. Atau, Anda dapat menggunakan shortcut:

- `Command + Option + i` (Mac)
- `Ctrl + Shift + i` (Windows/Linux).

Mozila Firefox

Firefox Developer Tools memungkinkan Anda untuk memeriksa, mengedit, dan men-debug HTML, CSS, dan JavaScript di desktop dan di ponsel. Selain itu, Anda dapat mengunduh versi Firefox yang disebut Firefox Developer Edition yang disesuaikan untuk developer, menampilkan fitur Firefox terbaru dan alat pengembang eksperimental. **Pelajari selengkapnya tentang Mozilla Firefox DevTools di sini**.

Untuk membuka Alat Firefox Developer Tools, klik kanan pada elemen halaman mana saja dan pilih `Inspect Element` atau buka menu pengaturan Firefox di sudut kanan atas jendela browser Anda dan pilih `Developer`. Atau, Anda dapat menggunakan shortcut:

- `Command + Option + i` (Mac)
- `Ctrl + Shift + i` (Windows/Linux).

Microsoft Edge

Microsoft Edge memperkenalkan improvement baru pada Developer Tools F12 yang terlihat di Internet Explorer. Improvement tersebut dibangun di TypeScript, dan selalu berjalan, jadi tidak diperlukan reload. Selain itu, dokumentasi Developer Tools F12 sekarang sepenuhnya tersedia di GitHub. Sama seperti Internet Explorer, untuk membuka alat pengembang di Microsoft Edge cukup tekan F12. Pelajari selengkapnya tentang Microsoft Edge DevTools di sini.

Console.log

Developer tools memungkinkan anda untuk men-debug dan menguji ide-ide anda langsung di browser. Jika anda terbiasa dengan HTML atau CSS, anda mungkin telah menggunakan developer tools untuk bereksperimen dengan style dari web. Tetapi anda juga dapat menggunakannya dengan JavaScript.

Developer tools sering digunakan sebagai sandbox. Dengan kata lain, tempat untuk mencoba apapun dengan kode apapun tanpa konsekuensi jangka panjang. Anda dapat menggunakan developer tools untuk men-debug masalah yang anda hadapi atau untuk menguji sepotong kode yang baru saja anda pelajari. Jika anda membuka situs web apa pun yang menggunakan JavaScript, console akan memberitahu anda jika ada peringatan atau kesalahan dan juga akan menampilkan output apa pun dengan console.log.

`console.log` digunakan untuk menampilkan konten ke **konsol JavaScript**. Jalankan kode berikut di console:

```
console.log("hiya friend!");
```

Prints: "hiya friend!"

Untuk pengguna Chrome, jika anda tidak melihat outputnya, klik *"Default levels"* di konsol dan pastikan "*Info*" dicentang.

Pesan yang Anda cetak adalah "hiya friend!". "hiya friend!" adalah **string** (urutan karakter).

Data Types & Variables

Data dan tipe data adalah fondasi dari setiap bahasa pemrograman, karena mereka membantu kita mengatur informasi dan menentukan bagaimana program kita akan berjalan. Penting untuk mengetahui tipe data apa yang kita gunakan dan kapan waktu yang tepat untuk menggunakannya.

Numbers

Number

Mendefinisikan angka di JavaScript sebenarnya cukup sederhana. Tipe data angka/number mencakup bilangan bulat positif atau negatif, serta desimal. Memasukkan angka ke console akan mengembalikan nilainya kepada Anda.

```
3
```

Returns: 3

Arithmetic operations

Kita juga dapat melakukan perhitungan angka dengan cukup mudah. Pada dasarnya ketikan operator matematika seperti yang kita ketikkan di kalkulator.

```
3 + 2.1
```

Returns: 5.1

Comparing numbers

Bagaimana dengan membandingkan angka? Bisakah kita melakukannya? Tentu saja bisa! Sama seperti dalam matematika, kita dapat membandingkan dua angka untuk melihat apakah salah satunya lebih besar, lebih kecil, atau sama dengan yang lain.

```
5 > 10
```

Returns: false

```
5 < 10
```

Returns: true

```
5 == 10
```

Returns: false

Perbandingan antara angka akan dievaluasi menjadi benar atau salah. Berikut beberapa contoh lainnya :

Operator	Deskripsi
<	Less than (Lebih kecil dari)
>	Greater than (Lebih besar dari)
<=	Less than or Equal to (Lebih kecil atau sama dengan)
>=	Greater than or Equal to (Lebih besar atau sama dengan)
==	Equal to (Sama dengan)
!=	Not Equal to (Tidak sama dengan)

Comments

Kita dapat menggunakan **komentar** untuk membantu menjelaskan kode dan membuatnya lebih jelas. Dalam JavaScript, komentar ditandai dengan garis miring ganda //. Apa pun yang ditulis pada baris yang sama setelah // tidak akan dieksekusi atau ditampilkan. Agar komentar mencakup beberapa baris, tandai awal komentar dengan garis miring dan bintang, lalu sertakan komentar di dalam bintang dan garis miring / *...* /.

```
// this is a single-line comment

/*
this is
a multi-line
comment
*/
```

Strings

Saat kita mengetik pesan di dalam console.log, pesan itu sebenarnya adalah **String**. String dapat berupa huruf tunggal, seperti karakter h, atau bahkan mengandung angka, seperti '123'. Yang penting adalah menggunakan tanda kutip untuk menandakan sebuah string. Tidak masalah apakah tanda kutip tunggal atau ganda, tetapi tanda kutip tersebut harus cocok. Jika kita mencoba menuliskan string ke console.log tetapi lupa menggunakan tanda kutip, JavaScript akan menampilkan error.

String Concatenation

String adalah kumpulan karakter yang berada di dalam tanda kutip ganda atau tunggal. Kita dapat menggunakan string untuk merepresentasikan data seperti kalimat, nama, alamat, dan lainnya. Tahukah anda bahwa kita dapat menggabungkan 2 buah string? Dalam JavaScript, ini disebut concatenating. Menggabungkan dua string bersama sebenarnya cukup sederhana!

```
"Hello," + " New York City"
```

Returns: "Hello, New York City"

Variables

Dengan variabel, kita tidak perlu lagi bekerja dengan data sekali pakai. Di awal kursus ini, kita mendeklarasikan nilai string, tetapi tidak memiliki cara untuk mengakses atau menggunakan kembali string tersebut nanti.

```
"Hello"; // Here's a String "Hello"
"Hello" + " World"; // Here's a new String (also with the value
"Hello") concatenated with " World"
```

Menyimpan nilai string dalam variabel seperti membungkusnya untuk digunakan nanti.

```
var greeting = "Hello";
```

Sekarang, jika kita ingin menggunakan "Halo" dalam berbagai kalimat, kita tidak perlu menggandakan string "Halo". Kita bisa menggunakan kembali variabel greeting.

```
greeting + " World!";
```

Returns: Hello World!

```
greeting + " Mike!";
```

Returns: Hello Mike!

Anda juga dapat mengubah awalan greeting dengan mengubah kembali nilai string baru ke variabel greeting.

```
greeting = "Hola";
greeting + " World!";
```

Returns: Hola World!

```
greeting + " Mike!";
```

Returns: Hola Mike!

Naming conventions

Saat membuat variabel, kita menulis nama variabel menggunakan camelCase (kata pertama adalah huruf kecil, dan semua kata berikutnya adalah huruf besar). Usahakan untuk menggunakan nama variabel yang akurat, tetapi secara ringkas mendeskripsikan tentang apa data tersebut**.**

```
var totalAfterTax = 53.03; // menggunakan camelCase apabila nama variabel terdiri dari beberapa kata
var tip = 8; // menggunakan huruf kecil apabila nama variabel menggunakan satu kata
```

Tidak menggunakan camelCase untuk nama variabel kita tidak akan merusak apa pun di JavaScript. Namun ada panduan yang direkomendasikan dan digunakan di semua bahasa pemrograman yang membantu menjaga kode tetap konsisten, bersih, dan mudah dibaca. Ini sangat penting saat mengerjakan proyek yang lebih besar yang akan diakses oleh banyak developer.

String Index

Indexing

Tahukah Anda bahwa kita dapat mengakses suatu karakter dalam sebuah string? Untuk mengakses suatu karakter, kita dapat menggunakan lokasi karakter dalam string, yang disebut **Index**. Masukan indeks karakter di dalam tanda kurung siku (dimulai dengan [0] sebagai karakter pertama). Misalnya:

```
"James"[0];
```

Returns: "J"

atau umumnya, kita akan melihatnya seperti ini, dengan menggunakan variabel:

```
var name = "James";
name[0];
```

Returns: "J"

Escaping String

Ada beberapa kasus di mana kita mungkin ingin membuat string yang bukan hanya angka dan huruf. Misalnya, bagaimana jika kita ingin menggunakan tanda kutip dalam sebuah string?

```
"The man whispered, "please speak to me.""
```

Uncaught SyntaxError: Unexpected identifier

Jika kita mencoba menggunakan tanda kutip dalam sebuah string, kita akan menerima SyntaxError seperti di atas. Karena kita perlu menggunakan tanda kutip untuk awalan dan akhiran string, mesin JavaScript salah mengartikan string kita dengan berpikir "The man whispered, " adalah stringnya. Kemudian, ia melihat sisanya please speak to me."" dan mengembalikan SyntaxError.

Jika kita ingin menggunakan tanda kutip di dalam string, dan agar JavaScript tidak salah memahami maksud kita. Kita memerlukan cara lain untuk menulis tanda kutip. Untungnya, JavaScript memiliki cara untuk melakukannya dengan menggunakan karakter garis miring terbalik (\ ).

Escaping Characters

Di JavaScript, kita menggunakan garis miring terbalik untuk keluar dari karakter lain (Escaping Character). Escaping Character memberi tahu JavaScript untuk mengabaikan arti khusus karakter tersebut dan hanya menggunakan nilai literal dari karakter tersebut. Ini berguna untuk karakter yang memiliki arti khusus seperti pada contoh kita sebelumnya dengan tanda kutip "...". Karena tanda kutip digunakan untuk menandai awal dan akhir string, kita dapat menggunakan karakter garis miring terbalik untuk keluar dari tanda kutip untuk mengakses karakter tanda kutip literal.

```
"The man whispered, \"please speak to me.\""
```

Returns: The man whispered, "please speak to me."

Dengan ini, mesin JavaScript tidak salah mengartikan string dan tidak menghasilkan kesalahan.

Special characters

Tanda kutip bukan satu-satunya karakter khusus yang perlu diloloskan, sebenarnya ada beberapa. Namun, agar tetap sederhana, berikut adalah daftar beberapa karakter khusus yang umum dalam JavaScript.

Code	Character
\\	\ (backslash)
\"	" (double quote)
\'	' (single quote)
\n	newline
\t	tab

Dua karakter terakhir pada tabel, baris baru \n dan tab \t, adalah karakter unik karena karakter tersebut menambahkan spasi tambahan ke String kita. Karakter \n akan menambahkan jeda baris dan karakter \t akan memajukan baris kita ke tab berikutnya.

```
"Up up\n\tdown down"
```

Returns:

Up up

down down

Comparing String

Cara lain untuk bekerja dengan string adalah dengan membandingkannya. Kita telah melihat operator perbandingan == dan != saat kita membandingkan angka untuk persamaan. Kita juga dapat menggunakannya dengan string! Sebagai contoh, mari bandingkan string "Yes" dengan "yes".

```
"Yes" == "yes"
```

Returns: false

Saat Anda menjalankan ini di konsol, hasilnya false. Mengapa demikian? "Yes" dan "yes" adalah string yang sama, bukan? Tidak juga.

A. Case-sensitive

Saat kita membandingkan string, huruf besar-kecil adalah penting. Meskipun kedua string menggunakan huruf yang sama (dan huruf tersebut muncul dalam urutan yang sama), huruf pertama pada string pertama adalah huruf kapital Y sedangkan huruf pertama pada string kedua adalah huruf kecil y.

```
'Y' != 'y'
```

Returns: true

B. Internal Working

Dalam Javascript, string dibandingkan karakter dengan karakter dalam urutan abjad. Setiap karakter memiliki nilai numerik tertentu, berasal dari nilai ASCII dari karakter yang dapat dicetak. Misalnya, karakter 'A' memiliki nilai 65, dan 'a' memiliki nilai 97. Kita dapat melihat bahwa huruf kecil memiliki nilai ASCII yang lebih tinggi daripada karakter huruf besar. Jika kita ingin mengetahui nilai ASCII dari karakter tertentu, Anda dapat mencoba menjalankan kode di bawah ini:

```
// Pick a string. Your string can have any number of characters.
var my_string = "a";

// Calculate the ASCII value of the first character, i.e. the character at the position 0.
var ASCII_value = my_string.charCodeAt(0);

// Let us print
console.log(ASCII_value);
```

Pada contoh di atas, jika kita ingin mencetak nilai ASCII dari semua karakter dalam string, kita harus menggunakan Loops/Perulangan. Sekedar referensi, berikut adalah cara menggunakan perulangan untuk mencetak nilai ASCII dari semua karakter dalam sebuah string.

```
var my_string = "Udacity";

// Iterate using a Loop
for (var i = 0; i < my_string.length; i++) {
  console.log(my_string.charCodeAt(i));
}
```

Nilai ASCII [A-Z] termasuk dalam kisaran [65-90], sedangkan nilai ASCII [a-z] termasuk dalam kisaran [97-122]. Oleh karena itu, saat kita membandingkan string, perbandingan terjadi karakter demi karakter untuk nilai ASCII.

Booleans

Variabel boolean hanya bisa mengambil salah satu dari dua nilai - true atau false. Misalnya,

```
var studentName = "John";
```

```
var haveEnrolledInCourse = true;
var haveCompletedTheCourse = false;
```

Variabel boolean sangat penting dalam mengevaluasi hasil dari kondisi (perbandingan). Hasil perbandingan selalu berupa variabel boolean. Kita akan mempelajari kondisional dalam pelajaran mendatang, tetapi mari kita lihat contoh sebelumnya untuk memahami peran boolean dalam kondisional:

```
if (haveEnrolledInCourse){
     console.log("Welcome  "+studentName+"  to  Udacity!");  //
Will run only if haveEnrolledInCourse is true
}
```

Mari kita lihat contoh yang akan menjelaskan peran variabel boolean sebagai perbandingan.

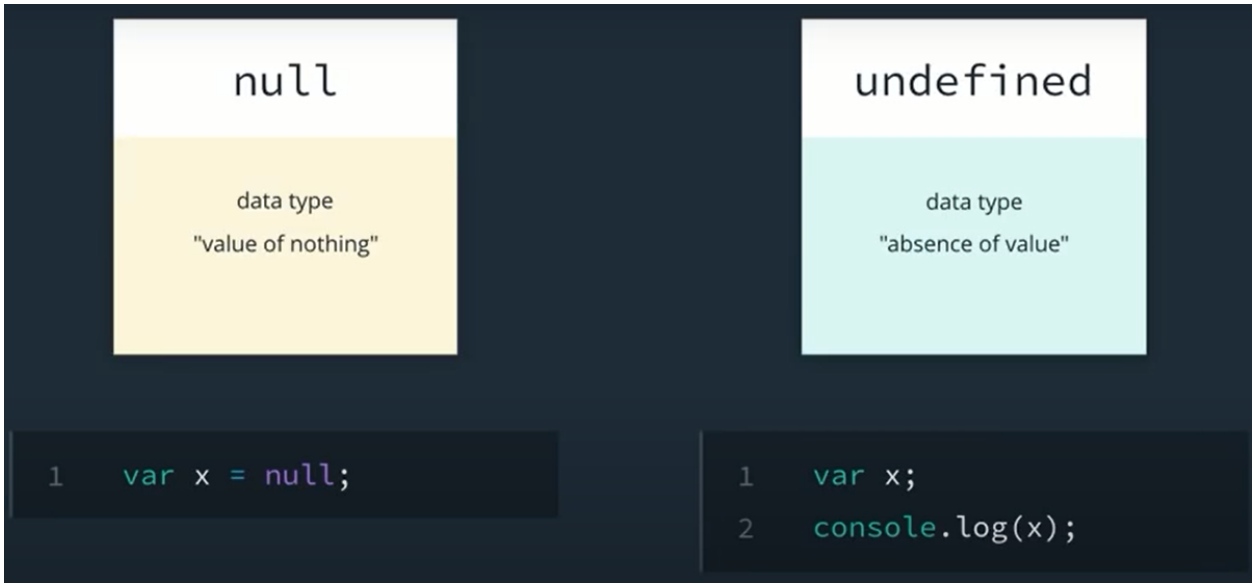
```
var a = 10;
var b = 20;
// a comparison - we will study this in detail in upcoming
lesson
if (a>b) // The outcome of a>b will be a boolean
     console.log("Variable `a` has higher value"); // if a>b is
true
else
     console.log("Variable `b` has higher value"); // if a>b is
false
```

Dalam kasus umum, **true** sesuai dengan angka 1, sedangkan **false** mewakili angka 0. Misalnya:

```
if(1){
     console.log("This  statement  will  always  execute  because
conditional is set to 1 i.e., true");
}

if(0){
          console.log("This statement will NEVER execute because
conditional is set to 0 i.e., false");
}
```

Null, Undefined and NaN



null = nilai yang menunjukan bahwa nilai tersebut kosong atau tidak ada

undefined = tidak ada nilai yang dideklarasikan

NaN = "Not-A-Number" dan sering dipakai untuk menunjukkan kesalahan dengan operasi angka. Misalnya, jika kita menulis beberapa kode yang melakukan perhitungan matematika, dan perhitungan tersebut gagal menghasilkan angka yang valid, NaN mungkin dikembalikan.

```
// calculating the square root of a negative number will return
NaN
Math.sqrt(-10)

// trying to divide a string by 5 will return NaN
"hello"/5
```

Equality

Sejauh ini, kita telah melihat bagaimana kita bisa menggunakan == dan != untuk membandingkan angka dan string untuk persamaan. Namun, jika kita menggunakan == dan != dalam situasi dimana nilai yang kita bandingkan memiliki tipe data yang berbeda, ini dapat menghasilkan beberapa hasil yang menarik. Misalnya,

```
"1" == 1
```

Returns: true

```
0 == false
```

Returns: true. Operator == tidak dapat membedakan 0 dengan false.

```
' ' == false
```

Returns: true. Kedua nilai diubah menjadi nol terlebih dahulu sebelum perbandingan.

Ketiga contoh di atas bernilai benar. Alasan untuk hasil yang menarik tersebut adalah Konversi Jenis (Type Conversion). Dalam kasus perbandingan reguler, nilai di kedua sisi operator == pertama-tama dikonversi menjadi angka, sebelum perbandingan. Oleh karena itu ' ' dianggap false, dan 0 semuanya dianggap sama. Demikian pula, '1' dan 1 juga dianggap sama. Jika kita tidak ingin mengkonversi nilai sebelum perbandingan, kita harus menggunakan perbandingan ketat ===

Implicit type coercion

JavaScript dikenal sebagai bahasa dengan pengetikan yang longgar (loosely typed language). Pada dasarnya, saat kita menulis kode JavaScript, kita tidak perlu menentukan tipe data. Sebaliknya, saat kode kita ditafsirkan oleh mesin JavaScript, kode tersebut akan secara otomatis diubah menjadi tipe data yang "sesuai". Ini disebut **implicit type coercion** dan kita telah melihat contoh seperti ini sebelumnya ketika kita mencoba menggabungkan string dengan angka.

```
"julia" + 1
```

Returns: "julia1"

Dalam contoh ini, JavaScript mengambil string "julia" dan menambahkan angka 1 sehingga menghasilkan string "julia1". Dalam bahasa pemrograman lain, kode ini mungkin akan menampilkan error, tetapi dalam JavaScript angka 1 diubah menjadi string "1" dan kemudian digabungkan menjadi string "julia".

Perilaku seperti inilah yang membuat JavaScript unik dari bahasa pemrograman lain, tetapi dapat menyebabkan beberapa perilaku aneh saat melakukan operasi dan perbandingan pada tipe data campuran.

DEFINISI: Bahasa yang diketik dengan kuat (**Strongly typed language**) adalah bahasa pemrograman yang lebih cenderung menghasilkan kesalahan jika data tidak cocok dengan tipe yang diharapkan. Karena JavaScript adalah *loosely typed language*, kita tidak perlu menentukan tipe data; namun, ini dapat menyebabkan kesalahan yang sulit dicari karena **Implicit type coercion**.

Contoh dari **Strongly typed language code**

```
int count = 1; string name = "Julia"; double num = 1.2932;
float price = 2.99;
```

Kode yang sama jika diimplementasi pada JavaScript

```
var count = 1; var name = "Julia"; var num = 1.2932; var price
= 2.99;
```

Strict equality

Dalam JavaScript lebih baik menggunakan persamaan yang ketat (**Strict Equality**) untuk melihat apakah angka, string, atau boolean, dll. identik dalam tipe data

dan nilai tanpa melakukan konversi jenis terlebih dahulu. Untuk melakukan perbandingan yang ketat, cukup tambahkan tanda sama dengan = di akhir operator == dan !=.

```
"1" === 1
```

Returns: false

Contoh ini mengembalikan false karena string **"1"** bukan tipe data dan nilai yang sama dengan angka **1**.

```
0 === false
```

Returns: false

Contoh ini mengembalikan false karena angka **0** bukan tipe data dan nilai yang sama dengan boolean **false**. Sama seperti operator strict equality, ada juga operator ***strict non-equality*** yang ketat **!==** yang dapat digunakan sebagai pengganti **!=** jika kita tidak menginginkan konversi tipe sebelum perbandingan. Misalnya,

```
0 !== true
```

Returns: true

```
'1' !== 1
```

Returns: true

Conditionals

If...Else Statements

If...else statements memungkinkan Anda untuk mengeksekusi potongan kode tertentu berdasarkan kondisi, atau kumpulan kondisi, yang terpenuhi.

```
if (/ *this expression is true* /) {
  // run this code
} else {
  // run this code
}
```

Ini akan sangat membantu karena memungkinkan kita untuk memilih bagian kode mana yang ingin dijalankan berdasarkan hasil dari kondisinya. Misalnya,

```
var a = 1;
var b = 2;

if (a > b) {
  console.log("a is greater than b");
} else {
  console.log("a is less than or equal to b");
}
```

Prints: "a is less than or equal to b"

Beberapa hal penting yang perlu diperhatikan tentang pernyataan if...else. Nilai di dalam pernyataan **if** selalu diubah menjadi true atau false. Tergantung dari nilainya, kode di dalam pernyataan **if** akan dijalankan atau kode di **else** yang dijalankan. Kode di dalam pernyataan **if** dan **else** diapit oleh **kurung kurawal** {...} untuk memisahkan kondisi dan menunjukkan kode mana yang harus dijalankan.

Else If Statements

Dalam beberapa situasi, dua kondisional tidak cukup. Perhatikan situasi berikut ini.

Kita mencoba memutuskan apa yang akan dipakai besok. Jika turun salju, maka kita pasti ingin memakai mantel. Jika tidak turun salju dan tidak turun hujan, maka kita

sebaiknya memakai jaket. Dan jika tidak akan turun salju atau hujan, maka kita hanya akan memakai pakaian yang dimiliki.

Dalam JavaScript, kita dapat merepresentasikan pemeriksaan sekunder ini dengan menggunakan pernyataan if ekstra yang disebut **else if statement**.

```
var weather = "sunny";

if (weather === "snow") {
  console.log("Bring a coat.");
} else if (weather === "rain") {
  console.log("Bring a rain jacket.");
} else {
  console.log("Wear what you have on.");
}
```

Prints: Wear what you have on.

Dengan menambahkan pernyataan **else if**, kita menambahkan pernyataan kondisional. Jika tidak akan turun salju, maka kode akan masuk ke pernyataan **else if** untuk melihat apakah akan turun hujan. Jika tidak akan hujan, maka kode akan masuk ke pernyataan **else**.

Pernyataan **else** pada dasarnya bertindak sebagai kondisi "default" jika semua pernyataan **if** lainnya salah.

Logical Operators

```
var colt = "not busy";
var weather = "nice";

if (colt === "not busy" && weather === "nice") {
  console.log("go to the park");
}
```

Prints: "go to the park"

Perhatikan **&&** pada kode di atas. Simbol **&&** adalah operator logika AND, dan digunakan untuk menggabungkan dua pernyataan logika menjadi satu pernyataan logika yang lebih besar. Jika kedua pernyataan yang lebih kecil benar, maka seluruh pernyataan dievaluasi menjadi benar. Jika salah satu dari pernyataan yang lebih kecil salah, maka seluruh ekspresi logika salah.

Cara lain untuk membacanya adalah ketika operator **&&** ditempatkan di antara dua pernyataan, kode tersebut secara harfiah berbunyi, "if Colt is not busy *AND* the weather is nice, then go to the park".

Logical expressions

Logical expression mirip dengan ekspresi matematika, hanya saja logical expression mengevaluasi true atau false.

```
11 != 12
```

Returns: true

Kita telah melihat logical expression saat menulis perbandingan. Perbandingan hanyalah ekspresi logika sederhana.

Mirip dengan ekspresi matematika yang menggunakan **+**, **-**, *****, **/** dan **%**, ada operator logika **&&**, || dan **!** yang dapat kita gunakan untuk membuat ekspresi logika yang lebih kompleks.

Logical operators

Operator logika bisa digunakan bersamaan dengan nilai boolean (true dan false) untuk membuat ekspresi logika yang kompleks. Dengan menggabungkan dua nilai boolean dengan operator logika, kita membuat ekspresi logika yang mengembalikan nilai boolean lainnya. Berikut adalah tabel yang menjelaskan berbagai operator logika:

Operator	Meaning	Example	How it works
&&	Logical AND	value1 && value2	Mengembalikan **true** jika **value1** dan **value2** bernilai **true**.
||	Logical OR	value1 || value2	Mengembalikan **true** jika salah satu

			dari **value1** atau **value2** (atau keduanya!) bernilai **true**.
!	Logical NOT	!value1	Mengembalikan kebalikan dari **value1**. Jika **value1** adalah **true**, maka **!value1** adalah **false**.

Logical AND and OR

&& (AND)

A	B	A && B
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

|| (OR)

A	B	A || B
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

Tabel kebenaran digunakan untuk mewakili hasil dari semua kemungkinan kombinasi input dalam ekspresi logika. A mewakili nilai boolean di sisi kiri ekspresi dan B mewakili nilai boolean di sisi kanan ekspresi. Tabel kebenaran dapat membantu untuk memvisualisasikan hasil yang berbeda dari ekspresi logika.

Truthy and Falsy

Setiap nilai dalam JavaScript memiliki nilai boolean bawaan. Saat nilai tersebut dievaluasi dalam konteks ekspresi boolean, nilai tersebut akan diubah menjadi nilai boolean bawaan tersebut.

Falsy values

Nilai menjadi salah jika dikonversi menjadi **false** saat dievaluasi dalam konteks boolean. Misalnya, String kosong "" adalah salah karena, "" ketika dievaluasi menjadi **false**. Kita sudah mengetahui pernyataan if...else, jadi mari kita gunakan untuk menguji kebenaran "".

```
if ("") {
    console.log("the value is truthy");
} else {
    console.log("the value is falsy");
}
```

Returns: "the value is falsy"

Berikut adalah list semua nilai **falsy :**

- Nilai **false** pada boolean
- **null**
- **Undefined**
- Angka **0**
- String kosong **“”**
- **NaN**

Truthy values

Suatu nilai menjadi benar jika diubah menjadi **true** ketika dievaluasi dalam konteks boolean. Misalnya, angka **1** adalah benar karena, **1** bernilai **true**. Mari gunakan pernyataan if...else lagi untuk menguji ini:

```
if (1) {
    console.log("the value is truthy");
} else {
    console.log("the value is falsy");
}
```

Returns: "the value is truthy"

Berikut contoh nilai yang benar :

```
true
42
"pizza"
"0"
"null"
"undefined"
{}
[]
```

Ternary Operator

Kadang-kadang, kita mungkin menemukan jenis kondisional berikut.

```
var isGoing = true;
var color;

if (isGoing) {
  color = "green";
} else {
  color = "red";
}

console.log(color);
```

Prints: "green"

Dalam contoh ini, warna variabel **color** berisi "**hijau**" atau "**merah**" berdasarkan nilai dari variabel **isGoing**. Kode ini berfungsi, tetapi ini cara yang agak panjang untuk menentukan nilai ke variabel. Untungnya, dalam JavaScript ada cara lain.

Ternary operator memberikan kita shortcut untuk menulis pernyataan if...else yang panjang.

```
conditional ? (if condition is true) : (if condition is false)
```

Untuk menggunakan ternary operator, pertama berikan pernyataan kondisional di sisi kiri **?**. Lalu, antara **?** dan **:** tulis kode yang akan dijalankan jika kondisinya **true**.

Di sisi sebelah kanan : tulis kode yang akan dijalankan jika kondisinya **false**. Misalnya, kita dapat menulis ulang kode contoh di atas sebagai berikut:

```
var isGoing = true;
var color = isGoing ? "green" : "red";

console.log(color);
```

Prints: "green"

Jika kita melihat kodenya, kondisi **isGoing** ditempatkan di sisi kiri **?**. Kemudian, ekspresi pertama setelah **?** adalah kode yang akan dijalankan jika kondisinya **true** dan ekspresi kedua setelah **:** adalah yang akan dijalankan jika kondisinya **false**.

Switch Statement

Switch Statement adalah cara lain untuk menyusun beberapa pernyataan **else if** yang berdasarkan pada nilai yang sama **tanpa menggunakan pernyataan kondisional**. Sebagai gantinya, kita cukup mengganti bagian kode mana yang dieksekusi berdasarkan nilai.

```
switch (option) {
  case 1:
    console.log("You selected option 1.");
  case 2:
    console.log("You selected option 2.");
  case 3:
    console.log("You selected option 3.");
  case 4:
    console.log("You selected option 4.");
  case 5:
    console.log("You selected option 5.");
  case 6:
    console.log("You selected option 6.");
}
```

Setiap pernyataan **else if** (**option === [value]**) telah diganti dengan **case** (**case [value]:**) dan pernyataan tersebut telah dibungkus di dalam pernyataan **switch**.

Saat pernyataan switch pertama kali dievaluasi, ia mencari **case** pertama yang nilainya sama dengan hasil pernyataan dari switch. Kemudian, ia mengeksekusi kode di dalam **case** tersebut.

Jadi, jika kita mengubah **option** menjadi 3…

```
var option = 3;

switch (option) {
  ...
}
```

Prints:
You selected option 3.
You selected option 4.
You selected option 5.
You selected option 6.

..kemudian pernyataan switch mencetak option 3, 4, 5, dan 6. Namun itu jadi tidak persis seperti kode if...else. Jadi apa yang hilang?

Break statement

Break statement dapat digunakan untuk mengakhiri pernyataan switch dan mengeluarkan eksekusi kode. Dengan menambahkan **break** ke setiap **case**, kita telah memperbaiki masalah pernyataan switch yang masuk ke dalam case lainnya.

```
switch (option) {
  case 1:
    console.log("You selected option 1.");
    break;
  case 2:
    console.log("You selected option 2.");
    break;
  case 3:
    console.log("You selected option 3.");
    break;
  case 4:
    console.log("You selected option 4.");
```

```
    break;
  case 5:
    console.log("You selected option 5.");
    break;
  case 6:
    console.log("You selected option 6.");
    break; // technically, not needed
}
```

Prints: You selected option 3.

Falling-through

Dalam beberapa kondisi, kita mungkin ingin memanfaatkan perilaku "gagal" dari pernyataan switch.

Misalnya, ketika kode Anda mengikuti struktur tipe hierarki seperti ini.

```
var tier = "nsfw deck";
var output = "You'll receive "

switch (tier) {
  case "deck of legends":
    output += "a custom card, ";
  case "collector's deck":
     output += "a signed version of the Exploding Kittens deck,
";
  case "nsfw deck":
      output += "one copy of the NSFW (Not Safe for Work)
Exploding Kittens card game and ";
  default:
    output += "one copy of the Exploding Kittens card game.";
}

console.log(output);
```

Prints: You'll receive one copy of the NSFW (Not Safe for Work) Exploding Kittens card game and one copy of the Exploding Kittens card game.

Dalam contoh ini, berdasarkan campaign Exploding Kittens Kickstarter yang sukses (permainan kartu lucu yang dibuat oleh Elan Lee), setiap tier berturut-turut membangun tier berikutnya dengan menambahkan lebih banyak output. Tanpa

pernyataan break dalam kode, setelah pernyataan switch melompat ke "nsfw deck", itu gagal terus menerus hingga mencapai akhir pernyataan switch.

Perhatikan juga case default.

```
var tier = "none";
var output = "You'll receive ";

switch (tier) {
  ...
  default:
    output += "one copy of the Exploding Kittens card game.";
}

console.log(output);
```

Prints: You'll receive one copy of the Exploding Kittens card game.

Kita bisa menambahkan case default ke pernyataan switch. Ketika tidak ada nilai yang cocok dengan nilai ekspresi switch, maka case default akan dijalankan.

Loops

While Loops

Ada banyak jenis looping, tetapi semuanya pada dasarnya melakukan hal yang sama: mengeksekusi kode beberapa kali.

Tiga hal utama yang harus dimiliki setiap looping adalah:

1. **Kapan untuk memulai**: Kode yang mengatur perulangan — misalnya menentukan nilai awal variabel.
2. **Kapan harus berhenti**: Kondisi logika untuk menguji apakah perulangan harus dilanjutkan.
3. **Cara mendapatkan item berikutnya**: Langkah penambahan atau pengurangan — misalnya, x = x * 3 atau x = x - 1

Berikut adalah contoh while loop dasar yang menyertakan ketiga bagian tersebut.

```
var start = 0; // when to start
while (start < 10) { // when to stop
  console.log(start);
  start = start + 2; // how to get to the next item
}
```

Prints:
0
2
4
6
8

Jika looping kehilangan salah satu dari tiga hal tersebut, maka kita mungkin menemukan masalah. Misalnya, kondisi stop yang dihilangkan bisa mengakibatkan perulangan yang tidak akan berakhir!

Jangan jalankan kode berikut!

```
js while (true) { console.log("true is never false, so I will never stop!"); }
```

Jika kita mencoba menjalankan kode itu di console,tab browser kita akan crash.

Berikut adalah contoh looping dimana kondisi cara menuju ke item berikutnya, dihilangkan; sehingga variabel x tidak akan pernah bertambah. x akan tetap 0 sepanjang program berjalan, sehingga perulangan tidak akan pernah berakhir.

Jangan jalankan kode berikut ini!

```
var x = 0;
while (x < 1) {
  console.log('Oops! x is never incremented from 0, so it will ALWAYS be less than 1');
}
```

Kode ini juga akan membuat tab browser crash. Sehingga tidak disarankan untuk menjalankannya.

For Loops

Looping menggunakan **for** secara eksplisit memaksa kita untuk menentukan titik awal, titik berhenti, dan setiap langkah perulangan. Kita akan mendapatkan **Uncaught SyntaxError: Unexpected token** jika kita melewatkan salah satu dari tiga bagian yang diperlukan.

```
for ( start; stop; step ) {
  // do this thing
}
```

Berikut adalah contoh perulangan **for** yang mencetak nilai dari 0 sampai 5. Perhatikan titik koma yang memisahkan berbagai pernyataan:

```
var i = 0; i < 6; i = i + 1
```

```
for (var i = 0; i < 6; i = i + 1) {
  console.log("Printing out i = " + i);
}
```

Prints:
Printing out i = 0
Printing out i = 1
Printing out i = 2
Printing out i = 3
Printing out i = 4
Printing out i = 5

Nested Loops

Kita dapat membuat looping di dalam looping. Cobalah kode berikut dan lihatlah hasilnya.

```
for (var x = 0; x < 5; x = x + 1) {
  for (var y = 0; y < 3; y = y + 1) {
    console.log(x + "," + y);
  }
}
```

Prints:
0, 0
0, 1
0, 2
1, 0
1, 1
1, 2
2, 0
2, 1
2, 2
3, 0
3, 1
3, 2
4, 0
4, 1
4, 2

Perhatikan urutan dari output yang dihasilkan.
Loop terluar dimulai dengan x = 0, kemudian loop bagian dalam menyelesaikan perulangan dengan semua kondisi nilai y:

```
x = 0 and y = 0, 1, 2 // corresponds to (0, 0), (0, 1), and (0, 2)
```

Setelah loop bagian dalam selesai melakukan perulangan atas y, maka looping terluar berlanjut ke nilai berikutnya yaitu x = 1, dan seluruh proses dimulai lagi.

```
x = 0 and y = 0, 1, 2 // (0, 0) (0, 1) and (0, 2)
x = 1 and y = 0, 1, 2 // (1, 0) (1, 1) and (1, 2)
x = 2 and y = 0, 1, 2 // (2, 0) (2, 1) and (2, 2)
etc.
```

Increment and Decrement

Kita sering kali perlu menambahkan atau mengurangi nilai variabel untuk melakukan perulangan. Dengan JavaScript dan bahasa pemrograman lainnya. Sebenarnya ada operator increment. Operator tersebut memberikan jalan pintas untuk melakukan hal yang mirip. Dan itu ditulis dalam dua cara berbeda, yaitu x++ dan ++x. x++ sebenarnya sama dengan x = x + 1. Namun, yang akan dilakukannya adalah mengembalikan nilai x sebelum menambahkannya. Berikut conto operator increment dan decrement yang lain :

```
x++ or ++x // sama dengan x = x + 1
x-- or --x // sama dengan x = x - 1
x += 3 // sama dengan x = x + 3
x -= 6 // sama dengan x = x - 6
x *= 2 // sama dengan x = x*  2
x /= 5 // sama dengan x = x / 5
```

Functions

Function Example

Berikut adalah salah satu contoh function untuk memutar balikan urutan huruf dalam sebuah kata/kalimat :

```
function reverseString(reverseMe) {
  var reversed = "";
  for (var i = reverseMe.length - 1; i >= 0; i--) {
    reversed += reverseMe[i];
  }
  return reversed;
}
```

Declaring Function

How to declare a function

Function memungkinkan kita untuk mengemas baris kode yang kita gunakan (dan yang sering kita gunakan lagi). Terkadang function mengambil parameter. Seperti function reheatPizza() berikut ini, memiliki satu parameter: numSlices.

```
function reheatPizza(numSlices) {
  // code that figures out reheat settings!
}
```

Fungsi reverseString() yang kita lihat sebelumnya juga memiliki satu parameter: string yang akan dibalik.

```
function reverseString(reverseMe) {
  // code to reverse a string!
}
```

Dalam kedua kasus tersebut, parameter dicantumkan sebagai variabel dan berada di dalam kurung setelah nama fungsi. Jika ada beberapa parameter, kita cukup memisahkannya dengan koma.

```
function doubleGreeting(name, otherName) {
  // code to greet two people!
}
```

Namun, kita juga bisa memiliki fungsi yang tidak memiliki parameter apa pun. Sebaliknya, fungsi tersebut hanya mengemas beberapa kode dan melakukan beberapa tugas. Dalam hal ini, kita cukup mengosongkan tanda kurung. Lihatlah contoh ini, fungsi sederhana yang berfungsi untuk mencetak "Halo!".

```
// accepts no parameters! parentheses are empty
function sayHello() {
  var message = "Hello!"
  console.log(message);
}
```

Jika kita mencoba salah satu fungsi di atas di dalam console JavaScript, kita mungkin tidak menyadari ada banyak hal yang terjadi. Faktanya, kita bisa saja melihat **undefined** dikembalikan kepada kita. **undefined** adalah nilai pengembalian default di konsol ketika tidak ada yang dikembalikan secara eksplisit dengan menggunakan keyword **return**.

Return statements

Dalam function **sayHello()** sebelumnya, sebuah nilai dicetak ke console dengan **console.log**, tetapi tidak dikembalikan dengan pernyataan **return**. Kita dapat menulis pernyataan **return** diikuti dengan ekspresi atau nilai yang ingin kita kembalikan.

```
// declares the sayHello function
function sayHello() {
  var message = "Hello!"
  return message; // returns value instead of printing it
}
```

How to *run* a function

Saat ini, agar fungsi kita bisa melakukan sesuatu, kita harus memanggil function menggunakan nama function, diikuti dengan tanda kurung dengan argumen yang diteruskan ke dalamnya. Function bisa dianalogikan seperti mesin. Kita bisa membuat mesin, tetapi mesin tersebut tidak akan melakukan apapun selama kita tidak menyalakannya. Inilah cara kita memanggil function sayHello(), lalu menggunakan nilai dari return untuk mencetak ke console:

```
// declares the sayHello function
function sayHello() {
  var message = "Hello!"
  return message; // returns value instead of printing it
}

// function returns "Hello!" and console.log prints the return value
console.log(sayHello());
```

Prints: "Hello!"

Parameters vs. Arguments

Pada awalnya, mungkin agak sulit untuk mengetahui apakah kita membutuhkan parameter atau argumen. Perbedaan utamanya adalah di mana mereka muncul di dalam kode. **Parameter** akan selalu menjadi nama variabel dan muncul saat deklarasi fungsi. Di sisi lain, **argument** akan selalu berupa nilai (yaitu salah satu tipe data JavaScript - angka, string, boolean, dll.) dan akan selalu muncul di dalam kode saat fungsi dipanggil.

Return Values

Returning vs. Logging

Penting untuk dipahami bahwa **return** dan **print** bukanlah hal yang sama. Print ke console JavaScript hanya menampilkan nilai (yang dapat kita lihat untuk tujuan debug), tetapi nilai yang ditampilkannya tidak bisa digunakan untuk hal lain. Oleh karena itu, kita

harus ingat bahwa kita hanya menggunakan console.log untuk menguji kode kita di console JavaScript.

Cobalah fungsi berikut ke dalam console JavaScript untuk melihat perbedaan antara return dan logging(print):

```
function isThisWorking(input) {
  console.log("Printing: isThisWorking was called and " + input
+ " was passed in as an argument.");
  return "Returning: I am returning this string!";
}

isThisWorking(3);
```

Prints: "Printing: isThisWorking was called and 3 was passed in as an argument"
Returns: "Returning: I am returning this string!"

Jika Anda tidak menentukan nilai return, function akan mengembalikan **undefined**

```
function isThisWorking(input) {
  console.log("Printing: isThisWorking was called and " + input
+ " was passed in as an argument.");
}

isThisWorking(3);
```

Prints: "Printing: isThisWorking was called and 3 was passed in as an argument"
Returns: undefined

Using Return Values

Nilai return dari function bisa disimpan di dalam variabel atau digunakan kembali di seluruh program kita sebagai function argument. Di sini, kita memiliki function yang menjumlahkan dua angka, dan function lain yang membagi angka dengan 2. Kita dapat mencari rata-rata dari 5 dan 7 dengan menggunakan fungsi add() untuk menjumlahkan kedua angka tersebut, lalu dengan mengirimkan jumlah add(5, 7) ke dalam fungsi divideByTwo() sebagai argumen.

Dan terakhir, kita bahkan bisa menyimpan hasil akhir pada variabel yang disebut average dan menggunakan variabel tersebut untuk melakukan lebih banyak kalkulasi di tempat lain.

```
// returns the sum of two numbers
function add(x, y) {
  return x + y;
}


// returns the value of a number divided by 2
function divideByTwo(num) {
  return num / 2;
}


var sum = add(5, 7); // call the "add" function and store the returned value in the "sum" variable
var average = divideByTwo(sum); // call the "divideByTwo" function and store the returned value in the "average" variable
```

Scope

Mari luangkan waktu sejenak untuk membahas **scope**. Terlepas dari kesalahan sintaks, scope akan menjadi inti dari bug kode yang kita temui pada hampir semua bahasa pemrograman. Ketika developer berbicara tentang scope, mereka berbicara tentang bagian dari program seperti nama variabel atau function apakah dapat terlihat atau dapat diakses.

Scope Example

JavaScript memiliki dua jenis scope, **global scope** dan **function scope**. Jika kita mendeklarasikan identifier di luar dari semua fungsi yang ada, maka ini dianggap sebagai **global scope**. Yang berarti variabel tersebut dapat diakses di mana saja di dalam program. Semua function dalam program kita dapat mengakses variabel yang didefinisikan dalam lingkup global.

Di sisi lain, JavaScript juga memiliki function scope. Jika identifier didefinisikan di dalam function, berarti variable tersebut hanya bisa di akses di dalam function tersebut.

Shadowing

Scope bisa menjadi hal yang rumit, terutama saat kita menggunakan antara global scope dan function scope. Salah satu scope dapat menghasilkan yang disebut scope overriding, atau shadowing. Untuk menunjukkan cara kerjanya dan cara menghindarinya, mari kita lihat contoh kode berikut.

```
1   var bookTitle = "Le Petit Prince";
2   console.log(bookTitle);
3
4   function displayBookEnglish() {
5     bookTitle = "The Little Prince";
6     console.log(bookTitle);
7   }
8
9   displayBookEnglish();
10  console.log(bookTitle);
```

Di sini kita memiliki variabel bookTitle yang diberi nilai dalam dua scope yang berbeda. bookTitle dalam global scope diberi nilai “Le Petit Prince”, dan judul buku dalam function scope diberi nilai “The Little Prince”. Jika kita print nilai bookTitle di setiap bagian dari program ini, kita akan melihat di pernyataan console.log terakhir bahwa “The Little Prince” dicetak. Namun, pada titik ini, kita tidak lagi berada dalam function scope dari fungsi displayBookEnglish.

Lalu kenapa judul buku “Le Petit Prince” malah tidak dicetak? Nah, ini adalah kasus klasik dari scope overriding atau shadowing. Saat kita mencapai baris kelima dari kode

ini, variabel bookTitle dari scope global sebenarnya diubah dengan nilai “The Little Prince”.

Untuk mencegah hal ini terjadi, kita cukup mendeklarasikan variabel baru di dalam fungsi displayBookEnglish. Jadi alih-alih mengubah ulang nilai bookTitle, kita akan membuat variabel yang berbeda seperti berikut ini.

```
1   var bookTitle = "Le Petit Prince";
2   console.log(bookTitle);
3
4   function displayBookEnglish() {
5     var bookTitle = "The Little Prince";
6     console.log(bookTitle);
7   }
8
9   displayBookEnglish();
10  console.log(bookTitle);
```

Jadi sekarang kita mempunyai variabel di dalam displayBookEnglish yang berarti variable ini adalah function scope. Dan variable bookTitle yang dideklarasikan pada global scope tidak akan berubah. Dengan begitu, ketika kita mencapai baris kesepuluh dalam kode kita dan kita mencetaknya untuk ketiga kalinya, itu akan mencetak apa yang kita harapkan, yaitu Le Petit Prince.

Global Variables

Jadi kita mungkin bertanya-tanya:

> "Mengapa kita tidak selalu menggunakan variabel global? Sehingga, kita tidak perlu menggunakan function arguments karena SEMUA function kita akan memiliki akses ke SEMUANYA!"

Yah... Variabel global mungkin tampak seperti ide yang bagus pada awalnya, terutama ketika kita sedang menulis script dan program yang kecil. Tetapi ada banyak alasan mengapa kita tidak boleh menggunakannya. Misalnya, variabel global bisa conflict dengan variabel global lainnya dengan nama yang sama. Setelah program kita menjadi semakin besar, maka akan semakin sulit untuk melacak dan mencegah hal ini terjadi.

Ada juga alasan lain yang akan kita pelajari lebih lanjut. Namun untuk saat ini, usahakan untuk meminimalkan penggunaan variabel global sebanyak mungkin.

Hoisting

Di sebagian besar bahasa pemrograman, kita harus mendeklarasikan suatu fungsi sebelum kita dapat memanggilnya.

```
1   findAverage(5, 9);
2
3   function findAverage(x, y) {
4     var answer = (x + y) / 2;
5     return answer;
6   }
7
8
9
10
```

Pada dasarnya adalah bahwa kode dibaca dari atas ke bawah, jadi tidak mungkin kita dapat memanggil fungsi findAverage ini bahkan belum dideklarasikan. Secara harfiah, bagaimana kita bisa memanggil findAverage saat fungsi dideklarasikan lebih rendah daripada saat pemanggilan terjadi di kode. Jadi seharusnya ini tidak akan berhasil, bukan? Yah, sebenarnya memang begitu dan ini karena satu fitur yang sangat gabus dalam JavaScript yang disebut hoisting. Dan itu ada hubungannya dengan bagaimana kode JavaScript ditafsirkan. Pada dasarnya sebelum kode apa pun dijalankan, semua deklarasi fungsi diangkat ke atas scope saat ini.

Sekarang untuk memperjelas, kode yang kita tulis sebenarnya tidak akan berubah. Kode masih akan terlihat persis seperti yang Anda tulis. Hanya saja ketika kode sedang ditafsirkan, fungsi deklarasi benar-benar diangkat ke atas seperti berikut.

```
findAverage(5, 9);

function findAverage(x, y) {
  var answer = (x + y) / 2;
  return answer;
}
> 7
```

Hoisting juga bisa terjadi saat deklarasi variabel, mari kita lihat contoh sederhana ini.

```
function sayGreeting() {
  console.log(greeting);
}

sayGreeting();
> Uncaught ReferenceError: greeting is not defined
```

Jika kita memanggil fungsi sayGreeting seperti itu, kita akan mendapatkan error Uncaught ReferenceError dan itu karena variabel greeting belum didefinisikan atau dideklarasikan di mana pun. Bagaimana kita bisa mencetak greeting jika tidak ada? Untuk memperbaiki error ini, kita dapat mendeklarasikan greeting hampir di mana saja di dalam function karena pada akhirnya, kita tahu bahwa greeting akan diangkat (hoisting) ke posisi atas dari function scope. Sebagai contoh, jika kita mendeklarasikan greeting pada baris ketiga, kita tahu bahwa pada akhirnya ketika kode ditafsirkan, greeting akan diangkat ke bagian atas dari function.

```
function sayGreeting() {
  console.log(greeting);
  var greeting;
}

sayGreeting();
```

```
> undefined
```

Tentu saja, ketika kode ini dijalankan masih akan terlihat sama persis seperti yang kita tulis, tetapi sekarang ini dicetak sebagai undefined. Jadi itu masuk akal karena kita belum benar-benar memberi greeting. Jadi mari kita coba beri nilai dan lihat apa yang terjadi. Jadi daripada hanya greeting saja, mari kita tetapkan greeting dan string halo.

```
function sayGreeting() {
  console.log(greeting);
  var greeting = "hello";
}

sayGreeting();
```

```
> undefined
```

Oke, jadi kita membuatnya undefined lagi. Ini sebenarnya bug karena hoisting. Variabel greeting diangkat ke bagian atas dari function scope, tetapi penentuan isi variable tetap berada di tempatnya.

```
function sayGreeting() {
  var greeting;
  console.log(greeting);
  greeting = "hello";
}

sayGreeting();
```

Karena deklarasi variabel dihoisting di atas, tetapi nilainya masih belum ditentukan. Dan kemudian ketika kita mencetaknya, itu kembali memberikan hasil undefined.

Jadi inti dari cerita ini adalah untuk menghindari bug seperti ini, developer JavaScript biasanya mendeklarasikan fungsi di bagian atas script mereka, dan variabel di bagian atas functionnya. Dengan cara itu tampilan kode, dan cara kode berperilaku, konsisten satu sama lain.

```
1   function sayGreeting() {
2     var greeting = "hello";
3     console.log(greeting);
4   }
5
6   sayGreeting();
> "hello"
```

Function Expressions

Ingat bagaimana kita dapat menyimpan apapun yang kita inginkan dalam sebuah variabel? Nah, di JavaScript, kita juga bisa menyimpan fungsi dalam variabel. Ketika suatu fungsi disimpan di dalam variabel, itu disebut **function expression**.

Perhatikan bagaimana keyword **function** tidak lagi memiliki nama.

```
var catSays = function(max) {
  // code here
};
```

Ini adalah **anonymous function**, fungsi tanpa nama, dan kita telah menyimpannya dalam variabel bernama **catSays**.

Dan, jika kita mencoba mengakses nilai dari variabel **catSays**, kita bahkan akan melihat function dikembalikan kepada kita.

```
catSays;
```

Returns:

```
function(max) {
  var catMessage = ""
  for (var i = 0; i < max; i++) {
    catMessage += "meow ";
  }
  return catMessage;
}
```

Function expressions and hoisting

Memutuskan kapan menggunakan function expression dan kapan menggunakan function declaration dapat bergantung pada beberapa hal. Tapi, satu hal yang ingin kita berhati-hati adalah hoisting.

Semua function declaration diangkat (hoisting) dan dimuat sebelum script benar-benar dijalankan. Function expression tidak dihoisting, karena melibatkan penugasan variabel, dan hanya deklarasi variabel yang dihoisting. Function expression tidak akan dimuat hingga interpreter mencapainya.

Pattern With Function Expressions

Functions as parameters

Mampu menyimpan function dalam variabel membuatnya sangat mudah untuk meneruskan fungsi ke fungsi lain. Fungsi yang diteruskan ke fungsi lain disebut **callback**. Katakanlah kita memiliki fungsi helloCat(), dan kita ingin mengembalikan "Hello" diikuti dengan string "meows". Nah, daripada mengulangi semua kerja keras kita, kita dapat membuat helloCat() menerima fungsi callback, dan mengirimkan catSays.

```
// function expression catSays
var catSays = function(max) {
  var catMessage = "";
  for (var i = 0; i < max; i++) {
    catMessage += "meow ";
  }
  return catMessage;
};

// function declaration helloCat accepting a callback
function helloCat(callbackFunc) {
  return "Hello " + callbackFunc(3);
```

```
}

// pass in catSays as a callback function
helloCat(catSays);
```

Inline function expressions

Function expression adalah ketika suatu fungsi ditugaskan ke variabel. Dan, dalam JavaScript, ini juga bisa terjadi saat kita meneruskan fungsi sebaris sebagai argumen ke fungsi lain. Ambil contoh favoriteMovie misalnya:

```
/ Function expression that assigns the function displayFavorite
// to the variable favoriteMovie
var favoriteMovie = function displayFavorite(movieName) {
  console.log("My favorite movie is " + movieName);
};

// Function declaration that has two parameters: a function for
displaying
// a message, along with a name of a movie
function movies(messageFunction, name) {
  messageFunction(name);
}

// Call the movies function, pass in the favoriteMovie function
and name of movie
movies(favoriteMovie, "Finding Nemo");
```

Returns: My favorite movie is Finding Nemo

Tapi kita bisa melewati tugas pertama dari fungsi tersebut, dengan meneruskan fungsi tersebut ke inline fungsi movies() .

```
// Function declaration that takes in two arguments: a function
for displaying
// a message, along with a name of a movie
function movies(messageFunction, name) {
  messageFunction(name);
}

// Call the movies function, pass in the function and name of
movie
```

```
movies(function displayFavorite(movieName) {
  console.log("My favorite movie is " + movieName);
}, "Finding Nemo");
```

Returns: My favorite movie is Finding Nemo

Jenis sintaks ini, menuliskan function expression yang meneruskan fungsi ke dalam fungsi lain dalam sebaris. Hal ini sangat umum dalam JavaScript. Hal ini bisa sedikit rumit pada awalnya, tetapi bersabarlah, teruslah berlatih, dan kita akan mulai menguasainya!

Why use anonymous inline function expressions?

Menggunakan anonymous inline function expressions tampaknya seperti hal yang sangat tidak berguna pada awalnya. Mengapa menentukan fungsi yang hanya dapat digunakan sekali dan kita bahkan tidak dapat memanggilnya dengan nama?

Anonymous inline function expressions sering digunakan dengan function callback yang mungkin tidak akan digunakan kembali di tempat lain. Ya, kita dapat menyimpan fungsi dalam variabel, memberinya nama, dan meneruskannya seperti yang kita lihat pada contoh di atas. Namun, jika kita mengetahui bahwa fungsi tersebut tidak akan digunakan kembali, kita dapat menghemat banyak baris kode hanya dengan mendefinisikannya secara inline.

Arrays

Creating an Array

Array berfungsi untuk menyimpan beberapa nilai ke dalam data struktur yang terorganisir. Kita dapat mendefinisikan array baru dengan mencantumkan nilai yang dipisahkan dengan koma di dalam tanda kurung siku **[ ]**.

```
// creates a `donuts` array with three strings
var donuts = ["glazed", "powdered", "jelly"];
```

Tapi string bukan satu-satunya tipe data yang bisa kita simpan di dalam array. Kita juga dapat menyimpan angka, boolean… dan apa saja!

```
// creates a `mixedData` array with mixed data types
var mixedData = ["abcd", 1, true, undefined, null, "all the things"];
```

Bahkan kita bisa menyimpan array di dalam array (Nested Array)

```
// creates a `arraysInArrays` array with three arrays
var arraysInArrays = [[1, 2, 3], ["Julia", "James"], [true, false, true, false]];
```

Nested Array bisa sangat sulit dibaca, jadi biasanya daripada menulisnya dalam satu baris, kita bisa menggunakan baris baru setelah setiap koma:

```
var arraysInArrays = [
  [1, 2, 3],
  ["Julia", "James"],
  [true, false, true, false]
];
```

Accessing Array Elements

Ingatlah bahwa elemen di dalam array, diindeks mulai dari posisi 0. Untuk mengakses elemen dalam array, gunakan nama array diikuti dengan tanda kurung siku yang berisi indeks dari nilai yang ingin diakses.

```
var donuts = ["glazed", "powdered", "sprinkled"];
console.log(donuts[0]); // "glazed" is the first element in the `donuts` array
```

Prints: "glazed"

Hal yang harus diperhatikan adalah jika kita mencoba mengakses elemen pada indeks yang tidak ada, maka nilai undefined akan dikembalikan.

```
console.log(donuts[3]); // the fourth element in `donuts` array does not exist!
```

Prints: undefined

Jika kita ingin mengubah nilai elemen dalam array, kita dapat melakukannya dengan menyetelnya dengan nilai baru.

```
donuts[1] = "glazed cruller"; // changes the second element in the `donuts` array to "glazed cruller"
console.log(donuts[1]);
```

Prints: "glazed cruller"

Length

Kita dapat mencari seberapa panjang array dengan menggunakan properti length.

```
var donuts = ["glazed", "powdered", "sprinkled"];
console.log(donuts.length);
```

Prints: 3

Untuk mengakses properti **length**, ketikkan nama array, diikuti dengan tanda titik **.** beserta keyword **length** (kita juga akan menggunakan titik untuk mengakses properti dan metode lainnya). Properti **length** kemudian akan mengembalikan **jumlah elemen** dalam array.

TIP: String juga memiliki properti **length**! Kita dapat menggunakannya untuk mendapatkan panjang string apa pun. Misalnya, "**supercalifragilisticexpialidocious**".**length** akan mengembalikan **34**.

Push

Jadi kita dapat mencari seberapa panjang array, tetapi bagaimana jika kita ingin mengubah array?

Untungnya, array memiliki beberapa metode bawaan untuk menambah dan menghapus elemen dari array. Dua metode paling umum untuk memodifikasi array adalah push() dan pop().

Kita dapat menggunakan metode push() untuk menambahkan elemen ke akhir array.

Misalnya, bayangkan donat berikut ini.

Kita dapat merepresentasikan donat-donat tersebut menggunakan array

```
var donuts = ["glazed", "chocolate frosted", "Boston creme",
"glazed cruller", "cinnamon sugar", "sprinkled"];
```

Kemudian, kita dapat mendorong (**push**) donat ke ujung array menggunakan metode push().

```
donuts.push("powdered"); // pushes "powdered" onto the end of the
`donuts` array
```

Returns: 7

donuts array: ["glazed", "chocolate frosted", "Boston creme", "glazed cruller", "cinnamon sugar", "sprinkled", "powdered"]

Perhatikan, dengan metode push() kita harus menentukan nilai elemen yang ingin kita tambahkan ke akhir array. Juga, metode push() mengembalikan panjang array setelah elemen ditambahkan.

```
var donuts = ["glazed", "chocolate frosted", "Boston creme",
"glazed cruller", "cinnamon sugar", "sprinkled"];
donuts.push("powdered"); // the `push()` method returns 7
because the `donuts` array now has 7 elements
```

Returns: 7

Pop

Sebagai alternatif, kita dapat menggunakan metode pop() untuk menghapus elemen dari akhir array.

```
var donuts = ["glazed", "chocolate frosted", "Boston creme",
"glazed cruller", "cinnamon sugar", "sprinkled", "powdered"];

donuts.pop(); // pops "powdered" off the end of the `donuts`
array
donuts.pop(); // pops "sprinkled" off the end of the `donuts`
array
donuts.pop(); // pops "cinnamon sugar" off the end of the
`donuts` array
```

Returns: "cinnamon sugar"

donuts array: ["glazed", "chocolate frosted", "Boston creme", "glazed cruller"]

Dengan metode pop(), kita tidak perlu memberikan nilai; sebagai gantinya, pop() akan selalu menghapus elemen terakhir dari akhir array. Selain itu, pop() mengembalikan elemen yang telah dihapus jika kita perlu menggunakannya.

```
var donuts = ["glazed", "chocolate frosted", "Boston creme",
"glazed cruller", "cinnamon sugar", "sprinkled", "powdered"];
donuts.pop(); // the `pop()` method returns "powdered" because
"powdered" was the last element on the end of `donuts` array
```

Returns: "powdered"

Splice

splice() adalah metode praktis lain yang memungkinkan kita untuk menambah dan menghapus elemen dari mana saja di dalam array.

Sementara push() dan pop() membatasi kita untuk menambah dan menghapus elemen dari akhir array, splice() memungkinkan kita menentukan lokasi indeks untuk menambahkan elemen baru, serta jumlah elemen yang ingin dihapus (jika ada).

```
var donuts = ["glazed", "chocolate frosted", "Boston creme",
"glazed cruller"];
donuts.splice(1, 1, "chocolate cruller", "creme de leche"); //
removes "chocolate frosted" at index 1 and adds "chocolate
cruller" and "creme de leche" starting at index 1
```

Returns: ["chocolate frosted"]

donuts array after calling the splice() method: ["glazed", "chocolate cruller", "creme de leche", "Boston creme", "glazed cruller"]

Berikut adalah sintaks dari metode **splice()**:

arrayName.splice(arg1, arg2, item1, ..., itemX); Di mana :

- **arg1** = Argumen wajib. Menentukan posisi indeks awal untuk menambah/menghapus item. Kita dapat menggunakan nilai negatif untuk menentukan posisi dari akhir array misalnya, -1 menentukan elemen terakhir.
- **arg2** = Argumen opsional. Menentukan jumlah elemen yang akan dihapus. Jika diatur ke 0, tidak ada item yang akan dihapus.
- **item1, ..., itemX** adalah item yang akan ditambahkan pada posisi indeks arg1

metode splice() mengembalikan item yang telah dihapus.

splice() adalah metode yang sangat kuat, yang memungkinkan kita untuk memanipulasi array dengan berbagai cara. Dalam satu baris kode sederhana kita bisa menambahkan atau menghapus elemen dari array.

Array Loops

Setelah data berada di dalam array, kita ingin mengakses dan memanipulasi setiap elemen dalam array secara efisien tanpa menulis kode berulang untuk setiap elemen.

Misalnya, jika ini adalah array donat asli kita:

```
var donuts = ["jelly donut", "chocolate donut", "glazed donut"];
```

dan kita memutuskan untuk membuat semua jenis donat yang sama, tetapi hanya menjualnya sebagai *donut holes*, kita dapat menulis kode berikut:

```
donuts[0] += " hole";
donuts[1] += " hole";
donuts[2] += " hole";
```

donuts array: ["jelly donut hole", "chocolate donut hole", "glazed donut hole"]

Tapi ingat, kita memiliki alat canggih lain yang bisa digunakan, yaitu **loop**!

Untuk looping melalui array, kita bisa menggunakan variabel untuk mewakili indeks dalam array, dan kemudian mengulang indeks tersebut untuk melakukan manipulasi apa pun yang diinginkan.

```
var donuts = ["jelly donut", "chocolate donut", "glazed donut"];

// the variable `i` is used to step through each element in the array
for (var i = 0; i < donuts.length; i++) {
    donuts[i] += " hole";
    donuts[i] = donuts[i].toUpperCase();
}
```

donuts array: ["JELLY DONUT HOLE", "CHOCOLATE DONUT HOLE", "GLAZED DONUT HOLE"]

Dalam contoh ini, variabel **i** digunakan untuk mewakili indeks array. Saat **i** bertambah, kita melangkahi setiap elemen dalam array mulai dari **0** hingga **donuts.length - 1** (**donuts.length** berada pada out of bounds).

The forEach Loop

Array memiliki sekumpulan metode khusus untuk membantu kita mengulangi dan melakukan operasi pada kumpulan data. Beberapa hal penting yang perlu diketahui adalah metode **forEach()** dan **map()**.

Metode **forEach()** memberikan kita cara alternatif untuk mengulangi array, dan memanipulasi setiap elemen dalam array dengan inline function expression.

```
var donuts = ["jelly donut", "chocolate donut", "glazed
donut"];

donuts.forEach(function(donut) {
  donut += " hole";
  donut = donut.toUpperCase();
  console.log(donut);
});
```

Prints:

JELLY DONUT HOLE
CHOCOLATE DONUT HOLE
GLAZED DONUT HOLE

Perhatikan bahwa metode **forEach()** mengiterasi array tanpa memerlukan indeks yang ditentukan secara eksplisit. Pada contoh di atas, **donut** sesuai dengan elemen di dalam array itu sendiri. Ini berbeda dengan perulangan **for** atau **while** di mana indeks digunakan untuk mengakses setiap elemen dalam larik:

```
for (var i = 0; i < donuts.length; i++) {
  donuts[i] += " hole";
  donuts[i] = donuts[i].toUpperCase();
  console.log(donuts[i]);
}
```

Parameters

Fungsi yang kita teruskan ke metode **forEach()** bisa menggunakan hingga tiga parameter, ini disebut **element**, **index**, dan **array**, tetapi kita dapat memanggilnya sesuka hati.

Metode **forEach()** akan memanggil fungsi ini satu kali untuk setiap elemen dalam array (maka namanya **forEach**.) Setiap kali, ia akan memanggil fungsi dengan argumen yang berbeda. Parameter **element** akan mendapatkan nilai dari elemen array. Parameter **index** akan mendapatkan indeks dari elemen (dimulai dengan nol). Parameter **array** akan mendapatkan referensi ke seluruh array, yang berguna jika Anda ingin memodifikasi elemennya.

Ini contohnya:

```
words = ["cat", "in", "hat"];
words.forEach(function(word, num, all) {
```

```
  console.log("Word " + num + " in " + all.toString() + " is "
+ word);
});
```

Prints:

Word 0 in cat,in,hat is cat

Word 1 in cat,in,hat is in

Word 2 in cat,in,hat is hat

Map

Menggunakan **forEach()** tidak akan berguna jika kita ingin mengubah array asli secara permanen. **forEach()** selalu mengembalikan **undefined**. Namun, membuat array baru dari array yang sudah ada sangatlah mudah, yaitu dengan metode **map()**.

Dengan metode **map()** , kita dapat mengambil array, melakukan beberapa operasi pada setiap elemen dari array, dan mengembalikan array baru.

```
var donuts = ["jelly donut", "chocolate donut", "glazed
donut"];

var improvedDonuts = donuts.map(function(donut) {
  donut += " hole";
  donut = donut.toUpperCase();
  return donut;
});
```

donuts array: ["jelly donut", "chocolate donut", "glazed donut"]

improvedDonuts array: ["JELLY DONUT HOLE", "CHOCOLATE DONUT HOLE", "GLAZED DONUT HOLE"]

Metode **map()** menerima satu argumen, sebuah fungsi yang akan digunakan untuk memanipulasi setiap elemen dalam array. Dalam contoh di atas, kita menggunakan function expression untuk meneruskan fungsi tersebut ke **map()**. Fungsi ini menerima satu argumen, donat yang sesuai dengan setiap elemen dalam array donat. Kita tidak perlu lagi mengulang indeks. **map()** melakukan semuanya untuk kita.

2D Donut Arrays

Seringkali, donat disusun dalam kotak seperti ini:

Kita bisa menggunakan array yang memiliki nama setiap donat yang dikaitkan dengan posisinya di dalam kotak.

Berikut contohnya:

```
var donutBox = [
  ["glazed", "chocolate glazed", "cinnamon"],
  ["powdered", "sprinkled", "glazed cruller"],
  ["chocolate cruller", "Boston creme", "creme de leche"]
];
```

Jika kita ingin melakukan looping dari donutBox dan menampilkan setiap donat (bersama dengan posisinya di dalam kotak), kita akan mulai dengan menulis perulangan for untuk mengulang setiap baris kotak donat:

```
var donutBox = [
  ["glazed", "chocolate glazed", "cinnamon"],
  ["powdered", "sprinkled", "glazed cruller"],
  ["chocolate cruller", "Boston creme", "creme de leche"]
];

// here, donutBox.length refers to the number of rows of donuts
for (var row = 0; row < donutBox.length; row++) {
  console.log(donutBox[row]);
}
```

Prints:
["glazed", "chocolate glazed", "cinnamon"]
["powdered", "sprinkled", "glazed cruller"]
["chocolate cruller", "Boston creme", "creme de leche"]

Karena setiap baris adalah array donat, selanjutnya kita perlu menyiapkan looping bagian dalam untuk melakukan perulangan pada setiap sel dalam array.

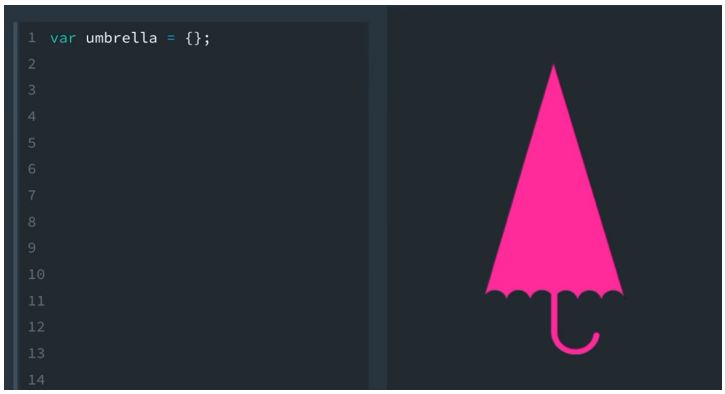
```
for (var row = 0; row < donutBox.length; row++) {
   // here, donutBox[row].length refers to the length of the
donut array currently being looped over
  for (var column = 0; column < donutBox[row].length; column++)
{
    console.log(donutBox[row][column]);
  }
}
```

Prints:
"glazed"
"chocolate glazed"
"cinnamon"
"powdered"
"sprinkled"
"glazed cruller"
"chocolate cruller"
"Boston creme"
"creme de leche"

Objects

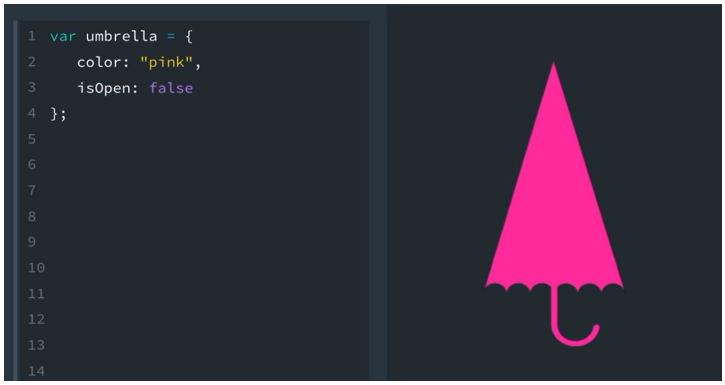
Objects

Jika kita menulis kode untuk mengontrol payung ini, pertama-tama kita akan membuat objek untuk mewakili payung. Salah satu cara untuk mendefinisikan objek adalah dengan membuat variabel dan menuliskannya ke sepasang kurung kurawal kosong seperti ini.

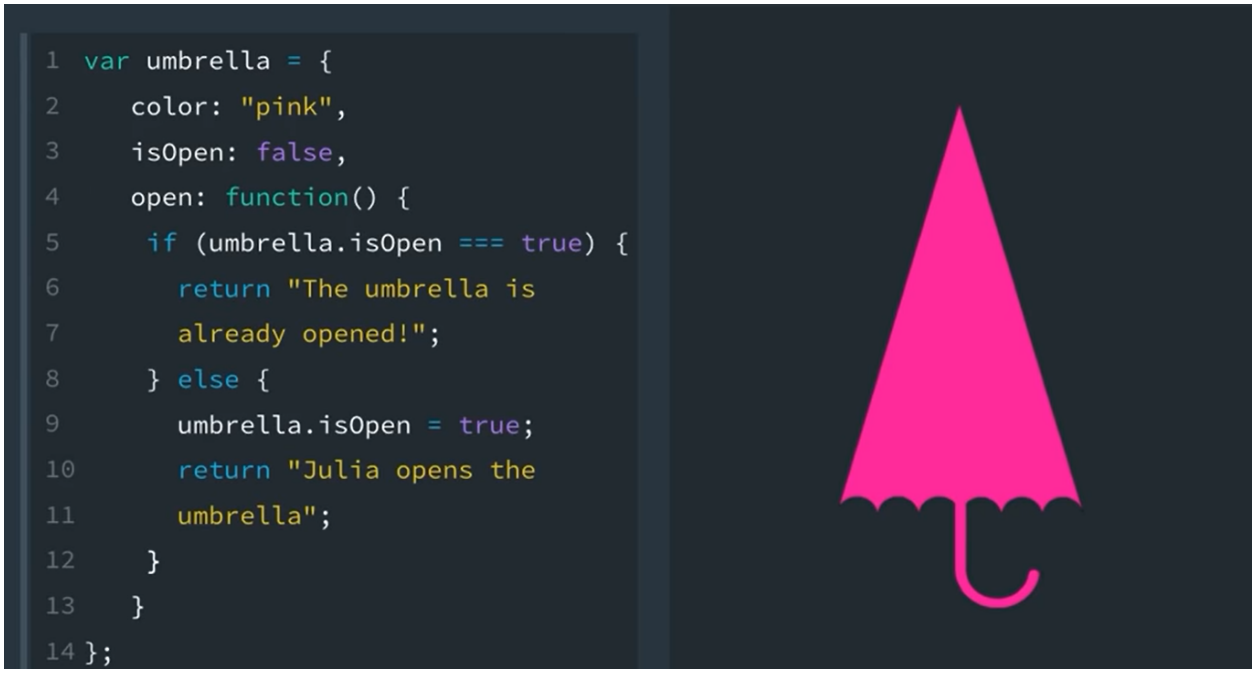


Kita dapat memverifikasi payung adalah objek, dengan menggunakan operator typeof. Operator typeof akan mengembalikan string yang memberi tahu kita tipe data.

Namun, objek kosong tidak terlalu menarik. Objek memiliki properti dan hal-hal yang bisa mereka lakukan. Untuk menambahkan informasi ini, kita dapat menentukan pasangan key-value untuk setiap bagian data. Di sini, kita telah mendefinisikan key properti yang disebut warna, dan valuenya pink, karena itu adalah payung merah muda. Payung ini juga tertutup, sehingga kita dapat menambahkan properti bernama isOpen dan menyetel value awalnya menjadi false.



Membuka payung adalah tugas yang kita ingin agar payung dapat melakukannya. Sesuatu yang dapat dilakukan objek adalah metode. Metode hanyalah fungsi yang terkait dengan objek. Kita sebenarnya sudah pernah melihat metode sebelumnya. Saat kita menggunakan push dan pop. Push dan pop adalah metode untuk menambah dan menghapus elemen dari array. Berikut adalah metode membuka payung.



Jika kita sedikit bingung, tidak apa-apa. Objek mungkin sulit untuk dipahami pada awalnya. Tapi bersabarlah. Kita akan membahas lebih detail tentang cara membuat objek, menentukan properti, dan metode, menggunakan objek ini dalam kode Anda.

Object Literals

```
var sister = {
  name: "Sarah",
  age: 23,
  parents: [ "alice", "andy" ],
  siblings: ["julia"],
  favoriteColor: "purple",
  pets: true
};
```

Sintaks yang kita lihat di atas disebut **object-literal notation**. Ada beberapa hal penting yang perlu kita ingat saat menyusun struktur dari objek literal:

- "Key" (mewakili **properti** atau nama **metode**) dan "nilainya" dipisahkan satu sama lain oleh **titik dua**
- Pasangan **key: value** dipisahkan satu sama lain dengan koma
- Seluruh objek dibungkus di dalam kurung kurawal **{ }**.

Dan, seperti halnya bagaimana kita bisa mencari kata di kamus, **key** dalam pasangan **key:value** memungkinkan kita mencari sepotong informasi tentang suatu objek. Berikut adalah beberapa contoh bagaimana kita bisa mengambil informasi tentang orang tua dari saudara perempuan dengan menggunakan objek yang kita buat.

```
// two equivalent ways to use the key to return its value
sister["parents"] // returns [ "alice", "andy" ]
sister.parents // also returns ["alice", "andy"]
```

Menggunakan **sister["parents"]** disebut **bracket notation** (karena menggunakan tanda kurung) dan penggunaan **sister.parents** disebut **dot notation** (karena menggunakan titik).

Objek di atas berisi banyak properti, tetapi tidak benar-benar mengatakan apa yang dilakukan saudara perempuan. Misalnya, saudara perempuan suka melukis. Kita mungkin memiliki metode **paintPicture()** yang mengembalikan "Sarah paints a picture!". Sintaks untuk ini hampir sama persis dengan cara kita mendefinisikan properti objek. Satu-satunya perbedaan adalah, nilai dalam pasangan key:value akan menjadi fungsi.

```
var sister = {
  name: "Sarah",
  age: 23,
  parents: [ "alice", "andy" ],
  siblings: ["julia"],
  favoriteColor: "purple",
  pets: true,
  paintPicture: function() { return "Sarah paints!"; }
};

sister.paintPicture();
```

Returns: "Sarah paints!"

dan kita dapat mengakses nama saudara perempuan dengan mengakses properti name:

```
sister.name
```

Returns: "Sarah"

Naming Conventions

Jangan ragu untuk menggunakan angka dan huruf besar dan kecil, tetapi jangan memulai nama properti kita dengan angka. Kita tidak perlu membungkus string dengan tanda kutip! Jika itu adalah properti lebih dari 2 kata, gunakan camelCase. Jangan gunakan tanda penghubung dalam nama properti.

```
var richard = {
  "1stSon": true;
  "loves-snow": true;
};

richard.1stSon // error
richard.loves-snow // error
```