Chapter 3. JavaScript ES6

let vs const vs var

Dari topik-topik sebelumnya, kita tahu bahwa untuk mendeklarasikan sebuah variabel, kita bisa menggunakan kata kunci `let` atau `const` dan diikuti nama variabelnya.

Wah bedanya apa ya? Coba simak penjelasan di bawah ini.

Deklarasi Variabel Menggunakan `let`

Saat mendeklarasikan variabel menggunakan `let`, variabel tersebut masih bisa diberi nilai yang baru nantinya.

Contoh:

```
let bilanganPi = 3.14;
bilanganPi = 3.;

console.log(bilanganPi); // Output: 3

bilanganPi = 3.2;
console.log(bilanganPi); // Output: 3.2
```

Penggunaan `let` cocok untuk variabel yang nilainya memang bisa berubah-ubah. Bagaimana kalau kita ingin variabel tersebut tidak bisa diubah nilainya?

Solusinya adalah menggunakan kata kunci `const`.

Deklarasi Variabel Menggunakan `const`

Syntax penggunaan `const` mirip dengan saat menggunakan `let`, yaitu diikuti nama variabelnya. Lalu apa bedanya `const` dengan `let`?

Nah, variabel yang dideklarasikan dengan `const` itu tidak bisa diberi nilai baru.

Contoh:

```
const bilanganPi = 3.14;
bilanganPi = 3.;  // Output TypeError: Assignment to constant variable
```

Karena variabel `bilanganPi` dideklarasikan menggunakan kata kunci `const`, kita tidak bisa memberi nilai baru pada variabel tersebut.

Deklarasi Object dan Array menggunakan `const`

Kesalahpahaman umum adalah `const` membuat sebuah variabel dengan nilai konstan. Ini pada umumnya benar, namun ada pengecualian. Untuk variabel dengan tipe objek atau array, variabel tersebut tidak bisa diganti nilainya, tetapi properti atau element-nya bisa.

Coba lihat contoh berikut:

```
const warna = ["merah", "kuning", "kelabu"];

// pemberian array baru pada variabel dengan `const` akan menampilkan error
warna = ["hijau", "abu", "biru"];  // Output TypeError: Assignment to constant
variable

// namun jika perubahan pada arraynya sendiri masih bisa dilakukan
warna.push("nila", "ungu");


console.log(warna); // Output: ["merah", "kuning", "kelabu", "nila", "ungu"]
```

Kalau kita lihat contoh di atas, saat kita berusaha memberi array baru pada variabel `warna`, akan muncul error. Sebab seperti yang tadi kita bahas, variabel yang dideklarasikan dengan `const` tidak bisa diberi nilai baru.

Namun lihat pada saat kita menggunakan method `push()` pada variabel `warna`. Tidak ada error. Kenapa? Karena kita memang hanya mengubah nilai (menambah element baru) dari variabel `warna`, bukan memberinya nilai baru.

Deklarasi variabel menggunakan var

Dalam JavaScript, sebetulnya ada satu lagi cara mendeklarasikan variabel, yaitu dengan kata kunci `var`. Sudah hampir tidak ada yang memakai `var` kecuali untuk kode JavaScript lama yang tidak mendukung ES6.

Kenapa tidak dipakai lagi?

Deklarasi variabel menggunakan `var` tidak akan memunculkan error walaupun variabel dengan nama yang sama sudah pernah dideklarasikan. Jadi misalkan kita pernah membuat sebuah variabel, lalu entah karena lupa atau tidak sengaja kita membuat variabel dengan nama yang sama, nilai dari variabel yang pertama akan ditimpa dengan yang baru.

Contoh:

```
// deklarasi variabel bernama bahasaFavorit
var bahasaFavorit = "JavaScript";

// baris kode lain
```

```
// ...

// tidak sengaja kita membuat variabel dengan nama yang sama
var bahasaFavorit = "Python";

console.log(bahasaFavorit); // Output: Python
```

Pada kode di atas, kita tidak sengaja membuat variabel lagi bernama `bahasaFavorit` padahal sudah pernah ada variabel dengan nama tersebut. Yang menjadi masalah adalah kode tersebut tidak menghasilkan error sama sekali.

Bandingkan kalau kita menggunakan `let` atau `const`

```
// deklarasi variabel menggunakan const dan let
const nama = "Rafi";
let umur = 16;

// kalau kita coba deklarasi lagi dengan nama variabel yang sama
// akan muncul error
const nama = "Farhan"; // Output SyntaxError: Identifier 'nama' has already been declared
let umur = 21; // Output SyntaxError: Identifier 'nama' has already been declared
```

Dengan begini kita jadi tahu bahwa sudah pernah ada variabel dengan nama yang sama.

Scoping Variabel dengan var

Kelemahan lain dari deklarasi variabel menggunakan `var` adalah ia tidak memiliki block scoping.

Apa itu block scoping?

Sebuah variabel yang dideklarasikan di dalam blok `if-else`, `switch-case`, `while`, dan `for` seharusnya memang tidak bisa diakses dan diubah nilainya dari luar blok tersebut.

Contoh:

```
var judulBuku = "Harry Potter";

if (judulBuku === "Harry Potter") {
  var pengarang = "J.K. Rowling";
}

console.log(pengarang); // Output: J.K. Rowling
```

Pada kode di atas, variabel `pengarang` dideklarasikan di dalam blok `if`. Tapi dia tetap bisa diakses di luar blok tersebut (lihat output dari baris terakhir).

Bandingkan dengan contoh menggunakan `let`:

```
let judulBuku = "Harry Potter";

if (judulBuku === "Harry Potter") {
  let pengarang = "J.K. Rowling";
}

console.log(pengarang); // Output ReferenceError: pengarang is not defined
```

String Literal

Masih ingat bagaimana cara menggabungkan string dengan nilai pada variabel lain (string concatenation)? Cukup dengan menggunakan operator + , kamu dapat melakukan proses penggabungan.

Contohnya seperti ini:

```
let nama = "sarah";
let umur = 24;

let kalimat = "Namanya " + nama + ". Umurnya " + umur + " tahun.";
console.log(kalimat);

/*
Output:
"Namanya sarah. Umurnya 24 tahun."
*/
```

Bayangkan jika variabelnya ada banyak, kemudian kalimatnya sangat panjang. Akan sangat capek sekali apabila kita harus menggunakan operator + untuk menggabungkan setiap variabelnya.

Di sinilah String literal berguna.

String literal adalah cara untuk menanamkan ekspresi JavaScript ke dalam sebuah string. String literal dibungkus dengan sepasang tanda `(backtick), dan ekspresi JavaScript-nya dibungkus dengan tanda `${ }`.

Syntax-nya seperti berikut ini:

```
let namaVariabel = `${ekspresiJavaScript}`;

*"Ekspresi? Menanamkan? Pelan-pelan dong Mbak"*
```

Ekspresi dalam JavaScript adalah sesuatu yang jika dievaluasi akan menghasilkan sebuah nilai tunggal. Bisa itu variabel, operasi matematika, atau operasi logika.

"Jadi di tengah-tengah string nanti bisa ada variabel atau operasi matematika gitu?"

Iya, Benar sekali!

Coba kita ambil contoh pertama ya.

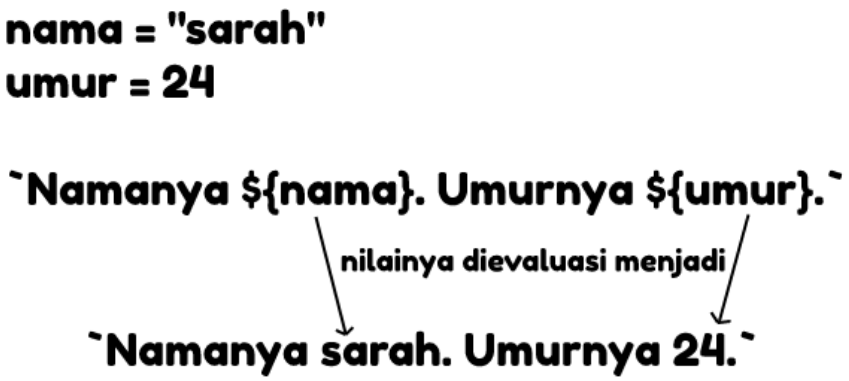
```javascript
let nama = "sarah";
let umur = 24;

let kalimat = `Namanya ${nama}. Umurnya ${umur} tahun.`

console.log(kalimat);

/*
Output:
"Namanya sarah. Umurnya 24 tahun."
*/
```

Pada contoh di atas, variabel `nama` dan `umur` akan dievaluasi nilainya menjadi `"sarah"` dan `24`. Kedua nilai ini baru dimasukkan menjadi bagian dari string.



Seperti yang tadi dibahas, kita juga bisa memasukkan operasi matematika atau operasi logika dengan String literal. Contohnya seperti ini:

```javascript
let umur = 16;

let kalimat = `Umur saya tahun depan adalah ${umur + 1}`;
```

```
console.log(kalimat); // Output: Umur saya tahun depan adalah 17

let kalimat2 = `Tahun depan saya ${umur + 1 >= 17 ? 'bisa' : 'tidak bisa'} ikut
pemilu`;

console.log(kalimat2); // Output: Tahun depan saya bisa ikut pemilu
```

Pada kode di atas, `umur + 1` di variabel `kalimat` akan dievaluasi menjadi `22` dulu sebelum ia menjadi bagian dari string. Begitu juga kode `umur + 1 >= 17 ? 'bisa' : 'tidak bisa'` akan dievaluasi menjadi `'bisa'` sebelum ia menjadi bagian dari string di `kalimat2`.

Jangankan variabel dan operasi matematika, fungsi pun bisa. Perhatikan contoh berikut:

```
// fungsi yang mengembalikan sebuah string
function namaSaya() {
  return "Krishna";
}

let kalimatPerkenalan = `Halo nama saya adalah ${namaSaya()}`;

console.log(kalimatPerkenalan); // Output: Halo nama saya adalah Krishna
```

Begitulah kegunaan template literal. Kamu dapat menanamkan semua jenis ekspresi JavaScript ke dalam sebuah string, tanpa perlu menggunakan string concatenation satu persatu lagi.

Arrow Function

Menggunakan Arrow Function

Masih ingat cara mendeklarasikan fungsi?

Untuk mengingat kembali, mari lihat contoh di bawah ini:

```
function ucapkanSalam() {
  return "Selamat Pagi";
};
```

Di Javascript ES6, ada cara baru dalam penulisan sebuah fungsi, yaitu menggunakan arrow function. Kelebihan menggunakan arrow function adalah penulisan fungsi menjadi lebih singkat dan lebih mudah dibaca.

Syntax menggunakan arrow function adalah sebagai berikut:

```
// Arrow function
const namaFungsi = (parameter1, ..., parameterX) => {
  // kode yang ingin dijalankan dalam fungsi
};

// atau bila fungsi tersebut tidak memiliki parameter sama sekali
const namaFungsiTanpaParameter = () => {
  // kode yang ingin dijalankan dalam fungsi
};
```

Contoh:

```
// fungsi dengan parameter
const operasiPenjumlahan = (bilangan1, bilangan2) => {
  const hasil =  bilangan1 + bilangan2;
  return hasil;
};

console.log(operasiPenjumlahan(3, 4)); // Output: 7

// fungsi yang tidak memiliki parameter
const namaJenisAnjing = () => {
  const anjing = ["Pug", "Bulldog", "Poodle"];
  return anjing[Math.floor(Math.random()*(anjing.length))];
}

console.log(namaJenisAnjing()); // Output: Pug (hasil random)
```

Implicit Return Value

Yang dimaksud dengan implicit return value adalah suatu kondisi di mana sebuah fungsi langsung mengembalikan nilai tanpa ada deklarasi variabel atau operasi lainnya di dalamnya.

Misalnya kita mempunyai satu fungsi seperti berikut ini:

```
function greeting(nama) {
   return `Hi ${nama}`;
}

console.log(greeting('Einstein')); // Output: Hi Einstein
```

Kode di atas bisa disingkat dengan menggunakan arrow function sebagai berikut:

```
const greeting = (nama) => `Hi ${nama}`;

console.log(greeting('Einstein')); // Output: Hi Einstein
```

Untuk fungsi yang tidak memiliki parameter, syntax arrow function untuk implicit return value adalah sebagai berikut:

```
const namaFungsi = () => nilaiReturn;
```

Jadi kita tidak lagi perlu capek-capek menulis kurung kurawal `{}` dan kata `return` lagi.

Default Parameter

Kalian masih ingat tidak apa itu `parameter` dan `argument`? Biar tidak lupa, mari kita coba bahas sedikit yuk.

Parameter adalah syarat input yang harus dimasukkan ke dalam suatu fungsi dan dideklarasikan bersama dengan deklarasi fungsi.

Sementara argument adalah nilai yang dimasukkan ke dalam suatu fungsi, sesuai dengan persyaratan parameter, di mana argument dituliskan bersamaan dengan pemanggilan fungsi.

Contoh

```
function panggang(bahan, durasi, suhu) {
  return `Panggang ${bahan} selama ${durasi} pada suhu ${suhu}`;
}

panggang("Roti", "10 menit", "100 C"); // Output: Panggang Roti selama 10 menit pada suhu 100 C
```

Pada contoh di atas, `(bahan, durasi, suhu)` itu merupakan parameter dari fungsi `panggang`. Artinya saat kita memanggil fungsi tersebut, ia akan menantikan argument yang sesuai dengan parameternya. Dalam kode di atas, argumentnya adalah `"Roti"`, `"10 menit"`, dan `"100 C"`.

Apa yang terjadi kalau kita lupa memberinya argument yang sesuai dengan parameternya?

Misalnya kita hanya memberinya 2 buah argument: `panggang("Roti", "10 menit")`.

Atau bahkan tidak sama sekali: `panggang()`.

Hasilnya akan seperti ini:

```
console.log(panggang("Roti", "10 menit"));
// Output: Panggang Roti selama 10 menit pada suhu undefined

console.log(panggang());
```

```
// Output: Panggang undefined selama undefined pada suhu undefined
```

Pada dasarnya jika kita tidak memberi argument padahal fungsi tersebut menantikan sebuah argument, nilai default yang akan dipakai adalah `undefined`..

Pada contoh kode di atas mungkin tidak ada error, tapi bagaimana jika kodenya seperti di bawah ini:

```
function cetakNama(orang) {
  return orang.nama;
}

cetakNama(); // Output TypeError: Can't read the property 'nama' of undefined
```

Kode di atas akan menghasilkan error. Karena saat kita tidak memberikan argument untuk parameter `orang`, nilai defaultnya adalah `undefined`. Dan saat fungsi `cetakNama` ingin membaca properti `nama` dari `orang`, di situlah muncul error karena `orang` bernilai `undefined`.

Bagaimana cara menangani masalah seperti ini?

Bagaimana caranya kita memastikan kalau parameter pada fungsi yang kita buat tidak bernilai `undefined`?
Solusinya adalah menggunakan default parameter.

Default parameter adalah nilai pengganti yang akan diberikan ke sebuah fungsi apabila fungsi tersebut tidak diberikan argument yang cocok dengan parameter fungsi tersebut (atau argument yang diberikan bernilai `undefined`).

Syntax-nya adalah seperti ini:

```
function namaFungsi(parameter1 = defaultParameter1) {
  // kode yang akan dijalankan di dalam fungsi
}
```

Jadi kalau misalnya fungsi tersebut tidak menerima argument yang seharusnya menjadi nilai `parameter1`, nilai default yang akan diambil oleh `parameter1` adalah `defaultParameter1`.

Kita ambil contoh fungsi yang pertama tadi.

```
function panggang(bahan = "makanannya", durasi = "yang diperlukan", suhu = "yang
cocok") {
  return `Panggang ${bahan} selama ${durasi} pada suhu ${suhu}`;
}
```

Sehingga kalau kita memanggil fungsi tersebut tanpa argument yang lengkap, hasilnya akan seperti ini:

```
panggang("Risoles", "5 menit"); // Output: Panggang Risoles selama 5 menit pada suhu yang cocok

panggang(); // Output: Panggang makanannya selama yang diperlukan pada suhu yang cocok
```

Rest Parameter & Spread Operator

Rest parameter dan spread operator itu bisa kita bilang kembar: *mirip tapi tidak sama*. Sebab keduanya sama-sama menggunakan tanda `...` (elipsis atau tiga buah titik) sebagai bagian dari syntaxnya, namun fungsinya sangat berbeda.

Penasaran tidak kegunaannya? Mari kita bahas satu per-satu.

Rest Parameter

Rest Parameter adalah parameter yang mewakili nilai dari semua (atau sisa) argument yang diberikan kepada suatu fungsi.

Contoh:

```
const foo = (...params) => {
  console.log(params);
};

foo("A", "B", "C"); // Output: ["A", "B", "C"]
```

Kalau kalian perhatikan kode di atas, nilai dari parameter `params` berisi semua nilai argument yang kita berikan pada saat pemanggilan fungsi `foo`.

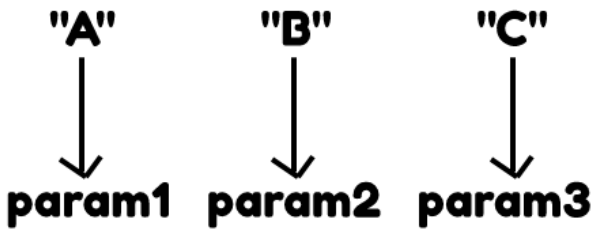
Kok bisa gitu?

Coba cermati parameter `params`. Ada `...` (elipsis) di depannya kan? Parameter yang diawali dengan elipsis menandakan ia adalah sebuah rest parameter. Jika sebuah fungsi dengan satu parameter berupa rest parameter dipanggil dan diberikan satu atau lebih argument, rest parameter-nya akan mewakili semua argument-argument tersebut.

tanpa rest parameter

```
function foo(param1, param2, param3) {
   //isi fungsi
}

foo("A", "B", "C")
```

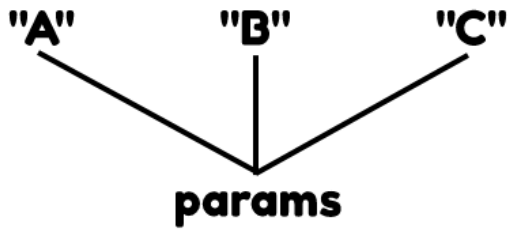


---

dengan rest parameter

```
function foo(...params) {
   //isi fungsi
}

foo("A", "B", "C")
```



Rest Parameter Sebagai Parameter Terakhir

Rest parameter juga bisa digunakan saat ada parameter lain pada fungsi yang sama. Dalam kasus ini, rest parameter akan mewakili sisa argument yang belum "dipetakan" ke parameter-parameter sebelumnya.

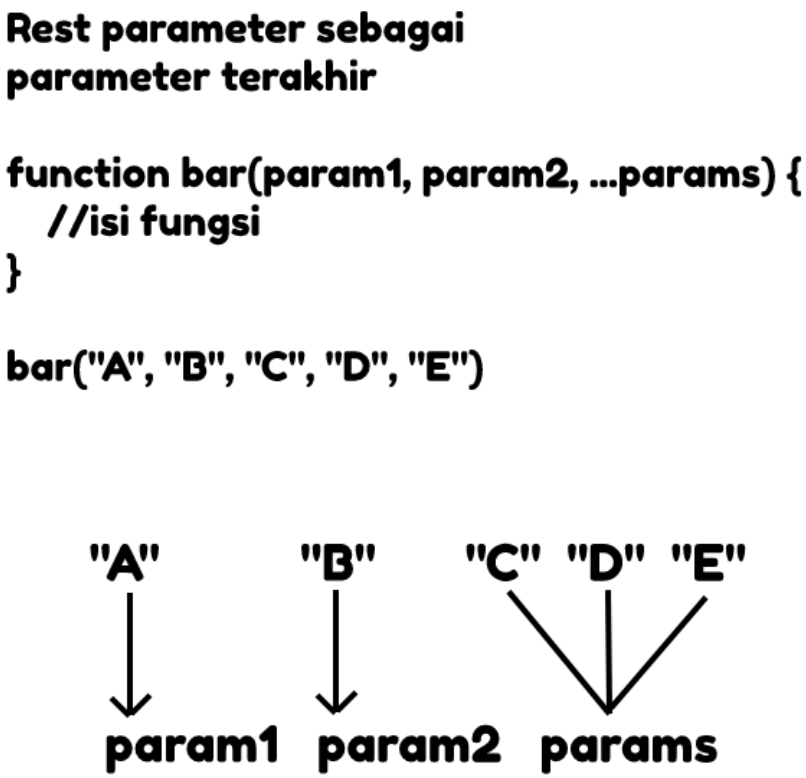
Contoh:

```
const bar = (param1, param2, ...params) => {
  console.log("Argument pertama ", param1);
  console.log("Argument pertama ", param2);
  console.log("Sisa argument ", params);
}

bar("A", "B", "C", "D", "E");
// Output:
// Argument pertama  A
// Argument pertama  B
// Sisa argument ["C", "D", "E"]
```

Pada kode di atas,

- `param1` mewakili argument pertama, yaitu `"A"`,
- `param2` mewakili argument pertama, yaitu `"B"`,
- `params` mewakili sisa argument, yaitu `"C"`, `"D"`, dan `"E"`.



Paham ya, kalau rest parameter ditaruh sebagai parameter terakhir, ia akan mengambil sisa dari argument yang kita berikan pada fungsi tersebut.

Pada dasarnya, hanya bisa ada satu rest parameter dalam sebuah fungsi. Dan dia harus ditempatkan di paling belakang. Kalau tidak, akan muncul error saat kita mendeklarasikan fungsinya.

Contohnya:

```
// Tempatkan rest parameter di tengah-tengah parameter lainnya
const buzz = (param1, ...params, param2) => {
  console.log(param1);
  console.log(params);
  console.log(param2);
}

// Output: SyntaxError: Rest parameter must be last formal parameter
```

Spread Operator

Berlawanan dengan rest parameter, sifat dari spread operator justru memisahkan/memecahkan. Ia memecah sebuah array menjadi element-elementnya (atau dalam kasus objek, memecah objek menjadi pasangan properti-nilai yang membentuknya).

Spread Operator Dengan Array

```
let arrayBilangan = [1, 2, 3, 4, 5];

// tanpa spread operator
console.log(arrayBilangan); // Output: [1, 2, 3, 4, 5]

// kalau pakai spread operator
console.log(...arrayBilangan); // Output: 1, 2, 3, 4, 5
```

Saat menggunakan spread operator, outputnya bukan lagi array, melainkan element-element pembentuknya secara individu.

Spread operator pada array memiliki beberapa kegunaan:

membuat array duplikat

```
let arrayAsli = [1, 2, 3];

let arrayDuplikat = [...arrayAsli];
```

Sekarang `arrayDuplikat` adalah array baru yang memiliki element-element yang sama dengan `arrayAsli`.
Kalian mungkin bertanya "*Kenapa tidak langsung seperti ini saja?*"

```
let arrayAsli = [1, 2, 3];
```

- `let arrayDuplikat = arrayAsli;`
  Dalam kode di atas, `arrayAsli` dan `arrayDuplikat` itu menunjuk pada array yang sama, sehingga kalau kita membuat perubahan pada `arrayDuplikat`, `arrayAsli` juga ikut berubah. Sebaliknya juga begitu.

menggabungkan array (concatenate)
```
let array1 = ["foo", "bar"];
let array2 = ["fizz", "buzz"];

// menggunakan spread operator
array1 = [...array1, ...array2];
```

- `console.log(array1); // Output: ["foo", "bar", "fizz", "buzz"]`

Spread Operator Dengan Objek

Mirip pada array, spread operator memecah objek menjadi pasangan properti-nilai yang membentuknya. Hal ini memiliki beberapa kegunaan:

menambah beberapa properti sekaligus
```
let orang = {
  nama: "Yudistya",
  umur: 32
};
```
Untuk menambahkan beberapa properti lainnya, kita bisa melakukan ini:
```
orang.pekerjaan = "arsitek";
orang.pendidikan = "S1";
```

atau bisa juga lebih singkat dengan menggunakan spread operator seperti ini:

```
orang = {...orang, pekerjaan: "arsitek", pendidikan: "S1"};
```

menggabungkan objek
Spread operator juga memungkinkan kita untuk menggabungkan dua (atau lebih) objek. Sebelum ES6, kita harus menggunakan method `.assign()` dari `Object` seperti ini:
```
const objek1 = {a: 1, b: 2};
const objek2 = {c: 3, d: 4};
const objekGabungan = Object.assign({}, objek1, objek2);

console.log(objekGabungan); // Output: {a: 1, b: 2, c: 3, d: 4}
```
Tapi dengan spread operator, kita cukup menambahkan elipsis di depan nama arraynya lalu membungkus semuanya dengan sepasang kurung kurawal.
```
const objek1 = {a: 1, b: 2};
const objek2 = {c: 3, d: 4};
```

```
const objekGabungan = {...objek1, ...objek2};
```

- `console.log(objekGabungan); // Output: {a: 1, b: 2, c: 3, d: 4}`

Destructuring

Destructuring/Destrukturisasi adalah cara untuk membongkar isi dari array atau objek dan menyimpan hasilnya ke dalam variabel-variabel lain.

Mari kita bahas lebih lanjut!

Destrukturisasi Array

Sebelum ES6, cara untuk menyimpan nilai dari masing-masing element dari sebuah array ke dalam variabel adalah sebagai berikut:

```
let buah = ["mangga", "pisang", "anggur"];

let buah1 = buah[0];
let buah2 = buah[1];
let buah3 = buah[2];

console.log(buah1); // Output: mangga
console.log(buah2); // Output: pisang
console.log(buah3); // Output: anggur
```

Selain tidak efisien karena harus menulis assignment untuk tiap variabel, kita juga harus menghitung nilai yang kita ingin ambil itu ada di index keberapa.

Untungnya ES6 menyediakan cara untuk mendapatkan hasil yang sama dengan cara yang lebih singkat.

Syntax-nya mirip dengan cara mendeklarasikan sebuah array, namun kali ini nama array-nya ada di sebelah kanan dan variabel untuk menyimpan element-element yang ingin dibongkar ada di sebelah kiri.

Contoh:

```
// cara mendeklarasikan variabel berupa array
let namaArray = [element1, element2, element3];

// cara melakukan destrukturisasi array
```

```
let [variabel1, variabel2, variabel3] = namaArray;
```

Kita ambil contoh array yang pertama ya:

```
let buah = ["mangga", "pisang", "anggur"];

// lakukan destrukturisasi array
let [buah1, buah2, buah3] = buah;

console.log(buah1); // Output: mangga
console.log(buah2); // Output: pisang
console.log(buah3); // Output: anggur
```

Lebih singkat bukan kalau menggunakan destrukturisasi array?

Kalian juga bisa memilih element mana saja yang tidak ingin disimpan di dalam variabel dengan menggunakan "koma kosong", atau tidak menyebutkan element tersebut saat destrukturisasi.

Contoh:

```
// gunakan koma kosong untuk melewati element kedua dan keempat
let [namaDepan, , namaBelakang, , buku] = ["Joanne", "K", "Rowling", "pengarang",
"Harry Potter"];

console.log(namaDepan); // Output: Joanne
console.log(namaBelakang); // Output: Rowling
console.log(buku); // Output: Harry Potter
```

Pada kode di atas, karena kita mengosongkan tempat kedua dan keempat pada saat destrukturisasi array, element kedua (`"K"`) dan element keempat (`"Pengarang"`) tidak akan ikut disimpan dalam variabel seperti element pertama, ketiga, dan kelima.

Kalian ingin menukar nilai dari dua variabel (atau lebih) dengan lebih cepat dan tanpa menggunakan variabel tambahan? Bisa pakai destrukturisasi array.

Sebelumnya mungkin kita harus melakukan ini:

```
let a = 10;
let b = 15;

// variabel tambahan sementara
let temp = a;
a = b; // nilai a sekarang adalah 15
b = temp; // nilai b sekarang adalah 10
```

Sekarang kita cukup melakukan ini:

```
let a = 10;
let b = 15

[a,b] = [b,a];
// nilai a sekarang adalah 15, dan nilai b adalah 10
```

Destrukturisasi Objek

Sama seperti destukturisasi array, sebelum ES6, kita harus menulis assignment satu per satu untuk tiap nilai dalam objek yang kita ingin bongkar. Seperti ini:

```
let orang = {
  nama: "Joko",
  umur: 18,
  sudahMenikah: false
};

let nama = orang.nama;
let umur = orang.umur;
let sudahMenikah = orang.sudahMenikah;

console.log(nama); // Output: Joko
console.log(umur); // Output: 22
console.log(sudahMenikah); // Output: false
```

Pada Javascript ES6, caranya lebih singkat, yaitu dengan destrukturisasi.

Syntax untuk destrukturisasi objek adalah seperti ini:

```
let { namaProperti1, namaProperti2, namaProperti3 } = namaObjek;
```

Cara di atas hanya bekerja jika nama variabel baru untuk destrukturisasi itu sama dengan nama properti pada objek yang akan didestrukturisasi ya. Kalau tidak nanti isi dari variabelnya adalah `undefined`.

Misalnya:

```
let orang = {
  nama: "Joko",
  umur: 22,
  sudahMenikah: false
};

// Destrukturisasi Objek
let { nama, umur, statusMenikah } = orang;

console.log(nama); // Output: Joko
```

```
console.log(umur); // Output: 22
console.log(statusMenikah); // Output: undefined karena nama variabel dan properti berbeda
```

Tapi kalau kalian ingin agar nama variabelnya berbeda dari nama properti, kalian bisa menambahkan titik dua setelah nama properti dan diikuti nama variabel yang baru, seperti ini:

```
let {
  namaProperti1: namaVariabelBaru1,
  namaProperti2: namaVariabelBaru2
} = namaObjek;
```

Contoh:

```
let orang = {
  name: "Joko",
  age: 22,
};

// buat variabel nama berisi nilai dari properti name pada orang
// buat variabel age berisi nilai dari properti umur pada orang
let { name: nama, age: umur } = orang;

console.log(nama); // Output: Joko
console.log(umur); // Output: 22
```

Module

Introduction

Mungkin aplikasi *project* kamu sekarang masih sangat sederhana dan hanya menggunakan satu *file JavaScript* saja yang dihubungkan ke *file* `.html`. Tapi, bagaimana jika nantinya aplikasi yang kamu buat menjadi semakin besar? Dengan semakin besarnya aplikasi, maka kode yang kita miliki akan semakin kompleks dan semakin mempersulit kita untuk mencari penyelesaian jika terjadi *error* atau *bug*. Untuk mempermudah jika kita ingin menambah atau mengurangi kode kita, kode-kode tersebut bisa kita pisahkan menjadi beberapa file.

Seperti seorang penulis ketika membuat sebuah jurnal atau buku, biasanya memisahkan menjadi beberapa bab atau *chapter* dengan tujuan untuk mempermudah pembacanya mengerti tentang topik apa yang sedang dibahas. Sama seperti buku, dengan menggunakan *module* kita bisa memisahkan kode pada aplikasi kita sesuai dengan *scope* yang seragam sehingga mudah dipahami apa yang ada di dalam *module* tersebut.

Module adalah sebuah cara bagi *JavaScript* untuk mengisolasi kode dari suatu *file* ke dalam sebuah *file* terpisah. Sehingga kode tersebut dapat digunakan berulang kali dengan cara di-*export* dari suatu *file* dan di-*import* ke *file* yang lainnya. Kita dapat melakukan *export* kode apapun pada *JavaScript* seperti *string*, *object*, *array*, *number*, hingga *function*.

Mengapa menggunakan *module*?

---

1. *Maintainability*
    - *Module* yang dirancang dengan baik bisa mengurangi ketergantungan pada bagian tertentu pada kode kita. Merubah satu *module* lebih mudah ketika *module* dipisahkan dari potongan kode lainnya daripada merubah dalam satu file yang terdiri dari ratusan ribu kode.
    - Mempermudah jika kita ingin menambahkan, menghapus, dan merubah kode kita karena tidak mempengaruhi keseluruhan aplikasi kita.
2. Penggunaan Nama Variabel
    - *Module* memudahkan kita untuk memberikan *alias* nama variabel yang di-*import*. Sehingga kita tidak mengalami kesulitan untuk mengganti nama variabel jika nama variabel yang kita *import* sama dengan nama variabel dalam *file* yang menggunakan *module* tersebut.
3. *Reusable Code*
    - Kita sangat sering menggunakan kode yang sama baik itu variabel atau *function* dari satu *file* ke *file* yang lain. Padahal jika kita ingin menjadi programmer yang baik, kita harus menggunakan prinsip DRY (Don't Repeat Yourself) pada kode kita. Untuk itu sebaiknya jika kita akan membuat kode yang nantinya akan dapat digunakan pada *file* yang lain, maka kita perlu membuat sebagai *module*.

Penggunaan *module* cukup mudah. Kita akan menggunakan *syntax* dari versi JavaScript ES6 seperti pada kelas *JavaScript Dasar*. Kamu cukup menambahkan *attribute* `type` pada *tag* `script` kemudian isi nilainya dengan `module`. Sehingga menjadi seperti ini :

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

module ini tidak dapat berjalan jika kamu mengakses file proyek `.html` menggunakan *url* direktori lokal. Untuk bisa menjalankan *file html* yang berisi *module* kita perlu menggunakan *static server* salah satunya dengan menggunakan *Live Server* pada *Visual Studio Code*.

Export and Import

Module pada *file* JavaScript membutuhkan penghubung antar satu *file* dengan *file* yang lain.

Untuk bisa menghubungkan *file* antar JavaScript kita bisa menggunakan `export` dan `import` sehingga memungkinkan untuk saling menggunakan kode antar *module*.

Export

Export digunakan untuk meng-*export* variabel pada *file JavaScript*. Variabel yang di*export* dapat berisi data seperti *string*, *object*, *array*, hingga *function*. Hal ini dilakukan agar *file JavaScript* tersebut dapat dijadikan sebuah *module*, sehingga variabel yang telah di-*export* dapat digunakan pada *file JavaScript* lain dengan menggunakan *import*.

Penggunaan *export* diawali dengan kata kunci `export` kemudian diikuti oleh nama variabel yang ingin di-*export* atau bisa digunakan di akhir kode kita, dengan nama variabel yang ingin di *export*.

Contoh dasar melakukan `export` pada variabel :

```
export let name = "Thoriq";
```

Kita tidak bisa langsung meng-*export* data tanpa disimpan ke dalam variabel terlebih dahulu seperti ini:

```
export 'Thoriq';
```

Kita juga bisa melakukan *export* pada objek *JavaScript*:

```
export let orang = {
  nama: "Thoriq",
  umur: 25,
  alamat: "Jl. Kemang Raya",
};
```

Selain variabel dan objek kita juga bisa meng-*export* sebuah *function*:

```
export function sayHello(user) {
  console.log(`Hello, ${user}!`);
}
```

Lalu kita juga bisa meng-*export* variabel objek dan *function* sekaligus:

```
export let name = "Thoriq";

export let orang = {
  nama: "Thoriq",
  umur: 25,
  alamat: "Jl. Kemang Raya",
};

export function sayHello(user) {
  console.log(`Hello, ${user}!`);
}
```

Atau juga bisa melakukan `export` secara terpisah untuk semua kode yang ingin kita `export` seperti contoh di bawah ini:

```
let name = "Thoriq";

let orang = {
  nama: "Thoriq",
  umur: 25,
  alamat: "Jl. Kemang Raya",
};

function sayHello(user) {
  console.log(`Hello, ${user}!`);
}

// Mengexport variable name, object orang dan function sayHello sekaligus
export { name, orang, sayHello };
```

Import

Import diibaratkan sebagai pasangan dari *export*. Jadi *import* digunakan untuk menggunakan variabel yang sudah di-*export* dari *module* lainnya.

Contoh dasar melakukan `import` variabel :

```
import { data } from "./namaModul.js";
```

Contoh penggunaan dari `import` dengan menggunakan *file export* dari contoh di atas :

```
// index.js
import { name, orang, sayHello } from "./user.js";

// Menggunakan hasil import
console.log(name); // Output: Thoriq
console.log(orang); // Output: {nama: "Thoriq", umur: 25, alamat: "Jl. Kemang Raya"}
sayHello(orang.nama); // Output: Hello, Thoriq!
```

Export As dan Import As

Apakah kamu pernah merasa nama data atau *function* yang kamu buat ternyata terlalu panjang atau sulit untuk diingat?

Pada topik ini, kita dapat memberi *alias* (nama pengganti) pada nama data yang ingin kita *export* maupun *import*. Sehingga kita tidak perlu mengubah nama pada data aslinya.

Export As

Pertama, kamu dapat memberi *alias* pada nama variabel di suatu tipe data, objek, ataupun *function* yang ingin di-*export* dengan menggunakan `export as`.

```
export namaVariabelLama as namaVariabelBaru;
```

Kita bisa lihat contoh di bawah ini menggunakan `export as`:

```
const warna = "Merah";

let orang = {
  nama: "Thoriq",
  umur: 25,
  alamat: "Jl. Kemang Raya",
};


function katakanHalo(user) {
  console.log(`Hello, ${user}!`);
}


export { warna as color, orang as person, katakanHalo as sayHello };
```

Pada *file JavaScript* lain ketika kita ingin meng-*import* variabel warna, orang atau katakanHalo, caranya adalah seperti contoh di bawah ini:

```
import color from "./user.js";
import person from "./user.js";
import sayHello from "./user.js";

// kode di bawah ini akan error
import warna from "./user.js";
import orang from "./user.js";
import katakanHalo from "./user.js";
```

Bila kita perhatikan kode di atas, kita sudah tidak bisa menggunakan nama variabel yang lama ketika meng-*import*. Ini dikarenakan variabel yang lama sudah diberikan *alias* ke nama yang baru ketika di-*export*.

📝 Catatan:

Penggunaan `export as` hanya bisa dilakukan dengan *export* secara sekaligus di akhir kode.

```
// Contoh tidak bisa menggunakan export as

export const warna = "Merah";

export let orang = {
  nama: "Thoriq",
  umur: 25,
  alamat: "Jl. Kemang Raya",
};

export function katakanHalo(user) {
  console.log(`Hello, ${user}!`);
}
```

Import As

Tidak hanya dapat memberi *alias* pada data yang di-*export*, kita juga dapat memberi *alias* pada saat melakukan *import* data menggunakan `import as`.

```
import { namaVariabelLama as namaVariabelBaru } from "./namaModul.js";
```

Contoh menggunakan `import as`:

```
// file user.js tempat mengexport
export const warnaKesukaan = "Merah";

export let orangBaru = {
  nama: "Thoriq",
  umur: 25,
  alamat: "Jl. Kemang Raya",
};

export function katakanHalo(user) {
  console.log(`Hello, ${user}!`);
}
// file index.js tempat mengimport
import {
  warnaKesukaan as favoriteColor,
  orangBaru as newPerson,
  katakanHalo as sayHello,
} from "./user.js";

console.log(favoriteColor); // Output: Merah
console.log(newPerson); // Output: {nama: "Thoriq", umur: 25, alamat: "Jl. Kemang
Raya"}
sayHello(newPerson.nama); // Output: Hello, Thoriq!
```

Export Default

Pada topik sebelumnya kita sudah belajar memberi label `export` pada variabel yang ingin di-*export*.

Dengan menggunakan `export default`, kode yang kita *export* akan bersifat lebih spesial pada *module* tersebut. Namanya juga spesial, berarti dalam satu *module* hanya boleh terdapat satu `export default`.

Biasanya `export default` digunakan untuk membuat salah satu variabel menjadi data utama yang akan di-*export* pada sebuah *module*. `export default` juga bisa digunakan jika hanya ada satu variabel pada suatu *module*.

Penggunaannya sama seperti *export* biasa, kamu cukup menambahkan kata kunci `default` setelah `export`.

```
export default data;
```

Contoh menggunakan `export default`:

```
// greeting.js
function sayHello(user) {
  console.log(`Hello, ${user}!`);
}

export default sayHello;
```

Cara *import*-nya pun sedikit berbeda ketika menggunakan `export default`. Kurung kurawalnya atau { } dihilangkan dan langsung memanggil nama data yang sudah di-*export* sebelumnya.

```
import data from "./namaModul.js";
```

Contoh melakukan `import` pada `export default`:

```
import sayHello from "./greeting.js";

sayhello("Thoriq"); // Output: Hello, Thoriq!
```

selain itu, kita juga dapat menggunakan *export* dan *export default* bersamaan dalam satu *file module*:

```
export const FONTS = {
  small: 10px;
  medium: 14px;
  normal: 16px;
```

```
  big: 20px;
}

export default function sayHello(user) {
  console.log(`Hello, ${user}!`);
}
```

cara *import module* di atas pada *file* yang membutuhkan variabel di *module* tersebut dapat menjadi seperti ini:

```
import sayHello, { FONTS } from "./greeting.js"
```

Asynchronous

Bahasa pemrograman JavaScript termasuk ke dalam *single-thread language* atau *synchronous* yang artinya hanya dapat mengeksekusi satu perintah pada satu waktu dan harus menunggu satu perintah tersebut selesai sebelum melanjutkan perintah selanjutnya.

Untuk bisa mengeksekusi urutan perintah dari kode yang kita tulis ada 2 istilah yang digunakan pada JavaScript yaitu *synchronous* dan *asynchronous*.

Introduction

Apa itu synchronous?

Synchronous adalah saat kita mengeksekusi perintah satu persatu dan berurutan. Analoginya seperti kita sedang mengantri di kasir atau loket. Ketika ada 1 perintah masuk maka dia akan dieksekusi terlebih dahulu. Jika perintah belum selesai dan sudah ada perintah baru maka perintah kedua (yang baru) akan mengantri sampai perintah 1 selesai. Proses seperti ini disebut *blocking* dan membuat perintah kita tereksekusi dengan lambat.

Contoh :

```
console.log("antrian 1");
console.log("antrian 2");
console.log("antrian 3");

// output
// antrian 1
// antrian 2
// antrian 3
```

Kode di atas bersifat *synchronous* yaitu kode dijalankan baris per baris. Maka output kode di atas tereksekusi sesuai urutan perintahnya.

Salah satu konsep lain di pemrograman adalah kebalikan dari *synchronous* yaitu *asynchronous*.

Apa itu Asynchronous?

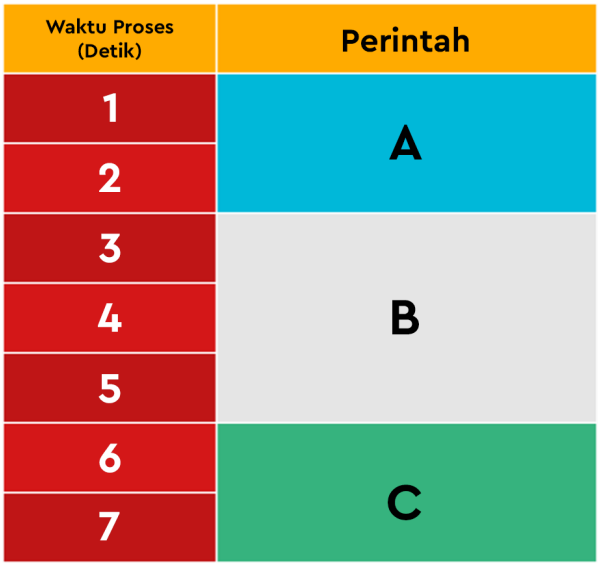
Asynchronous yang biasa dikenal juga dengan sebutan *non-blocking* mengizinkan komputer kita untuk memproses perintah lain sambil menunggu suatu proses lain yang sedang berlangsung. Ini artinya kita bisa melakukan lebih dari 1 proses sekaligus (*multi-thread*). Eksekusi perintah dengan *asynchronous* tidak akan melakukan blocking atau menunggu perintah sebelumnya selesai. Jadi sambil menunggu kita bisa mengeksekusi perintah lain.

Analoginya seperti saat kita mencuci baju di mesin cuci. Agar lebih produktif, sambil menunggu cucian selesai kita bisa melakukan pekerjaan lain misalnya menyapu dan mengepel. Artinya disini kita melakukan 3 proses sekaligus.

Untuk lebih jelasnya, mari kita lihat perbedaan jika menggunakan *asynchronous* dengan *synchronous* dengan memperhatikan waktu eksekusi perintah:

Perintah	Waktu Proses (detik)
a	2
b	3
c	2

Jika perintah di atas dieksekusi dengan perintah *synchronous*, maka eksekusi perintahnya akan terlihat seperti ini:



Dari gambar di atas, kita dapat lihat bahwa **setiap perintah dijalankan sampai selesai terlebih dahulu, baru mengeksekusi perintah selanjutnya**, sehingga **membutuhkan waktu 7 detik** untuk menyelesaikan semua perintah.
Namun, jika perintah di atas dijalankan secara *asynchronous*, maka eksekusi perintahnya akan terlihat seperti ini:

Dari gambar di atas, kita dapat lihat bahwa setiap perintah dijalankan saat perintah yang lain juga sedang dijalankan, sehingga membutuhkan waktu 4 detik untuk menyelesaikan semua perintah.

Menjalankan Asynchronous pada JavaScript

Jika JavaScript secara *default* bersifat *synchronous*, maka bagaimana jika ingin menerapkan proses *asynchronous* ? Ada banyak method *asynchronous* yang terdapat di JavaScript, salah satu contohnya adalah, `setTimeout(function, milliseconds)` digunakan untuk simulasi pemanggilan kembali proses asynchronous yang sedang/sudah selesai dijalankan.

Contoh *asynchronous* menggunakan `setTimeout()`:

```
setTimeout(() => {
  console.log("Cuci baju"); // proses asynchronous
}, 2000);
console.log("Menyapu");
console.log("Mengepel");
console.log("Memasak");

// 1000 ms = 1 second

// Output:
// Menyapu
// Mengepel
// Memasak
// Cuci baju
```

Menerapkan Asynchronous pada Aplikasi JavaScript

Dari contoh simulasi di atas model eksekusi *asynchronous* lebih effisien dibandingkan *synchronous*. Namun, permasalahan terjadi saat menggunakan *asynchronous*, ada satu perintah yang bergantung pada output eksekusi *asynchronous* sebelumnya. Dengan kata lain fungsi berjalan kejar-kejaran (*race condition*), sehingga data yang kita inginkan menjadi kosong. Sebagai contoh:

```
const user = getUser(); // fungsi async untuk mengambil data user dari API
console.log(user) // Output: null
```

Dari kode di atas, ada kemungkinan `user` masih bernilai `null`. Hal ini terjadi karena fungsi `getUser()` adalah fungsi *asynchronous* yang belum selesai dijalankan, namun perintah `console.log()` sudah menuntut untuk dijalankan.

Untuk mengatasi masalah tersebut, kita dapat menggunakan:

1. *Callback*.
2. *Promises*.
3.
4.

Lalu dalam kondisi apa saja kita perlu menggunakan *asynchronous*? Teknik *asynchronous* paling banyak digunakan dalam mengelola komunikasi ke server seperti proses *request* API (mengambil data dari server), operasi file, koneksi ke database, *real time communication* (messenger/chat), dan sebagainya.

Callback

Analogi dari konsep *callback* adalah seperti ketika kita membeli dari seorang penjual rujak untuk membeli rujak, sambil penjual membuat rujaknya, kita membeli ketoprak. Ketika penjual rujak sudah selesai, penjual rujak akan memanggil kita kembali (*callback*) untuk memberitahu kita bahwa rujaknya sudah siap dan harus kita bayar.

Apa itu *callback* dalam JavaScript?

Callback adalah sebuah *function*, namun bedanya dengan *function* pada umumnya adalah pada cara eksekusinya.

Jika *function* pada umumnya dieksekusi secara langsung, sedangkan *callback* dieksekusi di dalam *function* lain melalui parameter.

Kita akan menemukan proses *callback asynchronous* pada proses *ajax*, komunikasi *HTTP*, Operasi *file*, *timer* dan sebagainya.

Membuat *callback function*

Kita dapat memanggil *callback* pada sebuah *function* dengan cara memanggilnya ke dalam parameter dan digunakan di dalam *function*.

Pertama, kita deklarasikan dahulu *function* `greeting(name)` yang ingin kita panggil dalam callback *function* lain. *Function* `greeting(name)` berisi `console.log()` yang menerima sebuah parameter `name`.

```
function greeting(name) {
  console.log(`Halo ${name}`);
}
```

Kedua, buat sebuah *function* `introduction(firstName, lastName, callback)` dengan menerima parameter `firstName`, `lastName` dan `callback` lalu di dalam *function* tersebut kita menggabungkan parameter `firstName` dan `lastName` ke dalam variabel `fullName` untuk mengirimkannya ke dalam callback.

```
function introduction(firstName, lastName, callback) {
  const fullName = `${firstName} ${lastName}`;

  callback(fullName);
}

introduction("John", "Doe", greeting); // Halo John Doe
```

Ketiga, dalam pemanggilan *function* `introduction`, kita mengisi argumen dari parameter yang dibutuhkan yaitu `Miftah`, `Faris`, dan *function* `greeting` yang sudah kita buat sebelumnya lalu kita panggil `callback(fullName)` di dalam *function* `introduction` sehingga kita bisa mendapatkan hasil dari *function* `greeting`.

Synchronous dan Asynchronous pada JavaScript

Kita akan sedikit membahas kembali topik sebelumnya tentang *synchronous* dan *asynchronous*. Pada *synchronous* output di proses berdasarkan urutan kode.

Contoh proses *synchronous*:

```
function proses1() {
  console.log("proses 1 selesai dijalankan");
}

function proses2() {
  console.log("proses 2 selesai dijalankan");
}

function proses3() {
  console.log("proses 3 selesai dijalankan");
}

proses1();
proses2();
proses3();

/*
Hasil Output
proses1 selesai dijalankan
proses2 selesai dijalankan
proses3 selesai dijalankan
```

```
*/
```

Pada kode di atas, kita bisa melihat bahwa `proses1()`, `proses2()`, dan `proses3()` berjalan berurutan seperti yang seharusnya.

Sedangkan pada *asynchronous* yang biasa dikenal juga dengan sebutan *non-blocking* mengizinkan komputer kita untuk memproses perintah lain sambil menunggu suatu proses lain yang sedang berlangsung. Ini artinya kita bisa melakukan lebih dari 1 proses sekaligus (*multi-thread*).

Contoh proses *asynchronous*:

```
function proses1() {
  console.log("proses 1 selesai dijalankan");
}

function proses2() {
  // setTimeout or delay for *asynchronous* simulation
  setTimeout(function () {
    console.log("proses 2 selesai dijalankan");
  }, 100);
}

function proses3() {
  console.log("proses 3 selesai dijalankan");
}

proses1();
proses2();
proses3();

/*
Hasil Output
proses1 selesai dijalankan
proses3 selesai dijalankan
proses2 selesai dijalankan
*/
```

Bisa kita bisa lihat bahwa `proses3()` selesai terlebih dahulu dibanding `proses2()`. Hal ini terjadi dikarenakan `proses2()` melakukan `setTimeout()` yang merupakan proses *asynchronous* sehingga `proses3()` selesai terlebih dibanding `proses2()`.

Menggunakan *Callback*

Kita akan coba memperbaiki *asynchronous* di atas dengan memastikan output proses1, proses2, dan proses3 sesuai urutan dengan menggunakan *callback*.

```
function proses1() {
  console.log("proses 1 selesai dijalankan");
}

function proses2(callback) {
  setTimeout(function () {
    console.log("proses 2 selesai dijalankan");
    callback();
  }, 2000);
}

function proses3() {
  console.log("proses 3 selesai dijalankan");
}
proses1();
proses2(proses3);

/*
Hasil Output
proses1 selesai dijalankan
proses2 selesai dijalankan
proses3 selesai dijalankan
*/
```

Analogi kasus di atas adalah bayangkan kamu memiliki *method* yang melakukan proses menampilkan *image* lalu kita memerlukan *callback* untuk memastikan proses menampilkan *image* terpanggil terlebih dahulu sebelum menampilkan ke user yang ingin mengaksesnya. Jadi *callback* dapat digunakan untuk mengatur *order function* yang harus berjalan terlebih dahulu.

Promise

Sebelumnya kita sudah belajar cara menggunakan *asynchronous* pada JavaScript dengan *callback* lalu sekarang kita akan mencoba menggunakan *promise*. *Promise* sendiri adalah salah satu fitur dari ES6 (ES2015) JavaScript. Konsep *promise* hadir untuk memecahkan masalah yang bertele-tele dengan *callback*, semakin banyak kita menggunakan *callback* untuk proses *asynchronous* semakin kompleks dan sulit kode kita untuk dibaca dan dipelihara. Kita juga akan sering menghadapi *callback* di dalam *callback* dan seterusnya. Masalah seperti ini disebut dengan *Callback Hell*.

Contoh dari *callback hell* :

```
const verifyUser = (username, password, callback) => {
  dataBase.verifyUser(username, password, (error, userInfo) => {
    if (error) {
      callback(error);
    } else {
```

```
      dataBase.getRoles(username, (error, roles) => {
        if (error) {
          callback(error);
        } else {
          dataBase.logAccess(username, (error) => {
            if (error) {
              callback(error);
            } else {
              callback(null, userInfo, roles);
            }
          });
        }
      });
    }
  });
};
```

Contoh lain dari *callback hell*:

```
doFirst(data, function() {
  doSecond(data, function() {
   dothird(data, function() {
     // Callback Hell
    })
  })
 })
```

Konsep *Promise*

Promise sesuai dengan artinya adalah janji. Seperti ketika kita berjanji, jika apa yang kita janjikan bisa kita lakukan maka kita harus melakukannya, jika janjinya ada halangan maka kita tidak bisa melakukannya atau jika janji tersebut belum pada waktunya kita juga harus menunggunya.

Contohnya seperti :

Kita berjanji untuk menonton di bioskop dengan teman pada malam minggu besok
Dari janji tersebut kita mendapatkan beberapa poin

- Batas waktu dilaksanakan : Malam minggu besok
- Rencana : Menonton di bioskop

Lalu bagaimana jika tiba-tiba turun hujan, apakah kita masih akan berangkat untuk menonton di bioskop? Atau jika ada sesuatu terjadi sebelum janji kita berhasil dipenuhi?

Akhirnya kita mengganti dengan rencana lain jika kita tidak jadi menonton di bioskop dengan *streaming* film. Lalu, kita memiliki rencana lagi yang pasti akan kita lakukan baik kita jadi menonton di bioskop atau pun *streaming* film yaitu tidur.

Maka hasilnya rencana kita:

- Batas waktu dilaksanakan : Malam minggu besok.
- Rencana : Menonton di bioskop.
- Rencana jika gagal : *streaming* film.
- Rencana selanjutnya apapun yang terjadi : tidur.

Jika disesuaikan dengan analogi *promise* pada JavaScript menjadi

- *Pending* / tertunda = Jika kita belum melewati batas waktu dilaksanakan dan belum mengetahui janji tersebut bisa ditepati atau tidak.
- *Fulfilled* / terpenuhi = Jika janji berhasil dipenuhi sebelum batas waktu yang ditentukan.
- *Rejected* / gagal = Jika janji gagal ditepati karena suatu hal dan kita melakukan rencana lain.
- *Settled* / terselesaikan = Jika semua janji sudah selesai terpenuhi kita sudah bebas melakukan hal lainnya.

3 Status *Promise* di JavaScript

Analogi dari sebuah *promise* di JavaScript itu sama seperti kita saat mengambil suatu data baik itu dari *database* maupun *Request API*. Akan ada 3 kondisi yaitu data sedang diproses, data berhasil didapatkan, atau data gagal didapatkan.

Pada *promise* analogi di atas bisa diartikan seperti:

1. *pending*, jika data sedang diproses.
2. *fulfilled*, jika data telah berhasil didapatkan.
3. *rejected*, jika data gagal didapatkan.

State	pending	fulfilled	rejected
Result	undefined	value	error

Contoh menggunakan *promise*

```
let newPromise = new Promise((resolve, reject) => {
  if (true) {
    // apa yang dilakukan jika promise fulfilled
    resolve("Berhasil");
```

```
  } else {
    // apa yang dilakukan jika promise rejected
    reject("Gagal");
  }
});
```

Kita bisa membuat sendiri apa yang akan dilakukan pada sebuah *promise*. Di dalam *promise* ada 2 *keyword* yaitu `resolve()` dan `reject()`.

- `resolve()`, jika proses berhasil atau *fullfilled*.
- `reject()`, jika proses gagal atau *rejected*.

Contoh penggunaan *promise fullfilled*

Untuk *fulfilled* hanya bisa tereksekusi jika kita kondisi berhasil pada saat kita melakukan *async*. Kita *set condition* menjadi `true` untuk simulasi *fulfilled*.

```
const condition = true;

let newPromise = new Promise((resolve, reject) => {
  if (condition) {
    // apa yang dilakukan jika promise 'fulfilled'
    resolve("Berhasil");
  } else {
    // apa yang dilakukan jika promise 'rejected'
    reject("Gagal");
  }
});
```

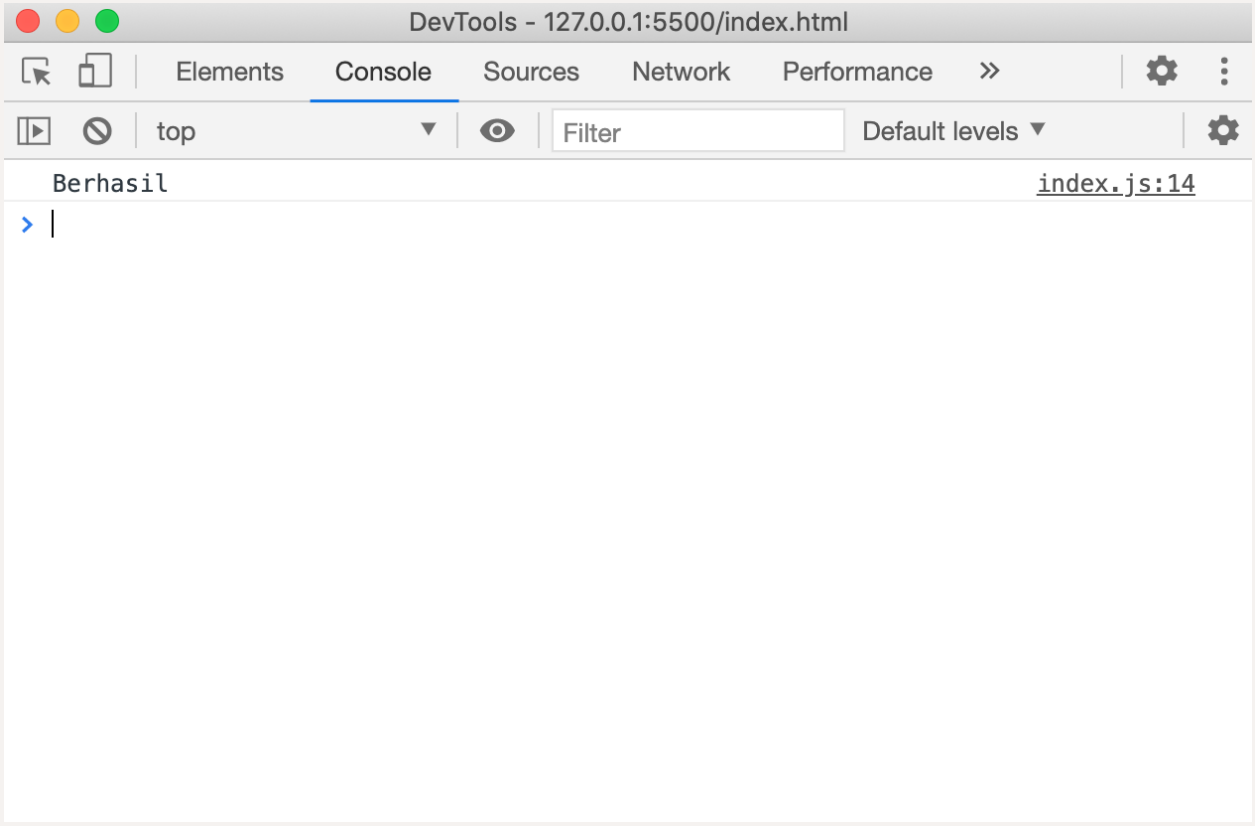
Untuk bisa mengeksekusi *promise* yang sudah dibuat kita bisa memanggil *promise* tersebut menggunakan `.then()`:

```
const condition = true;

let newPromise = new Promise((resolve, reject) => {
  if (condition) {
    // apa yang dilakukan jika promise 'fulfilled'
    resolve("Berhasil");
  } else {
    // apa yang dilakukan jika promise 'rejected'
    reject("Gagal");
  }
});

newPromise.then((result) => {
  console.log(result); // Output: "Berhasil"
```

```
});
```

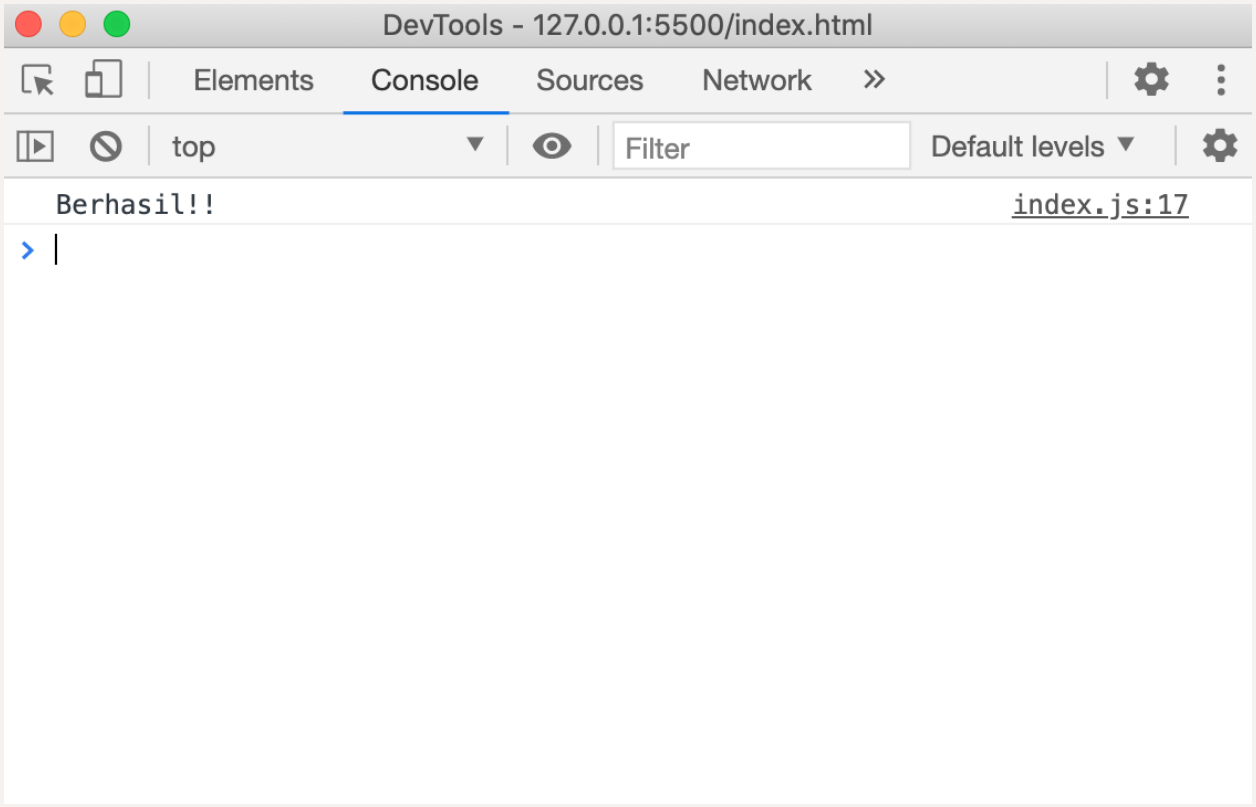


Selain itu kita juga bisa memanggil `.then()` lebih dari satu kali ketika dibutuhkan seperti contoh di bawah:

```
const condition = true;

let newPromise = new Promise((resolve, reject) => {
  if (condition) {
    // apa yang dilakukan jika promise 'fulfilled'
    resolve("Berhasil");
  } else {
    // apa yang dilakukan jika promise 'rejected'
    reject("Gagal");
  }
});

newPromise.then((result) => {
  return result;
})
.then((result2) => {
  console.log(result2 + "!!"); // Output: Berhasil!!
});
```



Contoh penggunaan *promise rejected*

Untuk *rejected* hanya bisa tereksekusi jika kita mengalami *error* pada saat kita melakukan proses *asynchronous*. Kita *set condition* menjadi `false` untuk simulasi *rejected*.:

```
const condition = false;

let newPromise = new Promise((resolve, reject) => {
  if (condition) {
    // apa yang dilakukan jika promise 'fulfilled'
    resolve("Berhasil");
  } else {
    // apa yang dilakukan jika promise 'rejected'
    reject(new Error("Error Gagal"));
  }
});
```

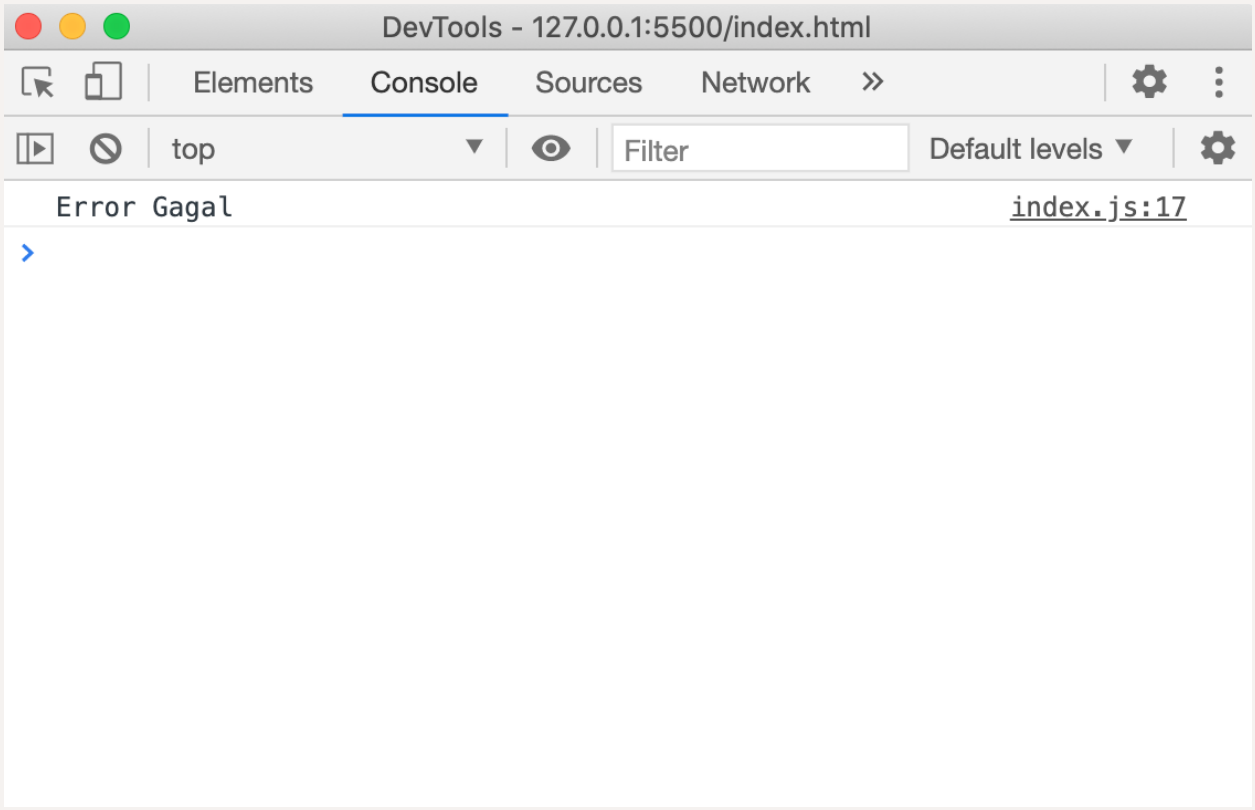
Untuk bisa mengantisipasi jika terjadi *error* kita bisa menambahkan `.catch()` pada *promise*. Sehingga, kita bisa memberi tahu pengguna jika terjadi suatu *error*:

```
const condition = false;

let newPromise = new Promise((resolve, reject) => {
```

```
  if (condition) {
    // apa yang dilakukan jika promise 'fulfilled'
    resolve("Berhasil");
  } else {
    // apa yang dilakukan jika promise 'rejected'
    reject(new Error("Error Gagal"));
  }
});

newPromise.then((result) => {
  console.log(result);
})
.catch((error) => {
  console.log(error.message); // Output: "Error Gagal"
});
```



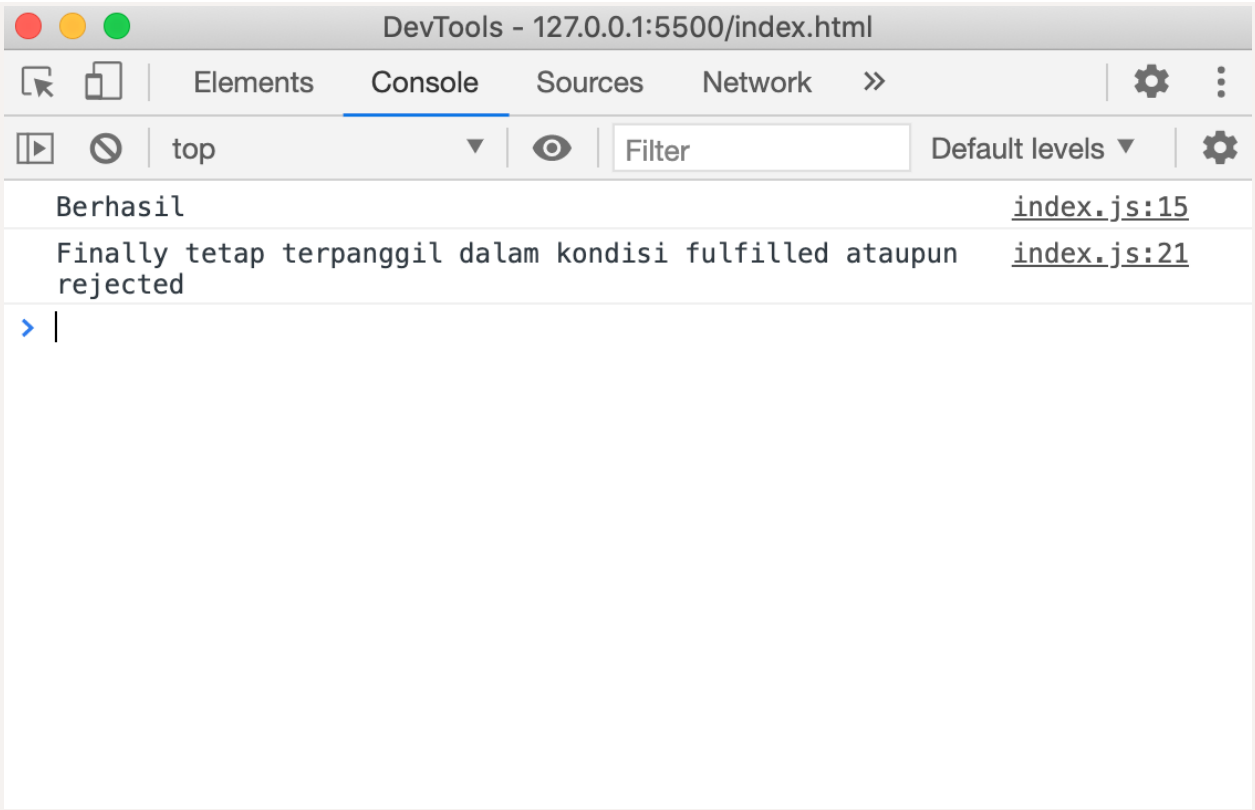
Selain `.then()` dan `.catch()` dalam *promise* kita juga memiliki `.finally()` dalam *promise* JavaScript.

`.finally()` adalah fungsi *callback* yang pasti tereksekusi dalam kondisi apapun (*fullfield* ataupun *rejected*).

```
const condition = true;
```

```
let newPromise = new Promise((resolve, reject) => {
  if (condition) {
    // apa yang dilakukan jika promise 'fulfilled'
    resolve("Berhasil");
  } else {
    // apa yang dilakukan jika promise 'rejected'
    reject(new Error("Error Gagal"));
  }
});

newPromise
  .then((result) => {
    console.log(result); // Output: Berhasil
  })
  .catch((error) => {
    console.log(error);
  })
  .finally(() => {
    console.log(
      "Finally tetap terpanggil dalam kondisi fulfilled ataupun rejected"
    ); // Output: Finally tetap terpanggil dalam kondisi fulfilled ataupun rejected
  });
```



Async/Await

Selain menggunakan *callback* dan *promise*, kita juga bisa menggunakan *async/await* untuk menggunakan *asynchronous* pada JavaScript. *Async/await* baru ada ketika *update* ES8 JavaScript dan dibangun menggunakan *promise*. Jadi sebenarnya *async/await* dan *promise* itu sama saja, namun hanya berbeda dari *syntax* dan cara penggunaannya.

Ada 2 kata kunci yang memiliki pengertian sebagai berikut:

- `async`, mengubah *function synchronous* menjadi *asynchronous*.
- `await`, menunda eksekusi hingga proses *asynchronous* selesai.

Sebuah *async function* bisa tidak berisi *await* sama sekali atau lebih dari satu *await*. *Keyword await* hanya bisa digunakan didalam *async function*, jika digunakan di luar *async function* maka akan terjadi *error*.

Async

Berikut ini contoh penggunaan dari `async` :

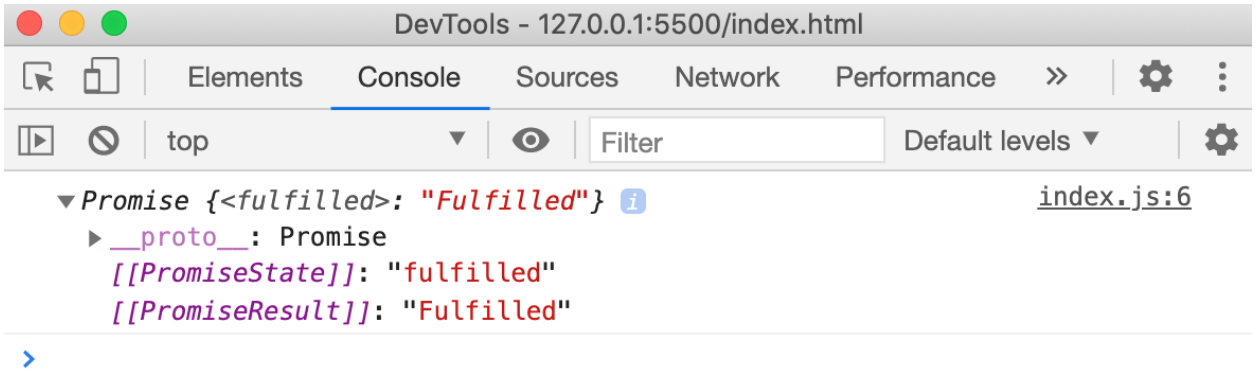
```
// async menggunakan keyword function
async function tesAsyncAwait() {
  return "Fulfilled";
}

console.log(tesAsyncAwait());

// async menggunakan arrow function
const tesAsyncAwait = async () => {
  return "Fulfilled";
};

console.log(tesAsyncAwait());
```

Jika salah satu kode di atas dijalankan, maka akan terlihat tampilan seperti berikut ini:



Await

`await` hanya bisa digunakan dalam *async function* dan `await` adalah *keyword* dalam `async` yang digunakan untuk menunda hingga proses *asynchronous* selesai.

Berikut ini contoh penggunaan dari *async/await* :

```
async function tesAsyncAwait() {
   await 'Fulfilled';
}
```

Kita juga bisa memberikan *error handling* pada *async/await*. Contoh lengkap penggunaan *async/await*:

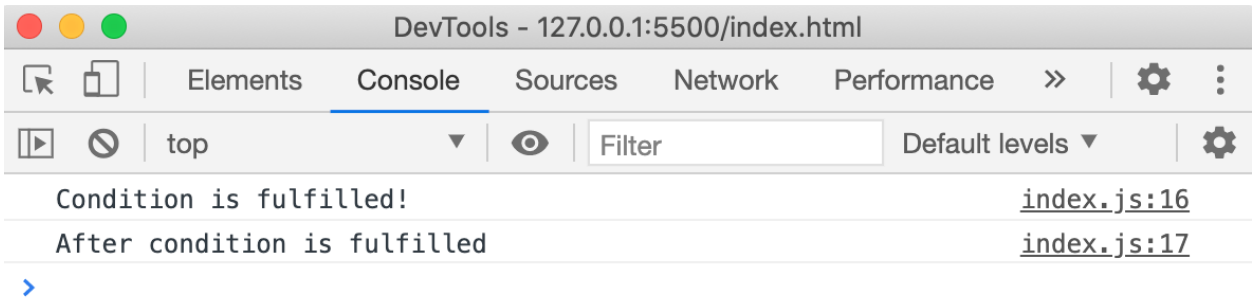
```
// Definisikan dahulu promise yang ingin digunakan
let condition = true;
let tesAsyncAwait = async (condition) => {
  if (condition) {
    return "Condition is fulfilled!";
  } else {
    throw "Condition is rejected!";
  }
```

```
};


// Membuat fungsi run menjadi asynchronous menggunakan async/await
const run = async (condition) => {
  try {
    const message = await tesAsyncAwait(condition);
    console.log(message);  // Output: Condition is fulfilled!
    console.log("After condition is fulfilled"); // Output: After condition is
fulfilled
  } catch (error) {
    console.log(error);
  }
};

run(true);
```

Dari kode di atas, kita dapat melihat bahwa `run` adalah sebuah fungsi *async* dan *await* dipanggil bersamaan dengan fungsi `tesAsyncAwait(condition)`. `await` pada fungsi ini artinya, `console.log` pada `message` dan `After condition is fulfilled` tidak akan dijalankan (ditunda) hingga proses `tesAsyncAwait(condition)` selesai dijalankan.

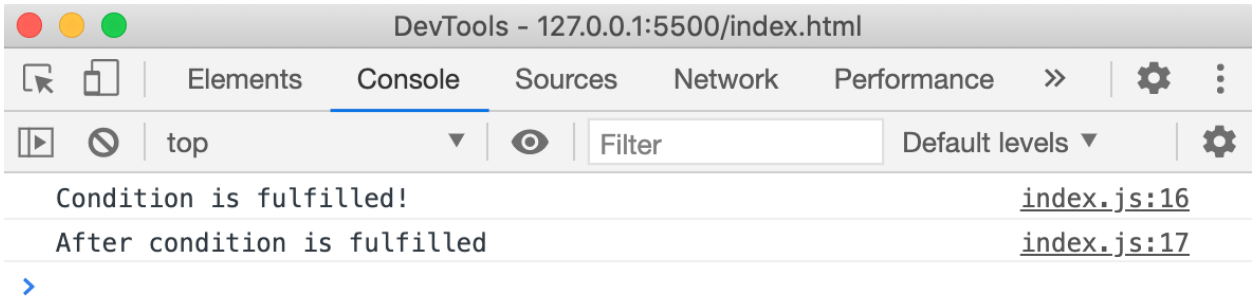


Berikut ini contoh perbandingan jika program sebelumnya dibuat menggunakan promise :

```
let condition = true;
let tesPromise = new Promise((resolve, reject) => {
  if (condition) {
    resolve("Condition is fulfilled!");
  } else {
    reject(new Error("Condition is rejected!"));
  }
});

tesPromise
  .then(result =>{
    console.log(result); // Condition is fulfilled!
    console.log("After condition is fulfilled"); // After condition is fulfilled
  })
  .catch(error =>{
    console.log(error);
  })
```

Berikut hasil `console.log()` jika kode di atas dijalankan:



Fetch

Dalam JavaScript kita bisa mengirimkan *network request* dan juga bisa mengambil informasi data terbaru dari *server* jika dibutuhkan.

Contoh *network request* yang biasa kita lakukan:

- Mengirimkan data dari sebuah *form*.
- Mengambil data untuk ditampilkan dalam *list/table*.
- Mendapatkan notifikasi.

Dalam melakukan *network request*, JavaScript memiliki metode bernama`fetch()`.

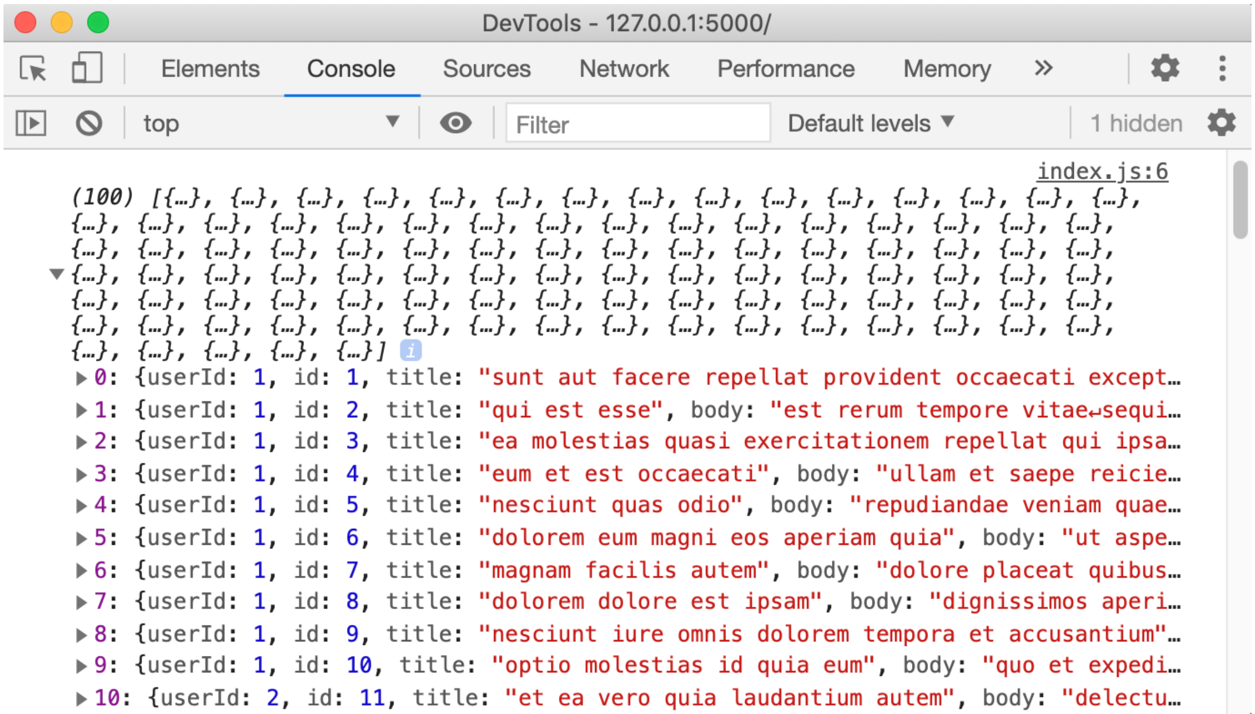
Proses melakukan `fetch()` adalah salah satu proses *asynchronous* di JavaScript sehingga kita perlu menggunakan salah satu diantara *promise* atau *async/await*.

Fetch dengan Promise

Berikut ini contoh *request* data dengan `fetch()` menggunakan *promise*:

```
fetch("https://jsonplaceholder.typicode.com/posts")
  .then(function (response) {
    return response.json();
  })
  .then(function (post) {
    console.log(post);
  });
```

Kode di atas mengambil data *end-point* dari API JSONPlaceholder. Berikut ini tampilan dari `console.log()` untuk data yang kita panggil:



Gambar di atas adalah data hasil *request* dari `fetch()` menggunakan *promise* yang kita lakukan.

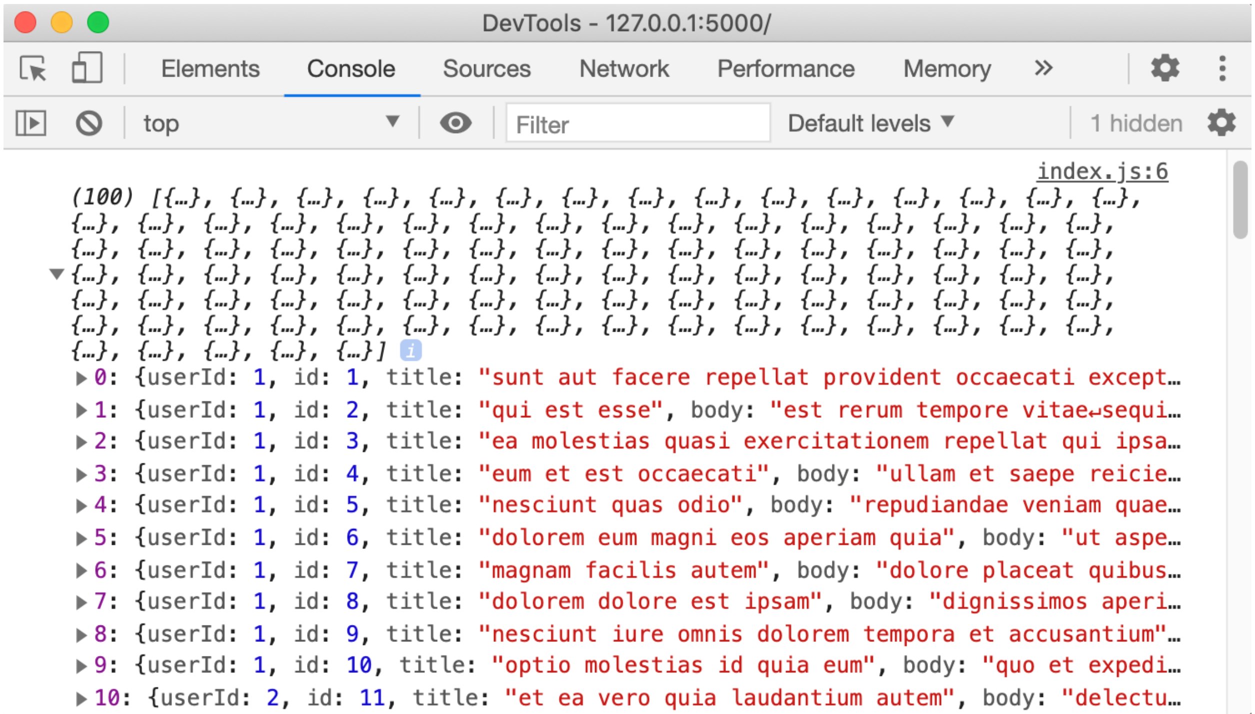
Fetch dengan async/await

Berikut contoh *request* data dengan `fetch()` menggunakan `async/await`:

```
const tesFetchAsync = async () => {
  let response = await fetch("https://jsonplaceholder.typicode.com/posts");
  response = await response.json();
  console.log(response);
};

tesFetchAsync();
```

Kita masih mengambil data dari sumber *end-point* yang sama dengan `fetch()` sebelumnya yang menggunakan *promise* sehingga hasilnya pun masih sama persis seperti sebelumnya.



Dalam penggunaan di dunia kerja dan aplikasi berskala besar kita bisa memilih menggunakan *promise* ataupun *async/await* tetapi kita lihat jika menggunakan *async/await*, kode kita terlihat lebih *clean* dan mudah dibaca.