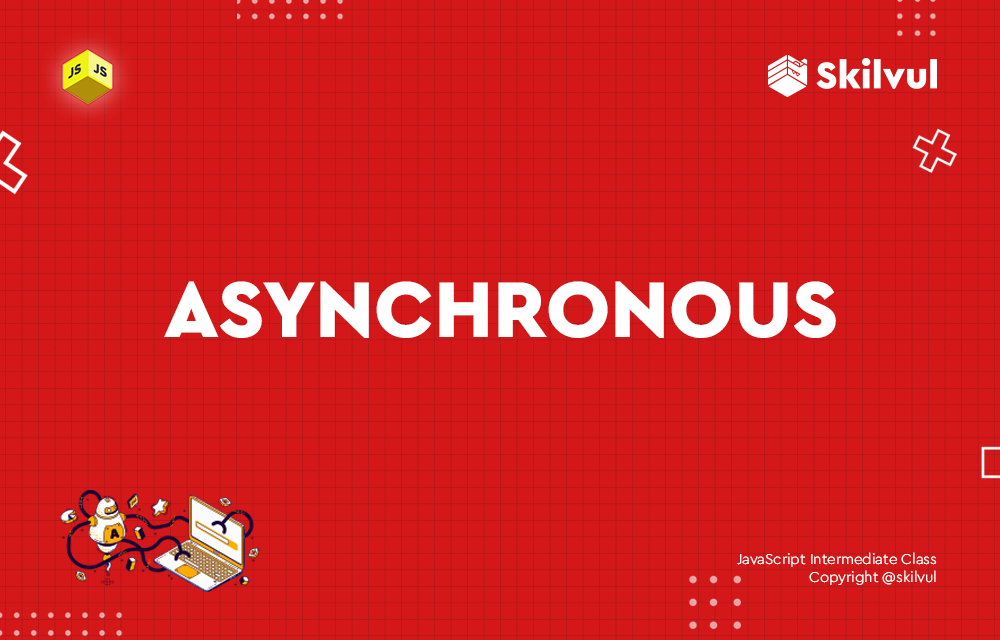
**Introduction**



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*.

**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?**

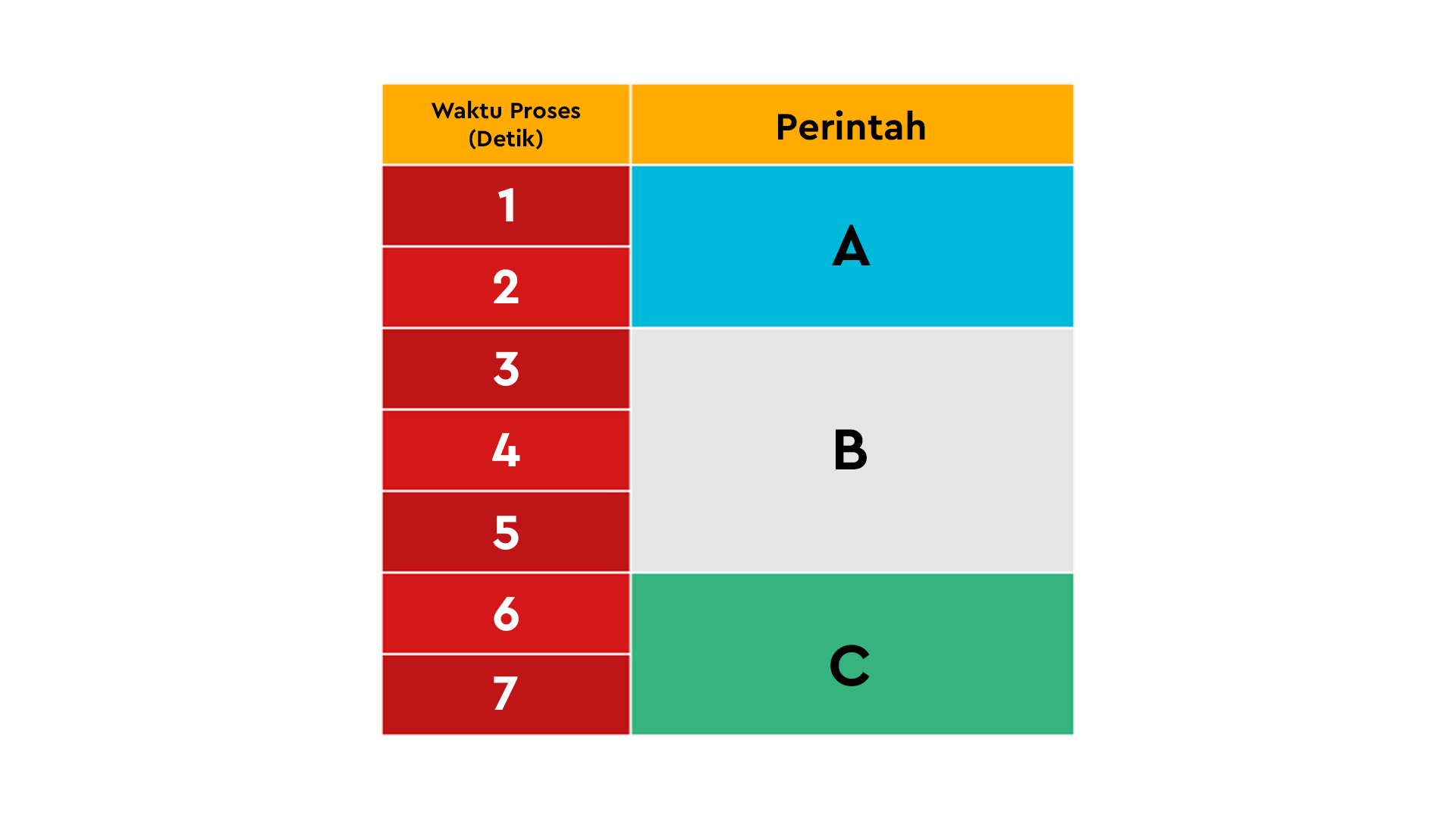
*Asynchronous* yang biasa dikenal juga dengan sebutan ***non-blocking*** mengizinkan komputer kita untuk memproses perintah lain sambil menunggu suatu proses lain yang sedang berlangsung. Ini artinya kita bisa melakukan lebih dari 1 proses sekaligus (*multi-thread*). Eksekusi perintah dengan *asynchronous*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

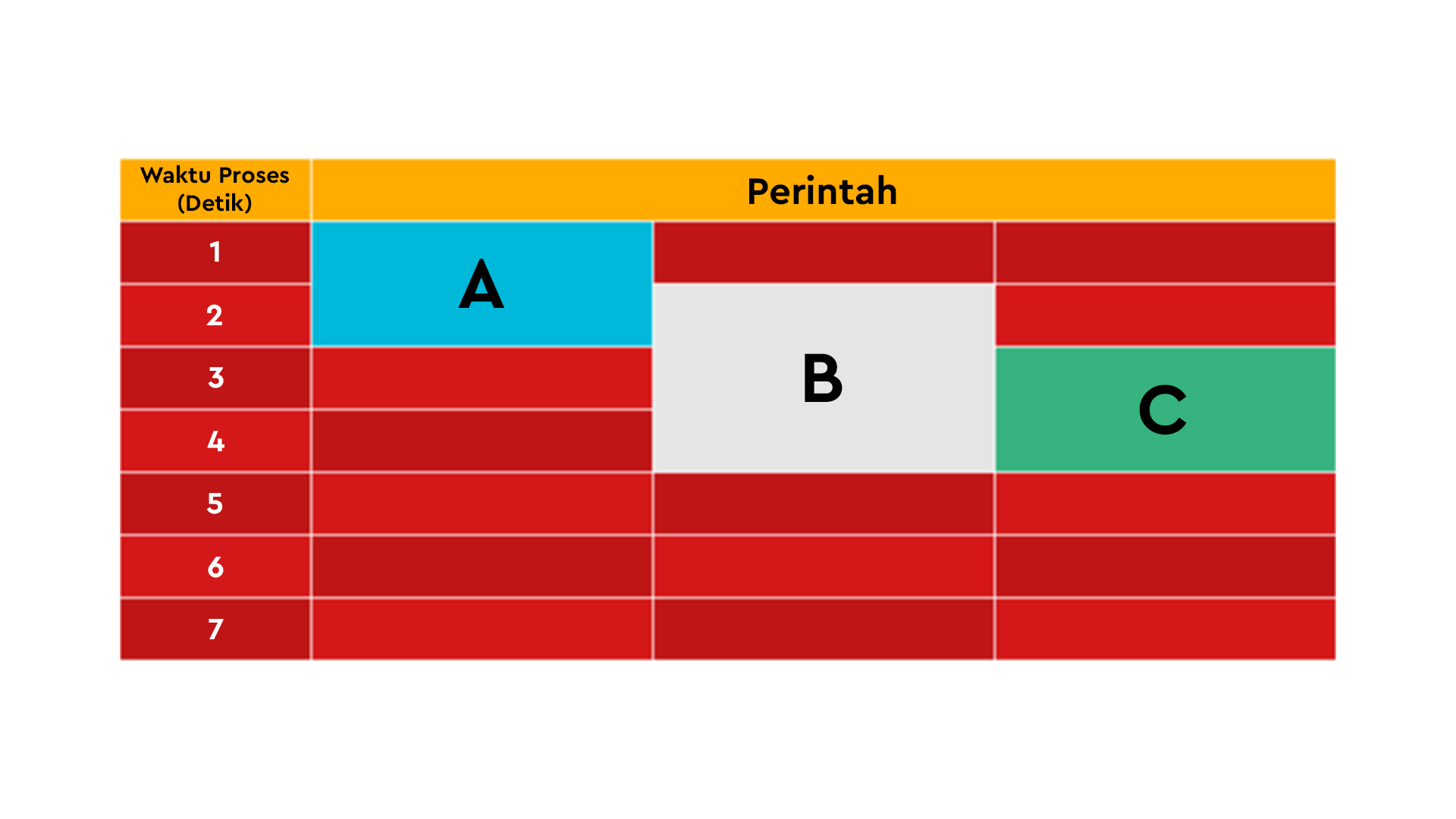
| **Perintah** | **Waktu Proses (detik)** |
| --- | --- |
| a | 2 |
| b | 3 |
| c | 2 |

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 beberapa cara untuk membuat proses *asynchronous*. Kami membatasi hanya memberikan 2 cara ini:

1. setTimeout(function, milliseconds) digunakan untuk simulasi pemanggilan kembali proses asynchronous yang sedang/sudah selesai dijalankan. Pemanggilan hanya dilakukan 1 kali.
2. setInterval(function, milliseconds) digunakan untuk simulasi pemanggilan proses asynchronous yang sedang/sudah dijalankan dalam interval waktu tertentu. Pemanggilan dilakukan berkali-kali sesuai interval waktu yang ditentukan.

Contoh *asynchronous* menggunakan setTimeout():

setTimeout(() => {

console.log("Cuci baju"); // proses asynchronous

}, 1000);

console.log("Menyapu");

console.log("Mengepel");

console.log("Memasak");

// 1000 ms = 1 second

// Output:

// Menyapu

// Mengepel

// Memasak

// Cuci baju

Contoh *asynchronous* menggunakan setInterval():

setInterval(() => {

console.log("Cuci baju"); // proses asynchronous

}, 3000);

console.log("Menyapu");

console.log("Mengepel");

console.log("Memasak");

// 3000 ms = 3 second

// Output:

// Menyapu

// Mengepel

// Memasak

// Cuci baju (x time)

// Cuci baju akan dijalankan setiap 3 detik sekali

Kita bisa lihat bahwa hasilnya urutan pertama adalah Menyapu, Mengepel, Memasak, dan Cuci Baju. Ini terjadi karena cara kerja *asynchronous* tidak akan menunggu suatu perintah sampai selesai, namun langsung mengeksekusi perintah lainnya.

**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. ***Async / Await***.

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.

## 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();

}, 100);

}

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.

## 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}, selamat datang di Skilvul!`);

}

**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("Miftah", "Faris", greeting); // Halo Miftah Faris, selamat datang di Skilvul !

**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

**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(new Error("Error 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(new Error("Error Gagal"));

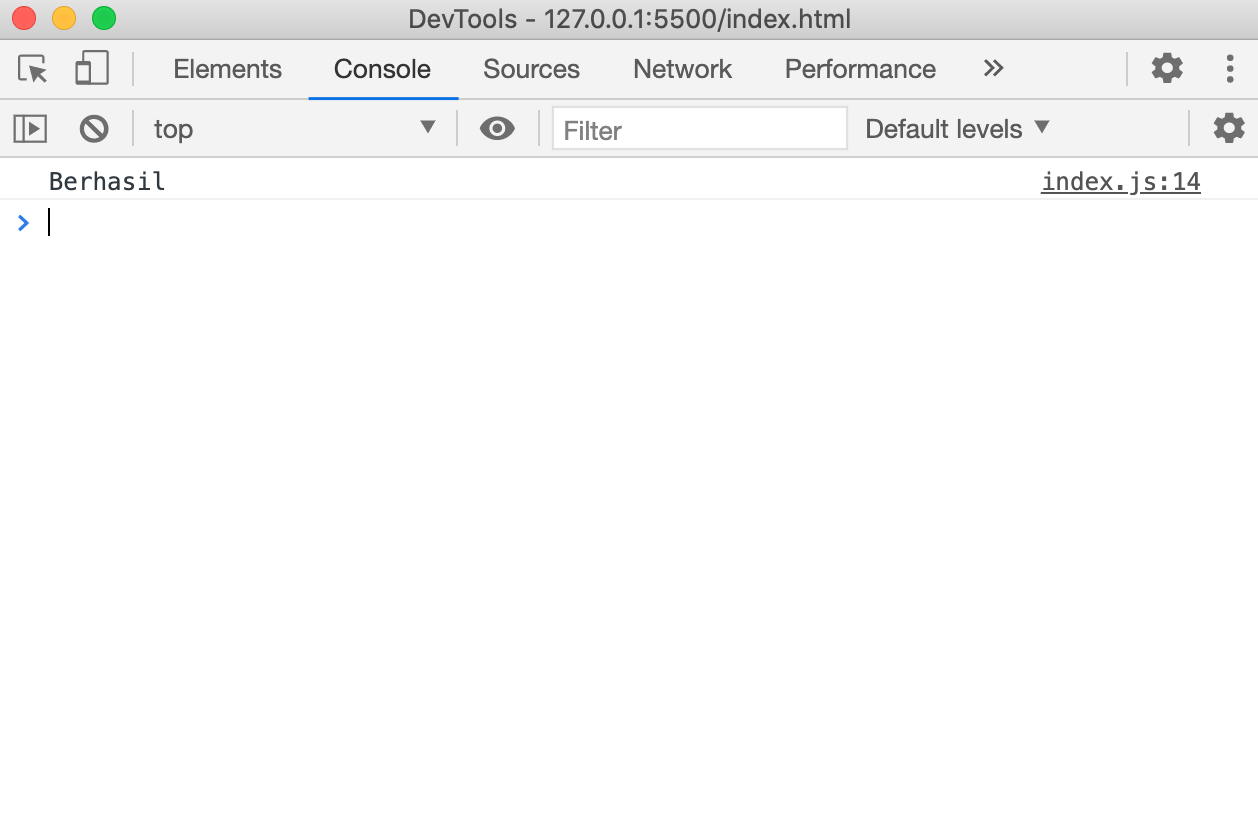
}

});

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(new Error("Error Gagal"));

}

});

newPromise.then((result) => {

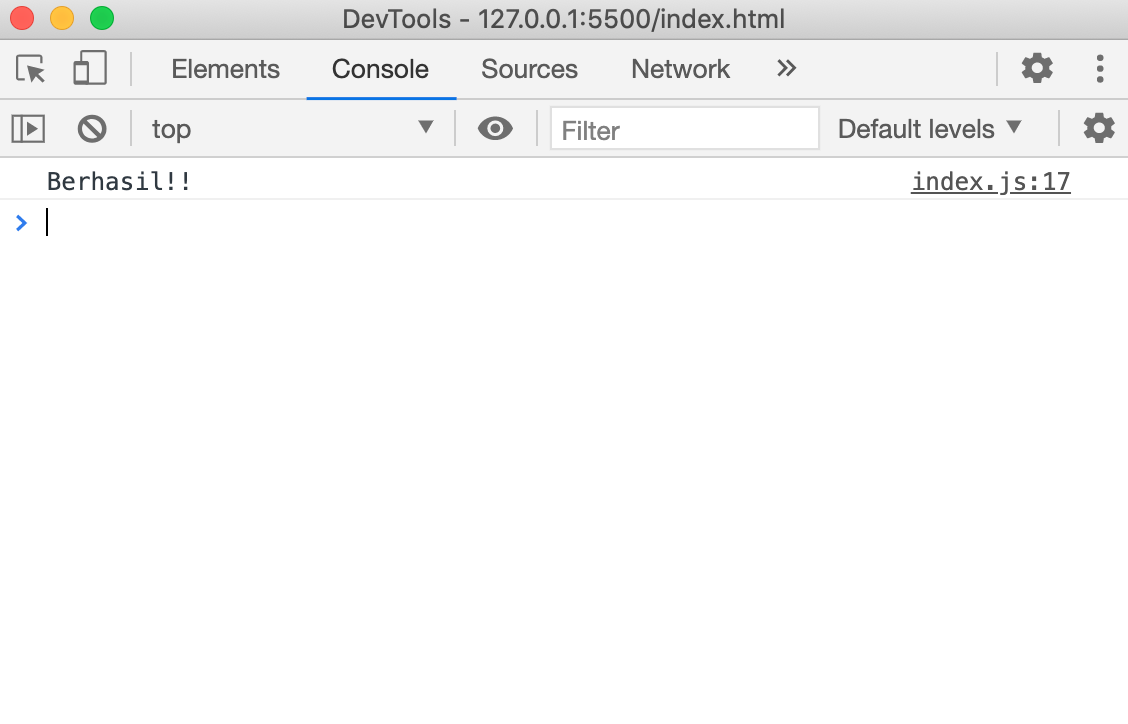
return result;

})

.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) => {

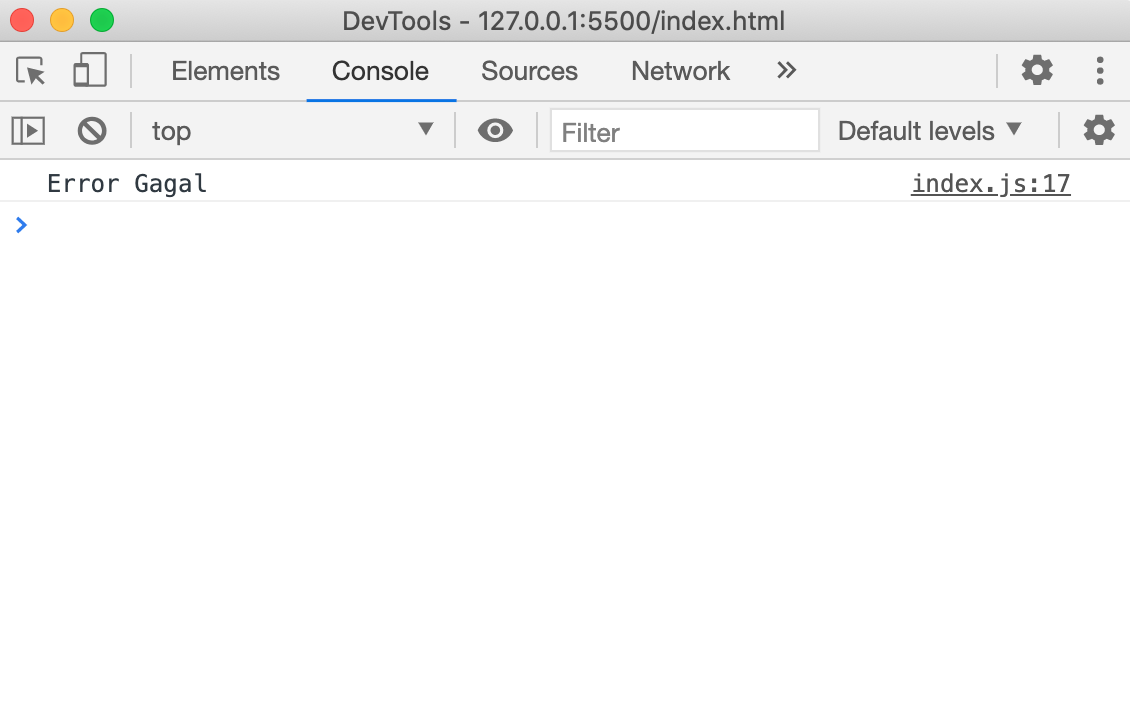
console.log(result);

})

.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

});

